Федерико Бьянкуцци, Шейн Уорден

Пионеры программирования

Диалоги с создателями наиболее популярных языков программирования

O'REILLY®

СИМВОЛ®

Masterminds of Programming

Conversations with the Creators of Major Programming Languages

Federico Biancuzzi,
Shane Warden

O'REILLY®

ПРОФЕССИОНАЛЬНО

Пионеры программирования

Диалоги с создателями наиболее популярных языков программирования

Федерико Бьянкуцци, Шейн Уорден

СИМВОЛ®

Санкт-Петербург — Москва
2011

Серия «Профессионально»

Федерико Бьянкуцци, Шейн Уорден

Пионеры программирования

Диалоги с создателями наиболее популярных языков программирования

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Т. Темкина*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Бьянкуцци Ф., Уорден Ш.

Пионеры программирования. Диалоги с создателями наиболее популярных языков программирования. – Пер. с англ. – СПб: Символ-Плюс, 2011. – 608 с., ил.

ISBN 978-5-93286-170-7

В книге собраны 27 интервью с людьми, стоявшими у истоков создания различных языков программирования, с гуру, чьи имена на слуху в мире разработки ПО. Их размышления позволят читателю подняться на новый уровень осмысления проблем развития компьютерной отрасли, увидеть скрытые процессы, которые привели к тем или иным конструктивным решениям, узнать, какие цели ставили перед собой разработчики, на какие компромиссы им приходилось идти и какое влияние оказала их работа на современное программирование.

Судьбы языков складывались по-разному – одни, сыграв свою роль, уступили место новациям, другие смогли чудесно возродиться с появлением новых технологий, но все они оставили значительный след в истории информатики.

ISBN 978-5-93286-170-7
ISBN 978-0-596-51517-1 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 17.09.2010. Формат 70×90 $^{1}/_{16}$. Печать офсетная.
Объем 38 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 9

Введение 11

Глава 1. C++ 15

Проектные решения 16
Применение языка 21
ООП и параллелизм 26
Будущее 31
Преподавание 36

Глава 2. Python 39

Как это делается в Python 40
Хороший программист 49
Реализации Python 56
Приемы и мастерство 61

Глава 3. APL 69

Бумага и карандаш 70
Основные принципы 73
Параллелизм 80
Наследие 84

Глава 4. Форт 89

Язык Форт и языковое проектирование 90
Аппаратное обеспечение 99
Разработка приложений 103

Глава 5. Бейсик 113

Цели создания Бейсика 114
Разработка компилятора 122
Язык и практика программирования 127
Разработка языка 129
Рабочие задачи 135

Глава 6. AWK 139
Жизнь алгоритмов 140
Разработка языка 142
Культура UNIX 145
Роль документации 151
Компьютерные науки 155
Разведение малых языков 158
Разработка нового языка 164
Культура наследования 174
Трансформирующие технологии 177
Мелочи, изменившие мир 183
Теория и практика 190
В ожидании прорыва 199
Программирование по примерам 205

Глава 7. Lua 211
Мощь скриптов 212
Опыт 216
Разработка языка 221

Глава 8. Haskell 231
Функциональная команда 232
Эволюция функционального программирования 235
Язык Haskell 243
Расширение (функционального) образования 252
Формализм и развитие 254

Глава 9. ML 263
Доказательство теорем 264
Теория смысла 274
За рамками информатики 282

Глава 10. SQL 291
Основополагающая статья 292
Язык 296
Обратная связь и развитие 301
XQuery и XML 308

Глава 11. Objective-C 313
Разработка Objective-C 314
Развитие языка 316

Образование и обучение ... 322
Управление проектами и устаревшее ПО ... 325
Objective-C и другие языки ... 334
Компоненты, песок и кирпичи ... 339
Качество как экономическое явление ... 348
Образование ... 351

Глава 12. Java ... 355
Сила простоты ... 356
Дело вкуса ... 360
Параллелизм ... 364
Разработка языка ... 366
Обратная связь ... 371

Глава 13. C# ... 375
Язык и конструкция ... 376
Развитие языка ... 383
C# ... 388
Будущее компьютерных наук ... 394

Глава 14. UML ... 401
Изучение и преподавание ... 402
Роль человеческого фактора ... 409
UML ... 414
Знания ... 418
Быть готовым к переменам ... 421
Применение UML ... 427
Уровни и языки ... 433
Немного о многократном использовании ... 439
Симметричные взаимоотношения ... 444
UML ... 449
Разработка языка ... 452
Обучение разработчиков ... 459
Творчество, изящество и шаблоны ... 462

Глава 15. Perl ... 473
Язык революции ... 474
Язык ... 479
Сообщество ... 486
Эволюция и революция ... 490

Глава 16. PostScript 497
Проектирование на века 498
Исследования и образование 509
Интерфейсы для долголетия 515
Обычные пожелания 519

Глава 17. Eiffel 523
День вдохновения 524
Многократное использование и универсальность 533
Проверка языков 538
Управление ростом и развитием 546

Послесловие 553

Об авторах 554

Участники интервью 555

Алфавитный указатель 574

Предисловие

Создание языков программирования – увлекательнейшая тема. Многие разработчики тешут себя мыслью, что способны придумать язык получше того, которым им в данный момент приходится пользоваться, многие даже полагают, что могут спроектировать язык, превосходящий все известные. Возможно, им это и удается, но результаты, по-видимому, не выходят за пределы письменного стола, так что эта книга не о них.

Создать язык программирования непросто. Мелкие недочеты в конструкции языка чреваты крупными ошибками в созданных на нем проектах, а даже небольшая ошибка может привести к серьезным последствиям и огромным убыткам. Уязвимости в популярных программах неоднократно позволяли злоумышленникам успешно атаковать их, что оборачивалось многомиллиардным ущербом для мировой экономики. Вопросы надежности и безопасности программного обеспечения постоянно поднимаются на страницах этой книги.

Судьбу создаваемого языка программирования невозможно предвидеть. Бывает, что язык, предназначенный для универсального применения и разрабатываемый при моральной и материальной поддержке крупных организаций, в конечном счете занимает на рынке весьма ограниченную нишу. Напротив, язык, предназначавшийся для ограниченного или местного применения, вдруг становится популярным и используется в таких условиях и для решения таких задач, о которых и не помышляли его создатели. Эта книга посвящена именно таким языкам.

У всех успешных языков есть одно важное общее качество: они явились плодом интеллектуальных усилий одного-единственного человека или небольшой группы единомышленников. Их разработчики – это выдающиеся умы программирования, которым хватило опыта, предвидения, энергии, настойчивости и таланта, чтобы пройти весь путь от начальной реализации – через эволюцию, учитывающую опыт, – к стандартизации на практике (де-факто) и соответствующими органами (де-юре).

В этой книге читатель сможет познакомиться с этими выдающимися людьми. Все они подробно отвечают на вопросы о созданных ими языках и обстоятельствах, способствовавших их успеху. Они открыто

признают роль как удачных решений, так и стечения благоприятных обстоятельств. Кроме того, сказанное разработчиками во время беседы позволяет понять их личные устремления и образ мысли, не менее интересные, чем сам язык.

Сэр Тони Хоар

Сэр Тони Хоар – лауреат премии Тьюринга, присуждаемой ACM, и премии Киото; в течение 50 лет является ведущим исследователем компьютерных алгоритмов и языков программирования. В своей первой научной работе (1969) исследовал идею доказательства корректности программ и предложил проектировать языки программирования таким образом, чтобы они облегчали написание корректных программ. К его радости, этой идее следует все больше создателей языков программирования.

Введение

Создание программ – тяжелый труд, во всяком случае, если эти программы должны выдержать проверку практикой, временем и разными условиями эксплуатации. Технологии программирования в течение последних пятидесяти лет стремились облегчить написание программ, с этой же целью проектировались и новые языки программирования. Но в чем заключаются существующие здесь трудности?

В большинстве книг и статей на эту тему речь идет об архитектуре, технических требованиях и аналогичных вопросах, относящихся к *программному обеспечению*. Но не связаны ли трудности с самим процессом *написания* программ? Иными словами, не следует ли уделить больше внимания такому аспекту работы программиста, как общение и *язык*, помимо чисто технологических проблем?

Ребенок учится говорить в самые первые годы своей жизни, а в пять-шесть лет мы начинаем учить его чтению и письму. Я не знаю ни одного великого писателя, который бы научился читать и писать уже в зрелом возрасте. А вы знаете какого-нибудь выдающегося программиста, который бы научился программировать только в поздние годы жизни?

Известно, что дети осваивают иностранные языки гораздо легче, чем взрослые, и об этом стоит помнить при изучении программирования, которое предполагает освоение нового языка.

Представьте, что вы изучаете иностранный язык и не знаете, как называется некий объект. Вы пытаетесь описать его теми словами, которые знаете, в надежде, что кто-то вас поймет. Не так ли мы изо дня в день поступаем при разработке ПО? Мы описываем воображаемый объект с помощью языка программирования и надеемся, что это описание будет понято компилятором или интерпретатором. Если этого оказывается недостаточно, мы снова вызываем свое мысленное представление и пытаемся разобраться, что мы упустили или исказили.

Размышляя над этими вопросами, я решил провести ряд исследований, чтобы выяснить, для чего создаются языки программирования, как это происходит технически, как преподают и изучают языки и как они эволюционируют со временем.

У нас с Шейном была замечательная возможность совершить такое путешествие вместе с 27 выдающимися разработчиками, благодаря чему мы можем поделиться с вами их мыслями и опытом.

Данная книга посвящена важнейшим теоретическим и практическим аспектам создания успешного языка программирования, залогам его популярности, а также решению текущих вопросов, встающих перед теми, кто занимается непосредственной разработкой на нем. Если вы хотите узнать, как создаются удачные языки программирования, эта книга наверняка будет вам полезна.

Если вы ожидаете вдохновляющих идей, касающихся программ и языков программирования, то приготовьте маркер или даже два, потому что, уверяю вас, на последующих страницах вы встретите немало таких мыслей.

Федерико Бьянкуцци

Структура книги

Порядок глав-интервью в этой книге выбран так, чтобы увлекательное путешествие по ней радовало разнообразием. Наслаждайтесь и перечитывайте.

Глава 1 «C++» (Бьерн Страуструп)

Глава 2 «Python» (Гвидо ван Россум)

Глава 3 «APL» (Эдин Д. Фалкофф)

Глава 4 «Форт» (Чарльз Д. Мур)

Глава 5 «Бейсик» (Томас Е. Курц)

Глава 6 «AWK» (Альфред В. Ахо, Питер Вайнбергер и Брайан Керниган)

Глава 7 «Lua» (Луис Энрике де Фигейреду и Роберто Иерусалимский)

Глава 8 «Haskell» (Саймон Пейтон-Джонс, Пол Худак, Филип Уодлер и Джон Хьюз)

Глава 9 «ML» (Робин Милнер)

Глава 10 «SQL» (Дон Чемберлен)

Глава 11 «Objective-C» (Том Лав и Брэд Кокс)

Глава 12 «Java» (Джеймс Гослинг)

Глава 13 «C#» (Андерс Хейлсберг)

Глава 14 «UML» (Айвар Якобсон, Джеймс Рамбо и Гради Буч)

Глава 15 «Perl» (Ларри Уолл)

Глава 16 «PostScript» (Чарльз Гешке и Джон Э. Уорнок)

Глава 17 «Eiffel» (Бертран Мейер)

В разделе «Участники интервью» приведены их биографии.

Принятые типографские соглашения

В этой книге приняты следующие обозначения:

Курсив

Для выделения новых терминов, URL, имен файлов и утилит.

Моноширинный

Для выделения содержимого программ, файлов.

Как с нами связаться

Комментарии и вопросы, связанные с содержанием книги, просим направлять в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международные и местные звонки)
707-829-0104 (факс)

У этой книги есть веб-страница, где приводятся замеченные ошибки, примеры и различная дополнительная информация. Ее адрес:

http://www.oreilly.com/catalog/9780596515171

Комментарии и технические вопросы по этой книге присылайте по адресу:

bookquestions@oreilly.com

Дополнительные сведения о наших книгах, конференциях, Центрах ресурсов и сети O'Reilly ищите на сайте:

http://www.oreilly.com

Safari® Books Online

Safari – это решение, превосходящее обычные электронные книги. Это виртуальная библиотека, позволяющая вести поиск среди тысяч лучших технических книг, копировать из них примеры кода, загружать главы и быстро находить ответ, когда вам нужна точная и актуальная информация. Бесплатно опробовать систему можно на *http://my.safaribooksonline.com*.

1

C++

C++ выделяется среди языков программирования любопытными особенностями: он основан на языке Си, включает объектные идеи из Simula, стандартизирован ISO и его проектирование велось под лозунгами «не нужно платить за то, чем не пользуешься» и «пользовательские и встроенные типы поддерживаются в равной мере». Несмотря на его популяризацию в 1980–1990-х годах в качестве инструмента для объектно-ориентированного программирования и создания графических интерфейсов пользователя, главным его вкладом в программирование стало повсеместное распространение общей технологии программирования, воплощенной в нем в виде стандартной библиотеки шаблонов (Standard Template Library). Более новые языки, такие как Java и C#, попытались вытеснить C++, но в ожидаемой новой версии стандарта C++ должны появиться новые долгожданные функции. Создатель этого языка Бьерн Страуструп (Bjarne Stroustrup) по сей день остается одним из главных его пропагандистов.

Проектные решения

Почему вы предпочли расширить существующий язык, а не создать новый?

Бьерн Страуструп: В начале пути, в 1979 году, я ставил перед собой цель помочь программистам разрабатывать системы. И сейчас стремлюсь к тому же. Чтобы язык действительно оказался полезным в практической работе, а не стал чисто академическим упражнением, он должен обладать всеми средствами для применимости в прикладной области. Язык, создаваемый не для теоретических исследований, должен решать какую-то задачу. Задачи, которыми я тогда занимался, были связаны с проектированием операционных систем, сетевым взаимодействием и моделированием. Мне и моим коллегам нужен был язык, на котором можно описать структуру программы, как это делается в Simula (то, что обычно называют объектно-ориентированным программированием), но на котором можно также писать эффективный низкоуровневый код, что позволяет делать Си. В 1979 году не было языка, решавшего обе эти задачи, иначе я воспользовался бы им. Мне не слишком хотелось изобретать новый язык программирования: мне просто нужен был инструмент для решения некоторых задач.

Надо сказать, что создание нового языка на базе существующего во многом оправданно. Базовый язык дает вам основные синтаксические и семантические структуры, полезные библиотеки и готовую культуру. Необязательно было выбирать Си, можно было строить C++ на базе какого-нибудь другого языка. Почему я выбрал Си? Вопрос может показаться лишним, если учесть, что я работал в Исследовательском вычислительном центре Bell Labs, где рядом со мной (или неподалеку) находились Деннис Ритчи, Брайан Керниган и другие гиганты мира UNIX. Но я серьезно подошел к выбору базового языка.

В частности, система типов Си проста и не строго соблюдается (как сказал Деннис Ритчи, «Си – это язык с сильной типизацией и слабым контролем»). «Слабый контроль» был предметом моего беспокойства и до сего дня служит источником проблем для программистов на C++. Кроме того, Си применялся тогда не столь широко, как сейчас. Выбирая этот язык в качестве базы для C++, я выражал доверие заложенной в основу Си вычислительной модели (входящей в нее «сильной типизации») и доверие своим коллегам. Выбор был сделан с учетом моего знакомства (в качестве пользователя или участника разработки) с большинством языков высокого уровня, применяемых в то время в системном программировании. Стоит напомнить, что в те времена основная часть программ «для работы с железом», к которым предъявлялись жесткие требования по производительности, создавалась на ассемблере. Появление UNIX стало крупным событием во многих отношениях,

в том числе благодаря использованию Си даже в тех задачах программирования, которые были наиболее требовательны к ресурсам.

Итак, я выбрал базовую модель Си, предпочтя ее системам с более строгим контролем типов. Но мне очень хотелось иметь возможность строить программы так, как это делается с помощью классов Simula, поэтому я переделал их для модели памяти и вычислений Си. В результате появилось нечто весьма выразительное и гибкое, работавшее тем не менее со скоростью, мало уступающей ассемблеру, и не требующее огромной системы поддержки этапа исполнения.

Почему вы решили поддерживать различные парадигмы?

Бьерн: Потому что сочетание разных стилей часто способствует появлению лучшего кода, если понимать под «лучшим» такой код, который самым прямым образом описывает замысел, быстрее работает, легче сопровождается и т. п. Те, кто с этим не согласны, обычно выбирают стиль программирования, содержащий все конструкции, которые могут понадобиться (например, «обобщенное программирование – просто разновидность ООП»), или отказываются от определенных областей применения (например, «компьютер 1 ГГц / 1 Гбайт есть у всех»).

Язык Java предназначен исключительно для объектно-ориентированного программирования. Есть ли ситуации, в которых код на Java оказывается из-за этого сложнее, чем соответствующий код на C++, использующий возможности обычного программирования?

Бьерн: Пожалуй, создатели Java – а в еще большей мере те, кто продвигал его на рынке, – придавали ООП такое большое значение, что дошли до абсурда. Когда Java только появился, обещая скромность и простоту, я высказал предположение, что в случае успеха Java значительно увеличится в размере и сложности. Так оно и случилось.

Например, необходимость приведения типа `Object` для того, чтобы получить значение из контейнера (вроде `(Apple)c.get(i)`), оказывается нелепым следствием невозможности установить тип объекта в контейнере. Многословно и неэффективно. Теперь, с некоторым опозданием, в Java появились средства обобщенного программирования (generics). Другими примерами возросшей сложности языка (все для удобства программистов) служат перечисления, отражение и внутренние классы.

Факт то, что сложность все равно где-то проявится, – если не в конструкции языка, то в многочисленных приложениях или библиотеках. Аналогичным образом, навязываемое Java размещение любого алгоритма (операции) в классе приводит к таким нелепым вещам, как классы без данных, состоящие только из статических функций. Все же в математике неспроста пишут `f(x)` и `f(x,y)`, а не `x.f()`, `x.f(y)` или `(x,y).f()`; в последнем случае делается попытка представить «подлинный объектно-

ориентированный метод» с двумя аргументами и избавиться от несимметричности, свойственной `x.f(y)`.

В C++ многие проблемы логики и обозначений объектной ориентации решаются сочетанием абстрагирования данных и методов обобщенного программирования. Классическим примером служит `vector<T>`, где `T` может быть любым типом, допускающим копирование, включая встроенные типы, указатели на ОО-иерархии и пользовательские типы, такие как строки и комплексные числа. И все это достигается без дополнительной нагрузки на вычислительные ресурсы, наложения ограничений на структуры данных или особых правил для стандартных библиотечных компонентов. Другой пример, когда классическая модель ООП с единичной диспетчеризацией оказывается неудовлетворительной, это операция, требующая обращения к двум классам, скажем `operator*(Matrix,Vector)`, и не являющаяся обычным «методом» какого-либо из классов.

Одно из фундаментальных различий между C++ и Java состоит в способе реализации указателей. В некотором смысле можно сказать, что в Java нет настоящих указателей. В чем разница между этими двумя подходами?

Бьерн: Нет, конечно, в Java есть указатели. На самом деле, там почти все неявно оказывается указателем. Просто там их называют *ссылками* (*references*). В том, что указатели неявные, есть свои преимущества и недостатки. Попутно отмечу, что наличие подлинно локальных объектов (как в C++) также имеет как преимущества, так и недостатки.

Принятое в C++ размещение локальных переменных в стеке и наличие подлинных переменных-членов всех типов обеспечивает удобство универсальной семантики, хорошую поддержку идеи семантики значений, компактность структуры и минимальные издержки обращения и лежит в основе поддержки общего управления ресурсами в C++. Это важно, а вездесущие неявно используемые в Java указатели (они же ссылки) закрывают путь ко всем этим возможностям.

Рассмотрим различные издержки, связанные со структурой. В C++ `vector <complex>(10)` представляется как структура, содержащая массив из 10 комплексных чисел и хранящаяся в свободной памяти. В сумме это составляет 25 слов: 3 слова на вектор, 20 слов на комплексные числа и 2 слова на заголовок массива в свободной памяти (куче). Аналогичная структура в Java (определенный пользователем контейнер для объектов определенного пользователем типа) займет 56 слов: 1 для ссылки на контейнер, 3 для контейнера, 10 для ссылок на объекты, 20 для объектов и 24 для заголовков в свободной памяти 12 независимо размещаемых объектов. Очевидно, это приблизительные расчеты, поскольку

стоимость размещения в куче определяется в каждом языке отдельно выбранной в нем реализацией. Тем не менее вывод ясен: благодаря повсеместному и неявному использованию ссылок в Java, возможно, удалось упростить модель программирования и сборку мусора, но расход памяти при этом резко вырос – и соответственно выросли стоимость доступа к памяти (осуществляемого более косвенным образом) и расходы на выделение памяти.

Чего в Java, к счастью, нет, так это допускаемого Си и C++ неправильного применения арифметики указателей. Но и эта проблема решается, если корректно писать код на C++, используя такие высокоуровневые абстракции, как потоки ввода/вывода, контейнеры и алгоритмы, а не занимаясь возней с указателями. В сущности, все массивы и значительная часть указателей относятся к нижним уровням реализации, с которыми большинство программистов не имеет дела. К сожалению, повсюду немало и скверного кода на C++, без всякой надобности действующего на низком уровне.

Есть, однако, важная область, где указатели и действия с ними оказываются большим подспорьем: прямое и эффективное описание структур данных. Ссылок Java здесь оказывается недостаточно: например, с их помощью не опишешь операцию swap. Другой пример – простота непосредственного доступа к реальной памяти с помощью указателей: в любой системе должен быть язык, который может это делать, и часто им оказывается C++.

«Отрицательная» сторона наличия указателей (и массивов в стиле Си) – это, конечно, возможность злоупотреблений: переполнение буфера, указатели на удаленную память, неинициализированные указатели и т. п. Но если корректно писать код на C++, все не столь страшно. Эти проблемы с указателями и массивами не возникают, если они употребляются внутри таких абстракций, как `vector`, `string`, `map` и т. д. Управление ресурсами с использованием областей видимости снимает большинство проблем; с остальными обычно можно справиться с помощью умных указателей и специальных дескрипторов. Те, кто привык работать на Си или придерживается старого стиля в C++, с трудом верят тому, что управление ресурсами на основе областей видимости представляет собой исключительно мощный инструмент, который при определении пользователем для соответствующих операций может решать классические проблемы с помощью меньшего объема кода, чем старые и ненадежные хаки. Вот, к примеру, простейший вид классического переполнения буфера, создающего угрозу безопасности:

```
char buf[MAX_BUF];
gets(buf); // Yuck!
```

При использовании стандартной библиотеки string проблема исчезает:

```
string s;
cin >> s;  // чтение символов, среди которых есть пробельные
```

Это тривиальные примеры, но подходящие «строки» и «контейнеры» можно придумать практически для любых задач, а начать полезно со стандартной библиотеки.

Что вы понимаете под «семантикой значений» и «общим управлением ресурсами»?

Бьерн: О *семантике значений* (value semantics) часто говорят в отношении объектов с тем свойством, что при копировании одного вы получаете два независимых экземпляра (с одинаковым значением). Например:

```
X x1 = a;
X x2 = x1; // теперь x1==x2
x1 = b;    // изменяет x1, но не x2
           // теперь x1!=x2 (если X(a)!=X(b))
```

Это как раз то, что у нас происходит с обычными числовыми типами, такими как целые, действительные и комплексные числа, и математическими абстракциями, такими как векторы. Это очень полезная идея, которую C++ поддерживает для встроенных типов и для любых пользовательских типов, если мы этого пожелаем. В Java сделано по-другому: это поддерживается для встроенных типов, таких как `char` и `int`, а для пользовательских невозможно. Как и в Simula, у всех пользовательских типов в Java поддерживается семантика ссылок. В C++ программист может выбрать поддержку той семантики, которая требуется для типа. C# следует практике C++ поддержки семантики значений пользовательских типов, но не полностью.

Общим управлением ресурсами (general resource management) называют распространенную технологию передачи объекту владения ресурсами, такими как дескриптор файла или блокировка. Если этот объект является переменной с ограниченной областью видимости, то максимальное время удержания ресурса не превосходит времени существования переменной. Обычно ресурс захватывается в конструкторе, а освобождается в деструкторе. Часто это называется RAII (Resource Acquisition Is Initialization – «получение ресурса есть инициализация») и прекрасно интегрируется с обработкой ошибок с помощью исключений. Очевидно, не со всеми ресурсами можно работать подобным образом, но когда это возможно, управление ресурсами происходит неявно и эффективно.

Принцип «ближе к аппаратной части», вероятно, был одним из главных при проектировании C++. Правильно ли будет сказать, что для C++ применялось скорее восходящее проектирование, а не нисходя-

щее, как для других языков, когда в них пытались включить абстрактно-умозрительные конструкции, которые компилятор должен был реализовать в имевшейся вычислительной среде?

Бьерн: Думаю, эпитеты «нисходящее» и «восходящее» не годятся для описания таких конструктивных решений. В контексте C++ и других языков «близость к железу» означает, что модель вычислений берется у компьютера – в виде ряда объектов в памяти и операций, определенных над объектами фиксированного размера, а не из неких математических абстракций. Это верно для C++ и Java, но не для функциональных языков. В отличие от Java, C++ основывается на реальной, а не некой абстрактной машине.

Настоящая проблема в том, как перейти от мира человеческих представлений о задачах и их решениях к ограниченному миру машин. Можно «игнорировать» человеческие интересы и получить код для машины (или хваленый машинный код, представляющий собой плохой код на Си). Можно игнорировать машину и получить прекрасную абстракцию, которая способна решать любые задачи ценой неимоверных затрат и/или утраты интеллектуальной строгости. C++ является попыткой дать самый непосредственный доступ к аппаратуре, когда он нужен (указатели и массивы), и в то же время обеспечить широкие механизмы абстрагирования, позволяющие описывать идеи высокого уровня (иерархии классов и шаблоны).

В то же время при разработке C++ и его библиотек неизменно проявлялась забота о скорости выполнения и экономии памяти. Это отразилось на всех возможностях базового языка и на средствах абстрагирования, что свойственно не всем языкам.

Применение языка

Как вы ищете ошибки в программах? Что бы вы посоветовали разработчикам на C++?

Бьерн: Путем самонаблюдения. Я долго изучаю программу, более или менее систематически углубляясь в нее, пока не возникнет понимание, позволяющее обоснованно предположить, где находится ошибка.

Другое дело – тестирование и проектирование с задачей минимизировать ошибки. Я крайне не люблю искать ошибки и делаю все возможное, чтобы избежать этой процедуры. Если я проектирую некую программную систему, то строю ее на базе интерфейсов и инвариантов, что оставляет меньше возможностей успешной компиляции и выполнения для кода с серьезными недостатками. Далее, я стараюсь обеспечить возможность его тестирования. Тестирование – это систематический поиск ошибок. Трудно систематически тестировать систему, если она

плохо структурирована, поэтому я еще раз подчеркиваю важность ясной структуры кода. Тестирование, в отличие от отладки, допускает автоматизацию и многократность. Заставить кучу людей случайным образом щелкать по экрану, проверяя, не удастся ли им завалить графическое приложение, – это не тот путь, который ведет к созданию систем высокого качества.

Советы? Общие советы давать тяжело, потому что лучшие приемы часто определяются тем, что доступно для данной системы в данных условиях разработки. Тем не менее скажу: следует определять главные интерфейсы, которые можно систематически тестировать, и писать скрипты, которые это делают. Сделайте выполнение этих скриптов возможно более автоматическим и почаще запускайте их. Обеспечьте возможность систематического тестирования всех входных и выходных точек системы. Старайтесь, чтобы система состояла из отдельных надежных компонентов: монолитность программ слишком затрудняет их анализ и тестирование.

На каком уровне нужно заниматься безопасностью программ?

Бьерн: Прежде всего, безопасность – это системная проблема. Одними только местными или частичными мерами ее не решить. Не нужно забывать, что каким бы совершенным ни был ваш код, все ваши секреты могут быть раскрыты, если удастся украсть ваш компьютер или резервную копию данных. Во-вторых, уровень безопасности и затраты на его достижение взаимозависимы: идеальная защита большинству из нас окажется не по карману, но можно постараться защитить свою систему в достаточной мере, чтобы злоумышленники задумались, не лучше ли для них будет не тратить зря время, а попытать счастья в другом месте. На практике я предпочитаю не хранить важные секреты там, где они доступны по сети, а серьезные проблемы безопасности предоставляю решать экспертам в этом деле.

Но о безопасности мы говорим еще в связи с языками и технологиями программирования. Возникла порочная тенденция считать, что каждая строка кода должна быть «защищенной» (что бы под этим ни подразумевалось), вплоть до предположения, что некто с нехорошими намерениями получил доступ к какой-то другой части системы. Это очень опасная идея, которая приводит к засорению кода беспорядочными проверками для защиты от плохо сформулированных воображаемых угроз. Код становится уродливым, громоздким и медленным. Из-за «уродливости» в нем легко скрыться ошибкам, «громоздкость» усложняет тестирование, а «медлительность» приводит к созданию обходных путей и нечестных приемов, порождающих множество дыр в защите.

Полагаю, что только одно правило должно неизменно соблюдаться в решении проблем безопасности: использовать простую модель защиты,

систематически применяя ее с помощью высококачественных аппаратных/программных средств к избранным интерфейсам. Должна быть граница, за которой можно писать простой, элегантный и эффективный код, не беспокоясь о том, что какая-то часть кода может воспользоваться другой частью со злым умыслом. Только тогда мы сможем сосредоточить свое внимание на корректности, качестве и эффективности. Думать о том, что кто-то может воспользоваться несанкционированным обратным вызовом, модулем расширения и чем-то подобным, – просто глупо. Нужно различать код, дающий отпор мошеннику, и код, который просто защищен от несчастного случая.

Не думаю, что можно придумать язык программирования, который будет полностью защищен и в то же время применим в реальных системах. Очевидно, все зависит от того, что понимать под «защищенностью» и «системой». Вероятно, можно достичь защищенности в языке для конкретной области, но моей главной областью интересов является системное программирование (в самом широком смысле), включая встроенные системы. Думаю, такого рода защищенность можно и нужно улучшить по сравнению с тем, что предлагает C++, но это лишь часть проблемы: безопасность типов и защищенность – разные вещи. Тот, кто пишет на C++, широко применяя неинкапсулированные массивы, приведение типов, неструктурированные операции `new` и `delete`, сам создает себе проблемы. Это стиль программирования, практиковавшийся в 1980-х. Правильная работа с C++ предполагает стиль с минимальным вмешательством в систему безопасности типов и управление ресурсами (включая память) и систематическое его применение.

Можно ли рекомендовать C++ для систем, где его использование обычно не приветствуется, например для системного программирования и встроенных приложений?

Бьерн: Разумеется, я его рекомендую, и не все против. На самом деле я не замечал особого сопротивления в указанных областях, исключая естественное нежелание испытывать что-то новое в устоявшихся организациях. Скорее, я наблюдаю уверенный и заметный рост применения C++. Например, я участвовал в разработке правил кодирования важного программного обеспечения для проекта истребителя JSF фирмы Lockheed Martin. Это самолет на «чистом C++». Возможно, вас не очень интересует военная авиация, но в том, как там используется C++, нет ничего специально военного, и меньше чем за год с моих личных страниц в Интернете правила кодирования JSF++ были загружены больше 100 000 раз, в основном, насколько я могу судить, разработчиками невоенных встроенных систем.

C++ применяется для встроенных систем с 1984 года, на C++ написано программное обеспечение для многих полезных гаджетов, и его приме-

нение быстро расширяется. Примерами служат мобильные телефоны с Symbian или Motorola, iPod, системы GPS. Особенно мне понравилось, как C++ использовался на марсоходах: в системах анализа сцен и автономного передвижения, в значительной части систем связи на Земле, для обработки графики.

Тот, кто убежден в несомненном превосходстве Си над C++ в эффективности, может прочесть мою статью «Learning Standard C++ as a New Language» («Стандартный C++ как новый язык», C/C++ Users Journal, May 1999), из которой узнает о философии проектирования и результатах нескольких простых экспериментов. Кроме того, комиссия по стандартам ISO C++ выпустила технический отчет о продуктивности, в котором много сказано о проблемах и мифах применения C++ в случаях, когда быстродействие имеет особое значение (вы найдете его в Сети, поискав строку «Technical Report on C++ Performance»).[1] В частности, в этом отчете говорится о проблемах встроенных систем.

Ядра таких систем, как Linux и BSD, по-прежнему пишутся на Си. Почему там не перешли на C++? Связано ли это с парадигмой ООП?

Бьерн: В основном, это консерватизм и инерция. Кроме того, GCC развивался недостаточно быстро. Ряд членов сообщества Си просто проявляют упрямство, исходя из опыта десятилетней давности. На C++ уже десятки лет пишут другие операционные системы, значительную часть системных программ и даже код, требующий выполнения в реальном времени и критичный в отношении надежности. Вот лишь некоторые примеры: Symbian, OS/400 и K42 IBM, BeOS, частично Windows. Есть и много программ с открытым исходным кодом, написанных на C++ (например, KDE).

Похоже, C++ приравнивают к использованию ООП. C++ никогда не был всего лишь объектно-ориентированным языком. В 1995 году я написал статью «Why C++ is not just an Object-Oriented Programming Language» («Почему C++ – это не просто объектно-ориентированный язык»); ее можно найти в сети.[2] Идея была и остается в том, чтобы поддерживать несколько стилей программирования («парадигм», если вам нравятся мудреные слова) и их комбинации. Другая парадигма, наиболее подходящая в контексте высокой скорости и близости к аппаратуре, это «обобщенное программирование» (generic programming, или GP). Стандартная библиотека ISO C++ в большей мере относится к GP, чем к ООП, благодаря существующей в ней поддержке алгоритмов и контейнеров (STL). Обобщенное программирование в типичном стиле

[1] *http://www.open-std.org/JTC1/sc22/wg21/docs/TR18015.pdf*

[2] *http://www.research.att.com/~bs/oopsla.pdf*

C++ с активным применением шаблонов широко используется там, где требуются одновременно абстракция и высокая производительность.

Я не видел программы, которую лучше было бы написать на Си, чем на C++. Не думаю, что такая существует. В конце концов, на C++ можно писать в стиле, близком к Си. Никто не заставляет вас пускаться во все тяжкие с исключениями, иерархиями классов и шаблонами. Хороший программист использует более продвинутые функции там, где они позволяют отчетливее выразить идею, избежав при этом лишних накладных расходов.

Следует ли программисту переносить свой код с Си на C++? Какие преимущества он получит, выбрав C++ в качестве языка программирования общего типа?

Бьерн: Вероятно, вы имеете в виду код, изначально написанный на Си программистом, начинавшим как Си-программист. Для многих, если не для большинства программ и программистов на C++ эта ситуация давно изжита. К сожалению, во многих учебных планах еще сохраняется правило «сначала Си», но оно уже не считается обязательным.

Иногда люди переходят с Си на C++, обнаружив, что поддержка стилей программирования, принятых в Си, в C++ лучше, чем в самом Си. Проверка типов в C++ более строгая (вы не забудете описать тип функции или ее аргументов), и для ряда стандартных операций есть поддержка безопасности типов в нотации, например при создании объектов (включая инициализацию) и констант. Мне встречались люди, очень довольные тем, что благодаря этим возможностям они избавились от своих прежних проблем. Обычно при этом также принимаются некоторые библиотеки C++, которые можно не считать объектно-ориентированными, вроде стандартных библиотек векторов, GUI или некоторых библиотек, ориентированных на приложение.

Обычное использование простых типов, определяемых пользователем, таких как `vector`, `string` или `complex`, не требует смены парадигмы. При желании можно работать с ними так же, как со встроенными типами. Если у кого-то в программе встретится `std::vector`, значит ли это, что он «применяет ООП»? Я бы не стал это утверждать. А если кто-то использует C++ GUI, не добавляя новую функциональность, «применяет ли он ООП»? Я бы сказал, что да, потому что в этом случае обычно требуется понимание и использование наследования.

Выбор C++ в качестве «языка для обобщенного программирования» дает вам готовые стандартные контейнеры и алгоритмы (в составе стандартной библиотеки). Это значительное подспорье во многих ситуациях и значительный шаг в абстрагировании по сравнению с Си. Кроме того,

открывается возможность использовать библиотеки, такие как Boost, и оценить некоторые технологии функционального программирования, входящие в обобщенное программирование.

Однако мне кажется, что этот вопрос несколько вводит в заблуждение. Я хотел бы представить C++ не в качестве «ОО-языка» или «GP-языка», а как язык, который поддерживает:

- Программирование в стиле Си
- Абстрагирование данных
- Объектно-ориентированное программирование
- Обобщенное программирование

Важно, что он поддерживает стили программирования, сочетающие перечисленные возможности (если хотите, называйте это «программированием во множественных парадигмах»), и все это с уклоном в сторону системного программирования.

ООП и параллелизм

В среднем, сложность и объем (как количество строк кода) программных систем растут год от года. Насколько хорошо масштабируется ООП в этих условиях и не создает ли оно проблемы? У меня такое ощущение, что стремление к созданию объектов многократного использования все усложняет, в конечном итоге удваивая трудозатраты. Сначала требуется разработать инструмент многократного использования. Затем, когда понадобилось какое-то изменение, приходится чем-то заполнять разрыв со старым компонентом, а это сужает круг предлагаемых решений.

Бьерн: Вы неплохо описали важную проблему. ООП – это мощный набор технологий, которые могут быть полезны, но для этого нужно правильно их использовать, применяя для решения тех задач, где эти технологии могут дать положительный результат. Весьма серьезная проблема при создании любого кода, использующего наследование и статический контроль интерфейсов, связана с тем, что спроектировать хороший базовый класс (интерфейс ко многим еще не известным классам) можно лишь, обладая даром предвидения и большим опытом. Откуда разработчику базового класса (абстрактного класса, интерфейса или как там еще вы его назовете) знать, определил ли он все, что может потребоваться всем порожденным от него классам в будущем? Откуда разработчику знать, будет ли определенный им класс разумно реализован всеми порожденными от него классами в будущем? Откуда разработчику базового класса знать, не станет ли определенный им класс

серьезной помехой для чего-то, необходимого классам, порожденным от него в будущем?

Вообще говоря, нам это неизвестно. В условиях, когда мы можем навязать использование своей конструкции, люди вынуждены будут приспосабливаться – часто путем создания каких-то уродливых обходных путей. Там, где нет одной ответственной организации, может возникать множество несовместимых интерфейсов, обеспечивающих по существу одну и ту же функциональность.

Общего решения таких проблем нет, но обобщенное программирование показывает выход во многих случаях, когда ОО-подход оказывается неудачным. Примечателен простой пример контейнеров: через иерархию наследования нельзя удовлетворительно описать ни понятие «быть элементом», ни понятие «быть контейнером». Однако с помощью обобщенного программирования можно дать эффективные решения. Примером тому служит STL (составная часть стандартной библиотеки C++).

Эта проблема специфична для C++ или касается также других языков программирования?

Бьерн: Это общая проблема всех языков со статическим контролем интерфейсов к иерархиям классов. Это такие языки, как C++, Java и C#, – в отличие от языков с динамической типизацией, таких как Smalltalk и Python. C++ решает эту проблему с помощью обобщенного программирования, и хорошим примером служат контейнеры и алгоритмы стандартной библиотеки C++. Главной особенностью языка здесь являются шаблоны, обеспечивающие модель позднего контроля типа, когда на этапе компиляции создается эквивалент тому, что языки с динамической типизацией осуществляют на этапе исполнения. Недавнее добавление в Java и C# «генериков» – это попытка следовать по пути C++, которую часто – по моему мнению, ошибочно – расценивают как усовершенствование шаблонов.

Особой популярностью пользуется «рефакторинг» как попытка решения этой проблемы грубым силовым методом – путем обыкновенной реорганизации кода, в котором первоначальная конструкция интерфейсов устарела.

Если это общая проблема ООП, можно ли быть уверенным, что достоинства ООП перевешивают его недостатки? Не является ли трудность хорошего объектно-ориентированного проектирования причиной всех прочих проблем?

Бьерн: Тот факт, что в каком-то, пусть даже большом числе случаев есть проблемы, не отменяет того, что существует множество красивых, эффективных и удобных в сопровождении систем, написанных

на этих языках. Объектно-ориентированное проектирование – это один из основных способов проектирования систем, а статический контроль интерфейсов – источник не только проблем, но и преимуществ.

В разработке программного обеспечения нет какого-то одного «корня всех зол». Проектировать трудно во многих отношениях. Часто недооценивают интеллектуальные и практические трудности, возникающие в процессе создания важных систем, использующих программное обеспечение. Разработку программного обеспечения не свести к простому механическому «сборочному конвейеру». Для создания хороших крупных систем требуются творчество, инженерные принципы и эволюционные изменения.

Есть ли связь между парадигмой ООП и параллелизмом? Влияет ли на реализацию проектов или природу ОО-проектов распространившаяся в настоящее время потребность в улучшенных возможностях параллельной обработки?

Бьерн: На самом деле, связь между объектно-ориентированным программированием и параллелизмом очень давняя. В языке программирования Simula 67, в котором впервые появилась прямая поддержка объектно-ориентированного программирования, был механизм описания параллельной работы. Первая библиотека C++ содержала поддержку того, что сейчас называют *потоками* (threads). В 1988 году у нас в Bell Labs C++ работал на 6-процессорной машине, и мы не были единственными, кто работал в этом направлении. В 1990-х было создано не меньше двух десятков экспериментальных диалектов и библиотек C++ для задач, связанных с распределенным и параллельным программированием. Нынешний ажиотаж вокруг многоядерности – это не первое мое столкновение с параллелизмом. На самом деле, распределенные вычисления были темой моей докторской диссертации, и я не перестаю следить за этой областью.

Нужно заметить, что те, кто впервые сталкивается с параллелизмом, многоядерностью и т. п., часто делают ошибку, недооценивая стоимость выполнения какой-то работы на другом процессоре. Стоимость запуска задачи на другом процессоре (ядре) и обращения этой задачи к данным, находящимся в памяти «вызывающего» процессора (в виде копирования или «удаленного» доступа), может быть в 1000 или больше раз выше, чем для привычных нам вызовов функций. Кроме того, как только появляется параллелизм, существенно возрастает вероятность ошибок. Чтобы эффективно использовать возможности параллельной работы, предоставляемые аппаратной частью, необходимо переосмыслить структуру наших программ.

К счастью, на помощь приходят исследования, проводившиеся на протяжении десятков лет, хотя их результаты могут озадачить. По правде говоря, этих исследований столько, что почти невозможно определить, что тут реально, а тем более оптимально. Хорошая отправная точка для знакомства с этой областью – доклад на HOPL-III о языке Emerald. Это первый язык, в котором при рассмотрении проблем языков и систем учитывалась стоимость. Важно также отличать параллельную обработку данных, которая десятилетиями ведется в научных расчетах – в основном, на Фортране, – от связанных между собой программных единиц с «обычным последовательным кодом» (например, процессов и потоков), выполняемых на разных процессорах. Думаю, широкое признание в этом новом мире множественных ядер и кластеров сможет получить такая система программирования, которая поддерживает оба типа параллелизма, причем, возможно, каждый из них в нескольких вариантах. Это совсем не просто, и возникающие здесь проблемы выходят далеко за рамки обычных проблем языков программирования: нам придется в комплексе рассматривать проблемы языков, систем и приложений.

Готов ли C++ к параллельной работе? Разумеется, можно создать библиотеки, которые будут делать все, что угодно, но не потребуется ли для перехода к параллелизму серьезно переработать сам язык и стандартную библиотеку?

Бьерн: Почти готов. C++0*x* будет точно готов. Чтобы быть готовым к параллельным вычислениям, язык должен прежде всего располагать точной спецификацией модели памяти, что позволит разработчикам компиляторов воспользоваться преимуществами современных процессоров (такими как длинные конвейеры, большие кэши, буферы предсказания ветвления, статическое и динамическое переупорядочение команд и т. п.). Кроме того, необходимы некоторые небольшие расширения языка: локальная память потоков и атомарные типы данных. Еще можно добавить поддержку параллелизма через библиотеки. Естественно, первой новой стандартной библиотекой станет библиотека потоков, поддерживающая переносимость программ между разными системами, например Linux и Windows. Конечно, такие библиотеки существуют давно, но не в виде стандартных.

Потоки плюс какая-то форма блокировки для избежания ситуации гонок – это едва ли не худший способ непосредственно осуществить параллелизм, но он необходим в C++ для поддержки имеющихся приложений и сохранения своей функции языка системного программирования в традиционных операционных системах. Существуют прототипы этой библиотеки, основанные на многолетней практике активного применения.

Одна из важных проблем параллелизма – выбор «упаковки» для задачи, которая должна выполняться параллельно с другими. Предполагаю, что в C++ предпочтение будет отдано «объекту-функции». Такой объект может содержать все необходимые данные и передаваться всеми необходимыми способами. C++98 хорошо решает эту задачу с помощью именованных операций (именованных классов, от которых порождаются объекты-функции), и этот прием всюду используется для параметризации в обобщенных библиотеках (например, STL). В C++0*x* облегчается написание «одноразовых» объектов-функций благодаря наличию «лямбда-функций», которые можно писать в контексте выражения (т. е. в виде функций-аргументов), и соответствующей генерации объектов-функций («замыканий»).

Следующие этапы интереснее. Сразу после C++0*x* комиссия планирует сделать технический доклад по библиотекам. Почти наверняка там будет сказано о пулах потоков и вариантах отъема нагрузки. Имеется в виду создание стандартного механизма, с помощью которого пользователь сможет запрашивать параллельное выполнение сравнительно небольшого объема работы («задач»), не занимаясь созданием, уничтожением, блокировкой потоков; при этом в качестве задач, возможно, будут выступать объекты-функции. Кроме того, появятся средства для обмена данными между территориально удаленными процессами с помощью сокетов, потоков ввода/вывода и т. д. в духе `boost::networking`.

Мне кажется, что всякие интересные вещи, связанные с параллелизмом, появятся в виде ряда библиотек, поддерживающих модели параллельной обработки с разной логикой.

Многие современные системы состоят из компонентов, распределенных по сети. В эпоху веб-приложений и мэшапов эта тенденция может усилиться. Следует ли отражать в языке эти особенности сетевой работы?

Бьерн: Существует много видов параллелизма. Одни из них стремятся улучшить производительность или время отклика программы на отдельном компьютере или в кластере, другие решают задачу географической разнесенности, а третьи действуют на том уровне, куда программисты обычно не опускаются (конвейеры, кэши и т. п.)

C++0*x* предоставит программистам ряд средств, которые избавят их от необходимости заниматься деталями низкого уровня благодаря появлению «контракта» между архитекторами машин и создателями компиляторов – «модели машины». Кроме того, в нем будет библиотека потоков, обеспечивающая основу соответствия кода процессорам. На этой основе другие библиотеки смогут поддерживать другие модели. Я был бы рад, если бы стандартная библиотека C++0*x* поддерживала модели

параллелизма более высокого уровня, более простые для применения, но в данное время это, видимо, маловероятно. В дальнейшем – будем надеяться, вскоре после C++0*x* – у нас появятся другие библиотеки, определенные в техническом докладе: пулы потоков и фьючерсы, а также библиотека потоков ввода/вывода для глобальных сетей (например, TCP/IP). Такие библиотеки есть, но некоторые считают, что они не доведены до уровня стандарта.

Когда-то я надеялся, что C++0*x* решит некоторые из давних проблем C++ с передачей данных, определив стандартный вид маршалинга (или сериализации), но этого не произошло. Поэтому сообществу C++ придется по-прежнему организовывать верхний уровень распределенных вычислений и создания распределенных приложений с помощью нестандартных библиотек и/или сред (например, CORBA или .NET).

Самая первая библиотека C++ (в действительности первая библиотека Си с классами) предоставляла облегченную форму параллелизма, со временем появились сотни библиотек и сред на C++ для одновременных, параллельных и распределенных вычислений, но сообщество не смогло достичь согласия по поводу стандартов. Мне кажется, проблема частично обусловлена тем, что получить какой-нибудь крупный результат в этой области можно только в результате больших финансовых вложений, а крупные игроки предпочитали тратить свои деньги на создание собственных закрытых библиотек, сред и языков. От этого сообщество C++ пострадало в целом.

Будущее

Дождемся ли мы когда-нибудь C++ 2.0?

Бьерн: Это зависит от того, что вы подразумеваете под «C++ 2.0». Если имеется в виду новый язык, созданный в какой-то мере с чистого листа и сочетающий в себе все достоинства C++, но лишенный его недостатков (в том или ином смысле «достоинств» и «недостатков»), то я не знаю ответа. Я был бы рад появлению значительного нового языка в традициях C++, но такого пока не предвидится, поэтому позвольте мне сосредоточить свои усилия на новом стандарте ISO C++ с кодовым наименованием C++0*x*.

Для многих он окажется «C++ 2.0», поскольку будет располагать новыми языковыми особенностями и новыми стандартными библиотеками, но он будет почти на 100% совместим с C++98. Мы называем его C++0*x* в надежде, что он станет C++09. Если мы задержимся и *x* придется сделать шестнадцатеричным, то я (и мои коллеги) будем весьма огорчены и расстроены.

C++0*x* будет почти на 100% совместим с C++98. У нас нет намерения сделать ваш код неработоспособным. Наиболее существенные несовместимости связаны с появлением нескольких новых ключевых слов, таких как `static_assert`, `constexpr` и `concept`. Мы постарались уменьшить их эффект, выбрав такие новые ключевые слова, которые не очень широко используются. Основные усовершенствования:

- Поддержка современных машинных архитектур и параллелизма: модель машины, библиотека потоков, локальная память потоков и атомарные операции, асинхронный механизм возврата значений («фьючерсы»).
- Улучшенная поддержка обобщенного программирования: система типов для типов, комбинации типов и комбинации типов и целых – идеи, которые должны улучшить проверку определений и применений шаблонов и улучшить перегрузку шаблонов. Выведение типа на основе инициализаторов (`auto`), обобщенные списки инициализаторов, обобщенные константные выражения (`constexpr`), лямбда-выражения и прочее.
- Ряд мелких расширений языка, таких как статические операторы контроля (assertions), семантика переноса (move semantics), улучшенные перечисления, имя для нулевого указателя (`nullptr`) и т. д.
- Новые стандартные библиотеки для поддержки регулярных выражений, хеш-таблиц (например, `unordered_map`), «умные» указатели и т. д.

Все подробности можно найти на сайте комитета ISO C++.[1] Общий обзор есть на моей странице C++0*x* FAQ.[2]

Учтите, пожалуйста, что говоря о целостности старого кода, я имею в виду базовый язык и стандартную библиотеку. Старый код не будет работать, если в нем есть нестандартные расширения, сделанные каким-нибудь производителем компиляторов, или устаревшие нестандартные библиотеки. Мой опыт показывает, что жалобы на «переставший работать код» или «нестабильность» связаны с наличием собственных функций или библиотек. Например, если вы переходите на другую операционную систему и у вас использована не переносимая библиотека GUI, а какая-то другая, возможно, вам придется поработать над кодом интерфейса пользователя.

---

[1] *http://www.open-std.org/jtc1/sc22/wg21/*

[2] *http://www.research.att.com/~bs/C++0xFAQ.html*

Почему вы не создадите какой-нибудь совсем новый язык?

Бьерн: Тут возникают некоторые важные вопросы:

- Какую задачу должен решать этот новый язык?
- Чьи задачи он должен решать?
- Что кардинально нового он может предоставить (в сравнении с существующими языками)?
- Можно ли организовать успешное развертывание этого нового языка (в условиях, когда есть много языков с хорошей поддержкой)?
- Не станет ли разработка нового языка лишь приятным развлечением вместо утомительной работы, помогающей людям создавать более эффективные инструменты и системы для реальной работы?

Я пока не смог удовлетворительным образом ответить на эти вопросы.

Это не значит, что я считаю C++ в своем роде идеальным языком. Это не так; я уверен, что можно было бы создать язык в десять раз меньше C++ (не важно, как мерить размер) и обеспечивающий примерно те же возможности, что и C++. Однако для нового языка недостаточно, чтобы он мог делать то же, что и уже имеющийся, только немного лучше и немного элегантнее.

Какие уроки изобретения, последующего развития и принятия вашего языка могут оказаться полезными разработчикам компьютерных систем в настоящее время и, возможно, в будущем?

Бьерн: Это важный вопрос: учит ли нас история? Если да, то как? Какие уроки можно извлечь? На раннем этапе разработки C++ я сформулировал несколько «эмпирических правил», которые опубликованы в «The Design and Evolution of C++» (Addison-Wesley) и обсуждались в двух моих докладах для HOPL. Очевидно, что всякий серьезный проект разработки языка должен основываться на некоторой группе принципов, и сформулированы они должны быть как можно раньше. Фактически это вывод, сделанный по опыту работы над C++: я слишком поздно сформулировал принципы проектирования C++ и не разъяснил их достаточно широко. В результате многие придумали собственные обоснования конструкции C++, которые иногда были довольно удивительны и привели к большой путанице. До сего дня некоторые считают C++ едва ли не провалившейся попыткой сконструировать нечто вроде Smalltalk (нет, C++ не задумывался как Smalltalk-подобный язык, он следует объектной модели Simula) или всего лишь попыткой исправить некоторые недостатки Си, чтобы писать код в стиле Си (нет, C++ не задумывался как Си с несколькими поправками).

Назначение языка программирования (если это не эксперимент) – помочь созданию добротных систем. Дело в том, что идеи проектирова-

ния систем и проектирования языков тесно взаимосвязаны. Под «добротностью» я понимаю в данном контексте «корректность, легкость сопровождения и приемлемые затраты ресурсов». Очевидно, что здесь отсутствует такой компонент, как «легкость написания кода», но для тех систем, которые меня больше интересуют, это качество вторично. «Ускоренные методы разработки» – не мой идеал. Иногда столь же важно указать, что не является главной задачей, как и что ею является. К примеру, я не имею ничего против ускоренной разработки – никто, находясь в здравом уме, не захочет тратить на проект больше времени, чем требуется, – но я бы предпочел, чтобы было меньше ограничений на области применения и производительность. Моей целью в разработке C++ была и остается возможность ясного изложения идей, которая приводит к эффективному по времени и памяти коду.

Си и C++ сохраняют стабильность на протяжении десятилетий. Это крайне важно для промышленного производства программ. У меня есть небольшие программы, которые фактически не модифицировались с начала 1980-х. За такую стабильность приходится платить, но языки, которые ее не обеспечивают, просто непригодны для крупных и долгоживущих проектов. Фирменные языки и языки, пытающиеся гнаться за модой, часто упускают это из виду, что приводит к большим неприятностям.

Приходится подумать о том, как управлять развитием. Насколько велик может быть объем изменений? Какова степень детализации изменений? Менять язык каждый год вместе с выпуском новых версий продукта – это слишком произвольно и приводит к появлению фактических подмножеств языка, отказу от некоторых библиотек или возможностей языка и/или большим затратам на обновление. Кроме того, одного года просто недостаточно для созревания важных функций, поэтому такой подход приводит к сырым решениям и тупикам. С другой стороны, 10-летний цикл стандартизации ISO таких языков, как Си и C++, слишком велик и приводит к закоснению некоторой части сообщества (и комиссии).

Успешный язык порождает сообщество: это сообщество пользуется общими методами, инструментами и библиотеками. Фирменные языки обладают здесь естественным преимуществом: они могут наращивать свою долю рынка с помощью маркетинга, конференций и «бесплатных» библиотек. Такие инвестиции могут окупаться благодаря значительному вкладу сторонних разработчиков, росту размеров и активности сообщества. Проекты Sun с Java показали, насколько любительскими и недостаточно финансированными были все прежние попытки создать общепринятый универсальный (более или менее) язык. Разительный контраст составляли усилия Министерства обороны США сделать Ada

основным языком и старания мои и моих друзей упрочить положение C++ в отсутствие финансирования.

Я не одобряю некоторые тактические приемы внедрения Java, например продажу продукта руководителям, не занимающимся программированием, но они показывают, чего можно добиться. Примером успеха, не связанного с фирмами, служат сообщества Python и Perl. Успехи в создании сообщества C++ слишком невелики и малочисленны, если учесть размер этого сообщества. Конференции ACCU замечательны, но почему не происходит регулярных крупных международных конференций по C++, начиная примерно с 1986 года? Библиотеки Boost замечательны, но почему не существует центрального хранилища библиотек C++, начиная примерно с 1986 года? Есть тысячи библиотек C++ с открытым исходным кодом. У меня нет даже полного перечня коммерческих библиотек C++. Я не стану отвечать на эти вопросы, но замечу лишь, что любой новый язык должен как-то управлять центробежными силами, возникающими в большом сообществе, иначе последствия будут тяжелыми.

Универсальному языку нужны реакция и одобрение нескольких сообществ, включая корпоративных программистов, преподавателей, исследователей в академических кругах и на производстве, сообщество open source. Эти сообщества могут пересекаться, но отдельные меньшие сообщества часто считают себя самодостаточными, обладающими точным знанием того, что правильно, а что – нет, и вступают в конфликты с другими сообществами, которые по тем или иным причинам с ними не согласны. Это может стать существенной проблемой на практике. Например, в некоторых частях сообщества open source есть предубеждение против C++, поскольку это якобы «язык Microsoft» или «им владеет AT&T» (а это не так), тогда как некоторые крупные фирмы видят проблему C++ в том, что *они* не его владельцы.

Действительно важной проблемой является то, что ряд сообществ навязывает ограниченную и местническую точку зрения на то, «что такое программирование» и «что нужно на самом деле»: «Если бы каждый делал все, как надо, никаких проблем бы не было». Задача в том, чтобы уравновесить разные взгляды и создать более крупное и разностороннее сообщество. По мере того как люди растут и сталкиваются с новыми задачами, универсальность и гибкость языка приобретают большее значение, чем нахождение оптимального решения для узкого круга задач.

Если вернуться к техническим вопросам, то я по-прежнему считаю, что гибкая, расширяемая и универсальная система статических типов замечательна. Моя оценка опыта применения C++ подкрепляет эту точку зрения. Мне также очень нравятся настоящие локальные переменные

пользовательских типов: методы C++ для работы с общими ресурсами, основанные на областях видимости переменных, оказались очень эффективными, с чем их ни сравнивай. Конструкторы и деструкторы, часто используемые совместно в RAII, позволяют писать очень красивый и эффективный код.

Преподавание

Вы покинули промышленность, перейдя в академическую сферу. Почему?

Бьерн: На самом деле, я не вполне порвал с промышленностью, поскольку сохраняю связь с AT&T Labs как сотрудник AT&T, ежегодно проводя много времени с людьми, работающими на производстве. Я считаю важным для себя поддерживать такую связь, потому что благодаря этому моя работа не отрывается от действительности.

Пять лет назад я стал профессором Техасского университета A&M, поскольку (после почти 25 лет в «лабораториях») ощутил потребность в переменах и поскольку мне казалось, что я могу сделать что-то полезное в области образования. Я также лелеял несколько идеалистические надежды, что после многих лет сугубо практических исследований и разработок смогу заняться более фундаментальными вещами.

Значительная часть исследований в компьютерных науках либо слишком далека от повседневных задач (даже от предполагаемых в будущем), либо углубилась в повседневность настолько, что мало отличается от простой передачи технологий. Я ничего не имею против передачи технологий (мы очень нуждаемся в этом), но должна существовать сильная обратная связь между промышленным производством и передовыми исследованиями. Краткосрочное планирование распространенное на производстве, а в академической среде – требования наличия публикаций и стремление получить штатную должность, отвлекают внимание и усилия от решения ряда очень важных проблем.

Что вы выяснили о преподавании программирования начинающим за годы своей университетской работы?

Бьерн: Самый конкретный результат моей работы в университете (помимо обязательных научных статей) – это новый учебник для тех, кто никогда раньше не занимался программированием: «Programming: Principles and Practice Using C++» (Программирование: принципы и практика применения C++), изд. Addison-Wesley.

Это моя первая книга для начинающих. До прихода в академическую среду я просто не был знаком со столькими начинающими, чтобы пи-

сать для них такую книгу. Однако я ощущал, что очень многие разработчики программного обеспечения весьма плохо подготовлены для своей работы в промышленности и других местах. Сейчас я обучил (или помог обучить) программированию более 1200 новичков и более уверен в том, что мои идеи в этой области могут получить распространение.

Книга для новичка должна решать несколько задач. Прежде всего, она должна заложить хороший фундамент для последующего обучения (если повезет, это станет началом жизненной карьеры), снабжая некоторыми практическими навыками. Кроме того, программирование – и разработка программного обеспечения в целом – не является ни чисто теоретическим мастерством, ни занятием, которое может быть успешным без знания некоторых фундаментальных идей. К сожалению, очень часто в преподавании не удается соблюсти правильный баланс между теорией/принципами и практицизмом/технологией. В результате встречаются люди, по существу презирающие программирование («чистое кодирование») и считающие, что программы можно разрабатывать исходя из основных принципов и не обладая практическими навыками. Есть и те, кто, напротив, убеждены в том, что все дело в «хорошем коде», который пишется путем редких обращений к справочникам в Интернете и многочисленных операций копирования/вставки. Мне встречались программисты, считавшие K&R[1] «слишком сложной и теоретической» книгой. По моему мнению, обе позиции, уходя в крайности, приводят к появлению плохо организованного, неэффективного и недоступного для сопровождения кода, даже если его вообще удастся заставить как-то работать.

Что вы думаете о примерах кода в учебниках? Нужна ли в них проверка ошибок/исключений? Должны ли они представлять собой законченные программы, которые можно скомпилировать и запустить?

Бьерн: Я предпочитаю такие примеры, в которых на возможно меньшем количестве строк иллюстрируется идея. Такие отрывки часто оказываются неполными, но я утверждаю, что мои примеры скомпилируются и будут работать, если их включить внутрь вспомогательного кода. По существу, мой стиль представления кода основан на K&R. В моей новой книге все примеры кода будут доступны в компилируемом виде. В тексте у меня чередуются небольшие фрагменты с пояснениями к ним и более длинные и законченные куски кода. В существенных местах я привожу пример в обоих этих видах, чтобы читатель мог взглянуть на важные операторы с двух точек зрения.

---

[1] K&R – книга Б. Кернигана и Д. Ритчи «The C Programming Language» – классический учебник по языку (Брайан Керниган, Деннис Ритчи «Язык программирования Си». – Пер. с англ. – Невский Диалект, 2000). – *Прим. ред.*

Некоторые примеры должны быть дополнены проверкой ошибок, и все они должны отражать конструкции, допускающие проверку. Помимо того, что об ошибках и их обработке говорится во многих местах книги, есть отдельные главы, посвященные обработке ошибок и тестированию. Я предпочитаю примеры, взятые из реально работающих программ. Не люблю замысловатые искусственные примеры вроде древовидной структуры наследования с животными или глупых математических паззлов. Пожалуй, сделаю на своей книге пометку: «В приведенных в этой книге примерах не пострадал ни один милый пушистый зверек».

2

Python

Python – это современный универсальный язык высокого уровня, который Гвидо ван Россум (Guido van Rossum) создал на основе опыта работы с языком программирования ABC. Философия Python отличается прагматизмом: пользователи Python часто говорят о его «дзене», стремясь выбрать для решения любой задачи один, самый очевидный путь. Существуют порты Python для виртуальных машин, например для CLR Microsoft и JVM, но главной реализацией служит CPython, над которым продолжают трудиться ван Россум и другие добровольцы, недавно выпустившие Python 3.0 – результат переработки некоторых разделов и базовых библиотек языка, не совместимый с прежними версиями.

Как это делается в Python

Каковы различия между разработкой языка программирования и разработкой «совместного» программного проекта?

Гвидо ван Россум: По сравнению с обычными программными проектами основными пользователями чаще оказываются сами программисты. В силу этого языковый проект характеризуется высоким уровнем «мета»-содержания. Если взглянуть на дерево зависимостей программного проекта, то языки программирования оказываются в самом его низу: все остальное зависит от одного или нескольких языков. По этой причине модифицировать язык трудно: несовместимые изменения будут иметь такие широкие последствия, что осуществлять их просто нереально. Иными словами, все ошибки фиксируются необратимо. Ярчайшим примером может служить C++, обремененный требованиями совместимости, в соответствии с которыми должен работать код, написанный лет 20 назад.

Как отлаживать язык?

Гвидо: Да никак. Разработка языка – это та область, в которой применять методологии ускоренной разработки просто не имеет смысла: пока язык не станет стабильным, у него будет мало пользователей, а ошибки в определении языка выявляются, когда у него столько пользователей, что слишком поздно вносить изменения.

Конечно, *реализацию* часто можно отлаживать как обычную программу, но разработка собственно языка требует тщательного предварительного проектирования, потому что ошибки конструкции обходятся слишком дорого.

Как вы определяете, что делать с новой функцией языка: поместить ее в библиотеку расширения или ввести ее поддержку в базовый язык?

Гвидо: Исторически, я научился неплохо справляться с этим вопросом. Я довольно рано заметил: все хотят, чтобы нравящаяся им возможность была добавлена в язык. Но большинство относительно слабо разбирается в разработке языков. Постоянно кто-нибудь предлагает: «давайте добавим в язык вот это» или «пусть у нас будет оператор для некоторого действия X». Во многих случаях находится ответ вроде «вы и так уже можете делать X или почти X с помощью двух-трех строк кода, и это совсем не сложно». Можно применить словарь, или сочетание списка, кортежа и регулярного выражения, либо написать маленький метакласс – что-то в этом роде. Не исключено, что я позаимствовал такой способ ответа у Линуса; кажется, у него аналогичная философия.

Ответить людям, что они уже могут что-то делать, и объяснить как – это первая линия обороны. Дальше можно сказать, что их предложение

полезно, и мы (или они сами) могли бы написать модуль или класс, инкапсулирующий данный вид абстракции. На следующем рубеже обороны мы говорим: отлично, предложение интересно и полезно, поэтому мы включим его в стандартную библиотеку, но это будет чистый код Python. Наконец, если что-то трудно реализовать на чистом Python, мы предлагаем сделать из этого Си-расширение и можем рассказать, как его создать. Расширения на Си – это последняя линия обороны перед тем, как признать: да, это очень полезно, и имеющимися средствами неосуществимо, поэтому мы внесем изменения в язык.

Есть и другие критерии определения целесообразности добавления чего-нибудь к языку или к библиотеке, потому что это связано с семантикой пространств имен или подобными вещами, и тогда не остается ничего иного, как изменить язык. С другой стороны, механизм расширений обладает такой мощью, что можно очень многое сделать с помощью Си-кода, расширяющего библиотеку, даже добавляющего новые встроенные функции, без фактической модификации языка. Синтаксический анализатор не меняется. Дерево синтаксического разбора не меняется. Документация по языку не меняется. Все инструменты сохраняют работоспособность, и, тем не менее, в системе появляется новая функциональность.

Наверное, есть интересные для вас функции, которые можно было реализовать в Python, только изменив язык, и потому вы их отвергли. По какому критерию вы определяете, что соответствует духу Python, а что нет?

Гвидо: Здесь гораздо сложнее. Вероятно, в большинстве случаев это определяется субъективными ощущениями. Люди часто употребляют слово Pythonic, оценивая нечто как «питоническое» или наоборот, но никто не смог дать четкого определения этого понятия.

У вас есть «Дзен Питона», а что еще?

Гвидо: Тут не обойтись без долгих объяснений, как с любой истинно священной книгой. Когда есть предложение, я могу определить, хорошее оно или плохое, но очень трудно написать свод правил, пользуясь которым всякий сможет отличить хорошую предлагаемую модификацию языка от плохой.

Похоже, это в значительной мере дело вкуса.

Гвидо: Что ж, нужно сразу отвечать всем «нет» и смотреть, будут ли они продолжать настаивать и не найдут ли способ удовлетворить свое желание, не изменяя язык. Примечательно, что это почти всегда срабатывает. Собственно, это принцип «не нужно менять язык» в действии.

Люди найдут способ сделать то, что им нужно, и без изменений в языке. Кроме того, часто дело связано с конкретными случаями в различ-

ных областях, где нет ничего специфичного для данной области. Если какая-то функция очень хороша для веб-приложений, это не значит, что ее стоит добавить в язык. Если же нечто позволяет писать более короткие функции или классы, которые легче сопровождать, то такую вещь, возможно, стоит добавить в язык. Она действительно должна быть независимой от конкретных предметных областей и делать код проще или красивее.

Если вы изменяете язык, это затрагивает всех. Нельзя запрятать функцию так глубоко, чтобы большинство о ней не знало. Рано или поздно человек натыкается на чей-то код, где она используется, или сталкивается с темным случаем, где придется разбираться, что это за штука, потому что код работает не так, как предполагалось.

Представления о красоте часто тоже индивидуальны. В одном из списков рассылки Python недавно была дискуссия, участники которой страстно доказывали, что «доллар» вместо self-точки гораздо красивее. Видимо, в определение красоты здесь входило количество нажатий на клавиши.

Это аргумент в пользу экономии, но сильно связанный с личными вкусами.

Гвидо: Красота, простота, обобщенность – все это в значительной мере зависит от личного вкуса, потому что если я где-то вижу для них широкую область, кому-то она может показаться недостаточной, и наоборот.

Как возник процесс «Предложений по развитию Питона» (Python Enhancement Proposal, PEP)?

Гвидо: С исторической точки зрения это очень интересно. Пожалуй, зачинателем и поборником этого был Барри Уорсоу (Barry Warsaw), один из основных разработчиков. Мы начали работать вместе в 1995 году, и где-то около 2000 года он предложил сделать процедуру модификации языка более формальной.

Я с такими вещами запаздываю. То есть потребность в почтовом списке рассылки обнаружил не я. И не я заметил, что почтовый список стал неудобным и нам нужна телеконференция. Веб-сайт – тоже не моя идея. И не я задумался о процедуре обсуждения и предложения модификаций языка, помогающей избежать случайных ошибок, когда предложения поспешно принимаются без анализа возможных последствий.

В период 1995–2000 годов я, Барри и несколько других основных разработчиков – какое-то время Фред Дрейк (Fred Drake), Кен Манхаймер (Ken Manheimer) – работали в CNRI, и в числе прочего CNRI организовала конференции IETF (специальной комиссии интернет-разработок). В CNRI было это маленькое подразделение, в конечном счете отколов-

шееся, которое занималось организацией конференций, и единственным клиентом его была IETF. Позднее они также некоторое время организовывали конференции Python. По этой причине было достаточно просто попасть на совещания IETF, даже если они проводились в другом месте. Я явно проникся духом процесса IETF с ее RFC, совещаниями групп и этапами, точно так же, как и Барри. Когда он предложил организовать нечто подобное для Python, это не вызвало возражений. Мы решили, что не станем действовать столь тяжеловесно, как IETF при разработке RFC в то время, поскольку стандарты Интернета – по крайней мере некоторая их часть – влияют на гораздо больший круг отраслей, людей и программ, чем модификации Python, но мы очевидным образом взяли эту модель за основу. У Барри есть талант придумывать хорошие названия, поэтому я не сомневаюсь, что PEP[1] было его идеей.

В то время мы были одним из первых проектов open source, обладавших чем-то подобным, и наш опыт довольно широко перенимался. Сообщество Tcl/Tk по существу поменяло название и воспользовалось точно такими же основными документами и процедурой, аналогичным образом поступили и другие проекты.

По вашему мнению, некоторая доля формализма действительно помогает кристаллизации проектных решений относительно развития Python?

Гвидо: Я думаю, что формализм стал необходим, потому что сообщество росло и я не всегда мог оценить действительную пользу каждого предложения. Мне действительно помогло то, что другие люди обсуждали разные детали, после чего приходили к довольно четким выводам.

Придя к единому мнению по поводу некоторого конкретного оформившегося набора перспектив и предложений, они все же допускают ваше вмешательство?

Гвидо: Да. Часто случается так, что в ответ на PEP я сначала поднимаю большой палец, в том смысле, что хочу сказать: здесь у нас проблема и хорошо, чтобы кто-то нашел для нее правильное решение. Часто в результате возникает несколько ясных предложений, как решить эту проблему, и остается ряд нерешенных вопросов. Иногда я «закрываю» эти вопросы, опираясь на интуицию. Я активно участвую в процессе PEP, если это касается области, которая меня волнует. Например, если речь идет о добавлении нового оператора цикла, я бы не хотел, чтобы его разрабатывали другие. Но иногда я несколько дистанцируюсь от таких вопросов, как, скажем, API баз данных.

---

[1] Pep (англ.) – бодрость духа, энергия. – *Прим. пер.*

Когда возникает необходимость в новой основной версии?

Гвидо: Смотря что считать «основным». В Python мы обычно считаем «основными» выпуски типа 2.5 и 2.6, которые появляются с промежутком 18–24 месяца. Только в этих случаях можно вводить новые возможности языка. Когда-то давно релизы делались по прихоти разработчиков (моей в том числе). Однако в начале этого десятилетия пользователям потребовалось более предсказуемое поведение: они были против того, чтобы новые функции появлялись в «дополнительных» версиях (например, чтобы в 1.5.2 появлялись значительные изменения в сравнении с 1.5.1), и хотели поддержки для основных версий в течение гарантированного минимального срока (18 месяцев). В результате теперь основные версии выпускаются более или менее по графику: мы заранее устанавливаем контрольные точки, ведущие к выпуску основной версии, например даты выпуска альфа- и бета-версий, а также кандидата на основную версию, учитывая такие обстоятельства, как, скажем, занятость руководителя выпуска, и настаиваем, чтобы разработчики осуществляли свои модификации задолго до даты окончательного выпуска.

Выбор функций для нового выпуска обычно утверждается основными разработчиками после достаточно долгого порой обсуждения их достоинств и точных спецификаций. Это и есть процесс PEP (предложений по развитию Питона) – основанной на документации процедуры, имеющей сходство с процедурой RFC IETF или процедурой JSR для Java, но не такой формальной, потому что наше сообщество разработчиков гораздо меньше. Если долго не удается устранить разногласия (по поводу достоинств функции или конкретных деталей), я могу выступить с решающим голосом. Мое решение основывается скорее на интуиции, поскольку к этому времени разумные аргументы уже кончаются.

Наибольшие споры обычно разворачиваются вокруг тех функций, которые видны пользователю; расширение библиотек обычно проходит легко (поскольку заметно только интересующимся пользователям), а внутренние исправления не считаются «фичами», хотя на них налагаются довольно строгие ограничения обратной совместимости на уровне API Си.

Так как голос разработчиков обычно слышнее прочих, я не могу сказать, кем предлагаются функции – пользователями или разработчиками; обычно разработчики предлагают новые функции, потребность в которых выясняется в процессе общения со знакомыми им пользователями. Если новую функцию предлагает пользователь, его предложение редко принимается, потому что без досконального знания реализации (и конструкции, и реализации языка в целом) практически невозможно корректно предложить новую функцию. Мы советуем пользователям

рассказывать о возникающих у них проблемах, не предлагая их конкретного решения, после чего разработчики предлагают решения и обсуждают вместе с пользователями достоинства различных вариантов.

Существует также понятие радикально новой версии, вроде 3.0. Исторически версия 1.0 была эволюционно близкой к 0.9, а 2.0 тоже относительно мало отличалась от 1.6. В настоящее время, когда круг пользователей значительно расширился, такие версии действительно выпускаются редко, и только в них допускается несовместимость в прежними версиями. Основные версии поддерживают совместимость с предшествующими основными версиями посредством специфического механизма объявления устаревших функций, которые планируется изъять.

Как вы пришли к числам произвольной точности (со всеми их преимуществами), отказавшись от старого (весьма распространенного) подхода, когда все определяет аппаратная часть?

Гвидо: Я взял эту идею у предшественника Питона, ABC. В ABC используются рациональные числа произвольной точности, но мне идея рациональных чисел не очень нравилась, поэтому я выбрал целые. Для действительных чисел в Python используется стандартное представление с плавающей точкой, поддерживаемое аппаратурой (что делалось и в ABC с некоторыми ухищрениями).

Первоначально в Python было два типа целых: обычные 32-разрядные (`int`) и произвольной точности (`long`). Так поступают во многих языках, но поддержка произвольной точности возлагается на библиотеку, такую как Bignum в Java и Perl или GNU MP для Си. В Python оба вида почти всегда соседствовали между собой в базовом языке, и пользователи могли выбрать тип `long`, приписав к числу букву `L`. Со временем было решено, что это неудобно, и в Python 2.2 мы ввели автоматическое преобразование к `long`, если математически правильный результат операции над `int` нельзя было представить как `int` (например, `2**100`).

Прежде в таких случаях генерировалось исключение `OverflowError`. Первое время результат молча усекался, но я стал генерировать исключение еще до того, как открыл язык для общего пользования. В начале 1990 года я потратил день на отладку коротенькой демо-программы, реализующей алгоритм с неочевидным использованием больших чисел. Такие отладочные упражнения полезны.

И все же иногда эти два числовых типа вели себя несколько по-разному. Например, при выводе `int` в шестнадцатеричном или восьмеричном формате получался результат без знака (скажем, –1 выводилось как `FFFFFFFF`), а числа типа `long` давали при этом результат со знаком (–1, в данном случае). В Python 3.0 мы сделали радикальный шаг, оставив

только один целочисленный тип; мы назвали его `int`, но реализация в основном совпадает с прежним типом `long`.

Почему вы считаете это радикальным шагом?

Гвидо: В основном потому, что это крупный отход от существующей сейчас в Python практики. Эта тема широко обсуждалась, предлагались различные альтернативы, в которых внутренне использовались два или более представлений, полностью или почти полностью скрытых от пользователей (но не от разработчиков Си-расширений). Возможно, это повысило бы производительность, но объем работы и так был велик, а чтобы правильно реализовать два внутренних представления, потребовалось бы еще больше труда, и создать интерфейс к ним из кода Си было бы еще труднее. Мы надеемся, что потеря производительности незначительна и ее можно улучшить другими средствами, например кэшированием.

Как возник философский принцип «должен существовать один — и, желательно, только один — очевидный способ сделать это»?

Гвидо: Вероятно, поначалу он был подсознательным. Когда Тим Питерс писал «Дзен Питона» (вы его цитировали), он сформулировал ряд правил, которыми я неявно руководствовался. Данное конкретное правило (которое часто нарушается – с моего согласия) происходит из общего стремления к красоте в математике и компьютерных науках. Авторы ABC тоже применяли его, предпочитая небольшое число ортогональных типов или понятий. Идея ортогональности взята непосредственно из математики, где она означает само *определение* того, что есть только один способ (или один правильный способ) выразить что-либо. Например, координаты x, y, z любой точки в 3-мерном пространстве однозначно определены, если вы выбрали начало координат и три базисных вектора.

Я также полагаю, что облегчаю жизнь большинству пользователей тем, что не требую от них выбирать между схожими альтернативами. Это представляет собой противоположность, например, Java, где если требуется структура данных типа списка, то стандартная библиотека предложит вам несколько вариантов (связанный список, массив-список и другие), или Си, где вам придется решать, как реализовать собственный тип списка данных.

Как вы относитесь к статической и динамической типизации?

Гвидо: Жаль, что я не могу сказать что-нибудь вроде «статическая типизация – это плохо, а динамическая – хорошо»: не все так просто. Есть разные подходы к динамической типизации, от Лиспа до Python, и разные подходы к статической типизации, от C++ до Haskell. Такие языки, как C++ и Java, наверно, создают статической типизации дурную сла-

ву, потому что требуют от вас многократно сообщать компилятору одно и то же. Зато в таких языках, как Haskell и ML, используется вывод типа, а это совсем другая вещь, обладающая некоторыми достоинствами динамической типизации, такими как более краткое выражение идей в коде. Тем не менее функциональную парадигму трудно использовать самостоятельно: такие вещи, как ввод/вывод и интерактивный GUI, плохо в нее укладываются и обычно организуются с помощью моста к более традиционным языкам, например Си.

В некоторых ситуациях многословность Java считается достоинством: она позволяет создавать мощные средства анализа кода, способные отвечать на такие вопросы, как «где изменилась эта переменная?» или «кто вызывает этот метод?». В динамических языках отвечать на такие вопросы труднее, потому что часто бывает трудно узнать тип аргумента метода, не проанализировав все пути по всему коду. Не знаю точно, как такие инструменты поддерживаются в языках типа Haskell, – вполне возможно, приходится пользоваться теми же способами, что и в динамических языках, поскольку этим и занимается вывод типа, насколько я могу судить!

Не движемся ли мы в сторону гибридной типизации?

Гвидо: Думаю, гибридизация в том или ином виде заслуживает пристального внимания. Я заметил, что в крупных системах, написанных на языках со статической типизацией, обычно есть существенная часть, в которой типизация по существу динамическая. Например, наборы компонентов для GUI и API баз данных для Java часто на каждом шагу сражаются со статической типизацией, стараясь по возможности отложить проверку корректности до времени исполнения.

Гибридный язык, сочетающий функциональные и динамические аспекты, может оказаться весьма интересен. Нужно добавить, что несмотря на поддержку Python некоторых функциональных средств, таких как `map( )` и `lambda`, в Python отсутствует подмножество функционального языка: в нем нет вывода типов и возможностей параллелизма.

Почему вы решили поддерживать несколько парадигм?

Гвидо: На самом деле, это не так: Python в некоторой мере поддерживает процедурное программирование и объектно-ориентированное. Различия между тем и другим не столь велики, и процедурный стиль Python сохраняет сильное влияние объектов (поскольку все базовые типы данных являются объектами). Python лишь в очень малой степени поддерживает функциональное программирование, но при этом не похож ни на один из реальных функциональных языков – и никогда не будет похож. Для функциональных языков характерно, чтобы как можно больше работы делалось на этапе компиляции – «функциональ-

ность» означает, что компилятор может провести оптимизацию с уверенной гарантией отсутствия побочных эффектов, если только о них не сказано явно. Для Python характерно, что компилятор может быть самым простым и тупым, и официальная семантика этапа исполнения препятствует таким умствованиям компилятора, как распараллеливание циклов или превращение рекурсии в циклы.

Репутация Python как языка, поддерживающего функциональное программирование, могла возникнуть в результате наличия в нем `lambda`, `map`, `filter` и `reduce`, но на мой взгляд, это всего лишь синтаксические украшения, а не основные конструктивные элементы, как в функциональных языках. Более фундаментальным свойством Python, общим с Лиспом (тоже не функциональный язык!), является представление функций в качестве объектов первого класса, которые можно передавать, как всякие другие объекты. Наряду с вложенными областями видимости и в целом Лисп-образным подходом к состоянию функций, это позволяет легко реализовать идеи, внешне напоминающие идеи функциональных языков, такие как каррирование, отображение и свертка. Примитивные операции, необходимые для *реализации* этих идей, строятся на Python, а в функциональных языках эти идеи *уже* являются примитивными операциями. Написать `reduce()` на Python можно в несколько строк. В функциональных языках это не так.

Создавая этот язык, думали ли вы о том, программистам какого типа он может быть интересен?

Гвидо: Да, но у меня не хватило воображения. Я представлял себе профессиональных программистов в UNIX или UNIX-подобной среде. Руководства для ранних версий Python возвещали что-то вроде «Python закрывает разрыв между Си и программированием оболочки», потому что именно это интересовало меня и моих ближайших коллег. Мне и в голову не приходило, что Python может стать хорошим языком для встраивания в приложения, пока меня не стали спрашивать об этом.

То, что он оказался полезен для обучения началам программирования в школе или колледже, – счастливая случайность, обусловленная многими характеристиками ABC, которые я сохранил: ABC был специально предназначен для обучения программированию непрограммистов.

Как вы сохраняете равновесие между свойствами языка, облегчающими его изучение новичками, и достаточно мощными средствами, которые нужны для эффективной работы опытным программистам? Или это ошибочное противопоставление?

Гвидо: Именно балансирую. Известно, каких опасностей нужно избегать: того, что может быть полезно новичкам, но раздражает специалистов, и того, что необходимо специалистам, но смущает новичков. Но

в промежутке есть достаточно возможностей удовлетворить обе стороны. Еще один путь – дать специалистам возможность делать сложные вещи, незаметную для новичков; например, если язык поддерживает метаклассы, новичкам ни к чему о них знать.

Хороший программист

Как вы определяете, хорош или плох программист?

Гвидо: Чтобы узнать хорошего программиста, нужно время. Выяснить это за один час интервью очень трудно. Но работая вместе с людьми над разными задачами, достаточно ясно видишь, кто чего стоит. Я затрудняюсь сформулировать точные критерии – в целом, хорошие программисты проявляют творческий подход и быстро учатся, начиная вскоре выдавать работоспособный код, не требующий большой доработки перед включением в систему. Надо учесть, что люди могут проявлять способности в разных аспектах программирования: одни сильны в алгоритмах и структурах данных, другие – в крупномасштабной интеграции, или разработке протоколов, или тестировании, или проектировании API, или пользовательских интерфейсах, либо в каких-то других областях программирования.

Как бы вы стали подбирать себе новых программистов?

Гвидо: Судя по моему прошлому опыту интервьюирования, традиционные способы едва ли могут мне подойти: мое умение проводить интервью можно считать нулевым – по любую сторону стола! Думаю, что выбрал бы какую-то форму ученичества, при которой я мог бы работать некоторое время в тесном контакте с людьми и в итоге выяснить их слабые и сильные стороны. Примерно так, как действуют проекты open source.

Есть ли какая-то главная характеристика, на которую нужно обращать внимание при поиске выдающихся Python-программистов?

Гвидо: Вы задаете этот вопрос как типичный менеджер, которому просто нужно нанять группу программистов на Python. Не думаю, что есть простой ответ на этот вопрос, и даже полагаю, что вопрос неверно поставлен. Не нужно нанимать программистов на Python. Нужно нанимать умных, творческих и инициативных людей.

Если посмотреть рекламные объявления о работе для программистов, то почти в каждом из них будет упомянуто «умение работать в команде». Что вы думаете о роли команды в программировании? Можно ли найти применение блестящему программисту, не умеющему работать с другими?

Гвидо: Я согласен с рекламными объявлениями в этом вопросе. Блестящих программистов, не способных к коллективной работе, нельзя принимать на обычные места программистов, иначе это становится катастрофой для всех сторон, а их код оказывается настоящим кошмаром для тех, кому приходится им заниматься после них. Я даже думаю, что программист, неспособный к групповой работе, не может быть блестящим. В наше время можно усвоить принципы работы с другими людьми, и если ты действительно такой умный, то сможешь легко научиться работать в команде: это не так трудно, как эффективно реализовывать быстрое преобразование Фурье, если только захотеть.

Когда вы пишете код на своем языке, есть ли у вас как автора языка преимущества перед другими опытными разработчиками на Python?

Гвидо: Вряд ли – к настоящему времени к языку и ВМ приложило руки столько людей, что иногда я сам с удивлением разбираюсь в некоторых деталях! Если у меня и есть преимущество перед другими разработчиками, то оно скорее связано с тем, что я работаю с этим языком дольше, чем кто-либо другой, а не с тем, что я сам его написал. За это долгое время у меня была возможность подумать, какие операции выполняются быстрее, а какие – медленнее; например, мне лучше, чем большинству, известно, что локальные переменные быстрее глобальных (хотя не я, но другие перегибают палку в их использовании!), или что вызовы функций и методов стоят дорого (дороже, чем в Си или Java), или что самый быстрый тип данных – кортеж.

Когда приходится пользоваться стандартной библиотекой и чем-то еще, я часто замечаю, что у других есть преимущество передо мной. Например, раз в несколько лет я пишу какое-нибудь веб-приложение и всегда обнаруживаю, как изменились технологии по сравнению с прошлым разом, поэтому каждый раз я пишу «первое» веб-приложение с использованием новой среды или метода. И мне до сих пор не удалось серьезно заняться XML на Python.

Для Python характерна краткость. Как это влияет на возможности сопровождения кода?

Гвидо: Я слышал об исследованиях и отдельных свидетельствах, указывающих на то, что количество ошибок на одну строку кода оказывается довольно постоянной величиной, не зависящей от языка программирования. Поэтому язык вроде Python, приложение на котором обычно оказывается гораздо меньше, чем та же функциональность, реализованная на C++ или Java, должен значительно облегчить сопровождение этого приложения. Конечно, из этого должно следовать, что каждый отдельный программист реализует больший объем функциональности. Это отдельная тема, но все равно результат говорит в пользу

Python: более высокая продуктивность каждого программиста позволяет сократить размер команды, а это уменьшает расходы на обмен информацией между программистами, которые, согласно «Мифическому человеко-месяцу» Фредерика Брукса, растут как квадрат численности команды, если мне не изменяет память.

Влияет ли, по вашему мнению, легкость создания прототипа с помощью Python на общий объем работы по созданию законченного приложения?

Гвидо: Я никогда не рассматривал Python как язык для создания прототипов. Не думаю, что языки для создания прототипов и «конечных продуктов» должны сильно отличаться друг от друга. В некоторых случаях лучшим прототипом оказывается небольшой одноразовый код на Си. В других случаях прототип можно создать вообще без «программирования» – например, с помощью электронной таблицы или группы команд `find` и `grep`.

Мои первоначальные намерения в отношении Python заключались просто в желании иметь язык, которым можно пользоваться в тех случаях, когда Си оказывается излишеством, а сценарии командного процессора становятся слишком громоздкими. Такие случаи часто возникают не только при создании прототипов, но и при написании «бизнес-логики» (как это называют сегодня), в которой требования к вычислительным ресурсам невелики, но кода нужно написать много. Я считаю, что обычно код Python пишется не для прототипа, а чтобы решить задачу. В большинстве случаев Python полностью справляется с работой, и получение конечного продукта не требует большой доработки.

Часто бывает, что простое приложение постепенно обрастает новыми функциями, сложность его увеличивается десятикратно, и нет точной границы между прототипом и окончательным приложением. Например, приложение Mondrian для рецензирования кода, которое я начал писать для Google, выросло в объеме раз в десять по сравнению с первой версией и по-прежнему целиком пишется на Python. Конечно, иногда Python в конце концов заменяется более быстрым языком – например, значительная часть ранних версий поискового робота Google была написана на Python, – но это исключение, а не правило.

Как влияет на процесс проектирования то, что Python функционирует в режиме реального времени?

Гвидо: Я часто работаю именно так, и обычно – мне, во всяком случае, – это помогает. В самом деле, я пишу много такого кода, который потом выбрасываю, но его объем гораздо меньше, чем он был бы, пользуйся я любым другим языком, а когда я пишу код (даже не запуская его),

это часто очень помогает разобраться в деталях задачи. Поиск способа реструктурировать код так, чтобы он решал задачу оптимальным образом, часто помогает лучше понять саму задачу. Конечно, это не значит, что можно не рисовать на доске конструкцию, архитектуру, взаимодействие, или не пользоваться другими приемами на начальном этапе проектирования. Весь фокус в том, чтобы выбрать для работы правильный инструмент. Иногда это карандаш и салфетка, а в других случаях – окно Emacs и приглашение оболочки.

Вы считаете, что для Python более подходит восходящая разработка программ?

Гвидо: Я не считаю восходящий и нисходящий стили столь идеологически противоположными, как, скажем, vi и Emacs. В процессе разработки программного обеспечения всегда приходится одну часть времени работать в восходящем стиле, а другую – в нисходящем. Нисходящий стиль может потребоваться, когда вам нужно тщательно изучить задачу и спроектировать решение, прежде чем начинать кодировать, тогда как восходящий стиль может означать, что вы строите новые абстракции поверх существующих, например при создании новых API. Я не хочу этим сказать, что можно начинать писать API, не имея в голове никакого конструктивного решения, но часто новые API логически вытекают из имеющихся API более низкого уровня, а проектирование фактически происходит во время написания кода.

Когда, как вам кажется, программисты на Python могут лучше оценить его динамический характер?

Гвидо: Динамические возможности языка часто бывают наиболее полезны, когда вы исследуете большую задачу или пространство решений и еще не сориентировались: можно проделать ряд экспериментов, каждый раз используя то, что вы выяснили ранее, и при этом объем кода будет не настолько велик, чтобы связать вас с каким-то одним подходом. Здесь вы действительно выигрываете от компактности кода Python: написать 100 строк на Python, чтобы провести эксперимент, после чего начать все заново, это гораздо эффективнее, чем написать 1000 строк для эксперимента на Java и убедиться, что они не решают задачу!

Что предлагает Python программисту с точки зрения безопасности программ?

Гвидо: Это зависит от того, какого рода атаки вас тревожат. Python автоматически выделяет память, поэтому написанным на нем программам не угрожает опасность таких стандартных ошибок кода на Си и C++, как переполнение буфера и использование освобожденной памяти, на чем основано множество атак на программы Microsoft. Конечно,

исполнительная среда Python сама написана на Си, и с годами в ней обнаружились уязвимости, а кроме того, намеренно созданы средства для выхода за рамки исполнительной среды Python, например модуль `ctypes`, позволяющий выполнить произвольный код Си.

Благо или нет динамический характер языка?

Гвидо: Не думаю, что динамический характер может приносить пользу или вред. Легко придумать динамический язык, в котором будет масса слабых мест, или статический, в котором их не будет. Однако наличие исполнительной среды, или «виртуальной машины», как сейчас модно говорить, полезно тем, что ограничивает доступ к настоящей машине. Кстати, это одна из причин, по которым Python стал первым языком, поддерживаемым Google App Engine – проектом, в котором я участвую в настоящее время.

Как программисту на Python проверить и повысить безопасность своего кода?

Гвидо: Думаю, программистам на Python не стоит особенно беспокоиться о защите – во всяком случае, если они не имеют в виду какую-то конкретную модель атаки. Главное требование одинаково для всех языков: с подозрительностью относиться ко всем данным, полученным из ненадежных источников (для веб-сервера таковыми является каждый байт входящего запроса, даже заголовки). Еще нужно следить за регулярными выражениями: легко написать регулярное выражение с экспоненциальным временем выполнения, поэтому веб-приложения, реализующие поиск и позволяющие пользователю вводить регулярные выражения, должны иметь механизм ограничения продолжительности выполнения.

Есть ли какая-нибудь базовая идея (общее правило, точка зрения, склад ума, принцип), которой нужно придерживаться, чтобы стать искусным программистом на Python?

Гвидо: Я бы назвал прагматизм. Если вы чрезмерно озабочены такими теоретическими понятиями, как скрытие данных, контроль доступа, абстракции или спецификации, то не станете настоящим Python-программистом и будете напрасно терять время на борьбу с языком вместо того, чтобы пользоваться им (получая удовольствие); кроме того, в этом случае ваше применение языка, скорее всего, окажется неэффективным. Python подойдет вам, если вы любите немедленно получать вознаграждение, как я сам. Он подойдет вам, если вы любите, скажем, экстремальное программирование или другие методы ускоренной разработки, но даже там я бы порекомендовал проявлять сдержанность.

Что вы имеете в виду, говоря о «борьбе с языком»?

Гвидо: В основном это касается попыток сохранить привычки, хорошо зарекомендовавшие себя при работе с другими языками.

Те, кто недавно перешли на Python и недостаточно привыкли к нему, очень часто предлагают избавиться от явного `self`. Для них это становится наваждением. Одни предлагают изменить язык, другие придумывают сверхсложные метаклассы, позволяющие каким-то образом сделать `self` неявным. Обычно такие вещи крайне неэффективны или не работают в многопоточном окружении или каких-то других крайних случаях, либо их так мучает необходимость вводить эти четыре буквы, что они меняют обозначение с `self` на `s` или `S`. Некоторые все превращают в классы, а все обращения выполняют с помощью методов доступа, что в Python делать не рекомендуется: ваш код просто станет многословным, его будет сложнее отлаживать, и выполняться он будет гораздо медленнее. Знаете выражение «Фортран-код можно написать на любом языке»? Java-код тоже можно написать на любом языке.

Вы потратили много времени, пытаясь создать (желательно) единственный очевидный способ сделать что-либо. Похоже, вы придерживаетесь мнения, что делая что-либо этим — «питоновским» — способом, действительно можно воспользоваться преимуществами Python.

Гвидо: Не думаю, что я в действительности тратил много времени, чтобы гарантировать наличие только одного способа. «Дзен Питона» появился значительно позже самого языка, и наиболее отличительные черты Python существовали задолго до того, как Тим Петерс изобразил их в поэтическом виде. Сомневаюсь, что он рассчитывал на такое широкое распространение и успех своего творения.

Такие фразы запоминаются.

Гвидо: Тим владеет словом. «Есть только один способ сделать это» – как правило, чистая ложь. Есть много способов формирования структур данных. Можно воспользоваться кортежами или списками. Во многих случаях не важно, что выбрать: кортеж, список или словарь. Обычно, если присмотреться, то одно решение объективно оказывается лучше других, потому что оно срабатывает и в ряде других ситуаций, и есть пара ситуаций, когда списки работают гораздо эффективнее кортежей, если их нужно постоянно расширять.

На самом деле, здесь прослеживается связь с изначальной философией ABC, проповедовавшей предельную экономию средств. Фактически, философия ABC была общей с Алголом 68 – окончательно вымершим, но оказавшим значительное влияние языком. Разумеется, там, где я оказался в 1980-х, он пользовался влиянием, потому что Адриан ван

Вейнгаарден (Adriaan van Wijngaarden) был крупной фигурой в Алголе 68. Он еще преподавал, когда я учился в колледже. В течение одного или двух семестров я слышал подробности из истории Алгола 68, когда у него было желание их рассказывать. Он был директором CWI.[1] Когда я туда поступил, там был другой директор.

Там было много людей, тесно связанных с Алголом 68. Кажется, Ламберт Меертенс (Lambert Meertens), главный автор ABC, был также одним из главных авторов отчета Алгола 68, то есть, возможно, он занимался и типографским набором, но не исключено, что ему приходилось довольно много размышлять и проверять. На него явно оказала влияние философия Алгола 68, согласно которой конструкции должны быть таковы, чтобы их можно было сочетать разными способами, создавая разнообразные структуры данных или структурируя программу.

Явно под этим влиянием было сказано: «У нас есть списки или массивы, и они могут содержать в себе любые другие вещи. Они могут содержать строки или числа, но могут также содержать другие массивы и кортежи других вещей. Все это можно объединять вместе». И тут оказывается, что не требуется особое понятие многомерного массива, потому что массив массивов обеспечивает любую размерность. Эта философия, когда берется несколько основных вещей, охватывающих разные направления, и разрешается их сочетать, играет большую роль в ABC. Я позаимствовал это все, почти не задумываясь.

Хотя Python пытается создать впечатление, что можно объединять вещи очень гибкими способами, не допуская лишь вложенность операторов в выражения, в синтаксисе достаточно особых случаев, когда запятая может быть разделителем для параметров, или указывать на элементы списка, или участвовать в описании неявного кортежа.

Есть целый ряд синтаксических случаев, когда не разрешается использовать определенные операторы, потому что они вступают в конфликт с каким-нибудь окружающим синтаксисом. На самом деле, это не проблема, потому что всегда можно заключить что-то в пару круглых скобок, чтобы оно заработало. Из-за этого синтаксис изрядно расширился, по крайней мере с точки зрения автора синтаксического анализатора. Такие вещи, как списочные выражения (list comprehensions) и генераторные выражения (generator expressions), синтаксически еще не полностью унифицированы. В Python 3000, полагаю, это сделано. Остались тонкие семантические различия, но синтаксис, по крайней мере, одинаков.

[1] National Research Institute for Mathematics and Computer Science.

Реализации Python

Упростился ли синтаксический анализатор в Python 3000?

Гвидо: Едва ли. Он не усложнился, но практически не стал и проще.

«Не усложнился» — уже, видимо, достижение.

Гвидо: Да.

Какой смысл в «самом простом и тупом компиляторе»?

Гвидо: Первоначально это было сугубо практическим соображением, поскольку у меня не было подготовки в области генерации кода. Я был один, и чтобы продолжить делать что-то интересное с языком, мне нужен был генератор байт-кода.

Я и сейчас считаю, что иметь простейший анализатор полезно; в конце концов, эта штука должна всего лишь превращать текст в дерево, которое представляет структуру программы. Если синтаксис столь двусмыслен, что для его разбора нужны сложные технологические ухищрения, то можно предположить, что он и человека часто будет ставить в тупик. Кроме того, для того чтобы написать новый анализатор, нужно много труда.

Python чрезвычайно легко анализируется, во всяком случае, на синтаксическом уровне. На лексическом уровне анализ сравнительно сложен из-за необходимости читать отступы в лексическом анализаторе, у которого маленький стек, и это хороший контрпример для сторонников разделения лексического и грамматического анализа. Тем не менее это правильное решение. Занятно, что мне нравятся автоматически генерируемые парсеры, но в автоматическую генерацию лексического анализатора я не очень верю. У Python всегда был сделанный вручную сканер и автоматизированный парсер.

Для Python написано много парсеров. Даже у переносов Python на другие виртуальные машины, такие как Jython, IronPython или PyPy, есть свои парсеры, и ничего удивительного, потому что анализатор – не самая сложная часть проекта, так как структура языка позволяет легко анализировать его с помощью самого элементарного предсказывающего рекурсивного нисходящего парсера.

Анализаторы медленно работают, если есть неоднозначности, разрешить которые можно, только просмотрев программу до конца. В естественных языках много примеров невозможности разобрать предложение, не прочтя его до конца, и произвольной вложенности внутри предложения. Или встречаются предложения, понять которые можно, только если вы знаете, о ком идет речь, но это уже совсем другой

случай. Для разбора языков программирования меня устраивает мой 1-предсказывающий алгоритм.

Видимо, в Python никогда не будет макросов, потому что они потребовали бы еще одного прохода анализатора!

Гвидо: Есть способы встроить в анализатор макросы, но я совсем не уверен, что макросы решили бы какую-то важную проблему в Python. С другой стороны, поскольку для этого языка разбор происходит так просто, то если будет предложен какой-то разумный набор макросов, вписывающихся в синтаксис языка, возможно, будет очень легко реализовать микровычисление в виде операций над деревом синтаксического разбора. Просто это не та область, которой я особенно интересуюсь.

Почему вы выбрали строгое форматирование исходного кода?

Гвидо: Группирование с помощью отступов – не открытие Python: я взял его из ABC, а еще оно было в более старом языке, occam. Не знаю, возникла ли эта идея у самих авторов ABC или они позаимствовали ее у occam, а может быть, есть какой-то общий источник. Возможно, она принадлежит Дональду Кнуту, который выдвинул такое предложение еще в 1974 году.

Конечно, можно было не перенимать эту идею у ABC, как я и сделал в других областях (например, не стал копировать принятое в ABC выделение прописными буквами ключевых слов языка и имен процедур), но мне очень полюбилась эта особенность, когда я работал с ABC, потому что она позволяла избежать весьма частых в то время бессмысленных споров между пользователями Си по поводу того, где ставить фигурные скобки. Я также хорошо понимал, что в коде, удобном для чтения, все равно специально делаются отступы для группировки, а также сталкивался с трудно обнаруживаемыми ошибками в коде, когда отступы не соответствуют группированию с помощью фигурных скобок, а программист и рецензенты, не зная этого, не замечают ошибку. Здесь я также извлек полезный урок из одного случая долгой отладки кода.

Строгое форматирование должно способствовать более легкому пониманию кода и, вероятно, сокращать различия для программистов, структурирующих свой код по-разному, но не заставляет ли это человека приспосабливаться к машине, а не наоборот?

Гвидо: Совсем напротив: человеку это нужно больше, чем машине, – см. приведенный пример. Достоинства этого метода лучше видны, когда приходится сопровождать код, написанный другим программистом.

Вначале это может отталкивать новых пользователей, но в последнее время я редко слышу такие жалобы; возможно, те, кто преподает Python, знают о таком явлении и научились успешно с ним бороться.

Поговорим о реализациях Python. Есть четыре или пять крупных реализаций, в том числе Stackless и PyPy.

Гвидо: Формально Stackless не является отдельной реализацией. Stackless часто считают отдельной реализацией, потому что это ответвление Python, в котором довольно малая часть виртуальной машины использует иной подход.

Собственно, диспетчеризацию байт-кода?

Гвидо: Диспетчеризация байт-кода по большей части идентична. Думаю, байт-код одинаков, и уж точно одинаковы все объекты. Разница в том, что происходит, когда одна процедура Python вызывает другую: это осуществляется путем манипуляций над объектами, когда в стек помещаются кадры, а выполнение продолжается в том же самом фрагменте кода Си. При этом в CPython в этот момент вызывается Си-функция, которая в итоге вызывает новый экземпляр виртуальной машины. На самом деле, это не вся виртуальная машина, а цикл, который интерпретирует байт-код. В Stackless в стеке C есть только один такой цикл. В обычном CPython таких циклов в стеке C может быть много. Вот и вся разница.

PyPy, IronPython, Jython – это отдельные реализации. Не знаю, есть ли какая-нибудь система для трансляции в JavaScript, но не удивлюсь, если кто-то далеко продвинулся в ее создании. Слышал об экспериментах по переводу на OCaml, Лисп и на что-то еще. Когда-то была одна штука, которая транслировала в код Си. Над ней работали в конце 1990-х Марк Хаммонд (Mark Hammond) и Грег Штайн (Greg Stein), но они выяснили, что ускорение, которого они могут добиться, очень и очень незначительно. В лучшем случае код выполнялся вдвое быстрее, но он был таким объемистым, что получались гигантские двоичные модули, и это стало проблемой.

Долгий запуск был проблемой.

Гвидо: Думаю, что разработчики PyPy идут правильным путем.

Похоже, в целом вы положительно относитесь к этим реализациям.

Гвидо: Я всегда положительно относился к альтернативным реализациям. Когда Джим Хугунин (Jim Hugunin) предложил свою более или менее законченную интерпретацию JPython, я был в восторге. Можно считать, это подтверждает, что язык спроектирован правильно. Кроме того, это значит, что люди смогут пользоваться своим любимым языком на той платформе, где были его лишены. Для этого еще предстоит поработать, но мне явно стало легче определить, какие возможности языка по-настоящему дороги мне, а какие составляют особенность конкретной реализации и не вызывают у меня возражений из-за того, что

кто-то реализовал их иначе. К этому мы и пришли на щекотливой теме сборки мусора.

Она всегда была щекотливой.

Гвидо: Но это неизбежность. Трудно поверить, что мы так долго смогли обходиться простым подсчетом ссылок, не имея возможности прервать цикл. Я всегда рассматривал подсчет ссылок как способ сборки мусора, и не самый плохой. Священная война между сторонниками подсчета ссылок и сборки мусора всегда казалась мне изрядной глупостью.

Если вернуться к реализациям, то я хочу сказать, что Python интересен наличием неплохой спецификации. Во всяком случае, по сравнению с такими языками, как Tcl, Ruby и Perl 5. Чем это вызвано — вашим желанием стандартизировать язык и его поведение, наличием нескольких реализаций или чем-то еще?

Гвидо: Пожалуй, это побочный эффект существующей в сообществе процедуры, связанной с PEP, и наличием нескольких реализаций. Впервые составляя документацию, я энергично принялся за справочник по языку, который хотел сделать такой точной спецификацией, чтобы даже инопланетянин мог по ней правильно реализовать язык. Мне не удалось даже приблизиться к своей цели.

Вероятно, к этой цели ближе всех других языков оказался Алгол 68 со своей сильно математизированной спецификацией. Другие языки, такие как C++ и JavaScript, выехали на силе воли комиссий по стандартизации, особенно в случае C++. Это очень впечатляющий проект. В то же время, чтобы написать спецификацию с такой точностью, нужно столько труда, что моим мечтам получить нечто аналогичное для Python не суждено было осуществиться.

Зато мы хорошо понимаем, как должен работать язык, и у нас достаточно модульных тестов и немало людей, которые могут быстро ответить на вопросы разработчиков других реализаций. Я знаю, например, что разработчики IronPython очень добросовестно стараются прогнать весь комплект тестов Python и в при каждой неудаче стремятся выяснить, отражает ли тест специфическое поведение реализации CPython или им нужно доработать свою реализацию.

Так же поступают в команде PyPy, но они пошли даже дальше. Там есть пара человек потолковее меня, и они предложили граничный случай — вероятно, в результате собственных размышлений над тем, как генерировать код и как анализировать код в среде JIT. Они действительно разработали многочисленные тесты, выявили неоднозначность и поставили ряд вопросов, обнаружив, что существует определенная комбинация условий, о которой никто никогда не задумывался. Это было очень по-

лезно. Когда создается несколько реализаций языка, это чрезвычайно полезно для устранения неоднозначностей в его спецификации.

Может ли случиться так, что CPython перестанет быть главной реализацией?

Гвидо: Трудно сказать. Ведь одни предполагают, что со временем миром станет править .NET, а по мнению других править миром будет JVM. По-моему, все они выдают желаемое за действительное. Но я не знаю, что случится в будущем. Возможно, произойдет какой-то квантовый скачок: даже если компьютеры останутся такими же, как сегодня, какая-то другая платформа может внезапно возобладать над другими, и правила игры изменятся.

Может быть, произойдет отход от архитектуры фон Неймана?

Гвидо: Я даже не задумывался над этим, но такая вероятность тоже есть. Я скорее думал о том, что мобильные телефоны могут стать повсеместно распространенными вычислительными устройствами. Мобильные телефоны по мощности всего лишь на несколько лет отстают от обычных лэптопов, и это означает, что всего через несколько лет мобильные телефоны, несмотря на свои крохотные клавиатуру и экран, будут обладать такой вычислительной мощью, что лэптоп вам больше не понадобится. Вполне может оказаться, что мобильные телефоны, независимо от платформы, будут располагать JVM или другой стандартной средой, где CPython окажется не лучшим подходом, и какие-то другие реализации Python окажутся гораздо более эффективными.

Кроме того, несомненно, возникает вопрос о том, как поступать, когда на чипе окажется 64 ядра, будь то лэптоп или сотовый телефон. Не знаю, действительно ли в результате должна резко измениться парадигма программирования в большинстве наших проектов. Возможно, некоторое применение найдут языки, позволяющие необычайно тонко задать выполнение параллельных процессов, но, как правило, средний программист все равно не умеет писать код, поддерживающий безопасные потоки. Не реалистично полагать, что пришествие многоядерных процессоров будет способствовать его прогрессу. Полагаю, что наличие нескольких ядер несомненно принесет пользу, но они будут использоваться для организации грубого параллелизма, но и это хорошо, потому что из-за огромной разницы в стоимости между удачным и неудачным попаданием в кэш основная память фактически перестала выполнять функцию разделяемой памяти. Процессы должны быть как можно более изолированными друг от друга.

Что делать с параллелизмом? На каком уровне браться, а еще лучше — решать эту проблему?

Гвидо: По моему мнению, и однопоточный код достаточно трудно писать, а многопоточный – гораздо труднее, настолько, что большинство, в том числе я, и не надеются с этим справиться. Поэтому я не верю, что мелкозернистые примитивы для синхронизации и разделяемая память дают нам решение; я бы горячо приветствовал возвращение моды на решения, основанные на передаче сообщений. Почти не сомневаюсь в том, что модификация всех языков программирования путем добавления синхронизирующих конструкций – это плохая идея.

Я также по-прежнему не верю, что изъятие GIL (глобальной блокировки интерпретатора) из CPython принесет пользу. Но я уверен, что часть решения составляет поддержка нескольких процессов (в отличие от потоков), и потому в Python 2.6 и 3.0 будет новый модуль стандартной библиотеки, multiprocessing, API которого похож на модуль поддержки многопоточности и позволяет управлять процессами. В качестве бонуса вы получите даже поддержку процессов, выполняемых на разных узлах!

Приемы и мастерство

Есть ли инструмент или функция, об отсутствии которых вы жалеете, когда пишете программы?

Гвидо: Если бы на компьютере можно было столь же легко делать эскизы, как с помощью карандаша и бумаги, я имел бы возможность создавать больше эскизов во время напряженного проектирования. Боюсь, что придется ждать, пока вместо мыши не станет всюду использоваться перо (или палец), которым можно рисовать на экране. Лично я чувствую себя крайне обделенным при работе с любой графической программой, хотя карандашом и бумагой пользуюсь неплохо – вероятно, это у меня от отца, который был архитектором и постоянно делал наброски, так что и я с детства рисую.

В другой области, вероятно, были бы полезны какие-то средства для исследования больших массивов кода. У программистов на Java теперь есть IDE, позволяющие быстро получить ответы на вопросы вроде «где происходит обращение к этому методу?» или «где присваивается значение этой переменной?». Для больших программ на Python такие средства были бы полезны, но необходимый для них статический анализ проводить труднее, потому что Python – динамический язык.

Как вы тестируете и отлаживаете свой код?

Гвидо: По обстановке. Я много тестирую, когда пишу код, но метод тестирования может отличаться для каждого проекта. Если это чисто алгоритмический код, обычно очень полезно модульное тестирование,

но если это код с сильной интерактивностью или предоставляющий интерфейс к старым API, я часто много тестирую вручную, с помощью буфера командной строки оболочки или перезагрузки страниц в броузере. Вот вам крайний случай: как написать модульный тест для скрипта, единственная задача которого – выключить машину, на которой он выполняется? Можно, конечно, заглушить ту его часть, которая фактически выполняет выключение, но все-таки и ее придется протестировать, иначе как вы узнаете, что ваш скрипт действительно работает?

Также бывает трудно организовать тестирование в разных средах. Buildbot замечателен для больших систем, но затраты на его настройку столь существенны, что для маленьких систем часто дешевле контролировать качество вручную. У меня хорошо развита интуиция при контроле качества, но, к сожалению, трудно объяснить, на чем она основана.

Когда нужно учиться отладке? И как?

Гвидо: Постоянно. Отладкой занимаются всю жизнь. Я только что «отладил» проблему с игрушечным поездом моего шестилетнего сына: в одном месте пути вагоны регулярно сходили с рельсов. Обычно отладка требует спуститься на один-два уровня абстракции, остановиться и внимательно вглядеться, подумать, а иногда и воспользоваться подходящими инструментами.

Не думаю, что есть какой-то один «правильный» способ отладки, которому нужно обучать в определенный момент, даже в таком конкретном случае, как поиск программных ошибок. Спектр возможных причин возникновения ошибок невероятно велик и включает в себя обычные опечатки, невнимательность, скрытые ограничения базовых абстракций и прямые ошибки в абстракциях или их реализациях. Правильный метод в каждом случае свой. Инструменты обычно нужны тогда, когда требуемый анализ («внимательное вглядывание») монотонный и повторяющийся. Замечу, что при программировании на Python обычно не требуется много инструментов, потому что пространство для поиска (отлаживаемая программа) значительно меньше, чем для других языков.

Как вы возобновляете прерванную работу над программой?

Гвидо: Это действительно интересный вопрос. Не помню, чтобы когда-либо задумывался, как я это делаю, хотя это происходит постоянно. Пожалуй, чаще всего я пользуюсь для этого системой управления версиями: вернувшись к работе над проектом, я выполняю сравнение (`diff`) между своим рабочим пространством и хранилищем, что позволяет мне узнать, в каком состоянии моя работа.

Если известно, что придется прерваться, и есть возможность, то я расставляю в незаконченном коде маркеры XXX, указывающие мне на конкретные подзадачи. Еще я пользуюсь тем, что перенял у Ламберта Меертенса (Lambert Meertens) лет 25 назад: ставлю специальную отметку в том месте исходного файла, где находится курсор. Я назвал эту отметку HIRO, в его честь. Это разговорное голландское слово, означающее «здесь», и я выбрал его, поскольку маловероятно, что оно может попасться в готовом коде. :-)

В Google у нас также есть инструменты, интегрированные с Perforce и помогающие мне на еще более раннем этапе: придя на работу, я могу выполнить команду, которая перечислит все незаконченные проекты в моем рабочем пространстве, что напомнит мне о проектах, над которыми я работал в предыдущий день. Я также веду дневник, в который иногда записываю специфические трудно запоминаемые строки (вроде команд оболочки или URL), нужные для выполнения конкретных задач в текущем проекте, например полный URL страницы статистики на сервере или команду для повторной сборки компонентов, над которыми я работаю.

Как, по-вашему, следует проектировать интерфейсы или API?

Гвидо: Это еще одна область, где я не слишком задумывался о том, какой процесс лучше всего, несмотря на то что разработал уйму интерфейсов (или API). Хорошо было бы здесь поместить рассуждения Джоша Блоха (Josh Bloch) на эту тему: он говорил о проектировании Java API, но большая часть сказанного подойдет для любого языка. Есть масса элементарных советов: что нужно выбирать понятные имена (существительные для классов, глаголы для методов), избегать сокращений, соблюдать последовательность в выборе имен, предоставлять малый набор простых методов, комбинирование которых обеспечивает максимальную гибкость, и т. д. Он прав, рекомендуя делать список аргументов коротким: два или три – это максимум, при котором можно избежать путаницы в их порядке. Хуже всего, когда несколько аргументов подряд имеют одинаковый тип: случайно переставив их, можно долго не замечать свою ошибку.

У меня есть несколько личных больных мест. Прежде всего – и это относится к динамическим языкам, – плохо, когда тип возвращаемого методом значения зависит от значения одного из аргументов: бывает трудно понять, что же возвращается, не зная об этой связи, – тот аргумент, от которого зависит тип, может быть взят из переменной, значение которой не вдруг определишь, читая код.

Во-вторых, мне не нравятся аргументы типа «флаг», которые должны существенным образом менять поведение метода. В таких API флаг

всегда оказывается константой в фактическом списке параметров, и вызов легче бы читался, имей API отдельные методы для каждого значения флага.

Еще меня раздражают API, в которых непонятно, возвращается ли новый объект или модифицируется уже существующий. Вот почему в Python метод `sort( )` для списка не возвращает значение: это подчеркивает, что он модифицирует существующий список. В качестве альтернативы есть встроенная функция `sorted( )`, которая возвращает новый отсортированный список.

Следует ли программистам приложений принять философию «меньше – значит, больше»? Насколько простым должен быть интерфейс пользователя, чтобы его можно было легко освоить?

Гвидо: Что касается графических интерфейсов пользователя, то мне кажется, что поддержка моего лозунга «меньше – значит, больше» наконец-то стала шириться. Фонд Mozilla нанял для проектирования интерфейса пользователя Азу Раскина (Aza Raskin), сына покойного Джефа Раскина (Jef Raskin) (соразработчика первоначального интерфейса Macintosh). В Firefox 3 есть по крайней мере один пример интерфейса пользователя, обладающего большой мощью при отсутствии кнопок, настроек, конфигурации и прочего: умная адресная строка следит за тем, что я печатаю, сравнивает с адресами, где я бывал раньше, и предлагает полезные завершения. Если я откажусь от того, что мне предлагается, введенные мной данные будут интерпретироваться как URL, а если он окажется неверным – то как запрос к Google. Очень толково! И заменяет три-четыре функциональных элемента, для которых потребовались бы отдельные кнопки или пункты меню.

Это соответствует тому, что Джеф и Аза повторяли в течение многих лет: клавиатура – очень мощное устройство ввода, и нужно искать новые способы ее применения, а не заставлять пользователя все делать мышью – самым медленным из устройств ввода. Прелесть в том, что для нее не нужна новая аппаратура, как в предлагаемых решениях в духе научной фантастики, вроде шлемов с виртуальной реальностью, датчиков движений глаз, а то еще и приемников волн, излучаемых мозгом.

Конечно, там еще многое недоделано – например, окно настроек Firefox выглядит ужасно, как будто изготовлено в Microsoft, с двумя или больше рядами вкладок и многочисленными модальными окнами, спрятавшимися в темных углах. Ну как можно запомнить, что для отключения JavaScript я должен открыть вкладку Content? А на какой вкладке искать cookies – Privacy или Security? Может быть, в Firefox 4 вместо диалогового окна настроек появится «умная функция», понимающая

ключевые слова, так что введя с клавиатуры `pass`, я попаду в раздел настройки паролей.

Какие уроки из вашего опыта изобретения, развития и распространения языка могли бы извлечь разработчики компьютерных систем настоящего и будущего?

Гвидо: У меня есть пара мыслей на эту тему. Не люблю философствовать, поэтому данный вопрос не относится к моим любимым или к тем, на которые у меня есть готовый ответ. Система должна предоставлять своим пользователям возможность расширять ее. Более того, крупная система должна быть расширяемой на двух или более уровнях.

С того момента, как впервые представил Python широкой публике, я получаю просьбы модифицировать язык, чтобы обеспечить поддержку в некоторых конкретных ситуациях. В ответ на такие просьбы я сразу предлагаю этим пользователям написать на Python код, который удовлетворит их потребности, поместить его в отдельный модуль и работать с ним. Это первый уровень расширяемости – если функциональность окажется достаточно полезной, она может в итоге попасть в стандартную библиотеку.

Второй уровень расширяемости – написать модуль расширения на Си (или на C++, или на каком-то другом языке). Модули расширения могут делать то, что недостижимо на чистом Python (хотя возможности чистого Python растут с каждым годом). Гораздо лучше добавить API на уровне Си, чтобы модули расширения могли залезать во внутренние структуры данных Python, чем менять сам язык, потому что изменения в языке должны удовлетворять самому высокому стандарту совместимости, качества, семантической ясности и т. п. Кроме того, язык может «разветвиться», если люди станут модифицировать реализацию языка в своих интерпретаторах и потом распространять дальше. Такие ответвления вызывают самые разнообразные проблемы, включая сопровождение собственных модификаций при изменении основного языка или слияние ветвей, созданных разными пользователями. Модули расширения таких проблем не создают: на практике большинство функций, необходимых расширениям, уже есть в API Си, поэтому для работы какого-то расширения редко требуется изменять API Си.

Другая моя мысль – это необходимость смириться с тем, что не все хорошо получится с первого раза. На ранних стадиях разработки, когда число пользователей еще невелико, следует скорейшим образом исправлять замеченные недоработки, не заботясь об обратной совместимости. Я люблю приводить замечательный рассказ, истинность которого подтверждена рядом очевидцев, – о том, как Стьюарта Фельдмана (Stuart Feldman), автора «Make» в UNIX v7, попросили заменить сим-

волы жесткой табуляции в синтаксисе Makefile. Он ответил в том смысле, что хотя и согласен с критикой, но вносить исправления слишком поздно, потому что у его программы уже около десятка пользователей.

По мере расширения круга пользователей приходится проявлять больше сдержанности, и в какой-то момент полная обратная совместимость становится абсолютным требованием. Но потом наступает другой момент, когда неудачных функций столько, что исправить их без нарушения совместимости оказывается невозможно. Тут можно поступить так, как я сделал с Python 3.0: объявить о новой версии, не поддерживающей обратную совместимость, воспользоваться этим, чтобы внести как можно больше исправлений, и дать пользователям достаточно времени для перехода на новую версию.

Для Python мы планируем параллельную поддержку Python 2.6 и 3.0 в течение долгого времени – гораздо дольше обычных сроков поддержки старых версий. Мы также предлагаем несколько стратегий перехода: средство автоматического преобразования исходного кода (далекое от совершенства) в сочетании с факультативными предупреждениями в версии 2.6 о том, что данная функциональность изменена в 3.0 (особенно если инструмент конвертирования не может правильно разобраться в ситуации), а также преобразование некоторых функций 3.0 обратно в 2.6. В то же время мы не стали полностью писать или проектировать 3.0 заново (как было с Perl 6 или, если речь о Python, Zope 3), тем самым минимизировав риск случайной потери важной функциональности.

Я заметил, что на протяжении последних четырех-пяти лет компании гораздо чаще стали выбирать динамические языки. Сначала PHP, в некоторых ситуациях Ruby, в других – определенно Python, в частности Google. Это меня заинтересовало. О чем они думали 20 лет назад, когда все эти удобные вещи уже могли делать такие языки, как Tcl и Perl, а несколько позднее – Python? Вы не замечали желания сделать эти языки более, так сказать, «корпоративно-дружественными»?

Гвидо: «Корпоративно-дружественными» обычно становятся, когда действительно толковые люди теряют интерес, и людям с более средними способностями приходится как-то позаботиться о себе. Не знаю, насколько сложно работать с Python при средних способностях. В известном смысле, можно предположить, что в Python большого вреда не нанесешь, потому что это язык интерпретируемый. С другой стороны, если писать действительно большую систему, не уделяя внимания модульному тестированию, можно оказаться неизвестно где.

Вы утверждали, что строка Python, или строка Ruby, или строка Perl, или строка PHP может превратиться в 10 строк кода Java.

Гвидо: Часто именно так и есть. Думаю, приятие в корпоративном мире, несмотря на наличие явно полезных функциональных пакетов, сдерживается страхом весьма консервативных менеджеров. Представьте себе ответственных за ИТ-ресурсы для 100 000 человек в компании, где ИТ не является главным продуктом – может быть, они выпускают автомобили, занимаются страхованием или чем-то еще, – но вся их работа выполняется с помощью компьютеров. Тот, кто отвечает за такую инфраструктуру, обязан быть очень консервативным. Они будут работать с тем, где установлено ПО с громким именем, скажем Sun или Microsoft, потому что они знают, что хотя Sun и Microsoft постоянно в чем-нибудь напортачат, но эти компании обязаны исправиться и устранить свои ошибки, даже если на это уйдет пять лет.

Проекты open source обычно не дают среднему ИТ-директору такого душевного спокойствия. Не знаю, изменится ли это когда-нибудь и каким образом. Может быть, если Microsoft или Sun вдруг станут поддерживать Python, каждая на своей виртуальной машине, программисты в компаниях действительно узнают, что с помощью более современных языков можно добиться более высокой производительности, ничего при этом не потеряв.

3

APL

В конце 1950-х, работая в Гарвардском университете, Кеннет Иверсон (Kenneth Iverson) придумал расширенные математические обозначения для точного описания алгоритмов. Затем команда, в которую вошли он, Эдин Д. Фалкофф (Adin D. Falkoff) и другие исследователи из IBM, превратила эту систему в полноценный язык программирования под названием APL. В этом языке используется расширенный набор символов, для которого нужна специальная клавиатура, а выглядит он как набор строк с незнакомыми символами, однако внутренняя последовательность языка облегчает его изучение, а беспрецедентные средства обработки массивов придают ему исключительную мощь. Его духовные последователи J и K развивают способность APL к кратким и мощным алгебраическим операциям.

Бумага и карандаш

Я читал статью, написанную вами и Кеном Иверсоном, где было сказано, что первые семь или восемь лет разработки прошли без участия компьютеров! Это позволило менять конструкцию языка, не заботясь о последствиях. Как повлияло на развитие языка появление первой программной реализации?

Эдин Д. Фалкофф: Да, первые годы развития APL, когда он назывался не иначе как «нотация Иверсона», мы в основном занимались математическими приложениями с помощью бумаги и карандаша, анализом цифровых систем и преподаванием. Мы представляли себе программирование в значительной мере как раздел математики, занятый исследованием и изобретением алгоритмов, и эту концепцию поддерживала символическая форма нотации. Привлекательность такой нотации как общего языка программирования стала очевидной не сразу и была поддержана усилиями разных людей (в частности, Херба Хеллермана (Herb Hellerman) из IBM, который стал экспериментировать с машинными реализациями важных элементов нотации, таких как функции-примитивы и операции над массивами). Все же правда, что на протяжении этого периода мы были полностью свободны в проектировании языка, не заботясь о проблемах «преемственности».

Наиболее существенными в раннем развитии языка были два этапа. Первым было создание и публикация формального описания System 360 («The Formal Description of System 360», *IBM Systems Journal*, 1964). Чтобы формально описать некоторые функции этой новой вычислительной системы, понадобилось ввести некоторые дополнения и изменения в ту нотацию, которая была описана в книге Иверсона («A Programming Language», Wiley). Вторым была разработка печатающего элемента для терминалов на базе Selectric, которую мы предприняли в ожидании применения языка на машине. Это наложило существенные ограничения, вызванные линейным характером печати и механическими требованиями конструкции Selectric. По-моему, в упомянутой вами статье «The Design of APL» (Проектирование APL, *IBM Journal of Research and Development*, 1974) довольно подробно рассказано о том, какое влияние эти два фактора оказали на эволюцию языка.

Первой полной реализацией языка стал, конечно, APL\360. В силу необходимости в него ввели средства для создания определений функций (т. е. программ), что на бумажно-карандашной стадии считалось само собой разумеющимся, и для управления окружением, в котором должна была выполняться программа. Появившиеся тогда идеи, такие как рабочее место и система библиотек, правила для областей видимости имен и использование общих переменных для обмена данными с другими системами, продолжают существовать до сего времени без особых

изменений. Написанные для APL\360 программы без модификации выполняются на известных мне современных APL-системах.

Правильнее сказать, что наличие реализации повлияло на дальнейшую эволюцию языка в результате строгого соблюдения принципа, согласно которому новые идеи всегда должны включать в себя старые, и, конечно, тщательного изучения поведения языка в новых и различающихся приложениях.

Определяя синтаксис, каким вы представляли себе типичного APL-программиста?

Эдин: Разрабатывая синтаксис, мы не думали о программистах как таковых, но предполагали, что этот язык должен стать средством общения между людьми, а заодно позволит людям общаться с машинами. Мы рассчитывали на таких пользователей, которых не смутит символический язык вроде алгебраического, а также предполагали, что они оценят мощь символического представления, поскольку оно облегчает формальные операции над выражениями и ведет к росту эффективности анализа и синтеза алгоритмов. Конкретно, мы полагали, что пользователю не нужно иметь большой опыт или знать математику, и действительно, с некоторым успехом применяли APL для обучения на уровне начальных и средних классов школы.

Со временем APL привлек внимание ряда очень талантливых и опытных программистов, которые работали с ним и приняли участие в его дальнейшем развитии.

Не ограничил ли сложный синтаксис распространение APL?

Эдин: Синтаксис APL и его влияние на принятие языка вполне можно обсудить, хотя я не согласен с утверждениями о его «сложности». APL основан на математической нотации и алгебраических выражениях, упорядочен путем удаления неправильных форм и обобщения принятой нотации. Например, было решено, что бинарные функции вроде сложения и умножения будут помещаться между своими двумя аргументами, а унарные функции для единообразия будут записываться перед своим аргументом без тех исключений, которые есть в обычной математической нотации, поэтому абсолютное значение записывается в APL как одна вертикальная черта перед аргументом, а не две черты по его сторонам, равно как символ факториала пишется в APL перед аргументом, а не после него. В этом отношении синтаксис APL проще, чем синтаксис его исторического предшественника.

Синтаксис APL был проще алгебраической нотации и других языков программирования еще в одном важном отношении: при вычислении выражений в APL все функции имеют одинаковый приоритет, поэтому не нужно помнить, что возведение в степень выполняется раньше умно-

жения, или каков приоритет собственных функций. Правило простое: первым вычисляется самый правый член выражения.

Поэтому я не считаю, что синтаксис APL ограничил его распространенность, хотя, возможно, на это повлиял набор символов, в котором много таких, которых нет на обычной клавиатуре.

Почему вы решили использовать специальный набор символов? Изменялся ли он со временем?

Эдин: Набор символов определялся применением стандартной математической нотации, которая была дополнена несколькими греческими буквами и самоочевидными значками вроде квадрата. Практическое влияние также оказал линейный характер печатающего устройства, что привело к изобретению символов, которые можно напечатать наложением одного на другой. Позднее, когда появились другие терминалы и устройства ввода, эти составные символы стали самостоятельными элементарными символами, и были введены появившиеся в этих устройствах новые символы, например ромб как разделитель операторов.

Вы старались более продуктивно использовать ресурсы, в то время ограниченные?

Эдин: Набор символов явно был выбран под действием стремления оптимально использовать имевшиеся ресурсы; но сжатая символическая форма разрабатывалась и сохранялась в силу уверенности в том, что она облегчает анализ и формальные операции над выражениями. Кроме того, краткость программ в сравнении с равноценными им, но написанными на других языках, облегчает понимание логики их выполнения, если взять на себя труд прочесть ее в сжатом представлении APL.

Наверное, чтобы освоить язык, особенно набор символов, нужно было долго учиться. Определялся ли уровень изучения программистом языка APL в процессе естественного отбора? Были ли эксперты по APL более продуктивны? Был ли их код выше по качеству, с меньшим числом ошибок?

Эдин: Изучить APL в такой мере, чтобы писать программы, скажем, на уровне Фортрана, на самом деле было нетрудно и не требовало много времени. Программирование на APL было более продуктивным благодаря простоте правил и наличию функций-примитивов для таких операций над данными, как, скажем, сортировка, или математических функций, например для обращения матриц. Эти факторы способствовали краткости программ на APL, благодаря которой их было легче анализировать и отлаживать. Высокую эффективность нужно также отнести на счет качества реализаций APL, использования рабочих мест

со всеми их удобствами и интерактивных интерпретирующих систем на базе терминалов.

Сверхкраткая форма описания может оказаться крайне полезной для устройств с маленьким экраном, таких как PDA или смартфоны! Если учесть, что изначально APL кодировался для таких монстров, как IBM System/360, удастся ли его расширить, чтобы сделать пригодным для современных проектов, где нужно управлять сетевыми соединениями и мультимедийными данными?

Эдин: Реализация APL для карманных устройств могла бы, по крайней мере, обеспечить создание очень мощного карманного калькулятора. Что касается сетей и мультимедиа, то я не вижу с этим проблем, поскольку такие приложения очень давно существуют в APL-системах. Средства для работы с GUI также обычно присутствуют в современных APL-системах.

Уже на ранних стадиях разработки APL-систем появились средства для управления операционной системой и аппаратной частью машины из функций APL, и с их помощью программисты APL-систем смогли управлять эффективностью самого APL. А коммерческие APL-системы с разделением времени, экономическая эффективность которых была связана с сетевым взаимодействием, использовали APL для организации работы в сети.

Верно, первые коммерчески успешные APL-системы кодировались на больших машинах, но самые ранние реализации, продемонстрировавшие осуществимость APL-систем, выполнялись на относительно небольших машинах, таких как IBM 1620 и IBM 1130, а также IBM 1500, которая широко применялась в образовательных задачах. Существовала даже реализация для одной ранней экспериментальной настольной машины, известной как «LC» (от low cost – низкая цена) и имевшей очень ограниченную память и маленький диск. Эволюция реализации IBM APL довольно подробно описана в статье «The IBM Family of APL Systems» (IBM-семейство APL-систем, *IBM Systems Journal*, 1991).

Основные принципы

Вы сознательно стремились к стандартизации?

Эдин: Мы действительно довольно быстро занялись стандартизацией: кажется, я написал об этом статью, и мы вошли в ISO. Мы всегда стремились к стандартизации и в значительной мере преуспели в этом. Мы старались воспрепятствовать тому, чтобы кто-то менял базовые структуры языка, делал произвольные добавления, осложняющие синтаксис, или нарушал основные принципы, которые мы хотели сохранить.

Что было главным в стремлении к стандартизации — сохранение совместимости или чистота идеи?

Эдин: Желание стандартизировать – это вопрос экономики. Мы хотели сделать APL экономически состоятельным, и поскольку его реализацией и применением было занято множество людей, мысль об установлении стандарта казалась правильной.

У каждого поставщика своя реализация APL. Если нет надежного стандарта, что делать с расширением, которое в одних системах работает, а в других — нет?

Эдин: Этот вопрос довольно тщательно прорабатывался комиссиями по стандартизации APL в стремлении достичь компромисса между расширяемостью и чистотой.

Вы хотели дать людям возможность решать проблемы, которые не могли предвидеть, но так, чтобы при этом не затрагивалась сущность вашей системы. Выдержал ли, по-вашему, язык 40-летнюю проверку временем? Верны ли принципы, на которых он основан?

Эдин: Думаю, что да: я не вижу существенных ошибок.

Это обусловлено тщательностью разработки или прочностью теоретической опоры на алгебру?

Эдин: Мы оба были неглупы, убеждены в необходимости простоты и практичности и не склонны отступать от своего мировоззрения.

По своему опыту с другими языками я знал, как трудно выучить и запомнить все правила, поэтому стремился сделать язык простым, чтобы с ним можно был работать.

Наш образ мыслей можно проследить по нашим статьям, особенно написанным совместно Иверсоном и мной. Сам я позднее написал статью «A Note on Pattern Matching: Where do you find the match to an empty array?» («Замечание по поводу поиска: как искать соответствие пустому массиву», APL Quote Quad, 1979), где путем рассуждений, подкрепляемых небольшими программами и алгебраическими принципами, получил последовательные и полезные результаты. Там рассматривались разные варианты и делался вывод, что лучший из них тот, который допускает простейшее выражение.

Очень заманчиво построить язык на основе небольшого набора принципов, выводя из этих принципов новые идеи. Это напоминает описание математического подхода. Какова роль математики в компьютерных науках и программировании?

Эдин: Я считаю компьютерные науки разделом математики.

Программирование математических вычислений – это очевидная математика, собственно численный анализ, требуемый для поддержания постоянного согласования дискретных цифровых операций с непрерывностью теоретического анализа.

Вот некоторые другие мысли, которые приходят в голову: влияние математических задач, решаемых только с помощью интенсивных вычислений, что стимулирует стремление к увеличению скорости; дисциплина логического мышления, необходимая в математике и распространяющаяся на все виды программирования; понятие алгоритма, являющегося классическим инструментом математики; различные специальные разделы математики, например топология, применимые к анализу вычислительных задач.

Я знаком с дискуссиями, в которых вы и некоторые другие предлагали использовать APL при обучении программированию и математике в младших и старших классах.

Эдин: Мы немного занимались этим, особенно в начале, и это было довольно интересно.

Тогда существовали только терминалы с пишущими машинками, и мы установили их в ряде местных закрытых школ. В частности, в одной из них, где обучались «трудные» подростки, мы дали задания для выполнения на машинке и натравили на них учеников.

Любопытно, что некоторые из этих учеников, считавшихся невосприимчивыми к обучению, проникали в школу во внеурочное время, чтобы дополнительно поработать. Терминалы с пишущими машинками были подключены к нашей системе с разделением времени.

Им так понравилось, что они неожиданно захотели поработать после уроков?

Эдин: Да.

Вы использовали APL для обучения «программистскому мышлению» непрограммистов. Почему APL оказался для них привлекателен?

Эдин: Когда все начиналось, не было, в частности, всех этих дополнительных трудов: чтобы сложить два числа, не нужно было писать объявления, поэтому если нужно было прибавить 7 к 5, вы писали 7 + 5, а не говорили, что есть число 7 и есть число 5, это числа с плавающей запятой или нет, и результат будет числом, и я хочу сохранить его тамто – поэтому в APL было мало препятствий для выполнения того, что требовалось.

То есть, если вы учитесь программировать, то первый шаг для получения первого результата очень невелик. Достаточно написать, что вы

хотите сделать, и не нужно ублажать компилятор, чтобы это заработало.

Эдин: Совершенно верно.

Легко начать работать. Но помогает ли это стать программистом или расширить свои познания в программировании?

Эдин: Такая доступность позволяет экспериментировать, а когда вы экспериментируете и пробуете разные вещи, вы учитесь, поэтому, мне кажется, складываются условия, благоприятствующие развитию мастерства программирования.

Нотация, выбранная вами для APL, отличается от традиционной алгебраической нотации.

Эдин: Ну, не сильно... приоритетность операций другая. Правило простое: двигаться справа налево.

Легче ли от этого в учении?

Эдин: Да, потому что правило только одно, и не нужно объяснять, что если это функция, то нужно действовать так, а если это возведение в степень, то его нужно выполнить перед умножением, и тому подобное. Просто говоришь: вот строка инструкций, и нужно выполнять ее справа налево.

Вы намеренно отошли от традиционной нотации и порядка выполнения действий для большей простоты?

Эдин: Именно так. Для большей простоты и большей общности.

Думаю, главная ответственность здесь на Иверсоне. Он хорошо знал алгебру и очень интересовался преподаванием. Его любимым примером было представление многочленов, которое осуществляется в APL крайне просто.

Впервые увидев эту незнакомую нотацию, я воспринял ее как весьма простую в целом. Как вы оцениваете простоту проекта или реализации? Нужен ли для этого тонкий вкус, большой опыт или есть какая-то жесткая процедура для определения оптимальной простоты?

Эдин: Думаю, в какой-то мере эта оценка неизбежно субъективна, поскольку зависит от вашего опыта и подготовки. Мне кажется, что чем меньше правил, тем обычно проще.

Вы начали с небольшого набора аксиом, возведя все на этом фундаменте, но если разобраться в этом маленьком наборе аксиом, то можно вывести из него более сложные вещи?

Эдин: Что ж, возьмем все те же правила порядка выполнения операций. По-моему, проще определить, что действия выполняются справа налево, чем заводить таблицу, в которой будет описано, что эта опера-

ция выполняется первой, а вот эта – второй. Здесь выбор между одним правилом и почти бесчисленным их множеством.

Видите ли, в каждом конкретном приложении вы задаете собственный набор переменных и функций, и для отдельного приложения может оказаться, что проще написать какие-то новые правила, но если это язык общего назначения, такой как APL, нужно начинать с возможно меньшего количества правил.

Чтобы дать более широкие возможности развития тем, кто разрабатывает системы с помощью этого языка?

Эдин: Тот, кто создает приложение, фактически создает язык: в принципе, программирование состоит в разработке языков, пригодных для конкретных приложений.

Вы описываете задачу на языке, специфичном для ее предметной области.

Эдин: Но все эти объекты, особенно существительные и глаголы, объекты и методы, их же нужно как-то определять, например с помощью такого универсального языка, как APL.

Так что вы описываете это на APL, а потом определяете свои операции, позволяющие делать то, что вам нужно, в этом приложении.

Вы видите свою задачу в создании конструктивных элементов, с помощью которых пользователь сможет описать свою задачу?

Эдин: Моя задача, если хотите, в создании элементарных конструктивных элементов, базовых инструментов для создания конструктивных элементов, пригодных для решения задач, возникших в той области, где работают пользователи.

Видимо, такую же задачу ставили перед собой другие разработчики языков – я вспоминаю Чака Мура с Фортом, Джона Маккарти с Лиспом, Smalltalk начала 1970-х.

Эдин: Думаю, это это как раз тот случай.

Маккарти — теоретик, и он хотел разработать систему, которая эффективно описывала бы лямбда-исчисление, но мне кажется, что обычно лямбда-исчисление не столь удобно, как добрая старая алгебра, от которой происходит APL.

Допустим, я хочу разработать новый язык программирования. Каким был бы ваш главный совет мне в таком случае?

Эдин: Мой главный совет – занимайтесь тем, что вам нравится, работой, которая доставляет вам удовольствие, тем, что поможет вам достичь вашей цели.

Мы всегда преследовали сугубо личные цели, и судя по тому, что мне довелось прочесть, это характерно для большинства разработчиков. Сначала они занимались тем, что их интересовало, а потом оказывалось, что это может иметь широкое применение.

Бывали ли при разработке APL моменты, когда вы видели, что двигаетесь не туда и что нужно избавляться от сложности, или когда вам удавалось объединить несколько решений в одном, более простом?

Эдин: Примерно так, но обычно возникал следующий вопрос: будет ли это обобщение включать в себя то, что у нас уже есть, и насколько вероятно, что оно заметно расширит наши возможности, очень незначительно увеличив сложность.

Мы уделяли значительное внимание граничным условиям, например, что получится в пределе, если мы перейдем от 6 к 5, потом к 4 и так далее до 0. То есть при редукции вы применяете к вектору такую функцию, как сложение, и если сначала у вас вектор с n элементами, а потом с $(n - 1)$ элементами, и т. д., то что произойдет, когда в конечном счете не останется элементов? Какова будет сумма? Она должна быть 0, потому что это нейтральный элемент.

Если это операция умножения, то умножение для пустого вектора дает 1, потому что это нейтральный элемент для данной функции.

Значит, вы пытались обобщить несколько решений, задаваясь вопросом, что произойдет, скажем, при стремлении к 0. И если еще не знали, что при редукции нужно заканчивать нейтральным элементом, когда n = 0, то могли рассмотреть оба эти случая и сделать вывод: в этом случае 0, а в этом − 1, потому что это нейтральный элемент.

Эдин: Верно. Такую процедуру мы тоже использовали.

Особые случаи очень важны, и при эффективной работе с APL вы используете тот же критерий с более сложными функциями, разрабатываемыми для конкретного приложения. Часто это приводит к неожиданному, но приятному упрощению.

Влияет ли конструкция языка на конструкцию программ на этом языке?

Эдин: Да, поскольку, как я уже говорил, программирование – это процесс проектирования языка. Я считаю, что это очень важно, хотя, насколько я знаю, об этом редко пишут.

Лисп-программисты это делают, но для многих других языков, появившихся позднее, особенно для Алгола и его Си-потомков, такой подход не характерен. Можно ли провести грань между тем, что встроено в язык, и всем прочим, считая это вторым сортом?

Эдин: Что считать вторым сортом? В APL этот второй сорт следует тем же правилам, что и первый, и никаких проблем не возникает.

Почти то же самое можно сказать о Лиспе, Scheme или Smalltalk, но в Си есть четкое различие между операторами и функциями, с одной стороны, и пользовательскими функциями – с другой. Следует ли считать это ошибкой проектирования?

Эдин: Не знаю, можно ли назвать это ошибкой, но мне кажется, что проще установить одинаковые правила как для примитивов, так и для не-примитивов.

Вспомните свою самую крупную ошибку в проектировании или программировании. Чему она вас научила?

Эдин: Когда работа над APL только началась, мы намеренно избегали таких проектных решений, которые ориентировались бы на компьютерную среду. Например, мы отказались от объявлений, считая их лишним бременем для пользователя, поскольку машина может легко определить размер и тип объекта данных по самому этому объекту в момент его ввода или генерации. Однако со временем APL распространился, в него стали вкладывать существенные средства, и не учитывать факторы, связанные с аппаратной частью, становилось все труднее.

Пожалуй, моя крупнейшая ошибка состояла в недооценке прогресса аппаратных средств и чрезмерном консерватизме в проектировании систем. Например, изучая возможность реализации APL на PC, я предлагал опустить последние расширения языка, относящиеся к обобщенным массивам и комплексным числам, потому что их удовлетворительную работу можно было обеспечить только при повышенных, по меркам того времени, требованиях к аппаратуре. К счастью, меня не послушали, и довольно скоро благодаря росту объема памяти и скорости процессоров ПК такие мощные расширения стали вполне осуществимыми.

Сказать, какие крупные ошибки были допущены в программировании, тяжело, потому что если вы пишете достаточно сложную программу, ошибки в ней неизбежны. И уже от инструментов программирования будет зависеть, как станет развиваться ошибка, когда она будет обнаружена, и сколько труда потребует ее исправление. Модульность и применение готовых идиоматических фрагментов кода, свойственные стилю функционального программирования, который поощряется APL, позволяют сократить возникновение и распространение ошибок, не давая им превратиться в крупные проблемы.

Что касается ошибок в конструкции самого APL, то их, в основном, удалось избежать благодаря нашему методу разработки, предполагаю-

щему выбор окончательного решения на основе единогласия, достигаемого между проектировщиками и программистами, учета замечаний пользователей, применяющих язык на практике в разных условиях, и нашего собственного опыта работы с языком.

Однако то, что для одного человека является осознанным выбором, другому может представляться ошибкой, и даже долгий опыт не всегда снимает это противоречие. Вспоминаются два случая.

Первый – это набор символов. С самого начала нас пытались убедить, что вместо абстрактных символов, выбранных нами для представления функций-примитивов, лучше использовать зарезервированные слова. Мы отстаивали свою позицию, заключавшуюся в том, что в действительности мы расширяем математику, а развитие математической нотации явно идет в сторону применения символов, облегчающих формальные действия с выражениями. Позднее Кен Иверсон, со своим непреходящим интересом к преподаванию математики, ограничился набором символов ASCII для языка J, над которым стал работать, чтобы сделать J доступным учащимся и другим людям, у которых не было специальной аппаратуры. Лично я продолжаю придерживаться символического подхода: он обеспечивает историческую преемственность и облегчает чтение. Время покажет, какое из этих направлений было ошибочным, или подтвердит несущественность различий между ними.

Второе, что приходит на ум как возможная ошибка в выборе направления, это подход к обобщенным массивам, т. е. массивам, скалярные элементы которых могут представлять собой некие структуры языка. После выхода APL\360 в качестве продукта IBM (один из первых случаев, когда в 1966 или 1967 году IBM стала раздельно поставлять программные и аппаратные продукты) мы задумались о расширении языка, допускающем более общие массивы, и стали интенсивно изучать и обсуждать их теоретические основы. В конце концов были созданы APL-системы с конкурирующими способами обращения со скалярными элементами и синтаксическим следствиями этого. Любопытно будет посмотреть на дальнейший ход событий в свете того, что общий интерес к параллельному программированию начинает приобретать все большее коммерческое значение.

Параллелизм

Каковы последствия (для проектирования приложений) представления данных в виде коллекций, а не отдельных единиц?

Эдин: Это довольно крупная тема, на что указывает распространение «векторных языков» (array languages) и введение массивов-примитивов

в такие языки, как Фортран, но мне кажется, что есть два важных аспекта представления в виде коллекций.

Во-первых, конечно, проще думать, когда не нужно погружаться в мелочи, связанные с обработкой отдельных элементов. Если, например, нужно узнать, сколько нулевых элементов в некоторой коллекции, то ближе к нашему обычному способу мышления будет запись простого выражения для получения нужного результата, а не оператора цикла в любой из имеющихся его форм.

Во-вторых, в программах, где операции выполняются над коллекциями, лучше видна возможность распараллеливания, а это ведет к более эффективному использованию современной аппаратуры.

Есть идея добавить функции более высокого порядка в современные языки, такие как C++ или Java, где тратится много времени на то, чтобы раз за разом писать циклы `for( )`. Например, у меня есть некая коллекция, и я хочу что-то проделать с каждым из ее элементов. Но в APL эта проблема решена 40–45 лет назад!

Эдин: Не знаю, сколько лет назад, но здесь просматриваются два этапа. Во-первых, нужно сделать массивы примитивами, а во-вторых, ввести оператор each, который применяет любую функцию к любой коллекции элементов. Но у нас возникали такие вопросы, как «не ввести ли примитивы специфических циклов?». Мы решили не делать это, чтобы не усложнять синтаксис, поскольку достаточно просто написать несколько нужных циклов обычным образом.

Не усложнять синтаксис – для реализации или для пользователей?

Эдин: Для всего: его нужно читать и человеку, и машине; либо это простой синтаксис, либо нет.

Вводя новые типы предложений, вы явно увеличиваете сложность. Возникает вопрос, оправданно ли такое усложнение, то есть вступает в игру здравый смысл проектировщика. И мы всегда соглашались с тем, что не будем вводить новый синтаксис для циклов, потому что вполне можно обойтись имеющимися средствами.

Вы сказали, что APL обладает реальными преимуществами для параллельного программирования. В отношении использования массивов в качестве элементарных структур данных в языке мне это понятно. Вы также упомянули общие переменные. Какую роль играют они?

Эдин: В APL общие переменные доступны одновременно нескольким процессорам. Использующие общий доступ процессоры могут быть процессорами APL или иметь иную природу. Допустим, у вас есть переменная X, и, с точки зрения APL, чтение и запись X происходят так же, как для обычных переменных. Однако у вас может быть другой про-

цессор, скажем, файловый, у которого тоже есть доступ к X, поскольку это общая переменная, и какое бы значение APL ни присвоил X, файловый процессор может использовать его по-своему. Аналогично когда он присвоит свое значение X и его прочтет APL, процессор APL так же применит свои знания и интерпретирует это значение каким-то своим способом. Вот что следует из того, что X – общая переменная.

В системах APL, например APL2 от IBM, есть некоторый протокол доступа к этим переменным, позволяющий избежать ситуации гонки.

Может ли компилятор автоматически определять параллелизм, о котором вы говорили? Допустим, я хочу перемножить два массива и добавить значение к каждому элементу массива. Это легко описать на APL, но сможет ли компилятор осуществить неявное распараллеливание?

Эдин: По определению в APL не важно, в каком порядке вы выполняете действия над элементами массива, поэтому компилятор или интерпретатор в любой реализации имеет право выполнять их одновременно или в любом произвольном порядке.

Помимо упрощения на уровне языка создатели реализации могут получить огромную свободу для изменения способа реализации, использования новых возможностей аппаратной части или создания такого механизма, как автоматическое распараллеливание.

Эдин: Верно, потому что согласно определению языка, которое, собственно, определяет, что происходит, когда применяется процессор, порядок не имеет значения. Это очень обдуманное решение.

Было ли это решение в свое время уникальным для языков программирования?

Эдин: Я не очень хорошо знаком с историей языков программирования, но поскольку наш язык был практически единственным серьезным языком для работы с массивами, то возможно, что это так.

Интересно было бы обсудить коллекции и большие наборы данных, которыми явно озабочены современные программисты. APL появился раньше, чем были изобретены реляционные базы данных. Сейчас большой объем данных хранится в структурах, содержащих данные разных типов, в реляционных базах данных и в крупных неструктурированных коллекциях, таких как веб-страницы. Хорошо ли справляется APL с такими данными? Есть ли в нем модели, которые могут быть полезны тем, кто программирует на более популярных языках, таких как SQL, PHP, Ruby и Java?

Эдин: Массивы APL могут содержать в качестве элементов как скаляры, у которых нет внутренней структуры, так и не скаляры, которые

могут иметь любую степень сложности. Не скалярные элементы рекурсивно образуются из других массивов. «Бесструктурные» коллекции вроде веб-страниц вполне можно представить массивами APL и обрабатывать с помощью элементарных функций APL.

Что касается очень больших массивов, то APL позволяет работать с внешними файлами как с объектами APL. После установления связи между именем в рабочем пространстве и внешним файлом к файлу можно применять операции с использованием выражений APL. Для пользователя все выглядит так, как если бы файл находился в рабочем пространстве, даже если в действительности этот файл во много раз превосходит рабочее пространство по размеру.

Очень трудно указать конкретно, чему разработчики других языков могут поучиться у APL, и с моей стороны было бы самонадеянностью обсуждать детали перечисленных вами языков, специалистом в которых я не являюсь. Однако судя по тому, что я о них читал, в целом принципы, положенные в основу разработки APL и описанные, например, в нашей статье 1973 года «The Design of APL», оказали влияние на последующую работу по проектированию языков.

Из двух важнейших принципов – простота и практичность – последнему повезло больше. Достичь простоты труднее, потому что сложность ничем практически не ограничена. Мы боролись за простоту APL, тщательно определив разрешенную область действия элементарных операций, поддерживая абстрактный характер объектов APL и борясь с соблазном включить особые случаи, представленные в операциях других систем.

Иллюстрацией к этому служит тот факт, что в APL нет понятия «файл». Есть массивы, действующие как файлы, если этого требует приложение, но элементарных функций, специально предназначенных для работы с файлами, нет. Тем не менее практическая необходимость в эффективной работе с файлами способствовала появлению на ранней стадии парадигмы общих переменных, которая является общим понятием, полезным во множестве случаев, где программе APL требуется вызвать другой (APL или не-APL) вспомогательный процессор.

Позднее было добавлено еще одно средство, использующее общее понятие пространства имен, с помощью которого программы APL могли непосредственно манипулировать объектами, находящимися вне рабочего пространства, включая доступ к полям и методам Java, сверхбольшим совокупностям данных, скомпилированным программам на других языках и прочему. Интерфейс пользователя к общим переменным и пространствам имен строго следует синтаксису и семантике APL, что обеспечивает его простоту.

Таким образом, не вникая в детали, можно сказать, что новейшие языки могли бы выиграть, если бы строго придерживались собственных элементарных понятий и определяли их возможно более общим образом в контексте тех применений, на которые они рассчитаны.

Что касается конкретных характеристик APL как модели, то APL продемонстрировал, что объявления не являются обязательными, хотя в некоторых случаях увеличивают эффективность исполнения, и что количество различных типов данных может быть весьма невелико. В новых языках было бы полезно придерживаться этих направлений, а не принимать как должное, что пользователь обязан помогать компилятору, передавая ему такую информацию, относящуюся к реализации.

Кроме того, указатели не являются в APL элементарным понятием, и нужды в них никогда не ощущалось. Конечно, по возможности нужно определить элементарные операции языка над коллекциями данных, имеющими абстрактную внутреннюю структуру, такими как обычные массивы, деревья и прочие.

Вы правы, отметив, что APL предшествовал изобретению реляционных баз данных. Доктор Э. Ф. Кодд и группа APL работали в Исследовательском центре Уотсона IBM в 1960-х годах, когда он разрабатывал теорию реляционных баз данных, и я считаю, что мы оказали на его работу сильное влияние. В частности вспоминается горячий спор между нами, когда мы продемонстрировали, что для описания связей между сущностями можно использовать простые матрицы, а не сложные системы скалярных указателей.

Наследие

Я знаю, что APL оказал большое влияние на разработку Perl. Поговаривают, что свою загадочность Perl частично унаследовал от APL. Не знаю, можно ли это считать комплиментом.

Эдин: Позволю себе привести пример подобного комплимента. В проектировании и применении языков программирования большую роль играют политические соображения, особенно в таком месте, как IBM, где с этим связан бизнес. В разное время устраивали эксперименты с целью выяснить, что лучше – APL или, скажем, PL1 или Фортран. Ответы всегда были предвзятыми, потому что судьи принадлежали к противоположным лагерям, но комментарий одного из функционеров я запомнил на всю жизнь. Он сказал, что APL не может быть хорошим языком, потому что даже двое самых умных из известных ему людей, Иверсон и Фалкофф, не могут заставить остальных поверить в него.

Чем может быть полезен ваш опыт создания, развития и распространения языка для разработчиков компьютерных систем сегодня и в обозримом будущем?

Эдин: Разработка систем не состоит из одних только технических и научных решений. Сильное влияние оказывают на нее экономические и политические соображения, особенно если лежащая в основе техническая сторона допускает большую гибкость, как в проектировании языков и систем.

В тот период середины 1960-х, когда APL становился важным инструментом для использования в IBM и рассматривался вопрос о превращении его в продукт, нам пришлось бороться с «царем языков» IBM, который постановил, что в будущем компания станет поддерживать только PL/1 – не считая, конечно, Фортран и Кобол, которые укоренились в отрасли и потому их нельзя было совсем забросить.

Как показала история, эта позиция была не реалистична и обречена на провал, но в то время это не было столь очевидно, учитывая доминирующее положение IBM в компьютерной индустрии и влияние определенных фракций в структуре управления компании.

Нам пришлось бороться с этой политикой, чтобы получить поддержку, необходимую для выживания APL. Битва велась по нескольким фронтам: будучи членами исследовательского подразделения IBM, мы максимально использовали такие возможности, как профессиональные переговоры, семинары и формальное обучение, чтобы распространять сведения об уникальных, по техническим меркам того времени, характеристиках APL; мы привлекали в союзники – когда могли их найти – влиятельных лиц внутри компании, чтобы противодействовать давлению административной структуры; мы передавали свою систему APL\360 на площадки, где велись разработки и производство, и поддерживали ее там; мы подогревали интерес важных клиентов к системам APL, чтобы сделать APL доступным вне компании, хотя бы в порядке эксперимента; мы завели себе союзников в маркетинговом отделе. И мы добились успеха, поскольку APL\360 был среди первых программных продуктов IBM, выведенных на рынок, когда в конце 1960-х продажи аппаратных и программных средств были разделены.

Очень важный шаг был сделан вследствие интереса, который наши рассказы и демонстрации вызвали в NASA, в Космическом центре Годдарда. В 1966 году эта организация попросила предоставить ей доступ к нашей внутренней системе APL, чтобы провести с ней эксперименты. Это был очень важный клиент, и представители маркетинга IBM потребовали, чтобы мы удовлетворили эту просьбу. Однако мы возражали, настаивая на том, чтобы сначала нам дали возможность провести недельный курс обучения в центре Годдарда.

Мы получили согласие, но возникла трудность с реализацией: системы с разделением времени, такие как APL\360, требовали в те времена подключения терминалов к центральной системе с помощью акустических модемов, работающих со специализированными телефонными устройствами. На другом конце были такие же устройства, подключенные к центральному компьютеру, и их не хватало. После того как были приняты все административные решения для продолжения проекта, мы обнаружили, что телефонные компании ни в Нью-Йорке, ни в Вашингтоне не могут предоставить нам устройства, необходимые для проведения занятий в Космическом центре.

Вопреки обычной практике работы только с собственным оборудованием, телефонная компания в Вашингтоне согласилась установить те телефонные устройства, которые мы сможем достать. Но как ни пытались связисты IBM уговорить нью-йоркскую компанию найти где-нибудь эти аппараты, той этого не удалось, однако нам намекнули, что они посмотрят сквозь пальцы, если мы используем уже имеющееся у нас их оборудование способом, который они не могут одобрить официально.

После этого мы отключили половину линий на нашем центральном компьютере и высвободившиеся аппараты отвезли в Космический центр в фургоне IBM. Они были негласно установлены местной телефонной компанией, и мы смогли провести свой курс, что стало первым случаем использования системы APL\360 за стенами IBM и не самой компанией, несмотря на политику поддержки только PL/1.

Что в языке вызывает у вас наибольшие сожаления?

Эдин: Мы отдали созданию APL все свои силы и много потрудились на политической ниве, добиваясь его признания и широкого применения. С учетом этих обстоятельств, я не вижу, о чем в языке можно жалеть. Сейчас можно пожалеть, что мы не начали раньше разрабатывать эффективный компилятор и не приложили к этому больше усилий, но неизвестно, на какие компромиссы пришлось бы тогда пойти, учитывая ограниченность имевшихся ресурсов. Кроме того, есть основания полагать, что сегодняшний интерес к параллельным вычислениям и введение операций с массивами в стиле APL в традиционные компилируемые языки вроде Фортрана приведет в конечном счете к равноценным результатам.

Что вы считаете успешным в своей работе?

Эдин: APL оказался очень полезным инструментом для развития многих аспектов деятельности IBM. Он обеспечил более простой доступ к компьютерам, что позволило исследователям и создателям продуктов более эффективно заниматься своими существенными задачами – от теоретической физики до разработки дисплеев с плоскими экранами. Он

также применялся для создания прототипов крупных бизнес-систем, таких как сборочные линии и склады, которые можно было быстро запустить и протестировать, не затягивая дело реализациями на других программных системах.

Мы успешно превратили APL в целую линейку продуктов IBM, указав путь другим компаниям, создавшим собственные системы APL в соответствии с международным стандартом.

APL нашел заметное применение в учебных заведениях в качестве инструмента и дисциплины, то есть достигнута одна из главных целей его разработки – использование в учебных целях.

APL стал предвестником языков программирования и систем, в которых массивы являются элементарными объектами данных, а общие переменные участвуют в управлении одновременной обработкой, и в этом качестве, несомненно, окажет большое влияние на будущие разработки в области параллельного программирования. Отрадно видеть, что за последние месяцы в компьютерной области возникло три разных консорциума для работы в этой сфере.

4

Форт

Форт (Forth) – это стековый, конкатенативный язык, созданный в 1960-х годах Чарльзом Муром. Его основная особенность: использование стека для хранения данных и слов для операций, которые берут из стека аргументы и помещают в стек результат. Язык настолько компактен, что может использоваться как во встраиваемых системах, так и в суперкомпьютерах, и достаточно выразителен, чтобы создавать полезные программы длиной в несколько сотен слов. Среди продолжателей идеи – colorForth самого Чака Мура (Chuck Moore), а также язык программирования Factor.

Язык Форт и языковое проектирование

Как бы вы определили Форт?

Чак Мур: Форт – компьютерный язык с минимальным синтаксисом. Он характеризуется наличием явного стека параметров, что позволяет эффективно вызывать подпрограммы. Отсюда постфиксные выражения (операторы пишутся после аргументов) и стиль программирования с высокой степенью структурирования, при котором множество коротких программ передают друг другу параметры через стек.

Приходилось читать, что название Forth означало четвертое поколение ПО. Не могли бы вы рассказать об этом подробнее?

Чак: «Форт» происходит от fourth (четвертый), что намекает на «язык программирования четвертого поколения». Кажется, я перескочил через поколение. Фортран/Кобол – языки первого поколения, Алгол/Лисп – второго. Во всех этих языках большую роль играл синтаксис. Чем сильнее развит синтаксис, тем больше возможностей для проверки ошибок. Однако большая часть ошибок – синтаксические. Я решил минимизировать синтаксис, отдав предпочтение семантике. В самом деле, в Форте слова полны смысла.

Вы рассматриваете Форт как языковый набор инструментов. Можно понимать под этим относительную простоту синтаксиса в сравнении с другими языками программирования и возможность создания словарей из более которотких слов. Или что-то еще?

Чак: Нет, главное то, что язык обеспечивает высшую степень структурирования. Программа на Форте состоит из множества коротких слов, тогда как программа на Си состоит из меньшего числа длинных слов.

Под коротким словом я понимаю его определение размером примерно в одну строку. Язык строится путем определения новых слов через уже существующие, и эта иерархия развивается, пока не наберется, скажем, тысяча слов. Проблема в том, чтобы: 1) решить, какие слова полезны, и 2) запомнить их все. Я сейчас работаю над приложением, в котором тысяча слов. И у меня есть инструменты для поиска слов, но искать слово можно, только если знаешь, что оно существует, и примерно помнишь, как оно пишется.

Это приводит к особому стилю программирования, и нужно некоторое время, чтобы программист привык так работать. Я видел множество Форт-программ, которые выглядят, как С-программы, буквально переведенные на Форт. Но смысл не в этом, а в том, чтобы работать совсем по-другому. Еще одна интересная особенность этого набора инструментов в том, что всякое новое определенное вами слово столь же эффек-

тивно и значимо, как слова, изначально определенные в ядре. Здесь нет никакой дискриминации.

Связана ли такая наглядная структура из множества коротких слов с реализацией Форт?

Чак: Она является результатом очень эффективной схемы вызова подпрограмм. Отсутствует передача параметров, потому что это стековый язык. Есть только вызов подпрограммы и возврат. Стек открыт. Машинный язык компилируется. Вход в подпрограмму и выход из нее – это буквально одна команда `call` и одна команда `return`. Кроме того, всегда можно опуститься ниже до эквивалента языка ассемблера. Можно определить слово так, чтобы оно выполняло машинные команды, а не вызывало подпрограммы, поэтому эффективность может быть такой же, как в любом другом языке, а то и повыше.

У вас отсутствуют накладные расходы вызовов Си.

Чак: Верно. Это очень расширяет возможности программиста. Если грамотно структурировать задачу, ее решение может быть не только эффективным, но и очень легко читаться.

С другой стороны, если сделать это плохо, может получиться код, непонятный никому, – вашему начальнику, например (если он хоть что-то понимает). И можно очень запутать дело. Так что это обоюдоострый меч: может получиться как очень хорошо, так и очень плохо.

Что бы вы сказали (или какой код продемонстрировали) разработчику, использующему другой язык программирования, чтобы вызвать у него интерес к Форту?

Чак: Опытного программиста очень трудно заинтересовать Фортом. Ведь он потратил силы на изучение своего языка/операционной системы и построил свою библиотеку для тех задач, которыми занимается. Рассказ о том, что на Форте все будет меньше, быстрее и проще, покажется недостаточно убедительным в сравнении с предстоящей необходимостью переписать весь код заново. Начинающий программист или инженер, которому нужно написать код, не сталкивается с таким препятствием и оказывается более благосклонным, как и опытный программист, начинающий работу над новым проектом в новых условиях, например, в моей ситуации – с многоядерными процессорами.

Вы сказали, что многие программы на Форте, которые вы видели, напоминают С-программы. Как правильно писать программы на Форте?

Чак: Снизу вверх.

Начать, видимо, придется с каких-то сигналов ввода/вывода, которые нужно генерировать, – вот и займитесь ими. Затем напишите код, который управляет генерацией этих сигналов. Потом вы поднимаетесь

выше, пока наконец не дойдете до слова самого верхнего уровня. Вы даете ему имя `go`, вводите с клавиатуры `go` – и понеслось.

Я не очень доверяю системным аналитикам, действующим в нисходящем направлении. Они определяют, в чем заключается задача, а потом разбивают ее на части так, что реализация бывает очень затруднена.

Проектирование, управляемое предметной областью, требует описать бизнес-логику с помощью лексикона заказчика. Есть ли связь между построением словаря и использованием технических терминов, принятых в предметной области?

Чак: Надо надеяться, что программист ознакомится с предметной областью, прежде чем начнет писать код. Обычно я беседую с заказчиком. Я слушаю, какие слова он употребляет, и пытаюсь их использовать, чтобы он мог понять, как работает программа. Форт дает возможность такого облегченного чтения благодаря своей постфиксной нотации.

Если бы я писал финансовое приложение, то, вероятно, в программе было бы слово «percent». Вы говорите «2.03 percent», и это 2,03 процента от аргумента, и все работает и читается самым натуральным образом.

Может ли проект, начатый во времена перфокарт, быть все еще полезен для современных компьютеров в эпоху Интернета? Форт проектировался на/для IBM 1130 в 1968 году. То, что он оказался превосходным языком для параллельных вычислений в 2007 году, поистине поражает.

Чак: Он развивался все это время. Но Форт – простейший из возможных компьютерных языков. Он никак не ограничивает программиста. Программист может определять слова, которые кратко отражают характеристики задачи в точном иерархическом виде.

Создавая программу, учитываете ли вы, что она должна хорошо читаться на английском языке?

Чак: Да, если это верхний уровень. Английский плохо подходит для описания функциональности. Он не был для этого предназначен, но у него есть такая же особенность, как у Форта: возможность определять новые слова.

Новые слова определяются в основном путем их объяснения с помощью ранее определенных слов. В естественных языках с этим могут быть проблемы. Если открыть словарь, то можно обнаружить, что определения часто образуют замкнутый круг, и ничего содержательного нельзя извлечь.

Слова вместо синтаксиса с разного вида скобками (как в C): можно ли проявить хороший вкус при написании программ на Форте?

Чак: Надеюсь, что да. Нужно, чтобы программист на Форте заботился о внешнем виде кода, а не только о его функциональности. Приятно, когда соседние слова составляют единый поток. Вот почему я разработал colorForth. Меня стал раздражать синтаксис Форта. Например, комментарий можно выделить с помощью круглых скобок.

Глядя на все эти символы пунктуации, я сказал себе, что должен быть какой-то лучший способ. Этот лучший способ оказался довольно накладным, поскольку к каждому слову исходного кода нужно было прикрепить тег, но когда я справился с возникшими издержками, то с удовольствием увидел, как вместо замысловатых символов появились разноцветные слова – по-моему, гораздо более приятный способ обозначить функциональность.

Меня постоянно критикуют те, кто не различает цвета. Они действительно возмущены тем, что я попытался лишить их права быть программистами, но кто-то в конце концов предложил вместо разных цветов использовать разные шрифты, что тоже хороший способ.

Главная особенность – 4-разрядный тег для каждого слова, что дает 16 вариантов действий, и компилятор может сразу определить, что от него требуется, а не догадываться об этом из контекста.

Языки второго и третьего поколений стремились к минимализму, например в реализациях с метациклическим интерпретатором. Форт – яркий пример минимализма в отношении конструкции языка и требуемой аппаратной поддержки. Это веяние времени или ваше достижение?

Чак: Нет, при проектировании намеренно ставилась цель получить минимальное по размеру ядро. Определить как можно меньше исходных слов, и пусть программист сам добавляет новые по мере надобности. Главной причиной была потребность в переносимости. Существовали уже десятки миникомпьютеров, а позднее появились десятки микрокомпьютеров. Я лично устанавливал Форт на многие из них.

Я стремился максимально облегчить задачу. На практике может существовать ядро примерно из сотни слов, которых достаточно, чтобы сгенерировать, так сказать, «операционную систему» (хотя это не совсем точно), в которой будет еще пара сотен слов. После этого можно писать приложения.

Я обеспечивал выполнение первых двух этапов и предоставлял прикладным программистам действовать на третьем, но нередко и сам писал приложения. Я определял слова, которые, по моему мнению, были необходимы. Первая сотня слов должна быть на машинном языке или ассемблере – во всяком случае, непосредственно взаимодействовать

с конкретной платформой. Вторые две-три сотни слов должны быть словами более высокого уровня, минимизирующими машинную зависимость от нижнего, ранее определенного уровня. После этого приложение становится практически машиннонезависимым, и его легко портировать с одного миникомпьютера на другой.

Легко ли вам было выполнять портирование начиная с этого второго уровня?

Чак: В высшей степени. Например, у меня был текстовый редактор, с помощью которого я писал исходный код. Обычно он переносился на другую машину без всяких изменений.

Поговаривают, что всякий раз, столкнувшись с новой машиной, вы немедленно начинали переносить на нее Форт.

Чак: Да. На самом деле, простейшим способом понять, как работает эта машина и каковы ее особенности, была оценка легкости реализации на ней стандартного набора слов Форта.

Как вы пришли к изобретению косвенного шитого кода?

Чак: Косвенный шитый (indirect-threaded) код – довольно тонкое понятие. Для каждого слова Форта есть запись в словаре. В косвенном шитом коде каждая запись указывает на код, который нужно выполнить, когда встретится это слово. Косвенный шитый код указывает на место, где находится адрес этого кода. Это позволяет получить другую информацию помимо адреса – например, значение переменной.

Вероятно, это было самое компактное представление слов. Доказано, что оно эквивалентно прямому шитому коду и подпрограммному коду. Конечно, в 1970 году эти понятия и термины не были известны. Но мне показалось, что это самый естественный способ реализовать широкий круг различных слов.

Какое влияние окажет Форт на компьютерные системы будущего?

Чак: Оно уже оказывается. Я уже 25 лет работаю с микропроцессорами, оптимизированными для Форта, и в самое последнее время – с многоядерным процессором, ядра которого представляют собой Форт-компьютеры.

Что дает Форт? Будучи простым языком, он довольствуется простым компьютером. 256 слов оперативной памяти, 2 стека, 32 команды, асинхронная работа, простая связь с соседями. Небольшой, с малым энергопотреблением.

Форт поощряет структурирование программ. В результате они хорошо удовлетворяют требованиям многоядерных процессоров для параллельной обработки. Большое количество простых программ побужда-

ет тщательно продумать каждую из них. И требуется при этом какой-нибудь 1% объема кода, который был бы написан в других случаях.

Слыша, как кто-то похваляется миллионами строк кода, я уверен, что этот человек вопиющим образом не разобрался в своей задаче. Нет в наше время задач, которые требовали бы миллионов строк кода. Зато есть небрежные программисты, плохие начальники и неоправданные требования к совместимости.

Запрограммировать на Форте множество небольших компьютеров – превосходное решение. Другие языки не обладают достаточной модульностью или гибкостью. В процессе уменьшения размеров компьютеров и налаживания взаимодействия между их сетями (smart dust – «умная пыль»?) формируется среда будущего.

Это похоже на одну из главных идей UNIX: множество взаимодействующих программ, каждая из которых делает что-то одно. Это по-прежнему лучшая конструкция? Не лучше ли вместо множества программ на одном компьютере иметь множество программ в сети?

Чак: Идея многопоточного кода, как она реализована в UNIX и в других ОС, была предтечей параллельной обработки. Но между ними есть существенные различия.

На большом компьютере можно позволить себе большие накладные расходы, обычно неизбежные при многопоточности. В конце концов, уже существует огромная операционная система. Но для параллельной обработки почти всегда чем больше компьютеров, тем лучше.

Когда ресурсы фиксированы, больше компьютеров значит – меньшие компьютеры. А маленькие компьютеры не могут позволить себе такие же накладные расходы, как большие.

Маленькие компьютеры должны объединяться – на одном чипе, на одной плате или беспроводными соединениями. У маленького компьютера маленькая память. Для операционной системы там нет места. Эти компьютеры должны быть автономными, с собственными средствами связи. Поэтому связь должна быть простой – никаких изощренных протоколов. Программы должны быть компактными и эффективными. Идеальная область для применения Форта.

Системы из миллионов строк кода отпадут. Они порождены большими централизованными компьютерами. Для распределенных вычислений нужен иной подход.

Язык, созданный для поддержки массивного синтаксического кода, способствует тому, что программисты пишут большие программы. Это поощряется и приносит удовлетворение. Нет факторов, побуждающих стремиться к компактности.

С помощью синтаксических языков можно писать компактный код, но обычно этого не происходит. Реализация обобщений, предполагаемых синтаксисом, приводит к неуклюжему и неэффективному объектному коду. Это неприемлемо для маленьких компьютеров. В правильно спроектированном языке исходный код переводится в объектный один к одному. Программисту очевидно, какой код будет сгенерирован из того, что он написал. Это служит источником удовлетворения иного рода, повышает эффективность и сокращает потребность в документации.

Одна из целей создания Форта – достижение компактности как исходного, так и двоичного кода, поэтому данный язык популярен среди программистов встраиваемых приложений. Но программисты во многих других областях могут обоснованно предпочесть другие языки. Есть ли в самом языке особенности, из-за которых исходный или конечный код может только пострадать?

Чак: Форт действительно компактен. Одна из причин – ограниченность синтаксиса.

В других языках можно наблюдать намеренное введение синтаксиса, создающего избыточность и возможность синтаксической проверки с целью выявления ошибок.

В Форте мало возможностей для обнаружения ошибок ввиду отсутствия избыточности. Благодаря этому исходный код становится компактнее.

Мой опыт работы с другими языками показывает, что большинство ошибок связано с синтаксисом. Разработчики языка будто намеренно дают возможность программистам делать ошибки, которые потом может выявить компилятор. Не думаю, что это продуктивно. Это только осложняет и без того трудный процесс написания корректного кода.

Примером служит контроль типов. Задание типов для разных чисел позволяет обнаруживать ошибки. Невольным следствием является то, что программист вынужден преобразовывать типы, порой стараясь обойти проверку типа, чтобы достичь своей цели.

Другое следствие синтаксиса – необходимость его пригодности во всех предположительных областях применения. В результате синтаксис усложняется. Форт – расширяемый язык. Программист может создавать собственные структуры, столь же эффективные, как предлагаемые компилятором. В результате не нужно заранее предвидеть и обеспечить все возможности, которые могут потребоваться в будущем.

Для Форта характерна постфиксная запись. Это упрощает компилятор и обеспечивает однозначную трансляцию исходного кода в объектный. Программист лучше понимает свой код, а код, полученный в результате компиляции, становится компактнее.

Защитники ряда новых языков программирования (в особенности Python и Ruby) указывают на читаемость как на важное преимущество. Насколько легки изучение Форта и поддержка программ на нем в сравнении с этими языками? Что могли бы другие языки позаимствовать у Форта в отношении легкости чтения?

Чак: Все компьютерные языки претендуют на легкость чтения. Без оснований. Возможно, тому, кто знает язык, кажется, что его легко читать, но новички всегда оказываются в трудном положении.

Проблема заключается в загадочном, произвольном и таинственном синтаксисе. Все эти скобки, амперсанды и прочее. Пытаешься выяснить, зачем они там нужны, и в итоге приходишь к выводу: низачем. Но эти правила приходится соблюдать.

А говорить на этом языке нельзя. Придется, как Виктор Борг, произносить все эти символы пунктуации.

Форт частично решает эту проблему, минимизируя синтаксис. Его тайные знаки @ и ! читаются как fetch (прочесть) и store (записать). Символы выбраны потому, что операции встречаются очень часто.

Программисту рекомендуется использовать слова естественного языка. Они соединяются без всякой пунктуации. Выбирая соответствующие слова, можно составлять разумные фразы. Есть даже стихотворения, написанные на Форте.

Другое достоинство – постфиксная нотация. Фраза вроде «6 inches» применяет оператор «inches» (дюймов) к параметру «6», что очень естественно. Прекрасно читается.

С другой стороны, программист должен составить словарь для описания задачи. Словарь может получиться довольно большим. Читателю кода нужно знать этот словарь, чтобы разобраться в программе. А программист должен постараться определить подходящие слова.

Как бы то ни было, чтобы прочесть программу, нужно приложить усилия. На каком бы языке она ни была написана.

Чем определяется успех вашей работы?

Чак: Красивым решением.

На Форте не пишут программы. Сам Форт – это и есть программа. Нужно добавить к нему слова, которые составят словарь для решения задачи. Оно становится очевидным, когда выбраны правильные слова, потому что позволяет интерактивно решать нужные аспекты задачи.

Например, я могу определить слова, описывающие электронную схему. Мне понадобится добавить эту схему в чип, показать структуру, проверить выполнение правил, запустить модель. Слова для всего этого со-

ставят приложение. Когда они правильно подобраны и составляют компактный и эффективный инструментарий, это и есть для меня успех.

Где вы учились писать компиляторы? Или в те времена этим все занимались по необходимости?

Чак: Я был в Стэнфорде в начале 1960-х, и там была группа старшекурсников, которые писали компилятор Algol для Burroughs 5500. Их, кажется, было всего трое или четверо, и я до глубины души был поражен тем, что три или четыре человека могут сесть и написать компилятор.

И я подумал, что раз они могут это сделать, то и я смогу – и сделал. Оказалось, что это не так трудно. В те времена компиляторы были окружены неким облаком таинственности.

Это и сейчас так.

Чак: Да, но в меньшей степени. Время от времени появляются всякие новые языки – не знаю уж, компилируемые или интерпретируемые, – и находятся люди хакерского склада, готовые этим заниматься.

Другая любопытная вещь – операционная система. Операционные системы чудовищно сложны и совершенно излишни. Билл Гейтс был гениален в том, что ему удалось убедить весь мир в необходимости операционных систем. Пожалуй, в истории человечества другого такого обмана не вспомнить.

Операционная система абсолютно не нужна вам. Если у вас есть подпрограмма, называемая драйвером дисков, и другая программа, обеспечивающая поддержку какого-то типа связи, то вам ничего больше не нужно от операционной системы. Windows тратит массу времени, работая с оверлеями, дисками и т. п., что никому не нужно. Сейчас есть гигабайтные диски и мегабайтная оперативная память. Мир изменился, и операционная система стала не нужна.

А как быть с поддержкой устройств?

Чак: Для каждого устройства есть подпрограмма. Нужна библиотека, а не операционная система. Вызовите нужные подпрограммы или загрузите их.

Трудно ли снова программировать после короткого перерыва?

Чак: Не вижу никаких проблем при коротком перерыве. Я сосредоточиваю все свое внимание на задаче, думая о ней днем и ночью. Наверно, это особенность Форта: полная отдача сил за короткое время (дни), пока задача не будет решена. Хорошо, что приложения Форта естественным образом распадаются на подпроекты. Обычно код на Форте прост и легко читается. Делая что-то действительно замысловатое, я пишу пространные комментарии. Хорошие комментарии помогают освежить в памяти задачу, но всегда нужно прочесть и понять код.

Какую самую крупную ошибку вы совершили в области программирования или проектирования? Какие уроки вы извлекли из нее?

Чак: Лет 20 назад я хотел создать инструмент для проектирования СБИС. На моем новом ПК не было Форта, поэтому я решил попробовать другой подход: с помощью машинного языка. Не ассемблера, а реальных шестнадцатеричных команд.

Я создал код так, как делал это в Форте: множество простых слов, иерархически взаимодействующих между собой. Получилось. Я пользовался им в течение десяти лет. Но с сопровождением и документацией были сложности. В конце концов я переписал программу на Форте, и она стала меньше и проще.

Я сделал вывод, что Форт эффективнее машинного языка. Отчасти благодаря интерактивности, отчасти – синтаксису. Удобная особенность кода на Форте – документирование чисел с помощью выражений, применяемых для их расчета.

Аппаратное обеспечение

Как следует разработчику рассматривать аппаратную платформу, на которой он работает, – как ресурс или как ограничение? Если как ресурс, то может возникнуть желание оптимизировать код, используя все возможности, предоставляемые аппаратурой; если как ограничение, то можно писать код в расчете на то, что с новым, более мощным аппаратным обеспечением он будет работать лучше, а при современных темпах прогресса этого не придется долго ждать.

Чак: Это очень верное наблюдение – что всякая программа рассчитана на определенное аппаратное обеспечение. Программы для ПК, несомненно, рассчитаны на появление более быстрых компьютеров, и потому им дозволена небрежность.

Но программы для встроенных систем предполагают сохранение аппаратуры в течение всего срока существования проекта. Из одного проекта в другой переходит лишь небольшая часть программного обеспечения. Поэтому аппаратная часть здесь ограничивает, но не абсолютно. В то же время для ПК аппаратное обеспечение является развивающимся ресурсом.

Переход к параллельной обработке может все изменить. Приложения, не способные выполняться на нескольких компьютерах, будут ограниченными, поскольку скорость отдельных компьютеров перестанет расти. Переделывать старые программы, чтобы оптимизировать их параллельное выполнение, невыгодно. А расчет на то, что положение спасут умные компьютеры, – бесплодные мечтания.

В чем главная проблема параллелизма?

Чак: Главная проблема параллельной обработки в скорости. Компьютер должен решать в приложении множество задач. Это можно осуществить на одном процессоре с помощью многозадачности. А можно одновременно выполнять задачи на нескольких процессорах.

Последнее осуществляется гораздо быстрее, и современное программное обеспечение нуждается в такой скорости.

Где кроется решение — в аппаратном обеспечении, в программном или в их сочетании?

Чак: Соединить несколько процессоров нетрудно. Таким образом, аппаратное обеспечение есть. Если программное обеспечение написано в расчете на него, то проблема решена. Но если можно переписать программу заново, то можно сделать ее настолько эффективной, что многопроцессорность не понадобится. Проблема в том, чтобы использовать многопроцессорные системы для старых программ, не меняя их. Задача создания таких интеллектуальных компиляторов не решена.

Меня поражает, что написанные в 1970-х программы нельзя переписать. Одна из причин может быть связана с тем, что в ту эпоху программирование было очень увлекательным: все делалось впервые, и программисты работали по 18 часов в день ради своего удовольствия. Сейчас программируют с 9 до 5 часов, в составе команды и по графику работы — удовольствия мало.

Поэтому пытаются добавить еще один уровень программного обеспечения, чтобы избежать переписывания старых программ. Во всяком случае, это интереснее, чем заново писать дурацкий текстовый редактор.

Современные компьютеры предоставили нам большие вычислительные мощности, но в какой степени эти системы действительно занимаются вычислениями? А в какой — просто перемещают данные и форматируют их?

Чак: Вы совершенно правы. По большей части компьютеры заняты перемещением данных, а не расчетами. Не просто перемещением, но сжатием и шифрованием. При больших скоростях передачи данных этим должны заниматься специальные схемы, поэтому непонятно, зачем вообще нужны компьютеры.

Следует ли что-то из этого? Может быть, нам нужно другое аппаратное обеспечение? Дональд Кнут однажды предложил: посмотрите, чем занимается ваш компьютер в течение отдельно взятой секунды. По его словам, результаты такого исследования могут значительно изменить наши представления.

Чак: В моем компьютере это учтено, и умножение выполняется просто и медленно. Оно используется нечасто. Зато важно, как происходит обмен данными между ядрами и с памятью.

С одной стороны — язык, дающий реальную возможность создавать свои словари, не слишком заботясь об аппаратной части. С другой стороны — очень маленькое ядро, тесно связанное с конкретной аппаратурой. Любопытно, каким образом Форт преодолевает этот разрыв. Правда ли, что на некоторых машинах у вас нет никакой операционной системы, кроме ядра Форта?

Чак: Форт совершенно автономен. Все, что нужно, есть в ядре.

Но это абстрагирует аппаратную часть для программистов на Форте.

Чак: Верно.

Что-то похожее делала машина Лиспа, но она никогда не стала популярной. Форт незаметно сделал то же самое.

Чак: Лисп не занимался вводом/выводом. Надо сказать, что Си тоже не занимался вводом/выводом, поэтому для него потребовалась операционная система. Форт занимался вводом/выводом с самого начала. Я не верю в наименьшее общее кратное. Я думаю, что когда вы беретесь за новую машину, она потому и новая, что делает что-то иначе, и вы хотите воспользоваться ее особенностями. Вам придется спуститься до уровня ввода/вывода, чтобы суметь это сделать.

Керниган и Ричи могли бы возразить, что для упрощения портирования им нужен был такой Си, который стал бы наименьшим общим кратным. Вы решили, что упростить портирование можно, не придерживаясь такого подхода.

Чак: Я подходил к этому стандартно. У меня было слово — кажется, `fetchp`, — которое получало 8 бит из порта. На разных компьютерах оно определялось по-разному, но в стеке оказывалась одна и та же функция.

Тогда в некотором смысле Форт эквивалентен Си со стандартной библиотекой ввода/вывода.

Чак: Да, но когда-то давно я работал со стандартной библиотекой Фортрана, и это было ужасно. Слова были совершенно не те. Она была очень громоздкой, неэкономной. Можно было с легкостью избавиться от лишней нагрузки готового протокола, определив полдюжины команд для операций ввода/вывода.

И часто вы занимались такими обходными путями?

Чак: С Фортраном — да. Когда имеешь дело, скажем, с Windows, деваться некуда. Система не подпустит тебя к вводу/выводу. Я почти намерен-

но сторонился Windows, но и без Windows процессор Pentium оказался самым сложным устройством для размещения Форта.

У него слишком много инструкций. И слишком много аппаратных функций вроде буферов ассоциативной трансляции и разного рода кэшей, которыми нельзя пренебречь. Пришлось с ними разбираться, и код инициализации для запуска Форта оказался очень сложным и громоздким.

Хотя он должен был выполняться единственный раз, мне все же пришлось потратить много времени, чтобы это делалось правильно. Но мы запустили на Pentium автономный Форт, так что труд был не напрасен.

Эта работа затянулась, наверное, лет на десять, отчасти потому, что пришлось гнаться за модификациями, которые делала Intel.

Вы сказали, что Форт реально поддерживает асинхронную работу. В каком смысле вы говорили об асинхронной работе?

Чак: В нескольких сразу. В Форте всегда была возможность мультипрограммности и многопоточности – средство под названием Cooperative.

Есть слово `pause`. Если некоторая задача доходит до такого места, где ей не нужно немедленно что-то делать, она должна сказать `pause`. Циклический планировщик даст компьютеру следующую задачу для выполнения.

Если не сказать `pause`, можно полностью монополизировать компьютер, но этого не произойдет, потому что это специальный компьютер. На нем выполняется единственное приложение, и все его задачи дружественны.

Думаю, в прежние времена все задачи были дружественны. Это один вид асинхронности, когда все эти задачи могли выполняться и заниматься своими делами без необходимости синхронизации. Опять-таки, одна из особенностей Форта состоит в том, что слово `pause` могло находиться в словах нижнего уровня. При каждой попытке чтения или записи диска слово `pause` выполнялось бы без вашего участия, потому что разработчики кода для работы с диском знали, что придется подождать конца выполнения операции.

В новых чипах – тех многоядерных чипах, которые я разрабатываю, – мы придерживаемся той же философии. Каждый компьютер работает независимо, и если на одном компьютере выполняется одна задача, а на соседнем – другая, они выполняются одновременно, но обмениваются одна с другой данными. Так же ведут себя задачи на многопоточном компьютере.

Форт лишь удачно выделяет эти независимые задачи. На самом деле, если это многоядерный компьютер, я, возможно, буду использовать немного другие программы, но я могу точно так же структурировать их для параллельного выполнения.

Был ли при кооперативной многопоточности у каждого потока свой стек и происходило ли переключение между ними?

Чак: При переключении задач иногда все, что требовалось (в зависимости от компьютера), это поместить слово на вершину стека и после этого переключить указатель стека. Иногда, действительно, приходилось копировать стек и загружать новый, но тогда я старался делать стек очень неглубоким.

Вы намеренно ограничивали глубину стека?

Чак: Да. Сначала стеки были произвольной глубины. У первого проектируемого мной чипа глубина стека была 256, потому что я боялся, что ее не хватит. У другого спроектированного мной чипа был стек глубиной 4. Сейчас я остановился на том, что глубина стека должна быть около 8 или 10, так что мой минимализм стал строже.

Я бы предположил, что тенденция была противоположной.

Чак: В моем приложении для проектирования СБИС действительно есть случай рекурсивной трассировки чипа, где мне пришлось увеличить глубину стека примерно до 4000. Для этого мог бы потребоваться стек другого типа – программно реализованный. Но на Pentium такой стек можно сделать и аппаратным.

Разработка приложений

Вы высказали мысль, что Форт – идеальный язык для объединения в сеть множества малых компьютеров, как в «умной пыли», например. Как вы считаете, для каких приложений могли бы с успехом использоваться такие малые компьютеры?

Чак: Несомненно, в области связи, и несомненно, в сборе данных. Но я только начинаю разбираться, как независимые компьютеры могут кооперироваться для выполнения сложных задач.

Многоядерные компьютеры, которые у нас есть, чрезвычайно малы. У них 64 слова памяти. Хотя можно считать, и что 128 слов: 64 RAM, 64 ROM. В каждом слове может храниться до четырех инструкций. В итоге у вас окажется в каждом компьютере не более 512 инструкций, поэтому задача должна быть достаточно простой. Спрашивается, если взять такую задачу, как, скажем, стек TCP/IP, то как разбросать ее по

множеству таких компьютеров, чтобы выполнялась нужная функция и при этом на каждом из компьютеров было не более 512 команд? Очень красивая проблема, за которую я как раз взялся.

Думаю, это касается почти любых приложений. Приложению гораздо проще работать, если оно разбито на независимые части, а не пытается делать то же самое последовательно на одном процессоре. Полагаю, это относится к генерации видео. Наверняка это справедливо для компрессии и декомпрессии изображений. Но я еще только учусь это делать. У нас в компании есть и другие люди, которые тоже учатся и получают от этого удовольствие.

Есть какая-то область применения, где этот подход не годится?

Чак: Конечно – для старых программ. Меня реально тревожит устаревшее программное обеспечение, и если вы хотите вернуться к этой проблеме, пожалуй, более естественно описать ее следующим образом. Мне кажется, что она тесно связана с тем, как мы представляем работу мозга с помощью независимых агентов Мински. Агент представляется мне маленьким ядром. Вполне возможно, что разум возникает в процессе взаимодействия между ними, а не как функция отдельного агента.

Старые программы – это недооцененная, но серьезная проблема. Со временем она будет лишь усугубляться – не только в банковской сфере, но и в аэрокосмической области и других отраслях. Проблема в том, что это миллионы строк кода. Их можно переписать заново, и это будут, скажем, тысячи строк на Форте. В машинном переводе смысла нет – это только увеличит размер кода. Но этот код невозможно проверить. Стоимость и риски будут ужасающими. Устаревший код может привести нашу цивилизацию к гибели.

Похоже, вы совершенно уверены в том, что в ближайшие 10–20 лет мы все чаще будем встречаться с тем, что программы образуются в результате свободного соединения множества мелких частей.

Чак: О да. Я уверен, что так и будет. Радиосвязь – чудесная вещь. Говорят, что в тело человека будут внедряться микроагенты, одни из которых будут служить датчиками, а другие – исполнительными механизмами, но связываться между собой они смогут только с помощью радио или звуковых волн.

Их возможности будут ограниченны – это всего несколько молекул. Поэтому это будет похоже на окружающий мир. Так организовано наше человеческое сообщество. Оно состоит из шести с половиной миллиардов независимых агентов, которые взаимодействуют между собой.

Плохой выбор слов может привести к плохо спроектированным и трудно сопровождаемым приложениям. Не приводит ли создание

крупного приложения из десятков и сотен мелких слов к появлению тарабарщины? Как этого избежать?

Чак: Практически никак. Я сам иногда обнаруживаю, что плохо подбираю слова. А тогда можно совсем запутаться. Помню, как в одном приложении я выбрал слово – сейчас уже забыл, какое – и потом изменил его, так что в итоге оно имело смысл, прямо противоположный звучанию.

Как если бы слово `вправо` сдвигало все влево. Это было ужасно. Какое-то время я боролся с этим, а потом переименовал слово, потому что совершенно невозможно было понять программу, пока это слово так затуманивало смысл. Я предпочитаю использовать английские слова, а не сокращения. Люблю писать их целиком. С другой стороны, я хочу, чтобы они были короткими. Короткие и осмысленные английские слова быстро кончаются, и приходится придумывать что-то другое. Терпеть не могу префиксы; это грубый способ создания пространств имен, чтобы многократно использовать одни и те же слова. Они мне кажутся отговоркой. Так легко различать слова, но неужели нельзя было придумать что-то поумнее!

Очень часто в приложениях на Форте есть отдельные словари, в которых можно повторно использовать слова. В одном контексте слово означает одно, в другом – другое. Когда я проектировал СБИС, от этих идей пришлось отказаться. Мне нужно было не меньше тысячи слов, и не просто английские слова, а названия сигналов и другое в этом роде, и мне быстро пришлось вернуться к определениям и словам со странным правописанием, а также к префиксам и подобным вещам. Такой код трудно читать. С другой стороны, там много слов вроде `nand`, или `nor`, или `xor`, обозначающих разные вентили. Я использую слова, где только можно.

Я вижу, как люди программируют на Форте. Не хочу сказать, что только я умею правильно программировать на Форте. Некоторые умеют придумывать очень хорошие имена, другие справляются с этим очень плохо. Одни создают очень хороший синтаксис, другие не придают этому значения. У одних получаются очень короткие определения слов, у других слово занимает полстраницы. Нет правил – есть только стилистические договоренности.

Кроме того, у Форта есть важное отличие от Си, Пролога, Алгола и Фортрана, связанное с тем, что обычные языки старались предусмотреть все мыслимые структуры и синтаксис, встроив их в язык. Из-за этого языки получались очень неуклюжими. Я считаю, что Си – очень неизящный язык со всеми своими квадратными и фигурными скобками, двоеточиями и точками с запятой и всем прочим. В Форте все это отсутствует.

Я не ставил целью решение общей задачи. Я лишь собирался сделать инструмент, с помощью которого кто-то другой мог бы решить любую задачу, вставшую перед ним. Возможность сделать любую вещь, а не возможность делать все вещи.

Нужно ли поставлять вместе с микрокомпьютерами исходный код, чтобы их можно было исправить даже десять лет спустя?

Чак: Верно, исходный код составил бы прекрасную документацию для микрокомпьютеров. Компактность Форта способствовала бы этому. Но дальше придется также включить компилятор и редактор, чтобы можно было посмотреть и модифицировать код микрокомпьютера без помощи другого компьютера или операционной системы, которые могут быть утеряны. colorForth и представляет мою попытку сделать это. Все, что требуется, это несколько килобайт исходного и/или объектного кода. Их легко записать во флеш-память и использовать в будущем.

Как связаны между собой конструкция языка и конструкция программ на нем?

Чак: Применение определяется его особенностями. Это видно на примере языков общения между людьми. Взгляните на различия между романскими (французский, итальянский), западными (английский, немецкий, русский) и восточными (арабский, китайский) языками. Они влияют на соответствующие культуры и мировоззрения. Из всех этих языков особой краткостью выделяется английский, и его распространенность все время растет.

То же касается языков для общения человека с машиной. Первые языки (Кобол, Фортран) были очень многословны. В последующих языках (Алгол, Си) был избыточный синтаксис. Такие языки неизбежно приводили к тому, что описания алгоритмов были большими и неизящными. На них можно было описать все, что угодно, но плохо.

Форт решает эти проблемы. Он в значительной мере лишен синтаксиса. Он способствует написанию коротких и эффективных определений. Он минимизирует потребность в комментариях, которые часто бывают неточными и отвлекают внимание от собственно кода.

В Форте также организован простой и эффективный вызов подпрограмм. Вызов подпрограмм в Си требует дорогостоящей подготовки и последующего восстановления. Это отталкивает от его употребления. Способствует созданию сложных наборов параметров, чтобы окупить стоимость вызова, но приводит к большим и сложным подпрограммам.

Эффективность вызова допускает разбиение программ Форт на множество мелких подпрограмм, что обычно и происходит. Для моего стиля характерны однострочные определения – сотни мелких подпрограмм.

В этом случае выбор используемых в коде имен приобретает большое значение – как для мнемоники, так и для читаемости программы. Когда код легко читается, меньше потребность в документации.

Благодаря отсутствию в Форте синтаксиса можно позволить себе вольности. Для меня это означает возможность проявить творчество и писать код, на который приятно смотреть. Некоторые рассматривают это как недостаток, мешающий контролю со стороны руководства и стандартизации. Думаю, вина в таких случаях лежит на руководстве, а не на языке.

Вы сказали, что большинство ошибок возникает из-за синтаксиса. Как избежать в программах на Форте других ошибок, например связанных с логикой, сопровождением и выбором плохого стиля?

Чак: Основные ошибки в Форте совершаются при работе со стеком. Можно случайно что-нибудь оставить в стеке и потом поскользнуться на этом. Очень важно связывать со словами комментарии к стеку. Нужно описать, что находится в стеке при входе и что – при выходе. Но это всего лишь комментарии. Полностью доверять им нельзя.

Иногда их реально выполняют для верификации и выяснения состояния стека.

В принципе, выход в разбиении. Если определение вашего слова занимает одну строку, можно прочесть его до конца, проследив поведение стека и проверив, что все правильно. Можно протестировать слово и убедиться, что оно работает так, как вы задумали, но при этом допустить ошибку со стеком. Слова `dup` и `drop` встречаются постоянно, и ими нужно корректно пользоваться. Очень большое значение имеет возможность выполнять слова вне контекста, просто задав для них входные параметры и посмотрев на выходные. Опять-таки, действуя в восходящем порядке, вы можете быть уверены, что все ранее определенные вами слова работают правильно, потому что вы их протестировали.

Кроме того, в Форте мало условных операторов. Есть конструкции `if-else-then` и `begin-while`. Моя философия, которую я стараюсь систематически распространять, состоит в том, что количество условных операторов в программе должно быть минимально. Вместо одного слова, которое проверяет некое условие, а потом выполняет либо одно, либо другое, лучше иметь два слова: одно выполняет это, другое – то, а вы решаете, которое из них применить.

В Си так не получится, потому что вызов обходится дорого, и стараются ввести такие параметры, чтобы одна и та же подпрограмма выполняла разные действия в зависимости от того, как ее вызвали. Отсюда все ошибки и осложнения в старых программах.

В попытках обойти недостатки реализации?

Чак: Да. От циклов не уйти. Циклы могут быть весьма и весьма полезны. Но в Форте, во всяком случае, в colorForth, циклы очень просты: у них один вход и один выход.

Что вы посоветуете новичкам, чтобы они могли программировать эффективно и с удовольствием?

Чак: Я наверняка не удивлю вас, сказав, что нужно учиться писать код на Форте. Даже если вы не собираетесь профессионально писать программы на Форте, поработав с ним, вы извлечете некоторые уроки, которые пригодятся вам, с каким бы языком вы ни стали работать. Если бы я писал программу на Си – а я почти не писал их, – то стал бы делать это в стиле Форта, с множеством простых подпрограмм. Даже если это потребует больше труда, вы облегчите поддержку.

Другой принцип – простота. Чем бы вы ни занимались – проектированием самолета или написанием программы, даже обычного текстового редактора, вы неизбежно начинаете добавлять все новые и новые функции, пока цена не станет неприемлемой. Лучше сделать несколько текстовых процессоров, ориентированных на разные рынки. Глупо писать электронное письмо с помощью Word: 99% его возможностей окажутся невостребованными. Для электронной почты нужен свой редактор. Когда-то такой был, но мода идет в другом направлении. Непонятно почему.

Будьте проще. Если вы разрабатываете приложение, если входите в команду проектировщиков, постарайтесь убедить остальных, что нужно стремиться к простоте. Не нужно прогнозировать. Не нужно решать задачу, которая, как вам кажется, может возникнуть в будущем. Решайте ту задачу, которая стоит сейчас. Стремление предугадать неэффективно. Вы рассчитываете на 10 событий, из которых в действительности случится одно, в результате кучу сил потратите впустую.

Каковы признаки простоты?

Чак: Мне кажется, что наука о сложности только зарождается, и одна из ее догм – измерение сложности. Мне нравится определение – не знаю, есть ли другие, – что из двух концепций проще та, у которой короче описание. Если вы описали нечто короче, значит, описали его проще.

Но здесь скрыт подвох, поскольку любое описание зависит от контекста. Вы напишете очень короткую программу на Си и будете считать, что она очень проста, но в действительности вы опираетесь на то, что есть компилятор Си, операционная система и компьютер, где все это будет выполняться. Поэтому в более широком контексте у вас окажется не простая, а довольно сложная вещь.

Я бы сравнил это с красотой. Невозможно определить ее, но, видя красоту, вы узнаете ее; простое – значит маленькое.

Как работа в команде влияет на программирование?

Чак: Групповую работу слишком превозносят. Первая задача группы – разбить задачу на относительно независимые части. Назначьте исполнителя для каждой части. Руководитель команды отвечает за стыковку отдельных частей между собой.

Иногда могут работать вместе два человека. Обсуждая задачу, они делают ее более понятной. Но интенсивный обмен информацией становится самоцелью. Групповое мышление не способствует творчеству. И когда несколько человек работают вместе, наверняка всю работу делает один.

Это происходит в любом проекте? Если требуется богатая функциональность, как в OpenOffice.org... ведь это достаточно сложно?

Чак: Такие вещи, как OpenOffice.org, разбиваются на подпроекты, каждый из которых программируется одним человеком, а общение поддерживается на уровне, необходимом для обеспечения совместимости.

По каким признакам вы узнаете хорошего программиста?

Чак: Хороший программист быстро пишет хороший код – корректный, компактный и читаемый. «Быстро» значит несколько часов или дней.

Плохому программисту нужно обсудить задачу, и он будет тратить время на планирование, вместо того чтобы писать код, и сделает своей профессией написание и отладку кода.

Что вы думаете о компиляторах? Не кажется ли вам, что они позволяют скрыть уровень мастерства программиста?

Чак: Компиляторы, пожалуй, худшие образцы кода. Их пишут те, кто никогда не писал компиляторов раньше и никогда не будет делать этого впоследствии.

Чем изощреннее язык, тем более сложным, изобилующим ошибками и непригодным для работы оказывается компилятор. Но простой компилятор для простого языка – важный инструмент, хотя бы для документации.

Редактор важнее компилятора. Большое разнообразие редакторов позволяет каждому программисту выбрать какой-то свой, что мешает успешному сотрудничеству. На этой основе процветает производство доморощенных трансляторов из одного формата в другой.

Другая беда авторов компиляторов – стремление использовать все специальные символы, имеющиеся на клавиатуре. Так клавиатуры никогда не станут проще и меньше. А исходный код становится непостижимым.

Но мастерство программиста не зависит от этих инструментов. Он может быстро освоиться со всеми их недостатками и начать писать хороший код.

Какой должна быть программная документация?

Чак: Я меньше придаю значения комментариям, чем это обычно принято. Причины:

- Краткие комментрарии обычно малопонятны. Приходится догадываться об их смысле.
- Пространные комментарии забивают код, в который помещены для пояснения. Бывает трудно найти код, связанный с комментарием.
- Часто комментарии плохо написаны. Программисты обычно не отличаются литературным талантом, особенно если английский не является для них родным языком. Жаргон и грамматические ошибки часто делают комментарии нечитаемыми.
- Самое главное, комментарии часто оказываются неточными. Код может измениться, а в комментариях это не отражается. Комментарии реже подвергаются критическому анализу, чем код. Неточные комментарии чреваты большими неприятностями, чем их отсутствие. Читателю приходится самому выяснять, где правда – в коде или в комментариях.

Комментарии часто бывают не по делу. Они должны пояснять цель кода, а не сам код. Излагать код другими словами бесполезно. А если еще и неточно, то это просто вводит в заблуждение. Комментарии должны рассказывать, для чего нужен этот код, что он должен делать и какими приемами это достигается.

В colorForth комментарии выделяются в затененный блок. Это отделяет их от кода и облегчает чтение. В то же время они легко доступны для чтения и обновления. Кроме того, размер комментариев при этом ограничивается размерами кода.

Комментарии не заменяют настоящую документацию. Для каждого модуля кода нужно написать поясняющий его текстовый документ. Он должен быть значительно подробнее комментариев и давать грамотное и полное объяснение.

Конечно, это делается редко, и не всегда для этого есть возможности, да и потерять документ, который отделен от кода, достаточно легко.

Цитирую по *http://www.colorforth.com/HOPL.html*:

«Вопрос патентования Форта долго обсуждался. Но поскольку патенты на программы являются спорными и могут потребовать участия Верховного суда, NRAO отказалась от преследования этой цели. По

этой причине права вернулись ко мне. Глядя назад, можно признать, что единственным шансом для Форта был общественный доступ. Так он и достиг расцвета».

Патенты на программы и сейчас вызывают споры. Осталось ли прежним ваше мнение о патентах?

Чак: Я никогда не был сторонником патентов на программы. Это похоже на патентование мыслей. А патентование языка/протокола особенно сомнительно. Язык может стать удачным только тогда, когда им пользуются. Все, что ограничивает его распространение, – глупость.

Вы считаете, что патентование технологии сдерживает или ограничивает ее распространение?

Чак: Торговать программами трудно, потому что их легко копировать. Компании изо всех сил стараются защищать свои продукты, в результате чего они иногда делаются непригодными для работы. Я считаю, что продавать нужно «железо», а программы отдавать бесплатно. Аппаратную часть трудно копировать, а с разработанным для нее программным обеспечением она становится дороже.

Патенты – один из способов решения этих проблем. Они доказали свою плодотворность для развития инноваций. Но необходимо соблюдать тонкую меру, отвергая необоснованные патенты и поддерживая преемственность с прежними методами/патентами. Кроме того, выдача патентов и их реализация сопряжены с огромными расходами. Недавние предложения по реформированию патентного законодательства несут в себе угрозу подавить личное изобретательство, отдав предпочтение большим компаниям. Это стало бы трагедией.

5

Бейсик

В 1963 году Томас Курц (Thomas Kurtz) и Джон Кемени (John Kemeny) придумали Бейсик (BASIC) – универсальный язык для обучения новичков программированию, а также для написания полезных программ опытными пользователями. Первоначально задача заключалась в том, чтобы абстрагироваться от аппаратной специфики. Этот язык получил широкое распространение после появления микрокомпьютеров в 1970-х; многие персональные компьютеры поставлялись со своими вариантами Бейсика. Впоследствии исчезла нумерация строк, используемая в операторах `GOTO`, но через Microsoft Visual Basic и Бейсик Курца многие поколения программистов познали радость составления программ на языке, поощряющем эксперименты и вознаграждающем любознательность.

Цели создания Бейсика

Как лучше всего учиться программированию?

Том Курц: Нужно, чтобы начинающим программистам не приходилось углубляться в руководства. Руководства, в большинстве своем, слишком многословны, чтобы удержать внимание учащегося. Для этого нужны простые задания по написанию кода, возможность простой реализации и множество примеров.

Некоторые преподаватели предпочитают учить языкам, для применения которых нужно накопить значительный опыт. Вы же решили создать язык, с которым может быстро начать работать программист любого уровня, а знания можно углублять по мере приобретения опыта.

Том: Да. Научившись программировать, легко осваивать новые языки. Самый сложный язык – первый. Если только новый язык не отличается какой-нибудь исключительной бестолковостью, он не слишком отличается от уже известных вам языков. Как и с естественными языками (которые изучать гораздо труднее): выучив первый романский язык, гораздо легче освоить второй. Прежде всего, всюду похожая грамматика, много одинаковых слов, а синтаксис достаточно прост (например, глагол находится в середине, как в английском, или в конце).

Чем проще первый язык, тем легче его изучить среднему студенту.

Решив создать Бейсик, вы руководствовались этим эволюционным подходом?

Том: Когда мы думали о том, чтобы создать Бейсик (я и Джон Кемени, примерно в 1962 году), я рассматривал возможность создания упрощенного подмножества Фортрана или Алгола. Из этого ничего не вышло. В большинстве языков программирования есть туманные грамматические правила, мешающие начинающим студентам. В Бейсике мы постарались обойтись без них.

Бейсик основан на нескольких принципах:

- Одна строка – один оператор.

  Можно было бы заканчивать оператор точкой, как, кажется в JOSS. Принятая в Алголе точка с запятой нас не удовлетворяла, как и продолжение строки в Фортране (`C`).

- Для адресации оператора перехода `GOTO` служат номера строк.

  Нам пришлось ввести нумерацию строк, потому что до появления редакторов WYSIWYG было еще очень далеко. Идея «меток операторов» нас не впечатляла. (Позднее, когда писать и редактировать программы стало проще, мы разрешили не пользоваться номерами

строк, если отсутствуют операторы GOTO, но к тому времени Бейсик был полностью структурирован.)

- Все арифметические операции выполняются с плавающей запятой.

  Новичкам сложнее всего усвоить различие между типами чисел – целыми и с плавающей запятой. Почти все языки программирования того времени шли на поводу у аппаратной архитектуры, где почти всегда были операции с плавающей запятой для инженерных расчетов и операции с целыми числами, которые выполнялись быстрее.

  Проводя все арифметические операции с действительными числами, мы оградили пользователя от проблем числовых типов. Внутри самого языка нам пришлось пойти на некоторые сложности, потому что иногда требуется целое число (например, индекс массива), а пользователь может задать не целое (скажем, 3.1). Тогда мы просто выполняли округление.

  Аналогичные проблемы возникали из-за неравенства десятичных и двоичных дробей. Например, в операторе

  ```
  FOR I = 1 TO 2 STEP 0.1
  ```

  десятичная дробь 0.1 представляется бесконечной периодической двоичной дробью. Чтобы определить, когда нужно завершить цикл, нам пришлось учитывать погрешность измерений.

  (Некоторые проблемы соответствия двоичного и десятичного представлений были решены не в первоначальном Бейсике, а в гораздо более позднем True BASIC.)

- Число – это число (is a number).

  Никаких требований к формату при вводе числа в коде или операторе data. Команда PRINT выводит результат в установленном по умолчанию формате. Оператор FORMAT и его эквиваленты в других языках весьма трудны для освоения. Начинающий пользователь может недоумевать, зачем ему нужно все это учить, когда он хочет получить простой ответ!

- Разумные значения по умолчанию.

Если есть какие-то сложности для «более продвинутого» пользователя, то начинающему они не должны быть видны. Можно согласиться, что в первоначальном Бейсике было не так много «продвинутых» функций, но идея по-прежнему верна.

Правильность нашего подхода подтвердилась тем, что обучить новичка писать простые программы на Бейсике можно было примерно за час. Сначала наш курс состоял из четырех одночасовых лекций, потом он

был сокращен до трех, потом до двух, и наконец всего до пары видеолент.

Я пришел к выводу, что вводный курс компьютерных наук можно преподавать с использованием определенного варианта Бейсика (не того, который был изначально, а с добавлением конструкций структурного программирования). Единственное, чего нельзя было сделать, так это ознакомить слушателей с понятием указателей и выделения памяти!

Другой аспект. В прежние времена для выполнения программы требовалось пройти несколько этапов: компиляцию, компоновку и загрузку, запуск. Мы решили, что в Бейсике *любой* прогон программы будет включать все эти шаги, так что пользователь даже не узнает об их существовании.

В те времена большинству языков программирования требовался многопроходный компилятор, требующий слишком много ценного машинного времени. Поэтому мы компилировали один раз, а выполняли многократно. Но маленькие учебные программы компилировались и выполнялись сразу. Это потребовало от нас разработки однопроходного компилятора и перехода прямо к выполнению, если на этапе компиляции не было ошибок.

Кроме того, выводя для учащегося сообщения об ошибках, мы ограничивали их число пятью. Я помню, как Фортран распечатывал кучу страниц, подробно показывая *все* синтаксические ошибки в программе, обычно возникавшие из-за какого-нибудь пропущенного символа пунктуации в ее начале.

Я видел руководство по Бейсику выпуска 1964 года. В его подзаголовке сказано: «Простой алгебраический язык для системы с разделением времени в Дартмуте». Что имелось в виду под «алгебраическим языком»?

Том: Так как мы оба были математиками, естественно, что некоторые вещи в языке имели математический вид, например возведение чисел в степень и тому подобное, и мы ввели математические функции, такие как синус и косинус, поскольку предполагали, что с помощью программ на Бейсике студенты будут делать расчеты. У нас был явный уклон в сторону численных расчетов – в отличие от других создаваемых тогда языков, таких как Кобол, перед которыми ставились другие задачи.

Мы тогда присматривались к Фортрану. Доступ к Фортрану на любой большой машине IBM осуществлялся с помощью перфокарт, имевших 80 колонок. У себя в колледже для ввода данных мы подключили к компьютеру телетайпы, совместимые с телефонными линиями, потому что

хотели, чтобы терминалы в разных точках кампуса соединялись с центральным компьютером телефонными линиями. Это осуществлялось с помощью аппаратуры, изначально предназначенной для связи, например с помощью телетайпов, хранения и передачи сообщений и т. п. Так что мы отказались от перфокарт.

Во-вторых, мы хотели избавиться от ограничений перфокарт (данные должны располагаться в определенных колонках) – мы хотели, чтобы формат был более или менее свободным и позволял печатать на клавиатуре телетайпа – обычной клавиатурой «qwerty», только с прописными буквами.

Так появился формат языка, в котором было легко печатать и который поначалу игнорировал пробелы. Не было никакой разницы, вводили вы пробелы или нет, потому что язык тогда был устроен так, что компьютер правильно интерпретировал введенные данные независимо от наличия пробелов. Так было сделано, потому что многие, особенно преподаватели, не слишком хорошо печатали.

Игнорирование пробелов проникло и в некоторые ранние версии Бейсика для персональных компьютеров, что приводило к забавным сюрпризам при интерпретации данных, введенных пользователем.

В Dartmouth BASIC никаких неясностей не возникало. Только спустя годы развитие языка потребовало пробелов: имя переменной должно было оканчиваться пробелом или символом.

Один из критиков Бейсика говорил, что этот язык пригоден только для обучения: как только начинаешь писать большие программы, они превращаются в хаос. Что вы скажете по этому поводу?

Том: Так мог сказать только тот, кто не следил за развитием Бейсика все эти годы. Я сам писал на True BASIC программы длиной в 10 и даже 20 тысяч строк, а можно было бы и больше – в 30 или 40 тысяч, – и никаких проблем не возникает, и на скорости выполнения это не отражается.

Реализация языка и его конструкция – разные вещи.

Конструкция языка определяет, что должен вводить пользователь, чтобы решить свою задачу. Если разрешено создавать библиотеки, можно делать все, что угодно. Поддержка программ произвольного размера закладывается в реализацию языка, и в True BASIC она есть.

В других версиях Бейсика это не так. Например, в Microsoft Basic и основанном на нем Visual Basic есть некоторые ограничения. В других общедоступных версиях Бейсика есть свои ограничения, но они связаны с реализацией, а не с конструкцией языка.

Какие характеристики True BASIC позволили вам писать большие программы?

Том: Только одна: инкапсуляция, модули. Мы называем наши инкапсулирующие структуры *модулями*.

Фактически они были стандартизированы Комиссией по Бейсику перед тем, как та прекратила деятельность. Это произошло в первые годы существования True BASIC. Данная функция была введена в стандарт в 1990 или 1991 году.

В современных компьютерах столько памяти и такие быстрые процессоры, что такие вещи реализуются без труда.

Даже если опять вернуться к двухпроходной компиляции?

Том: Редактор связей тоже написан на True BASIC. Фактически это грубая версия True BASIC, или упрощенная версия True BASIC. Она компилируется в B-код подобно тому, как Паскаль компилируется в P-код.

Для фактической компоновки выполняется этот B-код, и есть очень быстрый интерпретатор, выполняющий инструкции B-кода. True BASIC, как и первоначальный Dartmouth BASIC, компилируется. Dartmouth BASIC компилировался прямо в машинные инструкции за один проход. В True BASIC результатом компиляции является B-код, который очень прост, поэтому выполнение B-кода, происходящее сейчас в очень быстром цикле, написанном на Си, первоначально было реализовано для DOS-платформ.

Все происходит очень быстро. Не так быстро, как в языке, специально ориентированном на скорость, но все равно чрезвычайно быстро. Как я сказал, B-код состоит из двухадресных команд, отсюда и быстрота.

В прежние времена интерпретация не приводила к замедлению, потому что все операции с плавающей запятой приходилось выполнять программно. Мы настояли на том, чтобы в True BASIC и оригинальном Dartmouth BASIC действия всегда производились с числами с двойной точностью, поэтому в 99% случаев пользователям не приходилось заботиться о точности. Сейчас, конечно, мы применяем стандарт IEEE, который автоматически поддерживают все процессоры.

Вы считаете, что язык, предназначенный для обучения, отличается от языка для профессионального программирования тем, что первый легче изучить?

Том: Нет, просто таков был ход развития. В должное время появился Си, чтобы дать доступ к аппаратному обеспечению. Сейчас мы всюду видим объектно-ориентированные языки, которые преподаются и на которых пишут профессионалы. Они являются производными соответствующей среды, и потому их так трудно изучать.

Дело в том, что те, кто пользуется этими производными языками, имеют профессиональную подготовку и работают в группе программистов, могут строить значительно более сложные приложения, например для создания видео, обработки звука и т. п. Гораздо легче делать такие вещи с помощью объектно-ориентированных языков типа Objective-C, но если перед вами стоит другая задача и нужно просто написать большую прикладную программу, то можно воспользоваться True BASIC, который происходит от Dartmouth Basic.

К чему мы стремимся, упрощая работу с языком программирования? Появится ли настолько простой язык, что любой пользователь компьютера сможет писать собственные программы?

Том: Нет, и многое из того, что мы сделали в Dartmouth BASIC, теперь можно осуществить с помощью других приложений, например электронных таблиц. Электронные таблицы позволяют выполнять довольно сложные расчеты. Кроме того, некоторые математические приложения, которые мы тогда задумывали, теперь можно реализовать с помощью программных библиотек, предлагаемых профессиональными сообществами.

Детали языка программирования не имеют большого значения, потому что новый язык можно выучить за день. Легко выучить новый язык, когда есть приличная документация. Не понимаю только, почему каждый новый язык претендует на идеальность. Правильный язык может существовать только для каждой конкретной области, и никак иначе. Ну какой из всех мировых языков лучше всего подходит для устной и письменной речи? итальянский? английский? какой-то еще? Вы можете мне сказать? Нет, потому что все устные и письменные языки связаны с тем, как устроена жизнь в том месте, где они используются, поэтому идеальных языков не существует. Точно так же не существует идеальных языков программирования.

Вы рассчитывали, что каждый, написав несколько сотен маленьких программ, станет считать себя программистами?

Том: Такова была наша цель, но необычным оказалось то, что по мере развития языка и без особого его усложнения стало возможным писать программы длиной в 10 000 строк. Это произошло потому, что мы старались сохранить простоту. Собственно смысл и фокус с разделением времени заключаются в том, что весь цикл завершается очень быстро, и не нужно беспокоиться о том, как оптимизировать программу. Оптимизировать нужно личное время.

У меня был случай, когда за несколько лет до изобретения Бейсика я работал над программой для MIT. Это была программа на символическом языке ассемблера (SAP) для IBM 704. Я старался написать эту

программу и воплотить все возможное, и я использовал световые индикаторы, чтобы оптимизировать программу и не выполнять одни и те же вычисления лишний раз. Я сделал все, от меня зависящее. Оказалось, что эта чертова штука не работает, и мне понадобился месяц, чтобы это выяснить, потому что я выходил на машину раз в две недели. Цикл между запуском и получением результата составлял две недели.

Не знаю, сколько минут или часов машинного времени это заняло. На следующий год появился Фортран, и я перешел на него – и написал программу на Фортране, потратив, наверное, всего минут пять машинного времени.

Вся процедура оптимизации и кодирования поставлена совершенно неверно. Не нужно этим заниматься. Вы должны выполнять оптимизацию только тогда, когда это действительно необходимо, откладывая ее до последнего. Языки высокого уровня оптимизируют машинное время автоматически, потому что с ними вы делаете меньше ошибок.

Такую точку зрения встретишь не часто.

Том: Компьютерные теоретики плохо разбираются в этом вопросе. А мы, компьютерные программисты, заняты сложными и интересными мелкими деталями программирования и не способны взглянуть на оптимизацию системы в целом с широкой инженерной точки зрения. Все пытаемся оптимизировать на уровне битов и байтов.

Во всяком случае, это лишь мое личное мнение. Сомневаюсь, что смог бы его обосновать.

Повлияла ли эволюция аппаратного обеспечения на эволюцию языков?

Том: Нет, потому что мы считали, что язык должен ограждать от необходимости знать аппаратное устройство. При разработке Бейсика мы сделали его независимым от аппаратного обеспечения: ни в самом языке, ни в появившихся в нем позднее функциях нет ничего, что как-то отражало бы аппаратуру.

Иное положение с некоторыми ранними версиями Бейсика для персональных компьютеров, очень слабо связанными с тем, чем мы занимались в Дартмуте. Например, в одной из таких версий были средства для записи или чтения определенных адресов памяти. В нашем Dartmouth BASIC такого никогда не было. Естественно, что такие версии Бейсика для персональных компьютеров оказывались крайне зависимыми от аппаратного обеспечения и конструкция этих языков отражала доступную для них аппаратуру.

Если спросить тех, кто делал Microsoft Basic, то они скажут, что да, на характеристики языка повлияла аппаратная часть. Но в Dartmouth BASIC дела обстояли иначе.

Чтобы облегчить жизнь пользователю, вы решили выполнять все арифметические операции как операции с числами с плавающей запятой. А как насчет работы с числами в современных языках? Не следует ли нам перейти к точной форме представления в виде чисел произвольной точности, которые можно рассматривать как некие «массивы цифр»?

Том: Есть множество способов представлений чисел. Надо сказать, что большинство языков в прежние времена, как и в нынешние, учитывает возможности представления чисел современным аппаратным обеспечением.

Например, если вы станете сегодня программировать на Си, то будете оперировать с числами, соответствующими возможностям аппаратного обеспечения по представлению чисел, т. е. это будут числа с плавающей запятой одинарной точности, двойной точности, целые числа одинарной точности, двойной и т. д. Все это учитывает Си, который был создан для доступа к аппаратуре, поэтому пришлось обеспечить доступ к тем представлениям чисел, которые поддерживает компьютер.

Далее, какие числа могут представлять компьютеры? Для последовательностей двоичных цифр фиксированной длины – а это то, с чем работает большинство компьютеров, – возможно лишь конечное число десятичных цифр, что ограничивает представляемые числа и их типы и ведет к известным ошибкам округления.

Есть языки, предоставляющие неограниченную точность, например 300 десятичных цифр, но они это делают программным образом, представляя очень большие числа как массивы цифр неограниченной длины, но раз все выполняется программно – значит очень медленно.

Наш подход в Бейсике состоял просто в том, чтобы считать число числом: «3» – это число, и «1,5» – тоже число. Мы избавили своих студентов от необходимости учитывать такую разницу; если они вводили какое-то число, мы старались создать его представление в аппаратной части нашей машины, оперирующей числами с плавающей запятой.

Нужно еще заметить, что когда мы решали вопрос о выборе компьютера (в 1964 году мы, конечно, остановились на компьютере GE), то настаивали на том, чтобы этот компьютер мог аппаратно выполнять операции с плавающей запятой, потому что не хотели возиться с программированием арифметики, а числа у нас были именно такими. Конечно, в такое

представление всегда заложена некоторая потеря точности, но с этим приходится мириться.

Диктовался ли выбор операторов GOTO и GOSUB ограничениями тогдашнего аппаратного обеспечения? Нужны ли они в современных языках программирования?

Том: Не думаю, что это вопрос аппаратуры: она не имеет к нему отношения.

Это требовалось в некоторых структурированных языках, но то было в далекие времена, 20 или 30 лет назад. Не думаю, что здесь есть проблема.

Тогда это было важно, потому что так писали компьютерные программы на машинном языке и ассемблере. Когда мы создавали Бейсик, идея структурного программирования еще не возникла. Кроме того, мы строили Бейсик по образцу Фортрана, а в Фортране оператор `GOTO` есть.

Развивая дальше Бейсик, какими критериями вы руководствовались, добавляя в язык новые функции?

Том: Ну, мы добавляли то, что тогда требовалось, не особенно теоретизируя.

Например, одной из первых после того, как в начале 1964 года появился на свет Бейсик, мы добавили возможность работы с нечисловой информацией – строками символов. Мы добавили символьные строки, чтобы, например, в игровых программах можно было вводить не «1» и «0», а «yes» и «no». Изначально в Бейсике «1» означало «да», а «0» – «нет», но довольно скоро мы добавили возможность работы с символьными строками. И это было продиктовано необходимостью.

Разработка компилятора

Создавая первую версию Бейсика, вы смогли написать однопроходный компилятор, тогда как все другие писали многопроходные. Как вам это удалось?

Том: Очень легко, поскольку язык устроен относительно просто. Есть много языков, простых в этом отношении. Все было известно, и единственным, что мы не могли сделать сразу и что требовало так называемых полутора проходов, была настройка переходов вперед. Только это мешало нам организовать полностью однопроходный компилятор.

Скажем, в первой сотне строк программы есть оператор GOTO, передающий управление на тысячу строк вперед. Тогда нужен этап компоновки.

Том: Что мы и сделали. Это было равносильно связному списку. То есть на самом деле мы не применяли структуру связного списка в ассемблере компьютера, на котором работали, но в принципе это было то же самое. Например, это могла быть таблица с адресами, которые мы вводили позднее.

Могли ли вы тогда одновременно выполнять синтаксический анализ и генерировать код?

Том: Да. Помимо этого мы намеренно упростили язык, чтобы сделать возможным однопроходный синтаксический анализ. Иными словами, имена переменных были жестко ограничены. Буква или буква с цифрой, а имена одно- или двумерных массивов всегда состояли из одной буквы, за которой следовала круглая скобка. Разбор был простейшим. Не было справочной таблицы, и более того, мы воспользовались простым приемом, поскольку одна буква или буква с цифрой дают нам 26×11 имен переменных. Мы выделили память, фиксированное пространство адресов, по которым находились значения этих переменных, если у них были значения.

Нам даже не понадобилась таблица символов.

Нужны ли были объявления переменных?

Том: Совершенно не нужны. На самом деле, массив всегда представлял собой букву, за которой следовала левая скобка, что фактически и было объявлением. Попробую вспомнить точнее. Если вы использовали массив, например `a(3)`, то тогда он автоматически становился массивом – дайте вспомнить, кажется, от 0 до 10. Иными словами, автоматическое определение по умолчанию с нумерацией от 0, потому что когда математик пишет коэффициенты многочлена, первый из них имеет индекс 0.

Легко ли было это реализовать?

Том: Реализация была тривиальной. На самом деле, в написании компиляторов есть много совсем нетрудных вещей. Даже позднее, когда появилась более развитая версия Бейсика, где использовалась справочная таблица символов, она тоже не потребовала больших усилий.

Трудности возникают с оптимизацией.

Том: Нас не волновала оптимизация, потому что 99% всех программ, которые в то время писали студенты или сотрудники факультета, были крохотными и простейшими. Не было никакого смысла их оптимизировать.

Вы сказали, что полиморфизм влечет интерпретацию на этапе исполнения.

Том: Полагаю, это так, но никто не оспорил это мое утверждение, потому что я ни с кем его не обсуждал. Полиморфизм означает, что вы пишете такую программу, которая по-разному ведет себя в зависимости от полученных ею данных. Так вот, если не ввести это в исходную программу, то на этапе исполнения эта часть программы не будет знать, чем она занимается, пока исполнение не дойдет до нее, что я и считаю интерпретацией на этапе исполнения. Или я не прав?

Возьмем Smalltalk с предположительно доступным исходным кодом. Если вы действительно поздно принимаете решения по связыванию, можно ли это считать связыванием на этапе исполнения?

Том: Это сложный вопрос. Есть раннее связывание, позднее связывание и связывание на этапе исполнения. Это действительно непросто, и я надеюсь, что найдется способ с этим справиться.

Допустим, вы пишете программу сортировки. Если сортируются числа, то решить, какое из чисел меньше и т. п., достаточно просто. Если сортируются символьные строки, решение становится менее очевидным, потому что неизвестно, как сортировать – в порядке кодов ASCII, по алфавиту или как-то еще.

Если вы пишете программу сортировки, то знаете, какая сортировка вам нужна, и соответствующим образом организуете сравнение. Если вы пишете универсальную процедуру сортировки, то нужно вызывать подпрограмму или делать что-то аналогичное, чтобы выяснить, какой элемент (A или B) меньше в зависимости от его природы. Если вам нужно сортировать ключи записей или чего-то еще, вы должны знать, как упорядочивается то, что вы сортируете. Элементы могут иметь разную природу. Это могут быть символьные строки или что-то другое. Вы пишете алгоритм сортировки, не зная этих деталей, потому что они появятся позже. Если это происходит на этапе исполнения, то тогда, конечно, это интерпретация на этапе исполнения.

Поймите меня правильно, это можно делать эффективно, потому что можно создать маленькую подпрограмму или функцию, и все, что требуется пользователю, это чтобы там были заданы правила для упорядочивания элементов, которые он сортирует. Но это не происходит автоматически.

Полиморфизм не получается сам собой. Приходится писать полиморфные варианты.

Том: Кому-то нужно об этом позаботиться.

Другая тема разговоров для сторонников объектно-ориентированного программирования – это наследование. Оно имеет значение, только

если в языке есть типы данных. Я читал введения нескольких книг по объектно-ориентированному программированию, и всюду говорится о том, как некто пишет программу, рисующую круг. Возможно, кто-то применит ее для своих целей, но это очень маловероятно. Меня всегда смущало в подобных вещах то, что если хочешь написать программу, которая будет достаточно универсальной, чтобы ее захотел использовать кто-то другой, то нужно описать ее до малейших деталей и обеспечить ее доступность. Я хочу сказать, что здесь возникает уйма вопросов. Это почти то же самое, что написать законченное приложение, с документацией.

Для того рода программ, которые пишу я, это лишнее. Не знаю, что происходит в промышленности. Это другой вопрос.

Вы считаете идею дешевого и простого многократного использования кода скороспелым обобщением?

Том: В сущности, в наше время большинство людей не пишет программ. Часто то, для чего мы писали программы, сейчас можно сделать с помощью имеющегося в продаже приложения, или с помощью электронной таблицы, или как-то еще. Непрофессиональные программисты, люди, специализирующиеся в других областях, теперь значительно реже пишут программы.

Что меня тревожит в преподавании программирования – главным образом, в средней школе с углубленным изучением компьютерных наук, – так это чрезмерная сложность. Не знаю, какие языки там сейчас преподают, не интересовался.

Я когда-то попробовал составить первый курс колледжа по компьютерным наукам на основе Бейсика. Он позволял делать практически все, что я хотел бы включить в начальный курс компьютерных наук, за исключением указателей и распределения памяти. Это довольно сложная тема. Если брать в качестве языка Паскаль, возможно, придется влезать в указатели и распределение памяти, когда у людей даже нет понятия о том, что такое компьютерная программа, а это никуда не годится. Я никогда не навязывал свои взгляды. Здесь я в меньшинстве.

Ширится уверенность в том, что больше не нужно заниматься распределением памяти и указателями, если только вы не пишете виртуальную машину. Тем, кто пишет компиляторы, нужно, но это их работа.

Том: Пусть этим занимается компилятор, а не вы сами.

В True BASIC мы получили портируемость. Разработка была выполнена двумя блестящими молодыми людьми. Я работал в компании и занимался прикладным программированием. Они придумали промежу-

точный язык на манер P-кода в Паскале. Вместо двух адресов там было три, потому что, как оказалось, в Бейсике почти все команды трехадресные, например `LET A = 3`. Здесь три части: код операции и два адреса. Потом они сделали компилятор с помощью самого Бейсика, создав очень сырую поддержку, чтобы скомпилировать сам этот компилятор. Сам компилятор написан на True BASIC и выполняется на любой машине, где есть движок True BASIC, который мы называем интерпретатором.

Интерпретирование языка осуществляется на уровне исполнения, а не чтения. Таким образом, выполнение программы происходит в три этапа. Первый этап – компиляция, второй – компоновка/загрузка и третий – исполнение. Но пользователь всего этого не видит. Он просто вводит `run` или нажимает кнопку, и все происходит в один миг.

Скомпилированный код является также машинно-независимым. Он переносим на другие платформы.

На практике это довольно сложная языковая среда. Мы работали на разных платформах, в какое-то время их было четыре или пять, но большинство из них приказали долго жить. Сейчас осталось две основных платформы, может быть – три: UNIX, Microsoft и Apple, которая интересна нам, потому что Дартмут всегда сотрудничал с Apple.

Перенос на эти платформы оказался «еще той» проблемой. Управление окнами, графические элементы, кнопки и все прочее всюду свои, и приходится детально разбираться в том, как они действуют. Иногда они работают на очень низком уровне, и приходится строить это все самому.

На старых Маках было средство `Mac toolbox`. Некоторое время мы использовали программное обеспечение промежуточного уровня XVT, производимое в Боулдере (Колорадо), которое должно было решить проблемы с Windows и классическим Маком. Здесь мы кое-чего достигли. Прежде чем компания прекратила свое существование, программист успел написать версию для Windows, которая выходит прямо на прикладной уровень Windows.

Беда в том, что поскольку всеми этими вещами у нас занимается один программист, времени уходит много, выходят новые версии операционной системы, и обнаруживаются новые ошибки, которые нужно исправлять. Такой маленькой организации, как наша, было почти невозможно с этим справиться. В какой-то момент у нас было три программиста, потом их стало двое, и наконец остался один. Для одного программиста тут слишком много дел.

Базовый код, который теперь написан в основном на Си, содержит уйму `#ifdef`.

Язык и практика программирования

Как связаны конструкция языка и конструкция программ на этом языке?

Том: Очень тесно. Большинство языков разрабатывается с оглядкой на конкретный тип программ. Ярким примером служит APT (Automatic Programmed Tools – язык для станков с числовым программным управлением).

На ранней стадии разработки вы ввели оператор REM для создания комментариев. Изменилось ли с годами ваше представление о комментариях и документировании программ?

Том: Нет, это своего рода механизм самозащиты. Когда я пишу программы на True BASIC, то добавляю в них комментарии, чтобы напоминать себе, о чем я думал, когда писал этот код. Поэтому я считаю, что комментарии играют свою роль, но эта роль различна в зависимости от того, какие программы вы пишете, работаете ли вы в группе или никто больше не будет читать ваши комментарии. Комментарии полезны, но только там, где они необходимы.

Можете ли вы что-то посоветовать тем, кто разрабатывает ПО в составе команды?

Том: Нет, потому что мы никогда этим не занимались. Все сделанные нами программы – это труд одиночек. Пожалуй, при работе над True BASIC у нас было два или три человека, писавших код, но в действительности они работали над отдельными проектами. У меня просто нет опыта работы в команде.

У вас была машина с разделением времени, и вы советовали пользователям планировать свой сеанс работы, прежде чем садиться за телетайп. Девиз был «нажатие клавиш не отменяет необходимости думать». Верен ли он сегодня?

Том: Пожалуй, думать надо. Если крупная компания собирается разработать новый программный продукт, она глубоко продумывает его заранее, как мне кажется.

Лично я стараюсь не обдумывать все заранее, а просто сажусь и начинаю писать программу. Потом оказывается, что это не совсем то, что нужно, и тогда я все выкидываю и начинаю сначала. Это эквивалентно размышлению. Обычно я начинаю писать код, просто чтобы узнать, какие возникнут проблемы, а потом выкидываю этот вариант.

Думать над тем, что ты делаешь, очень и очень важно. Не уверен, но, кажется, это Ричард Хэмминг постулировал, что «нажатие клавиш не отменяет необходимости думать». То было на заре компьютерной эпохи,

очень немногие разбирались в компьютерах, и подобного рода советов было полно.

Как правильнее всего изучать новый язык программирования?

Том: Если человек умеет программировать и понимает основные идеи (например, как распределять память), изучить новый язык легко, если только есть справочная документация и хорошая реализация (т. е. компилятор). Я сам делал это много раз.

Учеба на курсах обычно пустая трата времени.

Любому программисту, оправдывающему свое звание, приходится в течение своей карьеры изучить несколько языков. (Я в своей жизни использовал их штук 20.) Изучать новый язык нужно, читая руководство. За редким исключением, все языки программирования схожи по структуре и способу функционирования, поэтому новый язык достаточно легко изучить, если есть приличное руководство.

А когда вы разберетесь с жаргоном (что значит *полиморфизм*?), все становится совсем легко.

Одна из проблем современного стиля программирования – отсутствие руководств: вам даются лишь средства построения интерфейса. Они спроектированы так, что программисту не нужно вводить букву за буквой многочисленные команды, и похожи на инженерные средства автоматизированного проектирования. Для таких программистов старой закалки, как я, это проклятие: я хочу ввести весь код сам, буква за буквой.

Когда-то делались попытки упростить ввод с клавиатуры (для тех, кто плохо печатает, для студентов) с помощью макросов (например, вводить ключевое слово `LET` одним нажатием клавиши), но они не прижились.

Сейчас я пытаюсь изучить некоторый язык, якобы являющийся «объектно-ориентированным». Справочное руководство отсутствует – во всяком случае, я его не нашел. В тех справочниках, которые есть, приводятся весьма тривиальные примеры, а 90% места посвящено утверждению превосходства ООП над другими «религиями». Некоторые мои знакомые посещали курсы C++, полностью несостоятельные с точки зрения преподавания. По моему мнению, ООП – один из величайших обманов общества. Все языки разрабатывались для определенного класса пользователей, например Фортран – для мощных числовых расчетов, и т. д. ООП разрабатывалось затем, чтобы его потребители могли претендовать на высшее знание, недоступное посторонним. Правда в том, что единственным существенным аспектом ООП является метод, придуманный десятилетия назад: инкапсуляция подпрограмм и данных. Все остальное – украшательство.

Разработка языка

Считаете ли вы современный Visual Basic компании Microsoft полноценным объектно-ориентированным языком, и если да, то одобряете ли вы этот его аспект (с учетом вашего неприятия парадигмы ООП)?

Том: Не знаю. Несколько простых опытов, которые я провел, показали, что работать с Visual Basic относительно легко. Сомневаюсь, что кто-либо за пределами Microsoft воспринимает VB как объектно-ориентированный. Фактически, True BASIC в такой же, если не большей мере, как VB, является объектно-ориентированным языком. В True BASIC есть модули, представляющие собой совокупность подпрограмм и данных, и они обеспечивают главную характеристику ООП, а именно инкапсуляцию данных. (В True BASIC нет наследования типов, поскольку типами, определяемыми пользователем, могут быть только многомерные массивы. И едва ли какой-либо язык поддерживает полиморфизм, поскольку на деле он требует интерпретации на этапе исполнения.)

Вы сказали, что Visual Basic обладает некоторыми серьезными ограничениями в сравнении с True BASIC. Вы имеете в виду, что в Visual Basic нет чего-либо похожего на вашу систему модулей?

Том: Не знаю. Я всего лишь написал несколько тренировочных программ – фактически, я ничего не писал на Visual Basic. Я просто убедился, что смог бы это сделать. У этой системы достаточно простой интерфейс пользователя – в отличие от других, с которыми я пробовал работать. Visual Basic – это тот же старый Microsoft Basic, который они пытаются выдать за объектно-ориентированный, всего лишь добавив инструмент для построения визуального интерфейса.

Интересно также ваше замечание о создании крупных программ для видео и аудио. Вы сказали, что такие сложные приложения проще строить с помощью объектно-ориентированных языков вроде Objective-C.

Том: Да, пожалуй, потому что для этого есть достаточная языковая среда.

Сейчас я безуспешно пытаюсь изучить Objective-C, во всяком случае это хорошая среда. Если разобраться, то можно разумным способом получить доступ ко всем элементам видео и аудио, предоставляемым платформой. Сам я не пробовал это сделать, поэтому не знаю, насколько это сложно, но в языковой среде разработчика все это есть.

Это не обязательно должна быть функция самого языка, это может быть функция его окружения.

Том: Это действительно имеет отношение не к языку как таковому, а к его окружению. Если языковой средой пользуется множество людей, то, возможно, у производителя найдется сотня программистов, чтобы заставить эту среду работать как надо.

Какие уроки из опыта изобретения, развития и распространения вашего языка могли бы извлечь разработчики компьютерных систем в настоящем и будущем?

Том: Да никакие.

Я помню компьютер Burroughs 5500 начала 1960-х. Он был сконструирован так, чтобы повысить эффективность стековых приложений вроде компиляторов Алгола.

Сегодня мода повернулась в другом направлении, в сторону RISC-машин.

Большинству языков программирования не требуется ничего специализированного. Отчасти потому, что скорость настолько велика – и все растет, – что компиляция и время обработки отдельных языков не составляют проблемы.

Может проявиться противоположная тенденция. Например, для обработки больших массивов можно построить специализированный компьютер, работающий с массивами, а тогда придется разработать для него особый язык программирования.

Если бы сегодня вам пришлось заново создавать язык для обучения программированию, был бы он похож на Бейсик?

Том: Весьма, потому что принципы, которым мы следовали, остаются теми же. Например, мы старались сделать язык легко запоминающимся, чтобы человек, который долго с ним не работал, мог сесть и вспомнить, как им пользоваться.

Мы старались по возможности избавить язык от непонятных требований. В Фортране, например, чтобы напечатать число, нужно использовать оператор форматирования, точно указав, как число должно быть выведено. Это малопонятная для изучения тема, особенно для тех, кто не слишком часто пользуется языком, поэтому в Бейсике мы печатали число так, как, с нашей точки зрения, было бы правильнее всего. Если это было целое число, то, несмотря на то что внутренне оно представлялось как число с десятичной запятой, мы выводили его как целое без запятой. Если вам нужно было ввести число, то можно было не беспокоиться о его формате: пишите его в своем любимом виде, и оно будет прочтено, – специальный формат для ввода данных не устанавливался.

Обычно таким же правилам следуют электронные таблицы – в них это хорошо получается. Для чисел в конкретной колонке можно задать формат вывода с заданным количеством десятичных знаков, иначе они будут показаны в общем виде. По-моему, это примерно то же, что мы делали в Бейсике. Разрешен ввод символов, причем в очень простом виде: просто вводите символы с клавиатуры, не следуя никаким правилам.

В 1970–1980-е годы мы сделали Dartmouth BASIC для структурного программирования, а потом добавили элементы поддержки объектно-ориентированного программирования. Не думаю, что следовало идти каким-то другим путем.

В ООП вас привлекает инкапсуляция?

Том: Совершенно верно. Обычно я говорил, что инкапсуляция составляет 70% того, что полезно в ООП, но сейчас я готов увеличить эту долю до 90%. Главное – объединить программы с их данными. Это действительно важно. Я теперь мало пишу программ, но когда я работал с True BASIC, то полностью полагался на инкапсуляцию.

У нас была возможность инкапсулировать группы подпрограмм в так называемые *модули*, что то же самое, только объединяет подпрограммы, и они отделяются от остальной программы, связываясь с ней только цепочкой вызова. Фактически, у них были свои закрытые данные. Это оказалось очень удобным для отделения функциональности и всего такого.

Многих разработчиков языков и систем я спрашивал, как они относятся к математическому формализму. Например, Scheme очень эффективно описывает лямбда-исчисление. Берете шесть примитивов, на которых все прекрасно строится. Такой подход характерен для математика.

Том: Да, это очень интересно. Это интересная математическая проблема, но при проектировании компьютерного языка необязательно этим заниматься, потому что все виденные мной компьютерные языки гораздо проще. Даже Фортран. Алгол прост; в нем используются рекурсивные определения, но и это довольно легко и просто.

Я не изучал теорию языков программирования, поэтому не могу к этому ничего добавить.

Задумывались ли вы о тех, кто будет использовать ваш язык, и о наиболее сложных проблемах, которые их ждут?

Том: Да. Самой большой проблемой для тех, кому предназначался наш язык, было не забыть его через неделю, если им приходилось программировать раз в две недели. Мы также хотели, чтобы языку програм-

мирования и его системному окружению можно было научить за пару часов, а не читать целый курс.

Именно так я и мои коллеги учились программировать. В начале 1980-х у нас были ПК с Microsoft Basic – Commodore 64 и Apple II. Там был Бейсик с командной строкой и подпрограммами, и практически ничего больше.

Том: Попадались причудливые вещи. В них действительно были странности. Я, например, сталкивался с Apple Soft BASIC. Не знаю, где его делали – в Microsoft или другом месте, – но все копировалось с первоначального Dartmouth BASIC. Там ввели многосимвольные имена переменных, но некорректно работали с ними. Если внутри многосимвольного имени переменной встречалось какое-нибудь ключевое слово, все ломалось.

Это происходило из-за игнорирования пробелов?

Том: Нет, из-за того, что они заявляли о поддержке многосимвольных имен переменных, которой в действительности не было. Это была фальсификация. Допустим, если вы давали переменной имя `TOT`, слово `TO` воспринималось как ключевое. Это было маркетинговой уловкой. В этих языках были мелкие функции, рассчитанные на рынок. Они думали, что многосимвольные имена переменных – это хорошая приманка. Те, кто работал с этими языками, изворачивались благодаря тому, что использовали многосимвольные имена с ограничениями.

Об этом не узнаешь, читая руководство.

Том: Да, об ошибках там не говорится.

Как появилось игнорирование пробелов?

Том: Все, что я об этом знаю, давно известно, а истоки частично связаны с тем, что Джон Кемени слабо владел клавиатурой. Не знаю, действительно ли причина в этом. Мы разрабатывали язык вместе, но эта функция – дело его рук.

Поскольку имена переменных уникальны и отличаются от ключевых слов, пробелы не нужны. Можно произвольно вставлять или удалять пробелы.

Все меняется. Сейчас я пишу программы на версиях True BASIC, которые все еще встречаются на разных машинах. Я пишу только на этом языке и пользуюсь пробелами для улучшения читаемости.

Не для того, чтобы обойти ограничения компьютера; для вас это вопрос человеческого фактора?

Том: Верно, потому что, знаете, главное, если вы пишете серьезную программу и собираетесь работать с ней дальше, – чтобы ее можно

было отложить на полгода, а потом снова взять и, прочтя код, понять, как она работает. Очень важно выбрать имена переменных так, чтобы по ним можно было определить величины, которые они представляют, или построить структуру: в основном это относится к подпрограммам, решающим одну-единственную задачу, – их имена должны ясно сообщать, чем они занимаются.

Сейчас, когда во всех компьютерных языках имена переменных могут быть любой длины, то пусть имя подпрограммы содержит 20 букв, лишь бы оно объясняло, что она делает. Тогда можно вернуться к программе через полгода и понять, как она работает.

Чак Мур говорит, что в Форте можно составить свой словарь и писать на языке предметной области, если правильно выбрать слова. Интересно, что такая идея возникала неоднократно.

Том: Такой компьютерный язык, как Бейсик, не рассчитан на профессиональных программистов. Если вы специалист в какой-то области и решили написать для нее приложение, то выберете для такой работы более простой язык, чем это сделал бы профессионал. В частности, я имею в виду широко распространившиеся объектно-ориентированные языки.

У меня есть некоторый опыт работы с одним из них, невероятно сложным. Это единственный язык в моей жизни, с которым я не справился, а количество языков, на которых я писал программы, приближается к трем десяткам.

Устраняет ли редактор WYSIWYG потребность в номерах строк? Вы говорили, что они нужны только для указания цели в операторах GOTO. Нужны ли номера строк программистам для ссылок при редактировании файлов?

Том: Совершенно не нужны (ответ на последний вопрос). Редактирование строк по номерам давно отмерло.

Как редакторы WYSIWYG повлияли на программирование?

Том: Никак. Редакторы WYSIWYG стали сложными и тесно связаны с обслуживаемым языком (в смысле отступов, цвета и т. п.).

Нет ли каких-то упущений в том, как сегодня преподаются компьютерные науки? Некоторые, например, считают, что мало внимания уделяется инжинирингу.

Том: Не знаю, поскольку я не в курсе, как сегодня преподаются компьютерные науки. Я вышел в отставку 15 лет назад и не преподавал ничего, кроме статистики и вычислений. Не имею представления о том, как развивалась эта область с той поры.

Как научиться отладке?

Том: Прежде всего, не надо все сводить к этому, то есть нужно лучше продумывать все заранее.

Один из наших бывших студентов стал сотрудником Apple, делал там очень важную работу и затем вышел в отставку. Прежде он работал в вычислительном центре в Дартмуте. Он написал компилятор PL/1 – это крупная штука – и проверял его, смотрел на него и так далее, но ни разу не тестировал и ни разу не запустил, пока все не было готово. Это 20 или 30 тысяч строк кода, и все тестирование заключалось в том, что он читал этот код. Затем он запустил его, и все заработало с первого раза!

Это исключительный случай в истории программирования! Если кто-то напишет программу в 20 000–30 000 строк, и она заработает с первого раза, это нечто необычное, верно? Но он сделал это, и это пример другим. Он работал в одиночестве – не в команде. Тот, кто работает один, всегда продуктивнее. Он очень тщательно проверял, как работают вместе разные части программы, и тщательно перечитывал код, а читая код, на самом деле эмулируешь то, что будет делать компьютер, поэтому проверяешь каждый шаг: это верно, это верно и т. д.

Поэтому когда он нажал кнопку «пуск» и все заработало, это удивительно – так ни у кого не бывает! Но идея понятна. Чтобы уменьшить количество ошибок, нужно прежде всего стараться не делать их.

Многие имеющиеся коммерческие программы кишат ошибками, потому что их пишут не хорошие программисты, а команды, а функции программы определяются в отделе маркетинга. Он должен представить в определенный срок программу с определенной функциональностью, поэтому в ней полно ошибок. Софтверные компании считают, что большинство пользователей лишь поверхностно используют свои программы, из-за чего не сталкиваются со многими ошибками.

Как вы решаете, что включить в сам язык, а что в библиотеку?

Том: Над этим мы тоже много думали. Все, что носило отпечаток загадочности и могло заинтересовать лишь часть пользователей, мы помещали в библиотеку. Так развивался язык на протяжении многих лет. И у нас получилась библиотека, позволяющая делать многие вещи; в современной версии Бейсик – True BASIC – мы используем библиотеку подпрограмм для доступа к объектам ООП, таким как кнопки, диалоговые окна и т. п. Мы используем библиотечную подпрограмму, и такого рода доступ не включается в сам язык, поэтому вы должны вызывать для этого подпрограммы из нескольких библиотек. В самом начале мы хорошо это продумали.

Когда программу пишут в команде, часто создают стандартные библиотеки общего пользования. Вы можете порекомендовать, как писать такие библиотеки на True BASIC?

Том: Я и сам писал библиотеки, но не знаю никакого особого совета, кроме того, что нужно стараться делать их проще. Это же общеизвестное правило: делай все проще и старайся избегать ошибок.

Например, важно, чтобы подпрограмма решала одну задачу. Подпрограмма не должна делать ничего постороннего, даже если это кажется вам неплохой идеей. Побочные эффекты могут быть катастрофическими. Есть много способов писать библиотеки так, чтобы сократить или исключить появление ошибок. Есть известные технологии сокращения ошибок, но я не знаю, часто ли они применяются в промышленности, потому что там на программистов сильно давит отдел маркетинга.

Рабочие задачи

Что для вас — успешная работа?

Том: В течение многих лет благодаря нашей работе – ее открытости, неограниченному доступу для студентов к участию в проекте – Дартмут обладал одной из лучших репутаций в компьютерном мире. К нам приезжали из России, Японии и других стран, чтобы посмотреть, как у нас все организовано. Это было до наступления эры персональных компьютеров. Сейчас у каждого есть персональный компьютер, поэтому такой проблемы нет, но в те времена было очень необычным, что мы разрешали студентам – любому студенту – заниматься любой работой на компьютере, в любое время без предварительного разрешения. В те времена это было в новинку. Благодаря этому Дартмут заслужил репутацию, которой хватило лет на 10 или 15, и получил преимущества в отношении привлечения финансовых средств, студентов и преподавательского состава.

Другим достижением было то, что многие студенты Дартмута, изучавшие там компьютерное дело, смогли благодаря полученным знаниям сделать выдающуюся карьеру. Эти люди не работали в вычислительных центрах корпораций – они работали в других подразделениях. Многие выпускники Дартмута стали миллионерами просто благодаря тому, что умели пользоваться компьютером!

Это основные показатели нашего успеха.

Что могут извлечь молодые люди из вашего опыта?

Том: Им следует думать о конечных пользователях программ, тех, кто будет их применять. Со многими современными приложениями очень трудно работать.

Мне известно, что несколько лет назад в Microsoft попытались ввести программу, дружественную пользователю, которая, кажется, называлась «Bob», но они не поняли сути; они решили, что «дружественная пользователю» программа должна опекать его, как если бы имела дело с ребенком. На самом деле дружественность пользователю имеет другой смысл.

Боюсь, наша отрасль в целом не понимает смысла дружественности пользователю. Не знаю, понимают ли смысл этих слов те, кто сейчас занимается компьютерными науками.

Мой совет: заботьтесь о людях, которым придется пользоваться вашей программой.

Должен ли интерфейс быть легким в изучении, как Бейсик?

Том: Да, легким в изучении, простым для описания в руководстве того или иного рода и делающим примерно то, чего вы от него ожидаете во время работы, не преподнося сюрпризов.

Вы сказали, что всегда были продуктивнее, работая в одиночку. Хотелось бы понять, что вы здесь имели в виду под продуктивностью.

Том: Смысл очень простой. Думаю, есть множество свидетельств в мою пользу. Я никогда в жизни не работал в составе команды программистов. Я всегда говорил: «OK, нужно сделать вот это, и я напишу для этого программу». Всем, что я написал, мог управлять я сам, единственный программист.

Я пользовался тем, что писали другие, но никогда не входил в команду разработчиков. Я написал бессчетное число строк кода, но есть несколько программ, которыми я до сих пор пользуюсь и в которых по 10 000 строк. Их легко писать. Их легко отлаживать. Если какая-то функция не нравится, ее легко поменять, и если работаешь самостоятельно, не нужно писать всякие докладные записки. Я не против документации, когда она есть. У большинства моих программ ее нет, потому что они предназначены для моего личного пользования, но я верю всему, что написал о программировании Фред Брукс в своей книге «Мифический человеко-месяц».[1]

Мне нравится его рассказ об оценке времени написания программы. Программист пишет в день три строки документированного кода или около того, и создание приложений у них непомерно затягивалось. Они не могли понять, в чем дело, пока не вскрылось, что программисты работают только 20 часов в неделю. Они присутствуют на работе 40 часов, но 20 часов тратится непроизводительно, например на совещания.

---

[1] Ф. Брукс «Мистический человеко-месяц, или Как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

Вот что я больше всего ненавижу – совещания. Иногда они бывают совершенно необходимы. Помню, как первоначально разрабатывался Бейсик для компьютеров GE225, GE235. Студенты-программисты раз в неделю собирались примерно на час. Обычно председательствовал Джон Кемени. Как он любил говорить, сам он принимал все второстепенные решения, например кому отдать предпочтение в алгоритме планировщика, а студенты принимали все важные решения, например какой бит для какой цели использовать.

У нас тогда было две машины, так что на каждой машине было по одному студенту. Это были второкурсники, и они должны были работать вместе, общаться между собой. В целом они занимались собственной работой. Когда мы писали компилятор или редактор, это была работа для одного.

Вы еще говорили о студенте, который написал компилятор PL/1, заработавший с первого раза.

Том: Это Фил Кох (Phil Koch), он работал в Apple. Сейчас он ушел из Apple, живет в штате Мэн. Он был потрясающим программистом. Он долго и скрупулезно вычитывал код.

Каков главный урок, извлеченный вами из своего долгого и обширного опыта?

Том: Делайте так, чтобы пользователям проще было работать с вашей программой.

Можете называть это дружественностью пользователю, если хотите, но учтите, что в отрасли определили дружественность, как мне кажется, в смысле «снисходительность». Настоящая задача дружественности пользователю в том, чтобы определить разумные значения по умолчанию в создаваемом вами приложении, чтобы человеку, впервые с ним встречающемуся, не нужно было изучать все варианты и степени свободы, которые ему предоставлены. Пользователь должен сесть и сразу начать работать. Если ему потребуется что-то иное, должен быть относительно простой способ это получить.

Чтобы это стало возможным, нужно иметь некоторое представление о своих будущих пользователях.

Я часто работал с Microsoft Word, и с моей точки зрения это совершенно не дружественное пользователю приложение. Лет 10 назад у Microsoft появилась эта программа «Bob», идея которой была ошибочна. Они не поняли, что в действительности означает дружественность пользователю.

Думаю, есть и дружественные пользователю приложения, но сейчас все заняты созданием веб-сайтов. Одни веб-сайты спроектированы хо-

рошо, другие плохо. Если, попав на веб-сайт, вы не можете понять, как получить дополнительную информацию, он плохо спроектирован.

Таким вещам трудно учить.

Бен Шнейдерман (Ben Shniederman), специалист по человеческому фактору в компьютерных науках в Университете Мэрилэнд, провел исследования,[1] из которых следует, что наши решения в Бейсике для структур DO, LOOP и IF лучше в отношении дружественности пользователю, чем отдельные структуры в других языках, например использование точки с запятой в Алголе и Паскале для завершения предложений.

В обычной жизни не пишут точку с запятой в конце предложения, поэтому данное правило приходится изучать специально. Помню, например, что в Фортране в одних местах нужна запятая, а в других – нет. В результате в программе Космического центра во Флориде возникла ошибка, и они потеряли ракету из-за пропущенной запятой. Кажется, Эд Тафт (Ed Tufte) писал об этом. Старайтесь избегать мест, где может возникнуть неоднозначность.

Я продолжаю всем говорить, что мы с Кемени потерпели неудачу, потому что не сделали дружественными компьютеры других пользователей, но мы проделали хорошую работу со своими студентами, потому что в течение 20 лет наши выпускники получали очень хорошо оплачиваемые должности в промышленности, потому что умели работать.

Таким успехом можно гордиться.

Том: Если вы преподаете, то это главное.

[1] Shneiderman, B. «When children learn programming: Antecedents, concepts, and outcomes», The Computing Teacher, v. 5: 14–17 (1985).

6

AWK

Философия UNIX, которая предполагает множество мелких утилит, способных образовывать мощные комбинации, ярко проявляется в языке программирования AWK. Его изобретатели, Ал Ахо, Питер Вайнбергер и Брайан Керниган, говорят, что это язык для синтаксически управляемой обработки по шаблонам (pattern matching). Его простой синтаксис и толковый набор полезных функций дают возможность легко преобразовывать текст однострочными командами, не имея при этом представления о синтаксическом анализе, грамматиках и конечных автоматах. Несмотря на то что его влияние распространилось на универсальные языки, такие как Perl, на всякой современной UNIX-системе по-прежнему устанавливается AWK, который продолжает незаметно и эффективно трудиться.

Жизнь алгоритмов

Как бы вы охарактеризовали AWK?

Ал Ахо: Я бы сказал, что AWK – простой в изучении и применении язык сценариев, отлично проявляющий себя в приложениях для стандартной обработки данных.

Какова ваша роль в разработке AWK?

Ал: В 1970-х годах я исследовал эффективные алгоритмы синтаксического анализа и обработки строк по шаблонам. Мы с Брайаном Керниганом обсуждали возможность усовершенствования grep с целью сделать текстовый разбор по шаблонам и обработку в ряде приложений более универсальными. Узнав об этом проекте, Питер Вайнбергер проявил к нему живой интерес, и мы быстро выпустили в 1977 году первую версию AWK.

После этого в течение нескольких лет язык существенно развивался в связи с тем, что некоторые наши коллеги стали использовать его для ряда задач обработки данных, многие из которых оказались для нас неожиданными.

В каком контексте наиболее уместно применение AWK?

Ал: Думаю, AWK по-прежнему непревзойден для простых приложений стандартной обработки данных. В нашей книге по AWK приведены десятки практических примеров, в которых программа из одной-двух строк AWK способна делать то, для чего в Си или Java потребуются сотни строк.

Что нужно учитывать программистам при разработке программ на AWK?

Ал: AWK – язык сценариев для создания коротких программ, выполняющих стандартную обработку данных. Он не предназначен для написания крупных приложений, хотя это часто делают, поскольку язык очень прост. Для крупных приложений я рекомендовал бы использовать обычную практику хорошего программирования: модульность, выбор осмысленных имен переменных, добротное комментирование и т. д. Эти приемы полезны и при создании коротких программ.

Как влияет рост доступности аппаратных ресурсов на мышление программиста?

Ал: Вне всякого сомнения, возросшие скорости, обилие памяти и хорошие IDE сделали программирование более приятным занятием. Кроме того, программы могут теперь обрабатывать значительно большие массивы данных, чем в прошлом. Для меня стало обычной практикой обрабатывать AWK-программами объемы данных, которые на несколько

порядков больше прежних, поэтому более мощное аппаратное обеспечение повысило мою продуктивность как пользователя.

Однако есть и обратная сторона: усовершенствование аппаратного обеспечения привело к резкому росту размера и сложности программных систем. С развитием аппаратуры программы становятся полезнее, но и сложнее, – не знаю, кто окажется победителем.

Разрабатывая алгоритмы для AWK, каким вы представляли объем данных, с которым сможет справиться ваш код?

Ал: Там, где это было возможно, мы реализовывали алгоритмы с линейным временем в худшем или среднем случае. Благодаря этому AWK смог осиливать все большие и большие объемы входных данных.

Мы проверяли AWK на наборах данных разного объема и измеряли, как меняется производительность с ростом объема. Мы всеми силами пытались сделать свой продукт более эффективным и использовали реальные данные, чтобы оценить свои успехи.

Вы оценивали возможный в будущем рост объема данных?

Ал: Когда мы разрабатывали AWK, мне казалось, что мегабайт данных – это очень много. Если учесть, что сейчас в Интернете доступны эксабайты данных, то по нынешним понятиям «большой» набор данных на много порядков превосходит наши прежние представления. Конечно, даже алгоритм с линейным временем оказывается слишком медленным для терабайта данных, поэтому для обработки данных в Интернете требуются совершенно иные методы.

Мне встречалось определение AWK как «языка разбора по шаблонам в простых задачах обработки данных». Если учесть, что AWK создан больше 30 лет назад, какие изменения произошли за это время в обработке по шаблонам?

Ал: Объем и разнообразие разбора и обработки по шаблонам неизмеримо выросли за прошедшие 30 лет. Параметры задач заметно расширились: шаблоны стали сложнее, а размер данных значительно увеличился. Сегодня стало обычным делом применение поисковых механизмов для обработки текста по шаблонам на всех страницах Интернета. Нас также интересует интеллектуальный анализ данных (data mining) – поиск любых закономерностей в гигантских цифровых библиотеках, таких как базы данных геномов и научные архивы. Правильнее будет назвать обработку строк по шаблонам одной из фундаментальных задач компьютерных наук.

Появились ли более мощные алгоритмы обработки по шаблонам или их реализации?

Ал: Обработка по шаблонам в AWK осуществляется посредством быстрого и компактного ленивого алгоритма построения переходов состояний, с помощью которого по регулярному выражению создаются переходы детерминированного конечного автомата, требуемого для обработки по шаблонам. Этот алгоритм описан в «книге красного дракона».[1] Время работы алгоритма по существу линейно зависит от длины регулярного выражения и размера входного текста. Это лучшее известное время для регулярных выражений. Можно было реализовать алгоритмы Бойера–Мура или Ахо–Корасик для особых случаев, когда регулярное выражение представляется одним ключевым словом или конечным множеством ключевых слов. Мы не стали этого делать, поскольку не знали, какие регулярные выражения будут использоваться в программах AWK.

Можно отметить, что применение в программных системах сложных алгоритмов имеет свои отрицательные стороны. Эти алгоритмы могут быть непонятны другим (да и самому автору – по прошествии некоторого времени). Я включил в AWK некоторые сложные технологии обработки по шаблонам с помощью регулярных выражений. Они описаны в «книге красного дракона», но Брайан Кернган, взглянув однажды на мой модуль обработки по шаблонам, сделал к нему одно дополнение, комментарий на староитальянском языке – в переводе «оставь надежду всяк сюда входящий».[2] В результате ни Кернган, ни Вайнбергер не стали касаться этой части кода. Исправлять ошибки в этом модуле всегда приходилось только мне!

Разработка языка

Что бы вы посоветовали разработчикам языка программирования?

Ал: Всегда помнить о будущих пользователях. Когда кто-то говорит, что твой инструмент помог ему решить задачу, это очень приятно. Также приятно, когда на основе твоей работы кто-то создает еще более мощные инструменты.

Каковы взгляды Кернигана и Вайнбергера на разработку языка?

Ал: Если кратко описать главную идею каждого из нас по поводу создания языка, то для Кернигана это простота изучения, для Вайнберге-

---

[1] Aho, Alfred V. et al. Compilers: Principles, Techniques, and Tools (Addison-Wesley, 1986); Альфред Ахо, Рави Сети, Джеффри Ульман «Компиляторы. Принципы, технологии, инструменты». – Пер. с англ. – СПб.: Вильямс, 2003.

[2] Lasciate ogni speranza, voi ch'intrate – надпись над вратами ада («Ад», Песнь III «Божественной комедии» Данте).

ра – надежность реализации, а для меня – практичность. Мне кажется, AWK сочетает все три этих качества.

Как отражается стремление к практичности на конструктивных решениях? Как оно влияет на проектирование?

Ал: Сознательно это происходит или нет, но выживает, несомненно, то, что практично. В духе дарвинизма. Вы создаете понятия и стиль, полезные для решения задач, которые вас интересуют, но если они не годятся для решения задач, которые интересуют других, то ваши труды тщетны. Практичность возникает в результате выживания наиболее жизнеспособных идей. Непрактичные языки отмирают.

Мы не искусствоведы, тем не менее наблюдаем дихотомию между красивой программой и функциональной программой.

Ал: Разве нельзя соединить оба эти качества?

Люди склонны проводить между ними границу. Вопрос в том, является ли программирование творческой деятельностью, искусство это или ремесло.

Ал: Кнут, несомненно, придавал очень большое значение эстетической стороне программирования. Он считал, что программы должны быть красивы. Почти все знакомые мне программисты согласятся с тем, что программе должна быть присуща элегантность.

Мастер-мебельщик может сказать так: «Вот стул. На него можно сесть или встать. На него можно положить стопку книг. Но поглядите, как он изящен, как красивы сочленения, как прекрасна резьба!» Красота свойственна даже функциональному предмету.

Ал: Но ведь и в минимализме есть красота, то есть чтобы получить красивую вещь, не нужны всякие узоры и завитушки.

Как усовершенствовать свое мастерство программиста?

Ал: Прежде всего, советую думать до того, как начнете программировать. Далее, рекомендую писать побольше кода, просить экспертов оценить ваш код, читать хороший код других авторов и участвовать в рецензировании кода. Если хватит смелости, попробуйте учить студентов, как писать хороший код.

Я пришел к выводу, что лучший способ изучить какой-то предмет – обучать ему других. В процессе преподавания вам приходится так организовать и представить материал, чтобы предмет стал понятен другим. Во время лекции студенты задают вопросы, которые отражают различные точки зрения на рассматриваемые вами проблемы. В результате вы приходите к более глубокому и точному их пониманию.

К программированию это относится в полной мере. Если вы преподаете программирование, то учащиеся спрашивают, а нельзя ли сделать так-то или так-то, и вы осознаете, что есть много способов программного решения некоторой задачи. Вы понимаете, что люди думают по-разному, а думая по-разному, они предлагают разные способы решения задачи, и благодаря этому вы получаете гораздо лучшее представление о разных подходах к решению задач вообще.

Я определенно выяснил, что когда пишу книгу и собираюсь поместить в нее программы, эти программы становятся эффективнее и короче по ходу написания книги. За тот год, что мы писали книгу по AWK, многие вошедшие в нее программы укоротились на 50%. Это произошло благодаря тому, что мы научились использовать в AWK абстракции еще более эффективно, чем предполагали первоначально.

Обнаружили ли вы недостатки в конструкции AWK, когда писали эту книгу?

Ал: Когда с помощью AWK стали решать многие задачи, которые мы изначально не рассматривали, обнаружились некоторые аспекты языка, связанные с тем, что мы не задумывали его как универсальный язык программирования. Я не стал бы называть их «недостатками»: они лишь показывают, что AWK – специализированный язык, не предназначенный для некоторых задач, где его пытались применять.

Можно было что-то сделать для решения таких задач, или вы упорно сопротивлялись превращению AWK в универсальный язык?

Ал: После создания первоначальной версии AWK язык лет десять развивался с добавлением новых конструкций и операторов, но он остался языком для работы с шаблонами, языком решения задач обработки данных. Мы не стали выводить его из той ниши, которую он занимал.

Как довести идею синтаксически управляемых преобразований до пользователей, которые могут мало или вообще ничего не знать о конечных автоматах или автоматах с магазинной памятью?

Ал: Разумеется, пользователю AWK необязательно знать об этих понятиях. С другой стороны, если вы занимаетесь разработкой или реализацией языка, знание конечных автоматов и контекстно-свободных грамматик необходимо.

Должен ли пользователь lex или yacc разбираться в контекстно-свободных грамматиках, если пользователям генерируемых ими программ не требуется понимать их?

Ал: В большинстве случаев от пользователей lex не требуется понимание того, что представляет собой конечный автомат. Пользователь yacc в действительности пишет контекстно-свободную грамматику, следо-

вательно, с этой точки зрения пользователь уасс должен, конечно, разбираться в грамматиках, но для работы с уасс не нужно знать теорию формальных языков.

Иначе вас завалит ошибками «конфликт shift/reduce».

Ал: Полезное свойство уасс заключается в том, что, автоматизируя создание детерминированного парсера по грамматике, он сообщает разработчикам языка программирования о наличии неоднозначных или с трудом поддающихся анализу конструкций. Без этого они могли и не заметить существование таких неудачных конструкций. Благодаря уасс разработчики языков не раз восклицали: «О, а я и не заметил, что эту грамматическую конструкцию можно интерпретировать двояко!» После чего удаляли или изменяли спорную конструкцию. Неоднозначность в приоритетах или ассоциативности легко разрешалась с помощью простых механизмов описания: «мне нужен такой-то порядок приоритетов операторов языка и такой-то порядок ассоциативности».

Как создавать язык, облегчающий отладку? Как при проектировании языка добавить или убрать функции, чтобы улучшить отладку?

Ал: В разработке языков программирования прослеживается тенденция создавать языки, повышающие надежность программ и продуктивность программистов. Нужно разрабатывать языки параллельно с надежными приемами создания программ, чтобы задаче создания надежных программ были подчинены все этапы жизненного цикла программного обеспечения, в особенности ранние стадии проектирования систем.

Нельзя разрабатывать системы, рассчитывая на то, что люди смогут безошибочно написать миллионы строк кода, и одной только отладки недостаточно для эффективной разработки надежного программного обеспечения. Систематичность синтаксиса и семантики – хороший путь для исключения случайных ошибок.

Культура UNIX

В ранней культуре UNIX существовало представление, что если есть задача, то для ее решения нужно написать небольшой компилятор или небольшой язык. Когда, по-вашему, лучше создать язык для решения некоторой конкретной задачи, чем писать программу на имеющемся языке?

Ал: В современном мире есть тысячи языков программирования, и возникает вопрос, для чего они нужны. Практически в любой сфере человеческой деятельности есть свой профессиональный язык. У музыкантов есть специальная система записи музыки; юристы пользуются

своим жаргоном; у химиков есть специальные диаграммы, описывающие атомы и молекулы и способы их соединения. Нередко люди договариваются о том, чтобы создать на базе системы обозначений язык для решения задач в некоторой области.

С помощью универсального языка можно описать любой алгоритм, но, с другой стороны, часто бывает удобнее, экономичнее и даже эффективнее располагать специализированным языком для решения определенного класса задач. Нужно ли создавать новый язык, подскажет здравый смысл, но если это важная область с особыми чертами, поддающимися автоматизации, то нет ничего удивительного в том, что появляется язык программирования для описания решений задач в данной области.

Снижает ли это затраты на программистов или машинное время?

Ал: Языки, по крайней мере раньше, возникали потому, что выявлялись некоторые важные классы задач, которые нужно было решать, и для решения задач в этих областях придумывали языки, позволявшие эффективно использовать аппаратное обеспечение. По мере того как аппаратное обеспечение становилось более дешевым и быстрым, языки приобретали возможности более высокого уровня, а эффективность использования аппаратуры теряла значение.

Вы считали, что AWK сам по себе достаточно силен в своей нише?

Ал: Заложенная в AWK парадигма действий с шаблонами хорошо подходит для решения больших классов часто возникающих задач обработки данных. Изменение этой парадигмы повредило бы языку, сделав его менее привлекательным для того класса задач, который мы рассматривали. Кроме того, этот язык очень удобен для программирования в командной строке UNIX.

Это напоминает философию UNIX, предполагающую сочетание множества мелких инструментов, каждый из которых эффективен в своей области.

Ал: Думаю, это очень удачное описание.

Чаще всего AWK встечался мне в командной строке или в сценариях оболочки.

Ал: Среди программ AWK очень популярны приложения, где можно написать решение задачи в командной строке или создать сценарий оболочки, состоящий из команд UNIX. Такой стиль решения задач характерен для ранних приложений AWK для UNIX и даже многих современных UNIX-приложений.

Для UNIX «все является файлом». Могли бы вы назвать нечто таким «файлом» для Интернета?

Ал: Файл – это прекрасная простая абстракция, которую нужно использовать по мере возможности. Но сегодня в Интернете стало очень много типов данных, и программам часто приходится работать с параллельными потоками интерактивных мультимедийных данных. По-видимому, лучшим решением является создание стандартных корректных API для работы с данными, а правильную реакцию на данные в недопустимом формате должны обеспечивать программы, отвечающие за безопасность.

Каковы ограничения инструментов командной строки и графических интерфейсов?

Ал: С помощью AWK очень удобно преобразовывать формат выходных данных одной программы, чтобы их можно было направить на вход другой программы. Если в графическом интерфейсе подобное преобразование запрограммировано так, что выполняется одним щелчком, то это, понятно, удобнее. А если нет, то добраться до внутренних форматов, чтобы выполнить необходимые преобразования, может быть очень нелегко.

Есть ли связь между идеей объединения программ в командной строке с помощью конвейера и написанием малых языков для отдельных областей?

Ал: Думаю, такая связь есть. Несомненно, поначалу конвейеры облегчали композицию функций в командной строке UNIX. Берутся входные данные, неким образом преобразуются, и результат по конвейеру подается на вход другой программы. Это мощный способ быстро создать новую функциональность путем простого объединения программ. Многие пытаются решать задачи по этому принципу. Язык Perl Ларри Уолла, который я считаю потомком AWK и других инструментов UNIX, сочетает многие аспекты подобной композиции программ в единый язык.

Когда вы говорите о «композиции функций», в памяти возникает математическое понятие композиции функций.

Ал: Именно это я и имею в виду.

Повлиял ли этот математический формализм на изобретение конвейера или это стало метафорой позднее, когда было замечено сходство?

Ал: Думаю, так было с самого начала. Изобретение конвейера ставится в заслугу Дугу Макилрою, по крайней мере, в моей книге. Он рассуждал как математик, и я думаю, что он видел эту связь с самого начала. Мне представляется, что командная строка UNIX – это прототип функционального языка.

В какой мере полезна формализация семантики и языка? Основан ли AWK на формализме?

Ал: AWK построен по принципу синтаксически управляемой трансляции. Я весьма интересовался компиляторами и их теорией, поэтому при создании AWK реализация была осуществлена в виде синтаксически управляемой трансляции. Формальный синтаксис AWK был представлен в виде контекстно-свободной грамматики, и трансляция с исходного языка в целевой выполнялась с помощью семантических действий, основанных на этой формальной грамматике. Это облегчило расширение и разработку AWK. В нашем распоряжении были новые инструменты для создания компиляторов lex и yacc, которые весьма помогли в экспериментировании и разработке языка.

Саймон Пейтон-Джонс из команды Haskell заявил, что формализовано было 80–85% этого языка, а на дальнейшую формализацию просто не стоило тратить силы ввиду снижения отдачи.

Ал: По соображениям безопасности специфицированность приобретает все большее значение в разработке языков и систем. Хакеры часто используют необычные или не специфицированные части системы, угрожая безопасности.

Добавим сюда задачу разработки библиотек, и проблема усложнится еще больше. «Я описал формализмы языка, но теперь мне нужна библиотека для работы с Интернетом. Определил ли я формализмы этой библиотеки? Соответствуют ли они формализмам языка? Не нарушают ли его формализмы и гарантии?»

Ал: Работая в области телекоммуникаций, я заметил, что почти все спецификации интерфейсов, для которых Bell Labs создавала оборудование, были написаны на английском языке и соответствовали международным стандартам. То есть многие стандарты были написаны на английском и потому часто оказывались неоднозначными, неполными и противоречивыми. Однако, несмотря на все эти трудности, международные сети телекоммуникаций и Интернет хорошо работают и взаимодействуют – в значительной мере благодаря хорошо определенным интерфейсам между системами.

Сторонние разработчики часто создают драйверы устройств и приложения для операционных систем других производителей. Если драйвер устройства или приложение содержит ошибки, то низкое качество программы припишут производителю системы, хотя он в этом не виноват. В последнее время исследователи достигли больших успехов в создании инструментов для верификации программ с помощью моделей и других мощных технологий, позволяющих проверить, правильно ли разработчики драйверов и приложений используют системные API. Эти новые средства верификации программ заметно влияют на качество программного обеспечения.

Принесет ли этот формализм пользу языкам?

Ал: Сегодня почти каждый язык в том или ином виде допускает описание формальной грамматики. Большой вопрос в том, насколько полно мы можем или хотим описывать семантику языка с помощью современных формализмов для описания семантики языков программирования. Семантические формализмы в значительно меньшей степени автоматизируемы по сравнению с построением парсера по контекстно-свободной грамматике. Несмотря на то что описывать семантику скучно, я большой сторонник планирования, описания и схематизации семантики языка до начала его реализации.

Вспоминаются две истории: классический случай с Make, когда Стюарт Фельдман решил, что нельзя убрать табуляцию, потому что у программы уже 12 пользователей, и статья Дика Гэбриела «Worse Is Better»[1] (Чем хуже, тем лучше), в которой он описывал подход в Нью-Джерси и в MIT. Победили UNIX, Си и подход Нью-Джерси.

Ал: Я всегда характеризовал это как победу дарвинизма. Я считаю, что удачные языки растут и развиваются в зависимости от использования их действующими программистами. Языки, возникающие в результате работы солидных комиссий, обычно не привлекают внимания программистов. Если не принуждать к их использованию, они обычно не выживают.

В популярных языках несколько тревожит то, что они неуклонно расширяются. Неизвестно, как удалять функции из уже существующих языков. Главные современные языки, например C++ и Java, в огромной степени расширились по сравнению с тем, какими они были при своем появлении, и сокращения их размеров в будущем не предвидится. Одному человеку уже невозможно разобраться во всех деталях этих языков, а компиляторы для них состоят из миллионов строк.

Это нерешенный вопрос в исследовании систем: как создать язык, чтобы его можно было расширить за пределы первоначальной предметной области, не модифицируя сам язык? У вас есть механизм расширения?

Ал: Есть проверенный временем подход – библиотеки.

Даже в C++ и Java ожидаются изменения.

Ал: Это так. Даже основные языки расширяются, но есть и сдерживающие силы, требующие сохранить совместимость основных языков с прежними версиями, чтобы не потерять работающие программы. Это ограничивает безудержное развитие языков.

---

[1] *http://www.dreamsongs.com/WorseIsBetter.html*

Хорошо ли это само по себе?

Ал: Конечно, очень желательно иметь возможность пользоваться старыми программами. Я когда-то написал статью для журнала «Science Magazine», озаглавленную «Software and the Future of Programming Languages» (Программное обеспечение и будущее языков программирования). Там я попытался оценить объем программного обеспечения, на работе которого держится наше существование, учитывая программы, используемые как организациями, так и отдельными людьми.

У меня получилось от половины до одного триллиона строк исходного кода. Исходя из того что создание конечной строки кода стоит от 10 до 100 долларов, я сделал вывод, что мы просто не можем позволить себе заново написать сколь-нибудь значительную часть имеющихся программ. Из этого следует, что нынешние языки и системы сохранятся достаточно долго. Аппаратное обеспечение во многом более подвижно, чем программное, поскольку мы всегда стремимся к созданию более быстрой платформы для работы старых программ.

Что касается UNIX, то эта операционная система оказалась очень подвижной. Было ли это связано с возможностью портировать ее или с желанием переносить имеющиеся программы на разные новые аппаратные платформы?

Ал: UNIX развивалась, но компьютеры развивались еще быстрее. Крупный сдвиг в UNIX произошел, когда Деннис Ричи создал язык Си, чтобы написать третью версию UNIX. Это сделало UNIX портируемой. В то относительно недолгое время, когда я работал в Bell Labs, UNIX была всюду – от миникомпьютеров до крупнейших в мире суперкомпьютеров, потому что она была написана на Си и существовала технология переносимой компиляции, благодаря чему мы быстро изготавливали компиляторы Си для новых машин.

Одной из задач UNIX было создание системы, облегчающей разработку программ, чтобы она нравилась программистам и они хотели разрабатывать в ней новые программы. Думаю, эта задача была успешно решена.

Это относится к большинству лучших инструментов и программ.

Ал: Вот интересный вопрос: кто делает лучшие инструменты – одиночки или специалисты? Думаю, однозначного ответа мы не найдем, но нет сомнений, что в ранние времена существования UNIX многие из самых полезных инструментов были созданы программистами, разработавшими оригинальные средства для решения своих задач. Это одна из причин появления AWK. Перед Брайаном, Питером и мной стоял определенный класс программ, которые нам нужно было написать, и мы хотели, чтобы они были как можно короче.

Побуждало ли наличие инструментов и быстрота практической реакции к поиску лучших инструментов и лучших алгоритмов?

Ал: Если взглянуть на раннюю историю UNIX и начало моей исследовательской карьеры, то на меня оказали большое влияние слова Кнута о том, что лучшая теория порождается практикой, но и лучшая практика полагается на теорию. Я написал десятки статей о том, как эффективнее, удобнее и быстрее выполнять синтаксический анализ конструкций, имеющихся в реальных языках программирования. Стив Джонсон, Джефф Ульман и я тесно сотрудничали при разработке этой теории и уасс, поэтому уасс служит отличным примером единства теории и практики.

Роль документации

Когда я пишу документацию, учебник или статью для своего ПО, то нередко замечаю, что замысел отдельных мест трудно или неудобно объяснить, в результате чего переделываю программу. Случается ли с вами что-нибудь похожее?

Ал: Весьма часто. Опыт разработки AWK существенно повлиял на мой курс языков программирования и компиляторов в Колумбийском университете. В этот курс входит проект продолжительностью в один семестр, в котором студенты, работая группами по пять человек, должны написать собственный небольшой язык и компилятор для него.

За те 20 лет, что я читаю курс компиляторов, не было случая, чтобы команда в конце семестра не смогла представить работающий компилятор. Такой успех достигнут не случайно, а благодаря моему опыту работы над AWK, наблюдению за разработкой программного обеспечения в Bell Labs, признанию важности облегченного процесса программной разработки в такого рода проектах и вниманию к мнению моих студентов.

Для успешного создания нового языка и действующего компилятора для него за 15 недель немаловажна технология программирования, применяемая в проекте разработки компилятора. Две недели студентам дается на то, чтобы решить, хотят ли они слушать этот курс. Через две недели формируются группы, а еще через две недели они должны представить доклад (по образцу «белой книги» Java) с описанием языка, который они хотят создать. Этот доклад представляет собой ценностное предложение для языка с указанием, почему он необходим и какими свойствами должен обладать. Главное значение этого доклада в том, что он заставляет студентов быстро решить, какой язык они хотят создать.

Через месяц студенты пишут учебник по языку и справочное руководство. Учебник пишется по образцу главы 1 книги Кернигана и Ричи

«Язык программирования Си», а справочное руководство – по образцу приложения А той же книги. Я подвергаю очень тщательному критическому разбору учебник и справочник, потому что на этой стадии студенты не осознают, насколько трудно сделать работающий компилятор даже для небольшого языка.

Я выясняю у студентов, реализацию каких функций они гарантируют, а какие они реализовали бы, если бы у них было больше времени. (Не было случая, чтобы они реализовали какие-либо дополнительные функции.) Цель этого задания – определить рамки проекта, который можно было бы осуществить в течение семестра, эквивалентного другим по затратам сил.

После формирования команд студенты выбирают себе руководителя проекта, системного архитектора, системного интегратора, ответственного за верификацию и специалиста по языку. Каждый из них играет важную роль в создании и подготовке к выпуску работоспособного компилятора.

В обязанности руководителя проекта входят создание графика выпуска конечных результатов проекта и контроль его соблюдения. Системный архитектор создает блок-схему компилятора, а системный интегратор определяет инструменты и среду разработки, с помощью которых будет создаваться компилятор. Как только написано справочное руководство по языку, ответственный за верификацию разрабатывает план тестирования и набор тестов для языка в целом. Специалист по языку следит за тем, чтобы свойства, описанные в техническом докладе по языку, оказались действительно реализованными.

Мы сделали комплект регрессионных тестов для AWK, хотя, возможно, с некоторым опозданием. Наш тестовый комплект оказал нам неоценимую услугу. Развивая язык, мы неизменно прогоняли набор регрессионных тестов, прежде чем внести изменения в наш основной каталог. Поэтому у нас всегда была работающая версия компилятора. Прежде чем добавить новые функции к языку, мы добавляли тесты для этих функций в комплект регрессионных тестов.

Я уже говорил, что ни разу ни одна команда студентов не провалила выпуск работающего компилятора в конце семестра. Ключевым для этого достижения было наличие комплекта регрессионных тестов: студенты представляли к концу семестра то, что у них работало к этому моменту. Но в работающем компиляторе должны быть реализованы все функции языка, заявленные в справочном руководстве.

Системный архитектор разрабатывает блок-схему компилятора, спецификации интерфейсов и назначает ответственных за реализацию компонентов и сроки для этого. Каждый член команды должен написать

в интересах проекта не менее 500 строк исходного кода, и все, включая руководителя проекта, должны принять участие в реализации. Студентам очень полезно (и интересно) писать программы, которые должны взаимодействовать с кодом, созданным другими людьми.

Системный интегратор должен определить платформу, на которой будет строиться компилятор, и какие инструменты, скажем, lex, yacc, ANTLR или эквивалентные, будут использоваться. Он должен также научиться применять эти инструменты и обучить правильной работе с ними других участников, чтобы в каждой команде был резерв специалистов по инструментам.

Самая интересная работа у специалиста по языку (language guru). Он отвечает за интеллектуальную целостность языка, чтобы свойства, сформулированные в техническом докладе по языку, действительно оказались реализованными. Он должен увязывать изменения в проекте и коде, так чтобы при внесении командой модификаций в язык они регистрировались и доводились до сведения всех участников, а также отражались на комплекте регрессионных тестов.

В процессе работы над проектом студенты осваивают три важных умения: управления проектом, работы в команде и общения, как устного, так и письменного. В конце курса я спрашиваю у студентов, что было самым важным из усвоенного ими в этом курсе. Часто они отмечают один из этих перечисленных мной навыков. Проект управляется документацией, и студенты вынуждены много заниматься описанием и обсуждением программного обеспечения. Они должны публично представить свой язык, постаравшись убедить при этом своих однокурсников в том, что этот язык необходим абсолютно всем. Первой команде я помогаю провести успешную презентацию. Следующие команды стараются превзойти своими презентациями первую, потому что им очень нравятся свои языки. Среди созданных языков были такие, которые моделировали квантовые компьютеры, сочиняли музыку, создавали комиксы, моделировали цивилизации, выполняли быстрые матричные преобразования, генерировали графику.

В конце курса студенты должны представить окончательный отчет о проекте с такими разделами: технический доклад по языку (white paper), учебник по языку, справочник по языку, описание управления проектом, составленное руководителем проекта, блок-схема и спецификации интерфейсов, описанные системным архитектором, описание платформ разработки и инструментов, составленное системным интегратором, план тестирования и контрольные примеры, составленные ответственным за верификацию, и процедура согласования компонентов языка, описанная специалистом по языку. Последний раздел озаглавлен «Выводы» и содержит ответы на вопросы: «Чему вы научились

как команда? Чему научились вы лично? Что в этом курсе вы предложили бы сохранить, а что изменить, если бы он был предложен на будущий год?» В приложении приводится листинг кода с указанием автора каждого модуля.

Если довольно долго совершенствовать что-то, обычно достигаешь хорошего результата. Я следовал советам, данным мне студентами, и несколько лет назад получил за этот курс награду «Лучший преподаватель» от сообщества выпускников Колумбийского университета.

Многие из тех, кто интервьюировал посещавших этот курс студентов при приеме на работу, заявили, что были бы рады, если бы их программные системы разрабатывались с помощью такого же процесса.

Студенты какого года обучения проходят этот курс?

Ал: Обычно старшекурсники и студенты магистратуры первого года, но есть много предварительных условий: углубленное знание программирования, теории компьютерных наук, структур данных и алгоритмов. Мне нравится, что студенты склонны к распределенной разработке программного обеспечения, поэтому они применяют такие вещи, как wiki и современные IDE. Многие студенты стажируются на производстве.

Я всегда подчеркиваю, что студенты должны поддерживать комплект регрессионных тестов в актуальном состоянии по мере развития языка. Регрессионное тестирование значительно повышает эффективность работы студентов, потому что те ошибки, которые они обнаруживают, оказываются их собственными, а не сделанными другими участниками команды.

Когда и как нужно учить отладке?

Ал: Думаю, отладке нужно учить параллельно с программированием. Брайан во многих своих книгах дает надежные практические советы по отладке. Но я не знаю ни одной хорошей общей теории отладки. Технология, применяемая при отладке компилятора, сильно отличается от той, которая используется при отладке программ численного анализа, поэтому лучшим подходом будет, возможно, уделять больше внимания к поблочным тестам, систематической процедуре тестирования и применению инструментов отладки в любых курсах программирования. Я также полагаю, что студентам весьма полезно подготовить спецификацию того, что должны делать их программы, до того момента, когда они непосредственно начнут их писать.

Одной из наших ошибок при разработке AWK было отсутствие строгого тестирования с самого начала работы. Строгое тестирование стало применяться уже после запуска проекта, но позднее стало ясно, что наша

работа была бы намного эффективнее, если бы мы с самого начала создали и развивали наборы строгих тестов.

Какие показатели должны оценивать разработчики, развивая базовый код, и каким образом?

Ал: Главную заботу должна составлять корректность реализации, но для ее достижения нет простого пути. Здесь нужно решать разные задачи, например продумывать инварианты, проводить тестирование и рецензирование кода. Оптимизация нужна, но она не должна быть преждевременной. Необходимо поддерживать соответствие документации и комментариев изменениям в коде, но об этом часто забывают. Непременным условием является использование современных IDE и добротных инструментов для разработки программного обеспечения.

Как вы возобновляете программирование, прервавшись на несколько дней? несколько месяцев?

Ал: Когда человек пишет программную систему (или, если хотите, книгу), он должен упорядоченно держать всю систему в голове. Отвлечения прерывают цепочку рассуждений, но если они недолгие, то обычно в нужное место системы возвращаешься, просто просмотрев код. Если перерыв длится несколько месяцев или лет, то чтобы вспомнить, что я кодировал, я обычно обращаюсь к статьям, книгам или заметкам, в которых записывал свои алгоритмы.

Из сказанного следует, что большим подспорьем самим разработчикам или тем, кому приходится долго сопровождать код, служат хорошие комментарии и документация. Брайан вел журнал регистрации основных решений, которые мы принимали во время разработки языка. Этот журнал был исключительно полезен.

Компьютерные науки

Что представляют собой исследования в компьютерных науках?

Ал: Чудесный вопрос, на который нет четкого ответа. Думаю, что круг исследований в компьютерных науках невероятно расширился. В компьютерных науках сохраняются глубокие, нерешенные, основополагающие вопросы: как доказать, что такие задачи, как разложение на множители или NP-полные задачи, действительно являются сложными; как моделировать сложные системы вроде живой клетки или человеческого мозга; как строить масштабируемые, надежные системы; как программистам создавать ПО с произвольной степенью надежности; как наделить программные системы дружественными человеку свойствами вроде эмоций или интеллекта; как долго может продолжаться действие закона Мура?

Сегодня масштаб и диапазон применения вычислительной техники невероятно велики. Мы пытаемся сделать доступной всю имеющуюся в мире информацию, и компьютеры и вычисления воздействуют на все сферы повседневной жизни. В результате в компьютерных науках возникли целые новые области исследований в междисциплинарных приложениях, где вычисления соединяются с другими сферами науки и приложения усилий человека. В качестве примера можно привести вычислительную биологию, робототехнику, киберфизические системы. Мы не знаем, как лучше всего применять компьютеры в образовании или здравоохранении. Защита информации и безопасность приобрели невиданное ранее значение. Я считаю компьютерную науку не менее захватывающей для исследователей, чем любую другую.

Какова роль математики в компьютерных науках и программировании?

Ал: Думаю, лучшие технические достижения возникают на прочном математическом фундаменте. Язык AWK разрабатывался на основе ряда взятых из теории компьютерных наук элегантных абстракций, таких как регулярные выражения и ассоциативные массивы. Эти конструкции впоследствии были приняты основными языками сценариев: Perl, JavaScript, Python и Ruby. Мы также воспользовались эффективными алгоритмами на базе конечных автоматов, чтобы реализовать примитивы для обработки строк. В целом, как мне кажется, AWK стал прекрасным примером соединения хорошей теории с надежной инженерной практикой.

Вы разрабатывали теорию автоматов и ее приложения к языкам программирования. Что стало для вас самой большой неожиданностью, когда вы приступили к реализации результатов своих исследований?

Ал: Вероятно, удивительнее всего оказалась их широкая применимость. Я могу представить теорию автоматов как формальные языки и автоматы, которые их распознают. Теория автоматов дает нам удобные понятия, в частности регулярные выражения и контекстно-свободные грамматики, позволяющие описывать важные синтаксические функции языков программирования. Автоматы, распознающие эти формальные языки, такие как конечные автоматы и автоматы с магазинной памятью, могут служить моделями алгоритмов, применяемых в компиляторах для сканирования и синтаксического разбора программ. Возможно, главный результат применения теории автоматов в компилировании – появление таких инструментов для построения компиляторов, как lex и yacc, которые автоматизируют построение эффективных сканеров и парсеров, основанных на этих автоматах.

Что мешает построить компилятор (и/или язык), который выявит все потенциальные ошибки? Где граница между ошибками, обусловлен-

ными некорректным построением программы, и ошибками, которые можно было бы обнаружить и предотвратить, если бы язык мог лучше упреждать события?

Ал: Неразрешимость не позволяет сконструировать компилятор, обнаруживающий все ошибки в программе. Тем не менее мы сильно продвинулись в создании полезных инструментов верификации программ, использующих такие мощные технологии, как проверка моделей, для поиска важных классов ошибок в программах. Думаю, в будущем среды для разработки ПО будут располагать разнообразными средствами верификации, с помощью которых программист сможет точно находить многие стандартные причины ошибок в программах.

Предвижу, что благодаря применению более сильных языков, более мощных средств верификации и усовершенствованных технологий программирования надежность и качество программного обеспечения возрастут.

Можно ли разработать алгоритмы обработки по шаблонам, задействующие параллелизм современных многоядерных процессоров?

Ал: Сегодня это активная область исследований. Многие ученые изучают платформы с параллелизмом и программные реализации алгоритмов обработки по шаблонам, например, алгоритма Ахо–Корасик и алгоритма конечных автоматов. Интерес к этой области мотивирован такими прикладными задачами, как анализ генома и системы обнаружения вторжения.

Что мотивировало вас и Корасик к разработке алгоритма Ахо–Корасик?

Ал: Его появление связано с весьма любопытной историей. В начале 1970-х я работал над книгой «The Design and Analysis of Computer Algorithms»[1] (Addison–Wesley) вместе с Джоном Хопкрофтом и Джеффри Ульманом. Я читал лекцию в Bell Labs о методах построения алгоритмов. В зале присутствовала Маргарет Корасик (Margaret Corasick), работавшая в библиотеке технической информации Bell Labs. После лекции она подошла ко мне и сообщила, что написала программу для библиографического поиска на основе булевых функций ключевых слов и словосочетаний. Однако в некоторых случаях сложного поиска стоимость прогона программы могла превышать разрешенные для поиска 600 долларов.

В ее реализации программы поиска применялся простой алгоритм обработки по шаблонам. Я посоветовал ей попробовать выполнять поиск

[1] Ахо А., Хопкрофт Дж., Ульман Дж. «Построение и анализ вычислительных алгоритмов». – Пер. с англ. – М.: Мир, 1979.

ключевых слов параллельно с помощью конечных автоматов и рассказал, что существует способ эффективного построения автомата для обработки по шаблонам с любым набором ключевых слов за линейное время.

Она появилась у меня на работе спустя несколько недель и сказала: «Помните ту программу поиска за 600 долларов? Я запрограммировала алгоритм, который вы предложили. Теперь поиск стоит 25 долларов. Любой поиск стоит теперь 25 долларов: это цена чтения ленты». Так родился алгоритм Ахо–Корасик.

Узнав об этом, начальник моей лаборатории Сэм Морган сказал: «Почему бы тебе не продолжить работу над алгоритмами? Думаю, когда-нибудь в будущем они понадобятся». Вот в чем состояло очарование Bell Labs в те времена: там были те, кто умел ставить задачи, и те, кто умел нестандартно подходить к решению этих задач. Когда эти люди встречались, появлялись замечательные открытия.

Разведение малых языков

Как вы пришли в программирование?

Брайан Керниган: Не могу вспомнить какой-то конкретный случай. Компьютер я увидел впервые где-то на младших курсах колледжа, а по-настоящему учиться программировать (на Фортране) стал только где-то год спустя. Думаю, больше всего удовольствия от программирования я получил, работая летом 1966 года над проектом MAC в MIT, когда писал программу, создававшую ленту задания для новехонького GE 645 с только что появившейся Multics. Я писал программу на MAD, что было гораздо легче и приятнее, чем на Фортране и Коболе, с которыми я работал до того, и использовал CTSS, первую систему с разделением времени, что было несравнимо проще и приятнее, чем иметь дело с перфокартами. Именно тогда появилась возможность заниматься интересной стороной программирования – решением задач, поскольку техническая работа стала меньше отвлекать.

Как вы учите новые языки?

Брайан: Мне проще всего изучать новый язык на тщательно отобранных примерах, решающих какую-нибудь задачу, близкую к тому, что я хочу сделать. Я копирую пример, адаптирую его к своим потребностям, а потом расширяю свое знание языка по мере того, как это требуется для конкретного приложения. Я пытаюсь что-то делать во многих разных языках, так что со временем границы стираются, и мне требу-

ется некоторое время, чтобы переключиться с одного на другой, особенно если они не похожи на Си, который я давно изучил. Полезно иметь хорошие компиляторы, которые сообщают не только о недопустимых, но и о подозрительных конструкциях: здесь удобны системы со строгой типизацией, такие как C++ и Java, а также опции, требующие строгого соответствия стандартам.

Вообще говоря, самое лучшее – написать побольше кода, желательно хорошего, и чтобы им пользовались другие люди. Следующее по полезности, хотя и более редкое, – чтение хорошего кода, по которому видно, как другие люди пишут программы. Наконец, помогает широкий опыт: каждая новая задача, новый язык, новый инструмент и новая система помогают совершенствоваться, устанавливая связи с тем, что тебе уже известно.

Какой должна быть структура руководства по новому языку?

Брайан: Такое руководство должно позволять легко находить нужное. То есть нужен хороший именной указатель, таблицы операторов и библиотечных функций должны быть краткими и полными (и легко отыскиваемыми), а примеры – короткими и предельно ясными.

Иное дело учебник, который, безусловно, должен отличаться от справочника. Думаю, для учебника лучше всего своего рода «спиральный» подход, когда сначала излагается небольшой набор базовых понятий, которого достаточно для написания законченных и полезных программ. Очередной виток спирали должен охватывать другой уровень детализации или альтернативные способы решения тех же задач, а примеры должны быть такими же полезными, но могут быть крупнее. В конце помещается хороший справочник.

Нужно ли включать в примеры – даже для начинающих – код обработки ошибок?

Брайан: Даже не знаю, что правильнее. С одной стороны, код, обрабатывающий ошибки, часто оказывается громоздким, малоинтересным и не наглядным, поэтому он часто мешает изучению и пониманию базовых конструкций языка. В то же время необходимо напоминать программистам, что существует такая вещь, как ошибки, и что их код должен уметь справляться с ними.

Лично я предпочитаю в начальных главах учебника по возможности не обращать внимания на ошибки, ограничиваясь напоминанием, что ошибки бывают, а также игнорировать возможные ошибки в большинстве примеров справочников, кроме разделов, специально посвященых ошибкам. Но при этом возникает подсознательная уверенность, что можно не обращать внимания на ошибки, а это, конечно, нехорошо.

Как вы относитесь к идее описывать имеющиеся ошибки в руководствах UNIX? Оправданна ли такая практика сегодня?

Брайан: Мне нравились разделы BUGS, но в те времена программы были маленькими и достаточно простыми, так что можно было указать отдельные ошибки. В BUGS часто описывались функции, еще не реализованные или реализованные некорректно, а не ошибки в обычном смысле, когда, скажем, происходит выход за границы массива. Не думаю, что это реально для большинства ошибок, которые можно найти в крупных современных системах, – во всяком случае, им не место в справочнике. Сетевые системы учета ошибок весьма полезны для управления разработкой ПО, но едва ли они нужны обычным пользователям.

Нужны ли сегодняшним программистам те выводы, которые вы сделали в своей книге о стиле программирования «The Elements of Programming Style»[1] (Computing McGraw-Hill)?

Брайан: О да! Основные идеи хорошего стиля, которые в принципе сводятся к тому, чтобы писать понятно и просто, столь же важны сегодня, как и 35 лет назад, когда мы с Биллом Плоджером впервые о них сказали. Что-то по мелочи изменилось, отчасти в связи с особенностями разных языков, но основы все те же. С простым, понятным кодом гораздо легче работать, с ним реже возникают проблемы. Поэтому по мере роста размера и сложности программ прозрачность и простота кода приобретают еще большее значение.

Влияет ли стиль вашего обычного текста на то, как вы пишете программы?

Брайан: Вполне возможно. Работая и с текстом, и с программами, я много раз переделываю материал, пока не удовлетворюсь результатом. Конечно, это больше относится к тексту, но и в коде я также стремлюсь выражаться как можно яснее и понятнее.

Становятся ли лучше программы от того, что разработчик понимает, какие задачи будет решать с их помощью пользователь?

Брайан: Если разработчик не сумел достаточно глубоко уяснить, для чего будет использоваться его программа, скорее всего, ничего хорошего он не напишет.

Удачна ситуация, когда разработчик понимает пользователя, потому что сам будет пользоваться создаваемой программой. Одной из причин первоначального успеха UNIX у программистов было то, что ее созда-

---

[1] Б. Керниган, Ф. Плоджер «Элементы стиля программирования». – Пер. с англ. – М.: Радио и связь, 1984.

тели Кен Томпсон и Деннис Ричи сами нуждались в системе для разработки ПО; в результате UNIX стала очень удобной для программистов, пишущих новые программы. С языком программирования Си та же история.

Если разработчики не слишком хорошо знают и понимают прикладную сторону, то очень важно как можно больше общаться с пользователем и изучать его опыт. Весьма полезно понаблюдать, как начинают работать с вашей программой новые пользователи: через считаные минуты может выясниться, что пользователь заходит в тупик, пытаясь совершить действия или делая допущения, которые вам и в голову не могли прийти. Если же вы не уследите за тем, как ведут себя пользователи, впервые столкнувшись с вашей программой, то не узнаете о возникающих у них проблемах, а при более позднем общении может оказаться, что они уже свыклись с недостатками вашего проекта.

Как программисту научиться лучше программировать?

Брайан: Пишите больше кода! Обдумывайте написанный код и старайтесь его переделать, чтобы он стал лучше. Желательно, чтобы ваш код читали другие люди – коллеги по работе или по проекту open source. Полезно также писать код разного типа и на разных языках – это расширит репертуар приемов, которыми вы владеете, и расширит ваши возможности при решении возникающих в программировании задач. Читайте код, написанный другими программистами, например, чтобы расширить его возможности или исправить ошибки; вы, таким образом, сможете познакомиться с тем, как решают задачи другие. Наконец, превосходный способ улучшить свой код – обучать кого-то программированию.

Всем известно, что искать ошибки вдвое тяжелее, чем писать код. Как, по вашему мнению, нужно учить отладке?

Брайан: Не уверен, что отладке можно учить, но, конечно, можно попытаться рассказать, как вести отладку систематически. В «The Practice of Programming» есть целая глава, где мы вместе с Робом Пайком попытались рассказать, как более эффективно вести отладку.

Отладка – это искусство, но совершенствоваться в отладке определенно можно. Молодые программисты допускают много ошибок по невнимательности, например выходя за начальную или конечную границу массива, передавая функциям аргументы неверного типа или (в Си) задавая некорректные символы преобразования в `printf` и `scanf`. К счастью, такие ошибки обычно легко обнаруживаются, потому что приводят к хорошо заметным сбоям. Но главное, их можно исключить при написании кода, проверяя граничные условия, – надо только, когда пи-

шешь код, думать о неприятностях, которые могут возникнуть. Обычно ошибки обнаруживаются в только что написанном коде или в том, который вы начали тестировать, – вот здесь и надо сосредоточить свои усилия.

Более сложные или неявные ошибки требуют больших усилий. Один из эффективных методов – «разделяй и властвуй»: исключать части данных или части программы, чтобы локализовать ошибку во все более и более узкой области. Часто в ошибках прослеживается закономерность: изучение особенностей ошибочных входных или выходных данных часто помогает разгадать причину ошибок.

Сложнее всего с ошибками, возникающими в результате неправильного понимания вами ситуации, когда вы вообще не видите проблемы. В таких случаях я стараюсь сделать перерыв, посмотреть листинг, рассказать о проблеме постороннему, воспользоваться отладчиком. Все это помогает мне по-другому взглянуть на проблему, и этого часто оказывается достаточно, чтобы локализовать ее. К сожалению, отладка всегда будет трудным делом. Чтобы избежать отладки, лучше всего очень аккуратно писать код.

Как доступные аппаратные ресурсы влияют на мировоззрение программиста?

Брайан: Обычно чем больше аппаратных ресурсов, тем лучше: например, меньше проблем с управлением памятью, которое служило постоянным источником головной боли и ошибок лет 20-30 назад (как раз тогда, когда мы писали AWK). Кроме того, можно использовать не самый эффективный код, в частности универсальные библиотеки, потому что скорость выполнения совершенно перестала быть проблемой, как это было 20 или 30 лет назад. Например, сегодня меня нисколько не беспокоит необходимость обрабатывать с помощью AWK файлы размером в 10 или даже 100 Мбайт, что в прежние времена было просто немыслимо. С ростом скорости процессоров и объема памяти становится проще быстро провести эксперимент или даже написать работающий код на интерпретируемом языке (таком как AWK), что было невозможно пару десятилетий назад. Все это замечательные достижения.

В то же время доступность ресурсов часто приводит к чрезмерно раздутым проектам и реализациям, к появлению систем, которые могли быть быстрее и проще в применении, если бы при их проектировании задавались несколько большие ограничения. Современные операционные системы явно страдают от этой проблемы: загрузка моих машин происходит все дольше и дольше, хотя, согласно закону Мура, они существенно быстрее прежних. Замедление происходит из-за нового программного обеспечения.

Что вы думаете по поводу проблемно-ориентированных языков (DSL)?

Брайан: Я много работал над тем, что сейчас часто называют «проблемно-ориентированными языками», хотя сам я называл их «малыми языками», а некоторые – «языками конкретных приложений» (application-specific languages). Идея в том, что, нацелив язык на конкретную, как правило, узкую область, можно хорошо приспособить к этой области его синтаксис, и писать код для решения задач в данной области станет просто. Примеров можно привести много: SQL, в частности, и, конечно, сам AWK – язык для упрощения и компактности некоторых видов обработки файлов.

Большая проблема малых языков заключается в том, что они имеют тенденцию к росту. Если они оказываются полезны, люди стремятся применять их более широко, раздвигая границы области первоначального предназначения языка. Обычно в такой язык добавляют дополнительные функции. Например, изначально язык мог быть чисто декларативным (без проверок условий и циклов) и не поддерживать переменные или арифметические выражения. Однако все это полезные вещи, которые хочется добавить. А после их добавления язык разрастается (его уже не назовешь «малым») и постепенно становится похож на все другие универсальные языки, хотя и с иным синтаксисом и семантикой, а иногда и более слабой реализацией.

Мне довелось работать с несколькими языками подготовки документации. Сначала, с Лориндой Черри, мы работали с EQN – языком для типографского набора математических выражений. Он был довольно удачен, и после того, как наше типографское оборудование расширило свои возможности, я также сделал язык для вычерчивания фигур и графиков, названный PIC. Этот PIC сначала мог только чертить, но вскоре стало ясно, что ему нужны арифметические выражения для вычисления координат и тому подобного, а потом потребовались переменные для хранения результатов, а также циклы для создания повторяющихся структур, Все это было добавлено, но не в очень удобном и надежном виде. В итоге PIC стал достаточно мощным, полным по Тьюрингу языком, но писать на нем большие программы никто не хотел.

Что вы считаете успехом в своей работе?

Брайан: Большое удовлетворение испытываешь, когда кто-нибудь говорит, что воспользовался твоим языком или инструментом и это помогло ему лучше справиться со своей задачей. Это действительно приятно. Правда, иногда вслед за этим тебе сообщают о возникших проблемах или отсутствующих функциях, но даже это полезно.

В каких контекстах AWK — мощное и полезное средство?

Брайан: AWK, по-видимому, остается лучшим средством для импровизированного анализа данных: найти все строки с неким свойством, просуммировать какие-то элементы данных, выполнить несложное преобразование данных. Часто мне удается с помощью пары строк AWK сделать больше, чем другие могут сделать с помощью 5 или 10 строк Perl или Python, и, как показывает опыт, мой код работает почти столь же быстро.

У меня есть набор маленьких скриптов AWK, с помощью которых можно, например, суммировать поля во всех строках, вычислить диапазон значений для каждого поля (чтобы быстро оценить некий набор данных). У меня есть программа на AWK, которая форматирует произвольный текст, так, чтобы строки были не длиннее 70 символов, и я раз сто на дню пользуюсь ею для обработки почтовых сообщений, и так далее. Все эти программы легко можно было бы написать на другом языке сценариев, и они работали бы не хуже, но на AWK их сделать проще.

О чем нужно помнить тем, кто пишет программы на AWK?

Брайан: Изначально этот язык был задуман для написания программ размером всего в одну-две строки. Если вам нужно написать что-то крупное, выбор AWK может оказаться неподходящим, поскольку в этом языке нет механизмов, полезных в больших программах, а ряд конструктивных особенностей может затруднить отыскание ошибок: например, отсутствие объявлений переменных и их автоматическая инициализация очень удобны в программах-однострочниках, но в больших программах это затрудняет поиск ошибок написания и опечаток.

Разработка нового языка

Как бы вы приступили к созданию нового языка программирования?

Брайан: Допустим, есть некий набор задач, определенная прикладная область, в которой, как вам кажется, некоторый новый язык программирования оказался бы лучше, чем все ныне существующие. Подумайте, что людям хотелось бы описывать на этом языке. Для каких задач и приложений планируется использовать этот язык? Как бы вы хотели описать их на этом языке? Каков наиболее естественный способ их описания? Какими наиболее важными и простыми примерами можно было бы заинтересовать людей? Постарайтесь подобрать самые понятные.

Главная идея в том, чтобы попытаться написать что-то на этом языке, когда его еще нет. Как можно что-то на нем высказать? Думаю, это вполне применимо к AWK, потому что его проектирование было на-

целено на то, чтобы легко писать лаконичные и полезные программы. В результате мы отказались от объявлений – отчасти из-за того, что у нас не было типов. Кроме того, у нас не было явных операторов ввода – так получилось, что ввод был совершенно неявным. У нас также не было операторов для разбиения входных строк на поля, потому что это выполнялось автоматически. Все характеристики языка были обусловлены задачей создать средство для простого описания простых вещей.

Стандартные примеры, использованные нами в первой статье по AWK, затем в справочнике и т. д., практически все записывались одной строкой. Если мне нужно вывести все строки длиннее 80 символов, достаточно написать `length > 80`, и все. Такой язык позволял достаточно легко понять, что мы хотели сделать, но потом, конечно, обнаруживалось отсутствие явно необходимых вещей, например возможности вводить данные из файлов, задаваемых по имени, и тогда мы ее добавляли. Конструкции, понадобившиеся для программ, не умещавшихся в несколько строк, например функции, были добавлены позднее.

Язык EQN, над которым мы работали с Лориндой Черри, представлял собой совершенно другой пример. EQN – это язык описания математических выражений для их вывода на печать. Задача заключалась в том, чтобы сделать язык максимально близким к тому, как люди читают математические выражения вслух. Если бы мне понадобилось описать вам математическую формулу по телефону, как бы я это сделал? Или если бы я писал формулу на доске в классе, какие слова я произносил бы при этом? Либо, как в моем случае, нужно было написать учебник для слепых. Как читать математическое выражение, чтобы всякий, не видя его, мог его понять? EQN был целиком направлен на то, чтобы облегчить запись математических выражений в том виде, как они произносятся вслух, не слишком заботясь о качестве вывода. Сравните это с TeX, в котором очень непросто выполнять набор и есть богатый синтаксис, но это очень мощный язык с большими возможностями управления выводом, а расплачиваться за них приходится изрядной трудностью применения.

Разрабатывая язык, много ли вы думали над его реализацией?

Брайан: Довольно много, потому что я всегда участвовал и в проектировании, и в реализации: если не видел пути для реализации чего-либо, то не настаивал на включении этого в проект.

Почти для каждого языка, который я разрабатывал, оказывалось, что он весьма прост и для него можно написать несложный специальный парсер, либо, при более изощренной синтаксической структуре, можно задать грамматику с помощью yacc.

Думаю, если бы мне пришлось разрабатывать язык вроде EQN или AWK без yacc, у меня ничего бы не вышло, потому что писать парсеры вручную слишком трудно. То есть сделать это можно, но слишком хлопотно. С помощью такого инструмента, как yacc, можно пробовать смелые и интересные вещи, быстро меняя конструкцию, если что-то не получается, потому что достаточно переписать кусочек грамматики, и существенное изменение в языке или добавление новых функций обходится без модификации значительного объема кода. С таким средством, как yacc, это делалось легко, с обычным рекурсивным нисходящим парсером все было бы гораздо сложнее.

Должны ли разработчики языка навязывать определенный стиль, помогающий избегать определенных частых ошибок? Как, например, форматирование исходного кода Python или отсутствие арифметики указателей в Java?

Брайан: У меня нет здесь определенного мнения, хотя обычно навязываемая дисциплина оказывается полезной, если к ней привыкнуть. Правила отступов в Python сначала раздражали меня, но когда я к ним привык, они перестали мешать.

Придумывая язык, нужно стремиться к тому, чтобы его конструкции позволяли людям легко сказать то, что им нужно, без неоднозначного толкования или слишком многих способов сделать одно и то же. В любом случае люди найдут самый естественный способ выражения. Поэтому, если в Java нет указателей, то это существенное отличие от Си и C++, но зато там есть ссылки, которые во многих случаях оказываются разумной альтернативой. В Java нет оператора `goto`. У меня никогда не возникало из-за этого проблем. В Си оператор `goto` есть, и хотя обычно я им не пользуюсь, иногда он оказывается удобен. Поэтому меня не смущают те или иные решения в таких вопросах.

Я уже говорил о языке PIC для вычерчивания картинок. Он хорош для простых фигур вроде стрелок, прямоугольников и блок-схем. Но народ пожелал рисовать картинки с помощью обычных структур. Поэтому я с некоторой неохотой добавил циклы `while` и `for`, и даже оператор `if`. Разумеется, это было сделано задним числом, с несколько кривым синтаксисом, который не вполне вписывался в язык PIC и не был похож на обычные языки. Это работало, но выглядело нескладно.

Картина такова, что сначала язык выглядит просто, а потом расширяется, и постепенно в него вводится все, что характерно для полноценных языков программирования со всеми их переменными и выражениями, условными операторами и циклами, а также функциями. Но обычно все эти конструкции выглядят неуклюже, синтаксис неправильный или, во всяком случае, необычный, механизмы не всегда работают гладко, и все в целом оставляет неприятное впечатление.

Это происходит из-за того, что язык изначально задумывался как ограниченный и его развитие в универсальный не предполагалось?

Брайан: Да, думаю, из-за этого. Я говорю только о своем личном опыте, но моим начальным замыслом всегда был какой-нибудь маленький язык, совсем простой и не предназначенный для больших задач и функций универсальных языков. Но когда язык оказывается удобным, люди пытаются расширить его границы и получить от него больше. А то, что им требуется, обычно относится к функциям универсальных языков, делающим их программируемыми, а не просто декларативными. Они хотят многократности без повторения одного и того же, а значит, нужны циклы и макросы или функции.

Можно ли разработать язык, который подойдет всем? Вы говорили об ограниченных языках, созданных для конкретных задач, но у меня сложилось впечатление, что вам нравится, когда разработчик пишет язык для собственных потребностей. Если у вас есть действующий язык, как расширить его, чтобы сделать более полезным и для других?

Брайан: Едва ли когда-нибудь появится язык, которым будут довольны все программисты, независимо от того, какие приложения они пишут, или хотя бы удовлетворяющий достаточно большую группу программистов, пишущих достаточно большой класс приложений.

У нас есть множество хороших универсальных языков. Для некоторых работ отлично подходит Си; C++, Java, Python – каждый хорош в своей области и может быть применен почти во всех других областях. Но едва ли я стану писать операционную систему на Python, как не хотел бы писать код для работы с текстом на Си.

Как вы определяете, в какой области некий язык оказывается особенно удобным или мощным? Например, вы сказали, что Python не годится для написания операционной системы. Это связано с самим языком или его реализацией?

Брайан: Думаю, значение имеет и то и другое. Реализация может быть причиной недопустимой медлительности готовой системы. Но если бы мне нужно было написать игрушечную или демонстрационную систему, Python мог бы вполне подойти. В некоторых отношениях он мог оказаться лучше. Но я едва ли стал бы использовать Python, скажем, для создания операционной системы, поддерживающей инфраструктуру Google.

На практике у программистов не всегда есть возможность выбора. Они вынуждены выполнять требования той среды, в которой работают. Например, программисту в большой финансовой фирме на Уолл-Стрит придется программировать на одном конкретном языке или выбирать из весьма ограниченного множества, скорее всего, C++ и Java. Вы не

сможете сказать: «О, я буду писать это на Си», или «Думаю, здесь лучше всего подойдет Python». Я знаю одну компанию, в которой комплект языков включает C++, Java и Python. Для некоторых задач был бы лучше Ruby, но писать на Ruby вам не дадут. У многих отсутствует свобода выбора языка, на котором им предстоит писать программу.

С другой стороны, когда есть конкретная задача, они могут свободно выбирать между C++, Java и Python. Тут уже можно при выборе рассмотреть технические соображения.

А пусть у каждого будет свой персональный язык!

Брайан: Личный язык, который никто больше не использует?

У каждого будет персональный синтаксис, транслируемый в общий байт-код, и тогда он станет универсальным.

Брайан: Это затруднит совместную разработку. :)

Что нужно делать после того, как создашь первый прототип?

Брайан: Сперва попробуйте сами написать код на предлагаемом языке. Какое возникает впечатление, когда вы пишете то, что нужно лично вам, и то, что, как вам кажется, захотят писать другие? В случае EQN это было совершенно очевидно. Как разговаривают математики? Сам я не математик, но имею представление, потому что прослушал множество курсов по математике. Вам нужно попробовать язык самому, а затем как можно скорее дать поработать с языком другим пользователям, причем таким, которые могут высказать полезную критику, т. е. повертят его и так и сяк и расскажут, что они обнаружили.

Огромным достоинством Bell Labs в 1970–1980-х было существование группы людей, более или менее близко связанных с разработкой UNIX и весьма преуспевших в подобном критическом анализе работы, сделанной другими. Критика часто оказывалась очень резкой, но вы быстро получали оценку того, что хорошо, а что плохо. Все мы от этого выигрывали, потому что критика помогала сглаживать острые углы, поддерживать культурную совместимость систем и отсеивать дурные идеи.

Мне кажется, что сейчас вы редко такое найдете. Благодаря Интернету вас критикуют многие из разных мест, но едва ли вы найдете целенаправленную критическую оценку со стороны группы очень талантливых и близких людей, с которыми ежечасно сталкиваетесь в коридоре и к которым всегда можно зайти в соседний кабинет.

Как вам удавалось проверять различные идеи и экспериментировать и в то же время построить уникальную и устойчивую систему?

Брайан: Это было не столь сложно, потому что AWK был очень невелик. Первая версия насчитывала всего около 3000 строк. Кажется, ее напи-

сал Питер Вайнбергер. Грамматика была получена с помощью yacc, что оказалось очень просто. Что касается лексики, то там был применен lex. А семантика в те времена была достаточно стандартной. У нас было несколько разных версий интерпретатора, но они были невелики. Довольно легко было быстро вносить изменения, сохраняя контроль над разработкой. На самом деле, язык по-прежнему остается очень небольшим и в той версии, которую я распространяю сейчас, 30 лет спустя, – немногим более 6000 строк.

Правда ли, что каждый из вас должен был писать тестовые модули для каждой новой функции, которую хотел включить в язык?

Брайан: Нет. Совершенно неверно. К моменту опубликования книги в 1988 году мы начали более регулярно подбирать контрольные примеры. И лишь несколько лет спустя я стал наводить некоторый порядок в тестах и примерно тогда решил, что если я добавляю новую функцию, то должен добавить и несколько тестов, проверяющих правильность работы этой функции. Долгое время я не вводил никаких новых функций, но набор тестов расширялся, потому что когда кто-нибудь находил ошибку, я добавлял один-два теста, с помощью которых можно было бы обнаружить эту ошибку раньше и гарантировать, что она не проявится вновь.

Думаю, это полезная практика. Жаль, что мы не применяли ее более тщательно и систематически с самого начала. В нынешний комплект тестов входят практически все программы из первой и второй глав учебника по AWK. Но они возникли уже после выхода книги, а та появилась намного позже создания языка.

За последние 40 лет было проведено много исследований в компьютерных науках и в области языков программирования. Способствовало ли это прогрессу в разработке языков, если не считать развития инструментов?

Брайан: Едва ли я располагаю достаточными данными, чтобы правильно ответить на этот вопрос. Для ряда языков – пожалуй, это касается языков сценариев – процесс проектирования остается весьма идиосинкразическим, основываясь на вкусах, интересах и убеждениях авторов. В результате у нас есть десятки языков сценариев, и я не вижу, чтобы в них нашли прямое отражение исследования, скажем, в теории типов.

В 1970-х существовали языки самых разных типов: Си и Smalltalk – очень разные языки. Сегодня набор языков изменился, но Си, C++ и Smalltalk по-прежнему входят в него. Не кажется ли вам, что можно было рассчитывать на более существенные нововведения и прогресс в области разработки языков и взаимодействия с компьютерами?

Брайан: Боюсь, у меня недостаточно информации об этой области в целом. Возможно, мы добьемся большего в своих попытках заставить машины трудиться вместо нас. Я имею в виду, что языки программирования перейдут на еще более высокий уровень и в большей мере станут декларативными, не требуя, чтобы мы подробно описывали все детали. Будем надеяться, что языки станут более надежными, предотвращающими написание неработающих программ. Возможно, появятся языки, легко транслируемые в какой-нибудь очень эффективный исполняемый формат. Честно говоря, больше ничего не могу придумать.

Возможна ли теория разработки языков? Можно ли применить к разработке языков научный метод, учитывая опыт предыдущих открытий и изобретений и двигаясь вперед? Или мы никогда не сможем избавиться от личных пристрастий разработчика?

Брайан: Думаю, личные вкусы и интуиция разработчика всегда будут играть большую роль в проектировании языков. Практически все языки – результат труда одного или двух, в крайнем случае, трех человек. Попробуйте вспомнить какой-нибудь язык, придуманный целой группой людей. Это говорит о том, что такая работа, скорее всего, требует индивидуального творчества.

В то же время мы лучше стали разбираться почти во всех аспектах языков программирования и можем рассчитывать на дальнейшее углубление наших знаний. Это заставляет предположить, что новые языки будут основываться на прочных принципах и их свойства станут понятнее. В этом смысле можно рассчитывать на более научное основание, чем, скажем, 10, 20 и уж точно 30 лет назад, когда основания практически не существовало. Но мне кажется, что разработка языков всегда будет в значительной мере определяться личным вкусом.

Разработчики будут предлагать то, что им нравится, и что-то из предложенного может также понравиться многим другим. Но языки вынуждены предлагать все больше функций, наличие которых уже воспринимается как само собой разумеющееся. Например, в любом серьезном современном языке почти обязательно должен присутствовать тот или иной объектный механизм, и его нужно встраивать с самого начала, а не «подклеивать» потом. Другая важная область – «параллелизм», потому что новые машины располагают не одним, а несколькими процессорами, и параллелизм должен стать неотъемлемой частью самого языка, а не реализовываться в дополнительных библиотеках.

Лучше разобравшись в ситуации, мы начинаем строить более крупные системы, то есть в некотором отношении наша мощь растет, но мы постоянно стремимся поднять планку еще выше. Мы собираем еще более крупные команды.

Брайан: Думаю, ваше замечание в основном совершенно справедливо. Мы постоянно стремимся создавать что-то еще более крупное, вечно забывая, зачем мы здесь. И вот, чем мощнее наши машины, чем эффективнее мы пишем программы, и чем лучше наши языки программирования, тем на большее мы замахиваемся. С задачей, которую в 1970-х пара сотен программистов решала бы год или два, сегодня один старшекурсник сможет справиться за пару недель – благодаря развитой системе поддержки и инфраструктуре, возросшей мощи компьютеров и большому объему ПО, которым можно воспользоваться. Поэтому, думаю, в каком-то смысле мы всегда будем с этим сталкиваться.

Придем ли мы к тому, что программы будут разрабатываться многотысячными коллективами, как Vista в Microsoft? Возможно, но нам явно нужно будет придумать, как разбивать большие проекты на много мелких и организовывать надежное взаимодействие между ними.

В какой-то мере это потребует усовершенствовать языки и найти лучшие технологии соединения составных частей, вне зависимости от того, на каких языках они написаны, а также передачи информации посредством интерфейсов.

Ранее вы сказали, что современные языки программирования обязательно должны поддерживать ООП. Насколько нас удовлетворяет ООП? Нельзя ли попробовать что-то еще, найти или добавить какие-то вещи, которые позволят упростить процесс создания крупных систем?

Брайан: ООП очень полезно в некоторых ситуациях. Если вы пишете на Java, у вас просто нет другого выбора; если вы пишете на Python или C++, можете применять ООП или отказаться от него. Мне кажется, что это правильная модель: в зависимости от конкретного приложения можно использовать ООП или нет. По мере развития языков, несомненно, появятся другие механизмы формирования модулей вычислений и структурирования программ.

Если взять COM, объектно-компонентную модель Microsoft, то она основана на объектно-ориентированном программировании, но не ограничивается этим, потому что компонент – не один объект, а целая их группа. Как организовать такую систему более упорядоченным образом, чем это сделано в COM, чтобы получить лучшее представление об имеющихся в ней связях?

Нужны механизмы для работы с большими количествами объектов. В результате роста размеров программ или объединения в них разнородных частей мы сталкиваемся с весьма сложными структурами объектов.

В UNIX берется язык Си, в котором не было никакой поддержки ООП, после чего строятся компоненты — объекты — в виде маленьких инструментов, легко объединяемых для выполнения сложных функций. Может быть, стоит не встраивать в язык концепцию объектов, а создавать мелкие инструменты как объекты или компоненты в виде отдельных программ? Если вспомнить, как устроена электронная таблица, то обычно это огромная программа, построенная из объектов, — возможно, с поддержкой подключаемых модулей и расширений, — но смысл в том, что объекты находятся внутри и интегрированы с помощью языка.

Брайан: Excel – прекрасный пример, потому что в нем собрано огромное количество объектов со своими методами и свойствами. Можно написать код, который будет управлять Excel, поэтому фактически Excel становится гигантской подпрограммой или просто новым вычислительным блоком. Передача данных организована значительно сложнее, чем, скажем, в конвейере UNIX, но весьма похожа, и включение Excel в состав конвейера не должно составить большого труда.

Мэшапы (mashups) близки по духу этой идее: они соединяют вместе крупные конструктивные элементы для специальных целей. Получается не так просто, как конвейер UNIX, но идея та же: объединение крупных самостоятельных объектов в еще более крупные системы.

Yahoo! Pipes – чудесный пример. Это действительно интересный ответ на вопрос «как взять достаточно сложные функциональные части и соединить их вместе?». Ко всей этой штуке они прицепили красивый графический интерфейс, но можно предположить, что то же самое достижимо с помощью текстовых механизмов, и в результате получить систему, позволяющую собирать произвольные коллекции вычислительных модулей, опять-таки с помощью текстовых программ. Явно стоит напрячься и подумать, как это сделать. Как эффективно создавать системы из существующих компонентов и какую помощь могут в этом оказать языки программирования.

В эпоху командной строки общаться с компьютером нужно было в письменном виде: вводить данные в виде текста и получать результат в виде текста. Сегодня помимо клавиатуры есть мышь, а результат мы получаем частично в графическом, а частично в текстовом виде. Является ли язык по-прежнему лучшим способом общения с компьютером? Не было ли использование командной строки в некотором отношении лучшим способом общения благодаря языку?

Брайан: Графические интерфейсы очень помогают неопытным пользователям при работе с незнакомыми системами или приложениями, которые редко используются или являются графическими по своей

природе, например создающими документ. Но через какое-то время вы обнаруживаете, что многократно повторяете одни и те же действия. Компьютеры очень эффективны при выполнении повторяющихся действий. Разве не лучше приказать компьютеру, чтобы он повторил то же самое еще раз, и еще раз?

Для этого есть специальные механизмы, например макросы в Word и Excel. Но мы видим появление программируемых API для таких систем, как Google, Yahoo!, Amazon и Facebook. Любые операции, выполняемые с помощью клавиатуры и мыши, можно автоматизировать. И для этого не нужно анализировать экран или HTML, как 10 лет назад.

В сущности, это возврат к командной строке, когда эффективнее всего оказывается обработка только текста. Допустим, чтобы выяснить, как нужно выполнять задание, нужно поработать какое-то время с клавиатурой и мышью, но когда вы обнаружите, что некоторые операции многократно повторяются, то можно механизировать процесс с помощью интерфейса командной строки и этих API, а не заставлять человека делать одно и то же.

Учитываете ли вы трудоемкость отладки каких-либо функций при включении их в разрабатываемый язык? AWK критикуют за автоматическую инициализацию переменных при отсутствии объявлений. Это удобно, но когда возникают орфографические ошибки или опечатки, их бывает очень трудно обнаружить.

Брайан: Это всегда компромисс. Компромиссы есть в любом языке, и в AWK мы шли на них, чтобы добиться как можно большей простоты. Нашей целью были однострочные программы, и мы полагали, что в основном будут писать программы из одной-двух строк. Переменные без объявлений с автоматическим присваиванием начальных значений соответствовали поставленной задаче, потому что с объявлениями и инициализацией объем программы утраивается. Для маленьких программ это очень удобно, а для больших – плохо. Ну что тут поделаешь?

В Perl есть режим с выводом предупреждений, когда вы просите компьютер сообщать вам о замеченных им глупостях. Можно обезопасить себя так, как это делается в Python: переменные нужно инициализировать, но обычно можно обойтись без объявлений. Либо можно взять отдельный инструмент, lint для программ AWK, который скажет: «У тебя тут две переменные с очень похожими именами – ты ничего не напутал?»

Более сомнительное конструктивное решение в AWK – запись конкатенации без специального оператора: ряд соседних величин просто конкатенируется. При отсутствии объявлений получается, что едва ли не

любой текст становится допустимой программой AWK. В таких условиях очень легко ошибиться.

Думаю, это пример неудачного проектирования. Ничего мы этим не выиграли: надо было использовать оператор. Но автоматически инициализируемые переменные были сознательным компромиссом проектировщиков в пользу маленьких программ, в ущерб большим.

Культура наследования

Допустим, я пишу некий новый небольшой язык для сотового телефона или встроенного устройства, который должен работать на двух мегабайтах памяти. Как подобные аспекты реализации влияют на уровень интерфейса? Когда кто-то станет пользоваться моей программой, сможет ли он понять, почему я выбрал то или иное конструктивное решение? Или такие ограничения нас уже не волнуют?

Брайан: Думаю, мы значительно меньше связаны такими ограничениями, чем прежде. Если взглянуть на историю первых программ для UNIX, к которым, конечно, относится AWK, то обнаружится, что очень часто жесткие ограничения на память отражались на языках программирования и определенных аспектах самой операционной системы.

Например, долгие годы для AWK существовали внутренние ограничения: можно было открыть лишь столько-то файлов, хранить лишь столько-то элементов в ассоциативном массиве и т. д. Все это было обусловлено тем, что памяти было очень мало, а процессы выполнялись довольно медленно. Постепенно эти ограничения отпали. В моей реализации больше нет никаких жестких ограничений. Жесткие ограничения – это место, где ограниченность ресурсов выходит на поверхность и становится видна конечному пользователю.

В AWK сделана попытка сохранить состояние переменной, поэтому когда переменная содержит число, которое преобразуется в строку для вывода, AWK запоминает текущее значение как в числовом, так и в строковом виде, чтобы не нужно было снова выполнять преобразование. В современной машине, работающей в 1000 раз быстрее прежних, нет нужды заниматься такими вещами. Вы просто преобразуете значение, когда это необходимо.

Даже изначально делать это было, наверное, глупо, потому что для такой схемы потребовался большой объем сложного кода, с очень тонкими настройками и, возможно, не всегда корректного. Сегодня я вообще не задумываюсь над этим. Уверен, что ни Perl, ни Python такими вещами не занимаются.

Как ни странно, в Perl 5 по-прежнему используется этот прием.

Брайан: Первая версия Perl была написана меньше чем через 10 лет после выхода AWK, когда еще было много ограничений на ресурсы. Во всяком случае, это примеры того, как ограниченность ресурсов заставляла вас поступать иначе, чем вы бы, скорее всего, поступили сегодня.

Если не ошибаюсь, у тех машин, на которых я начинал работать, общий объем памяти составлял 64 Кбайт. И мы тогда уже плотно работали с UNIX.

Питер Вайнбергер сказал, что в начале эпохи UNIX вы всегда считали, что если понадобится, то через год можно будет переписать программу. Необязательно было писать идеальные программы, потому что они не были ни большими, ни сложными, и их всегда можно было написать заново. У вас тоже было такое ощущение?

Брайан: Программы действительно часто переписывали. Не знаю, делали ли это с чистого листа. В своей работе не припомню случаев, чтобы я все выкидывал и писал заново. Модификации, которые я делал, чаще развивали имевшийся код, но многое переосмысливалось, и частью культуры, несомненно, был поиск путей уменьшения размера программы.

Судя по его словам, это был элемент культуры. При проектировании никто не предполагал, что программа сможет просуществовать 10, 20 или 40 лет. Заметили ли вы перемены в мышлении в пользу долгосрочных перспектив?

Брайан: Не знаю, есть ли сегодня те, кто думает о далеком будущем, но в прошлом такие люди были. Некоторые просто не могли иначе. Например, если я работал в телефонной компании и писал программы для коммутационного оборудования, то в добрые старые времена такой код создавался надолго и должен был обеспечивать совместимость с прежним кодом. Приходилось работать более осмотрительно. Возможно, мы просто были большими реалистами, понимая, что переписать его будет невозможно. Времени не хватит.

Кроме того, насколько я могу вспомнить UNIX 1970-х, появлялось столько нового и интересного, что программы постоянно модифицировались. Не думаю, что кто-то относился к своим программам как к написанным на века. Если бы в 1978 году Алу, Питеру или мне сказали, что через 30 лет мы будем обсуждать AWK, никто из нас не поверил бы этому.

Ядро UNIX действительно изменилось. Не всем это нравится, но Си по-прежнему остается одним из лучших языков для написания такого программного обеспечения, как AWK или ядро. Почему одни вещи оказываются долгожителями, а другие — нет?

Брайан: Отчасти выживание связано с тем, что эти программы действительно хорошо решают свою задачу. Си занимает прочную позицию среди языков программирования. Он чрезвычайно выразителен,

но в то же время не слишком сложен или велик, а кроме того, написанные на нем программы эффективны, и это всегда будет иметь значение на каком-то уровне. С этим языком удобно работать, потому что если нужно что-то выразить, он предоставляет для этого не так много разных способов. Я посмотрю на ваш код и скажу: да, мне понятно, что он делает. Едва ли то же самое можно сказать о таких языках, как Perl и C++. Я посмотрю на ваш код и останусь в недоумении, потому что тут есть много способов написать одно и то же.

C++ сложен и огромен, и выразить что-либо можно многими способами. Если мы с вами будем писать на C++, то можем прийти к весьма разным способам описания какой-нибудь большой задачи. В Си такого не бывает. Си сохранился потому, что у него оказалось хорошее соотношение выразительности и эффективности, и для важных приложений он остается лучшим инструментом.

Вот поэтому мы не заменили систему X Window в UNIX. Всюду используется Xlib или что-то, где используется Xlib. Какой бы вычурной ни была Xlib, она вездесуща.

Брайан: Именно так. Она делает свое дело, и достаточно хорошо. Переписывать ее с чистого листа – слишком большой труд.

Теперь, если обратиться к C++ , то при его проектировании ставилась задача – ошибочная или нет – обеспечить обратную совместимость с Си. Теоретически, если вы хотите сделать нечто взамен *X*, оно должно незаметным для пользователя образом выполнять строки программ, написанных на *X*. C++ не вытеснил Си во многих областях, хотя номинально такая задача стояла.

Брайан: Бьерн голову себе сломал, пытаясь добиться максимальной совместимости с Си. Одной из причин успеха C++ в сравнении с другими языками была хорошая совместимость на уровне как исходного, так и объектного кода, а это означало отсутствие необходимости полностью перестраивать работу, чтобы использовать C++ в среде Си.

За свои решения, связанные с совместимостью, Бьерну наверняка приходится постоянно слышать упреки, потому что людям не нравится то или иное. Но он принял эти решения совершенно сознательно, после долгих размышлений, потому что обеспечение совместимости имело важное значение и в итоге должно было оправдать себя.

Одним из крупнейших прегрешений считают чрезмерную близость к Си.

Брайан: Возможно, но чем дальше он отошел бы от Си, тем меньше были бы его шансы на успех. Здесь трудно соблюсти правильную меру, и я думаю, что он очень хорошо справился со своей задачей.

В какой мере стремление к обратной совместимости препятствует новаторству и революционным преобразованиям?

Брайан: Такая дилемма возникает абсолютно всюду, и я не вижу способа ее разрешить.

Вы не раз упоминали о том, как в малые языки начинают добавлять новые функции, превращая их в полные по Тьюрингу языки и лишая первоначальной идейной чистоты. Есть ли какие-то принципы, следуя которым можно не сбиться с пути при попытке сделать ограниченный язык более универсальным?

Брайан: Думаю, да. Я не раз об этом говорил и часто размышлял, в какой мере это было частным явлением. Все языки, с которыми я сталкивался, страдали от этого, но, возможно, других это не касалось. Может быть, дело было только во мне. Обращая взор в прошлое, я думаю, что в большинстве случаев более правильным было бы обеспечить синтаксическую совместимость новых функций с существующими языками, чтобы людям не приходилось изучать совершенно новый для них синтаксис.

Наблюдается ли возрождение малых языков или его можно ожидать в будущем?

Брайан: Не думаю, что «возрождение» здесь правильное слово, но малые языки будут создавать и в будущем.

Что могло бы в какой-то мере подстегнуть данный процесс, так это распространение API для веб-сервисов. У каждого из них есть API, позволяющий управлять этим веб-сервисом из программы, а не непосредственно. Сегодня, в основном, это API для JavaScript, но можно представить, как сделать их более доступными и управляемыми из командной строки UNIX или Windows, а не с помощью JavaScript внутри броузера, где для его запуска все равно нужно щелкнуть мышью.

Вы говорите чуть ли не о возрождении командной строки UNIX, которая станет действовать на всем пространстве Интернета.

Брайан: Неплохо сказано. А здорово было бы?

Трансформирующие технологии

Вы сказали, что с помощью уасс стало проще экспериментировать с синтаксисом языка, потому что можно изменить грамматику и запустить его снова, вместо возни с самодельным прямым нисходящим парсером. Был ли уасс трансформирующей технологией?

Брайан: Несомненно, уасс оказал огромное влияние на разработку языков. Лично я без него далеко не ушел бы в работе над языками, потому

что у меня почему-то всегда не ладилось с нисходящими рекурсивными парсерами. У меня всегда были проблемы с приоритетом операторов и ассоциативностью.

С yacc можно было не думать обо всем этом. Можно было написать разумную грамматику, а потом сказать: «Вот такими должны быть приоритеты и ассоциативность, а вот так нужно поступать в тех противных случаях, когда унарный оператор записывается одинаково с бинарным». Все это значительно облегчалось. Благодаря одному лишь этому инструменту можно было придумывать для языка такие вещи, которые иначе оказывались слишком сложными.

Конечно, yacc очень помог в разработке EQN. Грамматика была не слишком сложной, но с необычными конструкциями. Некоторые из них ранее не рассматривались в контексте языка программирования, и фактически EQN был декларативным, а не процедурным языком. В какой-то момент он даже подвергся обсуждению в CACM – по поводу хитростей с одновременным заданием у объекта подстрочного и надстрочного индекса, что вполне позволялось в грамматике yacc, но было очень трудно сделать иначе.

С теоретической точки зрения yacc был поразителен – в смысле его технологии языка, представления о том, как нужно проводить синтаксический анализ текста, превращая его в программу, – но он был и прекрасно сконструирован – по сравнению с тем, чем мы тогда располагали. В технологическом отношении долгое время не существовало ничего, что можно было бы сравнить с yacc.

lex отчасти обладал теми же свойствами, но почему-то не достиг того же уровня, возможно, потому что писать самому лексические анализаторы проще. В AWK у нас сначала был лексический анализатор lex, но со временем я решил, что его тяжело поддерживать в разных средах, и заменил самодельным лексическим анализатором на Си. Это на долгие годы стало источником всех ошибок в программе.

Какие еще технологии, помимо lex и yacc, делали разработку языков или программ легче, проще или эффективнее?

Брайан: Вообще говоря, когда вы работаете в такой операционной системе, как UNIX, все вычислительные задачи становятся проще. Возможность писать сценарии оболочки, запускать программу и перехватывать ее вывод, анализировать, возможно, редактировать, превращая в нечто иное, и это тогда, когда машины были медленными, – все это имело большое значение. В целом, наличие таких инструментов, в особенности, основных, таких как sort, grep и diff, позволяло контролировать свою работу даже в мелочах.

Не представляю, как можно компилировать программы без make, как, впрочем, не представляю жизнь без patch, а это был 1986 или 1987 год.

Брайан: До того как стал преподавать, я не использовал patch, потому что никогда не писал таких больших программ, чтобы заплатки оправдывали себя по сравнению с пересылкой всего исходного кода. Но несколько лет назад я решил, что мои студенты должны иметь представление о patch, поскольку именно с ее помощью распространяется большой объем кода, особенно для Linux. В одной из задач в моем курсе студенты должны загрузить мою версию AWK из Интернета, добавить конкретные функции вроде `repeat until`, придумать несколько тестов и запустить их с помощью сценариев оболочки, а потом отправить patch-файл. Это для них такой же опыт, как взять какую-то программу open source, немного изменить ее и послать обратно. Раньше я никогда не задумывался о применении patch. Обычно я требовал, чтобы мне присылали весь исходный код.

Заплатки легче рецензировать.

Брайан: Думаю, это тоже существенно. Patch-файлы много компактнее, и можно гораздо быстрее обнаружить, что они сделали.

Вы упомянули тестирование. Стали бы вы сейчас писать код иначе, чтобы облегчить модульное тестирование?

Брайан: Для тех программ, которые я писал долгие годы, модульное тестирование не очень оправданно, потому что эти программы очень маленькие и автономные. Идея модульного тестирования, заключающаяся в том, что вы пишете фиктивную главную функцию и помещаете в нее кучу вызовов маленьких функций, чтобы посмотреть, как они работают, для таких программ не оправданна, поэтому я не провожу модульное тестирование на таком уровне. Я пробовал несколько раз заняться им в моем курсе, и весьма неудачно.

В маленьких программах я предпочитаю сквозное тестирование методом черного ящика. Составьте набор контрольных примеров, какой-нибудь очень узко специализированный ограниченный язык, а потом напишите программу, которая будет автоматически запускать контрольные примеры и сообщать, какие из них не проходят. Это годится для отдельных частей AWK: просто прекрасно для регулярных выражений. Годится для кодировщиков и декодировщиков Base64, которые я иногда предлагаю написать своим студентам. Во всех этих случаях я выполняю внешнее, а не внутреннее тестирование. Я не помещаю объекты внутрь программы, чтобы протестировать их.

С другой стороны, сегодня я поступил бы иначе и постарался упростить проверку корректности внутренней работы, введя операторы контро-

ля и функции проверки на допустимость, добавив контрольные точки и средства вывода внутреннего состояния, чтобы это не отнимало много сил, – что-то вроде встроенных контрольных тестов, применяемых в аппаратных средствах.

В таком виде тестирование становится похожим на отладку. Возможно, между ними действительно нет четкой границы.

Брайан: Мне представляется, что идея операторов контроля связана с ситуациями, когда вы в значительной степени уверены, что все сделано правильно, но все же некоторые сомнения есть, поэтому вы готовите себе парашют, чтобы в случае аварии благополучно приземлиться. Метафора, видимо, не совсем удачна. Операторы контроля и проверки допустимости полезны тем, что если обнаружатся ошибки в работе, найти дефект программы будет гораздо проще, потому что известно, где начинать поиски. Кроме того, это подскажет, какой еще тест, возможно, следовало добавить.

Однажды я предложил своим студентам написать класс ассоциативного массива примерно того же типа, что ассоциативные массивы в AWK. Они писали его на Си, то есть можно было ожидать ошибок, связанных с обработкой строк. Создавая собственный вариант, я написал отдельную функцию контроля, которая просматривала структуры данных и проверяла, что количество элементов, полученное подсчетом изнутри структуры данных, совпадает с тем, которое получается тупым перебором снаружи.

По-моему, это похоже на варианты `malloc`, где проверяется состояние дел до и после каждой операции. Такая проверка как бы говорит: если возникнут ошибки, то они могут быть связаны с этим местом, поэтому нужно на всякий случай сделать проверку. Сейчас я бы гораздо чаще применял такие средства.

Связано ли это с вашим большим опытом разработчика, подсказывающим, где возникают ошибки, или с тем, что такой способ обходится намного дешевле?

Брайан: Я бы не стал претендовать на большой опыт разработки программ. Как правило, я пишу меньше кода, чем мне хотелось бы, да и когда пишу, он часто бывает дрянным, несмотря на мои поучения. Это, скорее, «делай, как я говорю, а не как я делаю».

Наш редактор слышал, как однажды на конференции вы расхваливали Tcl и Visual Basic. Что вы теперь думаете об этих языках?

Брайан: В начале 1990-х я немало поработал на Tcl/Tk. Я изучил их вдоль и поперек и написал ряд систем, которые хотя недолго, но использовались в Bell Labs. Я научился очень быстро создавать интерфейсы.

Tcl/Tk – прекрасная среда для создания пользовательских интерфейсов и значительный шаг вперед по сравнению с их предшественниками.

Как отдельный язык Tcl несколько идиосинкразичен. Он был хорош для тех задач, для которых создавался, но был весьма необычен, так что многие с ним помучились, и если бы не Tk, замечательно помогавший строить интерфейсы, о нем, возможно, давно забыли бы.

Visual Basic – на ранней стадии своего существования – предоставлял прекрасный язык и среду для создания Windows-приложений. В какой-то момент VB стал одним из самых популярных языков программирования. С его помощью легко было мастерить графические интерфейсы, поэтому в мире Windows он выполнял такую же роль, как Tk в мире UNIX и X11: позволял быстро создавать интерфейсы. Microsoft потихоньку убила Visual Basic, и в настоящее время я не стал бы писать на нем что-то новое. Я бы выбрал C#.

Что вы испытываете, когда у вас появляется новая функция или идея, и вам нужно предложить пользователям перейти на новую версию?

Брайан: К сожалению, эта ситуация из тех, когда нет удовлетворяющего всех выхода: всегда кто-нибудь останется недоволен. Если это моя программа, мне нужно, чтобы люди поддержали меня, а если это чья-то чужая программа, мне нужно, чтобы сохранились самые странные конструкции, если я ими пользуюсь. Я побывал и на той, и на другой стороне. Долгие годы меня удручало то, что из Bell Labs вышли разные версии AWK. Ал, Питер и я написали одну версию, а потом появилась еще одна версия под названием NAWK, другой группы разработчиков. Они хотели развивать язык в другом направлении, поэтому возникли две не вполне совместимые версии.

Закономерно. Вопрос в том, в каком случае ваша жизнь облегчится. Если удаление функции, которую трудно сопровождать или объяснять, на долгое время облегчит вам сопровождение программы, то это существенный аспект. Если переход на новую версию программы заставит вас переписать изрядный объем кода, то это другой предмет для беспокойства.

Брайан: В некоторых случаях это поправимо. Например, у Microsoft была программа для преобразования VB 6 в VB.NET. Первые ее версии плохо справлялись с этой задачей, но потом они становились совершеннее, и в какой-то момент конвертация стала вполне осуществимой.

Должен ли разработчик рассматривать создание красивого интерфейса как главную задачу реализации? Важно ли всегда помнить об этой цели или это зависит от других задач?

Брайан: Если это язык программирования, вы обязаны подумать о том, как люди будут писать на нем программы. Какие программы они станут писать? Вы должны сами попытаться написать разные программы, прежде чем что-то зафиксировать. Если речь идет об API, нужно обязательно подумать о том, как можно будет использовать этот API и как он решает сложные вопросы, такие как определение владельцев ресурсов.

У Мики Хеннинга (Michi Henning) есть чудесная статья о проектировании API, опубликованная в майском номере ACM Queue от 2007 года – я всегда перечитываю ее накануне обсуждения API в моем курсе. Он, в частности, делает вывод о возрастании значения API в связи с ростом их количества и усложнением функциональности.

Примером могут служить API веб-сервисов. Например, API для Google Maps очень разросся. Не помню, чтобы он был таким большим, когда я начал баловаться с ним три года назад: он явно расширился. Насколько я могу судить, он сделан хорошо. С другими интерфейсами работать труднее. Организация их работы – тяжкий труд. А как быть, если вы захотите что-то изменить?

Можно назначить «день флага» для обновления всех серверов.

Брайан: А может, изменить группу имен, чтобы обеспечить восходящую совместимость?

Допускают ли такие вещи дальнейшее развитие? Кажется, Стюарт Фельдман сказал, что «нельзя заменить табуляцию в Make – у нее уже 12 пользователей»!

Брайан: Да. Это один из недостатков Make, и я думаю, что Стюарт сейчас переживает по этому поводу не меньше, чем тогда. Когда есть реальные пользователи, что-то менять действительно тяжело. У Джошуа Блока есть рассуждение о разработке API, где он говорит: «API – это навсегда». Когда он сделан, менять что-либо трудно. Иногда можно сделать конвертер. Мы уже говорили о конвертере VB. Майк Леск когда-то изменил TBL. Раньше таблицы обрабатывались по столбцам, а он решил, что лучше по строкам, поэтому написал конвертер. Он работал не идеально, но его было достаточно, чтобы взять имеющуюся таблицу и преобразовать ее в новую. Иногда такой метод помогает. Есть транслятор из AWK в Perl, с довольно ограниченными возможностями, но для начала и такой годится.

Каков главный урок, извлеченный вами за долгие годы работы?

Брайан: Тщательно обдумывайте то, что хотите сделать, но потом долго пробуйте в работе и так и сяк, пересматривайте и исправляйте, пока не удовлетворитесь сделанным. Не выкладывайте для всех первый же вариант.

Встречаются системы, которые создают впечатление, что авторы опубликовали их первую версию. Конечно, она будет сырой! Возьмем такого гения, как Бетховен. В его рукописях сплошные исправления. Наверное, единственным композитором, который мог с первого раза написать идеальное произведение, был Моцарт.

Одно дело — поразительные работы гениев, рождающихся раз в столетие, другое — все остальные, в том числе наши.

Брайан: В своей автобиографии Айзек Азимов утверждал, что он просто писал свои тексты и отдавал в печать, и большинство их действительно вполне приличны. Он сказал, что никогда не исправлял написанного, и он мог себе это позволить, но не думаю, что это норма для большинства.

Здесь в университете на одной стене висит стихотворение Пола Малдуна, напоминающее рукописи Бетховена. Лист пестрит вымарками и вставками; кто-то поместил его в рамку и повесил на стенку в качестве напоминания о том, как трудно что-то сделать правильно с первого раза. В программировании то же самое. Не выкладывайте сразу первое, что у вас получилось.

Мелочи, изменившие мир

Правда ли, что AWK начался с того, что вы с Алом Ахо обсуждали, включать ли в ваш проект БД парсер для расширяемых языков?

Питер Вайнбергер: У меня другие воспоминания по этому поводу, хотя память может мне изменять. Я был сотрудником отдела обработки данных (на компьютерах Univac), а Ал и Брайан хотели добавить к командам UNIX что-то в стиле работы с базами данных. Вполне возможно, что вначале у них были более амбициозные планы, но мне запомнилось, как вскоре мы выбрали в качестве перспективного направления сканирование данных.

Почему вы решили создать средство для извлечения информации из файлов? Почему не предпочли, скажем, функцию вставки данных?

Питер: Одной из общих черт утилит командной строки UNIX было то, что они действовали с файлами, составленными из строк (в те времена это был ASCII). Вставку можно и нужно было выполнять с помощью редактора, а другие способы модификации файла обычно означали, что нужно создать новый файл с измененным содержанием. Возможны были и другие пути, и ими занимались, но они были в стороне от главного направления.

Я слышал, что вы занялись чтением данных, потому что не хотели решать проблемы одновременной записи данных.

Питер: Это не совсем точно; на деле получилось не так. :)

Сегодня вы бы приняли такое же решение?

Питер: Нет, думаю, что если бы мы создавали систему сегодня и не ставили перед собой нескромных целей, то вряд ли пользователь обнаружил в ней внешние признаки параллелизма, хотя мы наверняка воспользовались бы многоядерной архитектурой или иными возможностями для выполнения параллельных вычислений. Несомненно, это добавило бы нам проблем, но мы с ними справились бы. Здесь возникает или может возникнуть интересный вопрос: повлияло бы это на конструкцию языка, и если да, то в какой мере?

Не знаю, над этим надо думать. Если вы считаете, что процессор не будет загружен полностью и свободного времени окажется много, можно пойти разными путями. Во-первых, можно отказаться от его использования – оставим его другим задачам. Для AWK или аналогичной системы это не самое плохое решение, потому что если вы считаете, что она в основном будет использоваться в конвейерах, то и другим участникам конвейера потребуется процессорное время.

Напротив, если вы предполагаете, что система будет применяться для относительно сложного преобразования файлов, можно обеспечить ее средствами для одновременного выполнения на нескольких процессорах, чем мы, конечно, не занимались, потому что тогда таких машин не было.

Как вы считаете, в каком контексте AWK может оказаться предпочтительнее, скажем, SQL?

Питер: Ну, это по существу несравнимые вещи. В AWK нет явных типов, а SQL ими просто перегружен. Далее, AWK читает и пишет строки, но готов рассматривать некоторые строки как числа, если его об этом попросить. SQL выполняет объединения, а чтобы то же самое сделать в AWK, нужно сначала пропустить данные через другую программу, может быть, join. SQL выполняет сортировку и агрегацию, а в UNIX-контексте для этого сначала применяют sort, а затем снова подают на вход AWK или другой команды UNIX. Короче, AWK предназначался для применения в цепочке команд, связанных конвейером; SQL предназначался для работы с данными, которые хранятся в закрытой структуре, а пользователю предоставляется лишь некоторая ее схема. Наконец, на оптимизацию запросов SQL потрачены годы труда, в то время как в AWK все осуществляется наглядно.

Какие удобства вы усматриваете в том, чтобы хранить журналы (UNIX) в текстовых файлах и обрабатывать их с помощью AWK?

Питер: Текстовые файлы дают большие преимущества. Вам не нужны специальные средства для их просмотра, и вам на помощь придут все известные команды UNIX. Если этого недостаточно, легко преобразовать эти файлы и загрузить их в какую-то другую программу. Это универсальный тип данных для ввода в самые различные программы. Кроме того, они не зависят от порядка байтов в архитектуре процессора. Возможна такая простая оптимизация, как хранение файлов в сжатом виде, что требует помнить, какая команда использовалась для сжатия, а вариантов обычно предоставляется несколько. Что касается обработки их с помощью AWK, то для получения нужного результата бывает достаточно цепочки из нескольких команд. Если нет – текстовые файлы легко можно прочесть с помощью таких языков сценариев, как Perl или Python. Если этого мало, можно привлечь Си или Java.

Текстовые файлы прекрасно подходят для журналов. Когда-то в качестве аргумента против них приводились необходимость выполнить разбор, преобразовать числа в двоичный вид и т. п. Но преобразование практически не отражается на времени работы процессора, а синтаксический анализ текстовых строк не сравним по сложности с анализом XML. С другой стороны, двоичные структуры фиксированного размера не требуют анализа, но они встречаются редко, и это тот нечастый случай, где преимущество не на стороне текстовых файлов.

AWK стал одним из первых доказательств мощи применяемой в UNIX идеи объединения множества мелких программ в единое целое. Обычно это программы для обработки текста. Применима ли эта концепция к нетекстовым данным и мультимедиа?

Питер: Полезно разобраться, в чем же действительно заключалась «концепция UNIX». Это был стиль, когда многие полезные программы имели один вход и один выход и использовали синтаксис командной строки, а система обеспечивала единообразную поддержку ввода и вывода (системные вызовы `read` и `write`, независимо от устройства) и конвейеры (pipes), избавлявшие от необходимости заботиться об именах и размещении временных файлов. Примеры областей, в которых применение этой идеи идеально: перекодировка и сжатие данных, даже если они являются звуком или видео. Но даже если ограничиться текстом, есть масса приложений, которые работают иначе, особенно если они предполагают интерактивное взаимодействие с человеком. Например, команда `spell` выводила список слов, которые, по ее мнению, были написаны с ошибками, но она не была интерактивной: пользователь должен был вернуться и отредактировать свой документ.

Поэтому смысл вашего вопроса мог быть таков: если у нас есть только командная строка, с помощью каких команд можно обработать мультимедийные данные? Но это не конструктивно. Сейчас есть другие спо-

собы взаимодействия с программами и другие возможности для разбиения задач. Они не обязательно лучше или хуже старых – просто другие. Сравните, например, TeX и Word. Разве одна из этих программ лучше другой? Сомневаюсь, что по этому вопросу будет единогласное мнение.

В чем вы видите ограниченность командной строки и графических интерфейсов?

Питер: Это старая тема, и границы между тем и другим стали менее четкими. Наверное, по этому поводу можно высказать глубокомысленное суждение. Но вот вам поверхностный ответ. Когда мне нужно объединить группу программ, удобно использовать сценарий оболочки для вызова инструментов, работающих из командной строки. Он также позволит согласовать параметры и настройки различных компонентов. Но графические интерфейсы оказываются гораздо удобнее, когда у вас не слишком много параметров и нужно выбрать для них значения, а также могут быть эффективнее для упорядочения всей информации.

Многие из собеседников подчеркивали, как важно хорошему программисту знать математику. Интересно, можно ли получить нужные знания именно в тот момент, когда в них возникнет потребность? Например, с помощью Интернета можно довольно быстро найти и узнать все необходимое.

Питер: И да, и нет. К несчастью, для того чтобы освоить некоторые вещи, нужно не только получить о них информацию, но и практически поработать с ними, поэтому в одном случае вы зайдете в Интернет, прочтете то, что нужно, и этого будет достаточно, а в другом ничто не заменит вам годы напряженной работы. Скажем, в разгар работы над каким-то проектом вы приходите к выводу, что для решения вашей задачи нужно разбираться в линейном программировании; то, что вы найдете в Интернете, может не удовлетворить вас, а если на решение задачи у вас всего неделя, вряд ли вы выберете метод, изучение которого требует большого труда, даже если он гораздо эффективнее, – если только вы не были знакомы с ним ранее. Бывает и так.

Какова роль математики в компьютерных науках, в частности в программировании?

Питер: У меня диплом математика, поэтому мне хотелось бы верить, что математика – основа всего. Но есть немало областей компьютерных наук и немало видов программирования, где можно добиться неплохих успехов без всякой математики. Применение (или применимость) математики имеет несколько уровней. Тот, кто не понимает статистики и случайности, будет раз за разом ошибаться, интерпретируя данные окружающего мира. Математика нужна в графике, в очень большой мере – в машинном обучении (которое статистики, наверное,

представляют как вид регрессии), а многочисленные разделы теории чисел – в криптографии. Не имея некоторых математических знаний, люди просто лишены возможности понять большие разделы компьютерных наук.

Чем доказательство теоремы отличается от построения реализации?

Питер: В самом общем виде, когда вы доказываете теорему, у вас уже есть какие-то знания об окружающем мире, но нет полной уверенности в них. Это безусловное знание. Когда же вы пишете программу, то получаете возможность делать то, чего, вероятно, не могли раньше. В некотором смысле вы изменяете действительность. Обычно это совсем небольшое изменение. Математика и программирование очень сильно различаются. Наверно, легче всего это заметить, сравнив математическое доказательство теоремы с программой, генерируемой инструментом для автоматического доказательства теорем. Математические доказательства отличаются краткостью и часто отражают озарение. В машинных доказательствах вы ничего этого не найдете. Написание программы напоминает такие машинно-сгенерированное доказательство: нельзя ошибиться ни в одной мелочи, что требует от программиста больших умственных усилий, как и мастерства в проведении тестирования.

Открываете ли вы для себя что-то новое в процессе реализации?

Питер: Конечно. Обычно открытие состоит в том, что нужно все выкинуть и начать реализацию заново. В каждом проекте есть десятки конструктивных решений, и поначалу невозможно отдать предпочтение какому-либо одному из них, либо выбор осуществляется на основе интуиции. Практически всегда, когда дело доходит до реальной работы кода, выясняется, что можно было выбрать более удачное решение. А с течением времени, когда накапливается опыт работы кода в неожиданных ситуациях, все больше решений начинают казаться плохими.

Есть ли польза от функционального программирования?

Питер: Если смысл вашего вопроса в том, способны ли функциональные программы как более математические в каком-то отношении лучше описывать результаты, чем обычные программы, то я не вижу особой разницы между теми и другими. Языки с однократным присваиванием проще обсуждать, но это не значит, что проще писать на них программы, и убедительных подтверждений этому нет. На самом деле, большинство сравнений языков, технологий кодирования, методологий разработки и техник программирования проводятся на удручающе низком научном уровне.

Вот цитата из «Шарлатанской науки» Р. Боселла (R. Bausell «Snake Oil Science», Oxford University Press):

> Тщательно проводимые исследования (например, рандомизированные контролируемые испытания) с получением численных данных позволяют с большей надежностью определить, что действенно, а что – нет, чем если основываться на мнении экспертов, опыте, догадках или поучениях заслуженных людей.

Программирование по-прежнему остается ремеслом, как, скажем, изготовление мебели. Есть Чиппендейлы, мастера и рядовые кустари. Но я несколько отклонился от вашего изначального вопроса.

Что бы вы посоветовали тому, кто хочет усовершенствоваться как программист?

Питер: Как насчет «изучить математику»? Ладно, лучше по-другому. Как насчет «изучить числа с плавающей запятой»? Пожалуй, и это не то. Разным людям требуется разное.

Думаю, важно изучать новые технологии и алгоритмы. Без этого, как мне кажется, человек вскоре чрезмерно специализируется, теряя широту кругозора. Кроме того, в наше время нужно уметь писать безопасный и надежный код. Пользователи и системы подвергаются многочисленным атакам, и хотелось бы гарантировать, что опасность исходит не от твоего кода. Особенно сложно это сделать для веб-сайтов.

Когда нужно обучать отладке? И как?

Питер: Обсуждение отладки должно быть неотъемлемой частью всех курсов программирования (как и процедуры разработки языка). Писать корректные последовательные программы для отдельных машин достаточно трудно. Но писать код с множественными потоками исполнения еще труднее, при этом средства отладки пока оставляют желать лучшего. При проектировании нужно стремиться к упрощению отладки. Не слишком большим преувеличением будет сказать, что в каждый момент времени программист либо решает, что делать дальше, либо занимается отладкой. Все остальное занимает незначительное время.

Что вы считаете своей самой большой ошибкой за все то время, что занимались разработкой или программированием? Какой урок вы из нее извлекли?

Питер: Не знаю, была ли одна самая крупная ошибка. Люди постоянно ошибаются. Ошибки позволяют выработать некоторые полезные принципы разработки (даже если вы не можете их четко сформулировать). Но если слишком широко их применять, они вас подведут, и тогда вы либо учтете в своих принципах новые полученные уроки, либо ваш код всегда будет носить шрамы ваших изживших себя правил разработки. Заметив, что включил в сообщения об ошибках недостаточно полезной информации, я часто возвращаюсь к ним и делаю более подробными. Типичное противоречие: когда возникает ошибка, вам нужна как мож-

но более полная информация о ней. Если она не возникает, жалко потраченного труда и места на экране. Приходится искать компромисс.

О чем вы больше всего сожалеете в отношении AWK?

Питер: Думаю, безумная идея использовать пробельный символ для конкатенации строк оказалась не столь удачной, как мы рассчитывали. С явным оператором было бы лучше. Кроме того, синтаксис в целом страдает от противоречия между поддержкой коротких командных строк и возможностью написания больших программ. Сначала мы не задумывались о втором аспекте, поэтому некоторые наши решения были неудачны.

Что оказалось популярным (или полезным) неожиданно для вас?

Питер: Весь язык в целом оказался гораздо популярнее, чем мы – или я, во всяком случае, – рассчитывали. Одной из идей, которыми мы руководствовались при разработке языка, была легкость его изучения теми, кто имеет представление о работе в UNIX, – в частности, о Си и grep. Это не предполагало его привлекательности для массы секретарш (как их тогда называли) или фермеров. Но как-то в начале 1990-х я встретил на свадьбе фермера-овцевода, который вел свой учет с помощью UNIX и был большим любителем AWK. Подозреваю, что к данному моменту он продвинулся еще дальше.

Как вы стимулируете творчество в команде разработчиков ПО?

Питер: Лучший способ создать ПО высокого качества – привлечь талантливых специалистов, имеющих представление о том, что у них должно получиться в итоге. Есть и другие пути, но они более обременительны. Не представляю, как можно создавать хорошие программы без талантливых программистов, хотя допускаю, что это возможно.

Как у вас была организована групповая работа при разработке языка?

Питер: Мы вместе обсуждали синтаксис и семантику, а затем каждый писал код. После этого любой из нас мог модифицировать этот код. У нас почти нет кода, написанного кем-то одним, хотя на протяжении многих лет его сопровождал Брайан. Кроме того, мы не замахивались на большие задачи. Думаю, этому способствовала машина, для которой все делалось: у нее было всего 128 Кбайт памяти.

Для проектирования мы собирались вместе, вели обсуждение и писали на доске, но потом при кодировании случалось обнаружить, что мы упустили что-то важное. Тогда возникало неформальное обсуждение.

Если в базовом коде обнаруживались повторяющиеся проблемы, как вы определяли, что лучше: решать их отдельно в каждом случае или внести глобальное изменение?

Питер: Программные проекты бывают двух видов: те, которые оканчиваются провалом, и те, которые превращаются в кошмар сопровождения старого кода. Единственный способ избежать второго варианта – написать код заново, если изменится его вычислительная среда. Беда в том, что в большинстве проектов такая роскошь недоступна, поэтому под давлением обстоятельств приходится писать местные заплатки. Когда их накапливается достаточно много, код становится негибким и действительно сложным для сопровождения. Переписать код заново тоже непросто, если нет первоначальных разработчиков или исключительно добротных спецификаций. Не все так просто в жизни.

Если бы вам предложили дать читателям один главный совет на основании вашего опыта, что бы вы им сказали?

Питер: Я бы процитировал – возможно, неточно – Эйнштейна: «Делайте как можно проще, но не проще того».

Нельзя потакать своим желаниям, хотя это нелегко. Если совершенно ясно, что люди начнут просить о некоторой функции, то можно ее включить. Требуется здравый смысл, чтобы сделать просто, но не проще, чем это определяется необходимостью, о чем и говорится в этой цитате.

Самое простое, что сможет работать? По-моему, это сказал Кент Бек. Как добиться простоты и отказаться от того, в чем нет прямой нужды?

Питер: Это зависит от того, кто находится перед вами. Во многих случаях есть хорошая проверка: «Будет ли это понятно вашим родителям?» Не всегда это удается, но мне кажется, что это может быть хорошей отправной точкой. При более общей проверке стоит задуматься о тех, на кого рассчитана ваша функция: «Поймет ли это средний пользователь?», а не «Найдутся ли пользователи, способные в этом разобраться?»

Теория и практика

До работы в Bell Labs вы преподавали математику. Нужно ли преподавать компьютерные науки так же, как преподают математику?

Питер: Есть разные цели преподавания математики. Одна из них – подготовка будущих математиков, и это примерно то, чем я занимался как преподаватель. Другая – обучение математике с учетом ее практического применения. Но мне кажется, что несколько проще определить то, чем занимается математика, а не компьютерные науки. Компьютерные науки охватывают разные типы программирования, и трудно понять, как их оценивать. Тут вы сталкиваетесь и с разными структурами данных, и с разными алгоритмами и оценками сложности. Не вполне понятно, что может понадобиться разным людям, применяющим ком-

пьютерные науки, в отличие от сложившихся представлений о том, что может потребоваться потенциальным пользователям математических знаний. Словом, при обучении математике вы знаете, что потребуется инженерам: в данное время, полагаю, известно, что нужно тем, кто будет заниматься статистикой или экономикой, но мне кажется, что задачи математиков несколько проще.

С другой стороны, мне кажется, что специалисты в компьютерных науках должны лучше знать математику, и это пережитки той эпохи, когда я был математиком.

Итак, есть эта проблема и то, что можно вольно назвать окружающей действительностью: факультеты компьютерных наук, во всяком случае, в Америке, испытывают некоторые трудности в привлечении студентов по своему главному профилю, по крайней мере, в последние годы – по непонятным причинам, – но те, которые справились с этой проблемой, существенно изменили состав преподаваемых дисциплин. Таким образом, содержание подготовки в компьютерных науках претерпевает изменения.

Из ваших ответов складывается впечатление, что вы считаете самым подходящим для программирования некое промежуточное место между чисто теоретическим подходом, который иногда слишком отрывается от практических потребностей, и чисто практическим подходом, когда задачу можно решить, сложив вместе куски кода, взятые из разных источников. Так?

Питер: Да, но мне кажется, что главная проблема в том, что очень трудно правильно провести границу. Все зависит от того, какие надежды вы возлагаете на свой код. Если вы хотите, чтобы им пользовались долго, то нужно писать его так, чтобы было легко исправлять ошибки.

Другая трудность связана с тем, что если вы слишком быстро набираете слишком много пользователей, исправлять конструктивные ошибки становится очень сложно. Если я пишу программу для себя, то всякий раз, когда мне что-то не нравится, я просто беру и переделываю это. Если программа написана для относительно небольшой группы пользователей, то люди не сразу начинают жаловаться на несовместимые изменения, поскольку знают, что программа экспериментальная. Но если вы пишете программу для многочисленных пользователей или у нее образовалась большая группа пользователей, то вносить несовместимые исправления становится гораздо сложнее, и вам приходится сохранять решения, сделанные раньше.

Возможно, это одна из проблем со старыми программами, когда берут части кода из разных источников, и проблемы, связанные с этим кодом, размножаются и живут десятилетиями.

Питер: Да, мне кажется, что до сих пор используется много кода, написанного давным-давно и не рассчитанного на столь долгое применение.

Один из факторов долголетия AWK состоит в том, что часто берут чьи-то готовые скрипты и модифицируют их для решения других задач.

Питер: Да, это так, и такая цель ставилась при разработке. Примерно таким мы и предполагали использование кода – в значительной мере. Мы думали, что можно будет брать код, который делает почти то, что вам требуется, и лишь дорабатывать его.

Применима ли эта идея программирования на примерах в больших проектах?

Питер: Думаю, они не могут быть слишком большими, потому что пример должен быть достаточно мал, чтобы увидеть и понять. Проще всего это делать на уровне нескольких строк кода. Может быть, на уровне одного экрана, чтобы можно было проследить, что там происходит. Все должно быть настолько просто, чтобы, взглянув на код, можно было понять, что в нем нужно изменить, или хотя бы увидеть, что нужно изменить, чтобы поэкспериментировать с кодом и убедиться, что вы правильно его поняли.

Идея писать короткие «одноразовые» скрипты весьма соблазнительна. Опыт работы с большим базовым кодом и другими языками программирования научил вас распознавать, когда можно переработать базовый код, а когда нужно писать его заново?

Питер: На практике тяжело начинать переписывать все заново. Если сообщество ваших пользователей невелико, можно с ними переговорить. Либо можно переписать, если у вашего кода хорошо проработан интерфейс. Если интерфейсы не очень корректны, а сообщество пользователей велико, очень трудно вносить изменения, ничего при этом не нарушая. К сожалению, это так, даже если вносимые изменения не слишком кардинальны. На самом деле, это даже хорошо, поскольку если почти любые изменения нарушат нормальную работу, пользователям не станет намного хуже, если ваши изменения окажутся весьма значительными. По прошествии нескольких лет новый код практически всегда требуется полностью переписать заново. Пользователи начинают применять код такими способами, которые не приходили в голову разработчикам, и многие решения в реализации оказываются неэффективными, особенно на новой аппаратной базе.

Ситуация с AWK сложилась несколько иначе. Мы действительно переписывали его несколько раз заново, но потом объявили, что с этим покончено. Было что модифицировать, но все наши идеи по этому поводу оказывались несовместимыми с базовыми принципами. Я почти уверен, что это было правильное решение. Все мы занялись другими ве-

щами, вместо того чтобы расширять возможности системы. Единственное, чего, по-моему, не хватает AWK в той маленькой нише, которую он сегодня занимает, это способности работать со входными данными в кодировке UTF-8.

Брайан сказал, что вы очень быстро писали код. В чем ваш секрет?

Питер: Едва ли здесь есть какой-то секрет. Просто все люди разные. Например, я не уверен, что если бы я писал то же самое сейчас, то делал бы это столь же быстро. Отчасти это был оптимизм невежества. То есть уверенность, что можно просто написать код и этого будет достаточно. Отчасти это степень владения имеющимися инструментами и языком. Одним удобно работать с инструментами, другим – нет. Это как умение подбирать краски при работе акварелью: одни это делают с легкостью, другие – с трудом.

Я считаю верным следующее: если вы собираетесь писать код профессионально, зарабатывая этим на жизнь, то должны делать это с достаточной легкостью, иначе вы обречены на мучения. Это все равно что писать рассказы: если вы не можете с легкостью делать это, обеспечивая некоторый приемлемый уровень, то, как мне кажется, вам всегда будет очень трудно этим заниматься, хотя я здесь и не специалист. Чтобы придать вещи законченную форму, нужно много труда.

Пишете ли вы прототипы, чтобы затем превратить их в код профессионального качества? Или вы сначала экспериментируете с идеей, а потом переписываете все начисто, чтобы получить окончательный результат?

Питер: Нельзя заранее предвидеть, как пойдет дело. Приступая к созданию прототипа, можно иногда определить, на какие компромиссы вы идете. Иногда эти компромиссы таковы, что вам будет трудно преобразовать прототип в конечный продукт. Во всяком случае, есть множество вещей, которые могут осложнить такое преобразование, и тогда вам придется все переписывать заново, но иногда удается как-то постепенно трансформировать прототип. Нужно быть готовым к тому, что прототип придется выкинуть и написать все заново.

Прежде всего, маловероятно, что большинство ваших решений сразу окажутся правильными. Вы что-то напишете, поэкспериментируете с ним и в результате измените. Через некоторое время – разве что вам очень повезет – код начинает выглядеть ужасно. Ему действительно понадобится рефакторинг, но, скорее всего, лучше просто его переписать. Этого следует ожидать: все придется написать заново. И конечно, первая реализация AWK была чисто проверкой идеи, потому что она генерировала Си-код; и это совершенно не соответствует тому, какое применение вы задумывали.

Том Курц, создатель Бейсика, сказал, что когда вы пишете код, перед вами открываются те аспекты задачи, над которыми вы не задумывались.

Питер: Это верно – то, над чем у тебя не хватило ума подумать, пока не столкнулся непосредственно. По-моему, когда собираешься принять кого-то на работу, то смотришь, служит ли для этого человека код естественной формой выражения мысли, выражает ли он таким способом свои алгоритмические идеи.

В чем различие между написанием программы и созданием языка?

Питер: В некоторых отношениях писать язык проще, чем обычное ПО, но, возможно, я не прав. Мне кажется, что принимаемые решения концентрируются, потому что они должны быть согласованными, и есть относительно немного способов решения каждой задачи. После того как принято решение относительно главных характеристик языка, в голове возникает основная часть общей схемы: как будут работать функции, будет ли осуществляться сборка мусора и прочее. Из каких примитивов состоит язык? Реализация осуществляется слоями. Мне кажется, что так обычно немного проще. Конечно, когда, углубившись достаточно далеко, обнаруживаешь, что было принято действительно неудачное решение, приходится выбрасывать все сделанное.

Влияет ли реализация на конструкцию языка?

Питер: Разумеется. Думаю, это заметно вне всяких сомнений. Мне кажется, например, что долгое время сборка мусора была относительно редким явлением. Ею занимались разработчики Лиспа и кое-кто из работавших в функциональном программировании, а все остальные лишь наблюдали, поскольку было неясно, как это сможет работать, скажем, в таких языках, как Си. Но потом разработчики Java объявили, что сделают это. Это был компромисс совершенно иного рода, в виде относительно небольшого изменения в характеристиках языка; я не утверждаю, что именно так произошло с Java, но, отказавшись от доступности фактических адресов памяти для программистов, можно было попробовать осуществить сборку мусора или сжатие сборщиков мусора и так далее.

Думаю, сейчас языки без сборки мусора значительно проигрывают, несмотря на все ее несовершенство. Распределение памяти – очень утомительная задача, никогда не бывающая тривиальной, но сейчас значительно больше известно о том, как реализовывать языки, и вариантов выбора стало гораздо больше, особенно если создавать что-то облегченное.

Если вам нужна в языке функция, которую трудно реализовать, то неизвестно, стоит ли ей заниматься, потому что реализация сложна, а вы

делаете что-то простое. Если вы пытаетесь создать язык, в котором можно делать непростые вещи, то, наверное, нужно составить перечень того, что вам нужно сделать, и смириться с тем, что это может доставить вам немало хлопот.

Как язык влияет на продуктивность программиста? Насколько важны способности программиста?

Питер: Бог мой, если бы я это знал! Раньше мне казалось, что я знаю ответы на такие вопросы. Ясно, что программисты сильно различаются по способностям. Разница бывает на порядок, а может быть и гораздо больше. Это же технология программирования, и никаких эмпирических свидетельств здесь нет, но по моему мнению, язык не должен играть роли. То есть если группа работает над каким-то проектом, то не важно, какой там язык, хотя для отдельных программистов, как мне кажется, это важно. Думаю, человек благодаря своим личным качествам, или в силу того, что он изучал сначала, или по каким-то другим причинам легче осваивает один язык, чем другие. Отсюда всякие забавные споры.

Понятно, например, что есть такие приложения на Лиспе, функциональность которых очень трудно было бы реализовать на Си, и наоборот, приложения на Си, которые трудно было бы написать на Лиспе. В то же время есть широкий круг приложений, где можно использовать самые разные языки, но даже опытные программисты с разным успехом применяют разные языки, и причины мне непонятны. Разумеется, нужно время, чтобы глубоко освоить язык. Но кто-то один язык осваивает быстрее, чем другие.

Ведутся широкие споры по поводу языков и того, какие из многочисленных предлагаемых функций нужно реализовывать, а какие не нужно, и как плохо, когда их нет, и так далее. Но неясно, действительно ли в этом есть смысл. Выскажу еще одну спорную мысль: программу для марсианского спускаемого модуля можно написать на любом языке; в каждом случае у нее будут свои особенности, но они в большей мере будут зависеть от того, кто писал программу и как была организована работа, чем от выбранного языка. Каждый будет энергично защищать свой выбор, но не думаю, что он имеет значение.

Си не поддерживает объекты, но при этом практикуется создание небольших инструментов — компонентов UNIX, которые можно совместно использовать для создания сложных функций. В какой мере эффективнее создавать объекты внутри языка как части большой программы, чем строить компоненты, входящие в состав системы?

Питер: Здесь у меня возникает два вопроса. Первый касается связывания или модулярности. Вопрос в том, что делать с помощью языка,

а что – пытаясь комбинировать инструменты. Более тесной и, конечно, более сложной связь между компонентами окажется, если все они будут реализованы в языке. Отчасти это обусловлено просто эффективностью вычислений, но, мне кажется, здесь и проблема концептуальной последовательности.

Другой вопрос касается идеи «объектной ориентированности» в целом, и иногда восторги по поводу того, насколько удачной оказалась объектная ориентированность в том или ином случае, кажутся мне чрезмерными. Оставим в стороне этот несколько спорный вопрос, но ведь относительно множества языков утверждается, что они «объектно-ориентированные», а когда присмотришься к ним, оказывается, что они делают совершенно разные вещи. Не всегда понятно, что понимают под этим термином. Фактически не прекращаются путаные споры, потому что всегда есть соблазн объявить, что тот способ работы с объектами, применяемый в твоем языке, как раз и означает «объектную ориентированность». Думаю, на самом деле у этого понятия нет простого, относительно прямого и общепринятого определения.

Как выбор языка влияет на безопасность кода?

Питер: Несомненно, какая-то поддержка многочисленных аспектов безопасности нужна. Я считаю, что с безопасностью программ связано, грубо говоря, два вида проблем. Первый вид – это определенные логические ошибки, в результате вы вводите в программу данные, и она ошибочно поднимет ваши права доступа или делает еще что-то, чего делать не должна. Другой вид – все эти штуки вроде переполнения буфера и прочие ошибки реализации, на которых основывается взлом, реальные дефекты как следствие недомыслия. И мне кажется, что их по большей части просто не должно быть. Языки низкого уровня заключают в себе возможность переполнения буфера при небрежно выполненном программировании, поэтому обеспечить безопасность нелегко.

Кажется, какое-то время ходили слухи, что Microsoft при создании системы, превратившейся в Vista, собиралась всюду заменить Си и C++ на C#, после чего у вас не было бы никаких переполнений буфера, потому что их просто не могло быть. Но, конечно, этого не произошло. Взамен выполняемые модули на машинном языке подвергаются изощренному контролю в надежде помешать применить переполнение буфера и аналогичные вещи.

Но есть еще один вид проблем безопасности, появляющихся на стыке между программами из-за того, что ряд интерфейсов определен не очень корректно или весьма неформально, – таких как межсайтовый скриптинг для HTTP и XML и аналогичные. С безопасностью нужно что-то делать, но что – я не знаю.

Есть ли польза от того, что язык препятствует возникновению определенного рода ошибок?

Питер: Да, конечно, но, как мы уже говорили, неизвестно, в какой мере. Было время, когда я довольно много писал на Python, и у меня возникали удивительные ошибки – разумеется, по недомыслию и в результате дурного стиля. Это было связано с правилами отступов в Python, и когда цикл становился слишком длинным – у меня был вложенный цикл, – то для перехода в конец цикла мне нужно было вернуться на два отступа, чтобы выйти из цикла и выполнить некоторые действия, а я возвращался не на два отступа, а на один, потому что так мне казалось правильным на экране; в результате я выполнял трудоемкие операции при каждой итерации внешнего цикла, что было совершенно не нужно. Программа работала корректно, но очень медленно.

Мораль: как бы тщательно ни был разработан язык, всегда найдется место для глупых ошибок программиста. И я не знаю, есть ли какие-то научные подходы к тому, как сделать эти ошибки менее вероятными. Технологии программирования во многих отношениях являют собой жалкое зрелище, потому что в значительной мере основываются на случайностях, личных мнениях и даже эстетических пристрастиях. Не знаю, какая часть критериев, участвующих в обсуждении языков программирования, прямо и непосредственно связана с написанием корректных программ, или программ, удобных в сопровождении, или программ, которые легко модифицировать.

Научные исследования обычно помогают реализации, но в особенностях проекта обычно отражаются личные предпочтения разработчика.

Питер: Да, и на самом деле мне кажется, что в действительно удачных языках всегда есть то, что не является непосредственно реализацией опубликованных научных результатов. Кто-то решил, что ему будет интересно этим заняться. То, что вы включаете в язык программирования, может быть темой для размышлений и обсуждений по поводу языков, но непонятно, что нужно делать в отношении языков или приемов работы с ними, чтобы код стал лучше и улучшилась процедура программирования и сопровождения. У всех есть свои твердые взгляды по этим вопросам, но мне неясно, почему мы должны их разделять. Не вижу под ними никаких научных оснований.

Применять научный подход к разработке языка трудно, в частности, потому, что нет научного подхода к оценке того, насколько хорош или плох тот или иной язык.

Питер: Да, думаю, это так – и к оценке программирования в целом, не только языков. Многие считают, что они знают, как должно быть пра-

вильно, но я не вижу причин им верить, поскольку очевидно, что есть много разных способов успешной разработки программ.

Как вы выбираете для языка нужный синтаксис? На что больше ориентируетесь — на крайности или на среднего пользователя?

Питер: Ответ не будет неожиданным: на то и другое. Синтаксис должен быть настолько легок для восприятия, насколько это возможно для человека с его ограниченным интеллектом, но и в сложных случаях семантика должна быть понятна. Если язык получит распространение, им будет пользоваться множество людей, большинство из которых не разделяет точку зрения разработчика и его эстетические вкусы, и желательно, чтобы их не ввели в заблуждение необычные функции или граничные случаи. Кроме того, появятся программы, которые генерируют программы на новом языке, что станет неожиданностью для реализации.

Когда вы проектируете язык, учитываются ли вопросы отладки при оценке возможных функций?

Питер: Это сложная проблема. Хотелось бы рассчитывать, что среда разработчика в достаточной мере будет поддерживать то, что можно легко сделать. Мы знаем, как организовать предложение вариантов завершения и подсказку по параметрам, показать другие справочные сведения, если нужно, найти определения. Конечно, труднее найти функцию, которая выполняет нужное действие, но вы не знаете ее названия, – например, вы знаете, что где-то должна быть функция для форматирования чисел в машинном представлении и расстановки запятых или что-то в этом роде. Но как запомнить ее имя? Как искать имена таких функций? Поэтому авторы таких библиотек пытаются применять правила образования имен, неофициальные правила и тому подобное. Но часто такие приемы малоэффективны для больших библиотек.

А что вы скажете о текстах сообщений об ошибках?

Питер: Они оставляют желать лучшего. Я часто замечаю, что сообщения об ошибках выглядят так, будто программа адресует их самой себе, вместо того чтобы подсказать пользователю, что он может предпринять. А иногда бывает и того хуже.

Насколько подробными должны быть сообщения об ошибках?

Питер: Они должны быть как можно более полезными, хотя это не ответ на ваш вопрос. Одни ошибки в программе бывает значительно труднее понять, чем другие, хотя эвристически можно разобраться. Например, в Си-подобных языках ошибки при вводе разделителей и фигурных скобок часто сбивают с толку компиляторы, и им трудно составить правильное сообщение, но человек привыкает узнавать сообще-

ния, которые выдает компилятор при пропуске точки с запятой между определением класса и следующей функцией. Вы уже знаете, что это дебильное сообщение об ошибке не имеет никакого отношения к тому, что произошло на самом деле. Компилятор слишком углубился в следующую функцию, прежде чем заметил ошибку. А если вы пропустите закрывающую фигурную скобку, то получите аналогичное непостижимое сообщение об ошибке – привыкаешь узнавать его по внешнему виду. Можно было бы точнее квалифицировать ошибки, но на это нужно положить много усилий, в оправданности которых нет уверенности.

Можно поставить вопрос иначе: что вы предпочтете – дурацкие сообщения об ошибках, по которым можно догадаться, что не понравилось программе, или полезные сообщения об ошибках вроде советов, которые Microsoft помещала в Word и которые были почти бесполезными и всегда ошибочными?

По-моему, если хочешь сделать красивые сообщения об ошибках, то сначала нужно постараться, чтобы они, как правило, соответствовали истине, потому что мы привыкли получать плохонькие сообщения об ошибках, по которым можно догадаться, что происходит в действительности.

В ожидании прорыва

Что бы вы сейчас изменили в AWK, чтобы улучшить поддержку больших программ?

Питер: С учетом всего произошедшего за это время, вопрос можно поставить так: что нам пришлось бы придумывать – Perl или что-то другое? Думаю, по складу ума мы не годились для создания Perl в полном объеме, но если принять во внимание характер работы над AWK, то, решив, что он должен быть применим для создания больших программ, мы, возможно, пришли бы к чему-то подобному.

Думаю, другая причина того, что в какой-то момент мы прекратили дальнейшую работу, заключается в том, что момент был выбран удачно. Не помню, рассказывал ли я эту историю (кажется, это произошло через считаные месяцы после внутреннего релиза AWK, а может, и через пару лет): мне позвонил кто-то из вычислительного центра и сообщил, что у него возникли некоторые проблемы с AWK, и я пошел туда посмотреть на его программу. Я ведь уже говорил, что мы представляли себе AWK как средство для создания однострочных программ, мелких поделок? Так вот, тот человек написал на AWK ассемблер для какого-то таинственного устройства, и его код занимал 55 страниц. Мы были поражены. Конечно, нет ничего удивительного, что у него это получи-

лось: бывает, пишут еще более длинные программы на менее структурированных языках, но для нас это было большой неожиданностью.

Брайан сказал, что практически всегда, когда появляется какой-то малый язык, люди начинают пользоваться им и требовать циклы и прочее, поэтому приходится прекращать разработку, иначе язык...

Питер: ...становится все больше и больше, и нужно решать, готовы ли вы идти по этому пути или нет.

Если хотите создать универсальный язык, то и начинайте работу с этой целью, а не пишите малый язык, который придется расширять в случае успеха.

Питер: Я тоже так считаю. И это другой вопрос, с ним связана еще одна история, которую я, возможно, рассказывал. Обычно люди пишут программы и, скажем, языки, парсеры или компиляторы, предполагая, что кто-то будет вводить данные с клавиатуры, но иногда обнаруживается, что входные данные генерируются специально созданными для этого программами. Думаю, обычно впервые с этим сталкиваются при компиляции, потому что, скажем, никто не встречал оператор переключателя с 80 000 ветвей для Си-образного языка. Предполагалось, что никто никогда не сможет напечатать оператор `switch` с 80 000 ветвей. И генератор кода никогда не сталкивался с оператором `switch`, имеющим столько ветвей. Такие вещи случаются на любом уровне даже для универсальных языков.

Что вы думаете о расширяемых языках, которые пользователи могут модифицировать?

Питер: Все зависит от того, что конкретно под этим понимается. В принципе, хорошо, конечно, хотя возникает масса ограничений, если только вы не имеете в виду что-то вроде Лиспа, когда можно расширять язык с помощью макросов. Есть много причин, по которым бывает желательно добавить что-то в язык – для большей выразительности или для подключения библиотек, написанных на других языках и делающих какие-то сложные вещи.

Есть еще одна проблема, с которой, к счастью, нам не пришлось столкнуться в AWK, – это насколько сложно включение в ваш язык подпрограмм или пакетов на других языках. У нее могут быть разные решения.

Существует деление на математиков и не математиков. Есть ли разница между математикой и разработкой ПО? Си победил, Scheme – нет. Си победил, Лисп – нет. Снова «хуже значит лучше».

Питер: Думаю, есть некоторая разница между языками, проектируемыми математиками и не математиками, но различие между Scheme

и Си скорее связано с попыткой построить язык на простом основании или ее отсутствием. Здесь нет определенных границ: у нас нет точных знаний.

Очевидно, нет общего согласия по поводу причин, благодаря которым тот или иной язык достигает успеха, но способствовать ему может многое. У каждого языка есть свои излюбленные штучки, и невозможно удовлетворить всех. У первоначального Лиспа в соответствии с требованиями пуристов была очень простая модель, благодаря чему обеспечивалась удивительная мощь. Спрашивается, зачем понадобилось усложнять язык? Чего он не позволял, что потребовались усложнения? В последующих версиях Лиспа, а затем в Scheme и, наконец, в Common Lisp, есть излишние усложнения.

Вероятно, тут два обстоятельства. Наверное, это не совсем точно, потому что я не обдумывал тщательно этот вопрос, и скажу так, как мне это сейчас представляется. Первое связано с удобствами для программистов, а второе – с эффективностью программ. Мы еще доберемся до других языков, но что касается Лиспа, то я думаю, что языки подобного типа значительно прояснили некоторые из этих проблем.

Большое различие между первоначальным Лиспом и Common Lisp состоит, как мне кажется, в добавлении типов данных, хеш-таблиц и прочего. По существу, они введены для большей эффективности программ. С другой стороны – я не претендую на историческую точность, поскольку не очень хорошо знаком с Лиспом, – в какой-то момент разработчики Лиспа стали вводить макросы. В каком-то отношении макросы – вещь совершенно естественная. Просто для них существует другая среда вычисления, но – это как раз вопрос удобства для программиста – они не делают ничего такого, что программист не мог бы написать сам.

Они служат фактором эффективности.

Питер: По крайней мере, вы ждете этого от них. Но они служат и фактором запутанности, если применять их активно: ваш код фактически никто больше не сможет прочесть. Из-за них гораздо труднее объяснить, что делает ваш язык, даже неформально. Возникает множество любопытных патологических случаев. Даже в такой относительно незамутненной среде видны противоречия между тем, что можно назвать математической чистотой, и необходимостью решить задачу.

Всем языкам Лисп и всем прочим языкам, во многих из которых есть весьма формальные семантические определения, присуща одна и та же проблема. Есть по крайней мере два этапа жизненного цикла программы, в которых участвуют люди, плюс тут участвует компьютер. Чтобы сделать программу пригодной для выполнения компьютером, нужно совершенно точно определить, какой смысл имеет написанное челове-

ком. При этом написание программ должно быть посильным трудом. После того как программа некоторое время побудет в эксплуатации, вполне возможно, потребуется ее сопровождение, и нужно, чтобы те, кто его станет осуществлять, могли разобраться в коде и модифицировать его.

По моему «скромному, но верному» мнению, некоторые языки обладают рядом характеристик, облегчающих написание кода, но очень осложняющих его сопровождение. Особенно отличаются этим практически все объектно-ориентированные языки. Эти практичные языки, точная семантика которых понятна только компилятору и его автору, предоставляют большие удобства при написании нового кода. Хорошо, если язык позволяет выразить ваши намерения таким образом, чтобы они были понятны тому, в чьем распоряжении нет ничего, кроме этого кода. Я затрудняюсь в поиске каких-нибудь примеров. Хотя, конечно, я перепробовал не все языки. Но те, которые мне знакомы, не сильно отличаются в этом отношении в лучшую или худшую сторону.

Это два разных измерения.

Питер: Верно. Но ведь мы писали код в те золотые времена, когда по земле бродили великаны. Были и пигмеи, но у них были большие ящики. Никто не рассчитывал, что его код проживет 30 лет. Если бы вы сказали, что к тому времени UNIX будет еще жива или Фортран будет еще жив, каждый ответил бы: да, и мы перепишем их заново. Таков был наш образ жизни. Ты это написал, ты это и будешь переписывать. С каждым разом все лучше, и каждый раз добавляя несовместимости, пока не падешь жертвой эффекта второй системы, когда продукт становится очень слабо совместимым и значительно хуже. Теоретически нам была понятна мысль, что есть только два вида программных проектов: либо провалившиеся, либо ставшие кошмаром сопровождения.

Мы не понимали, что невозможно год за годом продолжать писать программы и при этом переписывать их заново. Объем программного обеспечения растет, и либо вы будете заниматься только переписыванием старых программ, либо оставите их такими как есть. За всем не угнаться. Проблема сопровождения выглядит все более грозно, хотя говорю это потому, что у меня много времени уходит на нее в Google, где она кажется мне довольно серьезной.

Оставив в стороне эту нерешенную проблему и возвращаясь к моим замечаниям по поводу разницы между языками математического и не математического типа: существуют также языки, разработанные математиками, либо бывшими математиками, либо людьми с математическим складом ума, и – людьми другого типа.

От первых следует ожидать почти полной спецификации, пусть даже неформальной. Нужно действительно зафиксировать, что должно происходить во всех мыслимых ситуациях. Нужно реально записать лексический состав, а не заставлять догадываться о нем. Все остальное вам, возможно, не удастся записать.

Саймон Пейтон-Джонс говорил, что им удалось специфицировать примерно 85% первой версии Haskell, а потом они решили, что на остальное не стоит тратить время.

Питер: Иногда старательность бывает излишней. То, что не самым лучшим образом вышло в Си с `short int` и `long`. Было два варианта.

Есть еще знаковость.

Питер: Давайте не будем даже трогать наличие знака числа или `const`, если на то пошло. Можно было сказать: «У нас будут `int8`, `int16`, `int24`, `int36`, `int64`, что бы они все ни значили, и либо язык будет делать все это в точности, либо мы постараемся, но на этом заканчиваем».

Все это в прошлом. Не уверен, что смог бы сделать это хотя бы так же, начни я все сначала. Или можно сказать: «Слушайте, у нас есть `short int` и `long`, мы объясним, что они значат, и отвалите. Со всем остальным справляйтесь как-нибудь сами». Авторов компиляторов, стремящихся выжать максимум эффективности для своих пользователей, это не застало бы врасплох. Взгляните на GCC, где вам милостиво предоставлено множество типов. И далее еще появляются беззнаковые типы, указатели и так далее, чего уже не охватить взглядом.

Строки – тоже интересный пример. Дойдя до этого, сознаешь, что если разрешить использовать в них любые символы, а не делать, скажем, строки UTF-8 или строки ASCII, то их нельзя вывести на печать. Хорошо иметь возможность заявить, что все поддерживаемые в твоем языке и в твоей программе структуры являются двоичными, исключая те, для которых явно задан печатный формат, но с этим связана большая головная боль.

Необходимы реальные теоретические прорывы. Их давно нет, и это меня угнетает.

Теперь и меня угнетает.

Питер: Сожалею. Поразительно, но все эти вещи каким-то образом работают, и на них действительно можно полагаться. Гарантий никто не дает, но фактически полагаться можно. Понимаете, автомобиль уже давно не может ездить без помощи компьютера, и в нем масса программного кода. И – плохо ли, хорошо ли – он обычно работает. Я знаю, что его работа не гарантирована и что иногда этим компьютерам требуется перезагрузка во время движения по шоссе, и все такое прочее, но

в принципе на его работу можно положиться. Я сетую на неэффективность, а не на фундаментальные пороки, но она раздражает.

Можно ли решить эти проблемы без прорыва?

Питер: Не просто «нет!», а «разумеется, нет!», но хочу высказать некоторые наблюдения. С одной стороны, машины, на которых мы пишем программы, располагают колоссальной вычислительной мощью. По большей части она не используется, потому что машина работает вхолостую. Какая-то удивительно большая часть ее расходуется на обеспечение интерфейса пользователя. А если вы, скажем, компилируете код Си, то в огромной мере вычислительная мощь уходит на то, чтобы считывать данные в память, а затем записывать различные вспомогательные файлы, которые затем будут снова считываться в память.

Далее. Если язык обладает некоторой структурной целостностью, позволяет создавать псевдонимы и многое прочее, то от компилятора хотелось бы получить гораздо больше. У программиста должен быть какой-то способ описания его намерений, с которым мы пока не вполне разобрались. На самом деле, все осложнилось. Было бы смело сказать, что потоки и объектная ориентированность независимы. Гораздо правильнее указать на их ужасное переплетение. Со всем этим мы живем. И нужно разгребать все то, что мешает пониманию того, как работают программы. С наступлением эпохи множественных процессоров это становится еще труднее.

Может быть, где-то есть светлые головы, занятые очень интересными проблемами. Может быть, что-то произойдет. У нас нет другого средства, кроме как просить у компьютеров помощи в том, чтобы сделать наши программы надежнее и понятнее, и пользоваться более совершенными языками, когда мы их придумаем. Трудно разглядеть – это как гигантский моток пряжи, в котором нужно найти правильную петлю и потянуть ее, чтобы найти конец. Не знаю, мой оптимизм небезграничен.

С другой стороны, все не так плохо. Просто все это несколько досадно.

Что вы считаете успехом в своей работе?

Питер: Когда мы занимались AWK и другими подобными вещами, нам казалось (возможно, мы заблуждались), что если есть идея, то надо немного над ней поработать, и если идея верна и реализована хорошо, ты внесешь большой вклад в компьютерное дело. Добиться соответствия такому стандарту сейчас, как мне кажется, чрезвычайно тяжело. В то время было легко оказать весьма существенное влияние в области компьютерных вычислений, которая приобрела потом важное значение, и я думаю, что подобное уже невозможно.

Мне кажется, что небольших влиятельных групп не так много. Это не значит, что невозможен успех – в смысле количества тех, кто пользуется вашими программами и высоко их ценит, – но оказать такое влияние, какое было возможно тогда, теперь, как мне кажется, нельзя. Теперь не увидишь такого, как в ярчайшем примере UNIX, когда относительно небольшая группа создала то, что существенно изменило мир. Может быть, я что-то упустил в последние 5–10 лет, и ваши читатели меня поправят, но я сомневаюсь. Думаю, сейчас группы должны быть больше, и сделать это гораздо труднее.

Полагаю, на ваш вопрос можно ответить так: нам повезло, и наш относительно небольшой труд имел существенное влияние. Какое это было время, какая замечательная мера успеха! Сейчас же, мне кажется, людям с гораздо большими талантами, чем были тогда у нас, трудно рассчитывать на такой успех. Конечно, во многих отношениях это хорошо. Это значит, что мы значительно продвинулись вперед, но при этом отдельному человеку приходится довольствоваться меньшим.

Программирование по примерам

Вы отметили, что долголетие AWK основано на программировании по примерам.

Питер: Это было намеренным конструктивным решением. С ним много связано – как плохого, так и хорошего. В AWK есть ряд «интересных» – назовем их так – синтаксических решений, из числа которых я бы лишь пару признал действительно ошибочными. Основной идеей было сделать его похожим на Си, чтобы избавиться от необходимости объяснений с теми, с кем вместе мы работали.

Тогда какой возникает вопрос? Мы считали, что раз все программы AWK будут размером в одну – максимум несколько – строк, то программирование на AWK будет заключаться в том, чтобы искать и находить примеры, в которых делается нечто похожее на то, что вам нужно. Просто «подгоните» найденное. Если требуется нечто более сложное, нужно последовательно делать то же самое, и все получится. В то время, когда мы создавали AWK, в Xerox PARC был проект, названия которого я, к сожалению, не помню, делавший примерно то же, что и AWK. Он должен был обрабатывать файлы. В Xerox PARC не рассматривали файлы как состоящие из строк и так далее, но сходство было близким. Система была рассчитана на секретарш. На странице было две колонки. В левой вы писали программу, а в правой – рабочий пример. Компилятор проверял, что ваша программа делает то, что описано в примере.

Толково.

Питер: Да, толково, а дальше они работали над тем, чтобы сделать синтаксис доступным для секретарш. Разумеется, ничего из этого не вышло. Успеха, в отличие от AWK, эта система не имела, и по многим причинам. UNIX стала популярной, а та система, для которой разрабатывался этот проект, – нет, и так далее, и так далее. Наш вариант основывался на идее, что можно найти программу, которая делает примерно то, что вам нужно. AWK всегда был явно рассчитан на программистов.

Конечно, не это стало причиной успеха, но относительная простота и наличие примеров, возможность взять «книгу AWK» и найти в ней образцы сыграли, как мне кажется, немалую роль.

При копировании/вставке и доработке программы семантика языка изучается в лучшем случае попутно.

Питер: Но идея книги AWK была в том, чтобы поместить между неформальным введением и всеми примерами достаточно полное описание того, что представляет собой язык и каковы его возможности. Думаю, оно там есть, и это помогло.

И люди это читают?

Питер: Кто-то читает, кто-то – нет. Я скажу немного иначе, хорошо? Читаете вы описание или нет, но в AWK можно работать по образцу, верно? Это подтверждено на практике. В Аде это можно?

Никогда не пробовал.

Питер: Мне представляется, что было бы очень нелегко создавать программы на Аде с помощью одних лишь примеров. Вполне вероятно, что программы C++ было бы трудно создавать по примерам. На каком-то достаточно простом уровне можно было бы найти к этому подходы.

Но на Си и не пишут однострочные программы.

Питер: Конечно, в этом разница. Очень трудно писать однострочники, помимо проблем с экраном. Маленькие примеры по своей природе оказываются большими.

Как и круг решаемых задач.

Питер: Верно. Это универсальные языки, в отличие от AWK. Одной из первых программ на AWK был ассемблер для какого-то дополнительного процессора. Я был в ужасе. Тот, кто его писал, толком не объяснил, почему выбрал этот путь, но было ясно, что гораздо легче начать с интерпретируемого языка, а мощи оболочки недостаточно.

Я был бы рад, если бы программирование стало доступнее обычным людям, но мне также нравится, когда программы становятся более надежными и легче объединяются в более крупные метапрограммы. Это нелегко.

Питер: Объединение может отчасти быть облегчено конструкцией языка и идиомами. С надежностью тяжело. И с четким проектированием тоже.

Что вы считаете четким проектом?

Питер: Это своего рода метапроблема. Раньше я был значительно более уверен в суждениях. Смотришь и пытаешься написать небольшие примеры. Слушаешь, что говорят об этом другие. Но практически всегда четкие примеры, которые принято помещать в учебники, оказываются совершенно нереалистичными. Может, и встречаются классы, похожие на те, которые можно найти в начальных курсах по объектно-ориентированному программированию, но я в это не верю. У всех у них много-много членов, которые могут реагировать на множество сообщений. Полезный объект, который можно включить в программу, так, чтобы понимать, в каком состоянии находится ваша программа, содержит много кода. Думаю, приходится мириться с определенным объемом сложностей, если награда за это кажется достаточной.

Или воспринимается таковой.

Питер: Верно. Это все, что вы получите. Такова природа программного инжиниринга. В нем нет ничего количественного. Восприятие и реальность совпадают, потому что мы не можем измерить реальность или, во всяком случае, показали, что не хотим.

Мы не можем измерить даже продуктивность, из-за чего трудно сказать, что лучше, а что хуже.

Питер: Да, но думаю, что не это главное. В инженерии, имеющей дело с реальностью, результаты измеримы. Вы строите мост. Во сколько он обошелся? Какие трудности были при строительстве? Будет ли он надежен?

Для программы можно измерить, насколько трудно было ее написать, сколько денег было потрачено на ее создание, но все остальное – полная тайна. Хорошо ли она делает то, для чего предназначена? Как можно это узнать? Как описать, что она должна делать?

Для программного обеспечения не существует материаловедения.

Питер: У меня такое предчувствие, что когда разработчики аппаратуры выйдут, наконец, из моды, программному обеспечению может быть уделено большее внимание инженерной науки. Все меняется очень быстро. Аналогия спорная, но если бы свойства бетона и стали менятья каждый год на 10%, строительные технологии выглядели бы сейчас совсем иначе.

Это гипотеза, потому что нет причин, по которым они должны бы выглядеть иначе. Просто были бы модели 2007, 2008, 2009. В программи-

ровании никогда не было никаких моделей, поэтому у нас вряд ли получится такое.

Мы говорим о программном обеспечении. Здесь нет атомов. Нет физических свойств.

Питер: Верно. Это не совсем похоже на математику. Программы не живут исключительно в голове человека. Они почти целиком созданы человеком, и ограничения на них представляют собой некую комбинацию математики, занимающейся вычислимостью и сложностью алгоритмов, а также того, что дает нам аппаратное обеспечение. Последнее достаточно быстро развивается.

Вы также отметили реальное отличие программирования от доказательства теорем. Можно доказать теорему и что-то узнать, но если вы написали программу для компьютера, то у вас внезапно возникает возможность делать то, что до этого было невозможно.

Питер: Я по-прежнему считаю, что это, в основном, верно. Конечно, в современных вариантах теоремы бывают связаны с алгоритмами, что не удивительно, поскольку компьютерные вычисления настолько полезны, что границы несколько стерлись.

До наступления компьютерного века вычисления были трудным делом. Расчетов требовалось много. Научная литература позволяет иногда заглянуть в рабочие тетради тех, кто занимался расчетами, и увидеть, какие чудеса они творили – искусные вычисления, и они подсказывают ответ. Не думаю, что отличие уж слишком разительно. В свое время это поразило меня, потому что я начинал как математик.

Произойдет ли в компьютерных науках революция, когда мы станем относиться к компонентам как к теоремам?

Питер: Только когда мы научимся их описывать. Те типы описаний, которые используются наиболее широко, являются чисто функциональными. Они сообщают, как входные данные преобразуются в выходные. Они ничего или почти ничего не сообщают о том, сколько это потребует времени. Почти ничего не говорится о том, сколько потребуется памяти. Не вполне понятно, какая среда нужна для выполнения. Теоремы, надо сказать, имеют человеческую природу, в отличие от машинных теорем, но в них есть гипотеза и выводы. Это доступно лишь умеренной интерпретации, тогда как в программном обеспечении нужно интерпретировать многое.

Есть множество неприятных примеров. В приведенном мной перечне отсутствуют даже такие вещи, как действительно тщательная спецификация входных данных. Известны программы, которые сбоили при переводе с 16 на 32 бита, потому что в каком-то месте были по существу 16-разрядными, а об этом не было известно. Трудно описать все, что

с этим связано. Лучшим примером служат программы, корректность которых доказана, но в которых есть ошибки. К несчастью, в какой-то своей части они неверно моделировали реальность. Никто этого не заметил.

У сторонников типов та же проблема. Они хотят гарантировать введение типов, что делает систему довольно слабой. Другая крайность в этом многомерном пространстве – шаблоны C++, с помощью которых можно вычислить все что угодно на этапе компиляции с достаточной скоростью, но поскольку вычислять можно все что угодно, это будет медленно.

Интересное замечание о том, что нас тревожит в компьютерных науках. Компьютеры не становятся быстрее. Они становятся шире.

Питер: Совсем неплохо жить, когда качество растет экспоненциально.

Я понял это недавно, когда рост объема данных с 1970-х затмил рост скоростей процессоров. Если взять SQL, который был тогда спроектирован, то он по-прежнему справляется с этим взрывным характером роста объема данных. Другим языкам это не так хорошо удается.

Питер: Я скажу, что с того времени скорость ЦП выросла примерно в тысячу раз. С данными это несравнимо. По опыту компьютерных вычислений можно прикинуть, что скорость растет как 10^n в зависимости от времени. Не будет большой ошибкой отнести $10^{n/2}$ на счет аппаратной части и $10^{n/2}$ на счет алгоритмов. Думаю, примерно так.

На создание оптимизаторов запросов и совершенствование проектирования баз данных потрачено много труда, что позволило работать с терабайтами данных.

Как рост аппаратных ресурсов влияет на мышление программистов?

Питер: Программисты слишком разные, чтобы делать такие обобщения. Нужно помнить об ограничениях. Например, скорость света – это скорость света, и ее не изменишь. Есть вещи, которые работают локально, но удаленный доступ к ним – это катастрофа. Разного рода уровни абстракции и библиотеки позволяют быстро доводить программы до рабочего состояния, но оказывают губительное влияние на скорость и надежность.

Как вы поступаете – сначала выбираете нужные алгоритмы, а затем наращиваете их скорость, или с самого начала ставите целью скорость?

Питер: Если вы хоть немного разбираетесь в задаче, то строите алгоритм так, чтобы он обеспечивал достаточную эффективность там, где вы считаете это необходимым, а затем, если нужно, осуществляете его доводку. Обычно важнее, чтобы алгоритм работал, а после существен-

ной доводки труднее перестроить реализацию. Но иногда обнаруживается, что нечто более громоздко или используется чаще, чем предполагалось, – скажем, квадратичные алгоритмы оказываются неприемлемыми или даже слишком много времени уходит на копирование и сортировку. Многие программы удовлетворительно работают без особой доработки. Современные компьютеры очень быстрые, а наблюдаемые задержки часто связаны с работой сетей и устройств ввода/вывода.

Как вы ищете проблемы в программном обеспечении?

Питер: Все дело в том, чтобы искать проблемы, которые вы можете решить. В программном обеспечении есть множество проблем, которые слишком сложны, а для других проблем достаточно увеличить объем задачи на пару порядков, и прежние решения могут оказаться малоэффективными.

Что вы делаете после того, как добились, чтобы программа в целом работала?

Питер: Если она предназначена для моего личного пользования, то я останавливаюсь на этом в надежде, что сохраню в памяти достаточно контекста для ее обновления. Если это профессиональный продукт, совершенно необходимо создать документацию, постараться защитить от различных неблагоприятных ситуаций и снабдить код комментариями, чтобы облегчить его сопровождение. Последнее сложно. Код с хорошими комментариями встречается редко.

Согласно философии UNIX, «если не умеешь делать что-то хорошо, не делай это». Можно ли распространить такой художественный подход за пределы UNIX?

Питер: Это проблема не только программного обеспечения, она значительно шире. По большей части нам приходится заниматься тем, что мы не очень хорошо умеем делать. С UNIX нам повезло в этом отношении, кроме того, родоначальники UNIX сумели сделать многое очень хорошо.

Было бы интересно проверить трезвую гипотезу, что когда бизнес не следует этому правилу, он не слишком процветает, но я не уверен, что найдутся убедительные свидетельства в ту или иную сторону. Естественно напрашивается сравнение (явно поверхностное) между Microsoft и Apple, но что должно служить критерием – восторги критиков или сумма прибыли?

7

Lua

Lua – очень маленький, автономный динамический язык, который Роберто Иерусалимский, Луис Энрике де Фигейреду и Вальдемар Целес создали в 1993 году. Небольшой набор мощных возможностей Lua и простой в применении C API облегчают встраивание языка и его расширение для конкретных предметных областей. Lua играет заметную роль в коммерческом программном обеспечении, где он применяется для скриптов и разработки интерфейса пользователя в таких играх, как World of Warcraft Blizzard и Crysis Crytek GmbH, наряду с Photoshop Lightroom Adobe. Предшественниками Lua выступают Лисп, Scheme и, возможно, AWK. По конструкции он схож с JavaScript, Icon и Tcl.

Мощь скриптов

Как бы вы определили Lua?

Луис Энрике де Фигейреду: Встраиваемый, облегченный, быстрый, мощный язык сценариев.

Роберто Иерусалимский: К сожалению, все чаще термин «язык сценариев» (scripting language) употребляют в качестве синонима для «динамического языка». Сейчас даже Erlang и Scheme стали называть языками сценариев. Досадно, потому что тем самым теряется точность при описании определенного класса динамических языков. Lua – язык сценариев в первоначальном смысле этого выражения. Язык для управления другими компонентами, обычно написанными на другом языке.

О чем нужно помнить, разрабатывая программное обеспечение с помощью Lua?

Луис: О том, что Lua, возможно, предлагает свой способ работы. Не следует пытаться эмулировать подходы, применяемые в других языках. Нужно пользоваться теми функциями, которые предлагает ваш язык, – это справедливо для любого языка. Что касается Lua, то к таким функциям относятся таблицы, применяемые повсеместно, и метаметоды для элегантных решений. А также сопрограммы.

Кому подходит Lua?

Роберто: Думаю, большинство приложений, не имеющих средств сценариев, могли бы извлечь пользу из применения Lua.

Луис: Проблема в том, что большинство проектировщиков слишком поздно замечают эту потребность – когда большая часть кода уже написана, скажем, на Си или C++. Разработчикам приложений нужно с самого начала рассматривать возможность создания сценариев. Тем самым обеспечивается гораздо большая гибкость. Удается также составить лучшее представление об эффективности, потому что приходится задуматься, в каких местах приложению действительно *нужна* максимальная производительность, а где о ней можно не думать, переложив задачу на более легкий и короткий цикл разработки сценариев.

Что предлагает Lua программистам с точки зрения безопасности?

Роберто: Ядро интерпретатора Lua построено в виде «отдельного приложения» (freestanding application). Термин взят из ISO Си, где означает, что программа не пользуется практически ничем из окружающей среды (никаких `stdio`, `malloc` и так далее). Все эти функции предоставляются внешними библиотеками. В такой архитектуре очень легко создавать программы с ограниченным доступом к внешним ресурсам. На-

пример, можно создавать песочницы внутри самого Lua, удаляя из его окружения то, что покажется опасным (например, `fileopen`).

Луис: Кроме того, Lua предоставляет определяемые пользователем ловушки для отладки, с помощью которых можно следить за выполнением программы Lua и, например, прерывать ее выполнение, если она работает слишком долго или использует слишком много памяти.

Каковы границы применимости Lua?

Роберто: Думаю, эти границы такие же, как для любого динамического языка. Во-первых, даже самая передовая технология JIT (а в Lua она одна из лучших среди динамических языков) не позволяет достичь такой же производительности, как у хорошего статического языка. Во-вторых, сложные программы могут иногда существенно выиграть благодаря статическому анализу (главным образом, статическим типам).

Почему вы решили воспользоваться сборщиком мусора?

Роберто: В Lua всегда был сборщик мусора, с самого начала. Я считаю, что для интерпретируемого языка сборщик мусора может быть гораздо компактнее и надежнее, чем подсчет ссылок, не говоря уже о том, что в результате не остается мусора. С учетом того, что в интерпретируемом языке обычно уже есть данные с самоописанием (значения с тегами и тому подобное), простой сборщик с пометкой и удалением может оказаться действительно простым и почти не будет влиять на другие части интерпретатора.

А в нетипизированном языке подсчет ссылок может быть очень затруднен. Если нет статических типов, одно-единственное присваивание может изменять счетчики ссылок, поэтому требуется динамически проверять их для прежнего и нового значения переменной. Последующие опыты с подсчетом ссылок в Lua нисколько не улучшили его эффективность.

Вас удовлетворяет, как Lua работает с числами?

Роберто: Думаю, числа в компьютерах всегда будут служить источником неожиданностей (как, впрочем, и вне компьютеров!). Я считаю, что использование в Lua чисел двойной точности в качестве единственного числового типа – разумный компромисс. Мы рассматривали много других вариантов, но обычно они оказывались слишком сложными, слишком медленными или слишком требовательными к памяти. Для встраиваемых систем даже использование чисел двойной точности не является рациональным выбором, поэтому мы можем скомпилировать интерпретатор с альтернативным числовым типом, например `long`.

Почему вы выбрали в Lua таблицы в качестве универсального конструктора данных?

Роберто: Меня вдохновил VDM (формальный метод, применяемый преимущественно в спецификациях программного обеспечения), которым я занимался в то время, когда мы стали работать над Lua. VDM предлагает три вида коллекций: множества, последовательности и отображения. Но множества и последовательности легко описать как отображения, поэтому у меня и возникла мысль использовать отображение в качестве универсального конструктора. У Луиса были свои соображения по этому поводу.

Луис: Да, мне очень нравился AWK, особенно его ассоциативные массивы.

Насколько ценны для программистов функции первого класса в Lua?

Роберто: Под разными именами – от «подпрограмм» до «методов» – функции уже более 50 лет составляют основу языков программирования, поэтому хорошая поддержка функций является ценной стороной любого языка. Lua позволяет программистам использовать ряд мощных технологий из области функционального программирования, например представление данных в виде функций. Например, фигуру можно представить как функцию, которая по входным значениям x и y определяет, находится ли эта точка внутри фигуры. Такое представление позволяет выполнять тривиальные операции объединения и пересечения.

В Lua функции иногда используются нестандартным способом, и то, что они относятся к первому классу, упрощает такое применение. Например, каждый блок (фрагмент кода, передаваемый интерпретатору) компилируется как тело функции, поэтому в Lua определение любой обычной функции всегда вложено в другую функцию. Это означает, что даже тривиальным программам Lua необходимы функции первого класса.

Зачем вы реализовали замыкания?

Роберто: Замыкания (closures) относятся к конструкциям, к которым мы всегда стремились в Lua, – простым, универсальным и мощным. В Lua функции были величинами первого класса, начиная с самой первой его версии, и они показали себя действительно полезными – даже для «обычных» программистов, не имевших опыта функционального программирования, но без замыканий их применение было несколько ограничено. Кстати, термин *замыкание* (closure) относится к способу реализации, а не к самой характеристике, которую следует называть «функции первого класса с лексической областью видимости», но «замыкание», конечно, короче. :)

Как вы собираетесь решать проблемы параллелизма?

Роберто: Мы не верим в многопоточность, в смысле общей памяти с вытесняющей многозадачностью. В своей статье в HOPL[1] мы написали: «Мы полагаем, что невозможно писать корректные программы на языке, в котором $a = a + 1$ не детерминировано». Избежать этой проблемы можно, устранив вытесняющую многозадачность или общую память, и в Lua поддерживаются оба эти способа.

Что касается сопрограмм, то у нас есть общая память без вытеснения, но в многоядерных платформах это бесполезно. Однако несколько «процессов» могут весьма эффективно исследовать такие машины. Под «процессом» я имею в виду поток Си с собственным состоянием Lua, поэтому на уровне Lua совместный доступ к памяти не происходит. Во втором издании «Programming in Lua» (Lua.org) я уже показывал прототип такой реализации, а недавно появились библиотеки, которые поддерживают такой подход (например, Lua Lanes и luaproc).

Вы не поддерживаете параллелизм, но реализовали интересное решение для многозадачности, а именно асимметричные сопрограммы. Как они работают?

Роберто: У меня есть некоторый опыт работы с Модулой-2 (моя жена написала законченный интерпретатор M-кода во время своей дипломной работы), и мне всегда нравилась идея использовать сопрограммы в качестве основы кооперативной многозадачности и для других управляющих структур. Однако симметричные сопрограммы в том виде, как они представлены в Модуле-2, не годятся для Lua.

Луис: В нашей статье в HOPL подробно обсуждаются все эти проектные решения.

Роберто: В итоге мы пришли к своей асимметричной модели. Основная идея проста. Мы создаем сопрограмму явным обращением к функции `coroutine.create`, передавая функцию, которую нужно выполнить как тело сопрограммы. При возобновлении сопрограммы ее тело выполняется до конца или до возврата значения; возврат значения сопрограммой происходит только при явном вызове функции `yield`. В дальнейшем можно возобновить ее выполнение, и оно продолжится с того места, где прервалось.

Общая идея напоминает генераторы в Python, но с важным отличием: сопрограммы Lua могут возвращать значения из вложенных вызовов, тогда как генератор Python может возвращать значение только из своей главной функции. Для реализации это означает, что у сопрограммы должен быть свой независимый стек, как у потока. Просто удивитель-

[1] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, «The evolution of Lua», Proceedings of ACM HOPL III (2007).

но, насколько такие сопрограммы «со стеком» мощнее «плоских» генераторов. Например, можно реализовывать поверх них однократные продолжения.

Опыт

Что вы считаете успехом в своей работе?

Луис: Успех языка определяется количеством работающих на нем программистов и успешностью созданных с его помощью приложений. Мы не знаем точно, сколько народу программирует на Lua, но есть много успешных приложений, использующих Lua, в том числе несколько очень успешных игр. Кроме того, диапазон использующих Lua приложений, простирающийся от обработки графики настольными системами до встроенного управления роботами, показывает, что область применения для Lua найдется. Наконец, Lua – единственный язык, созданный в развивающейся стране и получивший такое глобальное признание. Это единственный из таких языков, попавший в ACM HOPL.

Роберто: Это нелегко. Я работаю в нескольких направлениях, и в каждом из них ощущаю успех по-своему. В целом я назвал бы общим для всех определений успеха «достаточную известность». Всегда очень приятно, когда при знакомстве или встрече оказывается, что тебя знают.

Есть ли у вас какие-нибудь сожаления, связанные с языком?

Луис: У меня нет никаких сожалений. Конечно, с нашими теперешними знаниями некоторые вещи можно было бы сделать раньше!

Роберто: Пожалуй, у меня нет конкретных сожалений, но при разработке языка приходится принимать несколько жестких решений. Самые трудные из них для меня те, что связаны с простотой применения. Одной из целей Lua была легкость освоения непрофессиональными программистами. Сам я не отношусь к этой категории, поэтому некоторые принятые решения не идеальны для меня лично как пользователя. Типичный пример – синтаксис Lua: во многих случаях этот язык выигрывает от своей многословности, но я предпочел бы более компактную нотацию.

Были ли допущены ошибки при проектировании или реализации?

Луис: Мне кажется, что при проектировании и реализации Lua мы не допустили крупных ошибок. Мы просто учились развивать язык, что далеко не сводится к определению синтаксиса и семантики и последующей их реализации. Есть также важные социальные проблемы, такие как создание и поддержка сообщества, включая руководства, книги, веб-сайты, списки рассылки, чаты и так далее. Несомненно, мы позна-

ли значение поддержки сообщества и большого труда, который она требует, – наряду с проектированием и кодированием.

Роберто: К счастью, крупных ошибок мы не сделали, а мелких, пожалуй, было много. Но нам удавалось исправлять их по ходу развития Lua. Конечно, некоторые пользователи были недовольны несовместимостью версий, но сейчас Lua вполне стабилен.

Что вы посоветуете тем, кто хочет научиться лучше программировать?

Луис: Не бойтесь начать все с самого начала, хотя, конечно, это проще сказать, чем сделать. Невозможно переоценить внимание к мелочам. Не добавляйте сразу всю функциональность, которая только может понадобится: не исключено, что, добавив ее сейчас, вы лишитесь возможности добавить гораздо лучшую функцию потом, когда она действительно потребуется. Наконец, всегда стремитесь найти простейшее решение. «Как можно проще, но не проще того», – как говорил Эйнштейн.

Роберто: Изучайте новые языки программирования. Но только по хорошим книгам! Haskell – это язык, который должны знать все программисты. Изучайте компьютерные науки: новые алгоритмы, новые формализмы (лямбда-исчисление, если вы еще не знакомы с ним, пи-исчисление, теорию последовательных взаимодействующих процессов – CSP и так далее). Всегда старайтесь улучшить свой код.

Каковы важнейшие задачи компьютерных наук и их преподавания?

Роберто: Думаю, «компьютерных наук» как хорошо разработанного корпуса знаний не существует. Дело не в том, что компьютерные науки вовсе не наука, а в том, что до сих пор плохо определено, что относится к компьютерным наукам, а что – нет (и что имеет значение, а что – нет). У многих из тех, кто занимается компьютерными науками, нет формальной подготовки в области этих наук.

Луис: Я рассматриваю себя как математика, которому интересна роль компьютеров в математике, но, конечно, я очень люблю компьютеры. :)

Роберто: Даже среди тех, у кого есть формальное образование, нет единства: у нас отсутствует общая основа. Многие думают, что Java создала мониторы, виртуальные машины, интерфейсы (вместо классов) и так далее.

Не являются ли многие программы обучения компьютерным наукам в действительности программами профессионально-технического обучения?

Роберто: Да, являются. И у многих программистов нет даже диплома по компьютерным наукам.

Луис: Я так не считаю, но я не работаю программистом. Напротив, я считаю, что было бы неверным требовать от программиста диплом по

компьютерным наукам, сертификации и тому подобного. Диплом по компьютерным наукам не гарантирует, что человек умеет хорошо программировать, и у многих хороших программистов нет соответствующего диплома (возможно, так было в начале моей карьеры, и я слишком отстал). Я хочу сказать, что диплом по компьютерным наукам не гарантирует, что человек может хорошо программировать.

Роберто: Было бы ошибкой требовать от всех профессионалов наличия соответствующего диплома, но я хотел сказать, что существующая в этой области «культура» слишком слаба. Есть очень немного вещей, знание которых следует считать обязательным. Конечно, работодатель может требовать чего угодно, но не должно быть законов, требующих наличие диплома.

Какую роль играет математика в компьютерных науках, в частности в программировании?

Луис: Ну, я сам математик. Я усматриваю математику всюду. Вероятно, программирование привлекло меня потому, что ему явно присущи математические качества – точность, абстракция, красота. Программа служит доказательством сложной теоремы, которое можно постоянно уточнять и совершенствовать, и при этом оно что-то делает!

Конечно, когда я программирую, то думаю совсем не об этом, но мне кажется, что изучение математики очень важно для программирования в целом. Это помогает выработать определенный склад ума. Гораздо легче программировать, если имеешь привычку думать об абстрактных вещах, управляемых собственными законами.

Роберто: По словам К. Х. Пападимитриу, «компьютерные науки – это новая математика». Без математики возможности программиста ограниченны. С более широкой точки зрения, математика и программирование основаны на одной и той же умственной дисциплине – абстракции. У них также есть важный общий инструмент – формальная логика. Хороший программист постоянно применяет «математику», создавая инварианты кода, модели интерфейсов и так далее.

Многие языки программирования созданы математиками – может быть, поэтому программировать трудно?

Роберто: Предоставлю возможность ответить на этот вопрос находящемуся среди нас математику. :)

Луис: Что ж, я уже сказал, что программированию явно присущи математические качества – точность, абстракция, красота. Проектирование языка программирования напоминает мне построение математической теории: вы создаете мощные инструменты, с помощью которых другие могут делать хорошее дело. Меня всегда привлекали маленькие, но мощные языки программирования. Красота мощных базовых эле-

ментов и конструкций – та же красота мощных определений и базовых теорем.

Как вы распознаете хорошего программиста?

Луис: Это чувствуется. Со временем мне стало легче распознавать плохих программистов – не потому, что плохи их программы (хотя они часто бывают запутанными и неустойчивыми), но потому, что заметно, как неловко они чувствуют себя в программировании – как будто собственные программы для них обуза и загадка.

Как следует учить отладке?

Луис: Не думаю, что отладке можно учить – во всяком случае, формально, – но ее можно освоить, работая с кем-нибудь более опытным, у кого можно перенять приемы отладки: как сузить проблему, как предвидеть и оценивать результаты, что бесполезно и только отвлекает, и так далее.

Роберто: Отладка, по существу, представляет собой решение задачи. Это род деятельности, для которого могут понадобиться все ваши интеллектуальные ресурсы. Конечно, есть ряд полезных приемов (например, по возможности обходиться без отладчика, пользоваться средствами контроля памяти при программировании на языках низкого уровня, таких как Си), но они составляют лишь малую часть отладки. Отладке нужно учиться так же, как программированию.

Как вы тестируете и отлаживаете свой код?

Луис: Обычно я стараюсь строить и тестировать код частями. Я редко пользуюсь отладчиком – в основном, для кода на Си, но не для кода на Lua. Для Lua обычно достаточно расположить в нескольких местах операторы `print`.

Роберто: Мой подход аналогичен. Если я использую отладчик, то обычно только чтобы с помощью `where` найти место в коде, где происходит сбой. Для кода на Си полезны инструменты вроде Valgrind или Purify.

Какую роль играют комментарии в исходном коде?

Роберто: Весьма незначительную. Обычно мне кажется, что если нечто требует комментариев, оно плохо написано. Для меня комментарий означает практически то же, что пометка «этот код нужно потом переписать». Думаю, что понятно написанный код значительно легче читать, чем снабженный комментариями.

Луис: Согласен. Я считаю, что комментарии нужны тогда, когда код не говорит что-то сам за себя.

Как следует документировать проект?

Роберто: Возможно подробнее. Ничто не заменит хорошо написанной и продуманной документации.

Луис: Но написание хорошей документации по развитию проекта возможно лишь в том случае, если позаботиться об этом с самого начала. С Lua этого не получилось: мы никогда не рассчитывали, что Lua в такой степени разовьется и станет так широко применяться. Когда мы писали доклад для HOPL (что заняло почти два года!), оказалось, что трудно вспомнить, в какой обстановке принимались те или иные проектные решения. С другой стороны, если бы мы с самого начала вели формальный протокол своих совещаний, то, вероятно, лишились бы какой-то доли спонтанности и удовольствия.

Какие факторы вы оцениваете в процессе развития основного кода?

Луис: Я бы назвал «простоту реализации». От этого зависят скорость и корректность реализации. В то же время большое значение имеет гибкость, которая позволяет при необходимости изменить реализацию.

Как имеющееся аппаратное обеспечение влияет на мышление программиста?

Луис: Я уже ветеран. :-) Я осваивал программирование на IBM 370. На то, чтобы подготовить задание на перфокартах, поставить в очередь и дождаться распечатки, уходили часы. Мне встречались разные медленные машины. Думаю, программистам это знакомо, потому что не все имеют доступ к самым быстрым машинам. Те, кто пишет приложения для широкого круга пользователей, должны опробовать их на медленных машинах, чтобы иметь представление об ощущениях большинства. Конечно, при разработке можно пользоваться самыми быстрыми машинами: нет ничего скучнее, чем ждать конца компиляции. В эпоху всеобщего распространения Интернета веб-разработчикам следует проверять работу медленных соединений, а не тех сверхбыстрых, которые есть у них на рабочем месте. Рассчитывая на среднюю платформу, вы сделаете свой продукт более быстрым, простым и качественным.

В случае Lua «аппаратное обеспечение» представлено компилятором Си. Работая над реализацией Lua, мы уяснили, что стремление к переносимости оправдывает себя. Практически с самого начала мы реализовывали Lua строго в соответствии с ANSI/ISO C (C89). Это дало возможность Lua работать на особых аппаратных платформах, таких как роботы, прошивки принтеров, сетевые маршрутизаторы и прочие, которые мы никогда не рассматривали для себя в качестве конкретных целей.

Роберто: Есть золотое правило, согласно которому аппаратные ресурсы всегда нужно считать ограниченными. Они действительно *всегда* ограничены. «Природа не терпит пустоты»: любая программа стремится к расширению, пока не израсходует все имеющиеся ресурсы. Кроме того, одновременно с удешевлением ресурсов существующих платформ

появляются новые платформы, на которых действуют жесткие ограничения. Так произошло с микрокомпьютерами, с мобильными телефонами и будет продолжаться впредь. Если вы хотите первым выйти на рынок, нужно очень внимательно учитывать ресурсы, которые понадобятся вашим программам.

Какие уроки из опыта изобретения, дальнейшего развития и распространения вашего языка могли бы извлечь разработчики компьютерных систем в настоящем и обозримом будущем?

Луис: Думаю, нужно помнить о том, что не все приложения будут выполняться на мощных настольных системах или ноутбуках. Многим приложениям придется работать в устройствах с ограниченными возможностями, таких как сотовые телефоны, или еще меньших. Тот, кто проектирует и реализует ПО, всегда должен помнить об этом, потому что нельзя предугадать, где и как им будут пользоваться. Поэтому разрабатывайте его так, чтобы оно требовало минимум ресурсов, и вас приятно удивит, когда его начнут использовать в контексте, который вы не рассматривали в качестве своей цели или даже не подозревали о его существовании. Именно так случилось с Lua! И неспроста; у нас есть шутка (на самом деле не шутка): обсуждая возможность включения в Lua какой-то новой функции, мы спрашиваем: хорошо, а будет ли это работать в микроволновке?

Разработка языка

Lua легко встраивается и требует очень скромных ресурсов. Каковы особенности разработки для ограниченных ресурсов аппаратного обеспечения, памяти и ПО?

Роберто: Вначале мы не ставили перед собой именно эти задачи. Нам просто пришлось решать их при реализации проекта. Со временем эти задачи прояснились. Но я думаю, что главное – стремиться к экономии, всегда и везде. Например, если кто-то предлагает новую функцию, первым вопросом должно быть – а сколько она будет стоить?

Приходилось ли вам отказываться от функций, потому что они слишком дороги?

Роберто: Почти все функции «слишком дороги» в сравнении с тем, что они дают языку. Например, даже простой оператор `continue` не прошел по нашим критериям.

Насколько полезной должна быть функция, чтобы оправдать связанные с ней издержки?

Роберто: Жестких правил нет, но хороший критерий – может ли она нас «удивить», а именно, оказаться полезной в иных областях, чем те,

которые мотивировали ее появление. Я вспомнил еще одно практическое правило: много ли пользователей получит выгоду благодаря этой функции. Одни функции полезны небольшой части пользователей, тогда как другие – почти каждому.

Можете ли вы привести пример функции, добавленной потому, что она полезна многим?

Роберто: Цикл `for`. Даже он вызывал у нас сопротивление, но с его появлением были изменены *все примеры* в книге! Слабые таблицы тоже оказались на удивление полезными. Многие не применяют их, а зря.

После выхода версии 1.0 вы ждали семь лет, прежде чем добавить цикл for. Что вас удерживало? Почему вы все же решили включить его?

Роберто: Мы не включали его, потому что не могли найти для `for` такой формат, который был бы одновременно универсальным и простым. Мы включили его, когда нашли хороший формат с использованием функций генератора. Фактически главным ингредиентом, благодаря которому использование генераторов стало достаточно простым и универсальным, оказалось замыкание, потому что благодаря замыканию функция генератора сама может сохранять внутреннее состояние во время цикла.

Обновление кода с целью использования новых функций и выявившихся практических приемов – еще одна статья расходов?

Роберто: Никто не обязан использовать новые функции.

Часто ли, выбрав некоторую версию Lua, люди придерживаются ее в течение всего времени существования проекта, не переходя на более новые версии?

Роберто: Думаю, что в играх это именно так, но в других областях есть проекты, развивающиеся вместе с используемыми версиями Lua. Обратный пример – World of Warcraft, где произошел переход с Lua 5.0 на Lua 5.1! Но нужно учитывать, что Lua стал гораздо стабильнее по сравнению с первыми версиями.

Как вы распределяете между собой обязанности разработчиков, в частности по написанию кода?

Луис: Код первых версий Lua написал Вальдемар в 1993 году. Начиная примерно с 1995 года основной массив кода пишет и сопровождает Роберто. Я отвечаю за небольшую часть кода: модули дампа байткода и автономный компилятор luac. Мы всегда проводим ревизию кода и рассылаем по электронной почте предложения и сообщения об изменениях в коде, а также проводим долгие встречи, обсуждая новые функции и их реализацию.

Часто ли поступают отклики от пользователей, связанные с языком и его реализацией? Есть ли формальный механизм учета мнения пользователей относительно языка и его модификаций?

Роберто: В шутку мы говорим, что если чего-то не помним, значит, это не столь важно. Почтовый список Lua достаточно активен, но некоторые путают программы open software с общественными проектами. Когда-то я послал в список Lua следующее сообщение, в котором суммируется наш подход:

> Lua – программное обеспечение с открытым исходным кодом, но оно никогда не было открыто для разработки. Это не значит, что мы не прислушиваемся к мнению других. Мы читаем практически все сообщения в почтовом списке. Ряд важных функций Lua возник или был развит благодаря вкладу других разработчиков (например, метатаблицы, сопрограммы и реализация замыканий – если вспомнить только некоторые крупные вещи), но окончательное решение остается за нами. Не потому, что мы считаем себя умнее других. А лишь потому, что мы хотим, чтобы Lua был таким, каким мы его видим, и не стремимся сделать его самым распространенным языком в мире.
>
> По этой причине мы не создаем для Lua открытое хранилище. Мы не хотим связывать себя необходимостью объяснять каждое изменение, которое делаем в коде. Мы не хотим постоянно обновлять документацию. Мы хотим иметь свободу двигаться в неожиданных направлениях или отказываться от них без необходимости объяснять каждый свой шаг.

Почему вы предпочитаете получать предложения и идеи, а не сам код? Наверное, когда вы пишете код сами, то лучше понимаете задачу/решение.

Роберто: Что-то в этом роде. Мы хотим полностью знать, что делается в Lua, поэтому фрагмент кода – не слишком большая помощь. Фрагмент кода не объясняет, почему он написан так, а не иначе, но когда мы понимаем его основные идеи, написание кода становится удовольствием, которого мы не хотим себя лишать.

Луис: Пожалуй, мы также опасались включать сторонний код, право собственности на который не могло быть гарантировано. Нам не очень-то хотелось ввязываться в юридические процедуры, чтобы получить лицензии на код от правообладателей.

Настанет ли момент, когда в Lua будут все функции, которыми вы хотели его снабдить, а изменения будут касаться только реализации (например, LuaJIT)?

Роберто: По-моему, мы уже достигли такого момента. Мы добавили если не все, то большинство функций, которые хотели.

Как вы проводите дымовое тестирование, регрессионное тестирование? Думаю, одно из больших преимуществ открытого хранилища — то, что практически для каждой новой ревизии найдутся те, кто проведет ее автоматизированное тестирование.

Луис: Новые версии Lua не столь часты, поэтому до их выхода они уже проходят большую часть тестирования. Мы выпускаем рабочие версии (пре-альфа), только когда они уже достаточно надежны, чтобы люди могли увидеть, какие новые возможности появились.

Роберто: Мы проводим интенсивное регрессионное тестирование. Дело в том, что поскольку наш код соответствует ANSI C, проблемы совместимости возникают очень редко. У нас нет необходимости тестировать изменения на нескольких разных машинах. Я выполняю все регрессионные тесты после каждого изменения в коде, но все это автоматизировано, и мне нужно лишь ввести команду `test all`.

Когда вы обнаруживаете повторяющуюся ошибку, как вы решаете, что лучше — локальный обход или глобальное исправление?

Луис: Мы всегда стараемся исправлять ошибки, как только они обнаружатся. Но поскольку мы не часто выпускаем новые версии Lua, то обычно ждем, пока накопится столько исправлений, что будет оправданным выпуск новой дополнительной версии. Все изменения, не являющиеся исправлением ошибок, мы оставляем до основных версий. Если проблема сложная (а это случается весьма редко), мы выпускаем временную заплатку в дополнительной версии и делаем глобальное исправление в новой основной версии.

Роберто: Обычно временная заплатка появляется очень быстро. Мы идем на то, чтобы сделать временную заплатку, только когда действительно невозможно внести глобальное исправление, например если для глобального исправления требуется новый несовместимый интерфейс.

Вы по-прежнему разрабатываете с учетом ограниченности ресурсов, хотя с начала вашей работы прошло несколько лет?

Роберто: Конечно, мы постоянно держим это в голове. Мы следим даже за порядком полей в структурах Си, чтобы сэкономить несколько байт. :)

Луис: А сегодня Lua применяют в миниатюрных устройствах чаще, чем раньше.

Как стремление к простоте влияет на разработку языка с точки зрения пользователя? Я имею в виду поддержку классов в Lua, которая в чем-то сильно напоминает ООП в Си (но значительно меньше раздражает).

Роберто: Мы сейчас придерживаемся правила «механизм, а не система». Это упрощает язык, но, как вы сказали, пользователь должен вы-

работать собственную систему. Именно так происходит с классами. Их можно реализовывать многими способами. Одним пользователям это нравится, другим – нет.

Луис: Благодаря этому Lua в некотором роде становится набором «сделай сам».

В Tcl был сделан аналогичный подход, но это привело к дроблению стилей, потому что в каждой библиотеке или фирме применялся свой метод. Представляет ли дробление меньшую проблему в силу предположительных областей применения Lua?

Роберто: Да. Иногда такая проблема возникает, хотя для многих областей применения (например, в играх) ее нет. Lua чаще всего встраивается в некоторое другое приложение, которое и задает более жесткую структуру для унификации стилей программирования. В результате у нас есть Lua/Lightroom, Lua/WoW, Lua/Wireshark, и в каждой такой системе есть своя внутренняя структура.

Так вы считаете гибкость в стиле «мы предоставляем механизмы» огромным достоинством Lua?

Роберто: Не совсем так. Как чаще всего бывает, это компромисс. Иногда очень удобно иметь готовые для применения политики. «Мы предоставляем механизмы» дает большую гибкость, но приводит к большим затратам труда и дроблению стилей. Кроме того, это очень экономичное решение.

Луис: С другой стороны, бывает непросто объяснить это пользователям, чтобы они поняли, что представляют собой эти механизмы и чем вызвано их появление.

Это мешает использовать один и тот же код в разных проектах?

Роберто: Зачастую, да. Это также мешает разработке независимых библиотек. Например, в WoW есть уйма библиотек (там есть даже реализация решения задачи о коммивояжере с помощью генетического программирования), но никто не пользуется ими за рамками WoW.

Вас не беспокоит, что из-за этого Lua в какой-то мере раскололся на WoW/Lua, Lightroom/Lua и так далее?

Луис: Не беспокоит: это один и тот же язык. Только доступны разные функции. Думаю, в результате эти приложения в чем-то выиграли.

Пишут ли серьезные пользователи Lua собственные диалекты поверх Lua?

Роберто: Возможно. Во всяком случае, у нас нет макросов. Думаю, с макросами можно действительно создавать новые диалекты.

Луис: Не диалекты языка в подлинном смысле, но диалекты как специализированные языки, реализованные с помощью функций. Это и было задачей Lua. Когда Lua применяется только для файлов данных, он может быть похож на диалект, но на самом деле это всего лишь таблицы Lua. Есть несколько проектов, где более или менее реализованы макросы. Например, могу вспомнить Metalua. Для Лиспа это проблема.

Почему вы решили сделать семантику расширяемой?

Роберто: Вначале это было средством обеспечить возможности ООП. Мы не хотели добавлять в Lua механизмы ООП, но пользователи требовали их, поэтому возникла мысль таким способом предоставить пользователям механизмы, которых им будет достаточно для самостоятельной реализации механизмов ООП. Мы и сейчас считаем, что это было хорошим решением. Для начинающих это делает ООП на Lua более сложным, зато придает языку значительную гибкость. В частности, если использовать Lua с другими языками (что характерно для Lua), такая гибкость позволяет программисту согласовать модель объектов Lua с моделью объектов внешнего языка.

Насколько существующие сегодня аппаратные и программные среды, сервисы и сети отличны от тех, в которых первоначально разрабатывалась ваша система? Как эти изменения влияют на вашу систему и требуют ли ее адаптации?

Роберто: Поскольку Lua стремится к очень высокому уровню переносимости, я бы не сказал, что сегодняшние «среды» сильно отличались от прежних. Например, когда мы начинали разработку Lua, DOS/Windows 3 устанавливались на 16-битные машины, а некоторые старые машины были еще 8-битными. Сейчас уже нет 16-разрядных настольных машин, но ряд платформ, на которых применяется Lua (встроенные системы), все еще являются 16- и даже 8-разрядными.

Сильно изменился только Си. Когда мы только начинали работу над Lua в 1993 году, ISO (ANSI) C не был еще так распространен, как сегодня. На многих платформах по-прежнему используется K&R C, а во многих приложениях применяется сложная система макросов для осуществления компиляции с K&R C и с ANSI C, когда основным различием является объявление заголовков функций. В то время выбор ANSI C был смелым решением.

Луис: При этом мы пока не чувствуем необходимости переходить на C99. Lua реализован на C89. Возможно, нам придется частично использовать C99 (особенно новые типы с определенными размерами), если возникнут неполадки при переходе на 64-битные машины, но не думаю, что это произойдет.

Если бы вы стали заново создавать ВМ Lua, то по-прежнему остановились бы на ANSI C или предпочли бы язык, более подходящий для низкоуровневой межплатформенной разработки?

Роберто: Нет. Из всех известных мне в данное время языков ANSI C является самым переносимым.

Луис: Есть превосходные компиляторы ANSI C, но даже с помощью их расширений не удается существенно улучшить продуктивность.

Роберто: Не так легко усовершенствовать ANSI C, сохранив его переносимость и производительность.

Кстати, это C89/90?

Роберто: Да. C99 еще не слишком упрочил свое положение.

Луис: Кроме того, не думаю, что мы получили бы от C99 много новых функций. В частности, я имею в виду помеченные `goto`, которые gcc предлагает в качестве альтернативы `switch` (в главном переключателе выполнения ВМ).

Роберто: На многих машинах это позволило бы повысить производительность.

Луис: Мы проверяли это в самом начале, и кто-то проводил тестирование недавно; выигрыш не впечатляет.

Роберто: Отчасти из-за нашей архитектуры, основанной на регистрах. Для нее лучше, чтобы было меньше кодов операций, и каждый выполнял больше работы. В таких условиях значение диспетчера падает.

Почему вы сделали ВМ, основанную на регистрах?

Роберто: Чтобы избежать этих команд – getlocal/setlocal. Мы также хотели опробовать эту идею на практике. Мы думали, что если она окажется неудачной, то можно будет хотя бы написать по этому поводу какие-то статьи. В итоге все сработало весьма неплохо, а статью мы написали всего одну. :)

Облегчает ли отладку запуск через ВМ?

Роберто: Не «облегчает», а меняет всю концепцию отладки. Всякий, кому приходилось отлаживать программы как на компилируемых, так и на интерпретируемых языках (например, Си и Java), знает, какая между ними пропасть. Хорошая ВМ делает язык безопасным, в том смысле, что ошибки всегда можно понять в терминах самого языка, а не машины, на которой происходит выполнение (например, ошибка сегментации).

Как влияет на отладку независимость языка от платформы?

Роберто: Обычно она упрощает отладку, потому что чем меньше язык зависит от платформы, тем больше он требует надежного абстрактного описания и поведения.

Каждый знает, что ошибки в программах неизбежны. Задумывались ли вы над тем, какие возможности следует добавить в язык или изъять из него, чтобы облегчить этап отладки?

Роберто: Конечно. Первым делом нужно позаботиться о хороших сообщениях об ошибках.

Луис: Сообщения об ошибках в Lua улучшились по сравнению с ранними версиями. Мы заменили проклятое сообщение «выражение вызова – не функция», просуществовавшее до Lua 3.2, гораздо лучшими сообщениями, например «попытка вызова глобальной 'f' (нулевое значение)». Начиная с Lua 5.0 мы также применяем символическое исполнение байт-кода, стараясь выводить полезные сообщения об ошибках.

Роберто: Разрабатывая язык, мы всегда старались избежать конструкций, действие которых трудно объяснить. Если что-то трудно понять, то это труднее и отлаживать.

Как связаны конструкция языка и конструкция программ на этом языке?

Роберто: Для меня, во всяком случае, главный компонент при разработке языка – user cases, то есть представление о том, как пользователи станут применять отдельные функции языка и их комбинации. Разумеется, программисты всегда находят новые способы применения языка, и хороший язык позволяет использовать его неожиданным образом, но «нормальное» использование языка соответствует тому, что имели в виду разработчики языка при его создании.

В какой мере реализация языка влияет на его конструкцию?

Роберто: Это влияние обоюдно. Реализация оказывает огромное влияние на язык: нельзя вкладывать в него то, что вы не можете эффективно реализовать. Некоторые об этом забывают, но эффективность всегда является важным (даже важнейшим) требованием при разработке любого ПО. Но и конструкция языка может сильно влиять на его реализацию. На первый взгляд, некоторые отличительные черты Lua обусловлены его реализацией (небольшой размер, хороший API к C, переносимость), но в возможности такой реализации ключевую роль играет конструкция Lua.

В одной из ваших статей я прочел, что «в Lua используются написанные вручную сканер и рекурсивный нисходящий парсер». Почему вы решили сделать парсер вручную? Вы были изначально уверены, что

он может оказаться значительно эффективнее, чем сгенерированный с помощью yacc?

Роберто: Первые версии Lua использовали и lex, и yacc, но одной из главных целей при разработке Lua была возможность использовать его как язык описания данных, что-то вроде XML.

Луис: Но задолго до него.

Роберто: Lua вскоре стал использоваться в файлах данных объемом несколько мегабайт, и сгенерированный с помощью lex сканер быстро превратился в узкое место. Написать хороший сканер вручную довольно легко, и в результате одного этого нововведения эффективность Lua возросла примерно на 30%.

Решение перейти с yacc на парсер, написанный вручную, пришло значительно позже и далось нелегко. Все началось с проблем со скелетным кодом, который использует большинство реализаций yacc/bison.

В те времена они не обеспечивали хорошую переносимость (например, в некоторых использовался заголовок malloc.h, не соответствующий ANSI C), и мы не могли в достаточной мере контролировать их общее качество (например, обработку ими переполнений стека или ошибок выделения памяти), а также они не были реентерабельными (в смысле возможности вызова парсера во время проведения синтаксического анализа). Кроме того, восходящий анализатор не столь хорош, как нисходящий, когда нужно генерировать код динамически, как это делается в Lua, из-за трудности работы с «унаследованными атрибутами». После модификации мы увидели, что наш самодельный парсер несколько быстрее и меньше, чем генерируемый yacc, но не это было главной причиной перехода.

Луис: Нисходящий анализатор также позволяет лучше генерировать сообщения об ошибках.

Роберто: Но я не посоветовал бы применять какого-либо рода рукодельный парсер для языка, синтаксис которого не достиг достаточной зрелости. И несомненно, что LR(1) (или LALR, или даже SRL) гораздо мощнее LL(1). Чтобы получить приемлемый парсер, даже для такого простого синтаксиса, как в Lua, пришлось прибегнуть к некоторым ухищрениям. Например, процедуры для бинарных выражений вообще не следуют оригинальной грамматике; вместо этого мы применили хитрый рекурсивный метод, использующий приоритеты. В своих лекциях по компиляторам я всегда рекомендую студентам прибегать к yacc.

Не вспомните ли какой-нибудь интересный случай из своей преподавательской работы?

Роберто: Когда я начинал преподавать программирование, основной вычислительной техникой, доступной студентам, был мэйнфрейм. Од-

нажды случилось так, что задание по программированию, выполненное очень хорошей группой студентов, невозможно было даже скомпилировать. В разговоре со мной они клялись, что тщательно протестировали программу на нескольких контрольных примерах, и все было в порядке. Вычислительная среда у них и у меня была, разумеется, одна – тот самый мэйнфрейм. Загадка раскрылась через несколько недель, когда я узнал, что компилятор Паскаля был обновлен. Обновление произошло после того, как они закончили работу, и перед тем, как я начал ее проверять. В их программе была крохотная синтаксическая ошибка (кажется, лишняя точка с запятой), которую прежний компилятор не замечал!

8

Haskell

Haskell – чисто функциональный нестрогий язык, изначально разработанный, чтобы служить открытым стандартом для современных функциональных языков. В таком качестве Haskell Report появился в 1990 году, а в 1998 году был принят в качестве стандарта. Но с годами язык существенно развился, особенно в части системы типов, где было много нововведений. Недавно популярность Haskell выросла: появились многочисленные независимые библиотеки, практические приложения, заметно улучшенные реализации (среди которых выделяется Glasgow Haskell Compiler, GHC), а также постоянно растущее сообщество сотрудничающих между собой пользователей. Особый интерес Haskell представляет для исследования проблемно-ориентированных языков, параллелизма и упорядоченного контроля состояний. Высокий уровень абстракций, обеспечиваемый Haskell для решения задач, является непревзойденным – во всяком случае для тех, кто разобрался в подходе к разработке программного обеспечения, предлагаемом Haskell.[1]

[1] Эта глава представляет собой почтовую переписку с Полом Худаком, Джоном Хьюзом, Саймоном Пейтон-Джонсом и Филипом Уодлером, объединенную с телефонным интервью с Саймоном Пейтон-Джонсом.

Функциональная команда

Как вы работаете над языком в команде?

Саймон Пейтон-Джонс: Случилось так, что у нас была общая задача (разработка стандартного нестрогого функционального языка) и хорошо совместимые технические планы. В нашей статье об истории Haskell[1] описаны разные виды применявшейся тактики (личные встречи, электронная почта, «начальники» по редактированию и синтаксису). И у нас не было пользователей, которым требовалась обратная совместимость. В работе не участвовали никакие компании, что избавляло нас от проблем с (несовместимыми) корпоративными целями.

Джон Хьюз: Мы одинаково смотрели на вещи. Все мы были увлечены функциональным программированием. Эта область вызывала в то время огромный интерес, и все мы стремились внести свой посильный вклад в реализацию мечты о функциональном программировании. Помимо того, мы с большим уважением относились друг к другу. Думаю, что увлеченность и взаимное уважение существенно помогли нам принимать решения по многим сложным вопросам, которые нельзя было обойти.

Пол Худак: Одинаковые взгляды – прежде всего. Сомневаюсь, что без этого можно далеко продвинуться. Первый состав комиссии по Haskell отличался исключительной общностью взглядов.

Добавьте к этому энергичность. Комиссия по Haskell обладала потрясающей энергией. Она была похожа на дикую стаю.

Необходимо также смирение. Как в «Мифическом человеко-месяце»: добавляя новых работников, вы не обязательно ускоряетете получение результата, потому что, несмотря на общие взгляды, разногласия неизбежны. У нас была масса разногласий, но нам хватало смирения, чтобы находить компромиссы.

Наконец, необходимо руководство. Нам повезло, что мы смогли поделить руководство, – это достаточно необычно. У нас всегда был кто-то один, кто руководил работой, мы всегда знали, кто этот человек, и верили, что он справится с этой работой.

Как вам удавалось сплавлять идеи в единое целое?

Саймон: Мы много спорили, преимущественно в переписке. Мы излагали технические аргументы в пользу своей точки зрения и распространяли их. Мы стремились к компромиссу, потому что нам важно было

[1] «Being Lazy with Class: the history of Haskell», Proc Third ACM Conference on the History of Programming Languages (HOPL III), *http://research.microsoft.com/~simonpj/papers/history-of-haskell/index.htm*

получить готовый язык. И потому что мы признавали справедливость аргументов другой стороны.

Джон: Иногда мы сохраняли два перекрывающих друг друга метода, как, например, стили равенства и выражения, оба из которых поддерживаются в Haskell. Однако чаще мы долго обсуждали технические стороны конкурирующих идей, приходя в итоге к согласию. Думаю, важную роль сыграла семантика: хотя мы никогда не создали полной формальной семантики для всего Haskell, мы систематически формализовали отдельные части проекта, а семантическое уродство всегда было веским аргументом в пользу отказа от предложения. То, что мы не упускали из виду формальную семантику, помогло достичь цельной архитектуры.

Пол: Благодаря спорам – преимущественно на техническом уровне, где часто очевидно, что «правильно» и что «ошибочно», – но также и на субъективном/эстетическом/глубоко личном уровне, где нет правильного и ошибочного. Иногда эти споры затягивались надолго (некоторые ведутся и по сей день), но нам как-то удавалось сквозь них прорваться. Если проблема казалась несущественной, мы часто оставляли окончательный выбор за руководителем. Например, у нас был «начальник синтаксиса», который мог принимать окончательные решения по синтаксическим деталям.

Как вы определяли, какие идеи – лучшие, и как вы «расправлялись» с теми функциями, которые вам не нравились?

Пол: Лучшие идеи были видны невооруженным глазом – как и худшие. Сложнее было с теми проблемами, лучшее решение для которых не было очевидно.

Саймон: Если какие-то функции нам не нравились, мы спорили и выдвигали против них аргументы. Если возражающих было достаточно много, пробиться такой идее было трудно. Но я не помню случая, чтобы кто-то один или с небольшой группой упорно отстаивал идею, которая оказалась бы в итоге забаллотирована. Вероятно, это свидетельствует об общности технической базы участников проекта.

Можно сказать, что обычно разногласия не создавали проблем. Что было трудно, так это найти добровольцев для работы с мелочами. В языках огромное множество мелких деталей. Что происходит в том или ином неясном случае? Масса деталей в библиотеках. Романтики никакой, но делать это необходимо.

Джон: Лучшие идеи обнаруживаются выражением восторга, которое им сопутствует! Такой была система классов: увидев ее, мы все с ума посходили. Это не значит, что потом не пришлось много над ней работать,

занимаясь такими деталями, как экземпляры по умолчанию и взаимодействие с модулями.

Что касается функций, которые нам не нравятся, то они, как правило, оказываются самыми интенсивно обсуждаемыми. Пользователи постоянно выражают свое недовольство ими, и при подготовке новой версии кто-то обязательно скажет: а не пора ли нам избавиться от функции *X*? – и ее начинают снова обсуждать. Но, по крайней мере, мы можем ясно увидеть, почему у нас присутствуют функции, которые нам не нравятся, и почему мы не можем от них избавиться.

В ряде случаев проблемы возникали с самого начала. Например, вечно вызывающее недовольство «ограничение мономорфизма» испокон незыблемо – лишь потому, что оно решает реальную проблему и никто не придумал, как сделать лучше. В других случаях мы пересматривали свои решения в соответствии с полученным практическим опытом. Мы изменили обработку явной строгости после того, как оказалось, что первоначальное решение препятствует развитию программы, для чего пришлось пожертвовать красотой семантики – один из редких таких случаев. Мы убрали перегрузку списочных выражений (list comprehension), когда оказалось, что она вызывает затруднения у начинающих программистов. То, что мы смогли вернуться и исправить ошибки исходя из полученного опыта, даже внеся в язык изменения, приведшие к обратной несовместимости, в конечном счете способствовало усовершенствованию языка.

Есть ли преимущества у работы в команде? Приходилось ли вам идти на компромисс?

Саймон: Работа в команде – это *самое* важное! У всех нас были собственные языки, и мы считали, что, разработав *общий* язык, мы перестанем дублировать свою работу и добьемся большего доверия пользователей, потому что все мы станем поддерживать единый язык. Эти надежды полностью оправдались: Haskell стал огромным успехом по любым меркам, и в особенности в сравнении с нашими первоначальными ожиданиями.

Пол: Это было явным преимуществом. Несмотря на общее видение, каждый из нас обладал своими особенными навыками. Мы доверяли друг другу и многому друг у друга научились. Это было замечательное сотрудничество умных, энергичных и трудолюбивых личностей. Возможно, в одиночку разработать Haskell не удалось бы.

Джон: Компромиссы могут быть полезны!

Работа в команде явно дала нам преимущества. Наши опыт и знания дополняли друг друга, и я определенно считаю, что разработка языка более широкой группой привела бы к более широко применимому ре-

зультату, чем тот, который кто-либо из нас мог получить в одиночку. Возможно, работая в одиночестве, можно было бы получить более компактный, простой и даже красивый язык, но вряд ли он оказался бы таким полезным.

Кроме того, это позволяло изучать все трудные проектные решения с разных точек зрения, что и делалось. Одна голова хорошо, а две и больше – лучше. Решение, которое кажется вполне разумным одному, другому с первого взгляда представляется ошибочным, и мы не единожды отвергали идеи после того, как таким образом выявляли в них существенные недостатки, а в результате обнаруживали более удачные идеи. Думаю, наша тщательность существенно повлияла на качество разработки. Звучит вполне диалектически: тезис + антитезис = синтез.

Эволюция функционального программирования

Что отличает функциональные языки программирования от других языков?

Саймон: Ну, это совсем просто: контроль над побочными эффектами.

Джон: Очевидно, тщательный контроль над побочными эффектами. Функции первого класса (хотя они все чаще появляются и в процедурных языках). Краткие обозначения для чисто функциональных операций – от создания структур данных до списочных выражений. Думаю, очень важна также облегченная система типов – будь это чисто динамические системы типов, как в Scheme и Erlang, или полиморфные системы вывода типа, как в Haskell и ML. В обоих случаях системы облегченные – в том смысле, что типы не мешают даже интенсивному использованию функций высших порядков, а это сердцевина функционального программирования.

Думаю, большое значение имеют также отложенные вычисления, но они есть не во всех функциональных языках.

Пол: Абстракция, абстракция и еще раз абстракция – что, по моему мнению, включает в себя функции высших порядков, монады (абстракция управления), различные абстракции типов и так далее.

Чем хорош язык без побочных эффектов?

Саймон: Можно думать только о *значениях*, а не о *состоянии*. Передавая данные на вход функции, вы каждый раз получаете один и тот же результат. Это может быть использовано в логических выводах, компиляции, параллельных вычислениях.

Как говорит Дэвид Балабан (Amgen, Inc.), «ФП сокращает разрыв между мыслью и кодом, а это главное».

Джон: Сейчас таких языков фактически нет. Программы на Haskell могут иметь побочные эффекты, если выбран определенный тип или используются «небезопасные» операции. Программы на ML и Erlang тоже могут иметь побочные эффекты. Дело лишь в том, что в этих языках побочные эффекты не превращаются в основу программирования, а являются исключениями: они не одобряются и тщательно контролируются. Поэтому я бы переформулировал ваш вопрос так: в чем преимущества программирования, когда побочных эффектов почти нет?

Тут многие стали бы говорить о логических выводах, и я тоже так сделаю, но с весьма практической точки зрения. Представим тестирование функции в процедурном языке. Когда вы тестируете код, вам нужно взять различные входные данные и убедиться, что выходные данные при этом соответствуют спецификации. Но при наличии побочных эффектов эти входные данные состоят не только из параметров функции, но и из тех элементов глобального состояния, которые считывает функция. Точно так же выходные данные состоят не только из значения функции, но и из всех тех элементов состояния, которые она модифицирует. Чтобы по-настоящему протестировать функцию, у вас должна быть возможность поместить входные данные в тех участках состояния, которые она считывает, и прочесть участки состояния, которые она модифицирует... но у вас может даже не быть прав для прямого доступа к этим участкам, так что придется строить нужное вам для тестирования состояние косвенным образом, с помощью ряда вызовов других функций, и получать результат работы функции, вызывая другие функции после того, как отработает тестируемая функция, чтобы извлечь информацию, которую она, по вашим предположениям, изменила. Возможно, вы даже не будете точно *знать*, какие участки состояния считываются и записываются! И вообще говоря, чтобы проверить постусловие функции, требуется доступ к состоянию как *до* ее выполнения, так и *после* – одновременно! Поэтому нужно скопировать состояние перед тестированием, тогда у вас будет потом вся необходимая информация.

Сравните это с тестированием чистой функции, которая зависит только от своих аргументов и все действие которой заключается в возврате результата. Так жить гораздо проще. Даже если у программы должно быть множество побочных эффектов, имеет смысл выделить в ней как можно больше функциональности в отдельный эффективно тестируемый код без побочных эффектов и заключить его в тонкую оболочку, создающую побочные эффекты. Дон Стюарт прекрасно описывает в своем блоге[1] применение этого метода для оконного менеджера XMonad.

[1] *http://cgi.cse.unsw.edu.au/~dons/blog/2007/06/02#xmonad-0.2*

В результате передачи всего, от чего зависит функция, в виде аргументов, облегчается также выявление зависимостей. Даже на Haskell *можно* писать программы, манипулирующие одним большим состоянием, которое передается в виде аргумента во все функции и возвращается в качестве результата теми из них, которые его модифицируют. Но обычно этого не делают: вы передаете только ту информацию, которая нужна функции, и получаете от нее только ту, которую генерирует сама функция. В результате зависимости видны гораздо лучше, чем в процедурном коде, где каждая функция, в принципе, может зависеть от любой части состояния. А забыв о таких зависимостях, как раз и можно получить самые неприятные ошибки!

Наконец, если начать программировать с побочными эффектами, приобретает важность порядок вычисления. Например, сначала нужно открыть дескриптор файла, а потом выполнять с ним какие-то операции; нужно не забыть закрыть его, сделав это один раз, а после закрытия больше им не пользоваться. Каждый объект с состояниями налагает ограничения на порядок использования своего API, и эти ограничения наследуются в более крупных фрагментах кода. Например, требуется вызывать некоторую функцию перед тем, как закрыть журнальный файл, потому что она иногда делает в нем записи. Если забыть про эти ограничения и вызвать функции не в том порядке, то – бах! – сбой в программе. Это существенный источник ошибок. Например, Microsoft Static Driver Verifier фактически проверяет, учитываете ли вы ограничения для объектов ядра Windows, имеющих состояние. Пишите программы так, чтобы не было побочных эффектов, и вам не придется беспокоиться о таких вещах.

Самые непростые ошибки, которые попадались мне в последнее время, привели к библиотекам Erlang с хранящим состояние API, которые я применяю в коде с очень сложным и динамически определяемым порядком исполнения. В конце концов мне удалось заставить свой код работать только после того, как поверх стандартного API я построил другой, не имеющий побочных эффектов. Боюсь, я просто лишен способности заставить работать код с побочными эффектами! (Впрочем, не исключено, что заниматься процедурным программированием проще, не будучи отягощенным многолетним опытом функционального программирования...) :-)

Да, нельзя не упомянуть легкость распараллеливания.

Пол: Душевное равновесие и удовольствие от решения головоломки! :-)

Джон упомянул параллелизм. Есть ли какие-то другие изменения в области компьютеров, благодаря которым функциональное программирование становится еще более полезным и нужным?

Саймон: Думаю, есть устойчивая тенденция к увеличению контроля побочных эффектов в языках программирования всех типов. Что я имею в виду, может стать понятнее из этого коротенького видеоролика:

http://channel9.msdn.com/ShowPost.aspx?PostID=326762

Можно ли рассматривать стремление избавиться от побочных эффектов как естественный этап развития структурного программирования, сравнимый с переходом к языкам высокого уровня с развитыми управляющими структурами и циклами от простых переходов — операторов goto? Является ли программирование без побочных эффектов очередным шагом в том же направлении?

Саймон: Можно взглянуть на него и в таком ракурсе. Я бы проявил некоторую осторожность, потому что, говоря о «структурном программировании», многие подразумевают весьма разные вещи. Мне всегда казалось, что следует тщательно выбирать выражения, прежде чем делать лаконичные заявления.

Вспомним классическое письмо Дейкстры «Goto Considered Harmful» («Доводы против оператора `goto`»), в котором было сказано: уберите `goto`, и ваши программы станут понятнее, проще компилируемыми и так далее. В таком случае можно считать, что чистота, достигаемая устранением присваивания, позволяет с меньшими трудами разобраться в программе. Но мне кажется, что было бы ошибкой рассматривать функциональное программирование исключительно как упражнение в эстетизме («уберем все порочное и дурное и сделаем вашу жизнь скучной и невыносимой»).

Но мы говорим не просто «мы кое-что уберем», а «мы кое-что уберем, но взамен дадим и отложенное вычисление, и функции высшего порядка, и очень развитую систему типов, и эти штуки — монады». Такая замена вынудит вас совсем по-другому воспринимать программирование, поэтому такой переход не будет безболезненным, но он будет щедро вознагражден.

Каковы особенности обработки ошибок в функциональном программировании?

Саймон: Ошибки воспринимаются иначе — скорее, как «ошибочные значения», а не «распространение исключений». Ошибочное значение напоминает NaN в вычислениях с плавающей запятой. В результате обработка ошибок больше ориентируется на значения, чем на порядок исполнения, что в целом хорошо. В соответствии с этим тип функции вполне может выражать ошибку в ее поведении. Например, вместо:

```
item lookup( key ) /* Может генерировать исключение “не найден” */
```

мы пишем:

```
lookup :: Map -> Key -> Maybe Item
```

где тип данных `Maybe` описывает посредством значения возможность ошибочного выполнения.

Чем отличается отладка в функциональном программировании?

Пол: Во-первых, я всегда подозревал, что принятый в процедурных языках метод отладки путем «трассировки» порочен – даже для программ на самих этих языках! Некоторые известные процедурные программисты фактически избегают этого метода, предпочитая ему более строгие, основанные на тестировании или верификации.

А что касается функциональных языков, особенно «ленивых», то в них трудно разумным образом определить «порядок исполнения», поэтому трассировка – не самый удачный выбор для отладки. В GHC есть средство трассировки механизма редукции графа, на котором основан механизм вычисления, но, как мне кажется, оно слишком детально раскрывает процедуру вычисления. Вместо этого созданы такие отладчики, как Buddha, опирающиеся на «зависимости между данными», что гораздо более соответствует декларативным принципам функционального программирования. Но, как ни удивительно, за все годы программирования на Haskell я фактически не использовал ни Buddha, ни отладчик GHC, ни какой бы то ни было еще отладчик. Я считаю, что достаточно тестирования: тестирую небольшие фрагменты кода с помощью QuickCheck или аналогичного инструмента, обеспечив строгую проверку, а затем – и это главное – просто изучаю код и ищу, почему он работает не так, как надо. Думаю, подобным образом работают многие, иначе было бы гораздо больше исследований, посвященных отладчикам Haskell, которые одно время были в моде. Любопытно было бы узнать, как разные люди отлаживают программы на Haskell на практике.

Но нужно заметить, что есть и другой, гораздо более интересный вид отладки, а именно профилирование времени и памяти, в особенности памяти. Утечка памяти – скрытый бич функционального программирования, и профилирование памяти – важный инструмент борьбы с ними.

Легче ли изучать языки функционального программирования тому, у кого нет многолетнего опыта работы с процедурными языками?

Саймон: Вряд ли. Возможность освоить ФП, по-видимому, существенно коррелирует с общими способностями к программированию. Оно, несомненно, требует некоторой перестройки мышления, но толковым программистам это удается. Я считаю неубедительным объяснение ограниченной распространенности ФП тем, что большинство программистов первоначально изучает процедурные технологии.

Более веская причина – существование сильного эффекта блокировки. Многие работают на C++, поэтому для C++ есть фантастическое количество компиляторов, инструментов, резерв подготовленных программистов. Но и это не самая серьезная причина: посмотрите, как быстро обрели успех Python и Ruby.

Джон: Нет, это миф. Огромную часть опыта можно использовать в чистом виде, будь то понимание важности абстракции в программировании, знание алгоритмов или структур данных или просто понимание того, что языки программирования – это формальные языки. Опытные программисты на Си/C++ , которым я преподаю Haskell, в целом успевают значительно лучше новичков. Они понимают, что такое «синтаксическая ошибка»; они могут не знать систему типов, но они знают, что такое ошибка типа; они знают, что осмысленные имена переменных не помогут компьютеру «понять» их программу и исправить их ошибки!

Думаю, этот миф создали процедурные программисты, обнаружив, что функциональное программирование *сложнее, чем они предполагали*. Опытные программисты легко схватывают новые языки, потому что им не нужно осваивать такие базовые понятия, как переменные, присваивание и циклы. С функциональными языками это не проходит: даже опытному программисту нужно изучить некоторые новые понятия, чтобы суметь хоть что-то сделать. Поэтому они считают функциональное программирование «трудным», хотя осваивают его значительно быстрее и легче, чем полные новички.

Пол: Обычно я говорил, что переход затруднен из-за укоренившихся привычек, но теперь сомневаюсь в этом. Я считаю, что лучшие, умнейшие и опытнейшие программисты способны легко изучить и полюбить Haskell. Опыт помогает им оценить абстракцию, строгий контроль побочных эффектов, сильную систему типов и так далее. Менее опытным программистам это часто не удается.

Как вы думаете, почему ни одному функциональному языку не удалось захватить ведущие позиции?

Джон: Слабый маркетинг!

Я не имею в виду пропаганду – этого было достаточно. Я говорю о тщательном выборе ниши, которую можно занять, вслед за чем должны последовать решительные усилия, чтобы обеспечить функциональному программированию существенное превосходство в этой нише. В благословенные 1980-е мы считали, что функциональное программирование пригодно всюду, но сказать, что новая технология «пригодна всюду», – все равно, что объявить ее «нигде себя особо не проявляющей». Что требуется для бренда? Это проблема, которую Джон Лончбери (John Launchbury) четко описал, выступая на ICFP. Galois Connections едва

не обанкротилась, выступая под лозунгом «ПО на функциональных языках», но стала неуклонно крепнуть, сосредоточившись на «высоконадежном ПО».

Многие совершенно не представляют, как вводятся технологические инновации, и думают, что более эффективные технологии добиваются господства «сами по себе» (эффект «лучшей мышеловки»), но мир устроен иначе.

Мои взгляды на эти проблемы сильно изменились под влиянием таких книг, как «Crossing the Chasm» (Преодоление пропасти) Мура (изд. HarperBusiness) и «The Innovator's Dilemma» (Дилемма инноватора) Кристенсена (изд. Collins Business). Если в 1980-х и существовала подходящая ниша, то это было параллельное программирование, но ему не придавали особую важность до самого недавнего времени (когда наступила эпоха многоядерных процессоров) благодаря невероятной изобретательности архитекторов компьютерных систем. Думаю, эти проблемы были важнее технологических, которые тоже, конечно, важны, например низкая производительность.

Пол: Потому что отличие от традиционного программирования слишком разительно. Это отличие влечет трудности восприятия, трудности изучения и трудности поддержки (включая библиотеки, реализации и так далее).

Меняется ли это положение?

Саймон: Функциональное программирование – это ставка на далекий успех. Это в корне иной взгляд на программирование в целом. Из-за этого его трудно изучать, и даже после изучения трудно принять, поскольку различие носит революционный, а не эволюционный характер.

До сих пор не ясно, сможет ли ФП стать в итоге ведущим стилем. Зато ясно, что ФП оказало влияние на ведущие языки, и это влияние растет. Вот вам примеры: сборка мусора, полиморфные типы (генерики), итераторы, LINQ, безымянные функции и так далее.

Есть две причины роста влияния ФП. Во первых, по мере масштабирования программ и роста внимания, уделяемого корректности, цена неограниченных побочных эффектов и преимущества более функционального стиля становятся очевиднее. Во-вторых (хотя, вероятно, это более краткосрочное обстоятельство), многоядерные процессоры и параллелизм вернули интерес к чистым вычислениям, или хотя бы таким вычислениям, где тщательно контролируются побочные эффекты. Свежий пример – Software Transactional Memory (STM).

При всем этом в последнее время наблюдается существенное расширение сообщества Haskell, и не исключено, что какой-нибудь язык, который можно считать функциональным, попадет в число основных

языков. (Но я подозреваю, что если такое случится, его назовут Java3 и синтаксически он будет похож на ОО-язык.)

Джон: Наверняка. Возьмите Erlang – язык, ориентированный на весьма узкую область: надежные распределенные системы, требующиеся в телекоммуникациях, с огромным набором библиотек, решающих все задачи телекоммуникаций, и тем счастливым обстоятельством, что все интернет-серверы требуют фактически одинаковых характеристик. Возможно, Erlang не относится к основным языкам даже в области телекоммуникаций, но число его пользователей там очень велико и растет экспоненциально. Выбор Erlang для приложения в области телекоммуникаций сегодня не встречает сопротивления – это опробованная технология.

Haskell пока не достиг такого положения, но интерес к нему растет, и время от времени возникают неожиданные его применения. То же самое относится к OCaml.

Многоядерные процессоры создают уникальные возможности для функционального программирования: общепринято мнение, что мы не умеем писать для них программы, и есть масса людей, пытающихся найти альтернативные пути программирования параллельных систем, включая функциональное программирование. Любопытно, что до сих пор автоматическое распараллеливание старого кода иногда считают «временным» решением, рассматривая функциональное программирование как возможный «долгосрочный» подход. Но дело в том, что если стоит задача начать сейчас разработку продукта, который должен использовать восемь ядер и быть готов через год, то решение писать последовательный код на Си в расчете на его автоматическое распараллеливание оказывается *чрезвычайно* рискованным. Выбор же Concurrent Haskell или SMP Erlang не связан с риском, потому что их технологии действуют уже сегодня.

На рынке уже есть двухъядерные продукты Erlang, работающие вдвое быстрее благодаря добавочному ядру. Пройдет совсем немного лет, и легкость распараллеливания станет важнейшим преимуществом, так что функциональные языки имеют все шансы успешно пережить грядущие перемены.

Пол: Да, возможности более широкого распространения изменились по нескольким причинам:

- В другие языки были включены некоторые хорошие идеи, поэтому они больше не кажутся слишком радикальными.
- Те, кто пришел в программирование за последние 15 лет, лучше знакомы с идеями современных языков программирования, математи-

кой, формальными методами, что опять-таки делает эти идеи менее радикальными.

- Появилось много библиотек, реализаций и инструментов, делающих применение языка проще и практичнее.
- Появилось заметное количество удачных приложений, написанных на Haskell (и других функциональных языках), отсюда уверенность в том, что «это будет работать».

Если через 50 лет после своего возникновения функциональное программирование по-прежнему считается полезным, говорит ли это что-нибудь о современном состоянии компьютерной сферы?

Саймон: Думаю, это кое-что говорит о самом функциональном программировании. Мне нравится ФП, потому что оно сохраняет верность своим исходным принципам и в то же время имеет практическую ценность.

Говоря о «верности принципам», я имею в виду, что языки и их реализации (особенно чистые языки вроде Haskell) очень близко следуют положенным в их основу необычайно простым математическим принципам – в отличие от мощных, но гораздо более специализированных языков вроде Python или Java. Это означает, что ФП не должно выйти из моды: оно представляет собой фундаментальный способ компьютерных представлений, который не может быть подвержен влияниям моды.

Под «практической ценностью» я понимаю то, что ФП стало гораздо легче в применении, чем десяток лет назад, благодаря существенно улучшенным реализациям и библиотекам. Это делает преимущества принципиального подхода доступными гораздо более широкому кругу публики.

По мере того как все большую озабоченность вызывают:

- Безопасность
- Параллелизм
- Ошибки, вызываемые побочными эффектами

будут расти интерес к ФП и его применение. Если угодно, мы движемся к такому моменту, когда издержки ФП начинают играть сравнительно меньшую роль, а ценность преимуществ возрастает.

Язык Haskell

Возвращаясь к одному из предыдущих ответов Джона: что вас «свело с ума», когда вы разрабатывали систему классов?

Саймон: Мы знали, что есть целый ряд проблем, связанных с тем, как определить равенство для произвольных типов, как показывать и печатать произвольные типы и как оперировать числовыми данными. Нам были нужны целые числа и числа с плавающей запятой, а также целые числа произвольной длины. Не хотелось, чтобы программисту приходилось писать `plus int`, или `plus float`, или `plus arbitrary precision integer`.

Мы хотели, чтобы можно было просто написать `A + B` и получить правильный результат. ML решал эту проблему тем, что позволял написать `A + B`, но тип сложения определялся локально. Если вы напишете `f(x,y) = x + y + 1`, то система скажет: «Нет, мне этого мало. Укажите сигнатуру типа `f( )`, чтобы я знала, что это за плюс, – для целых, для `float` или для `double`».

Из-за этого приходится разносить информацию о типах по разным местам программы.

Саймон: Не только это. Данная функция может быть полезна при вызове и с действительными числами, и с целыми, и с числами двойной точности. Привести все к одному типу – это морока.

Теряется общность. Приходится писать не одну, а три функции: `f float`, `f double` и `f integer`, у которых одинаковое тело, но разный тип. А какую из них должен вызвать компилятор? Возвращается та же проблема `plus int`, но на следующем уровне.

Это нам не нравилось. Это было некрасиво. Это было неправильно. И система классов решила эту проблему, потому что позволяла написать `f(x,y) = x + y + 1` раз и навсегда. Она получает тип `Num a => a -> a -> a`, и это работает с любыми числовыми типами, включая те, которые вы еще не создали!

Они должны быть экземплярами `Num`, но прелесть в том, что тип можно придумать и позже: через 10 лет после того, как будет принят стандарт Haskell, определен класс `Num` и написана функция `f`. Сделайте его экземпляром `Num`, и ваша старая функция станет работать с ним.

Вот тут и возникла эйфория. Мы решили проблему, которая казалась нерешаемой. Решение предложили Филип Уодлер и Стивен Блотт (Stephen Blott).

При этом была решена и проблема равенства. ML решает проблему равенства иначе. Если определить член с типом `member :: [a] -> Bool`, который будет спрашивать, входит ли значение в список, то в такой операции нужно проверять равенство значений типа `a`. Возможное решение: заявить, что все значения поддерживают равенство, но нам это не нравится. Нет разумного способа сравнить функции на равенство.

ML скажет вам: «О, мы дадим вам специальную переменную для типов: `'a`. У члена будет тип `member :: a -> ['a] -> Bool`. Эта `'a` называется переменной равенства типов (equality-type variable). Она охватывает только те типы, которые допускают проверку равенства. Теперь можно применить член к целому или символу, но не к функции, потому что тогда `'a` будет заменена функцией, а это не допускается.

Это решение ML отличается от перегрузки числовых данных, но тоже пронизывает всю систему типов, так что эти `'a` всюду участвуют в описании. При этом совершенно особым образом решается узкая проблема проверки на равенство, но для упорядочения это ничего не дает. Что делать, если нужно сортировать список? Тут уже не просто равенство, а упорядоченность. Классы типов решают и эту проблему. В Haskell вы пишете

```
member :: Eq a => a -> [a] -> Bool
sort :: Ord a => [a] -> [a]
```

тем самым точно указав, какие свойства должны быть у типа `a` (равенства или порядка соответственно).

Вот мы и «балдели», потому что с помощью единственного мощного механизма системного уровня решили несколько проблем, для которых предлагались только узкие и различающиеся решения. Одним ключом открыли кучу дверей, когда прежние ключи и к отдельным дверям плохо подходили.

Филип Уодлер: Классы типов интересны еще тем, что они оказали влияние на механизм генериков в Java. Метод, описываемый в Java как:

```
public static <T extends Comparable<T>> T min (T x, T y) {
      if (x.compare(y) < 0)
    x;
      else
    y;
}
```

очень похож на соответствующий в Haskell:

```
min :: Ord a => a -> a -> a
min x y = if x < y then x else y
```

Только второй короче. В целом, в Java определение переменной интерфейса как расширения некоторого интерфейса (обычно параметризованного по переменной того же типа) играет ту же роль, что и определение в Haskell переменной как принадлежащей некоторому классу типов.

Не сомневаюсь, что здесь было прямое влияние, потому что участвовал (как и Мартин Одерски, Гилад Браха и многие другие) в разработке ге-

нериков для Java. Думаю, генерики в C#, в свою очередь, появились под влиянием этого решения, но поскольку над ними я не работал, то сказать наверное не могу. Новая идея «концептов» в C++ также очень близка, а в статьях о них в качестве сравнения приводятся классы типов Haskell.

В каких случаях, как вам кажется, программисты на Haskell могут оценить его строгую типизацию?

Саймон: Система типов Haskell достаточно мощна, чтобы способствовать выразительности разработки.

Контроль типов – не только способ избежать глупых ошибок вроде 5+True. Он создает дополнительный уровень абстракции для описания и обсуждения конструкции и архитектуры программы, и там, где ОО-программисты рисуют диаграммы UML, программисты на Haskell пишут определения типов (а программисты на ML пишут сигнатуры модулей). И это гораздо лучше, потому что обеспечивается точность и возможность машинного контроля.

Филип: Вот одна старая история.[1] Software AG выпустила коммерческую базу данных под названием Natural Expert, в которой манипуляции с данными осуществлялись с помощью похожего на Haskell функционального языка собственной разработки. Были организованы недельные курсы обучения. В начале курса разработчики часто жаловались, что контроль типов находит ужасно много ошибок. К концу курса обнаруживалось, что большинство написанных ими программ отлично работают, как только им удается пройти контроль типов. Таким образом, благодаря типам осуществлялась вся необходимая отладка. Короче, если в начале недели они считали систему типов врагом, то в конце уже воспринимали ее как своего помощника.

Я не хочу этим сказать, что *любая* написанная вами программа будет работать, как только вы проведете ее через контроль типов. Но типы отлавливают огромное количество ошибок и делают отладку гораздо проще.

Значение типов особенно возрастает при использовании более сложных возможностей языка. Например, значительно облегчается применение функций высшего порядка. Полиморфные функции хранят в своем типе огромное количество информации. Например, если у вас есть нечто вроде:

```
l :: (Int -> Int) -> [Int] -> [Int]
```

[1] Hutchison, Nigel et al. «Natural Expert: a commercial functional programming environment», Journal of Functional Programming 7(2), March 1997.

(принять функцию целого с булевым значением и список целых, возвратить список целых), то это может быть что угодно, но если тип такой:

```
m :: forall a b. (a -> b) -> [a] -> [b]
```

(для любых типов `a` и `b`, принять функцию из `a` в `b` и список `a`, возвратить список `b`), то вам становится известным очень многое. Фактически, сам тип представляет собой теорему,[1] которой удовлетворяет функция, и по этому типу можно доказать, что:

```
m f xs = map f (m id xs) = m id (map f xs)
```

где `map` применяет функцию к каждому элементу списка, создавая новый список, а `id` – тождественное отображение. Чаще всего само `m` будет `map`, поэтому `(m id)` окажется тождеством. Но `m` может также переставлять элементы, например обратить порядок элементов входного списка, а потом применить функцию, или применить функцию, а затем взять каждый второй элемент результата. Но этим все и ограничится. Типы гарантируют, что для получения элемента выходного списка *обязательно* применить функцию к элементу входного списка, и *нельзя* по значению элемента определять, что с ним делать, только указать его место в списке.

Для меня самое потрясающее в системах типов – их тесная связь с логикой. Существует глубокое и красивое свойство, называемое «выражения как типы», или изоморфизм Карри–Говарда, в соответствии с которым всякая программа подобна доказательству теоремы, тип программы подобен утверждению теоремы, которую доказывает программа, а выполнение программы подобно упрощению доказательства. Три базовых типа структур данных – записи, варианты и функции – в точности соответствуют трем базовым конструкциям в логике – конъюнкции, дизъюнкции и импликации.[2]

Оказывается, то же самое верно для самых разных логических систем и программ, а значит, это не случайное совпадение, а глубокий и важный принцип, который может быть применен при проектировании типизированных языков программирования. Действительно, вот вам рецепт проектирования: придумайте тип, добавьте в язык конструкторы для создания значений этого типа и деконструкторы для разборки таких значений на части, соблюдая закон, согласно которому, если вы что-то построили, а потом разобрали, то получите то, с чего начали (*бета-редукция*), а если разберете что-то на части, а потом снова собере-

[1] Wadler, Philip. «Theorems for Free», 4th International Conference on Functional Programming and Computer Architecture, London, 1989.

[2] Wadler, Philip. «New Languages, Old Logic», Dr. Dobb's Journal, December 2000.

те, то тоже получите то, с чего начали (*эта-преобразование*). Невероятная мощь и красота. Очень часто, когда что-то проектируешь, кажется, что можно сделать его произвольным образом, что есть штук пять разных подходов, и непонятно, какой из них лучше. Но теория говорит, что у функциональных языков есть основа, которая вовсе не произвольна.

Сейчас мы как раз приходим к тому моменту, когда обычной практикой среди специалистов в компьютерных науках становится ввести в компьютер свое доказательство, чтобы он мог проверить его правильность, и эта процедура основывается на тех же самых принципах и системах типов, что и функциональные языки, поскольку существует глубокая связь между программами и доказательствами, между типами и теоремами. Вот мы и видим, как начинают соединяться разные вещи, как типы позволяют все лучше описывать поведение программ, а компилятор получает возможность гарантировать все больше свойств ваших программ, и постепенно доказательство свойств вашей программы по мере ее написания будет становиться все более обычным делом. Правительство США иногда требует, чтобы свойства военного ПО, относящиеся к его надежности, были доказаны.

Эта тенденция продолжится. В настоящее время операционные системы дают не слишком надежные гарантии относительно своей безопасности, но я думаю, что такое положение изменится, и очень важную роль в этом сыграют системы типов.

Можно ли «ленивость» вычислений перенести в другие языки программирования или она больше пригодна для Haskell благодаря некоторым другим его характеристикам?

Джон: Ленивость приводит к сложной и непредсказуемой управляющей логике. В Haskell это не вызывает проблем, поскольку порядок вычислений не может повлиять на результат – управляющая логика может быть сколь угодно сложной, что не влияет на объем труда по доводке кода до рабочего состояния. Ленивость можно организовать и в других языках, что и было сделано, и смоделировать ее тоже несложно. Но сочетание ленивости с побочными эффектами оказывается смертельным. Заставить такой код работать практически невозможно, потому что нет никаких надежд понять, почему побочные эффекты возникают в том или ином порядке. Я проводил такие эксперименты в Erlang, моделируя ленивость в коде, который использовал библиотеку с побочными эффектами в интерфейсе. В итоге я смог добиться от этого кода того, что мне было нужно, только построив чисто функциональный интерфейс к библиотеке поверх того, который давал побочные эффекты, так что мой ленивый код избавился от побочных эффектов.

Поэтому, как мне кажется, ответ на ваш вопрос должен быть утвердительным: да, ленивость можно экспортировать в другие языки, но

программистам при ее использовании нужно обеспечить отсутствие побочных эффектов в соответствующей части кода. Хорошим примером, конечно, служит LINQ.

Есть ли у Haskell еще характеристики, которые можно было бы перенять другим языкам, чтобы повысить свою эффективность и безопасность?

Филип: Ряд характеристик Haskell уже включен или внедряется в некоторые основные языки.

Функциональные замыкания (лямбда-выражения) появились во многих языках, включая Perl, JavaScript, Python, C#, Visual Basic и Scala. Внутренние классы появились в Java как средство имитации замыканий, а предложение добавить в Java настоящие замыкания (как в Scala) широко обсуждается. Влияние в пользу замыканий оказывает не только Haskell, но и все функциональные языки, включая Scheme и семейство ML.

Списочные выражения есть в Python, C# и Visual Basic (в обоих случаях в связи с LINQ), и Scala, а также запланированы для Perl и JavaScript. Haskell не был первым языком со списочными выражениями, но весьма способствовал росту их популярности. В C#, Visual Basic и Scala выражения применимы не только к спискам, но и другим структурам, поэтому они больше напоминают генерацию монад или do-синтаксис, которые впервые появились в Haskell.

Обобщенные типы в Java появились под сильным влиянием полиморфных типов и классов типов Haskell: я участвовал в разработке Java-генериков и написании книги о них, опубликованной O'Reilly.[1] Эти возможности Java, в свою очередь, способствовали их появлению в C# и Visual Basic.

Классы типов есть и в Scala. Теперь и в C++ рассматривают возможность включения конструкции, названной «концептами» и весьма похожей на классы типов. Haskell также оказал влияние на ряд менее распространенных языков, в том числе Cayenne, Clean, Mercury, Curry, Escher, Hal и Isabelle.

Джон: Добавлю еще анонимные делегаты в C# и списочные выражения в Python. Идеи функционального программирования проявляют себя повсюду.

Пол: Я прочел много сообщений от тех, кто изучал Haskell, но редко использует его в своей работе; при этом они отмечали, что это изучение положительно повлияло на их мышление и стиль программирования

[1] Naftalin, Maurice and Philip Wadler. Java Generics and Collections (O'Reilly, 2000).

на процедурных языках. Влияние Haskell на основные языки программирования и те, которые появились недавно, – огромно. Значит, наша работа не напрасна, если мы сумели повлиять на мэйнстрим, даже если сами в него не попали.

Как конструкция языка влияет на конструкцию программ на этом языке?

Саймон: Язык, на котором пишется программа, оказывает глубокое влияние на ее конструкцию. Например, в мире ООП эскиз программы часто создается с помощью UML. В Haskell или ML вместо этого пишут сигнатуры типов. Начальный этап разработки программы на функциональном языке во многом состоит из определения типов. Однако, в отличие от UML, результат является составной частью конечного продукта и подвергается тщательной машинной проверке.

Определения типов также очень хорошо подходят для регистрации инвариантов, например «этот список не может быть пуст». В настоящее время такие условия не подвергаются машинной проверке, но, думаю, это впереди.

Жесткие типы меняют характер сопровождения программы. Можно изменить тип данных и быть уверенным, что компилятор укажет вам все места, которые придется модифицировать. Для меня это одна из главных причин, по которым стоит иметь выразительные типы: я не верю, что можно внести существенные изменения в большую программу с динамическими типами и при этом рассчитывать на сравнимую степень надежности.

Функциональные языки резко меняют подход к тестированию, о чем выше убедительно сказал Джон.

При работе с функциональным языком возникает сильная склонность к использованию чисто функциональных структур данных вместо структур данных, изменяемых по месту. Это может сильно повлиять на архитектуру программы. На Haskell можно писать процедурные программы, но они выглядят грубо, что побуждает программистов использовать чистый код, где только возможно.

Пол: Мне понравилось то, что сказал Саймон, хотя он в основном рассматривал влияние Haskell (и других функциональных языков) на архитектуру программы. Но есть и другая сторона вопроса: а как программное приложение влияет на архитектуру языка? Haskell и большинство функциональных языков разрабатывались как универсальные, но замечательная особенность программ, создаваемых в последние годы на Haskell, состоит в том, что многие из них основаны на проблемно-ориентированном языке (DSL), «встроенном» в Haskell (мы часто называем их DSEL). Примеров можно привести массу – в графике, ани-

мации, компьютерной музыке, обработке сигналов, синтаксическом анализе, подготовке к печати, финансовых расчетах, робототехнике и многих других областях, а также тьма библиотек, спроектированных по тому же принципу.

Как агент по недвижимости, который говорит, что главными для дома являются три вещи – «место, место и место», так и я повторяю, что три главные вещи в программировании – это «абстракция, абстракция и абстракция». И по моему мнению, правильно спроектированный DSL – это вершина абстракции предметной области: он содержит в себе ровно то количество информации, которое нужно, не больше и не меньше. Haskell удобен тем, что предоставляет средства для быстрого и эффективного создания таких языков. Эта методика не идеальна, но она очень хороша.

Филип: Функциональные языки можно легко расширять. Прекрасными примерами служат Лисп и Scheme – почитайте у Пола Грэма[1], как Лисп выступил секретным оружием при создании одного из первых веб-приложений (ставшего впоследствии продуктом Yahoo!) и, в частности, как макросы Лиспа сыграли ключевую роль в построении этого программного обеспечения. В Haskell тоже есть ряд особенностей, позволяющих легко усилить мощь языка, в том числе лямбда-выражения, ленивость, система монад и (в GHC) шаблон Haskell для метапрограммирования.

Пол уже рассказал, как благодаря таким возможностям Haskell стал излюбленным языком для встраивания предметно-ориентированных языков. Но мы встречаем его и на более низком уровне, когда создают небольшие библиотеки для комбинаторов парсеров или форматирования печати. Если вы действительно хотите оценить мощь функционального программирования, то можете начать с этих двух примеров.

Ленивость Haskell тоже сильно влияет на то, как пишут программы, поскольку позволяет так осуществить декомпозицию задачи, как едва ли возможно иными способами. Я люблю представлять это как преобразование времени в пространство. Например, вместо того чтобы думать о том, как последовательно (время) возвращать значения, я могу вернуть список, содержащий все значения (пространство), и ленивость гарантирует, что значения в списке будут вычисляться по одному, как и требуется.

Работать с пространством часто бывает легче, чем со временем: пространство можно визуализировать непосредственно, тогда как визуализация времени требует анимации. Сравните просмотр расписания мероприятий в течение дня с просмотром видеозаписи событий, про-

[1] Graham, Paul. Hackers & Painters (O'Reilly, 2004).

исшедших в течение дня! Поэтому применение ленивости может существенно изменить подход к задаче. Примером служат упомянутые выше комбинаторы парсеров, которые возвращают список всех возможных результатов анализа: ленивость гарантирует, что этот список будет составляться по мере надобности. В частности, если вас удовлетворит первый результат, остальные и не будут генерироваться.

Расширение (функционального) образования

Что нового вы узнали, преподавая программирование студентам колледжа?

Пол: В течение долгого времени – а возможно, и до сих пор – функциональные языки изучались преимущественно во вводных курсах благодаря легкости изучения и возможности абстрагироваться от многочисленных деталей процедурных вычислений. Сейчас я пришел к тому мнению, что в конечном счете мы совершали ошибку! Дело в том, что студенты приходили к выводу, что функциональные языки – несерьезные языки, поскольку преподают их в начальных курсах и с помощью надуманных примеров. А обнаружив «мощь» побочных эффектов, многие из них уже не возвращаются обратно. Очень жаль.

Мне кажется, что новичкам редко удается оценить достоинства функциональных языков. Они становятся очевидными только после приобретения некоторого опыта программирования.

У нас в Йеле есть курс функционального программирования, который посещают главным образом продвинутые студенты, специализирующиеся в программировании. У меня не возникает проблем, когда на больших и трудных задачах, с применением сложного математического аппарата, я стараюсь продемонстрировать им реальную мощь ФП. Что еще более важно, я могу сказать: «А теперь попробуйте сделать все то же с помощью процедурного программирования», – и мы часто сравниваем код Haskell с кодом Си, что бывает весьма поучительно и чего вы просто не сможете делать со студентами, начинающими изучение языков программирования с Haskell.

Что не так с компьютерными науками и тем, как их преподают? Что нужно изменить?

Пол: Хочу сказать о том, какую цель в деле образования поставил перед собой лично я, – в надежде, что кому-то она покажется интересной или даже поучительной.

Есть буквально сотни книг, которые учат, как нужно программировать вообще или писать программы на каком-то конкретном языке. Обычно в этих книгах есть примеры, взятые из различных источников, но

эти примеры часто весьма неудачны и относятся к таким задачам, как числа Фибоначчи, факториалы, обработка строк и текста, простые головоломки и игры. Мне интересно, можно ли написать книгу, основной темой которой будет не программирование, а что-то другое, но чтобы главным ее инструментом изложения основных понятий был язык программирования.

Вероятно, вы скажете, что такими будут книги, посвященные операционным системам, сетевому взаимодействию, графике или компиляторам, если в них активно использовать язык программирования для изложения материала, но я предполагал темы, не столь тесно связанные с вычислительной наукой. Я имею в виду какие-то другие науки – физику, химию, астрономию, возможно, даже общественные науки – в частности экономические. И мне хотелось бы выяснить, нельзя ли пойти еще дальше и применить тот же подход в преподавании каких-то дисциплин, относящихся к искусству, в частности музыке.

Я предполагаю, что функциональный язык, особенно такой как Haskell, с его сильной поддержкой предметно-ориентированных встроенных языков, мог бы стать отличным средством для преподавания идей, не относящихся к программированию. Программирование отличается тем, что требует точности, а функциональное программирование отличается тем, что эта точность сочетается с краткостью. И то и другое пошло бы на пользу в преподавании многих дисциплин, которые я перечислил.

Саймон: Это имеет отношение к тому, чем я интенсивно занимаюсь в Англии, в частности в школе. Я вхожу в управляющий совет школы – такие советы есть в каждой школе. В настоящее время преподавание компьютерных наук в школе находится на жалком уровне. Фактически, компьютерные науки там отсутствуют – все это лишь информационные технологии.

Под «информационными технологиями» я понимаю электронные таблицы, базы данных и так далее. Это все равно, что сказать: «Вот это – автомобиль, а вот так его водят». Вот так пользуются электронными таблицами. «А теперь, когда вы научились водить, поговорим о том, куда можно поехать. Хотите в Бирмингем? Тогда можно выбрать вот такой маршрут и взять с собой таких-то пассажиров».

Вы занимаетесь планированием проекта, анализом требований, системной интеграцией и тому подобным. В настоящее время в школе вам не рассказывают о том, что находится под капотом. В какой-то мере это объяснимо, потому что машину должен уметь водить каждый, верно? Кроме того, нужно иметь какое-то представление о том, куда можно поехать на ней и как не давить пешеходов по дороге.

Не обязательно, чтобы всех интересовало, как устроен автомобиль. Вполне допустимо, чтобы большинство могли только водить машину, но должны быть и те, кто хочет знать, как автомобиль устроен. Должны быть какие-то дисциплины, относящиеся к компьютерным наукам, которые нужно преподавать в школе, – по крайней мере тем, кто ими заинтересуется. А сейчас им говорят: «Вот что нужно знать о компьютерах», фактически отталкивая их, потому что это скучно.

Я член группы, которая работает в Великобритании, пытаясь помочь учителям в обучении компьютерным наукам на школьном уровне. Разумеется, в школе второй ступени, которая в этой стране соответствует возрасту 11–16 лет. На следующей ступени А уже есть настоящие компьютерные науки, но это относится к возрасту 16–18 лет, когда они уже утратили интерес.

Число изучающих компьютерные науки в старших классах падает, еще меньше их среди поступающих в университеты – как и в США. Отчасти это связано с тем, что у каждого подростка сейчас есть компьютер, поэтому все, что относится к ИТ, им уже знакомо. Когда их учат этому, многократно в разных курсах, они просто начинают думать: какая скука – что тут может быть интересного?

Думаю, это главная ошибка преподавания компьютерных наук в школе. Многим из тех, кого не интересуют технологии, достаточно научиться «водить». Такое обучение должно быть умеренным и интегрироваться с другими предметами. Это полезный инструмент, только и всего. Не стоит особого внимания. Но ведь учим же мы детей физике, которой реально потом будет заниматься очень малая их часть? Большинству из них не нужно и неинтересно будет знать что-либо о коэффициенте расширения. Точно так же, мне кажется, надо дать некоторое представление о предмете компьютерных наук, которым они могли бы увлечься, потому что это так интересно.

Формализм и развитие

Насколько ценно определение формальной семантики языка?

Саймон: Формальная семантика поддерживает все, что нами сделано в Haskell. Например, если вы посмотрите мои статьи, то увидите, что в большинстве статей есть некий формализм, с помощью которого я пытаюсь вести изложение. Даже для такой обязательной вещи, как транзакционная память, в соответствующей статье была формальная семантика, которая объясняла, что такое транзакции.

Формальная семантика – это прекрасный способ схватить некоторую идею, зафиксировать какие-то детали и заглянуть в некоторые мудре-

ные уголки. Но в реальном языке, где все работает вместе, фактически формализовать все, что касается языка, – непосильная задача. Снимаю шляпу перед авторами описания стандартного ML, потому что это исключительная работа. Пожалуй, это единственный язык, у которого есть почти полное формальное описание.

Думаю, с точки зрения выгоды нужность полноты сомнительна. Последние 10%, которые превращают набор формальных фрагментов, описывающих разные аспекты языка, в законченное формальное описание языка в целом, обходятся очень дорого. Это большой труд. Возможно, процентов 70 от всей работы. И много ли выгоды принесут вам эти последние 70% работы? Вероятно, 20% или около того. Я не вполне понимаю, почему так происходит. Судя по всему, соотношение затраты/результат резко возрастает, как только вы начинаете формализовать язык в целом, а не отдельные его части. Так бывает даже в самый первый раз.

Затем возникает вопрос: а что будет, если язык развивается? Мы продолжаем вносить изменения в Haskell. Если мне нужно формализовать все, что касается вносимой модификации, это существенно осложняет ее, что фактически и произошло с ML. Модифицировать ML очень тяжело именно потому, что у него есть формальное описание.

Вероятно, формализм может мешать нововведениям. Он будет стимулом для инновации, потому что позволяет лучше разобраться в ее сути, но он будет тормозить ее, если условия требуют полной формализации всего языка.

Возможно, есть разумная середина, эдакий полуформализм, вроде спортивного пиджака с джинсами?

Саймон: Думаю, Haskell как раз занял такое положение. Определение языка практически целиком написано на английском языке, но если почитать соответствующие научно-исследовательские работы, то в них обнаружится множество формализмов для отдельных фрагментов языка. Haskell не кодифицирован в докладе – во всяком случае, не в виде полного описания. Для языка, у которого нет формального описания, вы обнаружите весьма много формализованного материала, больше чем для C++, который очень неформален, хотя на это неформальное описание были потрачены огромные усилия.

Эта пропорция интересна. Я нимало не сомневаюсь, что формализм имел огромное значение для сохранения чистоты Haskell. Мы не вводили новшества не глядя. Каждое из них должно было вписываться в целое и соответствовать принципам. Вы получаете великолепную возможность сказать: «Как-то это выглядит неприятно, неужели нельзя сделать иначе?» Если нечто выглядит неприятно, то весьма вероятно,

что его будет трудно реализовать, а программисту будет трудно разобраться с вашей реализацией.

Филип: Первую статью о классах типов написали мы со Стивеном Блоттом, она напечатана в трудах «Симпозиума по принципам программирования» в 1989 году.[1] В ней было формализовано ядро классов типов, и мы старались писать как можно проще и короче. Впоследствии Корди Холл, Кевин Хэммонд, Саймон и я попытались описать значительно более полную модель.[2] Это описание появилось в ESOP в 1994 году – как видите, потребовалось пять лет, чтобы добраться до этого! Мы не формализовали Haskell полностью, но попытались формализовать все детали классов типов. Таким образом, для разных задач есть разные уровни моделирования.

Статья в ESOP дала непосредственную модель для реализации в GHC, в особенности для использования лямбда-исчисления высшего порядка в качестве промежуточного языка, что сейчас имеет важнейшее значение для GHC. В этом состоит замечательное свойство формализации: для ее проведения нужно потратить много труда, но когда она готова, реализация значительно облегчается. Часто бывает, что реализация вызывает затруднения, но после формализации реализация значительно упрощается.

Другой пример формализации – Featherweight Java, которую мы разрабатывали с Ацуси Игараси и Бенджамином Пирсом и опубликовали в OOPSLA в 1999 году (и перепечатали в TOPLAS в 2001 году).[3] В то время многие публиковали формальные модели Java, стараясь сделать их как можно более полными. Мы же со своей Featherweight Java, напротив, попытались создать как можно более простую модель, ограничившись минимальным синтаксисом и одной страничкой правил. И оказалось, что мы не ошиблись, ибо модель была настолько проста, что ею было удобно пользоваться тем, кто хотел лишь добавить какую-нибудь одну функцию и сделать модель. В результате эту статью часто цитировали.

С другой стороны, обнаружилась ошибка в начальной конструкции генериков, связанная с присваиванием и массивами, а мы ее не заметили, потому что не включили в Featherweight Java ни присваивание, ни мас-

---

[1] Wadler, Philip and Stephen Blott. «How to make ad-hoc polymorphism less ad hoc», 16th Symposium on Principles of Programming Languages, Austin, Texas: ACM Press (January 1989).

[2] Hall, Cordelia et al. «Type classes in Haskell», European Symposium On Programming, LNCS 788, Springer Verlag: 241–256 (April 1994).

[3] Igarashi, Atsushi et al. «Featherweight Java: A minimal core calculus for Java and GJ», TOPLAS, 23(3):396–450 (May 2001).

сивы. Это показывает, что нужно выбирать между простой моделью, которую легко понять, и более полной, которая позволит обнаружить большее количество ошибок. И та, и другая имеют ценность!

Я также занимался формализацией части определения XQuery – языка запросов для XML, являющегося стандартом W3C.[1] Конечно, в комиссиях по стандартизации приходится много спорить; в нашем случае было много вопросов вроде «что за штука эта формализация? как ее нужно читать?». Было противодействие тому, чтобы делать формализацию каноническим стандартом: члены комиссии хотели, чтобы каноном была обычная английская речь, потому что считалось, что разработчикам так легче читать. Но некоторые части системы типов было проще записать в виде формализации, а не обычной речи, и было решено, что для этих частей канонической станет формальная спецификация.

В какой-то момент было предложено внести изменения в архитектуру. И примечательно, что несмотря на все прежнее недовольство комиссия предложила тем, кто занимался у нас формализацией, формализовать это изменение. Занявшись этим, мы обнаружили, что хотя предложение о модификации, написанное на обычном языке, казалось точным, в 10 местах мы не знали, как его формализовать, потому что обычная речь допускала неоднозначное толкование. Разрешив эти проблемы, мы представили формальную спецификацию. После доклада этой формализации на очередном совещании модификация была принята единогласно без всякого обсуждения, чего до того момента еще не случалось на совещаниях по стандартизации. Так что в этом случае использование формализма привело к большому успеху.

Как заметил Саймон в отношении Haskell, обычно затраты на полную формализацию не оправдывают себя. Что касается XQuery, то мы формализовали его примерно на 80%, а оставшиеся 20% хотя и были важны, но их формализация потребовала бы такого огромного объема работы, что мы отказались от нее.

Тем не менее, думаю, проведенная формализация имела большую ценность.

Добавлю, что эта формализация легла в основу реализации Galax (осуществленной моими коллегами Мэри Фернандес и Джеромом Симеоном), которая сейчас является одной из основных реализаций XQuery. Здесь мы еще раз убеждаемся в том, как формализация облегчает реализацию.

---

[1] Simeon, Jerome and Philip Wadler. «The Essence of XML», Preliminary version: POPL 2003, New Orleans (January 2003).

Все знакомые мне математики говорят, что если математическая теория лишена красоты, то она вполне может быть ошибочной.

Саймон: Это так. Вот вам пример: мы сейчас заняты тем, что вводим в Haskell функции уровня типа и усиленно стараемся выработать для этого формализм. В этом году у нас об этом выходит в ICFP статья, но я не испытываю в связи с этим полного удовлетворения. Так что мы это бросим. Для реализации это имеет прямые последствия. Можно слепать какую-нибудь реализацию и сказать: вот что у нас есть, попробуйте с этим поработать. Вполне возможно, что на следующий день к нам обратятся со словами: «Вот программа, которая, как я думал, будет проверять типы, но она этого не делает – так и должно быть?» И нам придется отвечать: «Видите ли, данная реализация не проверяет типы, так что, может быть, это правильно. Но спросить – ваше право».

Я не страдаю от того, что мы не формализовали язык полностью. Но это не значит, что от этого не было бы пользы. Последние 70% усилий некоторую пользу приносят. Возможно, соотношение затраты/прибыль не столь велико, но прибыль имеется. Возможно, существует взаимозависимость между характеристиками языка, в которой вы не разобрались. Вы формализовали отдельные стороны, но не предвидели, что ваш хитрый план A и удачное изобретение B окажутся взаимоисключающими. Всем нам иногда грозит такая опасность.

Если сообщество пользователей вашего языка велико, они неизбежно столкнутся с такой ошибкой и сообщат о ней.

Саймон: Верно, и тогда вы смущенно признаетесь: ну да, пожалуй, – вот если бы мы дальше продвинулись с формализацией языка, то такого, вероятно, не случилось бы. Это очень важно. Но, в конце концов, мы сознательно не пошли тем же путем, что ML.

Есть ли у вас технология решения проблем обратной совместимости, когда такое происходит?

Саймон: Видимо, мы все еще вырабатываем ее, хотя в прошлом обычно игнорировали такие проблемы. Сегодня это не совсем так. Примерно 10 лет назад мы решили, что язык, который мы назвали Haskell 98, станет в некотором роде стабильным подъязыком. Мы решили, что впредь не будем его изменять. Компиляторы Haskell по умолчанию принимают Haskell 98. Если вы дадите им не Haskell 98, а что-то другое, нужно задать флаги, которые скажут: принять такое-то расширение.

Был такой флаг, который говорил «принять все», но сейчас он разделился на три десятка отдельных расширений. Тот прежний флаг включает около 15 из них. Взглянув на модуль исходного кода, вы увидите, какие расширения языка фактически используются в нем. В сущ-

ности, мы предлагаем программисту сообщить, которым из языков он пользуется.

Обычно накладывается ограничение, чтобы старые программы продолжали работать, хотя это не всегда так. В некоторых расширениях появляются новые ключевые слова, например `forall`. В Haskell 98 `forall` могло быть переменной типа в типе, но если разрешить типы высшего порядка, `forall` становится ключевым словом, и переменная типа с именем `forall` становится недопустимой.

Но так бывает очень редко. Обычно расширения обеспечивают совместимость вверх. Но уверяю вас, что если подключить достаточное число расширений, наверняка найдутся программы Haskell 98, которые перестанут работать.

Ждет ли нас момент, когда какой-нибудь Haskell 2009 или 2010 кодифицирует все это в виде нового стандарта?

Саймон: Да. Идет процесс под названием Haskell-прим, где «прим» означает штрих рядом с именем переменной, – в основном из-за того, что мы пока не решили, как его назвать. Вначале мы предполагали, что некая группа людей после публичного обсуждения предложит новый язык, своего рода стандарт, который можно будет объявить как Haskell 2010, а не так, как было с Haskell 98. Для того чтобы это произошло, нужно собрать достаточно много готовых потрудиться людей, что непросто.

Мне кажется, это оборотная сторона прежнего успеха. GHC, будучи наиболее широко используемым компилятором Haskell, превратился в некоторой степени в стандарт де-факто. Поэтому на практике не возникает особых трудностей из-за языковых несовместимостей между различными компиляторами. Не думаю, что это языку на пользу, но в результате снижаются побудительные мотивы для того, чтобы люди тратили свое самое дорогое достояние, время, на кодифицирование стандарта языка.

Могут ли возникнуть соперничающие между собой реализации, точно следующие стандарту языка GHC?

Саймон: Конкурирующие реализации уже есть, и они обычно более узко специализированы в определенных областях. На самом деле, совсем недавно, на конференции ICFP по функциональному программированию, мы сменили направления своей работы. Вместо того чтобы создавать единый монолит, каким является Haskell Prime, мы попробуем кодифицировать расширения языка. Не определяя их в GHC, мы предложим желающим высказаться по поводу того, какие расширения языка должны быть, по их мнению, кодифицированы, проведем недолгое обсуждение и найдем человека или небольшую группу людей,

которые напишут нечто вроде приложения к отчету со словами: «Это самостоятельное описание того, что должно делать это расширение».

После этого мы сможем заявить, что Haskell 2010 – это набор вот таких взаимно согласованных расширений. Фактически мы можем продолжить работу, сначала кодифицировав расширения и дав им имена, а потом объединив их в группу с особым названием, что будет напоминать «расширения Glasgow», но сделанные более последовательно.

Мы рассчитываем, что в смысле конструкции языка это напоминает работу сообщества open source, когда оно выпускает новую версию GNOME, Linux или чего-то еще. Незаметно для окружающих идет огромная работа, пока кто-нибудь наконец не обернет это все куском клейкой ленты и не объявит: «Все эти штуки работают друг с другом, и данный комплект называется GNOME 2.9».

Свободно отобранные результаты, достигнутые на базе общей философии и перевязанные красивым бантиком.

Саймон: Верно, вместе с обещанием, что они совместимы между собой. Это то, чем мы занимаемся по части языка. Язык почти полностью определяется реализацией, поэтому данная процедура уже весьма упорядочена. Если хотите, отсутствие побудительных мотивов объясняется чрезмерной упорядоченностью.

Что касается библиотек, ситуация полностью противоположна. Разработчиков библиотек очень много. Вы слышали о Hackage? Там чуть ли не каждый день появляется новая библиотека. Сейчас их там 700 с лишним. В результате очень трудно сказать, действительно ли работает какая-то из них и совместима ли она с другими. Это серьезная проблема для рядового пользователя, который пытается работать с Haskell.

В данное время моя цель как разработчика компилятора заключается в том, чтобы устраниться от разработки и сопровождения библиотек. Пусть другая группа людей занимается тем, что мы только что описали, но в отношении библиотек. Они собираются назвать это «платформой Haskell». По существу, это будет набор кодифицированных библиотек. Мне кажется, что это весьма обычное дело. Платформа Haskell фактически станет метабиблиотекой, зависящей от конкретных версий десятков других библиотек. Она как бы говорит: «Платформа Haskell предоставляет вам ряд библиотек, которые отмечены знаком качества и в известной мере взаимно совместимы».

Несовместимость может заключаться в том, что какие-то две библиотеки окажутся зависимыми от двух разных версий одной и той же общей базовой библиотеки. Если вы соедините их вместе, то у вас возникнет два экземпляра базовой библиотеки, что бывает нежелательно. Если

в базовой библиотеке определен некий тип, то два экземпляра этой библиотеки могут создавать разные версии этого типа, которые окажутся несовместимыми. То, на что вы рассчитывали, может не сработать. Не просто какой-то тип не будет работать, а возникнет озадачивающее сообщение об ошибке вроде «T из модуля M в пакете P1 версии 8 не соответствует T из модуля M в пакете P1 версии 9».

Видимо, это пространный ответ на ваш вопрос относительно обратной совместимости. Мы стали подходить к нему гораздо более серьезно. У нас старая проблема: при выпуске новой версии GHC компилятор оказывается довольно тесно привязан к базовому пакету библиотек.

Все также зависят от Prelude.

Саймон: Да, потому что Prelude очень удобна. Там есть много полезных функций. Говоря «тесно привязан», я имею в виду, что компилятор знает точную реализацию `map`, знает ее имя и где она определена. Есть библиотеки, у которых очень глубокая связь с GHC.

Это сделано в помощь компиляции?

Саймон: Да. Дело в том, что если компилятору нужно выдать код, который вызывает библиотечные функции, ему нужно знать, что такие функции существуют, и знать их типы. Это знание оказывается запаянным в компилятор. Можно представлять себе это так.

Из этого следует, что если изменить интерфейс к базовому пакету, как это случается при переходе от одной версии к другой, то в новые версии приходится включать некую прокладку для нового базового пакета, которая предоставит тот же API, какой был у прежнего базового пакета, чтобы при необходимости можно было не обращать внимание на модификации. Все это трудоемкая работа по обеспечению обратной совместимости, которой нам не приходилось заниматься раньше. Но это проблема не следования лозунгу «избегать успеха любой ценой».

У популярности особые последствия.

Саймон: Это правда, но в каком-то смысле это приятные проблемы.

Какие уроки из опыта изобретения, дальнейшего развития и распространения вашего языка могли бы извлечь разработчики компьютерных систем в настоящем и обозримом будущем?

Саймон: Функциональное программирование – это лаборатория, где исследуется масса интересных идей. Благодаря простоте начальных условий мы можем гораздо глубже изучить перспективные идеи (пример: системы типов, обобщенное программирование, реактивное программирование, продолжения). Так что, заглянув в лабораторию ФП,

вы обнаружите массу интересного. Оно может не быть вам интересно *непосредственно*, но будущее формируется там.

Пол: Для меня самый любопытный урок заключается в том, что стремление к идеалу – в случае разработки Haskell это чистота – может окупиться сторицей. Правильное решение может обнаружиться не сразу, и отдачи придется ждать еще дольше, но она наступит. Если не заглядывать далеко, то можно быстрее получить прибыль, но, поступившись своим принципами, вы в итоге проиграете.

9

ML

ML представляет собой универсальный функциональный язык, который Робин Милнер и возглавляемая им команда разработали в Эдинбургском университете в 1970-х. Он возник из металингвистического проекта, целью которого было описание математических доказательств. Возможно, самым ценным вкладом ML в проектирование языков стал алгоритм Хиндли–Милнера для вывода типов, применяемый во многих системах со статическими латентными типами. Этот язык послужил прообразом для Standard ML, Caml, Haskell, F# и других.

Доказательство теорем

Вы создали LCF, один из первых инструментов для автоматического доказательства теорем, и язык программирования ML для поддержки доказательств. Как это было?

Робин Милнер: В 1970-х годах было несколько проектов по автоматизированному доказательству. Им были свойственны крайности: в одних совершенно самостоятельно шел поиск доказательства (например, с применением знаменитого принципа разрешения Робинсона), в других работа шла совершенно не самостоятельно, то есть всего лишь проверялась логическая правильность каждого шага, сделанного человеком (как в системе автоматов де Брейна). Кстати, оба подхода вносят существенный вклад в современную технологию доказательств.

Я искал нечто промежуточное, чтобы человек мог разрабатывать тактику (или стратегию на основе мелких тактик) и вводить в машину ее и то, что подлежало доказательству. Предполагался диалог: если какая-то тактика оказывалась полностью или частично непригодной, машина сообщала бы об этом, и человек мог задать ей другую. Главное, что какими бы авантюрными ни были тактики, машина могла сообщить об успехе только в случае обнаружения реального доказательства. На самом деле, в случае успеха машина должна была экспортировать доказательство, чтобы его могла проверить другая независимая программа (полностью несамостоятельного типа).

Эта система работала главным образом благодаря тому, что в ML – метаязыке для описания пользователем своей тактики – была система типов (в известной мере, новинка), заставлявшая язык решительно отвергать претензии на успех, если только (вследствие удачной тактики, придуманной пользователем) он не мог во всех подробностях описать доказательство, для которого тактика делала всего лишь эскиз. Таким образом, с помощью ML осуществлялось сотрудничество между оптимистичным человеком и дотошной машиной.

Каковы границы возможностей LCF?

Робин: Каких-либо очевидных пределов я не вижу. В настоящее время таким системам, как HOL, или COQ, или Isabelle, предлагают очень смелую тактику, а решаемые задачи неуклонно усложняются. HOL фактически получила доказательство того, что система типов ML была верной, – это как подтвердить то, что репродуктивная система ваших родителей в порядке. Но мы далеки от того, чтобы в качестве тактики использовать идеи, возникающие у математиков. Предвижу, что большие успехи будут появляться в основном в результате сооружения башен, когда простые теоремы служат основанием для вывода более сложных, как это и происходит с большинством математических теорий.

Что касается доказательства работоспособности программ, то оно уже вполне осуществимо, если пользователь снабдит программу операторами проверки (assertions), введенными Флойдом и Хоаром еще в начале 1970-х, которые говорят что-то вроде «когда выполнение достигнет *этой* точки, между переменными программы должны выполняться *следующие* соотношения».

Можно ли таким методом анализировать исходный код программы, чтобы проверить его на отсутствие ошибок?

Робин: Да! И это часто делают, особенно для маленьких, но важных программ, встраиваемых в реальные продукты, например в тормозную систему автомобиля. Наибольшие проблемы возникают, если люди не могут (или не хотят) строго сформулировать нужное свойство!

Насколько трудно определить свойства в строгом виде?

Робин: На самом деле, это вопрос к тем, кто занимается логикой, с помощью которой можно описать эти свойства. Роль ML не в том, чтобы предоставлять такую логику, а в том, чтобы служить средством, позволяющим описывать по такой логике доказательства, а также эвристические алгоритмы для поиска этих доказательств. Так что ML предоставляет базу для такой логики. Первой логикой, базу которой составил ML, была LCF, логика вычислимых функций, которой мы обязаны Дане Скотту (Dana Scott). В этой логике ML (и его предшественник) использовались для поиска и/или доказательства некоторых теорем. Счастлив сообщить, что одной из этих теорем было (почти законченное) доказательство корректности компилятора одного очень простого исходного языка в другой очень простой целевой язык.

Переносимо ли это в другие языки программирования?

Робин: Объяснив роль ML как базы для логики, думаю, смогу ответить и на другой вопрос, очень близкий к заданному: могут ли другие языки стать столь же хорошей базой для логики доказательства? Уверен, что да, если они обладают богатой и гибкой системой типов и поддерживают функции высшего порядка и возможности императивного программирования. ML повезло, что он стал базой для ряда успешных разработчиков логики, а это значит, что ему по-прежнему отдают предпочтение.

Почему для того, чтобы стать хорошим основанием тому, что вы называете логикой доказательства, языку нужны функции высшего порядка?

Робин: Правила вывода реализуются как функции из теорем в теоремы. Это будет первый порядок. Теоремы по сути являются предложениями, поэтому правила вывода по сути являются функциями из строк в строки. Мы придумали штуку, названную тактикой: вы описываете цель – ваше предложение, которое хотите доказать, – а вам возвраща-

ют набор подзадач и функцию, которая при наличии доказательств этих подзадач выдаст доказательство конечной задачи. Таким образом, тактика является функцией второго порядка.

У нас было несколько таких тактик. Мы запрограммировали их, а затем нам потребовалось нечто соединяющее их для образования более крупной тактики. Так у нас появились функции третьего порядка, которые мы назвали *tacticals*, принимающие одни тактики и возвращающие другие, более крупные. Такая тактика могла предложить: «сначала уберите все импликации, сведя их в список допущений, потом примените правило индукции, а потом примените правила упрощения». Это была достаточно сложная тактика, которую мы назвали *стратегией*, и она опровергла несколько теорем.

Работая в Стэнфорде у Джона Маккарти, я сказал ему, что сделал коечто интересное: у меня есть стратегия, есть тактика и есть свойство строк, которое ею доказывается. Я просто описываю цель, которая представляет собой некоторый факт, относящийся к строкам, затем применяю свою тактику, и она генерирует это утверждение как теорему. Тогда он ответил мне: «Насколько общей является ваша тактика? Где еще можно применить ее? У меня есть идея». У него была некоторая вещь, которую он хотел доказать, и он предложил мне еще один пример. На самом деле я уже доказал этот второй пример с помощью той же тактики, не сообщив ему, и мы применили эту тактику, и она, естественно, сработала, на что он ничего не сказал, но его молчание означало согласие. Я смог продемонстрировать полиморфность нашей тактики: ее можно было применять к разным вещам.

Вы также разработали теоретическую основу для анализа параллельных систем, исчисление общающихся систем (Calculus of Communicating Systems, CCS) и последовавшее за ним пи-исчисление. Полезны ли они для изучения и улучшения параллельной обработки в современном аппаратном и программном обеспечении?

Робин: Размышлять над общающимися между собой системами я начал, работая в лаборатории ИИ Маккарти в Стэнфорде в 1971–1972 годах. Меня поразило, что для удовлетворительной работы с ними в существовавших тогда языках не было почти ничего. Главным образом, я искал математическую теорию, которую языки могли бы использовать в качестве своей семантики. Это подразумевает какой-то вид модульности: нужна возможность собрать (параллельную) общающуюся систему из более мелких.

В то время уже была прекрасная модель, созданная Карлом Адамом Петри, – сети Петри, – которая очень хорошо справлялась с причинными связями. Существовала также модель Actor Карла Хьюитта. Сети Пе-

три не были модульными, и я хотел ближе подойти к некой параллельной теории автоматов, чем возможно было в модели Хьюитта, а также взять в качестве базисного понятие синхронизации связи (handshake). Кроме того, теория автоматов со своей семантикой в виде формальных языков (наборов строк символов) плохо справлялась с недетерминированностью и взаимодействием. Поэтому я взялся за CCS.

Моим главным стремлением было применить алгебраический подход – сначала к статике, а потом к динамике. Многие годы ушли на его совершенствование, включая важное предложение Дэвида Парка (David Park) о введении понятия подобия (bisimilarity) на основе максимальных фиксированных точек. Изначально я хотел моделировать системы, которые перенастраивают свое состояние; например, A и B не могут связаться друг с другом, пока C, имея доступ к ним обоим, не пошлет A адрес B. Сначала, в дискуссии с Могенсом Нильсеном (Mogens Nielsen), мы потерпели поражение (математически); затем Могенс кое-что поправил в том, над чем мы недостаточно хорошо подумали, и мы пришли к пи-исчислению, разработанному вместе с Джоахимом Парроу (Joachim Parrow) и Дэвидом Уокером (David Walker).

Восхитительно было то, что «пи» не только справлялось с реконфигурацией, но и могло – без каких-либо дополнений – представлять типы данных. Поэтому его можно рассматривать как базовое исчисление для мобильных параллельных систем подобно тому, как лямбда-исчисление относится к последовательным системам.

Пи-исчисление находит широкое применение, даже в изучении биологических систем. Но еще важнее, что появляется целый ряд новых исчислений, лучше моделирующих распределенные системы, оперирующие такими вещами, как мобильность и стохастическое поведение. Похоже, что, не став шлифовать теорию параллельных процессов, мы положили начало очень полезному развитию.

Можно ли научно разобраться в системе до того, как вы спроектируете и построите ее?

Робин: Я много размышлял над этим в связи с «вездесущими компьютерами», думаю, это общая проблема. Вы обязаны иметь какую-то модель того, как будет работать ваша система. В простейшем случае это будет машина фон Неймана, которая является научной, или формальной, или строгой моделью, которая действительно дала рождение Фортрану и целому ряду последовательных языков. Это научная модель. Она очень проста, и в этом была ее прелесть.

Нужна такая модель, из которой язык программирования может быть как бы извлечен и определен. Для вездесущих систем такая модель может быть очень далека от машины фон Неймана: она должна действо-

вать для широких популяций агентов, которые все взаимодействуют и перемещаются, есть датчики и так далее.

Вы как будто говорите о ряде метаязыков, описывающих семантику.

Робин: Я считаю, что слово «язык» нужно использовать тогда, когда есть модель. Конечно, на практике сплошь и рядом происходит прямо противоположное: описывают языки с помощью метаязыков, и очень часто до того, как появится хорошая модель, – если, конечно, метаязык не предоставляет именно такую модель. В таком случае метаязык может служить синонимом для модели. Мы поступили так, описывая стандарт ML. Мы воспользовались метаязыком, предназначенным для индуктивного вывода, чтобы определить допустимые инструкции и то, что они делают. Думаю, это и есть общая модель. Вы правы: я говорю о группе или типе неизвестного семейства метаязыков. Каждый из них оказывается ориентирован на конкретную систему, которую можно назвать программой.

В этом смысле не являются ли компьютерные вычисления определением и формализацией моделей на нескольких разных уровнях, что позволяет вам строить модели на более высоких уровнях?

Робин: Да. И это очень тревожит меня в связи с «вездесущими компьютерами», потому что есть очень много идей, которые должно отражать поведение конкретной системы, но неизвестно, удастся ли их все включить непосредственно в одну модель. Я говорю о том, что модели строятся одна над другой, и в самом низу может находиться достаточно простой механизм. На верхних этажах появляются более интересные, более очеловеченные или более тонкие идеи, как, например, управление отказами, самоосознание, доверие, безопасность и так далее. Модели нужно строить каким-то многоуровневым способом, чтобы в каждой модели оговаривался достаточно удобный набор идей, которые потом реализуются в модели более низкого уровня.

В Лиспе и Форте часто обсуждается получение и построение систем из многократно используемых концептов смысла. В некотором отношении, вы разрабатываете богатый язык, чтобы решить задачу.

Робин: Если рассматривать стек моделей, то на нижнем уровне будет находиться то, что можно назвать программами. На более высоких уровнях находятся спецификации и описания того, что можно и чего нельзя, что должно или не должно произойти. Они могут существовать в любом виде – от логического формата до обычного языка. Опускаясь на нижние уровни, вы встречаете знакомые вещи, называемые программами, которые можно рассматривать как конкретные модели.

Это свидетельствует о том, что наши вычислительные модели по существу являются процедурными?

Робин: Да. Если взять более динамически точные модели, то они, думаю, будут процедурными. Можно создать модель спецификаций. Она может состоять из логических предикатов: это модель не вполне динамическая, но потом с помощью пар формул предикатов, выражающих пред- и постусловия, вы сможете оценить надежность реализации по тому, верифицируема ли она логически. Вы переходите от спецификации или модели, которая не выглядит динамической, к более динамичной. Это любопытно – я не вполне понимаю, как происходит переход от динамичности внизу к описательности наверху, но, кажется, это так.

В другом случае, как в методе абстрактной интерпретации, у вас сохраняется динамическая модель на верхнем уровне, но работа происходит с некоторой абстракцией данных. Она на самом деле не будет программой, но будет динамической. Это то, чем воспользовались французы для верификации встроенного программного обеспечения Аэробуса. Интересный и сложный вопрос – когда модель будет динамической, а когда лишь описательной.

Возможно, этот сдвиг происходит, когда нам приходится признать действие законов физики – например, поведение вентилей «Не-И». Мы понимаем эти физические процессы, но в определенный момент модели, которые мы строим, начинают включать этот уровень.

Робин: Да. Есть электрические схемы компьютеров, и они относятся к электронике, но над ними появляется код ассемблера, и вы забываете про электронику. Но при перемещении вверх вы все еще сохраняете в программе динамический элемент, как если бы он транслировался в динамический элемент в электрических схемах. Можно как бы проходить через разные понятия динамичности, сохраняя ее, но при подъеме вверх ее характер сильно меняется.

В логических моделях тоже часто есть динамический элемент. Например, так называемая модальная логика определяется в терминах возможных миров и перемещения из одного мира в другой. Тут есть динамический элемент, но он слегка завуалирован.

Могу представить себе возражения по поводу того, что ошибки или элементы неразрешимости на нижних уровнях могут повлиять на возможности вычислимости на верхних уровнях.

Робин: Это просто проверено жизнью. Если на нижнем уровне вам не удастся умерить неразрешимость, то на верхнем уровне это делается путем проверки типов. Типы являются абстрактной моделью, в которой можно получить разрешимость, потому что это слабая абстракция, и у вас нет компонентов, приводящих к неразрешимости. Конечно, они относятся лишь к какой-то одной стороне программы, поэтому разрешимость обретается только при движении вверх и отказе от деталей.

В таком направлении я не думал об этом.

Робин: Я тоже не так много об этом размышлял. Что касается проверки типов, то есть такие системы типов, для которых разрешим вопрос о правильной типизиции программы и ее типе, но после какого-нибудь совершенно незначительного изменения разрешимость теряется. При этом вы лишь чуть больше детализировали свою систему типов. Такое произошло с системой типов, которую мы использовали в ML: она была, в сущности, разрешимой, но после добавления так называемых конъюнктивных типов стала неразрешимой.

Такие конфликты возникают, когда хочется ввести удобную вещь и при этом сохранить полный контроль. Очень часто, даже если не всегда удается проверить что-то в системе с конъюнктивными типами, вы все равно добиваетесь огромных успехов с помощью достаточно интеллектуального средства доказательства теорем.

Все это можно описать с помощью того, что я называю «башней моделей». Поднимаясь на ее верхние этажи, вы постепенно теряете информацию. При этом могут появиться некоторые возможности для анализа, что может иметь ценность, поскольку вы будете анализировать свойство программ, которое полезно знать, даже если полной картины у вас не будет.

Я слышал, что можно пойти другим путем. Выражение в модели верхнего уровня означает, что можно в значительной мере сократить неразрешимость на нижних уровнях, если доказать, что некоторые условия никогда не выполняются.

Робин: Да, понятно. На нижнем уровне находится в сущности неразрешимая модель, но при определенных ограничениях на рассматриваемые элементы она может стать разрешимой.

Действительно ли неразрешимость всегда так плоха, как это кажется?

Робин: Нет, но в связи с этим возникает интересный вопрос о пользе моделей и их влиянии на неразрешимость. Думаю, это интересная тема.

Как следует нам — специалистам по компьютерным наукам и действующим программистам — преподносить идеи теорем, доказуемости и систем типов людям, которым всего лишь нужно решать конкретные задачи?

Робин: Скорее всего, в корне неверно начинать с этого обучение специальности. Мы против этого, и такая же ситуация встречается в математике. Вы рассказываете о вещах, которые позднее будете излагать в более абстрактном виде, но сначала говорите о них более конкретно, чтобы слушатели получили хотя бы смутное понятие. Вы рассказываете о евклидовой геометрии и помалкиваете о том, что существуют дру-

гие. Позднее, может быть, на втором курсе университета, уже может появиться понимание того, что представляют собой другие геометрии, но большинству 17-летних такой уровень абстракции недоступен. Было бы неправильно строить преподавание в расчете на то, что материал окажется доступен достаточной доле студентов.

Я знаю, что делаю ошибку в программе на получение степени, пытаясь преподавать то, что слишком абстрактно для последнего курса бакалавриата. Многое в теории вычислений часто оказывается слишком абстрактным даже для этого уровня. Приходится как-то с этим мириться. Беда в том, что для должного понимания предмета без этих абстракций приходится выстраивать некую иерархию понимания. Одни никогда не захотят говорить об этих абстракциях. Другие полюбят их, и все, что вам нужно, это убедиться, что между собой они могут говорить хотя бы о чем-то.

Ограничивает ли это полезность теории для программистов-практиков? Можно ли надеяться, что хотя бы 20% из них заинтересуются теорией?

Робин: Им необязательно знать теорию. Язык – это инструмент, а инструменты есть самые разные. Проверка модели – это инструмент, позволяющий избежать чрезмерного углубления в детали. И это нормально – при условии, что кто-то разбирается в сути и знает, что инструмент для проверки модели надежен. В сущности, в нашем деле есть множество инструментов, освобождающих людей от необходимости изучать некоторые вещи, чтобы они могли заняться делами поважнее. У них есть более значительные и срочные задачи, для решения которых как раз предназначены языки программирования высокого уровня. Мне очень нравится, что из некоторых теорий можно получать языки программирования.

Я работаю с графической моделью для «вездесущих компьютеров». Это описательный механизм, вероятно, мало кому понятный, но из него можно извлечь язык, который, как мне кажется, понять достаточно легко. При извлечении языка применяются различные метафоры – специальные метафоры, структурные ограничения и так далее, – чтобы переход от абстрактной модели к языку программирования создавал людям удобства, ограждая от того, с чем они не хотят связываться. Например, системы типов защищают вас от вещей, о которых в большинстве случаев вам не хотелось бы знать. Такова природа нашего предмета: мы движемся от низших моделей к высшим, достигая все большей абстракции, и каждый готов пройти определенный путь вверх – или вниз, – но не более того.

При перемещении вверх по башне моделей не обязательно усиливается абстракция – могут расти ограничения. Прекрасным примером служит

модель диаграммы последовательности сообщений, описывающая конечные участки параллельного прохождения сообщений и то, что может или не может при этом происходить. Мне представляется, что это ограничительная модель, которую легко перевести в более сложную модель с циклами рекурсии и всякими ужасными вещами вроде условий гонки, о которых вам не хочется думать. Прелесть модели диаграммы последовательности сообщений в том, что ее может понять ваше начальство, так что при перемещении вверх по башне моделей вы не только абстрагируете, облегчая теоретическое понимание, но можете ввести ограничения, чтобы облегчить понимание модели теми, кто не является узким специалистом.

С одной стороны, достигается большая степень обобщения, а с другой — большая конкретность.

Робин: Да, именно так. Для меня это загадка. Хотелось бы поместить более конкретные вещи вниз, а не вверх, а я помещаю их вверх. Главное, все видоизменяется, облегчая кому-то понимание за счет обобщения. Мне кажется, это стоит труда.

Если модель представляет собой набор теорем, построенных на более фундаментальных принципах, то как это влияет на идеи, которые можно описать с помощью данной модели?

Робин: У меня есть пример. Надеюсь, он не слишком искусственный — это касается модели, с которой я работаю. Есть модель мобильных систем — таких, где перемещаются сообщения, перемещаются датчики и актуаторы, как это происходит в «вездесущих компьютерах». Можно устроить такую модель, которая многое расскажет вам. Можно описать инварианты, например, чтобы не возникало состояние, когда в одной комнате находится больше 15 человек, или что-то в этом роде. Но в одном варианте модели нельзя проследить за конкретным лицом и сказать: он ни разу не был в этой комнате.

Пример может показаться странным, но он очень простой: эта модель не позволяет проследить за отдельной личностью в ходе разных событий и реконфигураций. Нельзя даже спросить, был ли когда-нибудь этот человек в этой комнате, потому что неизвестно, как сказать «этот» человек: «этот» предполагает постоянство личных данных во времени, особенно в связи с употреблением глаголов в разных временах.

Это пример модели, в которой некоторые вещи невозможно даже описать, и меня это очень заинтересовало, потому что для некоторых задач это может стать преимуществом. Это большое преимущество, если применять данную модель к биологическим системам, где речь идет о совокупностях миллионов молекул и неинтересно, какая из них где: вы всего лишь хотите знать, например, сколько этих молекул будет через

15 минут. Эта модель может быть очень полезна в биологии, потому что не требует идентифицировать отдельные молекулы.

Индивидуальность не так важна, как стохастическое описание?

Робин: Верно, в данном конкретном случае, где она может отсутствовать для многих задач. Конечно, я провожу аналогию между биологическими и вездесущими системами, где люди или агенты того или иного рода перемещаются по городу или в некоторой контролируемой среде. В последнем случае гораздо более вероятно, что вас заинтересует личность отдельного индивида. Он никогда не был в этом пабе, где было совершено преступление. Вам может понадобиться сказать это, поэтому вы должны знать, что означает «он» в зависимости от времени.

Если взять модели для «вездесущих компьютеров», то пространство может рассматриваться в них совершенно дискретным образом, поэтому нельзя ничего сказать о расстояниях. Можно лишь говорить о том, что некоторые объекты примыкают друг к другу или вложены один в другой. Не нужно моделировать непрерывность пространства, так что можно забыть о ней. Действительно оказывается, что есть множество характеристик, без которых модели могут свободно обойтись в некоторых приложениях, а для других целей их потребуется уточнить.

Допустим, что я пишу API. Чем лучше мои проектные решения, тем более выразительна и проста для понимания моя модель. Кроме того, мою систему может оказаться проще использовать корректно, чем наоборот.

Робин: Вероятно, это происходит с любым рассматриваемым семейством систем. Например, если речь идет о системах безопасности, то в какой-то конкретной модели вы окажетесь не в состоянии описать некоторые аспекты безопасности, но другие аспекты будут вам доступны. Либо модель сможет описать свойства безопасности, но ничего не сможет сказать о защите личных данных. Такие свойства, как подлинность (authenticity), конфиденциальность (privacy), безопасность (security) и защищенность (safety), имеют тонкие различия. Большинство моделей может описывать лишь часть из них.

Если снова обратиться к биологической метафоре, то мне не нужно понимать, какие превращения происходят с АТФ в моих клетках, чтобы взять на руки и погладить кошку, и этим клеткам ничего не известно о кошке. Для них это просто куча каких-то других клеток.

Робин: В этом отношении мы не очень отличаемся от естествознания. Можно рассчитывать движение тел, пользуясь механикой Ньютона, и не учитывать трение.

Можно ожидать возрождения доказуемости?

Робин: Думаю, что она будет нужна, пока продолжают разрабатывать языки с явной семантикой. Эта семантика будет совершенно бесполезна, если нет теорем о ней, которые можно доказать, например «ни одна программа никогда не сделает такую глупость» или какие-нибудь более положительные высказывания вроде «размер требуемой программам памяти всегда будет полиномиальным».

Это может побудить разработчиков новых языков к проведению такого рода доказательств.

Робин: Хорошо бы это было так. Обычно языки проектируют, не слишком задумываясь о таких вещах. Я бы хотел, чтобы это изменилось, но такова практика, и обычно этим занимаются целые комиссии. Бывают отличные проекты, но доказательства их обычно не заботят. И обнаруживаются ошибки, как ошибка в Паскале, связанная с вариантными записями, где нарушена дисциплина типов. Такие отрицательные теоремы о языках встречаются весьма часто.

Думаю, есть некий прогресс, потому что такие доказательства относительно языков действительно появляются и не требуют огромных усилий благодаря эффективности систем для доказательств, что позволит разработчикам языков получать данные от тех, кто проводит доказательства. Это будет прекрасно.

Теория смысла

Какова связь между конструкцией языка и конструкцией программ на этом языке?

Робин: Это громадный вопрос. Довольно давно, кажется, в 1960–1970-е, предполагалось, что появится единый Универсальный компьютерно-ориентированный язык (UNiversal Computer Oriented Language, UNCOL), но надежда не осуществилась. Предполагалось, что UNCOL будет использоваться всюду. Если бы такой язык появился, это означало бы, что существенной связи между структурами языка и программ нет!

Сейчас, как и ранее, мы наблюдаем последовательное появление языков, почти все из которых ориентированы на определенные приложения. Пролог благоприятствует приложениям, в которых удобно описывать действия логическими формулами, поэтому программу на Прологе следует организовывать логически. У ML и Haskell богатая система типов, поэтому конструкция программ на ML и Haskell часто тесно связана с системой типов. И так далее. Любую задачу можно описать на многих языках, и хотя структура программы в мозгу программиста всегда остается неизменной, но каждый язык лучше передает явным

образом какие-то одни части этой структуры, оставляя другие в неявном виде. Для каждого языка эти части различны.

Решая конкретную задачу, программист, как мне кажется, часто выбирает язык, выявляющий те аспекты задачи, которые кажутся ему главными. Однако некоторые языки идут дальше: они действительно влияют на то, каким образом программист представляет задачу в уме. Например, в этом сильно преуспели объектно-ориентированные языки, потому что понятие объекта способствует ясности мышления во многих видах приложений.

Какие парадигмы, помимо ООП, влияют на способ проектирования и мышления программиста?

Робин: Думаю, такое влияние оказали логическое и функциональное программирование. Думаю также, что влияние имеют парадигмы исчисления процессов. Оно ясно проявилось в LOTOS – языке спецификаций – и, как мне кажется, в Аде – среди прочего, в идее команд ALT.

Вместо того чтобы выбирать язык для каждой задачи, не лучше ли каждому программисту работать на собственном языке программирования? Нельзя ли свести все языки к нескольким семействам?

Робин: Если каждый программист станет пользоваться своим языком – это будет анархия, если только не ограничить смысл языка общепринятой теорией. В конце концов, как определить смысл его языка, если не в терминах принятой теории? Если есть теория, программист может изобретать синтаксические фразы, объясняемые этой теорией. Поэтому он будет пользоваться «собственным» синтаксисом, но смысл будет взят из теории, и при описании своего языка он явно опирается на эту теорию. Ничего страшного. Но поскольку за этими языками будет стоять теория, можно предполагать, что у них будет много общего.

В чем смысл разработки языка программирования? В создании инструмента для выражения идей или инструмента для выражения задач?

Робин: Если взять прекрасный пример функционального и логического программирования, то вначале была теория: для функционального программирования – теория функций, теория типов, значений, а для логического – развитая теория логики первого порядка. Теория существовала до появления языка, и язык более или менее основывался на ней, так что это примеры того, как теория предшествует языкам, и мне кажется, что их должно быть больше, – не знаю, сколько разных языков должно быть.

Можно сказать, что главное в разработке языка – цель, которой хочет достичь разработчик.

Робин: Возможно, вам потребовалось описать цель или свойства поведения программ на другом языке или с применением другого теоретического инструмента. Например, вы захотели записать свои спецификации с помощью некой логики, и язык программирования должен носить более алгебраический характер, но то и другое – алгебра и логика – должны быть удобно соединены, прежде чем вы станете проектировать какие-то их части как язык программирования.

Мне кажется, что инструмент, с помощью которого вы описываете задачи и желаемые свойства, может не совпадать с тем, который применяется для описания программы, но они должны быть связаны между собой в некой теории, которая, возможно, нужна не только для создания программ, но для понимания явлений природы, как в примере из биологии, который я приводил. Видимо, если мы сможем понять информатику, то сможем понять природные системы с точки зрения информатики, что и является задачей ученого-естественника. Но вполне вероятно, что теми же самыми формализмами, математическими конструкциями и свойствами мы сможем воспользоваться, чтобы определить языки, и потому внести артефакты, не являющиеся природными явлениями. Поэтому не вижу причин отделять описание с точки зрения информатики природных систем от описания систем программирования или систем программного обеспечения.

Предположим, что вы обнаружили ошибку в системе, написанной пять лет назад. Ваша спецификация синхронизирована с реализацией. Как быть, если ошибка допущена при разработке языка? Что делать, если некий дефект влечет определенный вид ошибок?

Робин: К большой моей радости, такого не случилось, и я не знаю, что бы мы делали в таком случае. Вероятно, мы заявили бы, что всем придется с этим смириться. Мы бы опубликовали что-нибудь со словами: «Да, здесь есть ошибка, но если поступать так-то и так-то, можно о ней не беспокоиться». Определения достаточно уязвимы. Некоторые работают над идеей сделать механизмы описаний менее уязвимыми и более модульными, а мне кажется, что это очень сложно. Не знаю, что тут можно сделать. Я склоняюсь к тому, чтобы ничего не менять, а просто объявить, что есть такая проблема. Это просто практический шаг, чтобы не рвать на себе волосы.

Если у вас есть язык с богатой системой типов, такой как ML или Haskell, какие идеи раскрывает эта система типов в структуре программ на этих языках?

Робин: Если система прошла компилятор, то есть контроль типов, то некоторые вещи просто не могут произойти. Можно быть уверенным, что не будет определенных типов ошибок времени исполнения. Это не

касается таких ошибок, как переполнение, и многих других неприятных вещей вроде бесконечных циклов, но все же вы будете знать многое.

У нас было приложение для доказательства математических теорем, и было удовольствием иметь возможность сказать: «Если вы считаете, что доказали теорему на ML, то есть что ваше представление правил вывода в логике на ML сделано корректно и программа ML дает доказательство теоремы, то теорема, несомненно, доказана». И это благодаря механизму абстрактных типов, позволяющему описать тип теорем как нечто, к чему применимы только правила вывода. Можно применить хитрые приемы для поиска возможных выводов, и если в одном из поисков возможных цепочек выводов будет достигнут успех, то вам нужно практически применить этот вывод к типу теоремы. Вы знаете: единственное, что вам доступно, это допустимые выводы. В некоторых вариантах поиска допустимых последовательностей вывода вы не получите результата, но если получите, то примените эти выводы, или система сделает это за вас. В рамках верификации правильности реализации и правильности конструкции языка вы получили теорему. В языке без типов вы этого сделать не можете.

У вас есть только набор операций.

Робин: Система говорит: «У меня есть теорема», а вы отвечаете: «А как я могу быть в этом уверен?» Это серьезно. Помню первые дни в Стэнфорде, когда мы проектировали первую версию, – это фактически даже не был ML. Мы работали над автоматизированной системой вывода и были уверены, что автоматизировали ее правильно, а то, что можно было вывести, выводилось только на основе правил вывода.

Помню, как поздним вечером я размышлял, что если получится и выскочит теорема, которая заявит «я теорема!», то мне не нужно будет сомневаться, потому что я уверен в типах и уверен в реализации. Я был настолько уверен, что мне казалось, даже если я стану творить за терминалом какие-нибудь чудовищные вещи, это не повлияет на надежность системы.

Это действительно очень сильная функция, которая, думаю, всегда была в таких системах, как Isabel и HOL, и всех других подобных системах, которые есть сейчас. Эта особенность удивительно раскрепощает, потому что наступает момент, когда вам не нужно беспокоиться.

Тогда вопрос: как заставить компьютер сообщить нам, что означает наша программа?

Робин: Отдельная программа, по-видимому, говорит примерно следующее: если ты сделаешь со мной это, произойдет одно, а если сделаешь со мной то, произойдет другое. Типы позволяют делать жесткие заявления такого рода. И тут на помощь приходит компьютер: компилятор

помогает вам, осуществляя контроль типов. Конечно, это необязательно должен быть разрешимый контроль типов: он может быть таков, что если он приходит к выводу, что программа правильно типизирована, то так оно и есть, но относительно некоторых программ он может не прийти ни к какому выводу. Существуют очень богатые системы типов, в которых есть неразрешимость, но есть и то, что называют «положительной безопасностью»: если теорема прошла, значит, это действительно теорема.

Что бы вы посоветовали тому, кто хочет научиться хорошо разрабатывать программы?

Робин: Решить, чего он хочет: зарабатывать деньги или заниматься наукой. Нельзя советовать человеку, что из указанного он должен выбрать, но есть масса способов зарабатывать деньги, избежав занятий наукой. И наоборот.

Если совет предназначен тому, кто хочет заниматься наукой, то я скажу так: общайтесь с людьми, занимающимися разработкой, и не сидите в одиночестве, придумывая красивую теорию, а постарайтесь, чтобы она имела какое-то отношение к практике.

Вы указали на проблему 2000 года как на хороший пример ситуации, в которой мы не знаем, с какими проблемами придется столкнуться. Как предотвратить возникновение аналогичных конструктивных проблем на стадии проектирования?

Робин: Не знаю. Рынок настолько жаден до программных продуктов, что если вы будете тратить время на анализ того, что хотите продать, кто-то другой перехватит ваш контракт. Звучит цинично, но мне кажется, что дело обстоит именно так. В реальном мире вы не достигнете успеха, пытаясь применять аналитические инструменты, если таковые существуют. Очень часто их просто нет.

Когда мы столкнулись с «ошибкой тысячелетия», у нас фактически была вся теория, необходимая для того, чтобы вовсе избежать этой проблемы, если бы мы только правильно писали программы: нужно было лишь потрудиться воспользоваться теорией типов, существовавшей к тому времени уже около 20 лет. Есть множество предположений, почему эту теорию проигнорировали, но мне кажется, что главная вина – на рыночных силах.

Может быть, помогла бы документация. Как разработчики должны писать документацию?

Робин: Разумеется, они должны снабжать код комментариями, но при этом следует соблюдать некие строгие правила. Трудность составления адекватных комментариев, я считаю, нелинейно зависит от размера

программы, поэтому когда у вас есть миллион строк, она значительно превышает то, что было бы адекватным для программы в несколько тысяч строк. Возникают значительные трудности: сложность взаимодействия отдельных частей программы растет быстрее, чем линейно, поэтому необходимость строгой спецификации для «настоящих» программ значительно выше.

Кстати, мы написали полную формальную спецификацию ML в виде правил вывода. Мы заложили основу формального описания языка. Чего мы не сделали, так это не описали, как реализации должны соотноситься с этой формальной спецификацией, но благодаря наличию этой формальной спецификации мы весьма хорошо понимали, что собираемся реализовать. Во-первых, у нас была очень четкая спецификация, а во-вторых, мы не собирались ее изменять или, во всяком случае, делали это очень и очень умеренно.

Что касается реально работающих программ, то они должны обеспечивать возможность модификации или удаления отдельных частей, их адаптации и добавления новых, потому что спецификация неизбежно будет меняться. Для реальных программ в реальном мире есть дополнительное основание очень осторожно подходить к связи между спецификацией и реализацией, потому что спецификация будет меняться, и нужно точно знать, как это отразится на модификации реализации.

Можете ли вы вспомнить какие-нибудь истории, связанные с разработкой ML?

Робин: Ну, во-первых, мы гораздо дольше обсуждали синтаксис, чем семантику. Мы в значительной мере достигли согласия по функциональному пониманию языка, но очень часто, когда выбор диктовался личными вкусами, – например, какое именно слово использовать в синтаксисе, – мы могли спорить бесконечно, поскольку не было научной основы для принятия решения.

Другая история. Мы снабдили ML тщательно продуманной системой типов, которая выглядела скромно в сравнении с некоторыми существовавшими в то время теориями типов. Мы применяли ML для формальных доказательств с помощью математической логики, и нам потребовалось обеспечить эффективную реализацию того, что называлось «правилами упрощения». Преобразование сложного выражения в выражение одного определенного вида, который называют *нормальной*, или *канонической формой*, происходит в соответствии с большим набором правил. Для быстрого выполнения преобразований они должны быть реализованы специальным образом, чтобы просмотр всех применимых правил и выбор из них подходящих мог происходить в какой-то мере одновременно.

Реализуя такой одновременный поиск в ML, мы обнаружили, что достижению достаточной эффективности что-то мешает, и, как оказалось, это связано с некоторыми ограничениями нашей системы типов. Мы решили отменить это правило ML для данной конкретной части реализации – в действительности, это была реализация аналитического инструмента, средства доказательства теорем, – после чего эффективность значительно выросла. Собственно говоря, в этом не было ничего особенно страшного, потому что используемую более общую систему типов мы достаточно хорошо понимали, но она была сложнее: мы хотели, чтобы у ML была простая система типов, и нам потребовалась немного более общая, но и немного более сложная система, чтобы более эффективно выполнять некоторую часть работы.

Если бы вам пришлось начать все заново и проектировать ML сегодня, сильно ли повлияли бы достижения современной компьютерной науки на ваши решения – или они оказались бы примерно теми же?

Робин: Язык разрабатывался для доказательства теорем. Оказалось, что доказательство теорем – настолько требовательная задача, что язык превратился в универсальный. Отсюда возникает вопрос: а для каких целей я стал бы проектировать этот язык сейчас? Если снова для доказательства теорем, то возникли бы те же самые проблемы. Нужна такая вещь, состояния которой можно менять. Чисто функциональный язык не годится, потому что менять состояния нужно часто: требуется управлять деревом вывода или чем-то другим, что вы выращиваете, либо деревом задач и подзадач. Это должно быть изменяемо.

Перейдя из того мира в область систем, динамичность которых гораздо более заметна, таких как «вездесущие компьютеры», я бы растерялся, пользуясь функциональным языком. Если бы я проектировал язык для доказательства теорем, то более удачной идеей могли бы стать те средства, которые есть в Haskell, – монады, с помощью которых там управляют последовательностью, – хотя я в этом не уверен. Мне пришлось бы хорошенько разобраться в том, для чего я проектирую язык.

Меня весьма озадачивает, что люди берутся разрабатывать языки без избранной предметной области, без каких-либо специфических задач, решение которых они хотели бы облегчить. Разработчики Java имели, по-видимому, хорошее представление о таких вещах, поэтому у них получился хороший язык. Но диапазон возможных предметных областей сейчас достаточно велик, поэтому мы наблюдаем появление множества разных языков для разных задач. Цель – это скрытый параметр. Если бы я проектировал язык для той же самой цели, вполне возможно, что у меня получился бы тот же самый язык.

Я видел вашу работу, и мне показалось, что вы действуете по плану: «Для этой задачи я создам набор многократно используемых примитивов — моих теорем, — и на их основе буду строить другие теоремы».

Робин: По-моему, для работы с ML не нужно держать в голове много примитивов, но вы, вероятно, имеете в виду разработчика, а не пользователя. Разумеется, когда мы разрабатывали эту систему и пользовались определенными функциональными структурами и семантикой, учитывалось, что нам нужно будет доказать определенные теоремы, относящиеся к языку в целом, например, что не будет висячих ссылок. Было довольно много вещей, истинность которых нам хотелось проверить, что и было позднее сделано с помощью автоматических и полуавтоматических систем доказательства. К большому моему облегчению.

У нас была неформальная уверенность, что висячих ссылок не должно быть, но хотелось иметь формальное доказательство, потому что могла быть допущена какая-то ошибка. С другой стороны, у нас действительно возникли проблемы со ссылочными типами, типами переменных, которым присваиваются значения. Несколько человек продемонстрировали, что эти типы некорректно ведут себя в нашей системе типов. Если бы мы наложили некоторое ограничение на язык, такой проблемы не возникло бы. При исследовании этого вопроса выяснилось, что ограничение затронуло бы только 3% программ! Удовлетворись мы оставшимися 97%, эту проблему можно было бы снять. Мы пересмотрели язык, чтобы наложить это ограничение, что и послужило причиной появления пересмотренной версии, выпущенной в 1997 году, против версии 1990 года с первоначальной семантикой.

Является ли формальная ревизия единственным способом обеспечить синхронность реализации со спецификацией при пересмотре языка?

Робин: Думаю, мы сохранили их синхронность. Мы смогли показать, что обеспечивается прямая совместимость. Иными словами, прежние реализации могли работать, если применялись к программам, имеющим некоторые небольшие ограничения. Прямая совместимость была важной задачей при ревизии.

На самом деле, я едва ли не жалею о сделанном пересмотре, но очень хотелось исправить ошибку и сделать все проще, как предлагали эти люди. Но, занимаясь исправлением, мы пересмотрели и ряд других вещей. Мы потратили на пересмотр больше труда, чем хотелось. С другой стороны, хорошо, что мы это сделали, потому что выяснились важные вещи относительно системы типов и мы смогли обеспечить более простой подход с этой стороны.

За рамками информатики

Какие главные проблемы существуют сегодня в компьютерных науках?

Робин: Последнее время я работаю над идеей структуры моделей. Текст на языке программирования высокого уровня описывается затем объектами языков более низкого уровня, затем кодом ассемблера более низкого уровня, а поведение кода ассемблера описывается на языке логических диаграмм, так что ваша модель становится не программной моделью, а моделью электронных объектов. Это, в свою очередь, объясняет такой артефакт, как компьютер, который в итоге будет выполнять вашу программу, находящуюся где-то на четвертом уровне иерархии моделей. И это еще не конец, потому что можно подняться еще выше – от языка программирования к языку спецификаций, в некотором смысле служащему моделью более высокого уровня, – вот вам уже пять уровней.

Теперь возьмем «вездесущие компьютеры» – системы, которые будут вести ваше домашнее хозяйство и сами загружать ваш холодильник, а также следить за состоянием здоровья, прикрепляясь к телу или даже перемещаясь внутри него. Чтобы понять такие системы, требуется много уровней моделирования, потому что их представляют как программные объекты, которые вступают в переговоры друг с другом, запрашивают друг у друга ресурсы, доверяют друг другу и отображают свое поведение – иными словами, демонстрируют разнообразные гуманитарные свойства. Частично поведение этих систем будет описываться логикой очень высокого уровня, относящейся к доверию, знанию, убеждениям и так далее. Следовательно, нужна теория такой логики, разъясняющая, как задавать спецификацию программы в терминах более элементарного поведения. Это должны быть спецификации обычного вида, применимого к функциональному поведению программы. Но тогда на более высоких уровнях должны возникать вопросы вроде «верно ли, что эта программа доверяет той?», или «как реализовать идею доверия между вычислительными агентами?», или «как вы представляете себе, каким образом можно сообщить, что один агент верит в намерения другого агента или что другой агент представляет собой угрозу его личным планам?».

Мне представляется, что такого рода вопросы располагаются где-то на три уровня выше, чем обычный уровень спецификаций для наших обычных программ. Любые модели либо моделируют ПО, либо моделируют другие модели, которые объясняют ПО на каком-то более низком уровне. Кроме того, если вы создаете что-нибудь вроде европейского Аэробуса, вам нужно объединить модель ПО с электромеханической

моделью работы самолета или даже с моделью атмосферных условий или погоды, в которых будет проходить полет. Поэтому возникает задача объединить эти модели, часть которых может иметь источником естественные науки, например метеорологию или электромеханику, а также модели ПО, чтобы они взаимодействовали между собой. На уровне таких комбинированных моделей вы должны быть в состоянии предсказывать функционирование этого Аэробуса.

Мне нравится мысль объединить природное с рукотворным, применив к обоим одну и ту же идею моделирования; просто в случае искусственного модель предшествует артефакту, а в случае природного у нас сначала есть явление, а потом появляется его модель. В некотором роде это отражает единство информатики и естествознания.

При разработке аппаратуры можно проверить ее работу физически. В программировании ошибки реализации могут не обнаружиться, пока не появятся конечные пользователи. Как можно было бы объединить эти различные этапы разработки, реализации и практического применения?

Робин: Если взглянуть на модели, созданные естественными науками, то в конечном счете они проходят проверку только в результате наблюдений, путем выяснения, что реальный мир ведет себя так, как предсказывают модели. Это означает, что невозможно полностью подтвердить эти модели: их можно только опровергнуть, если обнаружится, что в каких-то случаях результаты наблюдения не соответствуют предсказаниям модели. Подтвердить наблюдениями все предсказываемые явления невозможно, потому что это потребует бесконечного объема работы.

В лучшей ситуации мы оказываемся, например, занимаясь реализацией языка программирования высокого уровня с помощью языка более низкого уровня. У нас имеется формальное описание того, как ведет себя каждый из них, и потому мы можем подтвердить реализацию, являющуюся трансляцией программ верхнего уровня на нижний уровень, в результате наблюдения, что научное и теоретическое объяснение на верхнем уровне действительно согласуется с объяснением на нижнем уровне. Поэтому у нас появляется возможность подтвердить правильность способа, которым одна модель реализована с помощью другой на более низком уровне. Только когда мы спускаемся ниже для реализации программ нижнего уровня как физических артефактов, теряется возможность подобного рода математических доказательств, а на всех верхних уровнях у нас есть такая возможность, при условии, что модели хорошо описаны, а смысл объектов на каждом уровне является составной частью модели этого уровня. На каждом уровне есть

объекты и есть объяснение того, как они себя ведут, и это тот способ, которым предполагается подтвердить правильность реализации модели верхнего уровня с помощью некоторой реализации более низкого уровня.

Вот в этом направлении я и пытаюсь заставить людей думать. Например, одно из недавних моих выступлений называлось «Вездесущие компьютеры: поймем ли мы их?» («Ubiquitous computing: shall we understand it?»), и под «поймем» я подразумевал как раз то, что хотелось бы определить поведение одной из таких систем, как монитора поведения человеческого тела, и понять, как спецификация этой системы фактически реализуется с помощью агентов.

Не думаю, что будет просто убедить в этом людей: они часто говорят, что такое невозможно, поскольку системы окажутся слишком велики, чтобы с ними можно было справиться. Я и сам читал один европейский доклад, где говорилось, что невозможно анализировать поведение вездесущих вычислительных систем. Мне кажется, что это совершенно ошибочное заявление. Сможем мы анализировать системы или нет, зависит от того, как мы их спроектируем, и мы должны создать их в таком виде, чтобы анализ был возможен.

Какие вы видите связи между техникой и информатикой?

Робин: Очень часто технологии основаны на достижениях естественных наук, появившихся раньше. Химические технологии в большой массе своей возникли после того, как были придуманы и проверены на практике химические теории, поэтому химические технологии возникают как результат химической науки, наблюдавшей природные явления, и изобретение технологий возможно благодаря знаниям, полученным естественной наукой.

Думаю, то же самое относится к физике, но в программировании положение, видимо, иное, поскольку в природе программы не встречаются и, насколько я понимаю, было бы большой натяжкой утверждать, что программы существуют в нашем мозгу или где-то еще, поэтому нет развитой науки, которую можно было бы положить в основу программных технологий. Поэтому мне кажется, что связь между технологией и ПО представляет собой не столько связь, сколько ее противоположность: не существует программных технологий, основанных на общепринятых научных данных, тогда как большинство прочих видов технологий основано на общепринятых научных данных.

Какова роль математики в компьютерных науках?

Робин: Здесь используются разные разделы математики. Это логика, алгебра, теория вероятностей; в гибридных системах, где непрерывные

явления смешаны с дискретным поведением, мы применяем дифференциальное исчисление. Таким образом, найдется место для самых разных разделов математики. Не совсем ясно, как выбирается тот или иной раздел. Вы выбираете его потому, что вам нравится заниматься теорией вероятности, или потому, что вам нравятся индикаторы технического анализа? Или потому, что представляете себе некоторую компьютерную или информационную систему, которую можно будет описать?

Если вы математик, то можете выбрать для исследований то, что вам нравится; мы же должны посмотреть на то, какие системы встречаются на практике, естественные они или искусственные, и подумать, что нужно для их объяснения.

Недавно мне пришлось разобраться в стохастическом анализе, который занимается продолжительностью, ходом времени и тому подобным. Он абсолютно необходим, если мы собираемся использовать модели для объяснения биологических явлений. Мне пришлось изучать то, чем я никогда ранее не занимался, чтобы понять эту теорию. Я также начал разбираться в весьма абстрактных разделах математики, таких как алгебра, теория категорий и так далее. Обычно обнаруживается, что вам нужна лишь малая часть теории, поэтому вы не занимаетесь разработкой весьма изощренных чистых теорий, как это делают математики, а берете по кусочку здесь и там из разделов, которые хорошо разработаны или менее изучены, из-за того что недостаточно красивы, так что в результате иногда вносите вклад в чистую теорию, хотя вашей задачей было описание каких-то реальных явлений.

Кем вы считаете себя — компьютерным ученым или исследователем?

Робин: Мне не нравится термин «компьютерный ученый», потому что слишком большое ударение сделано на слове «компьютерный», а компьютер – лишь один из примеров информационного поведения, поэтому я предпочитаю выражение «специалист в области информатики». Конечно, все зависит от того, что вы понимаете под информатикой. Я склонен подразумевать под этим вычисления и обмен данными, причем последний играет очень важную роль.

Каковы ваши задачи как специалиста в области информатики?

Робин: Я пытаюсь создать концептуальную структуру, в рамках которой может осуществляться анализ. Для этого нужно принимать во внимание происходящее в сфере ПО, например, то самое понятие «вездесущих систем», но стараться несколько абстрагироваться от него. Это очень трудно: вы делаете ошибки, выдвигаете неверные идеи, которые мало что дают и слабо развиваются. Нужно найти такие элементарные понятия, которые могут быть расширены и применены для объяснения существующих или проектируемых крупных программных систем.

Думаю, идея общения агентов между собой очень важна, потому что это одно из первых понятий, сформировавшихся в теории вычислений, которое не было ранее исследовано логиками. Идея организованной популяции интерактивных агентов: какой должна быть организация? Как они должны соединяться между собой? Кто с кем может вступать в связь? Могут ли они порождать сами себя? Можно ли им адаптироваться к запросам и поведению своих соседей? Вы сталкиваетесь с огромным кругом вопросов, которые возникают лишь тогда, когда есть целые популяции агентов, а не отдельные программы для решения отдельных задач, как это было на заре программирования.

Поможет ли нам Интернет в поиске решений?

Робин: Думаю, Интернет может стать источником задач. Разобраться в нем – тоже задача. Было бы полезно создать концептуальные инструменты для анализа поведения Интернета: возможно, они помогли бы нам разобраться в других видах популяций агентов, например агентов для мониторинга человеческого тела или управления дорожным движением.

Большая часть того, что происходит в Интернете, отлично спроектирована. Это хороший образец для исследований, потому что он исключительно хорошо работает практически, так что я считаю, что здесь мы находим и решения, и задачи.

Каким вам представляется сегодня исследовательское поле компьютерных наук?

Робин: Я вижу его очень обширным. Создатели крупных систем не соблюдают их строгую спецификацию. Кроме того, те, кто работают на более низком или, если хотите, более формальном уровне, часто углубляются в чистую математику, не учитывая существующие реалии, в результате чего сообщества раздваиваются. Фактически, вы встречаете целый спектр представлений на пути от переднего края прикладных занятий к теоретическим обобщениям, и в каждом месте его обнаружите тех, кто плохо понимает своих соседей как справа, так и слева.

Мне кажется, что оно довольно разобщено. Мы разработали «фундаментальную задачу» в вычислительной области для Великобритании, под названием «Вездесущие компьютеры: эксперименты, проектирование и наука» («Ubiquitous computing: Experience, Design and Science»). Под экспериментами мы понимали разнообразные случаи применения инструментов в различных средах. Например, можно оборудовать здание компьютерами, регистрируя появление людей и их перемещения в каждом помещении. При этом одни люди будут экспериментировать с разными вариантами конструкции; другие будут в это время реализовывать проекты в соответствии с надлежащими инженерными прин-

ципами; ниже находятся ученые, устанавливающие связь инженерных работ с более абстрактными моделями, в то время как инженеры используют их изыскания в качестве инструментов для экспериментов в реальном мире. С помощью этих трех уровней – эксперимента, проекта и науки – делается попытка соединить те разрозненные части, о которых я говорил.

Я стал общаться с людьми, с которыми обычно не имею дела, – теми, кто размышляет о социальных последствиях появления вездесущих систем, и оказывается, что они представляют себе эти системы не как системы, используемые человеком, а как системы, в которых люди являются компонентами. Исходя из этих весьма конкретных уровней понимания систем, нужно проложить обратный путь к инженерным принципам и далее к некоторым понятиям, которые можно взять за основу анализа всей вещи в целом.

Видите ли вы какую-нибудь разницу между тем, как ведутся исследования сегодня, и тем, как они велись в 1960–1970-х годах?

Робин: Значительно вырос интерес к вездесущим системам, которые гораздо шире станут внедряться в наше окружение, и в этом большая разница. Вспомните, в частности, важнейшие программы реального времени, управляющие транспортными средствами и различными важными механизмами. Проверка правильности работы ПО реального времени требует совсем других аналитических инструментов – ранее вычисления в реальном времени были предметом меньших забот.

Несомненно, что сегодня разработчиков Аэробуса или каких-либо внутренних систем весьма интересует происходящее в реальном времени – как в физике, где вы знаете продолжительность событий.

Приведут ли вездесущие системы к успехам или революциям в области ИИ?

Робин: Да, но мне кажется, что подход к искусственному интеллекту (ИИ) должен быть косвенным. Мне не очень нравилось особое отношение к ИИ в 1960–1970-х. По-моему, на него возлагались порой чрезмерные надежды.

Используя при изучении популяций агентов такие слова, как «доверие» и «знание», мы начинаем рассматривать ИИ не как то, что либо существует, либо нет, а как то, к чему можно постепенно приблизиться. Системы становятся все более интеллектуальными и, так сказать, «рефлексирующими», то есть способными сообщать о том, что они делают, и анализировать свое поведение, и так постепенно по мере разработки этих систем в них в малых или даже больших количествах будет встречаться то, что считается принадлежностью ИИ.

Не уверен, что вся та работа, которая была проделана в области ИИ, пойдет на пользу. Думаю, при проектировании больших систем мы все чаще будем описывать события с помощью таких гуманитарных понятий, как «вера». Тогда задача определения того, обладает ли существо интеллектом, теряет свою жесткость, поскольку появляются градации.

Есть ли какие-то извлеченные из исследований уроки, которые не были усвоены?

Робин: Большинство языков программирования разработано без предварительного выбора теории, на которой должен базироваться смысл (meaning). Поэтому часто разрабатывают и реализовывают язык, не задумываясь о его смысле, не предвидя, что должно происходить при запуске любой программы. Конечно, в некоторых случаях предвидение было замечательным, например в Алголе-60: его описание 1960 года было настолько точным, что, пользуясь им, можно было проследить, что должно произойти. Так бывает не всегда. Даже в хороших языках формальные основы могут отсутствовать, пока не будет готов язык, поэтому усовершенствованием теории смысла языка занимаются уже позднее, а это может свидетельствовать о том, что при его проектировании не были использованы преимущества теоретических знаний.

Другим примером модернизированного анализа служат крупные программные системы. В Великобритании есть масса примеров невообразимого затягивания сроков и отдельных провалов. Для больших систем значительную пользу принесли бы строгие спецификации и некоторый вид научного анализа.

Что вы понимаете под «теорией смысла» языка программирования?

Робин: Это теория того, что должна делать реализация. У ML есть теория смысла, потому что исходя из действующей в нем семантики я могу доказать, что висячих ссылок быть не может. В последнее время доказывают различные вещи, касающиеся семантики Си, что создает теорию смысла Си. У вас есть семантика Си, и вы доказываете некоторые теоремы о любых программах на Си, использующих эту семантику. Недавно здесь были достигнуты большие успехи. Думаю, дело сдвинулось с места.

Для языка программирования имеются в виду спецификация и конструкция самого языка. Помогают ли они предотвратить какие-то ошибки пользователей этого языка?

Робин: Да, это значит, что ошибки пользователя можно выявить сравнением со спецификациями и найти несоответствия, прежде чем программа будет запущена или начнет использоваться на практике.

Я сейчас работаю над теорией поведения популяций взаимодействующих агентов. Она должна предсказать, как популяция, состоящая из

людей и механизмов, сможет существовать в искусственной среде, обмениваясь данными друг с другом. Та же теория, надеюсь, поможет понять биологические системы – например, как одна клетка создает новую клетку в виде некоего пузырька на своей поверхности.

Возможно, удастся создать обобщенную науку информатики, не зависящую от конкретных приложений. Прежде чем браться за язык программирования, хотелось бы иметь теорию, которой вы будете руководствоваться при разработке языка. Я хочу создать такую теорию, пока язык еще не оформился.

10

SQL

Если у вас есть большой набор структурированных данных, то как обеспечить эффективный способ сбора, извлечения и обновления информации, когда известно, какого рода операции потребуются? Эту базовую идею реляционной модели придумал Э. Ф. (Тед) Кодд. SQL – самое наглядное представление реляционной модели в виде декларативного языка, в котором описывается, что вам нужно, а не как это сделать. Дональд Чемберлен (Donald Chamberlin) и Рэймонд Бойс (Raymond Boyce) разработали SQL на основе идей Кодда.

Основополагающая статья

Как был придуман SQL?

Дон Чемберлен: В начале 1970-х широкое распространение интегрированных баз данных только начиналось. Развитие технологии и экономики впервые позволило предприятиям рассматривать свои данные как корпоративный ресурс, совместно используемый различными приложениями. Эта новая точка зрения на данные открыла возможность разработки технологий управления данными нового поколения.

В 1970-х основным продуктом IBM для работы с базами данных был IMS, но помимо группы разработчиков IMS проблемы баз данных изучали небольшие группы исследователей на нескольких площадках IBM. Руководителем одной из таких групп, находившейся в исследовательской лаборатории IBM в Сан-Хосе (Калифорния), был д-р Э. Ф. (Тед) Кодд. Мы с Рэем Бойсом входили в другую небольшую группу, находившуюся в Уотсоновском исследовательском центре IBM в Йорктаун-Хейтс (Нью-Йорк). Мы с Рэем изучали языки запросов к базам данных, пытаясь усовершенствовать те языки, которые были распространены в то время.

В июне 1970 года Тед Кодд опубликовал основополагающую статью,[1] введя реляционную модель данных и описав ее достоинства для независимости данных и разработки приложений. Статья Кодда привлекла к себе большое внимание как в IBM, так и за ее пределами.

Мы с Рэем Бойсом участвовали в симпозиуме по реляционной модели данных, который Кодд организовал в Уотсоновском исследовательском центре в 1972 году. Этот симпозиум послужил «обращению» меня и Рэя в новую веру. На нас произвели большое впечатление элегантность и простота хранения данных в реляционном виде, и мы увидели, что в реляционной форме легко выразить многие виды запросов. После симпозиума мы с Рэем начали «игру в запросы», соревнуясь в придумывании языков, которые были бы достаточно гибкими, чтобы описывать многие типы запросов.

В 1973 году идеи Кодда заняли такое видное положение, что IBM решила объединить исследования по базам данных, переведя их в одно место – к Кодду в Сан-Хосе, и разработать промышленный прототип под названием System R для проверки реляционных идей. Рэй Бойс и я вместе с рядом других исследователей IBM из Йорктауна и Кембриджа переехали в Калифорнию, чтобы присоединиться к команде System R. Так как мы с Рэем интересовались языками, нашей первой задачей

[1] Codd, E. F. «A Relational Model of Data for Large Shared Data Banks», Communications of the ACM, June 1970.

было проектирование языка запросов, который стал бы интерфейсом пользователя для System R. Мы изучили реляционные языки, предложенные Коддом и другими исследователями, и поставили перед собой следующие цели:

- Мы хотели создать язык, ключевые слова которого были бы обычными английскими словами, чтобы их можно было легко набирать на клавиатуре. Мы хотели, чтобы в основе языка были такие привычные понятия, как строки и столбцы. Так же, как в первоначальных предложениях Кодда по языку, наш язык должен был быть декларативным, а не процедурным. Мы хотели воспользоваться преимуществами реляционного метода и в то же время избежать математических понятий и терминологии, таких как универсальные квантификаторы или реляционные операторы деления из ранних работ Кодда. Мы также хотели включить некоторые понятия запросов высокого уровня – такие как группировка, – которые, по нашим представлениям, было затруднительно выразить с помощью других реляционных языков.
- Мы хотели не ограничиваться запросами, снабдив язык и другими функциями. Самое очевидное расширение – ввести операции для вставки, удаления и изменения данных. Кроме того, мы хотели включить в язык операции, традиционно выполняемые администратором базы данных, такие как создание новых баз и представлений, управление доступом к базе данных, задание ограничений и триггеров, поддерживающих целостность базы данных. Мы хотели, чтобы все эти задачи решались единообразными синтаксическими средствами. Мы хотели, чтобы авторизованные пользователи могли выполнять административные действия, такие как задание новых представлений, не останавливая систему и не применяя специальные инструменты. Иными словами, мы рассматривали запросы, обновление и администрирование базы данных как отдельные аспекты единого языка. В этом отношении у нас открывались уникальные возможности, потому что наши пользователи создавали свои базы данных с чистого листа и не ограничивались требованиями обратной совместимости.
- Мы хотели, чтобы наш язык можно было применять как автономный язык запросов для поддержки принятия решений, а также в качестве языка разработки для более сложных приложений. Последняя задача потребовала от нас найти интерфейсы из нашего языка к различным распространенным языкам программирования.

Частично основываясь на нашем прежнем опыте «игры в запросы», мы с Рэем разработали начальное предложение для языка реляционных запросов под названием SEQUEL (Structured English Query Language)

и опубликовали его на 16 страницах[1] доклада для ежегодной конференции ACM SIGFIDET (предшественницы SIGMOD) в мае 1974 года (та самая конференция, на которой произошел известный спор между Тедом Коддом и Чарльзом Бахманом). Вскоре после публикации этой исходной статьи Рэй Бойс внезапно трагически скончался в результате аневризмы головного мозга.

После публикации первоначального предложения SEQUEL этот язык прошел через этапы проверки и уточнения, продлившиеся примерно с 1974 по 1979 годы. В этот период SEQUEL был реализован как часть экспериментального проекта базы данных System R в исследовательской лаборатории IBM в Сан-Хосе. В System R изучались разные аспекты управления базами данных, включая индексы B-дерева, методы объединения, оптимизацию на основе затрат и непротиворечивость транзакций. Опыт данной реализации оказал влияние на развитие языка. На SEQUEL также повлияли оценки трех клиентов IBM, которые установили прототип System R и экспериментально работали с ним. Команда System R раз в квартал встречалась с командами клиентов и обсуждала с ними возможности совершенствования языка и его реализации.

Язык SEQUEL заметно развился на протяжении работы над проектом System R. Название языка сократилось от SEQUEL до SQL, чтобы избежать конфликта с зарегистрированной торговой маркой. Было добавлено универсальное средство `JOIN`, отсутствовавшее в первоначальном предложении. Были улучшены средства группировки, и в группы фильтрации добавилось предложение `HAVING`. Для работы с отсутствующими данными были введены значения `NULL` и трехзначная логика. Был добавлен ряд новых предикатов, включая `LIKE` для поиска частичного соответствия и `EXISTS` для проверки непустоты результата подзапроса. Были опубликованы новые статьи, документирующие эволюцию языка.[2, 3] На этом этапе разработки языка решения обычно принимались на прагматической основе в соответствии с нашим опытом реализации и пожеланиями наших экспериментирующих пользователей.

---

[1] Chamberlin, Don and Ray Boyce. «SEQUEL: A Structured English Query Language», Proceedings of ACM SIGFIDET (precursor to SIGMOD) Conference, Ann Arbor, MI, May 1974.

[2] Chamberlin, Don et al. «SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control». IBM Journal of Research and Development, November 1976.

[3] Chamberlin, Don. «A Summary of User Experience with the SQL Data Sublanguage». Proceedings of the International Conference on Databases, Aberdeen, Scotland, July 1989.

Этап исследований SQL в IBM завершился в 1979 году с окончанием проекта System R. После этого ответственность за язык перешла к командам разработчиков, которые превратили прототип System R в коммерческие продукты на разных платформах IBM. Однако первый коммерческий продукт на базе SQL, появившийся в 1979 году, был выпущен не IBM, а маленькой компанией Relational Software, Inc. Продукт назывался Oracle, и это имя позднее приняла сама компания, которая недолго оставалась маленькой. За Oracle вскоре последовали реализации SQL, выполненные IBM и фактически всеми основными производителями баз данных. Сейчас SQL – самый широко используемый в мире язык запросов к базам данных.

Чтобы способствовать переносимости приложений между различными реализациями SQL, Национальный институт стандартизации (ANSI) учредил проект по разработке стандартной спецификации SQL. Результат, названный Database Language SQL, был опубликован в качестве стандарта ANSI в 1986 году[1] и как стандарт ISO – в 1987 году.[2] На основе стандарта SQL продолжалось развитие SQL по мере добавления в язык новых функций в соответствии с меняющимися требованиями. Новые версии стандарта SQL публиковались ISO в 1989, 1992, 1999, 2003 и 2006 годах.

Эдин Фалкофф и Кен Айверсон работали над APL. Как и ваша работа над SQL, их работа основывалась на описании строго определенной модели. Полезен ли формализм, такой как нотация Айверсона или реляционная модель Кодда, для создания успешного языка программирования?

Дон: Полагаю, что реляционная модель данных имела фундаментальное значение для разработки SQL. Любому языку программирования, получающему детерминированные результаты, необходим четко определенный набор объектов и операторов, который можно назвать формальной моделью данных. Это основа детерминированного программирования.

Взглянув даже на языки со свободной типизацией, такие как Python, вы обнаружите, что их основа – хорошо определенная модель данных. Она более гибкая, чем реляционная, но должна быть корректно определена, чтобы послужить основой для семантики языка.

---

[1] American National Standards Institute. «Database Language SQL», Standard No. X3.135 (1986 и последующие обновления).

[2] International Organization for Standardization «Information Technology – Database Language SQL», Standard No. ISO/IEC 9075 (1987 и последующие обновления).

Если бы мне пришлось создавать новый язык, что бы вы посоветовали: начать со строгой модели данных или же вносить ее в язык по мере его роста?

Дон: В принципе, можно действовать обоими способами, но не всегда есть возможность разрабатывать новую модель данных параллельно с определением нового языка. Например, разработчики XQuery не изобретали XML – им пришлось работать с моделью данных, уже определенной в XML Schema и других стандартах W3C.

Язык

Как у вас возник интерес к языкам запросов?

Дон: Меня всегда интересовали языки.

Вы говорите на каких-то других языках (кроме английского)?

Дон: Нет, я не говорю ни на каких других естественных языках, но люблю читать и писать, и считаю языки захватывающей темой. Мне очень повезло с карьерой, я оказался в нужное время в нужном месте, когда у Теда Кодда возникли его фундаментальные идеи реляционной модели базы данных. Участие в проекте, столь повлиявшем на исследования в области реляционных баз данных, было счастливой случайностью. Интерес к языкам, который у меня был, помог мне найти свою нишу в этом проекте, и я счастлив, что мне предоставилась такая возможность.

Одним из первых проектных решений стал выбор декларативного, а не процедурного типа для SQL. Чем вы руководствовались в этом выборе?

Дон: Для этого было несколько причин. Во-первых, мы хотели, чтобы язык был оптимизируемым. Если пользователь подробно опишет все шаги, выполняемые алгоритмом при обработке запроса, у оптимизатора не будет свободы для внесения каких-либо изменений. Декларативный язык значительно удобнее для оптимизации, чем процедурный язык низкого уровня.

Вторая причина заключалась в том, что мы очень стремились к независимости данных, что означало возможность для администраторов добавлять и удалять индексы, изменять структуру данных и создавать новые представления данных. Нужно было обеспечить возможность писать такие приложения, которые не зависели бы от физической организации данных и маршрутов доступа на физическом уровне хранения. Таким образом, независимость данных является второй важной причиной, по которой нам требовался декларативный язык.

Третья причина была связана с продуктивностью пользователей. Мы считали, что пользователям будет легче описать свои намерения на высоком уровне с помощью знакомой им терминологии, а не выражать запросы на языке низкоуровневых машинных понятий, которые им известны хуже.

Поэтому мы подумали, что у декларативных языков есть заметные преимущества для обеспечения оптимизации, независимости данных и продуктивности пользователей.

Было ли в то время такое мнение общим в вашей группе?

Дон: Думаю, общие преимущества декларативных языков были достаточно ясны, но существовала значительная неуверенность в том, что такой сложный декларативный язык, как SQL, можно реализовать, обеспечив производительность, необходимую для коммерческих приложений.

Представления (views) скрывают физическую организацию данных, хранящихся на дисках. Ставилась ли в то время задача дать пользователям возможность работать с данными посредством представлений, вместо непосредственной работы с таблицами?

Дон: Мы считали, что представления станут широко применяться для запросов данных, поскольку разным представлениям требуются свои способы доступа к данным. Например, приложения могут рассматривать данные на разных уровнях объединения и могут иметь права доступа к разным частям данных. Представления дают очень естественный способ реализации таких различий при доступе к данным.

С другой стороны, изменение данных оказывается гораздо более сложной задачей, потому что если оно выполняется через представление, система должна отразить ваши изменения в хранящихся данных. В одних случаях это можно сделать, а в других отображение на данные оказывается неоднозначным. Например, запрос может отобразить в представлении среднюю зарплату по каждому отделу, но если вы попробуете изменить данные в этом представлении, то я не понимаю, что значит изменить среднюю зарплату в отделе. Поэтому выяснилось, что представления гораздо шире используются в приложениях, выполняющих запросы, а не модификации.

SQL стал одним из первых языков, решавших задачу параллельного доступа к общим данным. Как эта задача повлияла на конструкцию языка?

Дон: Поддержание целостности базы данных в условиях одновременного обновления было одной из главных тем исследований проекта System R в исследовательском центре IBM. В результате проделанной

работы появилось строгое определение свойств «ACID» для электронных транзакций, за которое Джим Грей получил в 1999 году премию Тьюринга. Такие свойства транзакций поддерживались в System R (и других реляционных системах) с помощью системы блокировок и журналов, почти незаметной для пользователей SQL.

Параллельный доступ к общим данным осуществляется в SQL преимущественно на основе идей транзакций и уровней изолированности (изолированность – это «I» в аббревиатуре ACID). Уровни изолированности позволяют разработчикам приложений находить компромисс между защитой пользователей друг от друга и увеличением числа одновременно поддерживаемых пользователей. Например, если приложение составляет статистический отчет, ему можно задать низкий уровень изолированности, чтобы не блокировать значительные участки базы данных. Для банковской операции можно задать высокий уровень изолированности, чтобы гарантировать упорядоченность всех транзакций, изменяющих данный счет.

Параллельные обновления могут проявляться для программистов SQL в виде потенциальных взаимных блокировок. При определенном стечении обстоятельств две одновременные транзакции SQL могут взаимно блокировать друг друга; в этом случае одна из транзакций получит код возврата, свидетельствующий о том, что был произведен откат ее операций.

Я читал об интересной истории, связанной с Хеллоуином и System R, в которой участвовали Пэт Селинджер и Мортон Астрахан.

Дон: Что касается праздника Хеллоуина, то, кажется, в 1975 году Пэт Селинджер и Мортон Астрахан работали над оптимизатором для первой реализации SQL, который должен был выбирать маршрут доступа при выполнении группового обновления данных – как, например, если нужно повысить зарплату всем низкооплачиваемым служащим. Сначала Пэт и Мортон думали, что для выявления служащих, получающих жалованье ниже заданной границы, будет эффективен индекс по атрибуту «зарплата». Поэтому оптимизатор с помощью индекса находил тех, кому нужно повысить зарплату, и попутно изменял ее. Однако было замечено, что когда зарплата служащего меняется, он перемещается в индексе на новое место, и при дальнейшем просмотре может встретиться снова и получить еще одну прибавку. Из-за этого сначала получались некорректные и непредсказуемые результаты.

Пэт и Мортон обнаружили эту проблему в пятницу днем в Хеллоуин, американский праздник. Пэт зашла ко мне в кабинет и спросила: «Что будем делать?» Я ответил: «Пэт, уже конец пятницы, мы не сможем решить эту проблему сегодня. Назовем ее „проблемой Хеллоуина“ и разберемся с ней на следующей неделе».

Каким-то образом это название закрепилось за проблемой, когда нельзя обращаться к данным по индексу, если поле для этого индекса подлежит изменению. Это проблема, которую должны решать все оптимизаторы баз данных, и потому ее название стало широко известным в нашей отрасли.

Какие уроки из опыта изобретения, дальнейшего развития и распространения вашего языка могли бы извлечь разработчики компьютерных систем в настоящем и будущем?

Дон: Думаю, история SQL иллюстрирует необходимость иметь ряд конкретных принципов, которыми вы будете руководствоваться при разработке языка. Я перечислил ряд принципов, которые (задним числом) считаю важными для проектирования языка программирования. Не стану утверждать, что все эти принципы строго соблюдались при разработке SQL; на самом деле, как многие заметили, некоторые из них как раз указывают на слабые места в начальном проекте SQL. В значительной мере эти первоначальные недостатки были преодолены при дальнейшем развитии языка.

Вот мой список принципов. Многие из них просто выглядят соображениями здравого смысла, но применять их на практике сложнее, чем может показаться.

Замкнутость

Язык должен быть определен в терминах модели данных, состоящей из набора объектов с четко определенными свойствами. Для каждого оператора языка должны быть определены его операнды и результат в виде объектов модели данных. Семантика каждого оператора должна определять результат действия этого оператора на все свойства участвующих в его работе объектов.

Полнота

Для каждого объекта модели данных должны иметься операторы, которые его создают, разлагают на элементарные части (если они есть) и сравнивают с другими объектами того же типа.

Ортогональность

Понятия языка должны определяться независимо и не подвергаться действию особых правил, ограничивающих их применение. Например, если скалярное значение входит в понятия языка, то любое выражение, возвращающее скалярное значение, должно быть допустимо к использованию всюду, где предполагается участие скалярного значения.

Единообразие

Стандартные задачи, такие как извлечение компонента структурированного объекта, должны решаться единообразно, в какой бы части языка они ни встретились.

Простота

Язык должен определяться через небольшое число относительно простых понятий. Разработчикам следует избегать соблазна добавления функций для особых целей. Если язык окажется удачным, сохранение его простоты потребует дисциплины и решимости отвергнуть многочисленные требования «усовершенствований». Эта постоянная борьба облегчается при наличии в языке хорошего механизма расширений (см. следующий пункт).

Расширяемость

У языка должен быть четко определенный универсальный механизм добавления новой функциональности, в идеале слабо или никак не влияющий на синтаксис. Например, язык запросов к базам данных может предоставлять средства для добавления пользовательских функций, написанных на отдельном языке программирования, полном по Тьюрингу.

Абстрагирование

Язык не должен демонстрировать особенности конкретной реализации или зависеть от нее. Например, удаление дубликатов из группы значений должно описываться в терминах абстрактной концепции, такой как «первичный ключ», а не физической стратегии, такой как «уникальный индекс». (В ранних версиях SQL такая ошибка присутствовала.) В области баз данных это понятие иногда называют «независимостью данных».

Возможность оптимизации

Язык не должен налагать ненужные ограничения на алгоритмы выполнения его выражений. Например, определение языка должно допускать некоторую свободу в порядке вычисления предикатов. Семантическая спецификация языка должна по возможности быть декларативной, а не процедурной, чтобы предоставить возможность автоматической оптимизации. Иногда лучше смириться с некоторой недетерминированностью (например, при обработке какого-то запроса ошибка может генерироваться или нет в зависимости от порядка вычисления предикатов).

Эластичность

Не всегда программы бывают корректны. Язык нужно проектировать так, чтобы значительную часть программных ошибок можно

было обнаружить и четко идентифицировать на этапе компиляции (то есть в отсутствие реальных входных данных). Кроме того, язык должен обеспечить механизм, позволяющий программистам обрабатывать исключительные ситуации во время исполнения.

Обратная связь и развитие

Исходная работа Теда Кодда с описанием реляционной модели данных была открыто опубликована, повлияв на многих за стенами IBM, например на Ларри Эллисона и группу Майка Стоунбрейкера в университете Беркли. Был ли этот процесс похож на модель open source? Как публичность повлияла на разработку SQL?

Дон: В 1970-х реляционная модель была новой идеей. Она стала предметом передовых исследований и прототипирования, и в целом не была доступна на рынке. SQL разрабатывался в рамках экспериментального исследовательского проекта System R, который не был обычным процессом разработки продукта в IBM. Исследовательское подразделение IBM традиционно публикует результаты своей работы в открытой литературе, и в отношении языка SQL и других элементов System R мы поступили так же.

Мы не публиковали исходный код своей реализации SQL, в этом коренное различие с действующей ныне моделью программного обеспечения с открытым исходным кодом. Мы не передавали в другие руки никаких программ, но описали некоторые интерфейсы и методы, использованные в нашей экспериментальной реализации SQL. Например, некоторые из наших приемов оптимизации были описаны в открытой литературе, и как вам известно, некоторые из этих статей оказали влияние на других людей в нашей отрасли, разрабатывавших аналогичное программное обеспечение.

Этот процесс раскрытия идей не был односторонним. На ранней стадии исследований по реляционным базам идеи свободно распространялись в ряде организаций, включая IBM, Калифорнийский университет в Беркли и другие – к их общей пользе.

Почему SQL стал популярен?

Дон: Думаю, главная причина популярности SQL заключается в силе и простоте реляционной модели данных Теда Кодда. Кодд осуществил теоретический прорыв, который революционизировал управление базами данных, а SQL был просто попыткой представить идеи Кодда в доступном формате. В сравнении с другими существующими технологиями реляционные базы данных обеспечили качественный скачок в про-

дуктивности работы по созданию и сопровождению приложений для работы с базами данных.

Конечно, SQL не был единственным основанным на идеях Кодда языком, предложенным в 1970-х годах. Думаю, некоторые специфические причины популярности SQL включают в себя следующие:

- Тот факт, что SQL поддерживал полный набор инструментов для администрирования баз данных, имел важное значение для успеха языка. С помощью SQL любой обладающий достаточными правами пользователь в любой момент посредством простых команд мог создавать или удалять таблицы, представления и индексы. Такие задачи традиционно требовали вмешательства администратора и остановки баз данных, стоили дорого и выполнялись долго. SQL сделал конечных пользователей независимыми от администраторов баз данных и позволил им легко экспериментировать с вариантами структуры баз данных.
- SQL достаточно прост для изучения. Подмножество SQL, необходимое для решения простых задач, можно освоить за несколько часов. По мере надобности пользователи могут подключать более сложные и мощные разделы языка.
- SQL был доступен в надежных многопользовательских приложениях по крайней мере двух производителей (IBM и Oracle), на многих платформах, включая OS/370 и UNIX. По мере роста популярности языка количество новых реализаций росло как снежный ком.
- SQL поддерживал интерфейсы к распространенным языкам программирования. Опираясь на эти языки, SQL обретал способность поддержки сложных приложений.
- Рано проведенная ANSI и ISO работа по стандартизации SQL дала пользователям уверенность, что их приложения на SQL будут переносимы между разными реализациями. Эта уверенность выросла еще больше, когда Национальный институт стандартов и технологий разработал тест на совместимость с SQL. Некоторые правительственные организации США требовали, чтобы закупаемые ими базы данных соответствовали федеральному стандарту обработки информации FIPS-127 для SQL.
- Разработка SQL происходила в удачное время, потому что как раз тогда многие предприятия создавали или модифицировали важные приложения для работы с объединенными корпоративными базами данных. Разработчиков приложений и администраторов баз данных не хватало. Те организации, которые приняли SQL, получили рост производительности труда, позволявший справиться с затягиванием разработки приложений.

Почему SQL остается популярным?

Дон: Многие популярные 25 лет назад языки фактически исчезли, в том числе те языки, которые поддерживались крупными корпорациями. Думаю, причины сохранения широкой популярности SQL следующие:

- Стандарт ISO SQL обеспечил возможность контролируемого развития языка в соответствии с запросами пользователей. В комиссию по этому стандарту входят как пользователи, так и производители, при этом производители выделили ресурсы для поддержки согласованности своих реализаций с развивающимся стандартом. С течением времени в стандарте SQL были исправлены ошибки, допущенные в первоначальном проекте языка, и добавлены важные новые функции, такие как внешнее объединение, рекурсивные запросы, хранимые процедуры, объектно-реляционные функции и OLAP (аналитическая обработка в реальном времени). Стандарт SQL также послужил сосредоточению ресурсов и интереса отрасли к созданию общей среды, в которой отдельные люди и компании могли разрабатывать инструменты, писать книги, читать учебные курсы и предоставлять консультативные услуги.
- SQL управляет постоянными данными с длительным жизненным циклом. Предприятия, вложившие средства в базы данных, склонны сохранять свои инвестиции, а не начинать все заново, используя новые подходы.
- SQL достаточно надежен, чтобы решать реальные проблемы. Он имеет широкий спектр применения – от анализа деловых данных до обработки транзакций. Его поддерживают многие платформы и рабочие среды. Несмотря на некоторые упреки в отсутствии элегантности, SQL успешно использовался многими организациями для разработки важных практических приложений. Думаю, такой успех отражает то, что вначале язык возник как экспериментальный прототип, учитывавший потребности реальных пользователей. Он также отражает прагматичные решения, принимавшиеся на всем протяжении его развития в соответствии с меняющимися требованиями.

Сегодня на Си пишут системы, которые, пожалуй, на несколько порядков крупнее систем, писавшихся на Си в 1970-е, но объемы обрабатываемых данных выросли за это время еще больше. Несколько строк SQL сохраняют способность работать с набором данных при его росте до невообразимых размеров: возникает впечатление, что SQL лучше справляется с увеличением объема данных. Так ли это и почему?

Дон: Возможно, это проистекает из дополнительного преимущества декларативных языков, делающего их более восприимчивыми к парал-

лельным вычислениям, чем процедурные языки. Если вы осуществляете операцию над большим набором данных и она описывается непроцедурным способом, у системы появляется больше возможностей распределить работу между несколькими процессорами. Реляционная модель данных и высокий уровень абстракции, который она поддерживает, оказались очень полезными для подобного масштабирования. SQL в качестве декларативного языка предоставляет компиляторам возможность применить неявный параллелизм.

Получали ли вы в прошедшие годы какие-либо отзывы от пользователей продуктов, основанных на ваших исследованиях?

Дон: С тех пор как в 1980-е главные продукты баз данных IBM начали поддерживать SQL, IBM проводила периодические рецензирования, так называемые «консультативные советы клиентов», на которых мы собирали отзывы пользователей об SQL и разных сторонах наших продуктов баз данных. Есть также независимая группа пользователей IDUG (международная группа пользователей DB2), которая раз в год проводит встречи в Америке, Европе и Азии, и на этих конференциях IDUG происходит широкий обмен информацией между пользователями DB2 и IBM. Значительная часть полученных данных передается в подразделения исследований и разработок IBM и учитывается при планировании дальнейшего совершенствования продуктов. Такие данные способствовали появлению многих новых функций, в том числе объектно-реляционных расширений и расширений OLAP.

Другим источником идей служат комиссии ANSI и ISO по разработке стандартов SQL, состоящие из представителей как пользователей, так и производителей. Эти каналы обратной связи в течение многих лет помогают развивать язык для удовлетворения меняющихся требований пользователей.

Вы также провели ряд проверок юзабилити с психологом Филлис Рейзнер в отношении двух языков, SQUARE и SEQUEL. Что выявили эти проверки?

Дон: Да, Филлис Рейзнер – это специалист по экспериментальной психологии, она работала с группой System R, проверяя заложенные в язык идеи на студентах колледжа. SQUARE – это ранняя попытка создать язык для реляционной базы данных на основе математической нотации, тогда как SEQUEL был аналогичным языком с нотацией, использующей слова английского языка.

Филлис провела эксперимент, в котором преподавала оба эти языка студентам колледжа, чтобы выяснить, какой из них легче в изучении и применяется с меньшим количеством ошибок. В целом, нотация со

словами из английского языка оказалась проще в изучении и использовании, чем математическая.

Любопытно, однако, что обычно ошибки, допускавшиеся студентами, были слабо связаны со структурой языка. Они касались таких вещей, как заключение данных в кавычки, использование заглавных букв – их можно считать тривиальными или нелогичными ошибками, не связанными со структурой языка или данных. Как бы то ни было, правильный подход к таким деталям вызывал у пользователей затруднения.

Сегодня очень распространены атаки на веб-сервисы, такие как SQL-инжекции, связанные с плохой фильтрацией входных данных, включаемых в запросы к базам данных. Что вы думаете по этому поводу?

Дон: Атаки с SQL-инжекцией – хороший пример вещи, которую мы и представить себе не могли в те далекие дни. Мы не предполагали, что запросы будут составляться из данных, вводимых пользователем в веб-броузере. Думаю, вывод из этого – необходимость тщательного изучения пользовательских данных программами перед их обработкой.

Пошло ли развитие SQL какими-нибудь неожиданными для его создателей путями?

Дон: SQL предполагался декларативным, непроцедурным языком, и этот свой характер он сохранил. Но с годами язык стал гораздо сложнее, чем мы себе представляли вначале. Его стали применять для таких задач, о которых мы тогда не помышляли. В него были добавлены такие вещи, как кубы данных и анализ OLAP. Он стал объектно-реляционным языком с типами и методами, определяемыми пользователем. Мы не предполагали появления всех этих новых применений. Сегодня пользователям SQL приходится сталкиваться с более высокой сложностью, и от них требуется большая техническая изощренность, чем мы могли ожидать.

Рэй Бойс и я предполагали, что SQL окажет влияние на индустрию баз данных, но это влияние оказалось не таким, как мы ожидали. Мы с Рэем думали, что разрабатываем язык, который будет использоваться, в основном, «случайными пользователями» для составления специфических запросов в системах поддержки решений. Мы хотели сделать базы данных доступными для нового класса пользователей, не изучавших компьютерные науки. Мы предполагали, что SQL станет непосредственно использоваться финансовыми аналитиками, проектировщиками городов и другими профессионалами, которым нужен доступ к данным, но которые не станут писать программы. Наши расчеты оказались слишком оптимистичными.

С самого начала SQL стали использовать главным образом подготовленные программисты. На самом деле, большой объем кода SQL генерировался автоматическими средствами, чего мы не предвидели. Те не-пишущие-программ профессионалы, которые, по нашим предположениям, должны были непосредственно применять SQL, чаще используют интерфейсы в виде форм, поддерживаемые приложениями с доступом к серверам баз данных SQL. Непосредственный доступ к данным случайных пользователей был отложен до развития таблиц и поисковых механизмов.

Вы работали над реляционными системами, а также над системой обработки документов Quill, которая изолирует пользователей от физического представления документов. Такие электронные таблицы, как Excel, тоже представляют данные очень интуитивным, ориентированным на пользователя способом. Есть ли у этих систем что-то общее? Можно ли расширить такого рода независимость данных, распространив ее на Интернет?

Дон: Quill и Excel поддерживают то, что можно назвать пользовательским интерфейсом прямого манипулирования данными, который позволяет оперировать со зрительным представлением данных, имеющим в своей основе некоторую логическую структуру. Такая метафора оказалась очень сильной. Можно провести аналогию с реляционными базами данных: пользователь оперирует с данными на верхнем уровне абстракции, не зависящим от лежащих в основе структур данных. Интерфейсы для прямого манипулирования легко изучать и применять, но на каком-то уровне они все равно должны опираться на специализированные структуры данных. Для того чтобы отобразить намерения пользователя в сами данные, нужен некоторый оптимизирующий компилятор или интерпретатор.

Что касается Интернета в целом, то он таков, каким получился, и в данный момент мы не сможем спроектировать его заново, но все популярные поисковые механизмы скрывают от пользователей детали обработки запросов на получение информации. Я уверен, что поисковые механизмы будут развиваться в сторону поддержки более высоких уровней абстракции и выяснения и использования семантики информации, находящейся в Интернете.

Вы хотели разработать инструмент, который полезен обычным пользователям, но работают с ним в основном программисты.

Дон: Мне кажется, мы были несколько наивно оптимистичны в выборе целей, начиная проектировать язык. Я работал с Тедом Коддом, когда реляционная модель только появилась, и в те дни Тед работал над про-

ектом Rendezvous – системой вопросов/ответов на естественном языке, основанной на реляционной модели. Я тогда считал, что использовать естественный язык в полной мере нереально, но надеялся, что мы сможем создать достаточно понятный человеку интерфейс, чтобы им можно было начать пользоваться после недолгого обучения.

В основном, как мне кажется, это получилось. SQL быстро развился до такого уровня сложности, что стал языком программирования и потребовал объема подготовки, сравнимого с другими языками программирования, поэтому его используют преимущественно профессионалы.

Я с огромным уважением отношусь к новейшим веб-приложениям вроде Google, с помощью которых можно получать полезную информацию, не имея вообще никакой подготовки. В 1970-х у нас просто не было соответствующих технологий.

В чем сложность – в объяснении того, как работает SQL, или в изложении идей реляционной модели, к чему не все могут быть готовы?

Дон: Думаю, оба эти фактора привели к необходимости определенного уровня технической подготовки для пользователей SQL. Еще один фактор связан с различием между точными и неточными запросами.

Кидая Google кучу слов для поиска, вы готовы принять неточный результат. Иными словами, Google старается найти документы, наиболее релевантные вашему списку слов. Это недетерминированный процесс, результат которого в большинстве случаев весьма полезен.

В SQL мы решаем задачи иного рода, в которых ответы детерминированы, а для детерминированных ответов нужен язык запросов с более высокой степенью точности. Например, нужно очень четко представлять себе разницу между «и» и «или». В Google семантика запросов может позволить себе большую расплывчатость, чем в области структурированных запросов, где применяется SQL.

Стоимость ошибки или стоимость получения неточного ответа в результате поиска в Интернете значительно ниже стоимости получения ошибочных значений зарплаты для ваших служащих.

Дон: Верно, а если вы допустите ошибку в написании или забудете, как называется столбец таблицы, по которой нужно выполнить объединение, то ваш запрос SQL вообще не выполнится, тогда как менее детерминированные интерфейсы вроде Google гораздо снисходительнее относятся к таким ошибкам.

Вы верите в важность детерминизма. Если я пишу строку кода, я должен понимать, что она будет делать.

Дон: Ну, для одних приложений детерминизм важен, а для других – нет. Традиционно существовала граница между тем, что можно назвать базой данных, и тем, что можно назвать извлечением информации (information retrieval). Обе сферы процветают, и каждая находит соответствующее применение.

XQuery и XML

Окажет ли в будущем XML влияние на способы работы поисковых механизмов?

Дон: Возможно. Поисковые механизмы уже используют ряд метаданных, имеющихся в тегах HTML, например в гиперссылках. Как известно, XML – более широкий язык разметки, чем HTML. С появлением большего числа основанных на XML стандартов для разметки специальных типов документов, таких как медицинские и деловые, поисковые механизмы могут научиться извлекать пользу из семантической информации, содержащейся в такой разметке.

Вы сейчас работаете над XQuery – новым языком для доступа к данным XML. XML отличается от реляционных данных тем, что содержит метаданные. С какими проблемами вы столкнулись, проектируя язык запросов для XML?

Дон: Одна из самых сильных сторон XML состоит в том, что документы XML самоописательны. Благодаря этому документы XML могут иметь разную структуру, а различия можно увидеть, прочтя метаданные в форме тегов XML, входящих в сами документы. Благодаря этому XML оказывается очень богатым и гибким форматом представления информации. В современных бизнес-приложениях, где происходит обмен документами, которые могут не совпадать по структуре, внутренние метаданные, входящие в формат XML, имеют очень большое значение. Одна из главных задач языка XQuery – использовать эту гибкость, чтобы запросы могли оперировать с данными и метаданными одновременно.

Одна из проблем, с которыми мы столкнулись при проектировании XQuery, – наличие многих различных окружений, где должен использоваться этот язык. Есть приложения, где типы очень важны. В этих приложениях нужен строго типизированный язык, выполняющий множество проверок типов и генерирующий ошибку, если объект не соответствует предполагаемому типу. Но есть другие среды, иногда называемые средами Schema Chaos, где тип данных менее важен. В таких средах вы можете быть готовы принять данные неизвестного или раз-

нородного типа и потребовать, чтобы язык обладал достаточной гибкостью для работы с данными многих различных типов.

Трудно было спроектировать язык так, чтобы он мог охватить широкий спектр применений от строгой типизации до слабой. Кроме того, система типов XML Schema гораздо сложнее системы типов реляционной модели данных, и проектирование языка для использования с такой сложной системой типов представляет собой очень трудную задачу.

В результате появляется язык, который сложнее SQL. Думаю, изучить XQuery труднее, чем SQL, но, справившись с этой сложностью, вы обретете возможность работать с более богатым и более гибким форматом данных, предлагаемым XML.

Известно, что вы участвовали в стандартизации двух языков запросов, SQL и XQuery. Что вам дал этот опыт?

Дон: Во-первых, я узнал, что стандарты очень ценны для создания формального определения языка, получения реакции пользователей и создания механизма управляемого развития языка в соответствии с меняющимися требованиями. Процесс стандартизации сводит вместе людей с разными точками зрения и специальными знаниями. Совместная работа при этом отличается медлительностью, но я считаю, что в итоге обычно получается относительно надежное определение языка.

Как показывает мой опыт, для обеспечения эффективности работы комиссии по стандартизации языка важны следующие правила:

- Во время разработки языка комиссия должна постоянно иметь синтаксический анализатор ссылок и использовать его для проверки всех примеров и случаев применения, имеющихся в спецификации языка и связанных с ней документах. Удивительно, сколько ошибок обнаруживается с помощью такой простой процедуры. Анализ ссылок обнаруживает проблемы юзабилити и реализации и гарантирует, что в грамматике языка не появятся неоднозначности и другие аномалии.
- Комиссия должна создать формальный набор примеров применения, иллюстрирующих предполагаемое использование языка. Эти примеры полезны для изучения альтернативных подходов в процессе разработки и в итоге могут стать образцами «лучших приемов» работы с языком.
- Определение языка должно подкрепляться набором тестов соответствия, и до принятия стандарта необходимо иметь хотя бы одну эталонную реализацию, прошедшую проверку соответствия. Такая практика позволяет выявить «крайние» случаи и гарантировать,

что семантическое описание языка является полным и однозначным. Ценность стандарта без объективной меры соответствия невысока.

Какую разницу вы почувствовали при разработке XQuery в сравнении с SQL?

Дон: Некоторую разницу я заметил. При разработке XQuery мы испытывали значительно больше ограничений, чем в случае SQL, и тому есть пара причин.

Одна заключается в том, что с самого начала XQuery интересовал очень многих. Мы разрабатывали язык в контексте международной организации стандартизации с представителями примерно 25 компаний, и все они заранее сформировали свое понимание того, каким должен получиться язык. Работа велась при полной гласности, все наши рабочие проекты публиковались в Интернете. В результате мы получили массу замечаний, большая часть которых оказались полезными.

SQL создавался в совершенно иных условиях. Работа велась очень маленькой группой, она никого не интересовала за пределами IBM, да и внутри компании о ней знали немногие, поэтому у нас было гораздо больше свободы для принятия независимых решений без необходимости объяснять и обосновывать их множеству людей, имеющих свое твердое мнение.

Пожалуй, незаметность порой очень раскрепощает.

Дон: Да, у нее много преимуществ!

Влияет ли на результат размер команды?

Дон: Да, я обнаружил, что для меня идеальная численность команды – от 8 до 10 человек. В таком составе можно сделать много, но людей достаточно мало, чтобы каждый знал, чем заняты все остальные, и информация распространялась без особых затрат и трудностей. Примерно такой была численность команды System R, сделавшей первую экспериментальную реализацию SQL.

Какие стимулы лучше всего действуют в команде, занимающейся исследованиями и разработкой?

Дон: Наверное, лучший стимул для такой команды – создание таких условий, чтобы ее работа имела значение. Когда люди видят, что их работа изменит мир, они очень мотивированы и усердно трудятся. Думаю, таким преимуществом часто обладают маленькие стартапы: они создают что-нибудь революционное и не отягощены наследием прошлого.

В крупных компаниях такие возможности встречаются реже, но все же они есть. Лично меня очень воодушевляло участие в пионерской

работе над технологиями реляционных баз данных. Мы видели, что наша работа может оказать революционизирующее воздействие. Работа над проектом с таким потенциалом вдохновляет людей на старательный труд.

Что вы считаете успехом в своей области?

Дон: Замечательный вопрос. Я бы сказал, что исследование успешно, если оно оказало длительное воздействие на технологию. Если мы можем разработать теории, или интерфейсы, или методы, которые широко используются и проходят проверку временем, то можно претендовать на то, что наше исследование имеет какую-то ценность.

Один из лучших примеров тому – работа Теда Кодда. Тед выдвинул идеи, достаточно простые для всеобщего понимания и достаточно мощные, чтобы почти 40 лет спустя по-прежнему доминировать в области управления информацией. Немногие из нас могут надеяться на такой успех, но именно так я бы определил идеальный результат исследовательского проекта.

11

Objective-C

Objective-C – это комбинация языков программирования Си и Smalltalk с добавлением поддержки объектов sobject из Smalltalk. Том Лав (Tom Love) и Брэд Кокс (Brad Cox) разработали эту систему в 1980-х. Ее популярность выросла, когда в 1988 году появились системы NeXT Стива Джобса, а в настоящее время она доминирует в Mac OS X Apple. В отличие от других ОО-систем своего времени, в Objective-C использовалась не виртуальная машина, а очень небольшая библиотека времени исполнения. Влияние Objective-C ощущается в языке программирования Java, а Objective-C 2.0 производства Apple находит широкое применение в создании приложений для Mac OS X и iPhone.

Разработка Objective-C

Почему вы решили расширять уже существующий язык, а не создали новый?

Том Лав: Это было вызвано требованиями совместимости в больших организациях. Очень важным было решение о том, чтобы можно было взять программу на Си и пропустить ее через компилятор Objective-C, ничего в ней не меняя. Все, что можно делать в Си, должно было остаться, и все, что можно делать в Objective-C, должно было быть совместимым с Си. Это было крупное и очень важное ограничение. Оно также позволяло легко комбинировать языки.

Почему вы остановились на Си?

Том: Вероятно, потому, что вначале мы проводили исследования на UNIX-системах, программировали на Си и пытались делать вещи, которые трудно делать на Си. В журнале Byte за август 1981 года появился рассказ о том, какие вещи можно делать с помощью Smalltalk, и для большинства он стал открытием. Брэд высказался в том смысле, что все описанные возможности Smalltalk он смог бы добавить в Си.

Мы работали в исследовательской группе ITT и занимались созданием распределенных программных сред для разработчиков систем телекоммуникаций. Мы подыскивали подходящие инструменты для создания набора того, что сегодня назвали бы средствами автоматизированного проектирования, но это было больше, чем просто инструменты CAD.

А сейчас — превосходит ли Objective-C в каких-нибудь отношениях Smalltalk?

Том: Сегодняшний Objective-C и сегодняшние библиотеки весьма отличаются от тех, которые были осенью 1984 или 1983 года, когда была выпущена первая версия. Мы обсуждали ранее вопрос о том, для каких приложений язык подходит, а для каких – нет. Smalltalk – совершенно замечательный язык для изучения объектно-ориентированного программирования, и меня весьма удивляет, почему он не используется шире в академической среде, так как с его помощью прекрасно изучать основные идеи. С другой стороны, если бы мне понадобилось написать новую операционную систему, я не выбрал бы для этого Smalltalk. Для создания исследовательских моделей некоторых видов, или прототипов, или чего-то подобного Smalltalk является отличным выбором. Думаю, у каждого языка есть область задач, в которой уместно его применение, и эти области перекрываются.

И для Objective-C, и для C++ отправной точкой является Си, но направления их дальнейшего развития весьма различны. Какой подход вы предпочли бы сегодня?

Том: Есть удачное направление, и есть тот подход, который Бьерн выбрал для C++. В одном случае появился небольшой, простой и, осмелюсь сказать, элегантный язык программирования, очень четкий и ясный. В другом случае мы имеем весьма некрасивый, запутанный, сложный язык с рядом весьма мучительных особенностей. Думаю, в этом и состоит разница между ними.

Вы считаете, что C++ в каких-то отношениях слишком сложен?

Том: Несомненно.

Он продолжает развиваться. В него до сих пор добавляют новые возможности.

Том: Давайте разберемся. Мне лично нравится работать с достаточно простыми языками. APL – прекрасный язык программирования, потому что он невероятно прост и чрезвычайно силен в определенного рода приложениях. Если мне нужно написать статистический пакет, то APL – классный язык для такой задачи, потому что я не знаю языка, который лучше него справляется с алгебраическими операциями над матрицами. Это всего лишь один пример.

Как вы думаете, почему C++ используется чаще, чем Objective-C?

Том: Потому что за ним стояла AT&T.

Только и всего?

Том: Думаю, да.

Что вы думаете о сегодняшнем Objective-C?

Том: Он до сих пор жив. Этого мало?

В Objective-C 2.0 появилось много интересных возможностей – он явно пользуется поддержкой Apple.

Том: Недавно я разговаривал с человеком, который пишет программы для iPhone. Он сказал, что скачал Developers Kit для iPhone, и тот целиком состоит из Objective-C. Жив язык.

Начиная разработку языка, думали ли вы, что его станут использовать в мобильных телефонах и портативных устройствах?

Том: Мы впервые познакомились с Брэдом, когда я взял его на работу в группу перспективных исследований для телефонной отрасли – ITT. Нам была поставлена задача заглянуть на 10 лет вперед. Один из полученных выводов: мы плохо умеем прогнозировать на 10 лет вперед, в особенности в области ПО. Мы хорошо предсказывали развитие аппаратных технологий на 10 лет вперед, но ошибались на 10 лет в отношении программ. То есть события, которых мы ожидали к 1990 году, в действительности случались на 10 лет позднее.

Даже в конце 1990-х люди продолжали с недоверием относиться к идеям Лиспа и других языков, успешно применявшимся в них уже 30 или 40 лет.

Том: Верно. Конечно, программисты славятся своим оптимизмом. Кроме того, сменяются поколения. Мы разрабатываем PC, разрабатываем PDA, программируемые телефоны, а группы людей, программирующих эти различные устройства, часто оказываются разными. Не получается так, чтобы одни и те же люди работали с одной и той же технологией. В давние времена была традиция, согласно которой с мэйнфреймами, или миникомпьютерами, или ПК, или рабочими станциями оказывались связанными разные группы людей. Каждой группе приходилось самостоятельно открывать одни и те же вещи. И так продолжается до сих пор.

Присмотритесь к участникам конференции по разработке приложений для iPhone. Они совершенно не похожи на тех, кого можно встретить сегодня на конференции по мэйнфреймам или хотя бы по разработке приложений для Windows. А программисты .NET – не просто иного сорта люди, это еще и другое поколение.

С аппаратным обеспечением положение такое же?

Том: Думаю, там нет таких различий в знаниях.

Почему развитие аппаратного обеспечения мы можем предсказывать на 10 лет вперед, а ПО – нет?

Том: У нас есть хорошие данные для аппаратуры, которые можно измерить количественно. Для программ таких количественных мер нет. Как вы, возможно, знаете, я преуспел в разнообразных расчетах. Я люблю числа. И уже лет 30 пытаюсь подсчитывать, сколько времени нужно программисту, чтобы написать класс, сколько времени он его тестирует, сколько тестеров требуется на одного программиста, сколько тестов нужно составить для каждого класса и сколько строк кода умещается в коробке бумаги: 100 000.

Развитие языка

Как вы полагаете, можно ли наращивать и развивать язык?

Том: Можно, но не спеша. Интересный и сложный вопрос возникает в связи с языками, находящимися в частной собственности, либо в открытом доступе, либо в разработке с открытым кодом: их проблемы решаются с известными трудностями. Если существует единственный авторитетный орган, принимающий решения по изменению языка, и эти изменения осуществляются медленно и методично, то это оказывается лучше всего, но не всем нравится платить за компиляторы, которыми

они пользуются, или ежегодно платить за сопровождение, когда компилятор меняется достаточно редко. Мы годами пытались решать эти вопросы. Это одна из проблем, возникающих при попытке разработать и распространить язык, – то же самое, конечно, происходит с операционными системами.

Как вы решаете вопрос о добавлении в язык той или иной функции?

Том: Необходим минимум функций, обеспечивающий максимум функциональности и гибкости.

Вы сказали, что область применения объектно-ориентированных языков несколько ограниченна. Можно ли каким-то образом ослабить эти ограничения?

Том: Какой бы язык вы ни взяли, есть круг систем, где его применение оправданно, и есть системы, которые не входят в этот круг. Тем не менее по-прежнему бывает, что если в некоторых специфических приложениях требуется достичь максимальной эффективности исполняемого кода, то пишут очень компактный код на ассемблере. Сейчас это бывает редко, но существуют физические ограничения, о которых нужно помнить. По моему мнению, как бы ни был широк диапазон возможного применения, всегда можно найти пример задачи, выходящей за его пределы. Я не имел в виду, что объектно-ориентированным языкам присущ какой-то особенно узкий диапазон. Например, если вы создаете электронную бортовую систему для беспилотного воздушного аппарата, являющегося, в сущности, моделью самолета, и на его борту устанавливаются миниатюрный процессор и миниатюрная система, то это совсем другая задача в сравнении с проектированием программного обеспечения для Boeing Dreamliner.

Я понимаю, почему вы выбрали Smalltalk: было очевидно, что это лучший вариант. Даже сейчас это хороший выбор.

Том: Это красивый язык. Я знаком с некоторыми более ранними языками, например с APL. В APL, как и в Smalltalk, заложен один грандиозный упрощающий принцип, на основе которого он был спроектирован и создан. Это имело колоссальное значение. Objective-C возник под влиянием идеи создания гибридного языка, и мы строго придерживались правила не изымать чего-либо из Си, а только добавлять. Таким образом, мы создавали не язык, производный от Си, а гибрид, основанный на Си.

Некоторые исходные решения были настолько важными, что сохранились до сих пор. Я часто вспоминаю, что именно на мне лежит ответственность за квадратные скобки в Objective-C, потому что у нас с Брэдом были долгие споры на эту тему. Должен ли наш Си-синтаксис сохранять единообразие с Си, или мы создаем гибридный язык, где

квадратная скобка, как я говорил, «переключает передачу при въезде в страну объектов»? Мы подумали, что, создавая гибридный язык, можно построить ряд базовых классов, чтобы в какой-то момент почти вся работа стала выполняться внутри квадратных скобок. В результате массу деталей можно скрыть от обычного прикладного программиста.

Квадратные скобки указывают на сообщение, отправленное в Objective-C. Начальная идея была в том, чтобы, построив набор библиотек и классов, большую часть своего времени вы действовали внутри квадратных скобок, так что на самом деле вы бы занимались объектно-ориентированным программированием на основе структуры объектов, разработанных в гибридном языке, представляющем собой комбинацию процедурного и объектно-ориентированного языка. По мере наращивания библиотек и функциональности все реже может потребоваться опускаться в процедурную область, и можно будет оставаться внутри квадратных скобок. Мы намеренно решили создать язык, в котором фактически должно было быть два уровня: после накопления достаточных возможностей можно начать действовать на верхнем уровне. Думаю, это одна из причин. Если бы мы выбрали синтаксис, слишком напоминающий Си, едва ли сейчас кто-нибудь смог бы вспомнить название этого языка, и вряд ли бы он еще где-нибудь применялся.

Другие две задачи в то время заключались в том, чтобы обеспечить простоту и элегантность. В то время было два десятка языков, на которых писали программы. Лично я обнаружил, что, пытаясь написать что-то серьезное на APL, открываешь его реальную мощь, вот и оказывается, что это грандиозная программа для таких приложений.

Моим первым домашним компьютером был IBM 5100, фактически являвшийся APL-машиной. Я заинтересовался, нельзя ли с помощью APL сделать на нем что-нибудь вроде полноэкранного редактора, и выяснилось, что это весьма сложная задача.

Вам пришлось бы рассматривать экран как матрицу символов. Трудновато.

Том: Да. Выбор был неудачен. Многим людям моего поколения довелось какое-то время поработать с языком обработки строк, с Лиспом, с языком для матричных вычислений и с объектно-ориентированным языком. Лично я, добавляя в свой арсенал очередной язык, чувствовал, что обогатил себя важными фундаментальными знаниями.

Понятно, отсюда и мечта о хорошем универсальном языке. Вам захотелось, начав с Си, добавить украшения Smalltalk?

Том: Мы пытались выяснить, какой язык лучше выбрать при создании программных окружений для крупных международных бригад, созда-

ющих телефонные станции. Ни один из имевшихся в то время вариантов нас не удовлетворил.

Когда вышел тот августовский 1981 года номер журнала «Byte», мы все по нескольку раз прочли его от корки и до корки. Однажды Брэд зашел ко мне в кабинет и сказал: «Можно я на недельку возьму этот компьютер себе домой? Мне кажется, что где-то через неделю я смогу продемонстрировать, как можно построить расширение языка Си, весьма похожее на Smalltalk».

В те времена взять компьютер домой было достаточно необычно. Он был размером с коробку для пары тогдашних ковбойских сапог. Он был изготовлен фирмой Onyx, ушедшей в небытие и мало кем сегодня вспоминаемой.

У меня есть книга Smalltalk 80 издания 1983 года. Похоже, работы тут больше, чем на неделю, но сам компилятор не очень сложен.

Том: Да, не очень. Языки со строгим обоснованием не бывают очень сложными. Напротив, можно себе представить, что компилятор C++ окажется весьма уродливой вещью, потому что язык неаккуратный. В нем полно всяких специфических и необычных конструкций, не вполне согласованных между собой, и это проблематично.

Некоторые участники интервью утверждали, что нужно начинать с небольшого базового набора идей, а все остальное строить на этом основании. У вас такой же опыт или представление?

Том: Думаю, это разумный подход. Я бы начал с нескольких примеров действительно простых и действительно четких языков, и первое, что тут приходит на ум, это Smalltalk и APL. Вероятно, можно вспомнить и другие. Еще один кандидат – Лисп.

Для сравнения можно взять какой-нибудь совсем уродливый язык. Я вам его назову; он имеет более важное значение, чем можно подумать. Это язык программирования MUMPS. На самом деле, язык простой. Он весьма безобразен, неопрятен и неудобен, но оказался неплохим для таких задач, как ведение электронного медицинского учета.

В действительности, это не просто язык, а среда программирования. Эта среда очень полезна для создания высокопроизводительных систем электронных медицинских карт. Самая крупная из действующих систем – это всеохватывающая MUMPS, построенная в Управлении по делам ветеранов.

Из 108 приложений около 100 написано на MUMPS. Это примерно 11 миллионов строк, их невозможно модифицировать, и вы нигде ничего подобного не увидите. Электронные истории болезней играют сейчас очень большую роль и в нашей стране, и во всем мире, а крупнейшая

из известных действующих систем написана на этом языке, важность чего большинство не осознает.

Один из моих коллег всегда утверждал, что самым распространенным в мире языком программирования является электронная таблица. Чему тут удивляться?

Том: Приведу пример патологического языка программирования, о котором вы, скорее всего, не слышали. Одно время мне довелось поработать в тихой и скромной фирме Morgan Stanley, занимавшейся созданием трейдерских систем, но удивительно не это.

Один из тамошних ребят решил, что нет в мире языка, идеально подходящего для создания высокоэффективных трейдинговых систем. Он решил написать свой собственный язык. Его конструкция по духу очень напоминала APL, но у автора было твердое убеждение, что код любого достойного компилятора должен занимать не больше 10 страниц.

По мере того как в язык и в вычислительную среду добавлялись все новые функции, он все пытался втиснуть их в свои 10 страниц. В какой-то момент он начал укорачивать имена переменных, чтобы их больше влезало в одну строку. Лет через 15 эти 10 страниц его кода напоминали дамп ядра. Было это в середине 1990-х. Все системы торговли ценными бумагами в Morgan Stanley были написаны на этом языке, который назывался A+.

По прибытии в Morgan Stanley у меня в подчинении оказалось 250 человек, работавших на трех площадках: в Токио, Лондоне и Нью-Йорке. Я начал с того, что не меньше получаса побеседовал с каждым из этих 250 человек на его рабочем месте.

Я заметил, что фактически все они используют множество языков программирования. Я завел таблицу для их учета, в которой в итоге оказалось 32 процедурных языка. Это не были языки запросов. Это не были управляющие языки. Все это были языки программирования. Я спросил у членов группы, не кажется ли им, что количество языков можно сократить до 16?

Я развернул кампанию за сокращение разнообразия. Все 32 языка были отмечены на одной большой диаграмме, висевшей у меня на стене, и для каждого языка была заведена отдельная карточка. На одной стороне было его название, а на обороте – перечеркнутое название, что означало отказ от использования языка. Однажды мой сотрудник позвонил из Лондона и сказал: «Том, переключи свой телефон на громкий прием и подойди к стене. Проведем небольшую церемонию. Сегодня мы официально изъяли из употребления еще один язык». Не помню, как назывался этот язык, но, посмотрев на стену, я не обнаружил его. «Его

нет на стене. Их снова 32. Хорошая новость – мы списали ненужный язык программирования. Плохая новость – их все еще 32».

Это забавная история, но представьте, что 250 человек пишут на 32 разных языках программирования, и какие это влечет проблемы с подбором кадров и выделением ресурсов. У вас есть 15 свободных человек, но не с теми 4 языками, которые будут нужны в следующем проекте. Это крайне затратный способ управления организациями.

В нашем деле «больше» – не значит «лучше».

Если бы сегодня вы начали проектировать новый язык, каким бы он стал?

Том: Учтите, что я готов расширять техническое разнообразие только в тех случаях, когда это совершенно необходимо. Я бы начал с вопроса: «А так ли уж мне действительно совершенно невмоготу без такого языка?» Еще один язык, который мы не назвали, – Ruby. Это четкий язык, который может быть достаточно эффективным – достаточно эффективным для множества приложений. У него замечательная понятная структура.

Я не испытываю потребности в новом языке. Большую часть своего времени я размышляю о проблемах, возникающих еще до того, как будет написана первая строчка кода.

На этой неделе я раздал некоторым своим друзьям кучу кружков со словом REQUIREMENTS (требования), перечеркнутым косой красной полосой, как на европейских дорожных знаках. На обратной стороне каждого кружка приведены 14 приемлемых вариантов замены этого слова. Очень часто между отдельными людьми или целыми группами идут разговоры типа «я не мог сделать эту работу, потому что ты еще не выдал мне технические требования» или «нужно создать группу для сбора технических требований для нашей новой системы».

Этот термин очень неточен. Его нужно заменить более точными терминами. В одном крупном проекте, которым я занимался, мы установили налог на упоминание технических требований. Тот, кто употреблял слова «технические требования» без каких-либо пояснений, должен был бросить два доллара в копилку. Если речь шла о сценариях использования, или карточках с историями, или нужно было обсудить метрики технических данных, или поговорить о деловых ситуациях или моделях бизнес-процесса, эти слова были допустимы. В этом случае вас не облагали налогом, потому что, заявив, что вам нужны сценарии применения, или функциональная спецификация, или макет приложения, вы точно выражаете свое требование.

Я вижу, что неправильно подходя к этому вопросу, проекты сталкиваются с трудностями. Написание кода перестало быть самой трудной частью задачи. Самое трудное – определить, что должен делать этот код.

Вы считаете, что мы достигли того уровня производительности, когда языки, инструменты, платформы и библиотеки уже не играют такой большой роли, как 20 лет назад?

Том: Думаю, это верное замечание. Я далеко не сразу понял, что создавая класс на объектно-ориентированном языке, я фактически занимаюсь расширением языка программирования.

В одних языках это более очевидно, чем в других. Например, в Smalltalk это достаточно очевидно, а, скажем, в C++ не очень.

Поскольку мы имеем возможность фактически создавать специализированные языки с помощью разрабатываемых сред или имеющихся библиотек классов, основные усилия сосредоточиваются на других участках цикла разработки. Я бы особенно выделил фронтальную часть процесса с вопросом «что же мы должны сделать?» и его обратную сторону, каковой является тестирование.

На прошлой неделе я говорил с участником разработки приложения, с которым, как ожидается, ежедневно будут работать 40 000 пользователей. Я попросил его рассказать о том, какие нагрузочные испытания они собираются провести для проверки способности системы выдержать одновременное нажатие клавиши Return всеми этими 40 000 пользователей.

Вероятно, моя непродолжительная работа в телефонной индустрии повлияла на то, что я рассматриваю это как задачу системного анализа. Что касается медицины, например, то мне довелось общаться с людьми, которые предполагали, что электронные истории болезни будут передаваться из центрального хранилища в любую географическую точку за время, не превышающее секунды. Я спросил: «Вы знаете, каков сегодня размер файла данных MRI? Это 5 гигабайт!»

Если подумать о том, какой нужен канал, чтобы меньше чем за секунду передать 5 гигабайт на другой конец страны, то выяснится, что он будет стоить очень дорого. Его можно сделать, но это будет дорого. Едва ли в этом когда-нибудь возникнет реальная необходимость.

Образование и обучение

Каковы ваши рекомендации по управлению сложными техническими концепциями?

Том: Думаю, пример нужно брать с профессиональной подготовки в Европе, какой она была в старые времена. Карьера человека в этом биз-

несе должна начинаться с усвоения простых аспектов того, что в нем делается, с написания контрольных примеров, с разработки функциональных спецификаций проектов и далее продвигаться через более технические вопросы проектирования решений, реализации этих решений и еще более сложные вещи, такие как нагрузочные испытания и реальное развертывание крупных систем. Мне кажется, что мы часто поручаем людям работу, для выполнения которой у них не хватает квалификации, а потом удивляемся, что они с ней не справляются.

Мне несколько лучше знакомо положение дел в Германии, чем в Италии, и по моим представлениям для того чтобы стать сертифицированным архитектором в Германии, нужно проработать некоторое время – думаю, месяцев шесть – в различных строительных профессиях. Нужно получить некоторое практическое представление об устройстве водопроводов, об электрических установках и о том, как реально строится дом, прежде чем вас официально допустят к таким работам в качестве сертифицированного архитектора. В нашем программном деле отсутствует подобная практика сочетания теоретического обучения с практическим опытом – и даже практического обучения после получения теоретического образования.

Насколько важен практический опыт?

Том: Проведу еще одну аналогию – с авиационным делом, где вы должны пройти очень методичный, тщательный, официальный процесс обучения полетам сначала на маленьких самолетах, потом на более крупных, потом на более крупных и более быстрых, прежде чем займете кресло пилота Боинга 757, несущего 300 человек через Атлантику. Разумеется, действующие в авиации правила появились потому, что их отсутствие привело к гибели многих людей. Правила возникли тогда, когда возникла в них необходимость.

Какие темы недостаточно изучаются в учебных заведениях?

Том: Как раз на прошлой неделе я встречался с рядом ветеранов программного бизнеса, и мы обсуждали, в каких учебных заведениях Соединенных Штатов лучше всего изучать программирование, достаточно ли высоко они котируются и хорошо ли финансируются. Ответы были не слишком обнадеживающими. Дело в том, что я четко отделяю образование в области программирования от образования в области компьютерных наук. Я говорю не о том, как спроектировать компилятор или написать операционную систему, а о том, как спланировать проект, как им управлять и как успешно справляться с разными функциями в рамках проекта.

Сейчас я не так хорошо, как прежде, осведомлен о состоянии обучения профессии программиста в Европе, но в Соединенных Штатах оно сей-

час, по моему мнению, весьма неудовлетворительно. Недавно шведский друг сказал мне: «Я собираюсь возглавить некую организацию, создающую программное обеспечение. Какие книги по программированию мне было бы полезно прочесть, чтобы справиться с этой работой?» Я назвал ему пять книг, и он спросил, где их можно купить. «Скажем, на Amazon или где-то еще, – ответил я, – но завтра я буду в книжном магазине одного крупного университета и, если хочешь, куплю их там и пришлю тебе».

Я поехал в Йельский университет, предъявил свой список из пяти книг в магазине, и оказалось, что ни одной из них нет в наличии, что стало для меня неожиданным открытием. Йель – не самое престижное учебное заведение Лиги плюща, а тем более США, в области компьютерных наук, но его программа имеет хорошую репутацию и существует уже давно. Тем не менее там не было даже учебников для студентов, не говоря уже о курсах, из которых можно было бы почерпнуть немного больше, чем из учебников. Вчера я увидел нечто забавное. Это была реклама нового продукта, и один из ее пунктов гласил, что он на 100% написан на Objective-C.

Как бы вы обучали разработчиков программного обеспечения?

Том: Сначала я сформировал бы из них группу тестеров и стал учить тестированию и чтению кода. Программирование – одна из немногих областей, где сначала учат писать и только потом – читать. Это явная ошибка. Нет ничего лучше, чем взять кусок скверного кода и попытаться в нем разобраться. Очень поучительно. Я бы также постарался познакомить учащихся с хорошо разработанными и хорошо спроектированными действующими продуктами, чтобы они увидели, как это выглядит не только снаружи, но и изнутри.

Вот пример очень хорошо написанного продукта нашей компании. Посмотрите на хорошо написанный продукт и хорошо спроектированный продукт и сравните с тем, над которым вы сейчас работаете. Я бы постоянно расширял их обязанности, но очень короткими циклами, чтобы можно было оценить прогресс и успехи, и при этом оказать помощь, если понадобится.

Как вы, нанимая, распознаете хорошего программиста?

Том: Это сложная тема. Едва ли вам известно, что темой моей диссертации было изучение психологических особенностей успешных программистов 1970-х.

Некоторые из них активно работают до сих пор.

Том: Действительно, кое-кто из них еще жив. Совершенно удивительно. Есть определенные психологические когнитивные свойства, на ко-

торые нужно смотреть: крепкая память, внимание к деталям. Я еще обращаю внимание на такие вещи, как коммуникативные способности, – и письменные, и устные. Способность работать в команде: очень важно уметь эффективно общаться с другими участниками команды, а если вы окажетесь на руководящей должности, то важно также уметь общаться с клиентами или экспертами в специальных областях или другими оперативными и сопровождающими подразделениями, с которыми нужно взаимодействовать при развертывании продуктов. Не стану говорить, что это обязательное требование, но если я ищу действительно превосходного программиста на должность главного разработчика или архитектора проекта, то обращаю внимание на их увлечения. Если они знатоки музыки, это очень хороший признак: знатоком я называю того, кто изучал классическую музыку и может сыграть фортепианную сонату по памяти. Это хороший показатель способности к запоминанию и внимания к деталям, да и звучать должно отлично!

Управление проектами и устаревшее ПО

Вы сказали, что программист может сопровождать примерно полкоробки бумажной распечатки.

Том: Верно. В настоящее время я участвую вместе с федеральной администрацией во множестве проектов. Удивительно, насколько полезным оказывается один этот маленький факт. 100 000 строк кода – это коробка бумаги для принтера. Разработать их стоит 3 млн долларов. Для сопровождения нужны два человека. Контрольные примеры для полного тестирования этого кода займут еще две-три бумажных коробки.

А от языка это зависит?

Том: Весьма слабо. Почти не зависит. По крайней мере, для объектно-ориентированных языков разницы нет. Для среднего объектно-ориентированного языка количество людей, тестирующих код, больше количества пишущих, потому что языки стали очень мощными. Сейчас я участвую в проекте, насчитывающем три четверти миллиона строк кода, и больше половины этого кода получено извне. На самом деле, это значительно ниже показателей наиболее современных технологий. С их точки зрения, если вы решите, что на каждого программиста у вас будет один тестер, то сильно недооцените объем необходимого труда, потому что каждый программист фактически вносит массу непроверенного кода со стороны, и в случае, например, медицинского приложения, его нужно подвергнуть доскональной проверке.

В итоге, при определенных обстоятельствах у вас может оказаться пять или шесть тестеров на каждого программиста. Если обратиться к нача-

лу 1980-х, то в среде языка Си у вас вполне могло быть шесть программистов на каждого тестера.

Это связано с особенностями Си и его уровнем абстракции или с многократным использованием кода?

Том: Это связано с размерами библиотеки, которую вы предлагаете со своей стороны. На заре появления Objective-C я часто пытался оценить, сколько строк будет в программе на Objective-C, если переделать в нее большую программу на Си. Коэффициент сокращения колебался около пяти: количество строк кода уменьшалось пятикратно.

Неплохой коэффициент сжатия.

Том: Это огромная величина. Года четыре назад мы делали проект, фактически сочетавший в себе код COBOL и C++ , при этом 11 миллионов строк кода сократились до полумиллиона. А это экономия 20 к 1 в ежегодном сопровождении. Такое сокращение не может не вызвать интереса.

Частично выгода обусловлена опытом, учтенным при повторной реализации системы.

Том: Так всегда бывает. Я объясняю это следующим образом. На практике создавать вторую версию системы всегда проще, чем первую. Отчасти потому, что не приходится размышлять о том, достижима ли цель. Раз система существует, значит, она возможна.

Вы, вероятно, знаете историю о том, как русские добыли B-29 – бомбардировщик, сбросивший атомную бомбу. Они сделали его точную копию, вплоть до заплатки на крыле, в точности повторявшей ту, которая была у оригинала, потратив на это два года и значительно меньше средств, чем те 3 млрд долларов, которые потратили США за пять лет, чтобы построить первый B-29, – он был дороже, чем весь манхэттенский проект![1]

Несомненно, когда идешь по второму кругу, все происходит гораздо быстрее.

Вы сказали, что объемы исходного кода могут отличаться более чем на порядок. А может ли программист сопровождать примерно одинаковое количество строк кода независимо от языка?

Том: Я очень часто обсуждал эту тему с хорошими программистами. Это количество может быть разным. Думаю, со средней величиной все согласны. В то же время разброс, как мне кажется, весьма велик. Встречается ясно написанный код, аккуратный и организованный,

[1] *http://www.rb-29.net/HTML/03RelatedStories/03.03shortstories/03.03.10contss.htm*

и тогда одному человеку под силу сопровождать 200 000 строк. Такое редко, но бывает.

Точно так же можно встретить помойку, и человек сойдет с ума, пытаясь поддерживать в рабочем состоянии 10 000 строк кода. Это наблюдается гораздо чаще. Часто встречается любопытное явление, когда, проделав действительно хорошую работу по проектированию и разработке новой системы и передав ее заказчику, оплатившему ваш труд, с изумлением видишь, с какой скоростью весь выстраданный порядок исчезает при эксплуатации. Люди начинают активно вмешиваться, не понимая, что они делают, и за короткое время могут нанести большой ущерб.

Вы учитывали эти организационные соображения, когда разрабатывали Objective-C?

Том: Фактически, да. Исследовательская группа, начавшая работать в ITT, была сформирована с целью помочь крупной международной телефонной компании в создании распределенной объектно-ориентированной системы цифровой телефонной коммутации, и сделать это нужно было силами географически удаленных команд разработчиков.

Мы глубоко погрузились в эти проблемы. Я пришел в ITT из General Electric, где занимался примерно тем же. В GE я назвал это подразделение «группой исследований психологии программирования». Мы изучали не только сравнительную важность характеристик языков программирования, обеспечивающих легкость чтения, понимания и сопровождения программ, но и организационные принципы формирования команд разработчиков и организационные проблемы, связанные с осуществлением крупномасштабных разработок.

Индустрия программного обеспечения выглядела бы совершенно иначе, если бы правительства возложили ответственность за обеспечение безопасности на разработчиков программ.

Том: Совершенно верно, и одно из правил, которыми я руководствуюсь при назначении руководителей проектов, состоит в том, что будущий проект должен не более чем на 20% отличаться от тех, с которыми они успешно справлялись ранее.

Это значит, что стать архитектором крупного проекта можно, только обладая огромным опытом.

Том: А разве это неправильно? Другой вариант – организовывать очень маленькие проекты. Но нужно учитывать, что есть такие проекты, для осуществления которых нужно три года, и если слегка преувеличить, то окажется, что после 40 лет работы найдется не так много проектов, которые вам можно поручить. Та же проблема в авиации. Ее решение – в создании реалистичных тренажеров, и я пытался провести такую

идею для руководителей проектов. Действительно, трудно выполнить 100 проектов в течение одной жизни, но если моделировать какие-то решения и условия, чтобы включать в свое резюме не только реальные, но и смоделированные проекты, то это помогло бы решить проблему.

От чего больше зависит продуктивность программиста — от его личных качеств или от особенностей применяемого языка программирования?

Том: Влияние личных качеств значительно превосходит влияние языка программирования. Еще в 1970-х исследования показали, что программисты с одинаковой академической подготовкой и равной продолжительностью работы по специальности могут иметь показатели продуктивности, соотносящиеся, как 26:1. Едва ли кто-нибудь станет утверждать, что его язык программирования в 26 раз лучше.

Вы сказали, что стали экспертом по реинжинирингу устаревшего ПО, а для этого нужно понимать, что означают три слова: «гибко», «устаревшее» и «реинжиниринг». Какой смысл вы в них вкладываете?

Том: Начнем анализировать в обратном порядке. Сначала поговорим о реинжиниринге. Здесь я имею в виду очень близкую к прежней функциональности замену с использованием современных приемов проектирования и технологий. Я провожу четкое различие между реинжинирингом и модернизацией. Модернизационные проекты страдают от давно известного эффекта второй системы, описанного Фредом Бруксом в «Мифическом человеко-месяце»: «Давайте на этом круге попробуем реализовать то, что на первом круге мы не знали как сделать, причем за половину отведенного времени». Неудивительно, что такие проекты постоянно сталкиваются с большими трудностями. Я до сих пор не обнаружил проекта модернизации для американского правительства, который оказался бы успешным. Крайне просто найти неудачные проекты, но я ищу тот, который окажется успешным.

Слово «реинжиниринг» я употребляю как обозначение серьезных ограничений, налагаемых на проект. Я не говорю, что нужно посмотреть на старую систему, потом придумать новую и начать все с чистого листа. Если можно повторно использовать рабочий процесс, или описания окон, или модели данных или хотя бы их элементы, либо повторно использовать сценарии применения, или документацию, или обучающие курсы, то вы сэкономите огромное количество времени и труда, а также риска быть отстраненным от проекта.

Гигантское преимущество проекта по реинжинирингу заключается в том, что у вас есть работающая система, к которой можно иногда обратиться за ответами на вопросы, которых вы нигде больше не найдете. Вот то, что касается одного из этих трех слов.

«Устаревшее», конечно, относится просто к существующей системе, часто масштаба предприятия. Это не обязательно системы 20-летней давности, построенные с помощью древних процедурных языков, или что-то еще более страшное. Я, например, слышал про устаревшие приложения на Smalltalk, которые нужно переделать и переписать на Java. «Устаревшими» (legacy) системами называют те, которые развернуты, действуют в настоящее время, выполняют в организации важные функции, но по ряду причин нуждаются в замене. Причиной может быть то, что программа выполняется на устаревшей операционной системе, которая больше не поддерживается и должна быть списана. А может быть, она написана на древнем языке, специалистов по которому уже нет в живых. Или какие-то аналогичные причины. Бывает и так, что характер ведения бизнеса изменился, и прежние функции хотя и нужны, но должны быть скомпонованы по-другому, либо вам нужна совершенно иная функциональность, и реинжинирингом уже не обойдешься. Нужен проект по разработке нового приложения.

Третьим словом было «гибко» (agile), и это просто относится процессу, который неоднократно продемонстрировал свою пригодность, причем в разных масштабах, и потому не желающий подвергаться рискам менеджер проекта должен очень серьезно рассмотреть возможность его принятия.

Как предотвратить возникновение проблем с «устареванием» при написании программ сегодня?

Том: Не уверен, что с этим можно бороться. У любого создаваемого вами продукта есть срок полезной жизни. Часто случается, что срок активной жизни продукта на целые десятилетия превосходит расчеты его разработчиков, и очень отрадно, если оказывается, что этот код хорошо структурирован, хорошо документирован и хорошо протестирован. Я сейчас работаю с клиентом из администрации, у которого есть 11 миллионов строк кода, написанного 25 лет назад, и контрольных примеров для него нет. Нет системной документации. Несколько лет назад для повышения эффективности из кода удалили комментарии, отсутствует управление конфигурацией, и начиная с 1996 года для системы ежемесячно изготавливается около 50 заплаток. Вот тут проблемы.

Можно взять все это и задним числом сказать, что не надо делать того-то и того-то, а вот так делать – правильно.

Что можно сказать о модульности проекта?

Том: Чем лучше спроектирована система, чем она более модульна, тем, по всей вероятности, лучше объектная модель этой системы, тем дольше она будет полезна. Конечно, система может быть прекрасно спроек-

тирована, но если условия бизнеса изменились, то неизвестно, какой объем изменений потребуется сделать в системе.

Есть ли у вас эмпирическое правило относительно количества языков программирования, допустимого в организации?

Том: У меня есть правило для руководителей проектов, но это немного разные вещи. Руководитель проекта должен уметь читать код на всех языках программирования, применяемых в проекте, – чего, кстати, почти никогда не бывает. Я считаю это одной из главных причин такого большого количества неудачных проектов.

У меня был руководитель проекта, который пришел ко мне и сказал: «Мы должны использовать здесь шесть разных языков. Вы действительно считаете, что я должен овладеть навыками шести языков программирования?» «Нет-нет, – сказал я, – это не единственный способ решения проблемы. Есть и другой способ – избавиться от нескольких языков».

Он понял, что я не шучу. Я очень часто присутствовал на совещаниях, где программист рассказывал о трудностях при написании некоторого класса, а у руководителя проекта не было ни малейшего понятия о том, что представляют собой классы в современных языках программирования.

Вы по-прежнему моделируете свои системы с помощью шариков из полистирола, представляющих отдельные классы?

Том: Да, это так. Мы сделали трехмерную анимацию на ту же тему, но она сильно проигрывает полистирольным шарикам. Есть нечто такое в выпуклой физической структуре, висящей под потолком и регулярно обновляемой в процессе работы над проектом, благодаря чему вы не только видите организацию создаваемой системы, но и текущее состояние каждого ее класса.

Я подсчитал, что мы делали это в 19 проектах. В одном из них было 1856 классов, что очень много – возможно, больше, чем следовало. Это был крупный коммерческий проект, так что все должно было быть крупным.

Сохраняет ли класс свою роль основной единицы измерения успешного развития системы?

Том: Это самая прочная вещь, на которую, как мне кажется, можно положиться. Нужно определить характер класса, который вы пишете. Если это начальный прототип класса, его можно создать за неделю. Настоящий класс, входящий в конечное приложение, потребует около месяца труда. Класс с высокой вероятностью многократного использования требует от двух до четырех человеко-месяцев труда.

Включая тестирование и документирование?

Том: На всю катушку.

Нужен день, чтобы прочесть и понять класс. На этом сегодня спотыкаются многие проекты, потому что если вы собрались использовать какую-то библиотеку классов, Swing или другую, которая вам нравится, и никто не скажет, действительно ли всем программистам нужно разобраться в 365 классах, то пройдет 365 дней, прежде чем вы будете готовы написать первую строку кода.

С другой стороны, если в начале вы забудете учесть, что нужно изучать код, у вас могут возникнуть потом большие отставания от графика.

Можно, например, наверстать время за счет отладки.

Том: Где-то вам придется потратить время. В какой-то момент вы должны будете понести эти расходы. Они велики. Если у вас проект на шесть месяцев, задержка в начале составит два года.

И этого не избежать?

Том: Иногда, но есть несколько способов борьбы с этим. Можно набрать работников, которые уже знакомы с этими классами. Можно разделить работу так, что каждому человеку необязательно будет знать все, и это почти всегда правильный подход. Об этом надо думать. Такие вещи не должны стать для вас неожиданностью.

Взгляните на некоторые современные проекты на Java. Они вполне могут начаться с 2000 классов, которые им необходимы. Если в году 200 рабочих дней, вы получите 10-летнее отставание от графика.

Вы сказали, что время написания кода долгие годы остается неизменным, но вы также отметили, что есть ряд факторов, повысивших нашу продуктивность. Некоторые достижения в продуктивности требуют вложения времени и труда в их освоение.

Том: Но разве не лучше потратить человеко-день на изучение класса, чем человеко-месяц на его написание заново? Это дорого, но у вас сразу появляется большой объем функциональности, тогда как в прежние времена приходилось писать его с чистого листа.

С учетом того, что в месяце 20 рабочих дней, вы повышаете продуктивность в 20 раз. Хороший обмен.

Том: Это колоссально. Возьмем некоторые цифры, которые я считаю средними в наше время. Допустим, вам нужно изучить 500 классов, чтобы писать серьезные современные приложения. Каждый раз, когда вы начинаете проект, это не будут 500 совершенно новых классов. Вы не станете изучать их все 500 в один прием. Вероятно, вы изучите их за пять или более подходов.

Как вы оцениваете простоту проекта?

Том: В прежние времена была такая оценка, как количество страниц с БНФ, описывающими язык, и она была неплохой, потому что она позволяла различать сложные и простые языки.

Если взглянуть под определенным углом зрения на программу на APL или Smalltalk, язык перестает быть виден. Он как бы исчезает, и это хороший признак.

Насколько велико справочное руководство по языку? Что в него включено? Язык Objective-C не очень сложен, но сложны написанные на нем библиотеки, накопившиеся с годами. Описывать их во всех деталях – трудно, громоздко и чревато ошибками. Сложно тестировать, сложно документировать.

Даже у такого простого языка, как Си, может быть сложная семантика – кто отвечает за управление памятью для данной конкретной библиотеки?

Том: Именно так. Ведь нередко приложения Microsoft начинают работать все медленнее и медленнее, потому что они плохо управляют памятью. Встречали ли вы операционные системы Microsoft трехлетней давности, с которыми хотелось бы работать? Я работаю на лэптопе, на котором есть зона, свободная от Microsoft. Просто удивительно, насколько эффективнее я работаю по сравнению с теми, кто сидит со мной в одной комнате за компьютерами Microsoft. Я включаю свою машину, делаю нужную работу и закрываю машину еще до того, как они откроют первую таблицу Excel.

Какой самый полезный совет вы могли бы дать на основании своего опыта?

Том: Могу укоротить его до четырех слов. Делайте интересные новые ошибки.

Не нужно переворачивать всю историю, но посмотрите на пройденный путь. Много ли вы знаете 25-летних выпускников компьютерной специальности, написавших хоть одну программу на APL?

Думаю, их почти нет. А ведь это важные знания, потому что это, конечно, специализированный язык для приложений определенного рода, но, думаю, опытный APL-программист при написании статистического приложения заткнет за пояс любого, кто попытается работать с каким-то другим языком.

Я беседовал с некоторыми людьми из Института программирования по поводу того, как сейчас учат программистов, если считать, что этим вообще кто-то занимается, и как мало программ для получения степени,

где сознательно стараются научить людей, как создавать системы, а не просто разрабатывать алгоритмы. Думаю, здесь у нас есть упущения.

Как было бы здорово – сейчас я так и делаю, – если бы при найме руководителя проекта он показал вам заверенный журнал, где были бы указаны все проекты с его участием и лица, у которых можно узнать о деталях каждого проекта, с количественными показателями для всех созданных продуктов. Сколько строк кода, сколько классов, сколько контрольных примеров, выполнение графика работ и так далее.

У меня есть разрешение на управление гражданским самолетом, и я немного этим занимаюсь. В авиации есть правила, в которых учтен печальный опыт летных происшествий, почти всегда приводящих к гибели людей, и такие правила и законы для авиации есть в нашей стране и во всем мире.

Обычно в США ежедневно совершается примерно 56 000 полетов, и в последнее время выпадают годы, когда не было ни одной аварии пассажирского самолета. Это просто поразительные результаты.

Некоторые законы удалось выявить. Например, при взлете пилот должен быть трезв. Он должен пару раз полетать с летчиком, знакомым с данным самолетом. Когда наша отрасль достигнет зрелого состояния, у нас тоже появятся какие-то правила в таком же духе. Одна из причин, по которым организации так редко задумываются о создании новых приложений или хотя бы реинжиниринге старых, это страх. Они считают, что любой проект, за который они возьмутся, окончится провалом, и не хотят нести за это ответственность.

Если бы у них, напротив, была уверенность в удаче на 90, 95 или – не дай бог – 99%, и они заранее знали, во что обойдется проект, то положение в нашей отрасли было бы гораздо более надежным.

Что вы понимаете под «успехом»?

Том: Есть разные цифры. Допустим, некоторый проект должен в итоге насчитывать миллион или больше строк кода. Вероятность успешного завершения такого проекта в США в настоящее время очень низка – значительно меньше 50%. Цифры спорные. Не знаю, где берут достоверные данные подобного рода, потому что никто не хочет огласки такой информации. Многие компетентные люди пытались добыть такую информацию. Но подчеркиваю: добыть ее можно, хоть и трудно. Всего лишь трудно.

———— ♦ ————

Objective-C и другие языки

Почему вы решили расширять уже существующий язык, а не создали новый?

Брэд Кокс: Меня вполне удовлетворял Си, не считая известных, но терпимых ограничений. Изобретать базовый язык для ООП было бы пустой тратой времени.

Почему вы остановились на Си?

Брэд: Потому что у нас не было ничего другого. Ада – это нечто немыслимое, Паскаль считался игрушкой для ученых. Оставались Кобол и Фортран. По-моему, все сказано. Ах, да, был еще Chill (язык для телефонии). Единственной разумной альтернативой для Си был Smalltalk, но Xerox не рассталась бы с ним.

Нашей задачей было перенести ООП из исследовательской лаборатории в заводские условия. Единственным надежным вариантом был Си.

Зачем понадобилось эмулировать Smalltalk?

Брэд: Меня осенило за 15 минут. Как кирпич на голову. В больших проектах на Си меня всегда раздражало отсутствие всякой инкапсуляции и оборачивания данных и процедур во что-либо, напоминающее методы. Вот и все.

Когда вы впервые узнали про C++ и какое у вас сложилось мнение о нем?

Брэд: Бьерн узнал о моей работе и пригласил меня в Лаборатории Белла, когда оба наших языка еще не были широко известны. Он ориентировался исключительно на статическое связывание и обновление Си. Я был заинтересован в добавлении в добрый старый Си динамического связывания самым простым и наименее разрушительным способом, какой только можно придумать. У Бьерна были амбициозные задачи: сложная производственная линия для программ на уровне изготовления логических узлов. Меня интересовало нечто гораздо более простое: этакий программный паяльник, с помощью которого можно собирать программные чипы, изготовленные на простом Си.

Как вы объясните разницу в скорости распространения этих двух языков?

Брэд: Objective-C был начальным продуктом небольшой компании, получавшей все свои доходы от продажи компилятора и вспомогательных библиотек. AT&T создавала C++ не с целью получения доходов и вполне могла позволить себе отдать его в общее пользование. Такой козырь, как бесплатность, почти всегда побеждает.

Вы участвовали в проекте, который Apple объявила как Objective-C 2.0?

Брэд: Я никак не связан с Apple, если не считать, что мне нравятся их продукты.

Что вы думаете о сборщиках мусора?

Брэд: Это замечательная вещь. Всегда так считал. Мне приходилось сражаться с маркетологами, которые считали это «особенностью» языка, которую можно без особых хлопот подрисовать к Си, не потревожив тех, кто выбрал Си из-за его эффективности.

Почему в Objective-C запрещено множественное наследование?

Брэд: Историческая причина состоит в том, что Objective-C является прямым потомком Smalltalk, где тоже не поддерживается множественное наследование. Если бы мне сегодня предоставилась возможность пересмотреть то давнее решение, то, возможно, даже одиночного наследования не осталось бы. Наследование не играет большой роли. Инкапсуляция – вот в чем главный вклад ООП.

Почему Objective-C не поддерживает пространства имен?

Брэд: Когда я непосредственно занимался этим языком, моей задачей было скопировать Smalltalk, а там понятие пространства имен отсутствовало.

То, что сегодня известно как Objective-C, в одинаковой мере продукт мой и Apple. Сегодня основная моя работа связана с XML и Java.

Была ли идея протоколов уникальной для Objective-C?

Брэд: К сожалению, не могу приписать себе эту заслугу. Эта идея входила в число добавленных сверху к главной чистой конструкции Objective-C – тому, что считаю минимальным набором, взятым из Smalltalk. В Smalltalk тогда не было ничего похожего на протоколы, и это добавление сделал Стив Нарофф (Steve Naroff), который сейчас отвечает за Objective-C в Apple. Если память мне не изменяет, идею он взял из SAIL.

Ваш проект, видимо, повлиял на Java, поскольку одиночное наследование было перенесено и туда. А из Java тоже можно было бы удалить одиночное наследование?

Брэд: Вероятно, да. Но этого не должно быть и не будет. Оно там есть, оно работает, и оно оправдывает свое назначение. Просто им иногда злоупотребляют, как и любыми возможностями языка, и оно не так важно, как инкапсуляция.

Сначала я активно использовал наследование, чтобы оценить пределы его применения. Затем я понял, что настоящим достижением ООП яв-

ляется инкапсуляция, и с ее помощью можно вручную сделать почти все, для чего я применял наследование, но более понятным образом.

С тех пор мое внимание было перенесено на объекты с более высоким уровнем детализации (OOP и JBI/SCA), которые, заметьте, вообще не поддерживают наследование.

Как вы решаете, нужна ли какая-то функция в проекте? Например, сборщик мусора может замедлять приложения Си, но при этом дает ряд преимуществ.

Брэд: Совершенно верно. На самом деле, мы в Stepstone разработали сборщик мусора, похожий на тот, который сейчас есть у Apple. Был даже интерпретатор Objective-C. Но маркетологам потребовалось что-то автоматическое, чтобы на равных соревноваться со Smalltalk, а не то, что очень напоминало Си.

Вы считаете необходимыми настройки по умолчанию и ограничения на изменение конфигурации?

Брэд: Что мы смогли, то и сделали. Против чего я возражал, так это против попыток ломать Си в соответствии с желаниями маркетологов.

Иногда разработчики языка начинают с описания маленького формализованного ядра, а затем на этом фундаменте возводят язык. Вы подошли к созданию Objective-C так же — или решили позаимствовать все, что можно, у Smalltalk и Си?

Брэд: Да уж, я точно не начинал с формального обоснования. Я думал о кремниевых кристаллах. Мы много консультировали крупное кремниевое производство и бывали на их фабриках. Я сравнивал их работу с нашей и обратил внимание, что вся их работа основана на многократном использовании кремниевых компонентов. Все их мысли были направлены на эти детали, а не на соединяющие их паяльники. Для нас, конечно, таким паяльником служит язык.

Я видел, что все заняты языками, то есть паяльниками, и никого не интересуют компоненты. Мне показалось, что этот подход устарел. Оказывается, очень важно то, почему у производителей чипов было такое видение: кремниевые компоненты состоят из атомов, и есть бизнес-модель для их покупки и продажи. Для программных компонентов такая бизнес-модель в действительности оказывается очень недолговечной.

Все программное обеспечение недолговечно.

Брэд: Верно, но потому мы и сосредоточиваемся на языках, а не на компонентах. В сущности, у нас нет сколь-нибудь устойчивых компонентов. Если провести аналогию с жилищами, то мы словно вернулись

в пещерный век, когда для того, чтобы построить дом, нужно было найти какую-то монолитную кучу – например, вулканического пепла, – и, боюсь, это приведет нас к загрузчику классов Java. Пещерное жилище строится так: берете большую кучу и удаляете из нее все ненужное, что в сущности дает модель загрузчика классов. Моя энергия сейчас направлена на работу с компонентами мелкой детализации, где начинаешь с пустого места, добавляя все по мере надобности, как и в современном домостроении.

Язык биологии и химии – английский. Может быть, проблема в том, что применяемые нами языки программирования слабее по своим возможностям, чем английский язык?

Брэд: Если бы компьютер был таким же умным, как человек, я бы поверил в такой подход. Но компьютер так же туп, как кирпич, и он не изменился принципиально с тех пор, как я начал работать с компьютерами в 1970-х.

С другой стороны, появился ряд замечательных языков, например функциональные языки, которые могут помочь при работе с многоядерными компьютерами. Я постоянно слышу уверения, что могут помочь, и у меня нет оснований сомневаться в этом.

Как ориентированность на параллельные вычисления влияет на парадигму ООП? Нужны ли изменения в ОО-методах?

Брэд: Несмотря на наследие ООП Simula, я всегда считал, что языки программирования должны обеспечивать достаточную поддержку потоков, чтобы можно было создавать компоненты высокого уровня, проявляющие свойства параллельности в ограниченном, но контролируемом виде. Например, фильтры UNIX, будучи реализованы на обычном Си, поддерживают параллелизм контролируемым образом. Я потратил немало времени, работая над аналогичным подходом в Objective-C: библиотекой многозадачности TaskMaster, построенной на одном лишь механизме `setjmp()`.

Другой пример – архитектура высокого уровня Отдела моделирования и имитации министерства обороны США, широко используемая для имитации в военном деле. Она реализована на нескольких языках – включая C++ и Java. Она поддерживает управляемую событиями модель параллелизма, насколько мне известно, не зависящую от поддержки потоков какими-либо языками.

Последний пример больше связан с моей нынешней работой. SOA поддерживает объекты крупной детализации, размещаемые в сети и по необходимости параллельные, поскольку обычно они располагаются на разных машинах. JBI Sun и SCA OASIS усиливают эту модель посред-

ством объектов/компонентов большей детализации, из которых собираются объекты SOA. Это первое проявление в разработке ПО подхода с различной степенью детализации, являющегося нормой в разработке аппаратуры: мелкие объекты (логические элементы) собираются в средние (чипы, «программные ИС»), которые собираются в более крупные объекты (карты), которые... и так далее. Главное отличие в том, что в материальных областях системы действительно фрактальны и имеют гораздо больше уровней, а у нас их всего три. Пока.

Несомненно, есть приложения, требующие более плотной интеграции. Просто я не занимался этой проблемой в свое время. Это упущение отчасти объясняется моим собственным неумением управлять системами с высоким уровнем параллелизма и моими сомнениями в том, что кто-либо вообще умеет это делать.

Мы наблюдаем неуклонный рост сложности и размера приложений. Чего больше от ООП – пользы или вреда? Мой опыт показывает, что идея создания объектов многократного использования увеличивает сложность и удваивает объем работ. Сначала пишешь объект многократного использования, а потом приходится модифицировать его и заполнять оставшееся из-под него место чем-то другим.

Брэд: Вы правы – если под ООП понимать инкапсуляцию в стиле Objective-C/Java, которую в своей второй книге «Superdistribution» (Addison-Wesley Professional) я назвал *интеграцией на уровне чипов*. Но если рассматривать интеграцию на уровне чипов лишь как один уровень в комплекте средств многоуровневой интеграции, являющемся нормой при проектировании аппаратных средств, то объекты уровня логических элементов гармонично вписываются в фрактальный мир инкапсуляции уровней чипов, карт и выше.

Именно поэтому я занялся сегодня SOA (уровень шасси) и JBI (уровень шины). Они поддерживают инкапсуляцию на том же уровне, что и традиционное ООП. И даже выше: они инкапсулируют не только данные+процессы, но и весь свой поток управления.

Самое замечательное то, что многоуровневая интеграция не стоит вам ничего и показала свою пригодность, будучи использована в любых масштабах в других областях производства. Единственная зацепка – в трудности убедить сторонников одноуровневого ООП. Все это мы видели – борьбу ООП и традиционного процедурного программирования, а сегодня – SOA с JBI и Java.

Все верят мифу о том, что новые технологии делают «устаревшими» старые, такие как ООП. Этого никогда не было и не будет. Новое всегда вырастает поверх старого.

Похоже, мы вступаем в эру экспериментов с языками и готовности программистов опробовать непривычные для них технологии вроде Rails и функционального программирования. Какие уроки из своей работы над Objective-C вы могли бы предложить модернизаторам и авторам языков?

Брэд: Я пытался, но не вполне успешно, понять синтаксис таких языков, как Haskell, – по крайней мере, не настолько успешно, чтобы составить твердое мнение о нем. Я довольно активно применяю XQuery, а это функциональный язык. И я считаю, что читать XQuery гораздо приятнее, чем XSLT.

Мне кажется, я умею осваивать новые методы, но что касается Haskell, то я просто не могу понять его синтаксис. Примерно та же проблема была у меня с Лиспом. Тем не менее на меня произвел впечатление проект для военно-морского флота, в котором сложная политика авторизации и аутентификации была выражена в виде правил Haskell, допускающих проверку.

Я вижу будущее не в том, чтобы непрерывно множить все новые и новые нотации для одной и той же работы, которой мы занимаемся десятилетиями, – написания процедурного кода. Это не означает, что работа на таком уровне не имеет значения: из нее в конечном итоге возникают компоненты более высокого уровня, и тут вряд ли что-то изменится. Я только хочу сказать, что новые способы написания процедурного кода – это не передний край инноваций.

Что я нахожу увлекательным, так это появление новых видов нотации для совершенно нового занятия, а именно для составления систем более высокого уровня из библиотек существующих компонентов. Конкретным примером служит BPEL, допускающий применение на двух уровнях интеграции: SOA для объектов крупной детализации и JBI (Sun) или SCA (OASYS). Посмотрите, например, редактор BPEL в NetBeans. В этой области OMG проделала отличную работу с архитектурой, управляемой моделями.

Компоненты, песок и кирпичи

Какие уроки из опыта изобретения, дальнейшего развития и распространения вашего языка могли бы извлечь разработчики компьютерных систем в настоящем и будущем?

Брэд: Многоуровневая интеграция (она же инкапсуляция) интересовала меня, сколько я помню, всегда: разбить работу на части и поручить их специалистам, затем воспользоваться тем, что они сделали, не нару-

шая или почти не нарушая инкапсуляцию, знать, «что внутри», чтобы успешнее применять.

В некоторых своих писаниях я использовал в качестве примера обычный деревянный карандаш. Когда я спрашивал у аудитории, что «проще» – цифровой карандаш вроде Microsoft Word или деревянный карандаш, большинство дружно заявляло, что деревянная разновидность проще. Затем я указывал, что Microsoft Word писали восемь программистов, а в создании деревянного варианта участвовали тысячи людей, ни один из которых не мог в полной мере оценить, насколько сложно валить лес, добывать графит, плавить металлы, делать лак, выращивать рапс для масла и так далее. Карандаш заключает в себе высокую сложность, но она скрыта от пользователя.

Меня не удовлетворяет в Си (и подобных ему языках) то, что на программистов полностью обрушивается вся сложность программы, за исключением небольшой ее части, инкапсулированной в функциях. Единственной действительной границей инкапсуляции было все пространство процесса, что равносильно чрезвычайно эффективной капсуле верхнего уровня, интенсивно используемой в UNIX в виде двухуровневой инкапсуляции посредством скриптов оболочки и понятия «каналов и фильтров».

В попытках объяснить преимущества объектно-ориентированного программирования я часто пользовался термином Software-IC (программные ИС), чтобы обозначить новый уровень интеграции, более крупный, чем функции Си (уровень логических элементов), и меньший, чем UNIX-программа (уровень шасси). Главным мотивом разработки Objective-C явилось для меня желание сделать простой программный паяльник, чтобы собирать большие функциональные блоки (уровня шасси) из многократно используемых «программных ИС». Напротив, C++ возник из совершенно другого представления: огромная автоматизированная фабрика для изготовления тесно связанных блоков из элементов уровня логических узлов. Если интерпретировать метафору аппаратуры для программирования, то интеграция на уровне чипов происходит только на этапе компоновки; интеграция на уровне логических элементов – на этапе компиляции.

Тогда (в середине 1980-х) облегченные потоки не были широко известны и не существовало ничего меньше процесса в стиле UNIX («тяжеловесного» потока). Я потратил некоторое время на создание библиотеки Objective-C под названием Taskmaster, которая должна была поддерживать облегченные потоки в качестве основы для интеграции на уровне карт. Появление RISC-процессоров положило этому конец: работа с кадрами стеков на таком низком уровне слишком осложняла задачи переносимости.

Большие перемены произошли потом в связи с появлением «вездесущих систем», открывших новые уровни интеграции, более крупные, чем пространство процесса UNIX. Например, в архитектуре, ориентированной на сервисы (SOA), Интернет представляет собой нечто вроде проводов между компонентами HiFi-системы, а сервисы (программы) на отдельных серверах сами функционируют как компоненты. Я считаю, что это невероятно интересно, поскольку это первый уровень интеграции, аппроксимирующий естественное для повседневной жизни распределение обязанностей. Нет интереса или необходимости рассматривать исходный код каждого используемого сервиса, если этот сервис выполняется на удаленном сервере, у которого другой хозяин.

Важно, что это также первый уровень интеграции, позволяющий исправить роковую ошибку идеи «программных ИС»: возможность создать побудительные мотивы для предоставления своих компонентов. Карандаши (и многие субкомпоненты для их изготовления) производятся потому, что для материальных товаров действует закон сохранения массы. Для цифровых товаров не существовало ничего подобного, пока SOA не открыла возможность развернуть полезный сервис и взимать плату за его использование.

В последние два года я познакомился с фундаментальной проблемой «старой доброй SOA» и теперь в значительной мере сосредоточился на ней. В крупных установках SOA (например, NCES в DISA и FCS в армии) существует большое сопротивление достижению консенсуса, необходимого для построения полностью однородных систем SOA, применяющих совместимые транспортные механизмы для сухопутных войск, надводных, подводных и воздушных судов, не говоря уже о совместимости определений конфиденциальности, целостности, предотвращения отказа от выполнения обязательств и так далее. И даже если расширить «предприятие», включив в него всю обширную организацию систем министерства обороны, то как быть с союзниками? Как обеспечить совместимость с другими правительственными агентствами? С штатами? Со службами быстрого реагирования? Как бы вы ни определили предприятие, вы упустите что-то, с чем в один прекрасный день понадобится установить связь. Возложение ответственности за это на каждого разработчика сервиса не выдерживает расширения.

Поэтому сейчас я рассматриваю Sun JBI (Java Business Integration) и ее многоязычного преемника SCA (Software Component Architecture) как способ поддержать более низкий уровень интеграции, с помощью которого можно построить старые добрые сервисы SOA.

Требуется ли для создания такой системы поддержка со стороны аппаратного обеспечения?

Брэд: Аппаратное обеспечение действительно дает надежный пример того, на что можно надеяться. Компьютерная аппаратура – один из многочисленных примеров инженерных достижений за 200 лет после промышленной революции. Человек достиг исключительных успехов в создании аппаратного обеспечения, так что, создавая будущую модель, мы можем многому поучиться в этой области. Но легких путей здесь нет, и одним нажатием кнопки качество не улучшишь.

В последнее время я привожу другой пример, чтобы показать, что я имею в виду, называя программное обеспечение примитивным. Рассмотрим такое занятие, как жилищное строительство. Люди занимаются этим тысячи лет. Кое-где это делают прежним примитивным способом, когда строители укладывают глину в формы и сами делают кирпичи, из которых потом строят дом.

Несмотря на некоторые признаки прогресса, программные системы в целом создаются точно таким же способом. Например, чтобы создать сервис SOA, каждая бригада разработчиков добывает в карьере java.net сырье, необходимое для любого сервиса SOA.

Возьмем, например, безопасность, для которой, в принципе, и возводятся стены дома. Министерство обороны предъявляет к безопасности исключительно строгие требования, обусловленные политикой. Оно также вкладывает большие средства в такие стандарты, как WS-Security. А недавно оно предложило процедуру сертификации и аккредитации (C&A), которую должны пройти все веб-сервисы. C&A является по сути трудоемким и дорогостоящим процессом, через который должны независимо пройти все веб-сервисы, чтобы гарантировать их достаточную защищенность для размещения в секретных сетях.

Эти веб-сервисы напоминают дома из глиняных кирпичей. Эти политики и стандарты составляют инструкции, которым должен следовать каждый разработчик при изготовлении своих кирпичей. Но эти кирпичи не вызывают доверия у других, потому что их качество зависит от того, кто их делал. Действительно ли их делали по стандартам? Правильно ли применяли стандарты? Пусть эти кирпичи бесплатные, но их качество известно только тем, кто их изготовил.

Строительная отрасль довольно давно перешла к относительно сложной форме производства, которую мы сегодня воспринимаем как нечто само собой разумеющееся. Строительные бригады больше не делают кирпичи сами, потому что никто не доверил бы им это дело. Стандарты и политики сохраняются, так же как процедуры сертификации и аккредитации (проверка качества кирпичей в промышленных лабораториях). Однако мы обычно не замечаем их. Мы полагаемся на эти невидимые процессы, которые гарантируют, что все имеющиеся в продаже кирпичи устойчивы к непогоде и выдержат крышу.

Но главным в этом процессе эволюции от глиняных до современных кирпичей была не техника. Главным было завоевание доверия. Мы доросли до понимания того, что должны быть независимые стандарты, описывающие функции кирпича, его размеры и устойчивость к погодным условиям. И есть независимые лаборатории сертификации, от которых мы неявно зависим. Большинство и не знает о существовании таких лабораторий – и не обязано знать: мы просто можем быть уверены, что кирпичи такие, какими должны быть.

И здесь мы возвращаемся к вопросу о значении экономических систем. Надежные кирпичи существуют потому, что есть стимулы для их изготовления: за каждый кирпич вам платят. И все, кроме нескольких специалистов по изготовлению кирпичей, могут позабыть о сложностях их изготовления и приступить к более творческим вопросам строительства.

Подробнее вопрос разницы между строительством из глиняных и современных кирпичей рассмотрен на *http://bradcox.blogspot.com/*.

Чтобы закончить на более оптимистичной ноте, замечу, что ситуация начинает исправляться. Ряд фирм приступил к выпуску компонентов безопасности SOA и начал долгую борьбу за принятие их министерством обороны в качестве надежных. Наиболее успешный из известных мне примеров – OpenSSL, но Sun, Boeing и ряд других компаний недавно выступили с аналогичными инициативами именно в области безопасности SOA.

Оказали ли влияние инкапсуляция и разделение обязанностей на разработку ПО?

Брэд: Думаю, да. В значительной мере это зависит от того, как бьются со сложностью в других отраслях. Пожалуй, людям свойственно применять инкапсуляцию, чтобы бороться со сложностью.

Можно ли считать SOA и ваши инициативы попытками ввести в ПО принципы компонентной организации?

Брэд: Именно так. Это то, за что я борюсь всю свою карьеру.

Вы всегда так считали или заметили, что обычные методы разработки ПО не соответствуют реальности?

Брэд: Я начал с последнего и стал искать людей, которые действуют иначе. Я начал с осознания того, что ПО пишут неправильно.

Совместим ли практический подход на основе крупных компонентов с математическим взглядом компьютерных наук на ПО?

Брэд: Возможно. Я не хочу углубляться в критику компьютерных наук. Но я не замечал, чтобы в них когда-либо учитывалось, как люди привыкли решать задачи, что проистекает из характерного различия точек

зрения. Специалист по компьютерным наукам исходит из того, что есть такая наука, как программирование, и что его задача – описывать ее и обучать ей. Я всегда исходил из того, что такой науки нет. Есть наука о том, как строить дома, и есть наука о том, как делать кремниевые микросхемы, а наше дело – понять, чему они научились за тысячи лет, и начать создавать науку программирования. Для меня этот стакан совершенно пуст, а они в этом стакане что-то видят.

Вы пару раз отметили, что экономическая модель ПО терпит крах.

Брэд: Дело не в том, что она рушится. Это означало бы, что есть какая-то модель, а теперь она распадается. Но ее никогда не было. Ее не создать, потому что ПО улетучивается как пар. Человечество просто не придумало, как строить экономику в таких условиях. Что-то отдаленно можно вообразить, как с сервисами SOA, для которых можно представить экономическую модель сервисов, потому что ПО сидит где-то на сервере и за него можно взимать плату.

Возьмем эти объекты с мелкой детализацией: у меня была надежда, что есть пути работы с мелкими объектами – песком и гравием вычислений, объектами java.net, но человеческая сложность организации решения для этого оказалась непреодолимой. Если взять промежуточную область, то перспективы среднего уровня с промежуточными увеличенными объектами составляют для нас ядро. Так что на уровне SOA какая-то надежда есть. А может быть, и на следующем уровне ниже. Последнее слово еще не сказано. Но думаю, что для объектов действительно мелкой детализации надежды нет.

Этот средний уровень – уровень структур?

Брэд: Это объекты как пещеры – монолитные компоненты, где вначале есть нечто очень крупное, а потом функциональность отключается. Думаю, такая модель есть. Она поддерживается, например, в JBoss. По существу вы раздаете кусочки и продаете доверие. Сделать такое очень сложно.

Что означает «доверие к программному объекту»?

Брэд: Над этим работает минобороны, и там есть ответы на такие вопросы. Это очень трудоемко и невероятно дорого, неудовлетворительно во всех мыслимых отношениях, но ответ у них есть.

Но оно настолько зациклилось на SOA, что не понимает, что только SOA недостаточно. Есть еще потребность в компонентах более мелкой детализации, которые могут быть многократно использованы для решения повторяющихся задач, не решаемых только с помощью SOA. Таких как безопасность и интероперабельность, по терминологии минобороны. Поэтому моя нынешняя работа заключается в создании компонентов

более низкого, чем SOA, уровня, которые можно соединять для решения скучных повторяющихся задач построения сервисов SOA. Надеюсь когда-нибудь провести эти глиняные кирпичи через процедуру сертификации минобороны, чтобы они стали настоящими кирпичами, которым можно так же, не задумываясь, доверять, как мы доверяем кирпичам, из которых строим свои дома.

Безопасность – пример того, что необходимо каждому SOA-приложению для минобороны. Это необходимое требование согласно политике. Единственное, что есть у создателя SOA-приложений, – это стандарты и политики. У минобороны нет поставщика кирпичей. Нет надежных компонентов – так сказать, настоящих кирпичей, – которые можно взять или купить и строить из них, не изучая целиком стандарты SOA, политику в области безопасности и детали применения компонентов java.net, чтобы удовлетворить требованиям минобороны по безопасности.

Как воспринимаются эти многочисленные уровни абстрагирования с точки зрения безопасности?

Брэд: Не уверен, что правильно понял ваш вопрос. Это все равно что спросить: «Как разделение труда в автомобильной промышленности влияет на безопасность езды?» Полагаю, что благодаря специализации над каждым продуктом плечом к плечу трудится много людей, и какая-то часть из них могут оказаться недобросовестными. Но то, что создается ими на некотором уровне, должно пройти приемные испытания на всех более высоких уровнях, поэтому у зловредных эффектов меньше шансов пройти, чем в случае монолитной конструкции.

Обратите внимание, что многоуровневая интеграция – это не несколько уровней «абстрагирования». Это в действительности многоуровневая «конкреция»: компоненты верхнего уровня собираются из многократно используемых, готовых, протестированных компонентов. Возьмите, например, безопасные сервисы SO, собранные из субкомпонентов библиотеки JBI путем оборачивания базовой функциональности сервисов SOA в компоненты JBI, каждый из которых обеспечивает атрибуты безопасности SOA: аутентификацию, авторизацию, конфиденциальность, предотвращение отказа от выполнения обязательств, целостность и так далее. Результат будет более надежным просто потому, что можно позволить себе работать лучше.

Есть ли ответы, которые могли бы удовлетворить всех остальных?

Брэд: Думаю, да. В чем различие между глиняным и настоящим кирпичом, благодаря которому мы не строим дома из глиняных кирпичей? В том, что глиняному кирпичу нельзя доверять. То же самое с программным обеспечением. Качество глиняного кирпича целиком зависит от того, кто его сделал, от мастерства изготовителя. Отвлечемся от

технических различий между настоящими и глиняными кирпичами. Главное отличие настоящего кирпича, помимо высокотемпературного обжига, заключается в прохождении через лаборатории проверки и сертификации. Чтобы стать поставщиком настоящих кирпичей, нужно пройти эту процедуру. В принципе, это и есть модель доверия минобороны.

Сертификация и авторизация сами по себе очень трудоемкие процедуры. По-другому они называются «стандартными критериями». Теперь они есть, и, думаю, это то, к чему мы пришли после тысяч лет неудовлетворенности глиняными кирпичами. «Мы» – это промышленность в целом, потому что сейчас много жалоб на ненадежность программ и не так много решений этой проблемы. В конце концов, люди не верят программам. И никогда не будут верить. В конце концов, мы придем к модели доверия, основанной на людях, которые поставляют программы.

Взгляните на Sun. По сути, она пытается создать совершенно новую бизнес-модель. Как вы знаете, недавно там приняли подход open source. Здесь очень подходит приведенная мною выше аналогия, согласно которой в старой модели бизнеса продавали предметы, а в новой продают доверие. Они действительно поддержали эту идею. В принципе, что они сделали? Отменили плату за продукты. Теперь можно загрузить из сети почти все, что производится Sun, и пользоваться им, если оно вам нужно. Предполагается, что при таких возможностях люди предпочтут покупать то, что поставляется с гарантией возмещения убытков, поддержкой и другими вещами, в которые я не стану углубляться в этом интервью. Время покажет, действен ли такой подход, но шансы на то велики.

Вы, вероятно, слышали о недавних проблемах штата Калифорния с системами, написанными на Коболе. Поможет ли строительство систем из мелких «кирпичиков» избегать в будущем проблем с устаревшим программным обеспечением?

Брэд: Я убежден в силе компонентного подхода, но не хочу излишне навязывать эту идею: компоненты решают не все проблемы. Компоненты – это изобретенный нами способ решения задач объема выше среднего: что-то из того, чем мы отличаемся от шимпанзе. Мы открыли, что можно решать задачи, просто перекладывая их на чужие плечи. Это называется разделением труда, только и всего. Вот этим люди отличаются от шимпанзе, которые до такого не додумались. Они умеют делать инструменты, у них есть язык для общения, поэтому, на первый взгляд, люди и шимпанзе мало отличаются друг от друга. Мы открыли способ решения проблем путем перекладывания их на других – посредством экономической системы.

Если мы найдем ответ, будет ли он как-то связан с прошлым, или это будет нечто совершенно новое?

Брэд: Эволюция – это замедленная революция. Это не две совершенно разные вещи. Но в упомянутой вами статье про Кобол также говорилось, что новейший стандарт Кобола поддерживает объектно-ориентированные расширения. Я не слежу сейчас за состоянием Кобола, но в 1980-е я ратовал именно за это: взять Кобол и добавить в него ОО, примерно как я сделал это для Си.

Это пример обычного, по-моему, хода вещей. Никто не стал выкидывать Кобол и заменять его чем-то другим. В него просто добавили то, чего в нем не хватало, сохранив все остальное.

Вы все еще считаете, что superdistribution (суперраспространение) имеет перспективы? Как насчет веб-приложений?

Брэд: Суперраспространение (в том смысле, как я его понимаю) применимо к объектам мелкой детализации. Для более грубой детализации (сервисов SOA, например) применимы более простые средства. Я положительно уверен в необходимости сильных стимулов. Но сейчас можно начать с более простых, чем объекты ООП, мест: в частности, с сервисов SOA. Раньше таких не было (не считая нашумевших «тонких клиентов»).

Проблема в том, что это все равно что заявить, что для решения палестинского конфликта нужно научиться ладить между собой. Очевидно, что это верно, потому что такое решение оказалось эффективным и для США, и для Южной Африки. Но оно совершенно неуместно в разгар боевых действий между владельцами и пользователями цифровых товаров. Ни одна из сторон не намерена идти на компромисс и учиться ладить с другой. Если вы предложите суперраспространение в качестве «решения» такого конфликта, камни полетят в вас с обеих сторон.

Является ли повсеместный доступ к сети необходимым условием строительства городов, следуя вашей аналогии?

Брэд: Если здания – это SOA, то несомненно, что для этого обязательно нужна сеть. Без сети вы не получите SOA. Но на практике в нынешние времена без сети вы и на горшок не сядете.

Робин Милнер (ML) предлагал параллельную работу множества малых туповатых машин. Вы ставите перед собой похожую задачу?

Брэд: Идея интересная. Я много занимался моделированием в военном деле, и такой подход очень привлекателен для возникающих там проблем. Возможно, есть и другие, неизвестные мне области приложений.

Качество как экономическое явление

Как можно улучшить качество ПО?

Брэд: Один из путей – хранить ПО на серверах, как в схеме «ПО как сервис» (Software as a Service, SaaS). Есть надежда, что такая система окажется экономически эффективной, хотя очевидно, что при этом пострадают определенные аспекты (секретность, безопасность, производительность и так далее).

В течение нескольких лет я работал в противоположном направлении, создавая экономическую систему для компонентов, локально работающих на машине конечного пользователя, но этот подход стал вызывать у меня все большие сомнения, потому что его обсуждение выливается в борьбу по поводу авторских прав на цифровой контент, которой не видно конца. Не знаю, как примирить тех, кто хочет производить и владеть, с теми, кто хочет отобрать право собственности. Без физического сохранения того, что считается собственностью, нам остается рассчитывать только на помощь законов, судов и юристов, что в итоге породит порядки полицейского государства. Представьте, что банки откажутся от сейфов и запоров, оставляя деньги на ночь на улице, и станут преследовать по закону тех, кто их украдет. Неутешительная картина.

Есть другое экономическое решение, сегодня весьма широко распространенное: реклама. Здесь пользователи не являются ни охотниками, ни добычей: они – приманка. Несмотря на успех Google, полагаю, этот путь не приведет ни к чему хорошему. Мы все свидетели того, к чему привела эта модель на радио и телевидении, и точно то же самое происходит с Интернетом.

Последним вариантом остается модель open source в каких-то из ее многочисленных вариантов и разновидностей – бесплатного, условно-бесплатного, добровольно оплачиваемого и так далее ПО. Сегодня я в основном работаю над этой моделью. Не потому, что лучше нее ничего нельзя придумать, а потому, что других не осталось, и она решает одну из проблем, в которых виновато само минобороны: замкнутость на проприетарности.

Что мешает нам создавать программные объекты такого же качества, как «реальные»?

Брэд: Экономическая система вознаграждения за усовершенствования.

И только?

Брэд: Это главный источник энергии, но после решения проблемы питания (экономической) потребуются дополнительные нововведения. Надежные примеры можно найти в биологии. Например, в Java есть

примитивные идеи инкапсуляции, которые можно сравнить с такими биологическими понятиями, как митохондрии, клетки, ткани, органы и так далее. В Java самой крупной капсулой оказывается вся JVM, а самой мелкой – пакет Java (.jar). Различные ухищрения с загрузчиками классов позволяют организовать поддержку других капсул (например, сервлетов, используемых в сервисах SOA).

Недавно я заинтересовался OSGI как первой серьезной попыткой создать развитый уровень инкапсуляции в промежутке от классов Java до JVM. Например, мои субкомпоненты SOA для безопасности и интероперабельности имеют вид связок OSGI.

Вы считаете, что для повышения качества ПО нужны лучшие стимулы?

Брэд: Если вы посмотрите, благодаря чему повысилось качество носков, свитеров и булочек Twinkie, то увидите, что двигателем этой системы служит экономика.

Именно негодная экономика ПО – причина его нынешнего примитивного уровня развития.

Фармацевтическая компания тратит миллиарды долларов на исследования, потому что от продажи лекарств, появляющихся в их результате, она получает десятки и сотни миллиардов. В их науке присутствует сильный монетарный фактор, хотя значительная часть исследований ведется открыто.

Брэд: Да, так можно что-то прихватить, чтобы стать владельцем. Когда дело касается кирпичей или кремниевых чипов, экономика подкреплена законами природы. Такой подход основан на действующих в обществе законах. Иными словами, нужно обзавестись адвокатами, патентами, профессиональными секретами, организовать судебные преследования, и такая модель может заработать. Только это очень противно.

По всем нашим представлениям не исключено, что банки станут открыто оставлять золото на ночь и преследовать тех, кто его украдет, вместо того чтобы запирать ценности в сейфах. Картина получается очень негуманная, но со всеми этими «законами об авторских правах в цифровую эпоху» (DMCA) мы вполне можем прийти к такому положению. DMCA, RIAA и подобные вещи вызывают у меня отвращение.

Может ли улучшить положение открытый доступ по модели open source?

Брэд: Думаю, open source – лучшая из нынешних экономических моделей. Моя работа большей частью связана с open source, потому что эта модель будет лучшей, пока SOA не достигнет такого уровня развития, что с ней придется считаться.

Но в принципе, рассуждая о глиняных кирпичах, я говорю об open source. Глиняные кирпичи делаются из бесплатного сырья, но тот, кто захочет что-то соорудить из них, столкнется с проблемой ужасной документации open source.

Смысл аналогии с глиняными кирпичами в том, что это приемлемо, когда нет других вариантов: лучше глиняные кирпичи, чем никаких. Но это лишь мое личное мнение, и оно поверхностно.

Какую роль играют Интернет и сети в разработке ПО?

Брэд: В наше время нельзя делать ПО без доступа к Интернету – даже представить себе это невозможно.

Но противоречие в том, что без правильной экономической модели Интернета или разработки ПО качество того, что мы здесь получаем, значительно ниже, чем на рынке «осязаемых» товаров.

Если начать рассматривать ПО как сервисы и компоненты – на более высоком, чем язык, уровне, – изменится ли от этого экономика ПО? Если продавать доверие, не разрушит ли это рынок товаров?

Брэд: Я не волшебник, чтобы предсказывать будущее. Не знаю. Может быть, через тысячу лет... – если столько времени понадобилось строительной индустрии, то можно ли думать, что нам потребуется меньше? Полагаю, через тысячи лет мы придем к этому, но меня уже не будет. Трудно предсказывать такое далекое будущее.

Кто-то скажет, что зато нам не мешают законы физики.

Брэд: Да, это должно быть преимуществом. Похоже, это больше вредит нам, чем помогает. Прежде всего, в результате мы лишились экономической модели, а это большая потеря.

Если бы вы делали Objective-C сегодня, то выбрали бы для него модель open source?

Брэд: Слишком многое нужно учесть, чтобы ответить на этот вопрос. Open source не было в те времена, а нам как-то нужно было зарабатывать на жизнь. Если бы сегодня у меня была надежная и не слишком обременительная работа, возможно, я тратил бы свободное время на open source. В конце концов, желание получить доход сходно с силой тяжести – слабейшей, а потому легко уступающей более мощным силам (скажем, стремлению к самореализации). Но доход является также самым дальнобойным критерием, а потому в долгосрочной перспективе его нельзя не учитывать.

Вы хотите сказать, что без финансовой поддержки проекты open source не являются серьезной альтернативой коммерческому ПО?

Брэд: Нет, я имел в виду то, что сказал. Доход может иметь разный вид: для многих из нас он может выражаться в росте репутации.

Веб-приложения — это положительное явление?

Брэд: SOA размещает программы на серверах, где конечные пользователи не могут их контролировать. Из этого со временем может возникнуть экономическая система для SaaS.

Этот подход пока не дал сколь-нибудь заметных результатов, поскольку SaaS все еще находится в начальной стадии своего развития, но предвижу, что такой подход может привести к экономической системе, способной улучшить ПО тем же путем, каким улучшаются осязаемые вещи, — под действием экономических рычагов.

Образование

У вас есть степени бакалавра по органической химии и математике и докторская степень по математической биологии. Как вы пришли к созданию языка программирования, имея такую подготовку?

Брэд: После завершения постдокторантуры я осмотрелся и понял, что меня больше интересуют компьютеры, чем открывавшиеся передо мной в то время возможности.

Влияет ли университетское образование на ваши взгляды, касающиеся разработки ПО?

Брэд: Несомненно. Постоянно.

Если вы изучаете экологические системы, то ПО во всех отношениях кажется вам экологической системой, за исключением того, что в нем нет ничего, подобного физическим законам сохранения — массы или энергии. Наш продукт состоит из битов, которые так легко копируются, что покупка, продажа и владение ими затруднены. Экономическая система, экология не работают.

Если бы леопард мог реплицировать то, чем питается, с такой же легкостью, как мы реплицируем ПО, был бы невозможен прогресс ни леопарда, ни тех, на кого он охотится.

У ПО есть постоянная проблема прав собственности и вознаграждения за продукты и труд.

Не сместился ли интерес исследований компьютерной науки из академической сферы в производственную?

Брэд: Никогда не считал себя «ученым в области компьютерных наук» или академическим работником (хотя и проработал какое-то время в этой сфере). Все остальное время я работал в условиях производства и, естественно, отдаю предпочтение ему.

При этом успехи академических исследований не производили на меня впечатления до недавней поры, когда я заметил растущее академиче-

ское участие в таких организациях, как W3C, Oasis и других, занимающихся выработкой стандартов. С моей точки зрения, это замечательно.

Мне нравится следующая цитата с virtualschool.edu:

> Таким образом, компьютерная наука не мертва. Она просто никогда не существовала.
>
> Главная идея новой парадигмы, которую я пытаюсь проповедовать в своей новой книге, состоит в том, что вирус болезни, симптомы которой мы называем кризисом программного обеспечения, возник из-за того, что мы имеем дело с субстанцией, состоящей из битов, а не атомов, и порожденной исключительно людьми, а не природой.
>
> Но поскольку эта субстанция не подчиняется законам сохранения массы, коммерческие механизмы, побуждающие людей совместно трудиться над изготовлением карандашей или лент для багажных конвейеров, совершенно не работают. Без развития торговли не может возникнуть развитый общественный порядок, поэтому мы застряли в том примитивном состоянии, когда все изготавливается из простейших элементов.
>
> Поэтому все уникально, и ничто выше уровня отдельных битов не обладает повторяемостью, оправдывающей экспериментальное изучение. Вот поэтому нет такой вещи, как компьютерная наука.

И десять лет спустя компьютерная наука по-прежнему мертва?

Брэд: Мне трудно что-либо добавить к тем наблюдениям, которые вы привели. В конце концов, все зависит от того, что понимать под словами «компьютерная наука» и «мертва»... а также от желания воевать с той или иной группировкой, честными или иными методами, которого у меня нет совершенно.

Единственное, что я могу добавить по существу, так это что пока программное обеспечение уникально и не управляется физическими законами (заметим, что растущее применение стандартов смягчает это давнишнее утверждение), мы имеем дело с некой общественной наукой, а это большое отличие от физических наук.

Изменились ли ваши взгляды под воздействием ООП, суперраспространения и SOA?

Брэд: Если хотите знать, мои интересы связаны вовсе не с языками программирования, а с выяснением причин, по которым так трудно программировать, хотя с другими вещами люди справляются без усилий. Например, с тем, чтобы кормить обедом миллионы жителей Нью-Йорка, поспевать за законом Мура, вычеркивать планеты (пока только одну), продолжать производство автомобилей и так далее. Это действительно удивительное явление, и ни один другой биологический вид на

такое не способен. Я все время пытаюсь понять, как это происходит, и нельзя ли это приспособить для создания программ.

В действительности умение справляться с такого рода сложностью настолько широко распространено, что я назвал бы его врожденным, если бы не сохранившиеся общества охотников-собирателей, которые его не открыли, и где все однородны... каждый возделывает свой огород, охотится на своего зверя, строит свои дома. Но даже там можно найти инкапсуляцию и разделение труда: женщины готовят еду, мужчины охотятся, дети помогают, старики дают советы.

Но в современной жизни получила безудержное развитие специализация – специфическая человеческая способность отделить кусочек своей задачи и озадачить им кого-то другого. Я приготовлю обед, если ты постоишь за прилавком, еще кто-то завезет продукты, еще кто-то вырастит пшеницу, еще кто-то изготовит удобрения, и так до бесконечности, до добычи руды в шахте. Это настолько очевидно, что мы принимаем его как данность, о чем свидетельствует даже отсутствие в словаре слов для описания многочисленных уровней производства, участвующих в таком обычном деле, как накрывание стола для обеда. Обратите внимание на две составляющие: 1) умение отделить часть задачи, то есть определить ее отделяемую часть, и 2) умение поручить ее кому-то другому. Я еще вернусь к этому ниже.

До появления ООП способность ограничить задачу была очень несовершенной. Чем больше людей работало над задачей, тем больше она выходила из-под контроля. Множество файлов с невразумительными именами (венгерская нотация), конфликты внешних переменных, почти полное отсутствие инкапсуляции. В точности как задача, с которой сталкиваются при разработке больших интегральных схем на уровне логических элементов. Как они решили ее? Инкапсулировав кучу логических элементов внутри чипа. Я буду проектировать чипы. Ты будешь паять из них платы. Objective-C был точным отражением этой идеи.

Разумеется, это было лишь началом, а не концом (вот почему меня иногда раздражает, если в центр внимания ставят язык). Языки – это инструменты, которые выбирают или откладывают в сторону в зависимости от решаемой задачи. И то же самое на каждом уровне. Например, по прошествии 20 лет та же проблема снова возникает с библиотеками Java, потому что они выросли до таких размеров, что никто не может их охватить взглядом и эффективно использовать (это я про J2EE).

Вернемся к суперраспространению, возникающему благодаря возможности поручить свои задачи кому-то другому, если речь идет о товарах, изготовленных из битов и не подчиняющихся законам сохранения, на которых с античных времен основывалось вознаграждение. Что заста-

вит кого-то заняться решением моих задач? Зачем это кому-то нужно, если я могу взять то, что он производит, и приспособить для собственных нужд? Моя книга о суперраспространении излагала возможное решение в контексте объектов ООП мелкой детализации, когда вознаграждение основывается на измерениях использования битов, а не на их покупке, как делается сегодня.

Но мерить использование скучно и трудно, если товары допускают значительное вмешательство со стороны недобросовестных пользователей. Напротив, сервисы SOA этому не подвержены. Они выполняются на сервере, которым управляет владелец, а не пользователь, и за их использованием можно следить, не опасаясь разнообразного мошенничества. Проблема честного дележа доходов для всех уровней организации производства сохраняется, но это бухгалтерская проблема, а не проблема защиты объекта от постороннего вмешательства.

Любопытно, что при подъеме по иерархии от ООП до JBI/SCA и SOA сохраняются все старые проблемы компьютерной науки. Единственная разница в том, что стандартное представление данных (схемы XML) устраняет необходимость специализированных парсеров для каждого представления. В результате генерация кода избавляется от деревьев DOM, строящихся стандартными парсерами XML под управлением все более графических «языков», таких как UML и DODAF. Все происходящее просто восхищает, хотя бы в сравнении с нескончаемыми спорами о том, какой язык ООП «лучше».

Все это я увидел и попробовал (работая с Си, Objective-C, Perl, Паскалем, Java, Ruby и так далее). Информационные технологии наскучили мне, и я занялся более интересными проблемами.

Каковы недостатки наших способов обучения разработке ПО?

Брэд: Лично я, занимаясь преподавательской работой, пытался информировать слушателей о недостатке надежных экономических рычагов, влияющих на качество ПО. Как правило, экономические вопросы отсутствуют и игнорируются в учебных программах по компьютерным наукам и программированию, им не уделяется никакого внимания.

Почему компьютерные науки не являются наукой в подлинном смысле этого слова?

Брэд: Каждый раз, когда вы сталкиваетесь с новым программным продуктом, он оказывается чем-то совершенно новым и уникальным. Какая может быть наука, если все предметы ее уникальны?

Если вы будете день за днем исследовать золото или свинец, то сможете измерять их свойства и применять научные методы исследования. С программным обеспечением это невозможно.

12

Java

Язык Java первоначально разрабатывался для программирования бытовых электронных устройств. Он приобрел популярность в связи с ростом популярности веб-броузеров и апплетов, но благодаря автоматическому управлению памятью, виртуальной машине, большому набору библиотек и относительной независимости от платформы («write once, run anywhere»[1]) язык превратился в универсальный. Несмотря на раскрытие компанией Sun Microsystems исходного кода и бесплатность языка, компания сохраняет некоторый контроль над его развитием и библиотеками посредством формальной процедуры Java Community Process.

[1] Принцип «write once, run anywhere» (WORA): напиши один раз – запускай где угодно. – *Прим. пер.*

Сила простоты

Вы сказали, что простота и мощь — это антиподы. Нельзя ли подробнее?

Джеймс Гослинг: Действительно мощные системы часто становятся сложными. Возьмите, например, спецификацию Java EE. Там сказано все необходимое о транзакциях и постоянстве (persistence), и это действительно очень мощные возможности. Но в первых версиях Java ничего этого не было.

Система была достаточно проста: можно было сесть и без особого труда разобраться с ней. Если ограничить Java языком и основными API, то и сейчас все окажется просто. Но при обращении к более мощным подсистемам, таким как Swing или Java EE, вы начинаете захлебываться. Возьмите библиотеки OpenGL – они очень мощные, но ни в коей мере не простые.

В частности, в OpenGL необходимо твердо понимать, как работает идеализированная графическая аппаратура.

Джеймс: Верно. Кажется, это Эйнштейна цитируют, говоря, что «системы должны быть как можно более простыми, но не проще того».

Постоянна ли простота или сложность в рамках системы? Ларри Уолл говорит, что сложность напоминает водяной матрас: если ужать сложность в одном месте языка, то она выпятится в другом. Не стала ли простота базового языка Java причиной сложности, свойственной, скажем, библиотекам?

Джеймс: Мне больше нравится сравнение с игрой «ударь крота». Часто люди говорят: «О, я решил проблему!», а немного покопавшись, обнаруживаешь, что они не решили проблему, а просто сдвинули ее в другое место.

Трудность борьбы с подобными недостатками языка связана, в частности, с тем, что языком пользуются разные люди. Если ввести в язык особую поддержку транзакций, то тем, кто использует транзакции, станет немного легче, но всем остальным это может значительно осложнить жизнь.

Они платят за теоретическую возможность.

Джеймс: Они платят за теоретическую возможность, а в зависимости от того, как она реализована, они, возможно, платят даже больше.

Платформа Java используется больше десяти лет и стала зрелой. Можно ли вернуть простоту системе, когда она достигла такого состояния?

Джеймс: Не знаю. Мне кажется, здесь нет однозначного ответа.

Ответ будет отрицательным, если учесть, сколько кода использует все эти навороты. Было бы прекрасно, если бы можно было избавиться от всех сложных вещей, которые мы накрутили. Но если вы посмотрите на действующие приложения, то увидите, что все это активно в них используется.

У вас была попытка заменить AWT на Swing, но люди продолжают использовать AWT.

Джеймс: Да, и удивительно, как много таких людей. Отчасти это связано с тем, что AWT сохраняется в сотовых телефонах.

Но дело и в другом: несмотря на противоположные заявления в прессе, Java невероятно широко используется в настольных приложениях, особенно при создании промышленных приложений. Их десятки тысяч, в них используются бета-версии библиотек, и они отлично работают, поэтому мало что заставит людей отказаться от них.

Ответ будет в некотором смысле положительным, потому что у объектно-ориентированных систем с достаточным уровнем абстракции есть приятная особенность, которая позволяет, когда меняются окружающие условия, найти, если вы того хотите, лучшие способы работы, и они не будут конфликтовать с прежними.

Для этого существуют пространства имен, абстракции и инкапсуляция.

Джеймс: Верно. В Java EE мы пошли на принципиальные изменения в пользу упрощения в версии EE5. Если вы посмотрите, что представляет собой Java EE сегодня, и возьмете руководство по EE5, то там все не так страшно в отношении сложности. Но если вы возьмете старое руководство EE и попытаетесь разбираться в обеих версиях, вам будет тяжело. Мы достигли значительного прогресса в новой версии EE, но тем, кому приходится стоять одной ногой в новой версии, а другой – в старой, не позавидуешь.

Обеспечить обратную совместимость всегда трудно. Sun намеренно проводит техническое обсуждение того, должны ли старые программы для Java 1.1 выполняться в новейших JVM?

Джеймс: Довольно любопытно, что сама JVM фактически не участвует во всех этих обсуждениях, потому что JVM отличается примечательной стабильностью. Все проблемы связаны с библиотеками. Библиотеками значительно легче управлять, чем базовой ВМ. Они проектируются как модульные. Они приходят и уходят. На самом деле, можно построить такие загрузчики классов, что они поделят пространство имен, и у вас фактически окажется две версии Java AWT. Есть масса инструментов, с помощью которых можно этим управлять. Если же вы беретесь за

саму виртуальную машину, положение значительно осложняется. Но с ней возникало не так много проблем.

Байт-коды на практике довольно стабильны. Вся работа над ВМ сводится к разработке высокотехнологичных оптимизаторов.

Я слышал, что в мире Java фактически два компилятора. Есть компилятор в байт-код Java и есть компилятор JIT, который фактически заново перекомпилирует все особым образом. Все ваши жуткие оптимизации относятся к JIT.

Джеймс: Точно. Мы сейчас почти всегда добиваемся превосходства над весьма неплохими компиляторами Си и C++. Если взять динамическую компиляцию, то, запуская компилятор в самый последний момент, вы получаете два преимущества. Во-первых, вы точно знаете, на какой элементной базе работаете. Очень часто при компиляции Си-кода приходится компилировать его для некой обобщенной машины с архитектурой *x*86. И почти всегда получаемые двоичные модули оказываются не слишком хорошо настроенными на особенности конкретной архитектуры. Если вы загрузите последнюю версию Mozilla, то она сможет работать почти на всех архитектурах ЦП Intel. Двоичный модуль для Linux практически один. Он довольно общий и скомпилирован с помощью GCC – не самого лучшего компилятора Си.

Когда работает HotSpot, ему точно известно, с каким чипсетом он имеет дело. Он точно знает, как устроено кэширование. Он точно знает, как устроена иерархия памяти. Он точно знает, как работают все взаимоблокировки конвейеров в ЦП. Он знает, какое расширение инструкций поддерживает данный чип. Он проводит оптимизацию именно для той машины, на которой вы работаете. С другой стороны, он реально видит приложение во время исполнения. Он может получить важную статистику. Он может выполнять встраивание команд, недоступное компилятору Си. Вставки, выполняемые в Java, просто поразительны. Добавьте к этому работу управления памятью вместе с современными сборщиками мусора. Современные сборщики мусора позволяют чрезвычайно быстро распределять память.

Просто выкинуть указатель.

Джеймс: Буквально выкинуть указатель. `New` в Java world обходится примерно во столько же, во сколько `malloc( )` в Си. В некоторых тестах – в 10 раз лучше, чем `malloc( )`. Там, где много маленьких объектов, `malloc( )` работает довольно плохо.

Если уж мы заговорили о Си: как нужно проектировать системный язык программирования? Что нужно учитывать, когда разрабатываешь язык программирования, который может стать системным?

Джеймс: Я стараюсь не размышлять слишком много о языках и характеристиках. В тех случаях, когда я занимался разработкой языка, что, к несчастью, бывало слишком часто, это всегда мотивировалось решаемой задачей. В каком контексте будет происходить исполнение? Что будут делать люди с его помощью? Что нового в мире? Для Java особые условия заключались в сети. Повсеместное проникновение сети заставляет несколько иначе смотреть на вещи, потому что возникает масса побочных результатов. Вроде компьютера в комнате вашей бабушки.

Или карманного устройства, совсем не похожего на компьютер.

Джеймс: В долгий перечень вытекающих из этого изменений входит и то, что вы больше не хотите сталкиваться с «синим экраном смерти». Не должно быть сложных функций для установки. Поэтому в Java были включены весьма мощные механизмы изоляции отказов. Многие этого просто не замечают, но указатели памяти, сборка мусора, обработка исключений – все это служит изоляции отказов. Все сделано для того, чтобы гарантировать непрерывную работу даже в случае мелких осечек. Если вы едете по дороге и у вас сломалась дверная ручка, машина от этого не остановится. Одна из проблем большинства программ на Си в том, что совершенно невинное действие приводит к грубой ошибке разыменования указателя, и все с грохотом рушится.

Нет совершенно никакой возможности предсказать, что выйдет из строя и где, равно как случится ли авария и когда.

Джеймс: Да. И я всегда считал это совершенно недопустимым. В те дни, когда создавался Си, первое время в Sun эффективность была превыше всего. Такие вещи, как проверка границ массива, были абсолютно неприемлемы. Когда появилась спецификация Java, там было сказано, что «проверку допустимости индекса массива отключить нельзя». Нет такой вещи, как «отсутствие проверки индекса массива». В этом отношении произошел кардинальный отход от Си, потому что в Си вообще нет проверки индексов, а у нас это неотъемлемая часть спецификации.

Если вдуматься, есть ли здесь вообще массивы?

Джеймс: Да. Есть сложение и есть довольно странный синтаксис сложения. Одно из чудес современных компиляторов состоит в том, что они могут доказать теорему путем проверки потенциально всех индексов. Хотя не всем нравится, что такие вещи, как проверка указателей, не могут быть отключены, они заблуждаются. Это очень правильно. Это не оказывает отрицательного влияния на производительность. Можно организовать какую-то проверку вне цикла, но внутри него она просто режет глаз.

Если бы в Си была решена проблема строк, когда невозможно узнать длину строки, потому что она оканчивается нулем, у нас уже лет 40 на-

зад мог появиться более быстрый Си. Раньше для меня это была наибольшая неприятность в Си.

Джеймс: Да. Люблю Си. Я очень долго профессионально работал на Си. Я перешел на Си, когда им еще почти никто не пользовался. Первые компиляторы Си работали на машинах с 32K памяти и для программ, которые должны были выполняться в 32K памяти. Первые компиляторы Си были изумительны, но сейчас трудно найти даже наручные часы, где будет всего 32K памяти. На кредитной карточке памяти обычно больше 32K.

Дело вкуса

Как вездесущее подключение к Интернету влияет на концепцию языка программирования?

Джеймс: Вопрос о том, как сеть влияет на разработку языка программирования, весьма обширен. Как только появляется сеть, вам приходится иметь дело с разнородностью, со связью, тщательно продумывать последствия отказов – значительно больше заниматься надежностью.

В частности, нужно думать о том, как строить системы, которые будут устойчивы и смогут продолжить работу в условиях частичных отказов, потому что большинство создаваемых систем, если они представляют какой-то интерес, это такие системы, в которых постоянно что-то не работает.

При этом к ПО традиционно подходили с меркой «все или ничего»: либо оно работает, либо нет. В значительной мере такого рода заботы нашли отражение в механизме исключений Java, системе строгой типизации, сборке мусора, виртуальных машинах и так далее. Хочу подчеркнуть, что сети оказали глубокое влияние на организацию Java – языка и виртуальной машины.

В чем, по вашему мнению, больше всего проявилось влияние ваших взглядов на проектирование и программирование? Можете ли вы в системах, проектированием или разработкой которых занимались, указать то, что считаете фирменным знаком Гослинга?

Джеймс: Ох. Хотел бы я, чтобы в жизни все было так просто.

Я встречал архитекторов, утверждавших, что у них есть «свой фирменный стиль» в подходе к работе. Я скорее не придерживаюсь определенного стиля. Если и считать что-то моей отличительной чертой, то читавшие написанный мной код ее вам назовут, потому что люди с ума сходят, когда им приходится сопровождать написанный мной код, ибо я обычно помешан на эффективности кода.

Я не усердствую с встраиванием машинного кода, но склонен применять сложные алгоритмы, когда вполне можно обойтись и простыми. Я горы сверну, чтобы – ну, я склонен применять кэширование чаще, чем это обычно принято. Я просто рефлективно размещаю буферы тут и там, потому что, не видя кэша, начинаю нервничать.

Вспоминается случай, когда на вопрос «почему здесь выполняется линейный поиск в массиве, когда можно применить быструю сортировку?» я получил ответ «потому что в нем будет от силы семь элементов, и применять быструю сортировку слишком накладно».

Джеймс: Ну, я бы не стал устраивать сложную структуру данных для семи элементов.

Многие программисты не задумываются о практических последствиях таких решений.

Джеймс: Что меня бесит, так это когда в систему включают то, что годилось во время разработки: обойдемся простым линейным поиском. Они же знают, что в реальных условиях там будет 100 000 элементов. Если я тестирую систему на 10 элементах, то можно обойтись и линейным поиском, но они всегда говорят: «Эффективностью я займусь когда-нибудь потом». Как у Роберта Фроста: «Хотя и догадывался в тот час, Что вряд ли вернуться выпадет случай»[1]. То же самое часто происходит с кодом. У меня есть сильное подозрение, что обещания вернуться позже и все исправить очень редко выполняются, а потому вокруг нас полно систем, работающих невероятно медленно.

Это следствие нехватки времени, или лени, или легкости, с какой теперь могут программировать непрограммисты?

Джеймс: Всего понемногу. А еще и того, что новые машины работают уже на трех гигагерцах, и бесконечные циклы выполняются на них за вполне приличное время. :)

Как вы пришли к тому, что Java будет использовать виртуальную машину?

Джеймс: Это очень помогает решать проблемы переносимости. Это удивительно способствует надежности и, как ни странно, заметно увеличивает производительность. Компиляция just-in-time позволяет существенно повысить эффективность программ. Так что кругом одни плюсы. И отладка заметно облегчается.

Думаете ли вы сейчас, что JVM можно было сконструировать иначе, или удовлетворены тем, как она работает?

---

[1] Цитата из стихотворения «The Road Not Taken» («Другая дорога») в переводе Григория Кружкова. – *Прим. ред.*

Джеймс: Я ею очень доволен. Конструкция JVM оказалась, пожалуй, самой устойчивой частью всей системной архитектуры. Если бы я освободился от всех дел и стал искать, где приложить силы, то в эту сторону и смотреть бы не стал, потому что она в очень хорошей форме: нет там важных проблем, с которыми стоило бы повозиться. Кроме того, над ней лет десять трудилось много талантливых людей, в основном специалистов по компиляторам с докторскими степенями.

Внесли бы вы какие-то изменения в конструкцию JVM с учетом ее нынешней популярности в других языках?

Джеймс: Возможно, кое-что я бы и подправил. Как раз в связи с этой темой сейчас обсуждается ряд вопросов проектирования языков, и оказывается, что на редкость трудно найти, какие изменения в ВМ могли бы существенно помочь другим языкам.

Если где-то обнаруживаются крупные проблемы, то они носят характер философских аспектов виртуальной машины. Например, реализовать такие языки, как Си и C++, на виртуальной машине Java очень сложно, потому что у нас запрещены непосредственные операции с указателями.

Разрешив действия с указателями, мы столкнулись бы с большими проблемами надежности, и поэтому решили, что никогда не сделаем этого. Подчеркну: это вызвало бы огромные проблемы как надежности, так и безопасности. Си и C++ на JVM? Никогда.

Вы привели пример, что когда едешь на машине, она не остановится из-за того, что, скажем, перестало работать радио. Может быть, для сборки программ нужны какие-то другие конструктивные элементы, чтобы избавиться от проблемы полной остановки из-за выхода из строя отдельных частей?

Джеймс: Проблема в значительной мере связана с тем, как некоторые языки организованы на низком уровне. «Все с треском встало» – это большая проблема для Си из-за того, как в нем реализованы указатели.

Как только из-за указателей возникает ошибка обращения к памяти, система пишет дамп ядра, и привет. В то же время в Java, во-первых, ошибка указателя менее вероятна, а во-вторых, если она случится, ее действие можно локализовать.

Система исключений эффективно снижает объем ущерба: если радио вышло из строя, можно попробовать просто отключить его.

Те, кто строит на Java большие промышленные системы, много времени тратят на то, чтобы обезопасить разные компоненты от взаимного влияния и обеспечить корректную обработку их отказов.

Если взглянуть, например, на спецификацию Java Enterprise, то там обнаружится много такого, что обеспечивает устойчивую работу среды при возникновении отказов.

Является ли объектно-ориентированная парадигма по-прежнему совершенной в этом отношении?

Джеймс: В данное время объектно-ориентированная парадигма действует очень хорошо. Всякого рода дебаты касаются второстепенных вопросов – различные теоретико-языковые споры ведутся, по существу, по незначительным модификациям, но основная идея объектно-ориентированного программирования оказалось удивительно удачной и не обнаружила каких-либо серьезных изъянов.

Иногда создается впечатление, что работа с объектами требует от разработчиков двойного труда. Сначала им приходится конструировать компонент для повторного использования. Затем, если нужно внести изменения, приходится писать нечто, точно заполняющее брешь, оставленную прежним компонентом. Фактически, я почти не вижу, как можно использовать объекты для более совершенных конструкций, без того чтобы эти объекты все сильно усложняли.

Джеймс: Конечно, нет сомнений в том, что объектно-ориентированная разработка в определенной мере требует хорошего вкуса. Можно видеть массу примеров того, как люди теряют чувство меры и делают несколько странные вещи, но это не опасно. Применение интерфейсов и объектов оказалось чрезвычайно успешным, и даже просто необходимость обдумывать отношения между подсистемами оказывается очень полезной.

Вся конструкция, когда можно разобрать систему на части и вынуть их по отдельности, очень сильно помогает при дальнейшем развитии, отладке, изоляции отказов и во многих других случаях.

Да, чтобы должным образом применять ее, нужно обладать некоторым вкусом, но это не так сложно. И она показала свою очень высокую ценность – в отличие от спагетти-кода, где все напрямую интегрируется между собой, и когда вы хотите изменить что-то одно, приходится менять все остальное. Бывает просто невыносимо.

Там, где не практикуется объектно-ориентированный подход, жизнь ужасна. ООП проявляет себя во всем блеске, когда системы увеличиваются в размерах, когда над ними работают большие команды, когда системы развиваются во времени.

В большей степени удается обеспечить модульность, отделить одну задачу программирования от другой и сузить круг того, что нужно дополнительно изменить, если что-то меняется – чрезвычайно полезная методология.

Параллелизм

Часто говорят о прекращении действия закона Мура, поскольку системы растут вширь, но необязательно становятся быстрее. Вы согласны с этим?

Джеймс: Да. Закон Мура относится к количеству логических элементов, и нетрудно убедиться, что в течение многих лет количество логических элементов увеличивалось в точном согласии с законом Мура. Но в отношении тактовой частоты – это какое-то совпадение, что столь долго можно было интерпретировать его и в отношении тактовой частоты.

Потребность в лучшей поддержке параллелизма влияет только на реализацию или и на саму конструкцию?

Джеймс: На конструкцию оказывается громадное влияние, хотя есть определенные различия, зависящие от предметной области.

В большей части математических программ, для того чтобы обеспечить работу математического ПО в условиях многопоточности, приходится существенно менять алгоритм. Но что касается промышленного ПО, то в значительной мере оно выполняется в рамках среды.

Например, для приложений Java есть среда сервера приложений Java EE, и в среде EE приложения могут и не подозревать, что выполняются в многопоточном окружении: это контейнеру, серверу приложений, нужно владеть всем, что относится к многопоточности, уметь работать с кластерами, мультипроцессорами и так далее.

Эти задачи решаются на ином уровне абстракции, благодаря чему программисту не нужно думать о них, и это хороший подход для промышленного ПО, для которого часто характерно выполнение не связанных между собой многочисленных мелких транзакций, просто одновременных.

Если это математическое ПО, то тут все тесно переплетено, тут общие данные и тому подобное, и все становится намного сложнее. Здесь нет однозначного ответа.

Потребуются ли нам новые языки или достаточно будет новых инструментов и новых библиотек? Может быть, отправить на переподготовку всех программистов, чтобы научить их мыслить по-новому?

Джеймс: Тут могут быть разные варианты. Думаю, предметная область имеет огромное значение. Для большинства промышленных приложений, интенсивно использующих транзакции, можно организовать такие среды, как Java EE, которые достаточно просто решают проблемы многопоточности. Если у вас работает 128-ядерная машина Sun – какие, надо сказать, действительно существуют, – разработчики ничего

даже знать об этом не будут, а все ядра будут использованы в лучшем виде и без особых усилий.

Даже не требуется особо подготовленный программист: достаточно хорошего программиста, не являющегося экспертом по многопоточности, а Java сама обо всем позаботится.

Джеймс: Верно. Вычислительная среда в значительной мере берет на себя все проблемы, связанные с потоками. Трудности возникают при столкновении с вычислительными задачами вроде разного рода моделирования. Всякие алгоритмы на графах и вычисления трудно поддаются распараллеливанию. Отчасти это вызывается необходимостью обеспечить доступ к структурам данных, установить блокировки и так далее. Порой это неотъемлемая сложность структуры данных и выбора правильного алгоритма. Задача о коммивояжере – пример исключительной трудности. Есть задачи попроще, вроде рендеринга трассировкой лучей, но это пример специфичной области, где оказывается, что если взять отдельные пикселы, то они будут полностью независимы.

Эта задача распараллеливается до уровня пикселов, если у вас достаточно мощная аппаратура.

Джеймс: Верно. На практике получается неплохо. Это делают большинство хороших трассировщиков лучей. Труднее такие вещи, как гидродинамические расчеты, заложенные в основу прогноза погоды или предсказания падения самолета, потому что разные части потока существенно влияют друг на друга. В этом случае не так сложно организовать систему с общим адресным пространством, но очень трудно организовать кластер, где адресные пространства раздельны. Например, алгоритмы вычислительной гидродинамики очень быстро исчерпывают себя из-за стоимости обмена данными между узлами кластера. В многоядерной системе они работают гораздо лучше, но все очень сильно зависит от конкретного алгоритма.

Что мне нравится в функциональных языках, таких как Scala, так это то, что если записать на Scala числовой алгоритм, то у компилятора появляется много возможностей рассудить, чем занимается ваша программа. Он может автоматически организовать выполнение алгоритма в многопоточной, многоядерной распределенной системе.

Это из-за того, что Scala – чисто функциональный язык?

Джеймс: Он не чисто функциональный. Одна из причин, по которым он многими успешно применяется, – его двойственность. Можно писать на нем, как на Java, а можно – функционально.

Есть ли такие предметные области, где многопоточность с общей памятью эффективнее функциональности?

Джеймс: Для промышленных приложений метод многоядерных распределенных систем на базе исполнительной среды действует действительно очень успешно. Не думаю, что такие системы, как Scala, имеют большие преимущества. Интересные вопросы возникают, когда вы занимаетесь чем-то вроде задачи о коммивояжере.

Осознанное решение, источник которого видится в Green Project или Oak Project, состоит в том, что при проектировании языка для работы с сетью в условиях повсеместного распространения сетей и многопоточности необходимо создать примитивы для синхронизации, а безопасность потоков должны обеспечивать основные библиотеки.

Джеймс: У нас есть множество механизмов, обеспечивающих безопасность потоков. На самом деле, здесь особый случай, потому что обычно это означает, что при работе в системах с единственным ЦП есть определенные издержки. Но в данном конкретном случае получается так, что на помощь приходит абстракция, потому что более абстрактные API и интерфейсы, такие как виртуальная машина Java, оказываются весьма абстрактными в сравнении с реальными машинами. Их основные механизмы способны весьма сильно адаптироваться. В случае многопоточного сбора данных оказывается, что BM HotSpot чудесным образом понимает, что одноядерные машины не такие, как остальные.

При компиляции на лету можно заявить: «Мне не нужно беспокоиться о синхронизации этой части, потому что я точно знаю, что взаимная блокировка невозможна: у нас никогда не будет конкуренции между потоками за этот конкретный участок памяти».

Джеймс: Да, все происходит автоматически и прозрачно. Никто ничего не замечает. Аналогичная история с 64-разрядными указателями.

Когда нужно, чтобы Си-приложение работало в 64-разрядной среде, приходится всюду делать оболочки. Для Java-приложений не нужно делать абсолютно ничего.

Разработка языка

Если бы не было Java, вашим любимым языком мог стать Scala?

Джеймс: Да, возможно.

Что вы думаете обо всех этих новых языках, которые кажутся не исследовательскими проектами, а довольно серьезными попытками построить настоящий мощный язык поверх JVM?

Джеймс: Думаю, что это замечательно.

Вы не боитесь, что они возьмут все ваши замечательные идеи и обойдут вас?

Джеймс: Нет, все существенные элементы Java заключены в ВМ. Это то, на чем основана интероперабельность. В некотором отношении Java и синтаксис ASCII были задуманы так, чтобы программисты на Си и C++ чувствовали себя комфортнее. Это неплохо удалось. Большинство программистов на Си и C++ может, взглянув на код Java, сказать: «О, да мне тут все понятно!»

На тактическом уровне мне понятно, что происходит, даже если я не знаю деталей API.

Джеймс: Одна из важных задач проекта именно в этом и состояла. Если задавать абстрактные вопросы вроде «какой из языков программирования – лучший на свете?», то такой задачи не возникает. Лично я считаю Scala чертовски интересным. Проблема Scala в том, что это функциональный язык, а большинству претит соответствующий образ мыслей.

Если бы я все бросил и разработал язык на собственный вкус, у большинства от него, наверное, вообще крыша поехала.

Тогда, может быть, Лисп?

Джеймс: Лисп едва ли, но кое-что из Лиспа я бы взял.

Билл Джой как-то сказал, что вашей целью было силком подтащить программистов на C++ поближе к Common Lisp.

Джеймс: В каком-то смысле это так, если посмотреть, что писалось про JVM.

Что ВМ не обязательно должна быть медленной или что повсеместная сборка мусора очень помогает программистам.

Джеймс: Люди явно не могли понять, что сборка мусора очень полезна для надежности и безопасности. Посмотрите, какого типа ошибки возникают в крупных системах: сплошь и рядом они связаны с управлением памятью. Некоторое время я собирал статистику по всем обнаруженным мной ошибкам, в частности по затратам времени на них. Одна из отличительных особенностей ошибок доступа к памяти – непомерное время, затрачиваемое на их обнаружение. Я поклялся, что больше ни часа времени не потрачу на это дело.

Я написал первые два сборщика мусора для JVM. Сборщики мусора мучительно трудно отлаживать, но после отладки им ничего не нужно, и они работают. Сейчас у нас есть несколько сборщиков мусора, действующих на солидной научной основе.

Какими были те сборщики мусора, которые вы написали сначала?

Джеймс: Мне нужен был такой сборщик, который мог бы работать в очень маленьком адресном пространстве. Принципиально это был алгоритм пометок (mark-and-sweep) со сжатием и некоторыми возмож-

ностями асинхронного выполнения. Для более современных конструкций было слишком мало памяти. Если были ссылки (handles), то они помогали сжатию.

Дополнительная косвенность указателей, но зато можно копировать.

Джеймс: Так как я пытался работать с библиотеками Си, начальные версии были полуточными. Точность обеспечивалась для указателей в куче, а неточность действовала в стеке, потому что фактически сканировался стек Си, чтобы узнать, нет ли там чего-то похожего на указатели.

Это единственный известный мне хороший способ работы со стеком Си, хотя можно вообще не пользоваться стеком Си, что имеет как преимущества, так и недостатки.

Джеймс: Верно. Именно так сейчас и действует JVM. Она не использует стек Си. Я хочу сказать, что у нее свой собственный механизм стека. Если вы выполняете код Си, у вас отдельный стек. Определенную трудность для JNI представляет переход между этими двумя состояниями.

Учитывая, что источником вдохновения для C# была Java, какие ее особенности могли бы перенять другие языки программирования?

Джеймс: По-моему, C# перенял все, хотя странным было решение отказаться от функций безопасности и надежности, введя небезопасные указатели, что представляется мне невероятно глупым, но большинство особенностей Java используется в том или ином месте.

Вы написали сборщик мусора, чтобы не тратить время на исправление ошибок управления памятью. Как вы сравнили бы указатели в том виде, как они реализованы в C++, со ссылками в Java?

Джеймс: Указатели в C++ ужасны. Они просто сами напрашиваются на ошибки. Дело не столько в непосредственной реализации указателей, сколько в том, что сборкой мусора приходится заниматься вручную, а главное, допускается приведение типов между указателями и целыми, а некоторые API устроены так, что оно просто необходимо!

Ссылки в Java задумывались вами как способ решения этих проблем при одновременном сохранении возможностей указателей C++?

Джеймс: Да, в Java вы можете выполнять все нужные вам вещи, которые вы делали бы в C++ с помощью указателей.

Есть ли другие повторяющиеся проблемы, которых можно избежать с помощью общего решения, предлагаемого языком?

Джеймс: Таких вещей в Java множество. Примером может служить механизм исключений. В C++ принята практика игнорирования кодов

ошибок, возвращаемых в различных ситуациях. Java позволяет легко обрабатывать возникающие ошибки.

Программы на Java обычно гораздо надежнее – отчасти благодаря сильному поощрению максимально внимательного отношения к ошибкам, отчасти потому, что возникающие ошибки в значительной мере изолируются.

Способствует ли тщательный выбор значений по умолчанию тому, чтобы программисты лучше писали код без помощи внешних библиотек или дополнительных модулей?

Джеймс: С течением времени в большинстве языков стало больше порядка в таких вопросах. Одной из самых больших проблем C++ остается многопоточность. Многопоточность очень тесно вплетена в код Java, в результате чего Java очень хорошо справляется с многоядерными машинами.

Какова связь между конструкцией языка и конструкцией программ на нем?

Джеймс: Скрытая связь обнаруживается повсюду. Объектно-ориентированный язык способствует тому, что создаваемые приложения имеют высокий уровень модульности. Наличие сильной системы обработки исключений способствует созданию устойчивых систем. Практически любая мыслимая особенность языка в какой-то мере отражается на структуре приложений.

Есть ли что-то такое, что вам хотелось бы включить в стандарт языка? Скажем, автоматический контроль кода?

Джеймс: Ну, у нас много инструментов такого рода. Если посмотреть, что делает makeme(?), то окажется, что это в принципе LINT в реальном времени – всевозможные средства высокого уровня. В нынешние времена уже нельзя рассматривать язык изолированно: нужно брать язык вместе со средой.

Думаете ли вы при разработке языка о том, как будет происходить его отладка?

Джеймс: Есть множество вещей, имеющих значение для создания надежных программ, но они не относятся конкретно к отладке. Есть ряд стандартов для отладки, например, описывающих связь системы с системой отладки.

На самом деле, в языке многое делается для того, чтобы избежать самой ситуации, когда может потребоваться отладка. Для этого в Java есть такие вещи, как администратор памяти, сильная типизация, модель потоков, которые помогают избегать отладки.

Как влияет на отладку независимость от платформы?

Джеймс: С точки зрения разработчика отладка не имеет границ, и на Маке можно без хлопот отлаживать программы для сервера Linux.

Как вы отлаживаете свой код Java?

Джеймс: С помощью NetBeans.

Что вы можете посоветовать программистам на Java?

Джеймс: Применяйте NetBeans, не скупитесь на инструкции assert, будьте очень внимательны при сборке задач – у вас есть JUnit, который весьма популярен, – соедините их вместе, и они будут прекрасно работать.

Какие недостатки вы видите в преподавании компьютерных наук?

Джеймс: Большинство университетов сосредоточиваются на формальной стороне вопроса. На практике программисту часто ставят задачу: «Вот тебе программа, найди в ней ошибку», а в колледже обычно дают задание написать программу, которая будет делать то-то и то-то, при этом работа начинается с чистого листа, и можно делать все, что угодно.

Кроме того, программирование в значительной мере связано с социодинамикой работы в команде, а этому практически не учат.

Что думаете о документации для ПО?

Джеймс: Чем ее больше, тем лучше.

Одним из уникальных новшеств Java было встраивание документации в код.

Для Java API есть инструмент Javadoc, который извлекает документацию из исходного кода. Одна из существенных проблем документирования программ состоит в том, что документация часто оказывается устаревшей в сравнении с реальными API. Автоматическое извлечение стандартной документации обеспечивает гораздо лучшую синхронизацию, и даже если вообще не писать комментарии, то Javadoc фактически правдоподобно сгенерирует полезную документацию по API.

Поэтому для документирования кода необходимо в первую очередь воспользоваться Javadoc, а затем уже, чем больше вы разместите в коде хороших комментариев, тем лучше для всех.

Нужно ли до начала работы над проектом создавать его формальную или полную спецификацию?

Джеймс: У меня смешанное отношение к формальным спецификациям. Они относятся к числу тех вещей, которые замечательно выглядят в теории, но не столь хорошо проявляют себя на практике. Часто они полезны для небольших проектов, но чем проект крупнее, тем меньше

пользы от формальных спецификаций – хотя бы потому, что большие спецификации писать трудно.

Еще важнее то, что создание формальных спецификаций не всегда решает задачу. Задача смещается: ошибки в программном обеспечении приводят к поиску ошибок в спецификации. А найти ошибки в спецификации бывает очень трудно.

Даже когда вы не пишете формальных спецификаций, вы проводите некоторый анализ требований – во многих организациях работают по каскадной схеме, когда некая группа составляет документ с требованиями, который передают другим людям, которые и делают всю работу.

Документ, содержащий требования, часто оказывается очень проблемным, но если не наладить строгую систему обратной связи, то ошибки в требованиях оказываются не видны; ошибки в спецификациях не видны. Так что, будучи большим поклонником спецификаций, требований и тому подобного, я все же стараюсь не относиться к ним слишком серьезно, и уж точно не жду от них решения больших задач.

Я брал интервью у создателей UML и других языков, и меня весьма заинтересовала мысль, что эти весьма высокоуровневые языки проектирования позволяют строить логику ПО. Подразумевается, что можно обнаружить логические ошибки модели прежде, чем начнешь писать код.

Джеймс: Да, в Java-программировании есть множество инструментов моделирования на верхних уровнях. Средства моделирования высокого уровня, имеющиеся во многих средах создания веб-приложений и пользовательского интерфейса, в средах UML, бывают очень мощными.

Если взять NetBeans, то там присутствует довольно изощренная система моделирования UML. С ее помощью можно создать первоначальную спецификацию программного продукта в виде модели UML и потребовать сгенерировать программу автоматически, либо можно воспользоваться средством моделирования UML как своего рода археологическим инструментом, позволяющим заглянуть внутрь программы, так что эти системы полезны, хотя не решают всех проблем.

Обратная связь

Насколько интенсивна у вас обратная связь по самому языку, а не его реализации?

Джеймс: Мы получаем массу отзывов о языке.

Как вы поступаете с ними?

Джеймс: Если некоторую функцию предлагают один-двое, обычно мы игнорируем такую просьбу.

В таком деле, как язык, нужно очень осторожно подходить к модификациям. В отношении API отсев менее строг, но обычно мы ничего не меняем, пока не возникает действительно сильная потребность в чем-либо.

Поэтому когда многие просят об одном и том же, мы решаем, что об этом стоит подумать. Но когда некую вещь просит один из миллиона разработчиков, мы склонны считать, что от нее будет больше вреда, чем пользы.

Чем обернулось для вас раскрытие исходного кода Java?

Джеймс: О, массой полезного. Код Java открыт с 1995 года, и те, кто его скачал, использовали его для самых разных вещей – от дипломных проектов до аудитов безопасности, и это оказало мощное воздействие.

В смысле совершенствования реализации или коллективного развития языка?

Джеймс: Полезно иметь много авторов. Возникает широкое обсуждение.

Можно ли разрабатывать язык на демократической основе?

Джеймс: Это очень тонкий вопрос, потому что чрезмерная демократия в итоге ни к чему, кроме хлама, не приводит. Но злоупотребление центральной властью приводит к продукту, который никому не нужен, потому что он отражает точку зрения единственного человека.

Поэтому очень важно вовлечь в обсуждение многих, соблюдая при этом достаточно жесткую процедуру принятия решений.

Как вы считаете: нужно развивать язык, или если у вас была задача, и вы создали инструмент, который может ее решить, то нужно браться за новый язык?

Джеймс: Думаю, верно и то и другое. Почему бы и не развить язык дальше, но мне кажется, что должен быть установлен достаточно высокий порог на пути вносимых изменений. Если нечто продемонстрировало свою реальную ценность, можно и добавить это в язык. Если ценность не слишком высока, то вносить произвольные синтаксические изменения бессмысленно. Если в одно прекрасное утро вы решите, что фигурные скобки не нужны, то это глупо, но когда нечто реально влияет на эффективность написания программ, то можно пойти на большие усилия, как было в случае добавления нами генериков несколько лет назад, которое глубоко оправдано.

Как вы решаете, что нужно ввести в сам язык, а что оставить для внешних библиотек?

Джеймс: Почти всегда это решается исходя из того, насколько универсальной является та или иная функция. То, что может быть интересно лишь небольшим группам пользователей, лучше поместить в библиотеку. Но в целом, если что-то можно делать в библиотеке, там это и нужно делать, а изменения в языке нужно оставить для тех случаев, когда они не укладываются в рамки никаких библиотек.

Какие критерии вы применяете при разработке API?

Джеймс: Мой главный принцип – минимизировать API. Другой принцип – разработка на основе практики его применения. Одна из причин появления неудачных API – разработка их в своего рода вакууме, когда разработчики рассуждают, что вдруг кто-то захочет сделать вот это или вон то, в результате чего API разрастаются до невероятных размеров и становятся гораздо сложнее, чем это необходимо.

Тогда как на самом деле нужно подумать: для чего люди захотят его применять? Взгляните на другие системы – для создания меню или открытия сетевых соединений. Что оказалось в них действенным? Чем люди пользовались в действительности и чего не было в API?

Можно узнать много интересного о разработке API и языков, проведя статистический анализ ПО, написанного на других языках.

Вынесли ли вы какой-то полезный опыт из разработки Java, который можно передать другим?

Джеймс: Разрабатывать язык не так уж сложно. Главное не в том, чтобы разработать язык, а в том, чтобы определить, для чего он нужен. Каков контекст? Какие задачи он поможет решать?

Решающее влияние на Java оказало то, что в основу была положена сеть. Какие последствия имеет сетевое взаимодействие для конструкции языка программирования? Они оказались очень глубокими, и в известном смысле конструктивные решения оказались достаточно просты, как только были оценены последствия работы в сети.

Повлияла ли Java на мнение общества о независимости от платформы?

Джеймс: Не думаю, что широкую публику интересуют вопросы независимости от платформы. Я хочу сказать, что с моей точки зрения – может быть, странной для конструктора крупномасштабных систем, – это один из тех вопросов, о существовании которых публике вообще необязательно знать.

Когда вы используете свою карточку Visa в банкомате или кассовом аппарате, можно не сомневаться, что в работу включается уйма кода Java,

исполняющегося на машинах Sun и IBM, Dell и HP, – тут вам будут архитектуры и *x*86, и PowerPC, и все, что только можно, – но вы и не заметите этого, проводя карточкой по считывателю.

Тем не менее невидимо для вас работает множество таких механизмов. Вы спускаетесь в метро; если в лондонском метро вы пользуетесь одной из бесконтактных карточек вроде Oyster, то это тоже система, написанная на Java.

Вы действительно используете все заложенные в систему возможности независимости от платформы, но если бы ее пользователи были обязаны знать, на каком языке она написана, такая система никуда не годилась бы. Одна из наших задач – обеспечить полную прозрачность и не мешать людям. Конечно, маркетологов это бесит. Они были бы рады, если бы каждый раз, спускаясь в метро, вы видели перед собой логотип Java, но это было бы нелепо.

13

C#

Когда Microsoft начала тяжбу с Sun Microsystems по поводу изменений, внесенных последней в язык программирования Java, она обратилась к ветерану разработки языков Андерсу Хейлсбергу с предложением создать новый объектно-ориентированный язык, который поддерживался бы мощной виртуальной машиной. В результате появился C#, заменивший в экосистеме Microsoft и Visual C++, и Visual Basic. Несмотря на неизбежные сравнения с Java и в синтаксисе, и в реализации, и в семантике, сам язык далеко опередил своих предшественников, вобрав в себя характеристики, присущие таким функциональным языкам, как Haskell и ML.

Язык и конструкция

Вы создали и сопровождали несколько языков. Вы начинали с реализации Turbo Pascal; переход от реализации к разработке — это естественное развитие?

Андерс Хейлсберг: Думаю, это вполне естественный процесс. Первый написанный мной компилятор был предназначен для подмножества Паскаля, а последовавший затем Turbo Pascal стал первой почти полной реализацией Паскаля. Но Паскаль всегда рассматривался как язык для обучения программированию, и в нем не было многих стандартных возможностей, считающихся обязательными при создании приложений, работающих в реальной жизни. Чтобы обеспечить ему коммерческий успех, пришлось немедленно заняться расширением его по нескольким направлениям.

Удивительно, что учебный язык смог столь удачно перекрыть пропасть между обучением и коммерческим успехом.

Андерс: Есть много языков для обучения программированию. Если взглянуть на путь, который прошел Никлаус Вирт, – он придумал Паскаль, а после того Модулу и Оберон, – то видно, что он всегда стремился к простоте. Обучение некоторому языку может быть основано на его пользе для иллюстрации некоторой идеи в отсутствие другого практического применения; но это может быть полноценный язык, который действительно учит основам программирования. Именно таким был задуман Паскаль.

По-видимому, есть два принципа обучения. В одних случаях — например, в MIT — начинают с Scheme. Другая школа нацелена на «практику». Сначала там учили C++. Теперь это Java, а в некоторых случаях C#. Что бы предпочли вы?

Андерс: Я, конечно, всегда тяготел к лагерю «практиков». Если хотите, я в большей мере инженер, чем ученый. По моим представлениям, если уж учить чему-то людей, то тому, чем они потом смогут воспользоваться на практике.

Естественно, не нужно впадать в крайности. Ответ где-то посредине. Мы всегда, применяя языки программирования на практике или создавая их реализации для промышленного применения, пользуемся результатами академических исследований. В данный момент можно наблюдать, как широко пожинаются плоды идей функционального программирования, которое бог весть как долго разрабатывалось теоретически. Думаю, что успех ждет нас на пересечении этих двух подходов.

Ваша философия разработки языка состоит в том, чтобы заимствовать все встретившиеся вам идеи и реализовать их на практике?

Андерс: Да, в какой-то мере. Думаю, нужно начинать с каких-то руководящих принципов. Один из таких принципов всегда простота. Кроме того, я отдаю предпочтение дальнейшему развитию, а не созданию с нуля.

Вам может понравиться какая-то идея, и чтобы реализовать ее, вы решите создать совершенно новый язык, в котором она замечательно воплотится. Но 90% того, что необходимо в каждом языке, это просто рутина. Можно вложить массу труда – а можно взять и расширить существующий язык (например, недавно мы сильно подвинули C# в сторону функционального программирования) и получить все это почти даром. И у вас есть огромная база пользователей, которым достаточно освоить эту новинку. Не без сложностей, но все же это гораздо проще, чем изучать целый новый язык и новую среду исполнения, чтобы научиться определенному стилю программирования.

Трудно провести границу между собственно языком и его экосистемой.

Андерс: Да, особенно в нынешние времена. Прежде, лет 20 или 30 назад, на изучение языка уходила большая часть времени. Изучение среды программирования сводилось к изучению языка. У языка была еще небольшая библиотека времени исполнения. Что-то еще могло быть связано с ОС, если только у вас был к ней доступ. А теперь взгляните на нынешние гигантские фреймворки вроде .NET или Java – в этих средах программирования один лишь объем API настолько впечатляет, что язык в нем как-то теряется. Пусть это некоторое преувеличение, но среда явно преобладает над языком и его синтаксисом.

Это налагает большую ответственность на разработчика библиотек?

Андерс: Работа разработчика платформы становится важнее, потому что огромное значение приобретают долговечность платформы и возможность реализовать на ее основе несколько разных языков, чему мы всегда придавали большое значение. .NET с самого начала разрабатывалась как платформа для нескольких языков, и чего только не увидишь на ней сегодня: статические языки, динамические языки, декларативные языки типа XAML и так далее. Тем не менее в основе все та же среда, те же API, и ее влияние огромно. Если реализовывать все это по отдельности, вы погибнете, пытаясь организовать взаимодействие и выделить необходимые ресурсы.

В общем, вы поклонник виртуальных машин-полиглотов?

Андерс: Думаю, это неизбежный путь. Вспомним добрые старые времена – 8 бит и 64K памяти. Вопрос был в том, чем заполнить эти 64K, что удалось достаточно быстро. Тогда не задавались целью строить системы на долгие годы.

Ваша реализация годилась на один-два месяца, и этого было достаточно. 640K – и вы управлялись с ними за какие-нибудь полгода. А сейчас это бездонная пропасть. Пользователи требуют все больше и больше, и нет никаких сил переписывать все заново. Выход – повторное использование и организация взаимодействия того, что уже создано. В противном случае вы будете вечно топтаться на одном месте, реализуя элементарные основы.

Если можно заложить общую основу для всего этого, добившись гораздо более высокой степени взаимодействия и эффективности общих системных ресурсов, то это тот путь, который нужно выбирать. Возьмите, к примеру, взаимодействие между управляемым и неуправляемым кодом. Здесь возникает масса проблем. Но лучше мы решим их, чем этим будет заниматься каждая отдельная среда программирования. Сложнее всего строить гибридные приложения, в которых одна часть управляемая, а другая – нет, и по одну сторону ограды у вас есть сборка мусора, а по другую она отсутствует.

По-видимому, одно из проектных требований к JVM – не нарушить обратную совместимость с более ранними версиями байт-кода. Оно исключает некоторые возможные конструктивные решения. На уровне языка можно принять проектное решение, но в фактической реализации, скажем, генериков потребуется затирание типов.

Андерс: Знаете, а мне кажется, что их проектным требованием было не только обеспечение обратной совместимости. Можно добавить несколько новых байт-кодов и все же сохранить обратную совместимость. Их задача была – ничего не добавлять ни в байт-код, ни в BM. Это большая разница. Фактически задачей проекта было ничего не развивать. Это полностью связывает вас. В .NET мы хотели достичь обратной совместимости, поэтому добавили новые возможности, новые метаданные. Несколько новых команд, библиотек и так далее, но весь .NET 1.0 API продолжал работать на .NET 2.0.

Я всегда недоумевал, почему они выбрали такой путь. Понятно, что это позволило сразу достичь желаемого имевшимися средствами, но ведь в нашей области деятельности все строится на развитии. Остановившись, вы подписываете себе смертный приговор. Ждать останется недолго.

Наше решение материализовать генерики, а не затирать типы, я считаю в высшей степени правильным и с лихвой окупившим себя. Я убежден, что вся работа, проделанная нами с LINQ, была бы просто невозможна без воплощенных генериков. Вся динамика, реализованная в ASP.NET, всякая динамическая генерация кода, осуществляющаяся нами практически в каждом выпущенном продукте, в огромной степени обязана тому, что генерики реально присутствуют на этапе исполнения и этап

компиляции гармонически связан со средой исполнения. Это чрезвычайно важно.

Delphi критиковали за сильное нежелание ломать код, которое повлияло на ряд решений относительно языка.

Андерс: Тогда давайте вернемся. Ваше выражение «ломать код», видимо, имеет в виду развитие. Скажем, $(N + 1)$-ю версию какого-то продукта. Можно доказывать, что иногда ломать код полезно, но в целом я не нахожу этому оправдания. Единственные слабые аргументы в пользу вмешательства в код, которые я слышу, это «так будет аккуратнее», или «это лучше с точки зрения архитектуры», или «это окажется полезным в будущем» и тому подобное. На это я отвечаю, что платформа живет лет 10 или 15, после чего так или иначе рушится под собственным весом.

Все более или менее устареет, может быть, лет через 20. К тому времени появится много нового, и такого, что не потребует дополнительных расходов. Если вы собрались ломать, ломайте по-настоящему. Ломайте все. Не жалейте ничего. Терпеть не могу косметику. Совершенно бессмысленно.

Это похоже на чехарду, когда между прыжками проходит лет 5 или 10.

Андерс: Вы либо играете в чехарду, либо углубляетесь в обратную совместимость, и в обоих случах за вами следует все сообщество ваших пользователей.

Управляемый код до некоторой степени это реализует. Можно использовать в процессе свои имеющиеся компоненты.

Андерс: Конечно, с самого начала .NET мы в каждой новой версии сохраняли обратную совместимость. Мы иногда исправляем ошибки, из-за которых какой-то код перестает работать, но нужно определить, в каких случаях допустимо, чтобы чей-то код перестал работать.

Ради безопасности, ради корректной работы какой-то программы или по другим причинам мы иногда меняем код, но это случается редко и обычно связано с конструктивной ошибкой в программе пользователя или с решением проблемы, о существовании которой он и не подозревал. В таких случаях это оправдано, но произвольное вторжение в код ради наведения красоты или по аналогичным причинам, по моему мнению, ошибочно. В молодости я достаточно часто занимался такими вещами и понял, что ни к чему хорошему это не приводит.

Трудно выдвигать в качестве аргумента одно лишь чувство вкуса.

Андерс: Что ж, ничего не поделаешь. У каждого свой взгляд на хороший вкус.

Если взглянуть на языки, в создании которых вы участвовали, от Turbo Pascal до Delphi, J++, Cool и C#, есть ли там какая-то общая тема? Послушав раннего Моцарта и его Реквием, можно уверенно сказать, что и там и там отчетливо прослушивается Моцарт.

Андерс: На все накладывает свой отпечаток время. Я рос во времена объектно-ориентированного программирования и всего такого. Нет сомнений, что начиная с середины Turbo Pascal и до настоящего времени все, чем я занимался, было по сути объектно-ориентированным языком. Многое изменилось. В Delphi мы проделали значительную работу по созданию модели, более ориентированной на компоненты, включая свойства, события и так далее.

Это перешло в мою работу над C#, и, конечно, там различимы следы моих прежних трудов. Стараюсь всегда держать руку на пульсе сообщества и представлять там соответствующие новости. У Turbo Pascal была оригинальная среда разработчика, а в Delphi появилось визуальное программирование, RAD. Для C# и .NET главным было наличие управляемой среды выполнения, безопасность типов и так далее. Учишься на всем, что видишь вокруг, – твоя ли это экосистема или твоего конкурента. Всегда стараешься найти, что у них получилось хорошо, а что вышло неудачно. В этом деле все мы стоим на плечах гигантов. Правда, поражает, насколько медленно развиваются языки программирования в сравнении с темпами развития аппаратуры. Просто удивительно.

После выхода Smalltalk-80 сменилось 15–20 поколений аппаратного обеспечения!

Андерс: Эта смена происходит практически каждые 18 месяцев, при этом нет большой разницы между теми языками программирования, которыми мы пользуемся сегодня, и теми, которые были придуманы, скажем, 30 лет назад.

До сих пор ведутся споры по поводу старых идей, например функций высшего порядка в Java. Эта тема обсуждается, наверное, уже лет 10.

Андерс: Что вызывает сожаление, поскольку, мне кажется, можно было решить этот вопрос быстрее. Едва ли есть сомнения в их ценности. Вопрос, скорее, в объеме процедур и затрат на их реализацию у программистов Java.

Если введение в язык «вызова с остаточными вычислениями» дает вам огромные преимущества, решитесь ли вы на него, когда разобраться в этом смогут не более 10% программистов?

Андерс: Это большой вопрос. Не думаю, что именно так обстоит дело, но посмотрите, что мы сделали с LINQ. Я искренне верю, что мы сделаем благо большинству программирующих на C#. Возможность писать запросы в более декларативном стиле и наличие единообразного

языка запросов для разных областей данных – большая ценность. В некоторых отношениях это Святой Грааль интеграции языков с базами данных. Возможно, мы еще не до конца решили эту задачу, но все же достигли значительного прогресса, оправдывающего дополнительную учебу, и можно изложить это таким образом, чтобы людям не пришлось изучать основы лямбда-исчисления.

Мне кажется, это прекрасный пример практического применения функционального программирования. Можно успешно применять его, даже не догадываясь, что вы занимаетесь функциональным программированием или что на его принципах основана ваша работа. Я очень доволен результатами, которые мы здесь получили.

Вы упомянули практическое применение. Как вы решаете, какие функции добавить, а какие выкинуть? Какими критериями вы пользуетесь, принимая такие решения?

Андерс: Даже не знаю. С годами обретаешь способность определять, что окажется достаточно полезным для ваших пользователей, чтобы они были готовы расширить свой теоретический багаж. Поверьте, наши пользователи выдвигают немало интересных предложений – «если бы только можно было...» или «как бы мне хотелось, чтобы...», – хотя довольно часто они узко направлены на какую-то конкретную проблему и имеют малую ценность в качестве общего теоретического понятия.

Нет сомнений, что лучшие языки создаются небольшими группами людей или отдельными лицами.

Есть ли различие между разработкой языка и библиотеки?

Андерс: Есть, и существенное. Очевидно, что API гораздо сильнее ориентированы на определенные области приложений, чем языки, а языки, если хотите, представляют определенный уровень абстракции над API. Языки закладывают структуру, так сказать, кварки, атомы и молекулы разработки API. Они определяют, как вы будете составлять эти API, но не что эти API будут делать.

В этом смысле, мне кажется, есть большая разница. Это возвращает меня к той теме, которую я хотел обсудить выше. Когда мы рассматриваем целесообразность добавления в язык новой возможности, я всегда оцениваю, будет ли она применима не в одной, а в нескольких областях. Хорошая функция языка отличается тем, что ее можно применять несколькими способами.

Снова приведу в качестве примера LINQ. Если посмотреть, какую работу мы проделали с LINQ, то можно выделить в ней шесть или семь разных возможностей языка, таких как методы расширения, лямбда, выведение типа и так далее. Можно соединить их вместе, создав своего рода новый API. В частности, можно реализовать в виде API механиз-

мы запросов, но сами эти возможности языка оказываются полезными во многих других случаях. Методы расширения используют для решения самых разных интересных задач. Например, очень удобной оказывается возможность вывода типа локальных переменных.

Вероятно, что-то подобное LINQ мы могли бы выпустить гораздо раньше, если бы сказали себе: «Вставим сюда SQL или что-нибудь весьма специфическое для SQL Server, сможем общаться с базой данных, и этого хватит», – но такое решение не было бы достаточно общим, чтобы оправдать его появление в универсальном языке программирования. Очень быстро язык превратился бы в инструмент для конкретной предметной области, от которой ему уже нельзя было бы освободиться.

Ваш прекрасный 3GL превращается в 4GL, и это конец универсальности.

Андерс: Да. Я это очень хорошо понимаю. Важная проблема, которой мы заняты, это параллелизм. Им заняты все, потому что жизнь заставляет. Это вопрос не желания, а необходимости. Опять-таки, в области параллельных вычислений можно было позволить языку навязать определенную модель параллелизма, но это было бы ошибкой. Мы должны стать выше этого и выяснить, какие возможности должны присутствовать в языке, чтобы позволить людям реализовывать свои замечательные библиотеки и модели программирования для параллельной обработки. Нам нужно, чтобы язык позволял лучше изолировать состояния. Нам нужна чистота функций. Нам нужна неизменяемость (immutability) как базовый принцип. Если можно ввести это в качестве базовых принципов, то пусть разработчики ОС и программных сред занимаются экспериментами с разными моделями параллельных вычислений, потому что, видимо, им всем это нужно. Тогда не нужно гадать, кто же выйдет победителем. Мы спокойно проследуем дальше, когда один выйдет из игры, а другой окажется более удачливым.

В любом случае мы окажемся в деле.

Похоже, вы хотите дать людям инструменты, с помощью которых можно создавать замечательные вещи, а не диктовать, какие вещи они должны создавать.

Андерс: Мне бы очень этого хотелось. Это гораздо сильнее содействует появлению новшеств в сообществе.

Где вы видите это в сообществе C#? Вам что – приносят код? Вы встречаетесь со своими пользователями? У вас есть MVP, активно участвующие в телеконференциях и группах пользователей?

Андерс: Все перечисленное плюс еще кое-что. Есть такой способ делиться своим кодом, как Codeplex. Есть разнообразные сообщества.

Есть коммерческие сообщества. Есть сообщества open source. Есть масса кода .NET в свободном доступе. Код идет отовсюду. Нельзя сказать, что источник один. Существует многообразная и сложная экосистема.

Куда ни пойдешь, всюду наткнешься на что-то интересное: «как они до этого додумались?» или «это изумительно!». Можно прикинуть, сколько труда это кому-то стоило. Даже если нет коммерческой ценности, все же прекрасная работа.

Я стараюсь обязательно следить за многочисленными блогами, посвященными C# и LINQ.

Это мои любимые ключевые слова, когда я шатаюсь по блогам, чтобы узнать, что делается в мире. Получаешь хорошее представление о том, правильно ли люди воспринимают сделанное тобой. Можно делать некоторые выводы на будущее.

Развитие языка

Как вы узнаете простоту?

Андерс: Есть настоящая простота, а есть ложная – то, что я называю «simplexity» и с чем часто встречаюсь. Это когда сначала строят нечто сверхсложное, а потом говорят: «Нет, в этом никто не сможет разобраться; это слишком сложно, но мы должны сохранить всю мощь нашей конструкции; попробуем построить поверх нее простую систему; обернем это все в какой-нибудь простой интерфейс».

В тот момент, когда вам требуется сделать нечто не совсем согласующееся с назначением этой системы, происходит обвал. Вы тонете в глубоком болоте сложности, скрытой под видным вам тонким декоративным слоем, потому что подлинной простоты, пронизывающей всю систему, не было. Не знаю, понятно ли вам, о чем я, но я представляю себе это именно так. Простота часто означает, что вы делаете больше меньшими усилиями. Все очень скромно, но работает не хуже или даже лучше остального. Смысл в том, чтобы делать больше меньшими силами. А не в том, чтобы делать больше большой системой с упрощающим уровнем поверх нее.

Стали бы вы придерживаться такого принципа, если бы вам сегодня пришлось создавать новый язык?

Андерс: Конечно. К данному моменту я создал множество языков программирования и уж точно множество их реализаций. Я считаю чрезвычайно важным перед тем, как браться за создание нового языка, очень четко определить, зачем вы его создаете и какую задачу хотите решить.

Часто люди совершают ошибку, создавая новый язык, потому что они одержимы некоторой конкретной задачей, которую хотят решить. Возможно, язык программирования поможет решить эту задачу, поэтому они начинают с него и часто делают отличную работу. Но каждый язык программирования – подчеркиваю, каждый – на 10% состоит из нового и на 90% из обычных для программирования вещей, которые просто обязаны в нем быть. И все эти новые изобретательные решения, которые вы встречаете в новых языках программирования, замечательны в своих 10% нововведений и ужасны в тех 90%, которые должны быть в каждом языке, чтобы на нем действительно можно было писать программы, и потому оказываются неудачными.

Крайне важно осознать, что есть целая куча очень скучных стандартных вещей, которые должны быть в каждом языке программирования. Если вы не понимаете этого, вас ждет неудача. И наоборот: если вместо того, чтобы создавать новый язык, вы сможете развить уже существующий, уравнение принимает совершенно иной вид, потому что 90% у вас уже есть. Фактически есть все 100%. Вы просто хотите добавить что-то новое.

Как в C++.

Андерс: Как в C++, являющем собой замечательный пример эволюции Си, или в разных версиях C#, которые мы создали, и так далее. Я большой поклонник эволюции. Конечно, всегда наступает момент, когда больше уже нельзя втиснуть ничего нового, – конфликты между тем, что вы хотите добавить, и старыми способами ведения дел слишком значительны, чтобы двигаться дальше. Создание нового языка – скорее исключение, чем правило.

Какой язык вы стали бы создавать: универсальный или для конкретной области?

Андерс: Думаю, ни тот и ни другой. Я постарался бы создать универсальный язык, пригодный для создания проблемно-ориентированных языков. Опасность, подстерегающая нас при создании языков для конкретных предметных областей, заключается в том, что со своей областью они могут справляться хорошо, а с универсальностью у них оказывается плохо. Есть несколько универсальных функций, необходимых буквально каждому языку для конкретной области. Если только проблемно-ориентированный язык не оказывается чистым языком описания данных, но тогда, мне кажется, с таким же успехом можно использовать XML.

Если вы действительно пишете язык программирования – с логикой, предикатами, правилами и прочим, – то у вас должны быть выражения, а в выражениях есть операторы, а еще вам могут понадобиться

стандартные функции, а ваши клиенты захотят делать то, что вам и в голову не приходило. Вам понадобится масса стандартных вещей. Если вместо этого можно создать проблемно-ориентированный язык на базе некоторого универсального языка, то, думаю, это будет гораздо удачнее для вас, чем начинать все с самого начала.

Одна из проблем современных универсальных языков программирования в том, что они становятся более пригодны для создания внутренних проблемно-ориентированных DSL, и в качестве примера можно рассматривать LINQ. А плохо в них в данное время то, что с паттернами применения этих внутренних DSL они справляются слабо. В каком-то смысле, создавая внутреннюю DSL, вы фактически хотите ограничить возможности универсального языка программирования. Вам нужно отключить универсальность языка и оставить ее только в некоторых местах своей DSL. Это один из недостатков современных универсальных языков программирования. Видимо, стоило бы этим заняться.

Брайан Керниган сказал, что если хочешь создать универсальный язык, то нужно ставить перед собой эту цель в самом начале работы. Иначе, если вы сделаете небольшой язык, то как только у него появятся пользователи, они станут просить, чтобы вы добавили для них новые функции. Расширение DSL обычно удается не очень хорошо.

Андерс: Это так. Кажется, Гослинг сказал, что все файлы конфигураций в конечном счете превращаются в отдельные языки программирования. Это верно замечено, и об этом всегда нужно помнить.

Вы сказали, что платформа в чем-то важнее языка. Будем выпускать компоненты для многократного использования?

Андерс: Я сказал это потому, что если посмотреть, как в последние 25–30 лет развивались языки, инструменты и программные среды, то бросается в глаза, как мало изменились языки программирования. В той же мере удивительно, как выросли за это время среды программирования и исполнения. Они выросли, наверное, на три порядка по сравнению с тем, какими были 25–30 лет назад. Когда я начинал работать с Turbo Pascal, в библиотеке времени исполнения было 100 или 150 стандартных функций, и все. Сейчас у нас есть среда .NET, в которой 10 000 типов и 100 000 членов. Очевидно, что важность повторного применения этой работы неуклонно растет. Важность обусловлена тем, что это определяет наше представление о задачах, но важность среды растет, потому что это то, что мы многократно применяем в своих программах.

Повторное использование – это сегодня все. С точки зрения программирования, ваш компьютер – бездонная яма. Вы можете всю жизнь писать для него код, и он никогда не наполнится. Возможности его огром-

ны, а конечные пользователи ждут еще большего. Единственный путь к успеху сегодня – это поиск умных способов повторного использования уже сделанной работы. 25–30 лет назад такой проблемы не было. Когда у вас 64K памяти, ее можно заполнить за пару месяцев.

В какой мере продуктивность программиста определяется языком и способностями?

Андерс: Думаю, то и другое тесно связаны. Язык влияет на то, как мы думаем. Работа программиста состоит, если хотите, в том, чтобы думать. Это сырье, чистая энергия, участвующие в процессе. Язык – это то, что определяет ваше мышление; его задача – помочь вам думать продуктивным образом. Например, языки с поддержкой объектов заставляют вас думать над задачей так. Функциональные языки заставляют вас думать над задачей иначе. Динамические языки заставят вас думать еще как-то. Выбрав язык, вы выбираете особый способ мышления. Иногда бывает полезно принять сразу два способа мышления и рассмотреть задачу с разных точек зрения.

Какую функцию вы предпочли бы добавить к языку – ту, которая немного повысит продуктивность всех пользователей, или ту, которая значительно повысит продуктивность нескольких разработчиков?

Андерс: Если это универсальный язык, то добавлять функции, полезные немногим, – плохая идея, потому что в итоге у вас получится мешок, набитый всякой всячиной. Отличительной чертой всякой хорошей функции языка бывает то, что у нее не одно, а сразу много полезных применений. Если взглянуть на все, что мы добавили в C# 3.0, – то, что в совокупности составляет язык интегрированных запросов (language-integrated query, LINQ), – на самом деле состоит из шести или семи отдельных возможностей языка, которые сами по себе имеют полезные применения. И пользу из них извлечет не какой-то отдельный программист. Им соответствует более высокий уровень абстракции. Для всякой хорошей функции языка нужно продемонстрировать возможность полезного использования в нескольких сценариях, иначе не стоит вводить ее в язык. Возможно, лучше поместить ее в какой-нибудь API.

Думаете ли вы, какие функции можно добавить или удалить, чтобы облегчить отладку? Учитываете ли вы опыт отладки в процессе разработки языка?

Андерс: Обязательно. Если взглянуть на основания C# в целом, то это язык с безопасными типами, то есть в нем нет таких вещей, как выход за границы массива или висячий указатель. Все ведет себя в соответствии с корректными определениями. В C# нет такой вещи, как неопределенное поведение. Обработка ошибок осуществляется с помощью исключений, а не кодов ошибок, которые можно проигнорировать,

и так далее. Поэтому все такие основные вещи, как безопасность типов, безопасность памяти и обработка исключений, крайне полезны для устранения или облегчения поиска целых классов ошибок. Мы постоянно думаем о таких вещах.

Как вам удается избегать таких повторяющихся проблем, не ограничивая разработчиков? Как вы примиряете безопасность и свободу действий разработчика?

Андерс: Думаю, у каждого языка есть свое место в диапазоне, так сказать, между мощью и продуктивностью. C# явно является более надежной и более защищенной средой в сравнении с C++, который, в свою очередь, безопаснее и защищеннее, чем код ассемблера. Общая тенденция универсальных языков программирования на протяжении всей их истории – повышение уровня абстракции и безопасности программной среды, или перекладывание на машину большей части хозработ программистов, чтобы дать им возможность сосредоточиться на творческой части процесса, где они действительно создают своим трудом ценность. Обычно программисты очень плохо управляют памятью. Анализ безопасности типов они также проводят плохо, откуда и ошибки.

Думаю, если нам удается переложить эти задачи с плеч программистов на машину, освободив их для творческого труда, то это правильное решение. Некоторые потери производительности это влечет, но они совсем невысоки. Если вы сделаете профилирование типичного приложения .NET и посмотрите, где программа тратит время, то сборку мусора можете вообще не увидеть. При этом ваша программа безопасна и не создает утечек памяти. Это замечательный компромисс. По сравнению с тем, с чем приходилось сталкиваться в системах с ручным управлением памятью, таких как C++ или Си, это просто фантастика.

Можно ли использовать научный подход к разработке и расширению языка? Я вижу, что научные исследования принесли пользу в реализациях, но разработка языка, по-видимому, остается предметом личных предпочтений разработчика.

Андерс: Думаю, разработка языка – любопытное сочетание науки и искусства. В нем велика роль науки – математический формализм в нотации для парсинга, семантика и системы типов, и что там еще – генерация кода? Сплошная наука. Сплошной инжиниринг.

Теперь об искусстве. На что похож язык? Какие процессы происходят в вашей голове, когда вы программируете на том или ином языке, и в чем их отличие? Что людям будет проще понять, а что – труднее? Думаю, мы никогда не сможем это измерить. Я никогда не стану на чисто научную точку зрения. Меня всегда будет интересовать разработка языка как чистое искусство. Есть хорошие картины, а есть плохие,

и можно рассуждать о них с «научной» точки зрения: вот тут плохо продумана композиция, краски не те подобраны. Но в конечном счете все определяет зритель. Есть нечто, не поддающееся формализации.

Как вы считаете, помогает ли в чем-то то, что вы говорите на нескольких языках? Иногда по-итальянски можно описать одним словом то, для чего в английском понадобится целое предложение (разумеется, бывает и наоборот).

Андерс: Не знаю. Хороший вопрос. Никогда не думал об этом. Возможно. Я совершенно уверен, что хороший разработчик языка должен знать несколько языков программирования – никаких сомнений. Помогает ли тут знание нескольких иностранных языков, не могу судить. Вполне возможно, что связь тут есть. У нас в команде разработчиков есть те, кто знает языки, и те, кто понимает музыку. Между ними есть какая-то связь, хотя я и не вполне понимаю, какая.

C#

Долгое ли будущее ждет C#? Вы занимаетесь им почти 10 лет.

Андерс: Проект C# начался в конце декабря 1998 года, так что мы достигли 10-летнего юбилея. Это не десять лет промышленного существования, а десять лет принятия внутри фирмы. Думаю, у нас есть в запасе еще как минимум 10 лет, но все зависит от случая. Я уже говорил, что давно отказался от долгосрочных предсказаний в нашей отрасли, потому что никакие прогнозы тут не оправдываются. Но для C# я предвижу крепкое здоровое будущее. Нас еще ждут нововведения и большой объем работ.

Глядя на эволюцию C# с точки зрения прикладной области, я вижу желание заменить C++ в качестве языка системного программирования.

Андерс: Можно использовать язык и для этого, но есть масса применений, для которых более подходит управляемая среда исполнения, такая как .NET или Java.

Сравнивая C# с Java, я наблюдаю за C# более сильное стремление к эволюции. Разработчики Java тяготеют к созданию некоего минимального набора, делающего весьма схожим код, написанный любыми авторами. Вы можете 10 лет работать на Java, а можете не иметь никакого опыта программирования или закончить 6-месячные курсы Java, – ваш код будет выглядеть одинаково. C# вбирает в себя новые идеи из Haskell и F#. Есть ли у вас желание добавить в язык новые возможности – такие, которых слушатели 6-месячных курсов C# не знают и не сразу поймут?

Андерс: Я пришел сюда не затем, чтобы конструировать новый Кобол, – сформулируем это так.

В чем источник интернет-революции и электронной революции, которые мы наблюдаем? В том, что наше развитие непрерывно. Я вижу причину в этом. Перестав развиваться, вряд ли вы создаете что-то новое. По большому счету, конечно. Разумеется, стабильность платформы имеет ценность, но вы можете обеспечить эту ценность, гарантировав обратную совместимость. Никто не запрещает сойти с поезда на C# 1.0 и дальше не двигаться. Если же вы действительно хотите увеличить эффективность своего труда и создавать приложения новых типов, например SOA или другие, и освоить более динамичные стили программирования – адаптируемые программы или более описательные стили программирования, как в LINQ, – тогда вы должны развиваться или уйти с дороги, и тогда вас заменит что-то еще.

Получаете ли вы отзывы на сам язык C#, а не только на его реализацию?

Андерс: Мы получаем их ежедневно, самыми разными способами. Мне пишут электронные письма. Я читаю блоги, читаю форумы, где задают технические вопросы, присутствую на конференциях – масса путей, которыми мы ежедневно узнаем, что хорошо, а что плохо в нашем языке. Эти отзывы мы передаем команде разработчиков и ведем длинный перечень всех идей, даже самых безумных. Некоторые из них никогда не попадут в язык, но мы храним их в списке, потому что в них есть нечто, что в один прекрасный день может стать источником хорошей идеи в той же области. Мы знаем, что пока у нас нет правильного решения, но есть желание что-то сделать.

Потом постепенно мы нашли решение проблем. Часть из них – простые вещи, о которых нас просят, и мы просто делаем их. Другие – крупные задачи, о которых нас никогда даже не просили, как LINQ. Никто никогда не просил нас встроить запросы в язык, потому что не представлял, что это возможно.

Не скажу, что возможен какой-то один особый способ получения отзывов. Это весьма органический процесс, и мы получаем данные из многих разных мест. Несомненно, мы не смогли бы разработать язык, не получая все эти отзывы, поэтому мы внимательно изучаем, как используют наш продукт.

Как вы управляете командой разработчиков? Как принимаете решения?

Андерс: Во-первых, получая отзывы от клиентов, мы часто слышим: «Мы будем очень благодарны вам, если вы сможете добавить вот такую функцию». Начинаем разбираться и выясняем, что им нужно сделать

то-то и то-то, и обычно они говорят, каким видят решение проблемы. Теперь ваша задача в том, чтобы выяснить, в чем на самом деле их проблема, и попробовать включить ее решение в общую структуру языка. В некотором смысле, получив отзыв, мы сначала занимаемся детективным расследованием того, что реально стоит за тем решением, которое предлагает клиент. В чем на самом деле заключается его проблема.

Потом я начинаю думать, что с ней делать. Развивая язык, всегда нужно тщательно следить за тем, чтобы невольно не добавить в него целую кучу функций, потому что чем больше функций вы добавите, тем быстрее состарится язык. В итоге язык просто обрушится под собственной тяжестью. Излишек возможностей ведет к частым конфликтам.

При добавлении новых функций нужно проявлять крайнее благоразумие, потому что в итоге не должно обнаружиться три разных способа сделать одно и то же, появившихся лишь в силу исторических причин. Поэтому не раз нам приходилось отвечать, что если бы мы начали все заново, то определенно включили бы ту функцию, о которой нас сейчас просят. Поскольку мы не можем начать все заново, мы не станем этого делать, потому что это относится к основам или в такой степени затрагивает их, что припарками тут не обойдешься. Это может привести только к появлению химеры, чего мы не допустим.

Что касается самого процесса разработки, то у нас очень сильная команда разработчиков C#, обычно от шести до восьми человек, которые регулярно встречаются. Последние 10 лет мы проводили регулярные встречи от часа до трех каждые понедельник, среду и пятницу. Иногда встреча отменяется, но тем не менее они отмечаются в наших расписаниях уже 10 лет и останутся в них. Участники процесса меняются. Я участвую в нем на протяжении всего времени. Скотт Уилтамут (Scott Wiltamuth) – тоже достаточно давно. Люди приходят и уходят, но процесс сохраняется достаточно долго.

Так у нас идет разработка. На этих совещаниях мы занимаемся текущей работой. Для того чтобы продукт непрерывно развивался, очень важно, чтобы разработка была непрерывным процессом. Очень часто люди работают порывами. «Ой, пора выпустить новую версию. Давайте несколько раз встретимся и решим, какой она должна быть». После этого проходит ряд совещаний, люди разбегаются, и потом год никакой разработки не ведется. Через год в срок выпуска очередной версии даже собрать вместе тех же самых людей не удается. В итоге получается шизофренический продукт, в котором ни одна версия не похожа на предыдущую. Когда разработка непрерывна, поддерживаемый вами продукт имеет узнаваемое лицо.

Кроме того, хорошие идеи возникают не по графику. Они просто возникают. Если у вас нет процедуры для регистрации хороших идей, если

вы не начнете разработку немедленно, идея вполне может затеряться. Мы всегда долго обсуждаем новую версию, которую собираемся выпустить следующей, и ту, которая выйдет после нее, делая это непрерывно. Думаю, это весьма эффективно.

C# участвует в процедуре стандартизации ECMA, что редко для языков. Почему?

Андерс: Во многих случаях стандартизация является необходимым условием для принятия. Есть организации – в меньшей степени бизнес, – но для правительственных учреждений стандартизация практически обязательна. В академической сфере тоже. Это действительно дает интересные преимущества, если Microsoft что-то стандартизирует. Когда мы создаем технологию вроде .NET, неизменно появляются ее реализации, сделанные сторонними фирмами для других платформ, и если позволить им наугад копировать то, что вы создали, это делается с ошибками, и получаются плохие результаты. Плохие результаты будут и у тех клиентов, которые в основном работают с вашей реализацией, но которым нужна и та, другая реализация, например для имеющихся у них старых машин.

Если все учесть, то в стандартизации действительно есть смысл, даже с точки зрения бизнеса. Кроме того, это заставляет быть очень аккуратным в работе, что имеет много внутренних преимуществ. Для стандартизации C# нам пришлось написать очень краткую спецификацию языка. Это очень точная спецификация языка – затраченные на нее средства многократно окупились в наших внутренних делах.

Это проявилось в улучшении сред для тестирования нашего отдела QA, в лучших инструментах исследования реализации новых функций языка, потому что стало совершенно ясно, как должны работать прототипы компиляторов. В изучении языка наличие очень краткой спецификации означает возможность использовать ее как справочник, вместо гаданий.

Стандартизация помогает нам гарантировать обратную совместимость кода. Так что есть масса неочевидных преимуществ. Пройдя процедуру стандартизации, вы подвергнете свой продукт рассмотрению очень грамотных технически специалистов. Мы получили массу отзывов от других компаний и отдельных лиц, участвовавших процедуре стандартизации, что позволило улучшить C#. Это тоже важно. Эти люди и организации могли бы и не проявить такого интереса, если бы не стандартизация.

Однако стандартизация не поспевает за развитием языка.

Андерс: Верно. Стандартизация в некоторой мере замедляет дальнейшее развитие. Многое зависит от формулировок. Иногда в стандарте

пишут так: «Вы должны реализовать это, и ничего больше, а расширение данной спецификации будет считаться нарушением стандарта». Мне никогда не нравились такие формулировки. Стандарты должны устанавливать общий минимальный набор и, возможно, способ гарантировать, что вы придерживаетесь этого базового набора и не нарушаете его. Но стандарты обязательно должны оставлять свободу для нововведений, потому что следующая версия стандарта так и создается – отбором некоторых из этих нововведений. Это нельзя отменить.

Для C# есть стандарт, но он не помешал нам развиваться дальше. Просто развитие совершается вне процесса стандартизации, потому что она не дает вам никаких нововведений. Это не ее задача. В рамках структуры, в которой вы действуете, инновации должны возникать в других местах.

Что вы думаете о формальном аспекте разработки языка? Одни предлагают сначала составить на листке формальную спецификацию, а потом писать код. Другие вообще игнорируют формальные спецификации.

Андерс: Не нужно впадать в крайности. Язык без формальной спецификации обычно превращается в помойку. Язык, в котором сначала все формально специфицируется, а потом с запозданием реализуется компилятор, тоже не очень приятен в работе. Работая над C#, мы параллельно писали компилятор и спецификацию, и эти два процесса оказывали сильное взаимное влияние. Когда мы писали компилятор, то сталкивались с проблемами, которые приходилось решать в спецификации. Либо при написании спецификации и попытке строгого анализа всех возможностей мы находили нечто, заставлявшее вносить изменения в компилятор, потому что обнаруживались другие случаи, о которых мы не подумали.

Думаю, важно и то и другое. Я рад, что мы осуществили стандартизацию, потому что она заставила нас очень четко разобраться с языком и его работой. Кроме того, она заставила нас написать формальную спецификацию, которой, как вы говорите, у некоторых языков нет, а это нехорошо. Когда спецификацией служит исходный код, то для того чтобы разобраться, что происходит в некоторой программе, нужно читать исходный код компилятора. На это способны немногие. Единственная альтернатива – гадать или писать тесты в надежде проверить все крайние случаи. Вряд ли это правильный путь.

Кстати, как вы отлаживаете код C#?

Андерс: Мой главный инструмент отладки – Console.Writeline. Честно говоря, думаю, так работают многие программисты. В более сложных случаях я применяю отладчик – если нужно сделать трассировку сте-

ка, посмотреть локальные переменные и так далее. Но очень часто можно быстро добраться до сути с помощью простых маленьких проверок.

Есть ли принципы, которых вы придерживаетесь при разработке API?

Андерс: Ну, прежде всего скажу, что они должны быть проще, но что это значит? Звучит глупо, правда? Я твердо верю в то, что API должны содержать как можно меньше методов и как можно меньше классов. Есть люди, которые считают, что больше значит лучше. Я не из их числа. Я считаю, что важно определить, для чего, как вам кажется, чаще всего будут использовать ваш API. Определите пять возможных типичных сценариев работы пользователей и сделайте так, чтобы API был для них максимально прост. В идеале – чтобы было всего одно обращение к API. Не должно быть так, чтобы в обычном сценарии приходилось писать много строк обращений к API. Это указывает на плохой уровень абстракции.

Однако я считаю, что в API всегда нужно оставлять выход. Нужен плавный переход от простейшего варианта использования API к более сложному применению, если это требуется. Во многих API есть некая ступенчатая функция. Да, там есть несколько простых методов, но в тот момент, когда потребуется сделать что-то чуть посложнее, вы резко спускаетесь с вершин на землю. Чтобы продвинуться немного дальше, вам придется изучить много такого, о чем вы понятия не имели. Я глубоко убежден, что этот переход должен быть более плавным.

Что вы думаете о документации?

Андерс: В целом состояние документации для ПО ужасно. Я постоянно убеждаю программистов и пытаюсь проповедовать это у себя, хотя и не всегда успешно, что ценность поставляемого вами клиенту продукта наполовину состоит из хорошей документации по API. Самый замечательный API бесполезен без документации, описывающей, что он делает и как им надо пользоваться. Это проблема. Во множестве компаний одни люди пишут код, другие документацию, и те и другие между собой не общаются. В итоге в документации вы видите «MoveWidget перемещает виджет» или пространное описание очевидного. Это безобразие. Программисты должны писать больше документации, чем делают это сейчас.

Как вы считате – размещать комментарии нужно внутри кода или в отдельных документах?

Андерс: Я всегда ратовал за комментарии в коде в формате XML. Когда вы поместили комментарии в код, есть шанс, что работающий с этим кодом программист обратит внимание на ошибки в документации. Может быть, он даже исправит их. Когда документация находится где-то

в отдельном файле, программист и не взглянет на нее, и она так и останется с ошибкой.

Смысл в том, чтобы свести код и документацию как можно ближе друг с другом. До идеала далеко, но мы стараемся.

Как, по вашему, можно стать хорошим программистом на C#?

Андерс: Это трудно. Я бы порекомендовал выбрать хорошую книгу по программированию на C#. Не буду перечислять названия – хороших книг много, и они помогут вам лучше освоить C# и среду .NET. Много полезного можно найти и в Сети. Есть такие вещи, как Codeplex. Есть целый ряд проектов open source, которые можно взять и учиться по ним, и так далее.

Чтобы стать хорошим программистом вообще, я лично старался смотреть на разные стили программирования и разные типы языков. За последние 5–10 лет я многому научился, присматриваясь к функциональному программированию – весьма специфическому способу программирования, в котором можно почерпнуть много полезного. Это тоже программирование, но под другим углом, и эту иную точку зрения я считаю крайне полезной.

Будущее компьютерных наук

Что вы считаете главными проблемами компьютерных наук?

Андерс: Если даже взглянуть выше, на метауровне, то постоянной тенденцией в развитии языков является неуклонный рост уровня абстракции. Если посмотреть на весь путь от коммутационных панелей и машинного кода до символического ассемблера, Си, C++, а теперь и управляемых исполнительных сред, то с каждым шагом мы поднимались на новый уровень абстракции. Главная задача – всегда искать очередной уровень абстракции.

Сейчас есть несколько конкурентов на крупный шаг вперед. Об одном мы уже говорили: это параллельная обработка, создание содержательных программных моделей для параллелизма, понятных широким массам программистов, а не только узкому кругу специалистов. Вот в таких условиях мы сейчас находимся. Даже крупные специалисты иногда с изумлением смотрят на собственный код. Это большая проблема.

Кроме того, сейчас много говорят о проблемно-ориентированных языках и метапрограммировании. По-моему, здесь больше разговоров, чем дел. Едва ли нам сейчас известны решения. Есть «аспектно-ориентированное программирование», есть «интенциональное программирование», но решение пока не найдено.

Одни вам скажут, что у нас нет проблемно-ориентированных языков, другие – что их у нас полно. Мы не можем даже договориться по поводу того, что такое проблемно-ориентированный язык, но ясно, что это связано с попытками разработать более декларативные способы высказываний. В некоторых отношениях императивный стиль программирования исчерпал себя. Со своими императивными языками программирования мы далеко не уйдем. Не стоит надеяться, что, добавив несколько новых операторов, мы увеличим свою продуктивность сразу в 10 раз.

Думаю, большинству сегодняшних языков программирования присуще одно свойство: они вынуждают вас чрезмерно точно описывать решение вашей задачи. Вам приходится писать всякие вложенные циклы и условные операторы, тогда как в действительности вы всего лишь хотели соединить два фрагмента данных. Но средств, чтобы сказать это, у вас нет. Приходится влезать во все эти хеш-таблицы, словари и прочее.

Вопрос в том, как перейти к такому более описательному уровню программирования. Конечно, чем ближе к нему, тем больше у вас образуется понятий, потому что ваш язык становится более проблемно-ориентированным. Этими мечтами о проблемно-ориентированных языках уже никого не удивишь, и тем не менее мы не нашли правильного способа их реализации. Пока. Поэтому такая проблема остается.

В настоящее время наблюдается любопытное возрождение динамических языков программирования. Такое ощущение, что это происходит не из-за их динамических свойств, а скорее, благодаря заложенным в них богатым возможностям метапрограммирования. Если взглянуть на Ruby on Rails, там все построено на средствах метапрограммирования Ruby, а его динамичность имеет второстепенное значение. Просто так оказалось, что `eval` и метапрограммирование значительно проще в динамическом, а не статическом языке.

С другой стороны, это дорогая плата за то, чтобы отказаться от завершения операторов, обнаружения ошибок на этапе компиляции и прочего.

Многие из тех, кто занимается динамическими языками, приводят в качестве аргумента броузер Smalltalk.

Андерс: Я бы не сказал, чтобы он мне очень нравился. Хорошо, когда у вас сравнительно маленькая задача, как оно и было, когда появился Smalltalk. При нынешних размерах программных сред нереально думать, что люди запомнят все API какого-нибудь объекта или даже захотят это сделать. Такие инструменты, как завершение операторов, Intellisense, рефакторинг под управлением метаданных этапа компиляции или статическая типизация, просто бесценны. Их ценность и дальше будет расти, поскольку окружающий мир становится все сложнее. Сейчас мы наблюдаем подъем динамических языков программирова-

ния, но, думаю, в значительной мере он обусловлен 1) интересом к метапрограммированию и 2) сложностью среды J2EE.

Я знаю многих Java-программистов, переметнувшихся к Ruby, потому что они уже не в состоянии выдерживать программную среду, Strut, Spring, Hibernate и прочее. Не будучи каким-то технологическим гением, невозможно самому собрать все это вместе.

Должно ли программирование стать более доступным для тех, кто не является технологическим гением и не хочет им становиться?

Андерс: Думаю, да. Все зависит еще от того, что считать программированием. Потому что в некотором смысле работа с электронной таблицей – тоже программирование. Если бы можно было сделать так, чтобы люди программировали, не замечая при этом, что они программируют, это было бы прекрасно. Не думаю, что обычных пользователей нужно учить писать программы в тех программных средах, которыми сегодня пользуются разработчики. Да, программировать, но на гораздо более высоком уровне.

С чем мы столкнемся сегодня и в ближайшие пять лет?

Андерс: Параллелизм – большая проблема уже сегодня. Мы уже столкнулись с ней и должны найти ее решение. Одна из сложнейших моих задач в обозримом будущем – развернуть нашу команду в этом направлении.

Конечно, хотелось бы осуществлять развитие постепенно, но как решить проблему общих состояний и побочных эффектов, не нарушив работу всего существующего кода? Пока неизвестно, но вполне может статься, что параллелизм повлечет крупные изменения в парадигме программирования, и потребуются совершенно новые языки или совершенно новые программные среды. Хотя, думаю, до этого еще не дошло.

Полагаю, мы многого достигли бы, создав средства, позволяющие писать высокопараллельные API, причем тому, кто хорошо разбирается в конкретной области, будь то преобразования, расчеты, обработка сигналов или изображений. При этом API должен выглядеть снаружи вполне синхронным, а внутри быть сильно распараллелен.

Чтобы сделать это возможным, в нынешние языки нужно кое-что ввести. Одна такая вещь уже есть – возможность передавать код в качестве параметров. По мере роста сложности API уже недостаточно передавать им простые значения или структуры данных. Им нужно передавать фрагменты кода, которые API организуют в определенном порядке и исполняют.

Вам понадобятся функции высшего порядка и такие абстракции, как map, fold и reduce.

Андерс: Именно так – функции высшего порядка. Для этого вам понадобятся лямбда-исчисление, замыкание и все такое прочее. Чтобы это работало в параллельной среде, нужно обеспечить чистоту этих лямбд или отсутствие побочных эффектов. Выполним ли этот код в параллельной среде автоматически или есть побочные эффекты, делающие это невозможным? Как это выяснить? В нынешних языках таких возможностей нет, но можно подумать, как их добавить. Конечно, при этом нужно, чтобы они не слишком вас сковывали и не слишком нарушали работу того кода, который уже есть. Это большая проблема.

Мы в нашей команде думаем над ней ежедневно.

Потребность параллелизма влияет только на реализацию или и на разработку языка?

Андерс: Конечно, разработка изменяется. Многие тешили себя надеждой, что можно будет добавить ключ parallel при вызове компилятора, и он быстренько все скомпилирует для параллельного исполнения. Такого никогда не будет. Пробовали – ничего с нашими основными императивными языками вроде C++, C# и Java не получается. Эти языки очень тяжело распараллеливать автоматически, потому что в программах очень многое зависит от побочных эффектов.

Нужно сделать несколько вещей. Прежде всего, нужно сконструировать новые API для параллельной обработки, которые будут действовать на более высоком уровне, чем потоки, блокировки и мониторы, которыми мы пользуемся сейчас.

Затем в язык нужно ввести некоторые вещи, которые сделают новый стиль программирования проще и безопаснее, например гарантию неизменяемости объектов, чистые функции без побочных эффектов, анализ изоляции объектных графов, чтобы вы знали, не была ли данная ссылка на объектный граф использована кем-то еще, и если не была, то можно изменить ее, а если была, то возможны побочные эффекты. Эти вещи и другие – когда компилятор проводит какой-то анализ и помогает обеспечить безопасность, вроде нынешней безопасности типов и памяти, и так далее.

Вот кое-что из того, что должно осуществиться в ближайшие 5–10 лет, чтобы помочь нам программировать на этих параллельных системах.

Фактически вы говорите компьютеру, что он должен делать.

Андерс: Это крупная проблема того весьма императивного стиля программирования, которого мы сейчас придерживаемся, где все непомерно расписано по деталям. И это причина, затрудняющая автоматическое распараллеливание.

Сможем ли мы в будущем возложить проблемы параллелизма на программную среду?

Андерс: Думаю, да. Есть разные виды параллелизма, но если вы о параллельной обработке данных, как при операциях над большими наборами данных, скажем, при обработке изображений, распознавании речи, расчетах, то я считаю очень вероятным и очень правильным иметь модель, где это рассматривается просто как API. Есть API более высокого уровня, где вы можете сообщить API: «Вот данные, а вот операции, которые мне нужно с ними выполнить. Пойди и сделай все как можно быстрее, используя все свободные ЦП».

Это интересно, потому что сегодня достаточно легко сказать: «Вот данные». Можно просто передать ссылку на большой массив, объект или что-то еще. Если указывать операции, то обычно нужно дать ссылки на фрагменты кода, с помощью делегатов или лямбда-функций, и неплохо было бы, если б компилятор проверил, нет ли у этих лямбд побочных эффектов, и предупредил, если есть. Это часть того, о чем я говорю, но это лишь один вид параллелизма. Есть другие типы параллелизма для асинхронных распределенных систем, это параллелизм другого типа, где тоже могут помочь языки программирования. Возьмите язык Erlang, используемый в сильно масштабируемых распределенных системах. Там совершенно иная модель программирования, гораздо более функциональная и основанная на асинхронных агентах и обмене сообщениями. Там есть интересные вещи, которые могут оказаться полезными и в наших языках.

Есть ли проблемы в объектно-ориентированной парадигме?

Андерс: Это зависит от того, что вы включаете в объектно-ориентированную парадигму. Полиморфизм, инкапсуляция и наследование сами по себе не составляют проблему, хотя в функциональных языках обычно другое представление о том, как должен применяться полиморфизм к их алгебраическим типам данных. Кроме того, думаю, самая большая проблема, касающаяся ООП, состоит в том, что ООП ведут в очень императивном стиле, когда объекты инкапсулируют изменяемое состояние, и вызывают их методы или посылают сообщения, заставляющие их модифицировать себя, оставляя в неведении других людей, использующих те же объекты. В результате получаются побочные эффекты, которых вы не ждете и причина которых вам неизвестна.

В этом смысле у ООП проблема, но вы можете применять его и к неизменяемым объектам. Тогда таких проблем не возникает. Например, нечто похожее происходит в функциональных языках.

С учетом вашего интереса к функциональному программированию, следует ли изучающим компьютерные науки уделять больше внима-

ния математике и экспериментам с функциональным программированием?

Андерс: Я совершенно уверен в необходимости включать функциональное программирование в учебные планы по компьютерным наукам. Должно ли оно преподаваться в самом начале, сказать не могу. Не уверен, что знакомство с программированием должно начинаться с функционального программирования, но в учебном плане оно должно быть обязательно.

Какие уроки можно извлечь из вашего опыта?

Андерс: Если взять первый продукт, над которым я трудился, Turbo Pascal, то для него очень характерно желание не идти стандартными путями. Не нужно бояться. Если вам говорят, что этого сделать нельзя, это не всегда означает, что это действительно невозможно. Это просто значит, что они не знают, как это сделать. Я считаю, что всегда интересно думать нестандартно и искать новые решения для старых проблем.

Думаю, всегда побеждает простота. Если можно найти более простое решение, то для меня это всегда было руководящим принципом. Всегда старайтесь сделать проще.

Думаю, чтобы достичь успеха в любом деле, нужно очень его любить. А этому научиться нельзя. Это либо есть, либо нет. Я занялся программированием не потому, что рассчитывал заработать много денег или кто-то меня к этому принуждал. Я занимался этим делом, потому что был совершенно им поглощен. Меня невозможно было остановить. Я ничего другого не хотел, кроме как писать программы. Я был очень, очень увлечен этим.

Чтобы достичь совершенства в чем-либо, необходима страсть, именно она заставит вас отдавать любимому делу все свое время, а время – это ключ ко всему. Нужно вложить очень много труда.

14

UML

Как передать другим идеи, касающиеся устройства ПО? У архитекторов для этого есть чертежи. UML (унифицированный язык моделирования) – это графический язык для представления артефактов программного проекта. Объединивший в себе объектно-ориентированный анализ Джеймса Рамбо, объектно-ориентированное проектирование Гради Буча и объектно-ориентированное программирование Айвара Якобсона, он позволяет разработчикам и аналитикам моделировать ПО с помощью особого вида диаграмм. Для этого языка есть ряд последовательных стандартов, но какими-то его идеями вы наверняка пользовались, когда наскоро чертили схемы на доске.

Изучение и преподавание

Я читал, что в начале своей работы в Ericsson вы почти ничего не знали о программировании. Как вы его изучили?

Айвар Якобсон: Я начал работать в Ericsson, ничего не понимая в телекоммуникациях, и это был ценный опыт. Но работая там, где разрабатывали аппаратные коммутаторы, я смог создать на этой основе целую теорию построения крупных систем. Я проработал там почти четыре года и научился представлять систему в целом. Это оказалось очень ценным навыком, потому что те, кто разрабатывал ПО, не имели опыта создания крупных систем.

Я был инженером-электриком и, наверное, единственным с научной степенью по инжинирингу. У большинства сотрудников не было научных степеней. В университете я научился решать задачи и обрел твердую уверенность, что смогу решить практически любую конкретную проблему.

Вы брали домой код ассемблера, изучали его по вечерам, и у вас возникали вопросы к его разработчикам.

Айвар: Документации почти не было: ее писали те, кто мало понимал в программировании. Они описывали технические требования и делали это в виде неких диаграмм, противоречивых и неполных.

У нас тоже были диаграммы, которые чертили и использовали наши сотрудники, но в них не было компонентов, поэтому они разрастались до огромных размеров. Мы писали комментарии к каждой строке кода ассемблера, но люди учились, в основном, путем совместной работы, общения и чтения кода. Вопросы, которые я задавал по поводу прочтенного накануне вечером кода, были в основном такими: «Что вы в действительности хотели тут сделать?»

Чтобы понять некий фрагмент кода, приходилось перечитывать его от трех до пяти раз, но я был упрям и разобрал во время вечерней работы большой объем кода. Днем я был сильно занят администрированием проекта. Я был руководителем проекта, а эта функция не предполагает глубокого знания технологий, но требует понимания проекта и хотя бы способности выяснить у людей, чем они заняты. Я был, скорее, администратором проекта. Мне не нравилась эта роль, и я, как только достаточно расширил свои знания, стал принимать более непосредственное участие в работе. Мне хватило всего трех месяцев, чтобы прийти к выводу: то, чем мы занимались, ни за что не станет продуктом.

В нашем проекте в тот момент было занято 75 человек, и наша работа имела первостепенное значение для Ericsson. Можете себе представить

руководителя проекта, который приходит к начальнику и говорит ему: «Это ни за что не станет продуктом»?

Как у вас появилась концепция сценариев использования?

Айвар: Вполне естественным путем. В телекоммуникационной отрасли есть понятие *сценариев телефонной нагрузки* (*traffic cases*). Они похожи на сценарии использования, но относятся только к телефонным звонкам. У нас не было никаких сценариев использования или телефонной нагрузки для других функций коммутатора, даже если они составляли более 80% кода, например эксплуатации и обслуживания. В этой программе мы просто называли их *функциями* (features). Существовал длинный список таких функций, и было очень трудно выяснить, как они связаны между собой.

У нас было два разных понятия – сценарии телефонной нагрузки и функции. Они существовали в этой отрасли уже лет 50, не меньше, но объединить их было не так просто. Я напряженно пытался найти одно универсальное понятие, пригодное для описания всех типов взаимодействия с системой. Я попытался взглянуть на систему снаружи, как на черный ящик. Я хотел выделить все сценарии, которые могли быть полезны пользователям. Точным переводом со шведского был бы usage case, но я не очень хороший переводчик, так что появилось просто use case, чему я теперь рад.

К апрелю 1986 года мои сценарии использования должны были стать самостоятельными, для чего я придал им подобие классов. Сценарий использования можно представить как объект, который существует, пока идет операция между пользователем и системой. Он может взаимодействовать и с другими пользователями, как в случае телефонного звонка. Одной из важных задач для меня было обеспечение повторного использования сценариев, поэтому мне требовалась более абстрактная система случаев использования, похожая на абстрактные классы в ООП. Аналогия с объектами и классами позволила мне найти объединяющее понятие (сценарий использования), которое можно было применить для описания и сценариев использования, и функций.

Потребовалось некоторое время, чтобы это понятие устоялось, но к 1992 году я определил все важное, что касалось сценариев использования. Я написал книгу «Object-Oriented Software Engineering» (Объектно-ориентированная разработка программ), изд. Addison-Wesley, и не вижу, чтобы с тех пор кто-либо добавил что-то существенное в эту область. Некоторым удалось лучше описать идею, например Курту Биттнеру (Kurt Bittner) и Яну Спенсу (Ian Spence) в книге «Use Case Modeling» (Моделирование сценария использования), изд. Addison-Wesley Professional. Оба они сейчас работают в моей компа-

нии. Их книга больше других подходит для первого знакомства с идеей и выяснения деталей.

Открытие аспектной ориентированности стало, конечно, шагом вперед, как и обнаружение того, что сценарии использования представляют собой хорошие аспекты. Результатом стало появление в 2005 году книги «Aspect-Oriented Software Development with Use Cases» (Аспектно-ориентированная разработка ПО со сценариями использования), изд. Addison-Wesley Professional, написанной совместно с Пан Вей Ыном (Pan Wei Ng), тоже работающим в моей новой компании.

Что произошло, когда вы ознакомили со своей идеей разработчиков в Ericsson?

Айвар: Первой реакцией моих коллег и руководства Ericsson в отношении методологии было «тут нет практически ничего нового». Я был уверен, что они просто ничего не видят. Мне сразу стало ясно, что сценарии использования служат еще и контрольными примерами, поэтому если заранее определить сценарии использования, вы получите много контрольных примеров, или вариантов тестирования (test cases). В 1986 году это было большой новостью. Мы могли вести разработку на основе сценариев использования, поскольку каждый сценарий использования содержит несколько планов действия, где описывается, как они реализуются путем взаимодействия между классами или компонентами.

Как можно распространять в области программирования опыт, подобный вашему?

Айвар: Это особая проблема, которой я занимался последние пять лет. Нужно разобраться, какими знаниями вы владеете, чтобы суметь описать их. Людям тяжело осваивать узкоспециальные идеи. Даже если у вас лучший в мире процесс, вы должны передать это знание другим людям. Нужны систематические идеи, а затем вы передаете свои знания путем передачи системы знаний. Это можно сделать лучше или хуже.

Пятнадцать лет назад у нас была методология Objectory. Из этого возникла модель жизненного цикла Rational Unified Process и был добавлен большой объем новых знаний, но технология фиксации знаний не была тогда полностью доступна. В то время это было лучшее, что можно сделать. Это была уникальная система, подобное еще не делалось в столь больших масштабах.

Сейчас мы продвигаем передачу знаний «на базе практик». Вместо того чтобы передавать все знания обо всем, что касается разработки ПО, вы передаете практики по одной, и только когда они очень нужны другим. Практики маленькие и достижимые, и все внутри одной практики логически совместимо, поэтому ее легко освоить, тогда как процесс тре-

бует постоянно держать в голове много материала. Можно сказать, что раньше процесс был просто винегретом идей. Теперь мы делаем процесс набором практик.

Каким должен быть подход к преподаванию компьютерных наук?

Айвар: Проблема в том, что у большинства университетских преподавателей очень слабое представление о технологии. Мало кто из них сам разработал какую-нибудь полезную программу. Бывает, кто-то из них разработал компилятор, но их компиляторы обычно слишком теоретичны. Бывает, что они пишут программы, облегчающие преподавание. Нельзя рассчитывать на то, что они смогут преподавать технологию создания ПО.

Можно научиться программированию на Java или тех языках, которые преподавали раньше, на университетском уровне, но если вы действительно хотите разбираться в программировании, необходима компетентность во многих других областях, таких как технические требования, архитектура, тестирование, блочное тестирование, интеграционное тестирование, системное тестирование, тестирование производительности, не говоря уже об управлении конфигурацией, управлении версиями, различии между созданием сред и созданием приложений, создании многократно используемого ПО, архитектуре, ориентированной на сервисы, архитектуре линейки продуктов и так далее.

Действительно сложным вещам в университете не научишься.

Должны ли студенты получать больше практического опыта, например участвуя в проекте open source или проходя практику в крупной компании?

Айвар: Обучение в университете происходит преимущественно путем получения образования и разработки простых программ.

У меня, разумеется, нет полной картины того, как обстоят дела в мире в целом, но сравним наше положение с другой инженерной дисциплиной, например со строительством. Там есть архитектурное образование, весьма далекое от возведения зданий. Несколько разных дисциплин соединены вместе. Если вы учите людей на архитекторов, то они должны уметь что-то строить, иначе зачем они нужны? Мечтать хорошо, но если ты не в состоянии реализовать свои мечты, какая от них польза?

Мы действительно не учим людей технологиям. Создание ПО – в большей мере технология, нежели искусство. Многим хотелось бы, чтоб было иначе, но заниматься искусством могут позволить себе лишь очень немногие профессиональные программисты. Остальные – инженеры. Это не значит, что их работа не творческая. Разве кто-то будет утверждать, что те, кто получил образование в области машиностроения и, скажем, умеют создавать различные механизмы, занимаются не творческой

работой? Разве построить корабль – не творчество? Или дом? Конечно, творчество. И архитектор – тоже очень творческая профессия.

Таким образом, мы должны осознать, что разработка ПО – это инженерная работа, а не искусство. Инженеров нужно учить инженерному делу. Тем не менее во многих университетах США и Европы существует давняя традиция, в соответствии с которой преподавание ведется в чисто академическом стиле. Я вижу фундаментальную проблему в том, что у нас нет настоящей теории для программной инженерии. Для большинства людей программная инженерия представляется набором идей для конкретных случаев. Это одна из главных проблем, которые мы должны решить.

Пожалуйста, поясните, почему вы считаете эту проблему такой важной.

Айвар: Наши взгляды на то, как нужно разрабатывать программы, резко меняются каждые два-три года – чаще, чем мода. Крупные компании во всем мире легко отказываются от значительных инвестиций в процессы и инструменты едва ли не сразу после того, как опробуют их. Вместо использования накопленного опыта они небрежно переходят на нечто, кажущееся им принципиально новым. На самом деле, мало что меняется. Здесь как в мире моды: много шуму практически из ничего. В таких тривиальных вещах, как одежда, это, может быть, и допустимо, но с учетом наших инвестиций в ПО это расточительно, дорого и абсурдно.

Последняя тенденция – гибкая (agile) разработка, воплощенная в Scrum. Это движение напомнило нам, что первое и главное в разработке ПО – люди. Здесь нет особой новизны: данная тема всплывает примерно раз в десять лет, когда наивные менеджеры пытаются механизировать и коммерциализировать то, что по существу является упражнением в творческом решении задач. Важно, чтобы мы не забывали, как работать в составе команды, как документировать свою работу, как планировать ее на день, на неделю, на месяц, и так далее. Но когда мы снова обращаем свое внимание на эти вещи, многое теряется или затемняется новыми названиями для старых вещей, создавая иллюзию полной новизны.

В результате напрасно тратится много труда, когда заново открываются и перелицовываются старые истины. Молодые и менее опытные сотрудники поддерживают новые тенденции, следуя за новыми гуру, в сопровождении рекламной шумихи в СМИ, всегда охочих до «новостей». Менеджеры, утратившие контакт с настоящими разработками, оказываются в безнадежном положении: сопротивляясь «новейшим веяниям», они выставляют себя как ретрограды. Запускаются опытные проекты, которые должны быстро подтвердить достоинства нового подхода, но

когда масштабы ограничены, воодушевленные разработчики могут заставить работать что угодно. В результате старый метод побеждается новым, и на помойку выбрасывается как то, что работало в старом подходе, так и то, что не работало. С большим опозданием обнаруживается, что новый подход тоже в одних местах работает, а в других – нет.

Корни этой проблемы – в глубоком непонимании природы разработки ПО. Ученые пытались решить эту проблему с помощью новых теорий, таких как формализм для доказательства корректности программ или формальные языки, никогда не получившие признания за пределами академического мира. Промышленные проекты годами пытались стандартизировать раздутые метамодели, которые невозможно понять без больших усилий.

Университеты и технические институты учат нас работать определенным образом. В каждом проекте есть особый метод, который нужно изучить, прежде чем можно приступить к работе. При каждом переходе на новое место работы нам приходится изучать новый метод, прежде чем реально браться за решаемую задачу. Это неэффективно: невозможно накопить опыт, потому что каждый раз приходится начинать заново.

Нужно прекратить вечную погоню за капризами моды и легкими решениями, которые всегда приносят разочарование. Но как? Это проблема, над которой я размышлял не меньше 10 лет, и сейчас у меня есть для нее конкретное решение.

Что вы предлагаете?

Айвар: Нам нужна базовая теория, которая ответит на вопрос, что на самом деле представляет собой разработка ПО. Мне кажется, эта теория у нас где-то под самым носом. Нужно лишь поймать ее. Взять все эти методы, процессы и практики и найти «истину» в разработке ПО. Например, можно сделать то, что мы сделали в моей компании и что теперь используется сотнями компаний во всем мире.

Прежде всего, нужно очертить круг важнейших вещей, которые у нас всегда есть или всегда делаются при создании программ. Например, мы всегда пишем код, всегда тестируем его (хотя не всегда записываем, как он тестировался), всегда учитываем технические требования (документированные и недокументированные), всегда имеем резервы (явные или подразумеваемые) и всегда действуем по плану, хранящемуся на бумаге или в голове. Прибегнув к затасканной метафоре, можно сказать, что мы должны найти ДНК для программных разработок.

Мы с коллегами выявили более 20 таких элементов, изучив около 50 методов, включая экстремальное программирование и Scrum. На первый взгляд кажется, что есть большие различия между этими методами

и тем, как мы их применяем. Например, зафиксировать технические требования можно с помощью функций или сценариев использования. Но у этих двух методов есть общая основа, которую я отразил в своих базовых элементах.

Затем, опираясь на эти базовые элементы, мы описываем широко применяемые и опробованные методы и практики: архитектуру, Scrum, компоненты, итерации и так далее. На сегодня разработано около 15 таких практик. Поскольку это ядро не связано с какой-либо конкретной практикой, легко выяснить, в чем реальные отличия разных практик, – не внешние, а по существу. Это уменьшает значение культа, возникающего вокруг каждого метода. Образование станет более логичным, если сосредоточится на отдельных элементах вместо конкретного набора идей, составляющих каждый метод, процесс или методологию. Думаю, студентам это понравится.

Было бы прекрасно, если бы наши технические институты и университеты учили студентов основам программирования, а затем обучали их некоторым хорошим практикам, используя такую базу. Здесь есть большой простор для научных исследований.

Вспомним слова Курта Левина: нет ничего практичнее хорошей теории. Хорошая теория помогает учиться и расширять знания, не впадая в излишнюю религиозность.

Вы много путешествуете. Заметили ли вы какую-нибудь разницу в методах программирования или разработки в разных странах?

Айвар: Конечно, но то, что сейчас происходит в США, должно произойти и в остальных частях света. США, возможно, идут несколько впереди, опробуя новинки раньше других, но и там тоже выкидывают наработанное. Многие компании в США быстрее отказываются от того, что у них есть, в погоне за новизной, в Европе больше колеблются.

Восточная Азия в отношении новых технологий отстает на пару лет, зато им, возможно, удается не повторять наших ошибок.

Отчетливую тенденцию я наблюдал в Китае. Они идут по стопам Индии, а потому там очень большое распространение получила CMMI, лет пять назад достигнув пика. Сейчас они видят, что CMMI решает лишь одну часть проблемы – совершенствование процесса. Но чтобы улучшать процесс, нужно располагать чем-то достойным этого названия, поэтому сейчас они видят, что им нужны хорошие практики, позволяющие разрабатывать хорошее ПО быстро и с малыми затратами.

Как влияет культура на подход к разработке ПО?

Айвар: Не знаю. Обычно я замечаю, что у финнов в большей степени пастушеский менталитет, чем у остальных скандинавов. Они ближе

к земле и добиваются результатов. В финском языке есть специальное слово sisu, которое означает «никогда не сдаваться», и они серьезно к этому подходят и потому не делают лишнего. Многие, наверное, скажут, что характер финнов очень подходит к технологии Agile, и это положительный фактор.

В Скандинавии тоже очень хорошо с разработкой ПО. Возьмите, к примеру, Ericsson. Но, думаю, не стоит развивать эту тему, потому что у меня недостаточно фактов.

Роль человеческого фактора

Как узнать, способен ли данный человек быть архитектором программного проекта?

Айвар: Позвольте внести ясность. Архитектура – очень важная вещь, но по многим причинам я стараюсь не характеризовать отдельных людей как архитекторов. Я неоднократно встречал компании, где есть команда архитекторов, которых они посылают в другие организации для работы над проектами. Это может быть хорошо, если они работают в рамках определенного проекта, но в таких компаниях, как крупные банки, обычно есть группа архитекторов масштаба предприятия, которые сидят и вычерчивают схемы архитектуры. Затем они перекидывают их разработчикам. Разработчики говорят: «Что это? Это бесполезно». Во многих компаниях главные архитекторы сидят, оторвавшись от реальной жизни и не делая ничего полезного.

Я всегда считал, что архитекторы не должны выделяться в особый класс людей, потому что ПО создается командами, а не вертикально организованными системами.

Во многих командах пытаются пытаются разрабатывать ПО силами структур, состоящих из нескольких отделов, подразделений или групп. Одна группа занимается техническими требованиями, другая кодированием, третья тестированием, могут быть и другие. Затем такой структуре поручаются различные проекты, и появляются менеджеры проектов, которые работают с такими отдельными группами. Ответственность за технические требования лежит на руководителе группы технических требований. За тестирование отвечает руководитель группы тестирования. Это не команды, это отдельные группы, поэтому вы не знаете, в каком состоянии проект. Руководитель проекта – это просто администратор, а не руководитель, который может направлять работу. Результат – медленная и дорогостоящая разработка слабого ПО, потому что требования, которые пишет группа технических требований, трудно понять остальным.

Мы должны работать командами, в составе которых есть те, кто способен справиться с техническими требованиями, те, кто умеет проектировать ПО, и так далее. У команды есть руководитель и есть собственная организация. Это как в футболе: есть нападающие, есть защитники и вратари, но они при необходимости меняются ролями. Иногда нападающий становится в защиту, а защитник забивает гол. Такая модель нужна в программировании.

Команда должна вместе сражаться за победу, и ее участники должны помогать друг другу, так, чтобы те, кто пишет технические требования, понимали, с какими трудностями сталкиваются разработчики. Тогда те, кто пишет требования, могут обеспечить возможность проверки этих требований, а не формальное их наличие.

Вот новая модель: команда, а не организационная структура.

Как вы понимаете термин «социальная инженерия»?

Айвар: Социальная инженерия помогает наладить совместную работу людей. Она занимается организацией команды. Она занимается организацией вашего времени в течение рабочего дня, недели, месяца и так далее. Она не связана с технологией: она занята тем, как мотивировать ваших сотрудников, вызвать у них удовлетворение своей работой и как получить результаты.

На эту тему всегда было много книг по менеджменту, но программирование – новая для них область. Движение Agile, главная суть которого – решение подобных вопросов, появилось, когда организации закостенели под действием методологий вроде CMMI и RUP.

Я никогда не верил, что люди станут работать согласно RUP так, как их пытались заставить, потому что RUP нужно использовать, скорее, как базу знаний, базу мышления, а затем организовывать работу так, чтобы она имела смысл для людей. Я всегда это говорил. К сожалению, RUP стали рассматривать как перспективную технологию, скажем, приготовления пищи. Никто из тех, кто занимался разработкой ПО, и помыслить не может, что это можно делать шаг за шагом, ставя галочки в списке.

Почему так медленно удается улучшать методы и процессы программирования?

Айвар: Это серьезный вопрос. Думаю, наша отрасль еще очень незрелая. Она чуть более зрелая по сравнению с тем, что было 20 лет назад, но сегодня мы создаем гораздо более сложные системы. Двадцать лет назад у нас были язык программирования и операционная система, сейчас же существуют всевозможные вычислительные среды.

Программная индустрия гонится за модой сильнее, чем все другие мне известные. Что-нибудь сногсшибательное должно появляться каждые два-три года, иначе людям кажется, что никакого прогресса нет. Характер восприятия новых идей таков, что мы выкидываем не только плохие или старые идеи, от которых нужно отказаться, но выкидываем практически все и начинаем все заново. Мы не движемся вперед вследствие систематического обновления старого и добавления нового, так что в некотором смысле мы стоим на месте. Реальный прогресс не ощущается.

Новые популярные методологии сегодня не слишком отличаются от тех, которые у нас были 20 или 30 лет назад, но у них новые акценты, и мы иначе обсуждаем их. Мы также наблюдали противодействие крупным и вполне удачным процессам, таким как CMMI и RUP. Противодействие выражается в осуждении всего, что относится к этим и аналогичным движениям, и в пропаганде чего-то нового и свежего, хотя на самом деле оно не новое и не свежее. Эти «новые методологии» по сути не новшества, а варианты того, что у нас уже было.

На самом деле, Agile все-таки содержит кое-что новое: повышенное внимание к людям и социальной психологии. Хотя это не новость для всех, кто когда-либо работал над успешным ПО. Люди – главный актив разработки ПО. Наличие знающих и мотивированных работников – главнейшее условие быстрого и экономичного создания хороших программ. Мы не всегда помним это.

Другая наша проблема, думаю, в том, что приходящие к нам выпускники университетов обучены всяким новомодным штукам, но не умеют обращаться с коммерческим ПО, унаследованным из прошлого. Молодые и энергичные, они не желают связываться с тем, что считают устаревшим. Они просто откажутся от предлагаемой работы, особенно если экономическая ситуация не слишком тяжелая. Эти молодые и неопытные, но хорошо обученные люди начинают преобладать в организациях, а в результате мы не движемся вперед.

Как можно решить проблему устаревшего ПО?

Айвар: Программы традиционно разрабатывались людьми, не имевшими понятия о методологии. Они не могли описать то, чем занимались. Они не писали документации к тому, что делали. До сих пор очень трудно разобраться в структуре системы, если вы начали работать над ней с середины процесса, из-за чего вы не понимаете архитектуры системы и ее основных идей. Продолжить работу над такой системой с новыми людьми очень трудно.

Если в какой-то фирме все люди уволятся одновременно, фирмы не станет. Даже если у вас есть деньги, чтобы нанять новых работников, они

не будут знать, что им делать. Создание ПО в этом отношении не является исключением. Такова природа бизнеса. Хорошо, если новые люди смогут разобраться в вашей системе, но чудес ждать не стоит.

Нам нужны программы, в которых можно разобраться, с хорошей архитектурой и хорошими моделями. Управлять кодом без явной архитектуры почти невозможно.

Серьезной проблемой для большинства крупных компаний становится замена устаревших систем и способов их разработки и/или расширения. Эти системы связаны со своими практиками, развившимися с течением времени, и многие из этих практик не только не Agile, но и не совместимы с Agile. Изменение методов разработки для новых систем или продуктов представляет собой гораздо меньшую проблему. Выбранный подход должен быть оптимизирован для устаревших систем. Моя точка зрения следующая. Разработка имеющегося продукта – это процесс замены управления, переход к чему-то большему. Разработка заново – это просто особый случай, переход от ничего к чему-то. С этой точки зрения нужно рассматривать все, что вы делаете, в том числе и практики, которые вы применяете при разработке ПО.

В принципе, есть два подхода к управлению устаревшими системами и их модернизации.

Во-первых, можно ввести практики, которые в действительности не меняют продукт, но улучшают методы вашей работы. Ими могут быть итеративная разработка, непрерывная интеграция, разработка на базе тестирования, разработка на основе сценариев использования, user story (список пользовательских требований), парное программирование и перекрестные команды. Стоимость и риски введения таких практик незначительны, хотя для больших компаний все же существенны.

Второй подход более глубок: изменить действующий продукт с помощью таких практик, как архитектура (на простом уровне), архитектура уровня предприятия, архитектура линейки продуктов, компоненты и т. д. Это крупная перестройка. Стоимость и риски выше, но возврат на инвестиции намного выше.

Позволяет ли выбор правильного метода избежать проблемы управления системой без явной архитектуры?

Айвар: Не избежать, но ослабить. Составление документации для вашей системы может не оказать достаточного эффекта, потому что люди все равно не читают документацию. И все же крепкая документация, сосредоточенная на главном, полезна, так как делает систему более доступной. Например, если вы можете описать свою архитектуру, значит, у вас есть архитектура!

Тем не менее не стоит рассчитывать, что одни люди будут уходить, а другие приходить и просто продолжать их работу. Необходим переходный период, во время которого новые сотрудники смогут изучить структуру, с которой им придется работать. Сколько ни учи людей, если нет явной архитектуры, невозможно без затруднений передать знания о системе.

В каком формате лучше всего передавать знания?

Айвар: Обычно те, кто работает с программами, не читают книг или руководств. Если кто их и читает, так это студенты. Рассказы о том, что при работе люди пользуются книгами и руководствами, – просто миф.

Я написал пару книг и очень рад, что мои книги люди покупают, но все прочие книги – их просто не читают. Это в природе вещей, что люди не читают книг ни по процессам, ни по языкам.

Вместо того чтобы изучать объемистые методологии и языки вроде UML или Java, займитесь практиками. Практики более доступны. Специалистом по практике можно стать, не будучи экспертом по методологии в целом. Большинство моих коллег, написавших книги по методологиям, в действительности не были экспертами в мелких частностях методологии – практиках.

Вместо работы над объемистой методологией или языком, сосредоточьтесь в каждый момент только на одной практике. Ни один человек не в состоянии владеть всеми хорошими и полезными практиками, но вам, возможно, удастся заставить какие-то практики работать. Последние пять лет я занимался тем, что пытался сделать практики как можно проще и отделить одну от другой, но так, чтобы можно было составлять из них более крупные процессы (или какие-то технологии работы).

Я читал об использовании карточек.

Айвар: Все методологии начинаются с какой-то интересной новой идеи, потом они заимствуют другие интересные идеи, делают из всего этого винегрет и называют его методологией, процессом, методом – как больше понравится.

Замечательно, если вам при всем при том удастся обеспечить последовательность, законченность и корректность. Кое-кому это удалось. Некоторые стали признанными гуру.

Но это лишь малая часть задачи. Главная трудность в том, чтобы заставить людей принять ее. Другая проблема в том, сможете ли вы внести изменения, если на горизонте появятся другие идеи.

Собственно, мы в целом не достигли успеха в создании методов.

Сотрудники моей компании (а именно Брайан Керр и Ян Спенс) предложили некоторые интересные решения этой проблемы. Одно из них

заключается в использовании карточек для описания важных аспектов вашей работы или ее результатов в процессе разработки ПО.

Карточки дают гибкий способ описания практик. На них записывается самое важное; остальное можно домыслить самому.

UML

Как вы определите UML?

Айвар: UML – это язык схем ПО для описания, планирования, проектирования, тестирования и использования программ.

Какая связь между UML и разными методами разработки программ?

Айвар: Все различные методологии разработки ПО, которые OMG определила в начале 1990-х (26 штук, если я правильно помню), имели собственные системы обозначений, но сейчас большинство из них приняло UML.

То, что в вашей группе было три разработчика, давало какие-то преимущества или только вынуждало искать компромиссные решения?

Айвар: У нас были горячие споры, но они способствовали тому, чтобы в итоге получился более совершенный язык, чем каждый из нас создал бы в одиночку. Мы не смогли бы достичь этого результата без сотрудничества с такими людьми, как Дэвид Харел (David Harel), Джим Оделл (Jim Odell), Крис Кобрин (Cris Kobryn), Мартин Грисс (Martin Griss), Гуннар Овергаард (Gunnar Overgaard), Стив Кук (Steve Cook), Брэн Селик (Bran Selic) и Гуус Рамакер (Guus Ramacker).

Что бы вы хотели поменять? Что может измениться в UML?

Айвар: Вот самые важные изменения:

- Язык слишком сложный. Это нужно изменить. 80% приложений можно проектировать, используя не более 20% UML. В нашей компании мы определили подмножество UML, которое станет *базовым UML* (Essential Unified Modeling Language). Мы также весьма особым образом описываем UML, что должно привлечь обычных пользователей. Традиционный UML предназначен для специалистов и разработчиков инструментов.
- Я бы хотел иначе организовать UML – в виде набора проблемно-ориентированных языков (DSL). Это можно сделать аналогично тому, как в нашей компании переделали Unified Process. DSL – это часть языка моделирования (примером которого служит UML). Язык моделирования создается в виде собрания нескольких таких DSL (аспектов), подобно тому как программную систему создают из ряда перекрестных соображений. Я заявил, что язык создавался не

для пользователей, а для методистов и производителей инструментов, но добавлю, что и для этих двух групп он не слишком подходит. Семантика UML определена слабо. UML – в особенности UML 2.0 – содержит столько элементов, взятых из разных лагерей методологии, что четко определить его семантику невозможно. Как и многие другие языки, UML стал, по выражению Джона Бэкуса в отношении Ады, «жирным и дряблым».

Большинство усилий были направлены на конкретный синтаксис (значки) и в какой-то мере на статическую семантику, но операционная семантика осталась неопределенной. Я был готов к критике, потому что стандартная практика разработки языков в то время использовала такие приемы, как денотационная семантика. Ее не последовало. Мы писали страницы одну за другой, зная, что их будет трудно понять. Мы могли использовать те же практики, с помощью которых определяли SDL (стандарт моделирования с использованием VDM был установлен в телекоммуникациях еще в 1984 году). SDL стал языком моделирования с корректной семантикой. Несмотря на то что в UML были включены существенные части SDL, мы не приняли практики разработки языков, применявшиеся 15 годами ранее. Печально!

При всем сказанном, даже несмотря на отсутствие формального определения UML, он был спроектирован гораздо лучше большинства популярных языков ОО-моделирования. С появлением UML практически все конкурирующие языки были забыты.

Как вы определяете, какие элементы можно убрать из UML? Какова процедура упрощения языка?

Айвар: Я бы начал с основ языка. Я не стал бы рассматривать язык в целом и удалять из него отдельные куски. Я знаю, какие элементы языка действительно полезны, а какие – нет. Есть такие элементы, на которые я даже смотреть не хочу. Не конкретизируя, скажу, что мы уже определили их, и они составляют не менее 20%.

Когда мы преподаем UML, то это Базовый UML, который мы определили по своему опыту. Мы используем при описании элементов языка те же идеи, что при описании элементов процессов или практик: мы используем карточки, и на каждой карточке показана конструкция языка, например компонент, интерфейс и так далее. Речь идет о преподавании. Мы не говорим о каких-то новых вещах или новых языковых конструкциях. Мы выяснили, что никто не читает и не любит толстые тома спецификаций языка, поэтому мы должны найти более доступный способ обучения. Вы будете один за другим усваивать объекты, интерфейсы, классы, компоненты и так далее.

Как вы собираетесь «реорганизовать UML в виде набора проблемно-ориентированных языков»?

Айвар: В UML есть универсальное базовое ядро. Я определю аспекты этого ядра и опишу UML, последовательно добавляя аспекты. Эти аспекты UML – то, что мы называем практиками применительно к процессам, и эти похожие на практики аспекты UML станут проблемно-ориентированными языками.

Проблемно-ориентированный язык, как и говорит его название, должен обслуживать конкретную область, например вертикальный промышленный сектор (промышленные системы, телекоммуникационные системы, системы здравоохранения и так далее), или дисциплину (технические требования, проектирование, работа в реальном времени, тестирование и так далее). Сравнительно небольшой аспект UML составляет проблемно-ориентированный язык. Таким образом можно составить UML из разных проблемно-ориентированных языков. У этих проблемно-ориентированных языков должно иметься общее ядро и общая семантика, в противном случае задача перевода из одной предметной области в другую будет очень затруднена.

Существуют ли практики, используемые для проектирования SDL, с помощью которых можно было бы улучшить UML?

Айвар: Пятнадцать или двадцать лет назад, когда мы проектировали SDL, мы применяли венский метод разработки (VDM), разработанный IBM в конце 1960-х или 1970-х. Это математический язык, с помощью которого можно математически описать такие понятия, как язык, операционная система или любая другая система. Он основан на дискретной математике: теории множеств, отображениях и тому подобном. Таким способом можно действительно описать математически смысл каждого элемента языка.

Сначала мы определили абстрактный синтаксис и описали абстрактный синтаксис средствами дискретной математики. На этой основе мы определили домены элементов. Мы определили статическую семантику, описав, какие условия должны иметь истинное или ложное значение для элементов этого домена. Затем мы описали операционную семантику путем описания смысла конкретного оператора. Вот математический способ описания языка. Наконец, мы отобразили графическую нотацию в абстрактный синтаксис.

Я плотно занимался SDL, но не смог убедить своих коллег по UML сделать что-либо аналогичное для UML. Они считали это чистым теоретизированием. Имея опыт работы с SDL, я не согласился с ними, потому что как только вам понадобится делать инструменты, вам потребуется знание точной семантики. В противном случае, людям придется гадать.

Когда Стив Кук из IBM и Брэн Селик из Objectime (позднее приобретенной Rational) собирались войти в команду, они сказали: «Это непрофессионально. Мы не станем участвовать, если язык не будет определен более формальным образом», – поэтому я предложил компромисс. Я сказал: «Давайте определим абстрактный синтаксис и статическую семантику математически, но опишем операционную семантику на обычном английском языке». UML 2.0 лучше UML 1.0, но этого недостаточно, если вы действительно хотите разобраться во всех деталях.

Что вы думаете о генерации кода реализации посредством UML?

Айвар: Принципиальной необходимости иметь два типа языков нет. Зачем нужен язык для описания вашего проекта, если ваш проект является абстракцией реализации? И зачем тогда еще один язык для описания реализации? Такова сегодняшняя ситуация, создающая перекрытие.

Есть несколько причин, по которым мы пользуемся двумя языками. Главная из них, видимо, та, что мы не сумели убедить теоретиков компьютерных наук в ценности языка моделирования: они считают, что достаточно языка программирования. В действительности, код – это язык, рассчитанный на машины (компиляторы и так далее) и не использующий все возможности человеческого мозга.

Думаю, когда-нибудь мы сумеем ясно продемонстрировать значение визуального моделирования, убедив ученых заняться исследованиями в этой области. Есть много исследований UML, и фундаментальные основания для того, чтобы иметь два вида языков, не найдены, но пока ситуация такова.

Дело лишь в том, чтобы направить сюда внимание?

Айвар: Дело в том, чтобы в академической сфере поняли, что не все может быть хорошо описано кодом. Многие там понимают это, но таких людей не достаточно. Мы должны продемонстрировать большие успехи.

UML принципиально лучше всего, что было прежде. SDL был очень полезен в области телекоммуникаций, но UML более универсален (в нем есть важные языковые конструкции, отсутствующие в SDL). UML создан в конце 1990-х, и поскольку за это время не появилось ничего принципиально лучшего, можно рассчитывать, что пройдет еще лет 20–30, прежде чем UML заменят чем-то другим. За это время мы можем усовершенствовать технику обучения UML.

Я считаю, что UML докажет со временем свою ценность. Нам нужно что-то вроде UML, чтобы облегчить масштабирование программных разработок. Может быть, больше людей с практическим опытом займется исследованиями. Может быть, они покажут, что так учить студентов, как это делается сейчас, бесперспективно.

Есть ли минимальный объем программного проекта, ниже которого применение UML увеличивает сложность и труд, не принося выгоды?

Айвар: Если к стоимости проекта добавить стоимость изучения UML и стоимость изучения инструментов поддержки UML, она может оказаться слишком высока, чтобы оправдать выбор UML. Но если, начиная новый проект, люди вынуждены разобраться в UML и хотя бы одном из поддерживающих его инструментов, картина становится иной.

Если вы собираетесь учить людей основам разработки ПО в обычное рабочее время, у них может быть трудно вызвать заинтересованность, особенно если это готовый небольшой проект. Для больших проектов мотивация совсем иная, потому что риски, связанные с плохим моделированием, очень высоки.

Допустим, я побаиваюсь использовать UML. Как бы вы могли убедить меня, что он поможет моей команде?

Айвар: Все зависит от того, что вы собой представляете.

Если вы ничего не знаете о программировании, то довольно просто объяснить, что вам нужен графический язык, потому что писание кода – это не лучшее занятие для людей. Код хорош для машин, а не для работы с ним людей.

Если вы опытный программист, я бы поинтересовался, как вы описываете свою систему, ее компоненты, их взаимодействие. Как вы описываете конкретный сценарий с точки зрения пользователя? Будет ли он реализовываться через взаимодействие между вашими компонентами или вашими объектами? Нет языков программирования, способных делать это разумным образом, так что это пример того, где можно использовать UML. Подобных примеров множество.

Есть люди, которых мне никогда не удастся убедить, потому что они работают с кодом очень долго. Но если спросить, каковы их ощущения при работе с совершенно незнакомым языком (например, Пролог), или с новым классом языков (например, декларативные языки), или функциональными языками (например, Scheme или Лисп), возможно, они почувствуют, что графический язык им сильно помог бы.

У меня не возникало реальных проблем с тем, чтобы убедить использовать UML тех, кто понимает требования создаваемой ими системы.

Знания

В какой мере понимание программных технологий связано с конкретными языками программирования?

Айвар: В очень незначительной. В университетах преподают языки программирования, поэтому людям кажется, что язык – это главное. Проблема в том, чтобы понимать программирование в целом. Как регистрировать требования? Как определить, ту ли систему вы строите? Как проверить, что вы построили нужную систему? Как осуществлять управление конфигурацией и версиями? Как применять те 30 или 40 практик, которым не учат в школе?

В школе учат простым вещам. На то она и школа. Языки программирования относительно легко преподавать и изучать. Во время моего пребывания в MIT я выбрал курс 6001, где мы использовали Scheme – разновидность Лиспа – для описания некоторых явлений в области компьютерных наук. Этот курс выбирали те, кто только что окончил школу, они писали код в продолжение этого курса, и это был один из самых фантастических курсов, которые я прослушал. С помощью языка они описывали такие явления, как компиляция, выполнение, интерпретация и многие интересные явления из области компьютерных науки. Они также изучали основные идеи программирования, и программировать становилось действительно легко.

Сейчас у нас появились среды программирования, но изучить среду гораздо труднее. И все же это относительно просто: это лишь одна из нескольких вещей, которые требуется знать, чтобы стать хорошим разработчиком ПО. Мы должны поднимать наш уровень знаний в программной инженерии.

Мы должны найти способ передавать знания тогда, когда они требуются, а не раньше.

Айвар: Да, и нельзя выбрасывать накопленное. Начните с того, что у вас уже есть. У каждого, кто сегодня пишет программы, есть какие-то практики, которые могут быть не столь хороши, но все же полезны. Нужно не пытаться изменить все сразу, а совершенствовать то, что больше всего в этом нуждается.

Может быть, вы хорошо умеете писать код и управлять конфигурацией, но плохо умеете составлять технические требования и проводить тестирование. Есть практики для этих вещей. Можно продолжать делать то, что вы делаете сегодня, изменяя то, что необходимо, и не отказываясь от всего имеющегося ради включения нового. Это естественный процесс развития.

Я читал, что вы предвидите, как в будущем умные агенты будут нашими партнерами в парном программировани. Каким образом?

Айвар: Разрабатывать ПО не бог весть как сложно. Взгляните на те 5–10 миллионов человек, которые считают себя разработчиками ПО. Очень немногие из них действительно могут творить и предлагать не-

что совершенно новое. К несчастью, окружающие считают программистов творческими и талантливыми людьми, а это далеко не так.

Есть научные данные, которые говорят, что сегодня 80% труда программистов в течение рабочего дня – разные обычные и совсем мелкие действия – не являются умственной работой. Они выполняют то, что делали уже 50, 100, 1000 раз прежде. Они действуют по шаблону в новых условиях.

Конечно, творческая работа существует, но большинство ею не занимается. Умственная работа составляет 20%. Даже она оказывается не слишком сложной: просто приходится подумать иначе, чем привыкли думать раньше.

80% работы ведется по правилам. В некотором конкретном контексте можно применять один шаблон за другим, создавая ПО. Шаблоны не всегда заданы, поэтому можно выбрать неверный шаблон и сделать плохую программу. Шаблоны могут быть разными, поэтому одни программы получаются хорошими, а другие – плохими.

Можно описать эти правила и применять их с помощью инструментов. В этом состоит идея умных агентов. Умные агенты знают контекст и что в нем нужно делать, и они это делают. Они могут многое делать сами, если нужно применять простые правила, или могут спросить совета у разработчика, с которым работают.

Основанная мной компания Ivar Jacobson International разработала умных агентов для создания ПО и достигла поразительных результатов. Tata Consulting Services с помощью довольно небольшого набора правил сократила затраты на 20%. Выросло качество и сократилось время обучения программистов и разработчиков. Новые сотрудники быстро начинали работать в полную силу.

Я не сомневаюсь в действенности этой технологии. Проблема в существовании большого числа разных платформ и инструментов, которыми хотят пользоваться. Если вы хотите разработать такое ПО, вам придется приспособить его к различным инструментам и разным видам платформ, поэтому маленькой фирме трудно разработать свои агенты. Другое дело в масштабах крупной компании, такой как TCS.

Потенциально, с помощью технологии умных агентов можно сократить издержки на 80%. У нас, например, умные агенты применяются для описания сценариев использования, разработки сценариев использования, тестирования сценариев использования и так далее. И это только начало. Не сомневаюсь: здесь есть технология, есть задача, есть деньги.

Сможет ли когда-нибудь любой человек интерактивно взаимодействовать с компьютером, указывая ему, что делать, или между программистом и пользователем всегда будет большая разница?

Айвар: Думаю, все большую часть работы будет выполнять сообщество пользователей, а не программисты. Один из путей к этому – программирование на основе правил. Оно позволяет не вникать в то, как происходит выполнение: вы просто пишете свои правила. Затем их интерпретирует некий движок. Этот подход не содержит принципиальной новизны, поскольку проповедуется сообществом ИИ уже 40 лет. Объектная технология позволила нам лучше понять, как строить средства моделирования. Лет 20 или 30 назад системы на базе правил были монолитными и очень негибкими. С агентами мы получаем некую объектно-ориентированную экспертную систему, которую гораздо легче модифицировать.

Как вы распознаете простоту?

Айвар: Простота – это главная составляющая того, чтобы быть умным, делать что-то с умом, да и вообще ума. Эйнштейн сказал что-то вроде «все должно быть как можно более простым, но не проще того». Полностью согласен. Это я и называю умом.

Если ты умен, то делаешь вещь как можно более простой, но не проще того. Все, что делается, должно делаться с умом. Если это архитектура, нужно моделировать как можно меньше, но столько, сколько необходимо. Если не моделировать, потратишь массу сил на то, чтобы описать, что ты хочешь сделать, и не получишь необходимого общего представления.

Например, составлять заранее технические требования и пытаться определить все требования, еще ничего не реализовав, не умно. Умным будет определить основные сценарии использования или главные функции и начать их реализовывать, чтобы получить какие-то ответные данные. Я нашел 10–15 таких случаев умных действий.

Мы должны начать вести себя с умом, разрабатывая ПО. «Умное» программирование – это развитие гибкого. Методология Agile в основном связана с социальной психологией, хотя к ней добавили еще многое. Не нужно быть умным, чтобы быть гибким, но чтобы быть умным, нужно быть гибким. В моей новой лекции говорится о том, как стать умным.

Быть готовым к переменам

У вас диплом MIT бакалавра физики, диплом Калтеха магистра астрономии и диплом MIT доктора компьютерных наук. Как университетское образование влияет на ваши представления о проектировании ПО?

Джеймс Рамбо: Думаю, мое многогранное образование усиливает проницательность и синергизм по сравнению с обычной подготовкой в об-

ласти компьютерных наук. В физике понятие симметрии является фундаментальным – оно буквально составляет сердцевину современной физики. Я пытался применить это понятие к моделированию. Например, ассоциации создают более симметричную точку зрения на ситуацию, чем традиционный подход на базе указателей, принятый в большинстве языков программирования. Изучая компьютерные науки в MIT, я участвовал в группе вычислительных структур профессора Джека Денниса – одной из первых групп, которые исследовали фундаментальные модели вычислений. Эта закваска идей вместе с интеллектуальной строгостью была вдохновляющей средой, до сих пор оказывающей влияние на ход моих мыслей.

Какие темы следует глубже изучать студентам?

Джеймс: Я не очень хорошо знаком с современными учебными программами, но, по моему впечатлению, во многих колледжах принят узкий взгляд на компьютерные науки с акцентом на конкретные языки программирования и системы, а не на понимание важных базовых компьютерных принципов. Например, я редко встречал программиста, который понимает принципы вычислительной сложности и применяет их на практике. Зато они суетятся по поводу всяких бессмысленных мелких оптимизаций, которым грош цена.

Думаю, главный талант в компьютерной области (а также в физике и других творческих сферах) – способность к абстрагированию. К сожалению, мой опыт показывает, что менее 50% программистов в достаточной мере способны к абстракции. Один из моих коллег полагает, что в действительности их меньше 10%. Возможно, он прав. К сожалению, многие из тех, кто занят в программной индустрии, не располагают основными навыками, необходимыми для этой работы.

Какой формат больше всего подходит для передачи знаний в области ПО? Сомневаюсь, что хоть кто-нибудь читает тысячестраничные руководства.

Джеймс: Если необходимо постоянно иметь под рукой справочник объемом в тысячу страниц, значит, с вашей системой что-то не так. Она плохо разбита на части. К несчастью, для многих в нашей отрасли сложность стала кумиром. IBM просто преклоняется перед сложностью. Разумеется, это помогает продавать консультативные услуги.

Инженеры во время своего обучения осваивают разнообразные искусства, сначала на занятиях в университете, затем во время производственной практики в реальных проектах. Главное – усвоить общие принципы. В инженерном деле это законы физики и технические основы конкретных дисциплин. В компьютерной области им соответствуют такие научные принципы, как алгоритмы, структуры данных и теория

сложности, а также принципы программной инженерии. В любой области важно достичь интуитивного понимания хода вещей. Если программные приложения следуют принятым нормам и спроектированы последовательно, опытному разработчику часто не нужно рыться в грудах документации, чтобы предугадать организацию и поведение новой системы.

Необходимо также предоставить руководство по эксплуатации системы. Недостаточно просто перечислить составные части в расчете, что каждый сам сообразит, как они взаимодействуют между собой. Если вы изучаете такое сложное приложение, как Photoshop, лучше всего начать с учебника, в котором рассказывается, как с помощью комбинаций основных команд решить типичные задачи. Детали всегда можно выяснить с помощью полного списка команд, но начинать с него изучение системы не стоит. Тем не менее многие разработчики систем считают, что выполнили свою работу, если предоставили полный список команд или процедур, образующих систему. Он бесполезен для понимания того, как работает система. Поэтому главным недостатком передачи системных знаний является чрезмерное внимание к статичным данным декомпозиции, а не к образцам применения. Движение за использование шаблонов (паттернов) очень правильно сосредоточилось на применении, хотя иногда они рассматривали шаблоны слишком узко.

Как вы находите нужного человека на роль архитектора программного проекта?

Джеймс: Здесь приходится балансировать. Хорошие архитекторы должны уметь находить правильное соотношение теории и практики, красоты и эффективности, опыта и дальновидности. Задача архитектора в том, чтобы правильно определить общую структуру системы, принять решения, важные для системы в целом. Сюда входят разбиение на модули, основные структуры данных, механизмы связи и оптимизируемые цели. Архитектор, чрезмерно увлеченный деталями кода, может упустить из виду общую картину.

Архитектор должен уметь эффективно общаться, чтобы обеспечить совместную работу разработчиков и программистов. Меньше всего хотелось бы иметь в качестве архитектора гения, который не умеет ясно изложить задачи обычным людям. Искушенность в вопросах политики является несомненным достоинством: часть функций архитектора состоит в том, чтобы обеспечить дружную работу соперничающих группировок.

У архитектора должен быть опыт работы над крупными системами. Невозможно все узнать только из университетских курсов и книг: необходимо сначала получить практический опыт и только потом браться за такую важную работу.

Как передавать опыт в области ПО?

Джеймс: Я как-то говорил, что проблема программирования в сравнении с другими творческими сферами в том, что не существует музея программ. Если вы художник, то можете изучать картины великих мастеров прошлого по книгам или оригиналам, висящим в музее. Если вы архитектор, то можете изучать разные виды построек. Программисты же предоставлены самим себе.

Движение за шаблоны создало каталоги полезных приемов, которые можно применять во многих различных случаях. Это хороший способ регистрировать лучшие практики ведущих программистов, чтобы ими могли воспользоваться все.

Однако необходимы также большие примеры, показывающие взаимодействие частей в законченных приложениях. В недавнее время движение open source дало примеры больших программ, доступных для изучения каждому. Но не все в этих системах равноценно, поэтому начинающим разработчикам нужен сопровождающий. Нам нужны аннотированные примеры, чтобы разработчики ПО могли понять, что хорошо, а что плохо в этих системах, – как комментарии к шахматным партиям или разбор компании в бизнес-школе. Это стало бы иллюстрацией хороших практик и указанием на то, как, возможно, не стоит поступать. Как и при освоении любого дела, полезно посмотреть примеры плохих решений, которых следует избегать.

В какой мере знания в области ПО связаны с каким-то конкретным языком программирования?

Джеймс: К сожалению, слишком много умственных усилий связывается с конкретными языками программирования. Большая часть разработки программы может быть проведена независимо от языка программирования. Конечно, язык программирования нельзя игнорировать, и на стратегическом уровне нужно учитывать существенные особенности языка, такие как работа с памятью, параллельные вычисления и так далее. Но основная часть разработки состоит из решения таких проблем, как структуры данных, вычислительная сложность и декомпозиция на отдельные потоки управления, которые выходят за рамки конкретного языка программирования.

Картина такая же, как в естественных языках. Можно набросать новостную заметку, не сильно заботясь о языке. Но если вы пишете стихи, то язык изначально имеет огромное значение. Если вы пишете программы, как стихи, значит, слишком потакаете своим желаниям. Но когда вы садитесь и пишете фактический код, то не переводите свой очерк с другого языка, а используете свое знание языка, чтобы выбирать хорошие выражения.

Всегда ли между программистом и так называемым пользователем будет большая разница — или в будущем каждый сможет указывать компьютеру, что тот должен делать?

Джеймс: Должен заметить, что одни люди могут четко изъясняться на обычном языке, а другие не могут, поэтому даже если можно будет обращаться к компьютеру на обычном человеческом языке, ему будет трудно понять некоторых людей, потому что они не умеют четко думать. Разница между людьми, которые умеют думать и ясно изъясняться, и теми, кто этого не умеет, будет всегда.

Кроме того, некоторые способы выражения мыслей становятся гораздо короче, если ограничить тему. Нотная грамота чрезвычайно компактно записывает музыку, а шахматная нотация отлично подходит именно для этой игры. С помощью чертежа вы скорее получите то здание, которое вам нужно, чем если будете общаться с рабочими исключительно на словах. Поэтому необходимо, чтобы люди мыслили ясно и точно и выражали свои мысли с помощью специализированных языков.

Я не питаю надежд, что в обозримом будущем мы сможем общаться с компьютерами с помощью естественных языков. Вспомните, что еще Кобол должен был дать возможность разговаривать с компьютерами на английском языке! Так что чрезмерный оптимизм по поводу общения с помощью естественных языков возник не вчера.

Какие уроки из опыта изобретения, дальнейшего развития и распространения вашего языка могли бы извлечь разработчики компьютерных систем в настоящем и обозримом будущем?

Джеймс: Во-первых, для успеха нужно немного удачи. Я оказался в нужном месте в нужное время. Мы разработали OMT в качестве одного из первых ОО-методов, и нам удалось написать об этом достаточно доходчивую книгу. Позднее появились не худшие методы, но было уже поздно. Мне также повезло, что я работал в исследовательском центре GE в тот период, когда GE не занималась серьезно программированием. Не знаю, почему нам позволили работать над ним столь долго, но мы могли работать так, что нам не пришлось расхваливать какую-то группу продуктов компании, благодаря чему доверие к нам было выше, чем к большинству других методов.

Опыт моей работы в Rational Software более смешанный. Сведя вместе изобретателей трех ведущих методов ООП, мы смогли придумать UML и добиться его широкого распространения. Дело не в том, что UML был намного лучше других существовавших методов (хотя ему удалось совладать с некоторыми имевшимися у них проблемами), а в том, что он позволил большинству людей понимать друг друга, а не обсуждать достоинства разных значков и другие загадочные различия. К сожале-

нию, Rational не сумела достаточно быстро выпустить эффективные и простые в работе инструменты и достичь такого же успеха, как в методологии. Мне кажется, что высшее руководство и большинство разработчиков не очень верили в моделирование – они все еще предпочитали «героическое программирование», – а это сказалось на инструментах. Стану ли я покупать инструмент у того, кто сам его не использует? Когда отношение изменилось, было уже поздно. Вот другой урок: если не верить в то, что делаешь, ничего хорошего не получится.

OMG (Object Management Group) – это конкретный пример того, что вмешательство политики может погубить любую хорошую идею. Первая версия UML была достаточно простой, потому что некогда было добавлять в нее лишний хлам. Главным ее недостатком было отсутствие последовательного взгляда: одни вещи были достаточно отвлеченными, а другие тесно связывались с конкретными языками программирования. Это должна была исправить вторая версия. К несчастью, многие из тех, кто завидовал нашему начальному успеху, встряли в эту вторую версию.

Им казалось, что они могли выступить не хуже нас. (Как впоследствии выяснилось, не могли.) Процесс OMG разрешил разнообразным группам особых интересов делать добавления в UML 2.0, а поскольку процесс обычно основан на консенсусе, то отвергнуть плохие идеи было практически невозможно. В результате UML 2.0 чудовищно распух, очень большая его часть вызывает сомнения, и по-прежнему нет последовательной точки зрения и путей для ее выработки. Своего рода дурной законопроект об ассигнованиях, в который вставлено все, что только можно. Пример порочности попыток выполнения творческой работы силами комиссии.

В целом, процесс иллюстрирует Закон Брукса для второй системы. Если вы никогда не читали «Мифический человекомесяц» Брукса, сегодня же достаньте эту книгу и прочтите ее. Вне всяких сомнений, это лучшая из всех написанных о разработке ПО книг. Самое печальное, что большая часть проблем, описываемых автором в книге, изданной 30 лет назад, сохранилась по сию пору. По-прежнему менеджеры пытаются спасти затянувшиеся проекты путем привлечения дополнительных работников, что приводит к еще большему запаздыванию, – все, как описывал Брукс.[1]

Возможно, будет правильным так описать проблему, вставшую перед компьютерной отраслью: большинство ее участников не знает истории, а потому, как сказал Тойнби о всемирной истории, обречено на повторение старых ошибок. В отличие от ученых и инженеров, которые опи-

[1] Ф. Брукс «Мистический человеко-месяц, или Как создаются программные системы». – Пер. с англ. – СПб.: Символ-Плюс, 2000.

раются на прошлые открытия, очень многие практики компьютерной отрасли воспринимают каждую машину или язык как нечто новое, не ведая, что похожие вещи происходили ранее. На первой конференции OOPSLA в 1986 году гвоздем программы было представление системы Sketchpad Айвена Сазерленда (Ivan Sutherland), придуманной в 1963 году. Она включала в себя некоторые идеи ООП задолго до того, как было изобретено ООП, и реализовывала их лучше, чем большинство современных систем ООП. Она выглядела свежо в 1986 году, как и 20 лет спустя. Так почему же многие современные графические инструменты слабее, чем Sketchpad?

Почему мы до сих пор имеем в операционных системах ошибки переполнения буфера, служащие плодородной почвой для различных вредоносных программ? Почему мы до сих пор пользуемся такими языками, как Си и C++, которые не поддерживают концепцию массивов с границами и потому способствуют ошибкам переполнения буфера? Конечно, можно определить в C++ массивы с границами, но программисты продолжают злоупотреблять указателями. Все это невежество, лень и самонадеянность разработчиков. Компьютерная область сложна, и избежать логических ошибок в сложных системах невозможно, но столько лет повторять простейшие ошибки неоправданно.

Так какие же главные вопросы нужно выяснить при разработке новой системы? Прежде всего, определите, для чего она предназначена и кого будет обслуживать. Не замахивайтесь сразу на многое: лучше быстро изготовить что-то полезное и потом расширять его, чем пытаться заранее предвидеть все возможные потребности. Это правильный принцип «гибкой разработки». Нельзя удовлетворить всех желающих, поэтому старайтесь принимать жесткие решения, но не забывайте, что если система окажется успешной, то она станет развиваться непредсказуемым образом, так что будьте готовы к внесению неожиданных модификаций.

Применение UML

Что вы думаете о генерации кода реализации из UML?

Джеймс: Думаю, это ужасная идея. Знаю, что многие другие специалисты по UML думают иначе, но в UML нет никаких чудес. Если можно генерировать код по модели, значит, это язык программирования. А UML не является правильно спроектированным языком программирования.

Хуже всего, что в нем отсутствует ясно очерченная точка зрения – отчасти намеренно, отчасти из-за тирании процесса стандартизации OMG, стремящегося дать всем то, чего они хотят. Нет четких базовых соглашений относительно памяти, хранения, параллелизма и почти всего остального. Ну как можно программировать на таком языке?

Дело в том, что UML и прочие языки моделирования не предназначены для исполнения. Смысл моделей – в их неточности и неоднозначности. Это сводило с ума многих теоретиков, пытавшихся сделать UML «точным», но модели не должны быть точными: мы отбрасываем то, что малозначительно, чтобы сосредоточиться на тех вещах, которые имеют важное или глобальное значение. То же происходит с физическими моделями: вы моделируете значительные влияния (такие как сила притяжения Солнца), а потом рассматриваете более слабые влияния как возмущения в основной модели (например, влияние планет друг на друга). Если вы попытаетесь непосредственно решать полный набор уравнений со всеми подробностями, у вас ничего не получится.

Я считаю, что в последнее время работа над UML часто велась в неверном направлении. Его никогда не собирались делать языком программирования. Его нужно применять для выработки правильной стратегии, а саму программу писать на подходящем для нее языке программирования.

К сожалению, мне не известны действительно хорошие языки программирования. В каждом из них есть масса изъянов, способствующих возникновению ошибок. Все семейство языков Си (Си, C++, Java и т. д.) жестоко страдает от недостатков (плохо поддающийся разбору синтаксис), но мы вынуждены работать с ними, нравится нам это или нет. Многие новомодные языки демонстрируют, что их разработчики не обладают глубокими знаниями теории языков. Напротив, многие более академические языки, слишком стремясь к красоте, пренебрегают важными характеристиками, такими как предоставление возможности заниматься одной и той же системой нескольким командам разработчиков.

Какими свойствами должен обладать язык, чтобы его могли использовать несколько команд разработчиков?

Джеймс: Хочу вспомнить Алгол-60, старый язык программирования, один из первых, с которыми я работал (возможно, еще до рождения многих читателей этого текста). Он ввел много важных понятий, таких как синтаксическая нотация БНФ, рекурсивные подпрограммы и управляющие структуры. Во многих отношениях это был более ясный язык, чем Фортран. Но у него было четыре главных недостатка, из-за которых он оказался непригодным для работы: у него не было встроенных конструкций для ввода/вывода; он не поддерживал числа с двойной точностью и матричные операции; он не поддерживал раздельную компиляцию подпрограмм; в нем отсутствовали стандартные интерфейсы к подпрограммам на машинном языке и Фортране. Это довольно мелкие проблемы с точки зрения теории, но большие препятствия для создания ПО.

Многие академические языки совершают одну и ту же ошибку: решая интересные математические задачи, они не уделяют внимание практическим аспектам использования языка, потому что практические проблемы неинтересны с точки зрения теории. А эти мелочи и определяют пригодность языка для работы.

Прежде всего, разработчикам нужна возможность работать над отдельными частями системы в отсутствие объявлений или остального кода системы. Затем им нужна возможность соединить эти части и убедиться, что они работают как система; думаю, для этого требуется какой-то способ объявления типов. Нужно снабдить язык разными механизмами обмена данными, потому что системы сейчас становятся высокопараллельными. В большинстве языков нет удобных способов описания динамического поведения. Думаю, нужны средства отладки, которые лучше интегрированы с языком, но могут быть включены или выключены во время выполнения. Сейчас написание кода и тестирование связаны слишком слабо.

Не является ли UML просто инструментом координации работы в больших командах разработчиков?

Джеймс: Это прежде всего инструмент для направления и организации мышления отдельных разработчиков. Вы должны работать на разных уровнях абстракции; код – это один из таких уровней, но он не очень удобен для понимания работы системы. Нужно подняться на более высокий уровень, а для этого избавиться от мелких деталей кода и рассматривать то, что важно на этом более высоком уровне. Вот почему будет ошибкой пытаться сделать UML исполняемым: это разрушит весь смысл абстракции.

Говоря об архитекторах, вы подчеркиваете важность налаженного общения. UML помогает решить эту проблему?

Джеймс: Он предоставляет стандартный набор понятий и нотацию. Это помогает общению. Невозможно общаться без общего словаря, иначе вы будете думать, что общаетесь между собой, а на самом деле люди называют одними и теми же словами совершенно разные вещи, что еще хуже, чем полное отсутствие общения.

Использование UML помогает архитекторам общаться?

Джеймс: Да в этом же весь смысл!

Когда я только начинал проталкивать идеи ООП в GE, я побывал в подразделении GE, занимавшемся разработкой авиационных двигателей. Убедить программистов в плодотворности объектного подхода оказалось очень трудно: они привыкли к старым методам (например, программированию на Фортране) и не понимали, о чем им говорят. Однако

несколько инженеров все прекрасно поняли и ухватились за идею ООП. Они привыкли создавать модели, оперируя понятиями высокого уровня абстракции, такими как «график работы двигателя» или «скорость сваливания и угол атаки». Они умели создавать идеальные объекты для представления физических понятий. Программисты не видели за деревьями леса: они погрузились в код и не понимали, что код служит представлению понятий более высокого уровня. Сегодня многие также не понимают этого.

Робин Милнер, создатель ML, проиллюстрировал идею иерархии моделей, связав вместе все — от языков проектирования верхнего уровня, таких как UML, до низкоуровневого кода ассемблера, физической модели аппаратного обеспечения и дополнительных моделей, представляющих среду, в которой работает аппаратура. Он привел пример самолета, в котором код от верхнего уровня до аппаратного, аппаратура со своими моделями и сам самолет в целом спроектированы в соответствии с моделями аэродинамики, физики и метеорологии! Когда пилот нажимает кнопку, все эти модели включаются в работу.

Не следует ли нам сократить количество уровней (до одной универсальной модели/языка) или увеличить абстракцию (и количество уровней)?

Джеймс: Отличная мысль. В физике (или, скорее, науке в целом) есть важная идея нескольких уровней эмергентных представлений. Каждое представление создается поверх того, что лежит ниже, но, будучи созданным, становится последовательным и самостоятельным. Можно, например, назвать такие уровни, как квантовая физика, химия, микробиология, биологические организмы, популяции, экосистемы, окружающая среда. Другую иерархию образуют вычисления: физика веществ, полупроводники, электронные схемы, цифровые системы, компьютеры, микропрограммы, операционные системы, прикладные среды, приложения, сети. Ни один из уровней не является «правильным», или «истинным», или «основным». Каждый из них по-своему содержателен и может быть определен в терминах лежащего ниже уровня. Но это не значит, что его можно понять на более низком уровне. Смысл каждого уровня особый в своем роде и может быть понят только на этом уровне. Это эмергентная система: смысл каждого уровня логически выводится из более простого нижнего уровня, но понят может быть только в собственных рамках. Поэтому, чтобы понять любую сложную систему (в предельном случае — вселенную), мы должны действовать одновременно на нескольких уровнях, ни один из которых нельзя считать главным.

Иерархия эмергентных уровней — это то, с чем языки моделирования справляются плохо. Нам нужны средства моделирования систем на не-

скольких уровнях одновременно. Речь не идет о 4-уровневой метамодели OMG. Она появилась в результате той же ошибки, которую сделал Бертран Рассел, решивший, что нельзя что-либо моделировать посредством самого себя. Можно, конечно, читайте книги Дугласа Хофштадтера, например «I Am a Strange Loop» (Я странная петля), изд. Basic Books. В этом ошибка и программистов, пренебрегающих моделированием. Они думают, что только код имеет значение. Это все равно что говорить «только микросхемы имеют значение» или «только физика полупроводников имеет значение». Все уровни важны, и для каждой конкретной цели нужно выбирать правильный уровень. Я бы предположил, что уровень кода плохо подходит для того, чтобы понять, как большая и сложная система выполняет какие-то полезные функции.

Единого универсального языка не получится. Нам нужна среда, которая позволит работать на нескольких уровнях абстракции. Это должен был делать UML 2.0, но не получилось. Дело не в ошибочности добавленных деталей, а в создаваемом ими усложнении: соотношение затраты/прибыль получилось плохим. Главное – отсутствие понятного способа построения уровней связанных моделей с соблюдением раздельности этих уровней.

Например, в UML есть понятия достаточно низкого уровня, более подходящие для языков программирования, такие как права доступа и указатели, а также понятия высокого уровня, но нет удовлетворительного способа отделить понятия низкого уровня от понятий высокого уровня. Была попытка сделать это с помощью профилей, но они не вполне справляются с этой задачей. Значительная часть битв, происходивших при создании UML, а также свойственная ему шизофрения обусловлены этим конфликтом между понятиями, относящимися к языкам программирования, и логическими понятиями высокого уровня.

Другая существенная проблема – разница в стиле между частями, выполненными разными людьми. Например, очень полезна добавленная в UML диаграмма последовательности сообщений, но она сильно отличается по стилю от диаграммы активности.

Какой процесс вы выберете, чтобы упростить язык?

Джеймс: Сомневаюсь, что его можно упростить с помощью стандартного процесса, такого как процесс OMG, приведший к выходу UML 2.0. Слишком противоречивы желания тех, кто хочет приложить к этому руку. Проблема в том, что процессы стандартизации слишком мало внимания уделяют последовательности, простоте и единообразию стиля, зато перегружают лишним контентом. На самом деле, я не очень доверяю комитетам по стандартизации: они часто создают раздутые продукты, некрасивые и малоудобные. Я неохотно принял участие в их работе, и мои опасения подтвердились.

Лучше всего, если один или несколько человек сделают свои сокращенные версии UML, и практика покажет, какая из них лучше. Не обязательно называть результат «UML», потому что этот термин может нести определенную юридическую или эмоциональную нагрузку. Важно, чтобы разработчики четко сформулировали назначение своей версии, не пытаясь угодить всем.

Как вы распознаете простоту?

Джеймс: Нужна особая готовность, чтобы делать меньше вместо того, чтобы делать больше. Не забывайте, что язык моделирования – не язык программирования: если в языке отсутствует какая-то возможность, разработчик модели всегда может чем-то заполнить этот пробел. Если вам нужно таскать с собой список функций, потому что иначе их не запомнить, это не простой язык. Если вам постоянно предоставляется четыре или пять вариантов моделирования простой ситуации, это не простой язык.

Допустим, я скептически отношусь к UML. Как бы вы попробовали убедить меня в его пользе?

Джеймс: Я и сам весьма скептически к нему отношусь. Думаю, он чрезмерно раздут из-за того, что на кухне OMG оказалось слишком много поваров. Кроме того, его пытались предлагать как универсальное средство на все случаи жизни. В компьютерной области вообще принято безудержно восхвалять каждую новинку. Есть также склонность к поиску универсального решения для всех задач. Жизнь и компьютеризация – слишком сложные явления, чтобы для них существовали простые решения.

UML – очень полезный инструмент для проектирования структур данных, довольно полезный при декомпозиции систем на многоуровневые модули и не слишком полезный для динамических задач, с которыми справляется не очень успешно. Он полезен, но не решает все ваши проблемы. Помимо него вам потребуются многие другие навыки и инструменты.

Есть ли минимальный объем программного проекта, ниже которого применение UML увеличивает сложность и трудозатраты, не принося выгоды?

Джеймс: Нет, но это не значит, что в очень маленьких и очень больших проектах его нужно применять одинаковым образом. В маленьком проекте вы меньше соблюдаете «формальности», применяя инструменты, модели, программные процессы и так далее. В маленьком проекте диаграммы классов и структур данных тоже полезны, просто перемещаться от реализации к проекту и обратно приходится реже. UML позволя-

ет сделать набросок первоначального проекта, но потом вы переходите к программированию и остаетесь в нем.

В большом проекте процесс разработки не менее чем на половину состоит в обмене информацией, а не просто в фиксировании проекта. В этом случае существенную роль играет наличие инструментов и процессов для декомпозиции системы, контроля доступа к моделям и коду, слежения за ходом проекта, иначе люди будут постоянно наступать друг другу на пятки. Я знаю, что многие программисты жалуются на необходимость подчиняться такого рода дисциплине. В спорте, в строительстве, в издании газет, проектировании ракет и почти в любой другой крупной совместной деятельности от таких недовольных избавляются без особых сожалений. Пора и нам занять такую позицию в отношении к программированию, если мы серьезно занимаемся им.

Уровни и языки

Выше вы сказали, что движение за шаблоны многого достигло, но оно слишком узко рассматривало шаблоны. Нельзя ли об этом поподробнее?

Джеймс: Однажды на семинаре, где присутствовал кое-кто из Hillside Group, я выяснил, что их взгляды на шаблоны проектирования были действительно очень узкими, едва ли не религиозными, и они нисколько не старались расширить их. Они очень строго держались своего официального взгляда на шаблоны и боготворили архитектора Александера. Они рассматривали шаблоны очень узко, тогда как я считаю, что паттерны можно применять на многих различных уровнях.

Не все согласны с той точкой зрения на шаблоны в диапазоне от маленьких до средних, которая отражена в книге «Паттерны проектирования» «банды четырех»[1] и материалах Hillside Group. Фактически даже внутри движения за паттерны были свои фракции, но идея, как мне кажется, прижилась, а это главное.

Создатели шаблонов повторяют то, что уже было сказано для многих других дисциплин: нужно собрать опыт лучших профессионалов и каталогизировать его, чтобы им могли пользоваться обычные люди. Это сделано и в машиностроении, и в живописи, и в строительстве – почти во всех творческих областях. В компьютерной области была и есть опре-

[1] Под этим названием известны Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес – авторы книги «Design Patterns: Elements of Reusable Object-Oriented Software» изд. Addison-Wesley Professional. («Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – Питер, 2003.)

деленная задержка с пониманием необходимости учиться на прошлом опыте. В этом и одна из причин моего недовольства. В компьютерной области много людей с очень высоким мнением о себе, забывших о прошлом опыте, который можно использовать. Многие пытаются заново изобретать то, что уже давно открыто.

По-моему, сторонники шаблонов пытаются бороться с этой проблемой, рассуждая так: «Мы можем зарегистрировать, понять и сделать доступными широким массам работников нашей отрасли некоторые вещи, до которых сами они не додумались бы, но смогут пользоваться ими, если их хорошо описать».

Это верно в любой области. В каждой сфере не так много революционеров-первооткрывателей. После начального прорыва идею подхватывают и развивают другие. Говорят, что после того как идея обнародована, ее автор теряет над ней контроль. Не может какая-то узкая группа управлять тем, как надо понимать идею. Другие люди найдут в ней дополнительный смысл, который не предполагался ее авторами.

Хочу провести параллель с тем, что вы уже сказали о физике, где есть много разных уровней. То, что применимо и свойственно одному уровню, необязательно свойственно другому, но при этом нельзя отрицать реальное существование других уровней.

Джеймс: В этом вся суть эмергентных систем. В этом суть науки и языка. Это одна из концепций теории сложности. Вы наблюдаете эмергентность, и при этом нет никакого основного уровня. Разумеется, в компьютерной области это уже давно поняли. Есть много уровней, и ни один из них не является «истинным».

Говоря, что движение за шаблоны ограничило себя очень узкими целями, вы имеете в виду, что оно выбрало себе один уровень, вместо того чтобы взять всю их совокупность?

Джеймс: Будем справедливы к «паттернистам»: среди них есть люди, работавшие в разных уровнях. Например, есть несколько книг по шаблонам в архитектуре. Возможно, вначале у Hillside Group был особый взгляд на то, что представляют собой паттерны, отразившийся на всей идее языка шаблонов Александера. Мне кажется, это слово должно применяться в более широком смысле.

«Шаблоны проектирования» часто критикуют за то, что значительная часть этих паттернов малоэффективна в других языках, кроме C++ и Java.

Джеймс: Одна из особенностей шаблонов состоит в том, что они должны конкретно соответствовать тому, над чем вы работаете. Нельзя собрать все вместе и на основе этого сделать самый общий метод. Если вы пишете шаблоны для языков программирования, многие из них будут

для конкретного языка, а некоторые окажутся общими и сгодятся для большой группы языков программирования. То же самое в технике: что-то подходит для стали, но не для дерева, и наоборот, а что-то и для того, и для другого.

Возможно, это и не удовлетворяло меня в моделировании. UML плохо различал особенности конкретного применения. В одних случаях вы моделируете применительно к конкретным языкам программирования, а в других – применительно к логическим объектам. UML допускал оба применения, сваливая все в одну кучу. Ответственность за это отчасти на мне. В первой версии мы смешали много разного. В первых версиях такое случается – недостаточно опыта для выделения частей. Я рассчитывал, что во второй версии мы исправим положение, и можно будет сказать, что такие-то функции применимы к Си или C++, потому что есть функции, несущие явный отпечаток Си или C++ и аналогичных языков. Нет ничего плохого в их присутствии, но они не такие общие, как некоторые другие функции, применимые к широкому спектру языков, и полезно отличать одни от других.

Есть функции моделирования для верхнего уровня и для нижнего уровня. Предполагалось, что механизм профилей позволит определять функции для конкретной точки зрения, но, к сожалению, он несколько грубоват. Он не дает возможность правильно разделять уровни. Он говорит: вот прикладная область, – но в каком-то смысле это одноуровневая штука. Нет способа четко организовать уровни моделирования.

Допустим, вы смогли бы создать UML 3.0, не обеспечив обратную совместимость. Как бы вам удалось это сделать, избежав всеобщего возмущения?

Джеймс: Вы исходите из того, что всеобщее возмущение неизбежно. Надо сказать, что Microsoft и Apple постоянно сталкиваются с этой проблемой. Можно поддерживать совместимость на протяжении ряда поколений, как это делает Microsoft, а можно периодически нарушать совместимость, как делает Apple. Преимущества и недостатки есть у обеих стратегий.

Невозможно вечно сохранять совместимость. Это нереально. В любой области вам в конце концов придется сказать «извините, мы этим больше не занимаемся» и «извините, вам придется купить что-то поновее». Взгляните на аналоговые телевизоры. В США они перестали принимать сигнал с антенны с июня 2009 года. Многие узнали об этом только после того, как это произошло. Многие из тех, кто читает такие тексты, знают о подобных проблемах, но отказ от совместимости никогда не бывает легким и безболезненным. Та же проблема с унаследованными системами.

Люди хотят какого-то волшебного решения проблемы устаревшего ПО, но простого решения быть не может. Это крупная и неприятная проблема. Придется засучивать рукава и работать. Здесь нет ничего уникального. Вы пытаетесь делать две несовместимые вещи, а это всегда мучительно и трудно и требует принятия решений о том, в какой момент придется огорчить своих клиентов.

Что вы предпочитаете — крупные изменения изредка или понемногу неприятностей, но часто?

Джеймс: Создавая заново любую систему, вы всегда обнаружите, что сделали ошибки. Можно какое-то время обходиться локальными исправлениями, но в конечном счете обнаруживается, что некоторые ваши исходные предположения и архитектурные решения более не приемлемы, и нужно вносить крупные изменения и все полностью перерабатывать, иначе дальнейшее движение будет невозможно. Я называю это «землетрясением». В системе это можно сделать один раз. Два будет много, потому что гораздо труднее. В итоге система окостеневает так, что внести в нее большие изменения уже невозможно. Думаю, это происходит со многими системами — не только в компьютерных, но и в других различных системах, которые мы видим вокруг, где в какой-то момент все застывает. Потом приходит какой-нибудь Александр со своим мечом и находит новое решение, заменяющее старые методы. То есть старое уже не исправляют. Кто-то предлагает нечто новое, и прежние проблемы становятся неактуальными.

Я считаю, что это вполне относится к компьютерной области. Жизнь трудна, но приходится принимать ее такой, какая есть. Не понимаю, почему это должно расстраивать. Мы не продолжаем жить вечно с одними и теми же вещами, всегда кто-нибудь предлагает новые. Рассматривая проблемы транспорта, глядя из XIX века, можно было бы предположить, что лошадей в скором времени станет так много, что все улицы будут завалены навозом, а самих лошадей будет негде держать. В XX веке было много разговоров о том, что скоро все станут телефонистами на коммутаторах. Так происходит потому, что люди не умеют представить себе нечто за пределами того, чем они заняты сейчас.

Кто-нибудь предлагает новый способ работы, до которого никто прежде не додумался, и прежние проблемы оказываются неактуальными. Думаю, это внушает надежды. Того, кто за свою долгую жизнь привык работать определенным образом и уже не может перестроиться, это обескуражит. Если хотите, последние лет 50 показали, что нельзя выбрать работу и заниматься ею всю жизнь, ничего не меняя, — прошли те времена, когда можно было проработать всю жизнь в одной компании по специальности, полученной в университете. Нужно быть готовым к переменам.

Хайнлайн сказал, что «специализация — удел насекомых».

Джеймс: Насекомые быстро размножаются и умирают в больших количествах. Человеку это не свойственно. Мне не нравится, что многие компании набирают работников по такому же принципу. Им нужны насекомые. Они нанимают специалистов в очень узких областях. Им нужно, чтобы человек сразу стал работать в полную силу. И избавиться от него, когда он сделал свое дело. Думаю, это печальная тенденция. Вряд ли она пойдет нам на пользу в конечном итоге. Нам нужны люди, которые могут думать, меняться и изучать то, что им может потребоваться знать. Не вижу пользы в том, что в учебных заведениях изучают какой-нибудь конкретный язык программирования. Изучать нужно саму идею языка программирования. Выучить новый язык легко. Нам нужны люди, способные меняться. Люди должны следить за тем, что происходит вокруг. Говорят, в наши времена каждый должен непрерывно заниматься самообразованием, и это верно.

Можно ли исправить какие-нибудь из недостатков, которые вы видите в UML?

Джеймс: Конечно, можно. Думаю, вся идея со стандартизацией была, в основном, маркетинговым приемом. Зачем нужно стандартизировать язык моделирования? Стандартизировать нужно то, что действительно исполняется. Думаю, ценность этой идеи стандартизации была чрезмерно завышена. Для моделирования не нужны стандарты. Если UML слишком раздут, люди вырежут из него те части, которые нужны им в работе. Все применяют модели классов, и многие берут такие вещи, как последовательные диаграммы и сценарии использования. Некоторые части UML задействуются редко. Это показывает, как опасно собирать вместе слишком много умных людей, когда возможности принятия решений ограничены. Они предложат кучу идей, которые кажутся полезными, и некому сказать: это прекрасная идея, но она не настолько полезна, чтобы класть ее в эту большую кастрюлю, где готовится кушанье для многих.

Люди будут использовать то, что им нужно. Это не слишком отличается, например, от работы с Photoshop. Я могу работать в Photoshop, но я не эксперт в этом деле. Я не помню, как работать с большей частью его функций, но могу выяснить, если это понадобится, и я умею настраивать параметры изображения, выделять фрагменты и прочее, что мне требуется постоянно. Если мне понадобится сделать что-то еще, я загляну в справочник. Профессиональный графический дизайнер должен уметь гораздо больше. Именно так люди работают с большинством сложных приложений или устройств вроде сотового телефона. Они не знают всех функций, потому что большинство этих функций нужны только для того, чтобы было что написать на коробке с товаром.

Трудно оценивать языки моделирования с точки зрения юзабилити. В отношении языка программирования можно попросить кого-нибудь решить реальную задачу и дать отзыв о том, насколько он хорош в работе.

Джеймс: Если взглянуть на имеющиеся языки программирования, то я сомневаюсь, что при их создании часто исходили из этого. Во многих случаях разработчикам приходят в голову умные мысли, которые они реализуют, не особенно заботясь о юзабилити, что позже порождает множество проблем.

Разработчики языков придумывают всякие мысленные эксперименты с задачами, которые, как им кажется, должен решать их язык. Иногда правильнее было бы сказать: не нужно решать эту задачу, есть другие способы справиться с ней, это не так важно, чтобы включать это в язык.

Включать какие-либо возможности в язык моделирования, в язык программирования или в прикладную систему нужно тогда, когда они достаточно полезны, чтобы пользователь захотел запомнить, как они работают, после чего их нужно протестировать и убедиться, что они действуют на практике. Если вы включите лишние функции, никто не сможет запомнить, как ими пользоваться, и они станут бесполезным грузом. Более вероятными станут проблемы в системе, потому что больше объем тестирования и больше мест, где можно ошибиться. Нельзя спрашивать, полезна эта функция или нет: нужно спрашивать, достаточно ли она полезна в сравнении с остальными, чтобы стоило потрудиться запомнить, как ею пользоваться.

UML мог бы развиваться в сторону использования некоторых его подмножеств, что, несомненно, и происходило постоянно. Сомневаюсь, что процесс OMG может решить какие-то задачи, потому что ему не свойственна решительность. Не думаю, что UML 3.0, сделанный OMG, сможет чего-то достичь, потому что там слишком много конкурирующих интересов. Столько людей старается ввести в язык нечто свое, что сделать его простым не удастся.

Я соглашусь с вашей дурацкой функцией, если вы согласитесь с моей дурацкой функцией.

Джеймс: Именно так. Слишком много компромиссов. Другой путь – если кто-то предложит нечто новое, отчасти основанное на UML, но под новым названием и несколько иными конструктивными решениями. Разумеется, так почти всегда и происходит в итоге. Иначе люди решают, что здесь больше нечего ждать, и переключаются на что-то другое.

Немного о многократном использовании

Складывается впечатление, что в среднем сложность и размер программ растут год от года. Хорошо ли масштабируется ООП в таких условиях или оно скорее осложняет работу?

Джеймс: Прежде всего, неясно, с какой скоростью растет размер систем. Нельзя просто подсчитывать байты в программах. Если вы генерируете код, нужно считать количество строк в исходном коде, а не количество сгенерированных строк, байтов и так далее. Если используются процедуры более высокого уровня, то сложность зависит от количества вызовов, а не от объема исполняемого кода. Работая на более высоких уровнях, мы создаем более крупные системы, но их внутренняя сложность может расти не в такой степени.

Даже если вы с этим не согласны (а я совершенно не согласен – системы реально становятся сложнее), то ОО-системы стоит развивать дальше. Но нужно разделять ОО и задачу повторного использования. Я знаю, что создатели Smalltalk считали многократное использование главной задачей, но мне кажется, что они ошибались. Можно создать ОО-структуру, не преследуя навязчивой цели создать библиотеку многократно используемых объектов, в которой есть все классы, задействованные в приложении. Многократное использование – хорошая вещь, но это не главная цель для большинства систем. Создание хороших библиотек для повторного использования – трудная задача, и большинство программистов не слишком успешно с ней справляются, и не нужно заставлять их это делать. Это отдельная задача, не связанная с созданием системы. Проектируя систему, используйте ОО-структуру, чтобы построить ясное приложение из классов, которые можно при необходимости легко модифицировать. Если обнаружится, что в итоге часто используются варианты некоторого класса, то можно взять на себя труд сделать его действительно многократно используемым.

Нужно также знать, когда остановиться. Я встречал множество новичков, озабоченных тем, чтобы сделать повторно используемые объекты из каждой строчки кода. Если вы можете описать простой алгоритм на естественном языке и написать для него код, не нужно заморачиваться разбиением его на мелкие части: просто напишите код. ООП существует для того, чтобы создавать структуры высокого уровня, когда обстоятельства не так просты, а не для того, чтобы мучиться с мелочами.

Как нам убедиться, что преимущества ООП перевешивают его недостатки?

Джеймс: Как я уже говорил, всегда можно воспользоваться структурой ООП. Проблема в том, чтобы определить, как глубоко можно с ней

опуститься. Истинно верующие всегда стремятся довести ее до самого низа. В программировании есть и другие проблемы помимо структур ООП, например выбор алгоритмов, структур данных, приемлемой вычислительной сложности, легкости понимания и так далее. Не все они связаны с ООП; в действительности, ООП составляет лишь малую часть общей картины. ООП предоставляет удобную среду для структурирования разработок и программ. Это существенно, поскольку иначе задача может вас подавить и ввести в замешательство. Но существенным содержанием любого проекта служит не ООП, а все остальное из перечисленного.

Вы сказали, что многократное использование — не главная цель объектного подхода.

Джеймс: Думаю, оно не должно быть главной целью. Многократное использование чрезмерно превозносится с самых первых дней. Оно было маркетинговой уловкой, соблазнившей многих руководителей. Обеспечить многократное использование чертовски трудно. Для того чтобы делать объекты, которые можно многократно использовать, требуется уровень мастерства гораздо выше обычного. Кто-то сказал — кажется, Брукс или Парнас, — что в три раза сложнее заставить что-либо работать в реальных условиях, а не в лабораторном прототипе, и еще в три раза сложнее обеспечить его многократное использование. Требовать, чтобы вместо одноразового объекта вам сделали многоразовый, значит в большинстве случаев тратить напрасно время и силы.

Тем не менее можно принять некоторые меры, которые облегчат модификацию в будущем. Во первых, не нужно ничего делать очень специфическим образом, если можно сделать это чуть более общим способом. Не нужно загонять себя в угол без необходимости. Если вы видите, как без особых хлопот достичь большей общности, сделайте это. Проектируйте с учетом того, что в будущем потребуется вносить изменения. Это не значит, что все такие обобщения нужно делать в первой же версии. Оставляйте дверь открытой. Оставляйте зацепки для будущих изменений. Вы пишете методы, которые можно будет заменить, но не пишите сразу сверхобобщенные методы, если не уверены, что они вам понадобятся. Проблема в том, что трудно предположить, что вам может потребоваться в будущем, и такие предположения часто оказываются ошибочными. Вы потратите время на обобщение в одном месте, а потом окажется, что изменения нужно делать совсем в другом.

Это напоминает оптимизацию. Я убежден, что программисты слишком часто стремятся к быстродействию не там, где нужно. Все озабочены оптимизацией уже долгие годы. Компьютеры стали гораздо быстрее, но на это не обращают внимания. Люди по-прежнему одержимы «микро-

оптимизацией». Не зная теории сложности, не понимая «порядка величин», люди озабочены мелким, ничтожным увеличением скорости.

Я написал пакет подпрограмм, после чего сделал его профилирование. Не нужно гадать, где требуется оптимизация, – нужно сделать замеры и внести исправления. Одна подпрограмма занимала 30% всего времени. Вот ее я и исправил.

Оптимизацией занимаются сверх всякой меры. В результате больше вероятность отказа, а сама оптимизация может быть сделана не в том месте, где она имела бы значение. Мне кажется, люди совсем не понимают этого. Они говорят: очень важно обеспечить производительность – это же встроенные программы. Ерунда. Все стало работать быстрее. Лишняя оптимизация не окупает ошибок, вызванных ее проведением. Я просто не могу понять, почему люди не делают доступных им простых вещей. Образ мышления многих разработчиков в чем-то глубоко порочен, если такие вещи до сих пор происходят. Думаю, среди них много любителей трудных путей, вроде скалолазов без страховки. Но если бы они лазали по скалам так же, как программируют, их бы уже не было в живых.

Если цель объектно-ориентированного подхода – не многократное использование, то что?

Джеймс: Многократное использование облегчается благодаря объектно-ориентированному подходу, но главной его целью является, по моему мнению, собственно использование, а не многократное использование. При такой структурной организации легче добиться, чтобы все сразу заработало, кроме того, облегчается модификация. Многократное использование тоже может быть в какой-то мере осуществимо, но это лишь дополнительный эффект.

Создавая любое приложение, вы знаете, что вам придется вносить в него изменения, но не знаете где. Организация приложения в объектно-ориентированном стиле облегчает последующие изменения, потому что получающуюся структуру проще модифицировать. Смысл в том, чтобы создать модифицируемый продукт, но не обязательно библиотеку для многократного использования. Вы же не пишете приложение в расчете на то, что все остальные сотрудники вашей компании станут использовать все классы вашего приложения. Кому-то другому в вашей компании, а может быть, и вам самому, придется в конце концов модифицировать программу, которую вы написали. Это неизбежно. Объектно-ориентированный подход облегчает будущие модификации. Думаю, в этом его главная ценность. Для первой версии программы с равным успехом можно выбрать любой подход. ОО оправдывает себя на второй версии.

Благодаря инкапсуляции?

Джеймс: Да, потому что у вас более четкая конструкция, в которой меньше путаницы. При такой организации удается избегать слишком запутанной функциональности. Очень сложная функциональность – реальная проблема, с которой трудно бороться. В известном смысле, вы повторно используете ее в новой редакции своего приложения. Можете называть это многократным использованием, если хотите. Лишь малая часть повторно применяется в других проектах.

Прежде чем вы начнете широко использовать что-нибудь повторно, нужно обязательно трижды опробовать его. Первый раз – особый случай, второй может быть совпадением, после третьего вы начинаете замечать какие-то повторения. Тогда, может быть, стоит взять на себя труд разобрать целое на части и сделать их по-настоящему надежными. Не нужно делать многократно используемым все. Найдите то, что полезнее всего, и его используйте многократно.

Это напоминает мне SOA, где, похоже, исходят из того, что можно определить сервисы, которые могут многократно использоваться в рамках всего предприятия.

Джеймс: На мой взгляд, SOA – непревзойденный маркетинговый трюк. Ничего глубокого я в ней не заметил. Она очень поверхностна.

Маркетинговых инструментов множество. Конечно, нужно прилагать усилия для распространения вашего продукта. Один из моих ранних проектов показал, что, выбрав для проекта хорошее название, можно год им не заниматься. За всю карьеру я придумал лишь одно замечательное название, но оно мне сильно помогло. На людей такие вещи действуют. OMT и UML – увы, мы не смогли придумать для них хорошие названия. Я не очень люблю акронимы и стараюсь их избегать, но иногда ничего иного не остается.

Какие связи вы видите между парадигмой ООП и нынешним интересом к параллелизму?

Джеймс: Концепция объекта как самостоятельного пакета, объединяющего структуры данных и поведение, идеальна для параллельной обработки. В реальном мире все самостоятельно и все параллельно, поэтому идея объектов в моделировании идеально подходит для параллелизма.

Однако это не распространяется автоматически на языки программирования. Почти все изучаемые ныне языки программирования по существу являются последовательными. Есть несколько языков в академической среде, параллельных в своей основе, но это не те языки, которым учат большинство программистов. Возможности параллельной обработки можно добавить в такие языки, как Smalltalk, C++ или Java,

но их базовые вычислительные модели и мышление в основном последовательны.

Так что проблемы не с парадигмой ОО, с языками программирования или системами. Разумеется, можно конструировать параллельные языки. Я сам сделал один такой язык для своей докторской диссертации в 1975 году, так же как и некоторые знакомые мне старшекурсники из Группы вычислительных структур Джека Денниса в MIT в начале 1970-х. Создать новый язык нетрудно (если только это не язык, позволяющий легко решить множество практических задач). Трудно добиться его распространения.

Создание языка не принесет вам дохода: суть популярности языка – в его широком использовании, а пользоваться языком, у которого есть владелец, никто не хочет. Отдельной компании трудно выделить необходимые ресурсы, если придется отдать результат работы бесплатно (а если они все же отдают продукт бесплатно, это может показаться подозрительным). Не верьте чепухе про изобретение еще одной мышеловки: чтобы добиться распространения чего-либо, нужны серьезные маркетинговые усилия. Поэтому я не питаю больших надежд на широкое распространение какого-нибудь хорошего параллельного языка программирования – не вижу, что может стать для этого побудительным мотивом.

Как разработать язык программирования, ориентированный на параллельную обработку?

Джеймс: Я работал с профессором Джеком Деннисом и его студентами в MIT над языками и машинами, применяющими потоки данных, когда писал свою докторскую диссертацию. Это были языки с очень развитым параллелизмом. Эти новшества породили широкие исследования в последующие годы. К сожалению, там было несколько проблем, которые не смог решить ни я, ни кто-либо еще. Идея была многообещающей, но осталась не доведенной до конца. Некоторые из тех идей я использовал в UML, но архитектура потоков данных в большинстве случаев вряд ли заменит архитектуру фон Неймана. Так что я попытался, но не вполне успешно.

Есть еще клеточные автоматы (cellular automata). Едва ли не половина моих однокурсников пыталась построить на их базе компьютер с высокой эффективностью распараллеливания. Наверное, это правильный подход, потому что именно так устроена вселенная. (А может, и не так. Современная физика удивительней фантастики. Новейшие теории предполагают, что пространство и время возникли из чего-то более примитивного.) Но клеточные автоматы оказываются применимыми только для решения некоторых геометрических задач – очень важных,

но недостаточно универсальных. Не удалось запрограммировать их для общего случая. Может быть, и общего случая не существует.

Важнейшей проблемой, видимо, оказывается интерфейс между потоком управления и структурой данных. Высокопараллельный поток управления плохо ладит с большими фрагментами данных, и непонятно, как обрабатывать данные параллельно, выполняя при этом привычную для нас компьютерную обработку. Возможно, нам нужна компьютерная обработка новых типов. Мозг – это высокопараллельный компьютер, который не применяет алгоритмы фон Неймана, но мы не имеем никакого представления о том, как программировать такую структуру, как мозг. Может быть, его вообще нельзя программировать: понятия эффективной программируемости и высокой степени параллелизма могут исключать одно другое. Может быть, это что-то вроде неопределенности Гейзенберга на более высоком уровне (еще одна физическая аналогия).

Когда вы говорите о языках программирования, приспособленных для параллелизма, возникает очевидная мысль о параллельных языках. В чем слабость функциональных языков и почему они не могут стать простым решением проблемы параллелизма?

Джеймс: Функциональные языки очень хороши для описания мира, который настолько параллелен, что и обсуждать параллельность не требуется. Но в реальном мире часто приходится занимать промежуточную позицию, когда некоторые виды параллельности явно нуждаются в обсуждении. Думаю, это еще одно нарушение симметрии. Тем не менее у функциональных языков есть масса достоинств: неплохо было бы использовать часть из них в более императивных языках.

Симметричные взаимоотношения

Вы сказали, что в большинстве языков программирования ассоциации дают более симметричную точку зрения, чем указатели. Нельзя ли рассказать об этом подробнее?

Джеймс: В реальном мире взаимоотношения – это действительно взаимоотношения. Они не ограничиваются действием в одном направлении. Иногда так бывает, но чаще если у A есть отношение к B, то и у B есть отношение к A.

Это двунаправленные взаимоотношения?

Джеймс: Это взаимоотношения. Взаимоотношения двунаправленны по своей природе. Математические отношения двунаправленны. Само слово «двунаправленный» вызывает представление об указателях. Взаимоотношения есть взаимоотношения. Две стороны связаны между

собой, и ни у одной из них нет первенства – так можно сказать. Двунаправленность, конечно, не означает симметричности: «человек кусает собаку» – не то же самое, что «собака кусает человека».

Представить структуру своих данных и своей системы в терминах взаимоотношений – хорошее начало в сравнении с попыткой инкапсулировать их в терминах указателей.

На уровне кода, конечно, вы зависите от конкретного языка программирования. В этом случае нужно думать о тех возможностях, которые предоставляет ваш язык.

Однажды я сделал язык программирования со встроенными взаимоотношениями, и он работал замечательно. Не нужно было связывать себя определенным направлением – можно было легко действовать в обоих направлениях. Издержки при этом не столь высоки. Удивляюсь, что этого нет в других языках.

Это был язык потоков данных?

Джеймс: Фактически это был пакет структур данных. Я назвал его DSM (Data Structure Manager) – менеджер структур данных. По сути это был набор процедур для структур данных, но я устроил так, что можно было мыслить в терминах отношений, действуя в обоих направлениях. Если вам нужно удалить кортеж из отношения, вы просто удаляете кортеж. Я провел оптимизацию, чтобы можно было действовать с любого конца, и организовал хеширование, чтобы сложность была линейной, благодаря чему снижение производительности было незначительным.

Все эти приемы хорошо известны, но средний программист не сможет тут же реализовать их. Нужно, чтобы их можно было взять из библиотеки или чтобы они были встроены в язык. Сомневаюсь, что большинство нынешних программистов умеет использовать хеширование. Этот язык не был крупным, но мы включили в него метод для хеширования множеств, отношений и расширяемых массивов. Когда вы записывали значения в массив, тот вдвое увеличивался в размере. Затраты оставались линейными за счет некоторого расхода памяти. Опять-таки, мы не стали переписывать заново весь язык, а перекрыли кое-что сверху. В результате многое упростилось. У нас никогда не возникало проблемы переполнения буфера. Такого типа подходы могут оказаться очень мощными.

Меня удивляет, что это не используется шире. Я знаю, что некоторые мои бывшие коллеги построили классы обобщенных шаблонов для C++. Не уверен, что они достаточно широко используются. Может быть, механизм в целом был достаточно сложен для использования в C++, или программисты поленились их изучить.

Что меня не удовлетворяет, так это водораздел между языками программирования и базами данных. С одной стороны – программисты, активно использующие указатели и озабоченные эффективностью и прочим, которые пишут ненадежные программы с ужасными ошибками.

С другой – те, кто занимается базами данных и мыслит отношениями. Они хорошо понимают концепцию отношений. От них я и узнал впервые это понятие. Они не озабочены эффективностью программ. В некотором смысле база данных очень неэффективна с точки зрения программирования, но это плата за ее надежность. Им нужны надежная работа с данными и отсутствие отказов. А между теми и другими никого нет. Наверное, это напоминает нашу нынешнюю политическую систему, когда есть только сторонники крайних точек зрения, а центристы повывелись. Я пытался разговаривать с теми, кто занимается базами данных, – они не проявили интереса к процессам в области языков программирования; то же можно сказать о представителях противоположного лагеря.

Уверен, что было бы полезно сблизить обе стороны, ослабив такое разделение между ними. Хорошо бы иметь языки, с помощью которых легко делать вещи в стиле реляционных баз данных. Они позволили бы легко работать в процедурном стиле и легко переходить от одного метода к другому и обратно, и повысить надежность приложений, используя встроенные средства. Это позволило бы эффективно справляться с вычислительной сложностью, чего не может делать средний программист, и меньше заниматься самостоятельным программированием на микроуровне. Люди думают, что они добиваются большей эффективности. Они ошибаются, потому что так теряется эффективность на верхнем уровне: за деревьями не видят леса. Они занимаются микрооптимизацией, что гораздо менее эффективно, чем оптимизация на верхнем уровне.

Кажется, в книге «The Practice of Programming» (изд. Addison-Wesley Professional)[1] Керниган писал, что в одной из старых утилит UNIX они воспользовались линейным сканированием.

Джеймс: Точно. Почему мы до сих пор используем UNIX? Какой сейчас год? А в каком году придумали UNIX? Системе 40 лет, а в ней до сих пор сохранились некоторые изначальные допущения, вроде помещения символов в байты. Нельзя поместить символ в байт. Мы ушли далеко вперед, а все еще представляем себе строки как массивы байтов. В наше время это уже не годится.

---

[1] Керниган Б., Пайк Р. «Практика программирования». – Пер. с англ. – Вильямс, 2004.

Конечно, с тех пор кое-что подновили, но UNIX исходно ориентировалась на PDP-7 с 64K памяти 18-битных слов (не каких-нибудь жалких 16-битных огрызков), и всю память нужно было периодически выгружать. Не знаю, сколько таких идей все еще таится там в глубине и готовит нам новые проблемы.

Здесь снова возникает та же проблема: люди пытаются писать новые операционные системы, но трудно добиться их широкого распространения. Мы пользуемся всяким старьем. В Windows есть код, написанный еще в 1980-е. Всего-то 20 лет прошло. Ого, да я старею. Тот, кто говорит «всего 20 лет», уже давно живет на свете.

Если знать историю, то множество сегодняшних «новинок» не воспринимаются таковыми, потому что вы все это уже видели не раз. Одно и то же великое открытие совершается заново, и каждый считает себя первопроходцем. Все это я уже видел неоднократно. Цитируя Элронда, «память моя уходит глубоко в древние времена, я видел много поражений и много бесплодных побед».

Позвольте высказать другую мысль. Я знаю многих, кто пытался убедить нас в существовании кризиса ПО. Под эту идею было продано множество разных средств моделирования. Я и сам замешан в этом деле. Но с другой стороны, можно пойти во Fry's Electronics и купить устройства или программы, которые год от года становятся все сложнее. И с каждым годом дешевле. Людям как-то удается быстро изготавливать новые приложения. Может быть, нет никакого кризиса ПО? Может, его размеры преувеличивают, пытаясь нас поразить?

Спорной является и такая вещь, как безопасность. Она у нас будет, когда люди решат, что это для них достаточно важно. Они говорят, что важно, но все свидетельствует, что не настолько важно, чтобы за это платить. Когда безопасность станет достаточно важна, поставщики ПО займутся ею. Не имеет значения, что люди говорят, если они не хотят потратить ради этого лишние деньги, циклы процессора, память или что-нибудь еще.

Если безопасность — важный вопрос, может быть, не стоит писать на Си или C++ с их опасными указателями?

Джеймс: Конечно. За что платишь, то и получаешь. Возьмите, к примеру, базы данных: если база данных теряет или портит ваши данные, долго она на рынке не протянет. Когда разработчики пишут приложение, управляющее базой данных, они тратят массу усилий, чтобы гарантировать надежность хранения данных. Плохо, когда происходят аварийные завершения, но когда теряются или искажаются данные, это гораздо хуже.

Работая над проектами, перед тем как выпустить версию, часто назначают приоритеты для выявленных ошибок. Обычно первоочередному исправлению подлежат фатальные ошибки, вызывающие аварийное завершение приложения. Думаю, это ошибочный подход. Если ваши данные испорчены, а система продолжает работать, это гораздо хуже, чем если программа просто аварийно завершилась, потому что вы не заметите проблемы и потеряете данные. На самом деле, даже неудобства в работе могут создавать больше проблем, чем фатальная ошибка.

Я построил первую версию инструмента моделирования. Я составил список приоритетов. В конце списка была функция, которой было тяжело пользоваться, а делать это приходилось постоянно. Эта, казалось бы, несущественная вещь настолько раздражала в работе, что я переместил ее на первое место в списке. То, что постоянно раздражает вас, может быть гораздо важнее ошибки, приводящей к краху системы, потому что если система аварийно останавливается, достаточно запустить ее заново. Если это вещь, которая досаждает при каждом щелчке мышью, таким инструментом вообще не будут пользоваться.

Меня чаще благодарят за исправление ошибок как раз такого типа.

Джеймс: Правильно. Речь идет о тех ошибках, которые действительно раздражают. Не знаю, читали ли вы книги Эдварда Тафта. Он писал про chartjunk, как он называл диаграммы, показываемые Excel: вся страница разукрашена, а информации почти никакой. Его идея была в том, что нужно стремиться к минимальному расходу чернил на единицу информации, находящейся на странице.

Ту же идею можно применить к интерфейсу пользователя. Я всегда считал, что лучшие приложения – те, где меньше всего щелкаешь мышью. Если вам нужно щелкать по всей площади окна, если нужно сделать два щелчка там, где хватило бы одного, это плохая разработка. Это один из тех раздражающих факторов, которые заставляют вас отказываться от программ.

Есть ли какой-то главный урок, который проектировщики и разработчики могли бы извлечь из вашего опыта?

Джеймс: Главный урок в том, что все подвержено изменениям. Это урок номер один по жизни в целом. Все меняется.

Нужно строить с учетом будущих модификаций. Если вы пишете приложение, постоянно помните, что в следующей версии все может измениться. Если вы получаете образование, постоянно помните, что знаний, полученных в колледже, не хватит, чтобы благополучно дожить до пенсии, а в течение жизни вы, возможно, не раз смените профессию.

Потребности делового мира изменятся. Сегодняшние проблемы могут исчезнуть, а могут остаться, но потерять важность по сравнению с но-

выми проблемами. Всюду царят перемены. Нужно быть готовым к ним, принять их и научиться ладить с ними, и тогда вас ждет успех. Тот, кто умеет приспосабливаться к переменам, успешен и счастлив. Остальных ждут неприятности. Не важно, чем вы занимаетесь – компьютерами или чем-то другим, – в наше время так происходит везде.

UML

Как вы определите UML?

Гради Буч: Унифицированный язык моделирования (UML) – это графический язык для визуализации, определения, обдумывания, документирования и конструирования системы, активно использующей ПО. В моем понимании UML – не язык программирования, а язык с более глубокой семантикой, чем у традиционных языков программирования, который позволяет работать на том же уровне абстракции, что и код, и/или выше.

На каких стадиях разработки эффективнее всего применение UML?

Гради: UML применим во время всего жизненного цикла преимущественно программных систем, от их рождения до смерти и возрождения. По моему личному убеждению, UML особенно полезен при обсуждении архитектуры системы. Я также сторонник модели представлений Крухтена «4+1», и, работая над справочником по программной архитектуре («Handbook of Software Architecture»), не встретил ни одной системы, для которой UML не позволил бы мне зафиксировать важные проектные решения.

Как он связан с разными методами разработки ПО?

Гради: Хотя у меня, Джима и Айвара были свои взгляды на методологию, мы не включили в UML ничего, что привязывало бы его к какому-то определенному процессу разработки ПО.

Допустим, я скептически отношусь к UML. Как бы вы попробовали убедить меня в его полезности?

Гради: Два момента: попробуйте с его помощью задокументировать уже написанную систему и посмотрите, не поможет ли он вам описать то, что находится выше кода; и еще – поищите в Сети примеры его использования и посмотрите, как его применяют (от MediaWiki до встроенных систем и множества систем в промежутке между ними).

Что вы думаете о применении UML для генерации кода реализации?

Гради: UML был предназначен – и по-прежнему применяется – для визуализации, определения, конструирования и документирования

систем, активно использующих ПО. В этой связи разработка на основе модели показала свою пользу при рассмотрении UML в качестве языка, позволяющего генерировать исполняемые модули. Тем не менее я отдал бы предпочтение использованию UML для визуализации конструкции строящейся или эволюционирующей системы (в противоположность системе, управляемой моделью).

Я читал, что в больницах палаты обычно двух- или четырехместные, потому что в споре часто двое объединяются против одного, причем в одиночестве все время оказывается тот же человек. Была ли работа втроем над UML связана с подобными трудностями?

Гради: Не забывайте, однако, что примерно с год нас с Джимом было лишь двое, следующий год мы работали втроем (Джим, Айвар и я), после чего участники исчислялись десятками, а потом и сотнями (когда UML решили сделать открытым стандартом). Так что состав участников был весьма динамичен. Во всяком случае, интересен не характер компромисса, а то, что нам удавалось прийти к общему мнению, несмотря на пылкий характер многих участников, чьи взгляды на проблему отличались. Это разнообразие с последующим представлением обществу обеспечило широкое распространение UML.

Если бы вам пришлось переделывать UML для версии UML 3.0, какой процесс вы бы предпочли?

Гради: Могу ответить точно, потому что я над этим размышлял. Сперва я бы перечислил сценарии использования, в которых собираюсь применять UML 3.0. В отношении процессов 1.*x* и 2.0 я сожалею, что многое из включенного сейчас в UML появилось стихийно снизу, а не на основе примеров, в которых говорилось бы: «Вот проблемы. Вот неприятные места. Как с ними поступить? Вот как мы поступаем в текущей версии UML. А вот как можно улучшить положение».

Сначала я собрал бы сценарии использования на практике, чтобы знать, какого рода вещи мы собираемся моделировать. После этого я постарался бы включить в версию 3.0 поддержку этих сценариев использования, а также провести рефакторинг метамодели с целью упростить ее.

Что бы вы в ней изменили?

Гради: UML нужно сделать проще, но это очень нелегко, потому что добавлять элементы для решения конкретных задач можно бесконечно. Кроме того, мне бы хотелось, чтобы UML развивался в сторону большей пригодности для использования в качестве системного языка.

Есть ли в UML те 20%, которыми все пользуются постоянно?

Гради: Как раз об этом я хотел сказать. В своей работе над справочником я до сих пор применяю классическое правило 80/20. Мне для ра-

боты нужно примерно 20% UML. Есть еще 80%, которые охватывают предельные случаи и детали, но для моих целей достаточно этого ядра.

Надо помнить, что разные люди используют UML по-разному. Если с помощью UML я создаю код, то, разумеется, мне нужно множество разных мелочей. Напротив, если я применяю его для обдумывания системы, визуализации, принятия обоснованных решений, то детали мне не нужны. Это своего рода господство меньшинства. Я так это называю. Потому что есть ряд применений UML, особенно для проектной документации, которые привели к значительному усложнению версии 2.0, а это осложнило его применение в других областях.

По большей части мне встречались дискуссии по конкретным случаям на белой доске. Вот мои модели классов. Вот модели сущностей. Вот взаимоотношения между ними.

Гради: Когда я занимался методом Буча, то совершенно не собирался превращать его в язык программирования. Если пойти таким путем и создавать язык визуального программирования, потребуется много дополнительного труда. Например, нужно зафиксировать семантику системы обозначений и метамодели. Ее не существует, и меня удивляет, что этим никто по существу не занимался.

Если у меня есть прямоугольник, что он означает? Честно говоря, формальной спецификации для него нет. Хотя семантике метамодели UML уделялось некоторое внимание, но связь нотации с метамоделью UML разрабатывалась хуже.

Возможны ли изменения, отражающие методы распределенной разработки и групповой работы?

Гради: Не могу ничего придумать. Я собираю различные шаблоны проектирования для распределенных систем, и там нет ничего, что я не мог бы описать на UML в его нынешнем виде. Что касается групповой работы, то я посвящаю много времени проблемам сред для коллективной разработки, включая такие экзотические вещи в виртуальных мирах, как Second Life; UML полезен, и я ничего не стал бы изменять в самом языке, потому что для разработки, распределенной во времени и в пространстве, важнее социальная динамика, чем технические проблемы.

Какие уроки из опыта изобретения, дальнейшего развития и распространения UML могли бы извлечь разработчики?

Гради: Я часто говорю, что для всей истории программирования характерен неуклонный рост уровня абстракций. Мы видим это по нашим инструментам, методам, языкам, средам. В этом отношении UML просто представляет естественный этап эволюции. На первых стадиях компьютерной эволюции господствовала аппаратная платформа; затем гос-

подствовал выбор языка; на нынешней стадии доминируют проблемы программной платформы (проявляющиеся главным образом в войнах операционных систем, особенно если взглянуть шире и рассматривать Интернет наравне с платформами). Это не означает, что прежние проблемы бесследно исчезли. Нет, они все в той или иной мере играют роль в современных системах. Если вернуться к UML, то он появился в тот момент, когда сложность программных систем достигла такого уровня, что провал или успех программного проекта в меньшей степени стал зависеть от проблем языков и программирования, чем от проблем архитектуры и организации совместной работы.

Я также часто повторяю, что разработка ПО была, есть и будет принципиально трудной задачей. Как красноречиво доказывает д-р Брукс, программам внутренне присуща сложность, от которой нельзя избавиться. Поэтому мы знаем, что как бы далеко вперед мы ни заглядывали, будущие успехи зависят от ПО, которое еще предстоит написать (хотя и развивая то, что есть сейчас). Программные системы прошлого кажутся нам сегодня тривиальными, хотя в свое время использовали все достижения практики, так же как сегодняшние системы, кажущиеся нам выдающимися достижениями, видимо, померкнут в сравнении со сверхсистемами будущего. Это заставляет нас неуклонно совершенствовать практику и профессию разработки ПО.

Разработка языка

Как конструкция языка влияет на конструкцию программ на этом языке?

Гради: Вы задаете очень старый вопрос, хотя он и облечен в новую форму: лингвисты и и специалисты по когнитивной психологии обсуждают его уже десятки лет, причем большинство споров ведется вокруг так называемой гипотезы Сепира–Уорфа. Эдвард Тафти тоже отмечает, что хорошее представление свертывает сложность, давая возможность содержательно обсуждать сложную информацию абстрактным образом.

Если подробнее, то гипотеза Сепира–Уорфа (по имени лингвистов Эдуарда Сепира и Бенджамина Уорфа) постулирует связь между языком и мышлением: синтаксические и семантические элементы разговорного языка влияют на то, как человек воспринимает окружающий мир и мыслит (и наоборот). Современные лингвисты, например Джордж Лакофф, автор книги «Women, Fire, and Dangerous Things» (изд. University of Chicago Press)[1], согласны с этим. Работа Тафти «The Visual Display

[1] Лакофф Дж. «Женщины, огонь и опасные вещи: что категории языка говорят нам о мышлении». – Пер. с англ. – Языки славянской культуры, 2004.

of Quantitative Information» (Визуальное отображение количественной информации), изд. Graphics Press, посвящена визуализации сложных данных и на многочисленных примерах показывает, что эффективная графика может сделать информацию понятнее.

Вас может заинтересовать, почему мне знакомы эти вопросы? Потому, что объектно-ориентированные разработки основаны на четко построенных абстракциях, а абстракция, в первую очередь, это проблема классификации, и такие лингвисты, как Хомский и Лакофф, повлияли на мои представления о классификации.

Как бы то ни было, если видоизменить ваш вопрос и спросить, есть ли связь между конструкцией языка и конструкцией ПО, – в предположении, что под языком можно понимать как классические текстовые языки программирования вроде Java, так и графические вроде UML, – моим ответом будет «возможно».

Совершенно явным образом языки, поддерживающие алгоритмическую декомпозицию (например, Фортран и Си), приводят к определенным типам структуры программы, которые весьма отличаются от типов, возникающих при работе с языками, поддерживающими объектно-ориентированную декомпозицию (например, Java) или, может быть, функционально-ориентированную. Однако мой опыт показывает, что есть факторы, оказывающие большее влияние на конструкцию: культура команды разработчиков, исторический контекст, конкретные силы, оказывающие давление на систему в любой данный момент времени. Можно доказывать, что язык относится к первопричинам, но я утверждаю обратное и считаю, что преобладающее значение имеют другие силы.

Чем разработка языка программирования отличается от «обычного» программного проекта?

Гради: Возможно, тем же, чем внесение проекта какого-нибудь общеполитического закона и осуществление этого закона на практике: вещи связанные, но весьма отличные одна от другой. Язык, будь он естественным или компьютерным, не имеет неограниченного числа степеней свободы, а формируется в результате действия технических, деловых, социальных, исторических и практических факторов. Языки программирования в особенности требуют точных синтаксиса и семантики, потому что в итоге с их помощью получают исполняемые артефакты. Не знаю, существует ли такая вещь, как «стандартный» программный проект, но там значительно больше неоднозначностей. Язык, будучи определен, относительно стабилен. Программный проект, если он представляет какой-то экономический интерес, существует в постоянно меняющейся среде. Таким образом, хотя как язык, так и проект являются техническими задачами, подлежащими решению, но если разобрать

действующие на них силы, то силы, влияющие на проект, оказываются более разнообразными и динамичными.

Как вы полагаете, нужно ли вначале иметь хоть какую-то формальную спецификацию ядра языка, чтобы потом развивать его дальше?

Гради: Обязательно. Фактически мы с Джимом и начали с попытки зафиксировать семантику того, что впоследствии стало UML, и унифицировать OMT и метод Буча. Сначала мы написали метамодель с помощью самого UML. Проблема в том, что один считает это формальным, а другой – неформальным. У нас вопрос стоял так: достаточно ли здесь формализма, чтобы сделать то, что нам нужно? И ответ был положительным.

Как вы определили, что достаточно?

Гради: Был такой судья Верховного суда – эту историю приводят достаточно часто, – которого спросили, как он определяет, что такое порнография. Его ответ был – вы, вероятно, слышали – «Я знаю это, когда ее вижу». Так же и здесь. Если вы начнете обсуждать, что такое смысл, то впадете в одну из совершенно метафизических дискуссий, потому что чем в действительности является формализм? Можно уйти в полные дебри, выясняя, что такое смысл. Даже в формализме нужно в какой-то момент остановиться.

Многие могут сказать: «Мы полагаемся на полноту по Тьюрингу и лямбда-исчисление, и вдобавок я умею применять функцию к аргументу».

Гради: Я очень счастлив, что такие люди в мире есть, но представьте, что с такой же строгостью мы подошли бы к Java или Vista, и всему тому, что понаписано для этих языков и платформ, для которых нет никакого формализма. Есть операционная семантика. Мы запускаем эти программы и видим, что они работают. Этого вполне достаточно для большей части инструментов, с помощью которых мы сегодня пишем программы. Это не значит, что формализм не нужен. Он нужен. Есть отдельные – буду очень тщательно подбирать слова – отдельные очень ограниченные области в нашей отрасли, где глубокая формализация имеет очень большое значение.

Нужно ли учитывать разницу между теми языками, посредством которых можно строить системы, и теми, где вам может потребоваться определенная степень формализма?

Гради: Конечно. А есть ли формальная семантика для Linux? Есть ли формальная семантика для C++? Конечно. Я предполагаю, что над их отдельными элементами велась работа, но это не помешало строить реальные вещи.

Тогда здесь присутствуют и очень практические соображения.

Гради: Разумеется. Я – прагматик. Если что-то работает и работает хорошо, то я использую это, откуда и возникают те 20% UML, которыми я пользуюсь. Для моих целей этого достаточно.

Ваши 20% могут не совпадать с моими 20%.

Гради: Я бы предположил, что они будут изрядно перекрываться. Скажем, ваши 19% и мои 19% совпадут.

Честно говоря, неплохое совпадение.

Гради: Да, действительно.

Где бы вы начали искать сценарии использования для этих 19%?

Гради: Я бы посмотрел, как люди используют UML. Я бы обратился к реальным проектам, где пытаются применять UML, и задал такие вопросы: «Покажите мне стандартные сценарии. Покажите мне главные сценарии использования и давайте сделаем так, чтобы в них было легко применять UML. Затем взглянем на какие-нибудь предельные случаи». Я бы исходил из реального, а не предполагаемого применения.

Настало время уплотнять и упрощать. Третья версия широко используемого языка – тут имеют значение не только инновации, но и рефакторинг. UML 2.0 в некоторой мере – я выскажусь немного резко – пострадал от эффекта второй версии, поскольку были огромные возможности и группы особых интересов, шумно требовавшие включения некоторых специфических функций, что привело к непомерному раздутию UML 2.0. Сейчас нужно отойти в сторону – переработать и упростить язык. У всех серьезных систем бывают периоды расширения и сокращения. Сейчас настало время упрощения.

Какая тут система – через раз?

Гради: Я как раз сейчас размышляю вслух о версиях операционной системы Microsoft. Если взглянуть на Windows 7, то да, похоже.

Или как с фильмом «Звездный путь».

Гради: Ну, тут явно нужно провести широкие формальные исследования.

Какое значение придается обратной совместимости с UML 2.0?

Гради: Нужно помнить, что в UML 2.0 много разного, и если бы мы должны были неуклонно двигаться вперед, обеспечивая, как вы сказали, полную совместимость с прошлым, объем языка продолжал бы увеличиваться. Видимо, есть небольшие частные случаи, где я хотел бы отказаться от совместимости, сказав при этом: «Это сильно все осложняет. Оно того не стоит. Извините, я вынужден кого-то огорчить. Вот возможные выходы из положения».

Если взять основную часть UML, которой все пользуются, то эти 20% мы ломать не будем. Обеспечить для нее обратную совместимость – очень важная задача.

Почти всем разработчикам языков приходится искать решение этой проблемы, и есть множество разных подходов. Лисп практически заявляет: «Не будет вам системы объектов, можете строить ее сами» и «Не будет вам таких-то управляющих структур, можете строить их сами». Все так и поступили и начали делиться своим кодом. Затем Common Lisp попытался зафиксировать то, чем пользуются все.

Гради: Это хорошее описание.

В какой-то степени это также алгоритм эволюции или алгоритм имитации отжига, когда сообщество определяет, что ему нравится, а разработчики вводят это в ядро.

Гради: Именно так. Вот почему я говорил о сценариях использования, взятых из самой промышленности. Как люди реально пользуются этим? Взгляните на это с точки зрения предприятия, где занимаются действительно крупными системами. Посмотрите также на какие-то малые системы.

Какова ваша точка зрения на процесс стандартизации с учетом вашего опыта его прохождения?

Гради: Я участвовал в нескольких процессах стандартизации. Это замечательная, интересная процедура, проследить за которой, как мне кажется, было бы любопытно многим социологам. Относительно процессов стандартизации трудно делать обобщения. Есть стандарты – не буду называть имен, – проталкиваемые в конкретных отраслях, и они становятся стандартами конкретных компаний, стандартами де факто. Есть другие, которые действительно являются результатами совместных усилий. Есть стандарты, политическая направленность которых очевидна, и их насильно навязывают другим. Так что есть много путей возникновения стандартов.

Самое восхитительное в том, что несмотря на все претензии к системе стандартов, она действительно полезна и действительно работает. Поэтому я благодарен таким организациям, как OMG, за то, что они тратят свои силы на управление подобными вещами и проводят обсуждение их дальнейшего развития. Без таких стандартов не было бы Сети.

Это тяжелый процесс, в нем много участников, они увлечены, у них свой взгляд на мир, в правильности которого они уверены; такова природа человеческого познания.

Не сложилось ли у вас впечатление, что все оказываются слегка недовольны, но чувствуют, что остальные тоже слегка недовольны, то есть все в порядке?

Гради: Не обязательно все так плохо, но стандарты постоянно требуют компромиссов. Это верно для любого мыслимого стандарта. Всегда кто-нибудь бывает в итоге разочарован. На самом деле, до сих пор есть те, кто разочарован избранием Обамы и находит утешение в Буше. Или, скажем, кому-то нравятся песни Бритни Спирс – вкусы бывают разные. Опять-таки это в человеческом характере, и давайте для разнообразия прислушиваться к ним.

Взять ядро Linux – там нет настоящего стандарта помимо POSIX. Все определяется вкусом Линуса Торвальдса и его помощников.

Гради: Пожалуй, это хорошее сравнение. В случае Linux есть лидер, которым в течение известного времени является Линус. В случае UML Джим Рамбо и я отстранились от процесса стандартизации, пустили его на волю. Там не оказалось достаточно мощной направляющей силы, и это, думаю, наложило отпечаток. Взгляните на C++: Бьерн все еще активно участвует в развитии C++. Его твердая рука способствует интеллектуальной целостности и последовательности.

С новой версией они тоже проходят процесс стандартизации.

Гради: Верно. Но в случае Linux слышен голос человека с большим опытом, на него возлагают большие ожидания, и он их оправдывает. Это не единственный голос, но он сильный и ясный.

Следующий вопрос, то есть два. В чем ценность стремления к стандартизации для идеи? Насколько успех определяется наличием сильного лидера с четким видением?

Гради: Последний вопрос относится к любому роду человеческой деятельности. Взгляните, как происходят изменения в мире. Я не сравниваю наш проект с делами этих людей, но посмотрите, чего добились Ганди или Мартин Лютер Кинг. В нашем мире личность играет огромную роль в осуществлении перемен.

Если говорить о технике, можно вспомнить целый ряд лидеров в области технологий, которые оказали большое влияние на ход событий. К примеру, Ларри и Сергей из Google воплотили свои идеи, вынесенные из Стэнфорда, и сейчас это огромная империя. Наличие сильной и прозорливой личности определяло прогресс очень во многих областях.

Я не предлагаю считать эти идеи конкурирующими.

Гради: Это просто пример в подтверждение того, что энергия личности или небольшой группы часто становилась источником фундаментальных перемен.

Допустим, я хочу создать язык программирования. Какую пользу я могу извлечь из стандартизации?

Гради: Когда вы создаете нечто новое, его успех в конечном итоге определяет рынок – независимо от того, удалось ли вам зарегистрировать свой стандарт. Многие языки сценариев возникли без участия комиссий по стандартизации, а просто трудами обычных людей, и когда была достигнута достаточная критическая масса пользователей, люди поняли, что им нужна более точная стандартизация, чтобы обеспечить совместную работу. Смысл в том, что во всяких вещах такого рода нужно учитывать, какова ценность вашего продукта и как на него будет реагировать рынок. Процесс стандартизации может помочь набрать некую критическую массу для воплощения ваших стремлений, но в конечном итоге все решит рынок.

Проблема любого нового языка в том, что он может быть совершенным с технической точки зрения, но есть тьма разных факторов, преимущественно социального характера, влияющих на успех этого языка. Смогу ли я собрать группу пользователей? Сможет ли язык решить острые практические проблемы, имеющиеся в данный момент в этой конкретной группе? Возникнет ли к нему достаточный интерес у отдельных лиц в разных организациях, чтобы они приняли участие в его развитии, если в силу недавнего появления для него не видно никакой бизнес-модели, а есть только ощущение правильности?

Многие из таких ранних разработок основываются, так сказать, на вере и оказываются в нужном месте в нужное время. Это, несомненно, относится к методу Буча, а если спросить у Джима Рамбо, то он, наверное, ответит, что это относится к Objectory и OMT, на которые повлиял UML. Мы оказались в нужном месте в нужное время, попав в болевую точку рынка.

И вот сегодня – нужен ли миру еще один язык? Тут может прийти на ум M, который выпустила на рынок Microsoft. Ждет ли M успех? Технически он очень интересен, но успех будет определяться тем, покажется ли он привлекательным рынку. Пусть даже он стандартизирован, пусть это стандарт де-факто усилиями Microsoft – решать будет рынок.

Похоже, помимо прозорливости нужно еще обладать большой практичностью.

Гради: Совершенно верно. Я уверен, повторяю, что будь у вас лучшая идея, техническое совершенство, полная формализация, глубокая продуманность и исчерпывающая документация, они не произведут впечатления, если вы не предлагаете решения реальных проблем в реальных проектах.

Не всем это понравится: похоже, существует технический утопизм, основанный на вере в победу самого совершенного в техническом отношении проекта.

Гради: Приятно думать, что есть такие оптимисты. Их оптимизм придает нам сил. Но я разрабатываю решения и имею дело с практиками. Я не занимаюсь исследованиями в компьютерных науках. Я скорее инженер. Прекрасно, что у нас есть оба эти взгляда на мир, потому что эти танцы, эта напряженность способствуют большей честности обеих сторон. Мной движет мой практицизм, но, с другой стороны, чистый теоретик заставляет меня уделять больше внимания чистоте и формализму, и это совсем не плохо. Со своей стороны, я побуждаю его быть практичным.

Между этими двумя полюсами возникает творческое напряжение?

Гради: Совершенно верно. Я не сомневаюсь в этом. По моим представлениям, творчество связано с напряжением и направляет это напряжение на решение реальных проблем. Есть замечательный сайт Gaping Void. Автор в основном занимается рекламой, а прославиться хочет благодаря художественным изображениям на обороте визитных карточек. Но у него есть очень популярный припев о том, что нужно для творчества. Можете дать своим читателям ссылку.[1] Это очень интересно, потому что он действительно говорит о важности своего рода практического напряжения.

Обучение разработчиков

Почему мы так медленно прогрессируем в области методов и процессов программирования?

Гради: Должен возразить против исходной посылки вашего вопроса. Наверно, это только кажется, что медленно, потому что мы у себя внутри знаем, что можно делать гораздо больше и делать лучше. Но взгляните – наша отрасль буквально изменила мир фактически за одно поколение. На мой взгляд, это быстро, а не медленно.

Как можно передавать свой опыт в области программирования?

Гради: В средние века главным механизмом передачи общественных знаний служили гильдии; сегодня в программировании отсутствует система ученичества. Все же значительная часть опыта передается через Сеть (например, Slashdot), книги, блоги и технические конференции. Кроме того, сам по себе работающий исходный код служит источником знаний, переданных нам из прошлого, и это одна из причин, по кото-

[1] *www.gapingvoid.com*

рым я сотрудничал с Музеем компьютерной истории с целью сохранения кода классических программ для будущих поколений.

Что следует глубже изучать сегодняшним студентам?

Гради: Я отвечу на этот вопрос двояко. С позиций программирования, любой хороший учебный курс научит вас основам программирования и разработки ПО. Однако я особо порекомендовал бы три вещи: учиться абстрагировать, учиться работать в команде и изучать чужой код. С более широких позиций, я бы посоветовал студентам увлеченно и самоотверженно заниматься любимым делом, но все же не забывать о своем росте как личности в целом.

Изучайте предметы за рамками своей профессии (в мире немало всего помимо программ), вырабатывайте у себя умение непрерывно учиться (потому что компьютерная область будет изменяться непрерывно) и не давайте угаснуть своей любознательности и готовности пойти на риск (это источник всех новшеств).

Какой один главный совет вы бы дали начинающему программисту, исходя из собственного опыта?

Гради: Как раз такой вопрос недавно задал мне один студент из Университета Южной Калифорнии. Несколько недель назад я прочел ряд лекций в Калифорнийском политехническом институте и в Университете Южной Калифорнии. Некоторые мои слушатели задали мне потом вопросы, и я приведу тот же ответ, который тогда дал им.

Прежде всего, следуйте своему увлечению, так чтобы оно доставляло вам радость. Конечно, стремление сделать карьеру и заработать средства на жизнь имеет смысл, но в конце концов разработка ПО и все прочее, чем мы занимаемся, это наша жизнь, и нужно прожить ее цельной личностью. Пожалуйста, получайте удовольствие, живите полной жизнью, набирайте опыт. Следуйте своим увлечениям, потому что иначе легко можно попасть на такую работу, которую вы просто возненавидите. Постарайтесь этого избежать. Вот вам мой совет.

Другой мой совет – постарайтесь приобрести практический опыт. Примите участие в каком-нибудь проекте open source. Найдите то, что вам интересно, и займитесь этим. Не бойтесь браться за новое и разнообразить свою жизнь. Поверьте, это принесет вам пользу, какой бы ни была эта новая область.

Снова и снова вспоминается музыка. И творческие искусства, особенно литература.

Гради: Если говорить о литературе, то я часто спрашиваю у преподавателей, есть ли у них курсы по чтению программного кода. Мне встре-

тились только двое ответивших утвердительно. Тот, кто специализируется в английской литературе, читает работы мастеров. Тот, кто хочет стать архитектором, строить дома, изучает работы Витрувия и Фрэнка Ллойда Райта, Кристофера Рена и Фрэнка Генри и других. В программировании такого нет. Мы не изучаем работы мастеров. Я советую изучать чужие работы и учиться на них.

Неплохо было бы взять за образец литературное произведение, сказав о нем: «Вот как выглядит великая программа на Паскале!»

Гради: Это то, что попытались сделать Энди Орам и Грег Уилсон в книге «Beautiful Code»[1] издательства O'Reilly. Есть еще книга одного новозеландца, который также составил список для чтения. Это один из двух известных мне людей, которые действительно это сделали.

У нас нет корпуса знаний, нужного для литературной критики программ. Работа Кнута по грамотному программированию была, кажется, одной из первых попыток в этой области. Обычно литературные качества кода весьма слабы. Можно сравнить с уровнем третьего класса, где учат разбирать структуру предложений. Идеальный код полон драматичности и красоты, изящества и хорошего стиля. У меня была возможность увидеть исходный код Mac Paint – примерно 10 000 строк Object Pascal, – и он прекрасно читается. Примеров открытого кода такого качества у нас нет.

Другая проблема в том, что если у вас есть система вроде ядра Linux, в коде которого 10 миллионов строк, читать это невозможно. Все равно что заново перечитывать «Войну и мир». Как мне показать его красоту и элегантность? В январе должна состояться конференция «Rebooting Computing» под руководством Питера Деннинга и Алана Кэя – там будет много участников, и я в том числе. В числе обсуждаемых проблем будет и эта. Что сделать, чтобы показать миру красоту этих совершенно скрытых интеллектуально сложных вещей?

Аналогичная проблема с Музеем компьютерной истории. Я там вхожу у состав попечительского совета. Мы учредили комиссию по коллекциям ПО, и у нас есть замечательные артефакты. Как нам представить красоту, присутствующую в Vista? Там есть красота. Как показать, как дать людям возможность прочесть, что у нее внутри? Если это архитектура, можно показать людям здание. Можно показать картину. Можно дать послушать музыкальное произведение. Но как продемонстрировать музыку программного обеспечения?

Может быть, нужно документировать алгоритмы и структуры данных?

---

[1] Э. Орам и Г. Уилсон «Идеальный код». – Пер. с англ. – СПб.: Питер, 2009.

Гради: Мне кажется, это слишком низкий уровень. Мое предложение – документировать шаблоны, которыми пронизана система.

Творчество, изящество и шаблоны

Как решать проблему устаревшего ПО?

Гради: Я часто повторяю, что хотя код – истина, но не вся: между видением и исполнением теряется информация. Мой опыт показывает, что с устаревшим ПО можно поступить девятью способами: выбросить, передать, не замечать, поместить в реанимацию, переписать, изъять здоровые органы, заключить в оболочку, трансформировать или сохранить. У каждого способа есть техническая и социальная стороны. С технической точки зрения, есть интересное текущее исследование возможности извлечения из кода шаблонов. С социальной точки зрения, решению может помочь метод изустных историй.

Что вдохновляет вас на разработку?

Гради: Я нахожу вдохновение в красоте сложных вещей. Это могут быть программы (одна из целей «Справочника» – кодифицировать ряд архитектурных шаблонов, объяснив, в чем их красота); это могут быть органические системы (на самом деле, любая органическая система: если система эволюционировала миллион лет под действием внешних сил, у нее явно есть чему поучиться); искусство (огромная красота создана многими художниками с помощью разных средств); музыка, промышленный дизайн, квантовая физика… Мой список бесконечен.

В какой мере творчество необходимо в программировании?

Гради: Обычно программирование не требует большого творчества, потому что вы решаете известные задачи, пытаясь найти для этого интересные способы. Это не лишено сходства с возведением пристройки к зданию. Нужно соблюсти определенные ограничения и следовать определенным лучшим практикам. Я найму для этой работы того, кто пусть не полностью готов к ней, но зато может действовать в соответствии с лучшими практиками. Ему придется придумывать что-то новое во время работы. Что-то вроде «тут небольшая нестыковка, поэтому нужно придумать какой-то новый способ сделать *X*». Вот такое новаторство для отдельных мелких разработчиков.

Значительная часть разработки ПО не требует особого новаторства. Есть известная технология строительства. Нужно применить ее новым образом. Нужно придать форму, сгладить острые края, где-то ввести дополнительные элементы. Решение таких задач может быть интересным. Несомненно.

Но есть случаи, когда инновации нельзя ограничивать. Никто не знал, как правильно делать ПО для гигантских систем глобального поиска, и тут появились Сергей и Ларри, создали прототип – и так возник Google. Мы не знаем, как правильно подойти ко всем этим каналам данных, поступающих от десятков тысяч, если не миллионов, видеокамер, размещенных на улицах Лондона, Нью-Йорка или Пекина. Какая архитектура здесь нужна? У нас нет готовых подходящих моделей, поэтому нам нужны необузданные, ничем не ограниченные инновации.

Но как только вы приблизились к подходящей архитектуре, возникают ограничения, и масштаб последующих инноваций ограничивается.

Вы снова употребили слова «инновации» и «ограничения». Наблюдается явная параллель с эволюцией языка.

Гради: Именно. Что больше всего тревожит художника, писателя и любого творческого работника, так это полная свобода в отсутствие всяких ограничений. Нет никаких ориентиров. Как только налагаются ограничения, происходит любопытная вещь. Вы получаете свободу, потому что можете теперь действовать в рамках этих ограничений и использовать все свои творческие способности для их преодоления.

Допустим, я музыкант и размышляю: однако – мне все позволено! В отсутствие ограничений возникают разные мысли: играть на фортепьяно или найти что-то другое? И стоит ли вообще? В действительности, богатство выбора смущает и не дает сконцентрировать свои силы. Возможно, мне придет в голову сделать собственное фортепьяно. Или поступить, как Дональд Кнут: «Я пишу книгу, и мне не нравится, как выглядит ее набор, так что я прервусь на пару лет и напишу язык, который даст возможность набирать книгу так, как мне нравится». Это не в упрек Дону, потому что он сделал некоторые замечательные вещи, но к тому, что в отсутствие ограничений трудно сконцентрировать свои силы в одном направлении.

По-вашему, чтобы построить новую систему, нужно сначала определить ограничения, а потом принять их?

Гради: Не думаю, что всегда можно действовать в таком порядке. С ними как бы соглашаешься и можешь выбрать для себя какие-то ограничения, после чего каждое мое решение означает согласие не заниматься другими вещами, что неплохо. Очень многие из первых принимаемых в таких проектах решений устанавливают ограничения путем акта веры. Вы не можете a priori решить, какими будут все эти ограничения, и надо попытаться опробовать хоть что-то, предъявив миру, а потом руководствоваться этим. Например, в простейшем случае я могу сказать: послушайте, я хочу разработать новый язык графического программи-

рования. И пошло-поехало. Ни с того ни с сего я принимаю решения вроде «это будет не не 3D-, а 2D-язык». Может быть, я решу, что для меня важен цвет, но вдруг пойму, что восстанавливаю против себя всех программистов-дальтоников. Каждое из решений становится ограничением, которое я должен выяснить, и мне придется встретиться с последствиями таких ограничений.

Здесь напрашивается итеративный процесс.

Гради: Совершенно верно. Жизнь вообще итеративна. Это возвращает нас к моему прежнему замечанию, что заранее нельзя даже знать, какие нужно задавать вопросы. Нужно совершить акт веры и двигаться вперед, несмотря на недостаточность информации.

Насколько вероятно появление в ближайшие 10 лет революционного языка или системы визуального программирования?

Гради: Так она уже есть. Это Lab View, созданный National Instruments. Самый умопомрачительный язык визуального программирования, какой я только видел. Не знаю, знакомы ли вы с ним. Он совершенно изумителен. Вы рисуете прямоугольники и линии, которые представляют виртуальные инструменты и виртуальные электронные устройства. При необходимости можно спуститься ниже и использовать, кажется, Си и C++. В принципе, можно строить виртуальные инструменты и подключать их к реальным устройствам, и это очень круто. Такая штука уже существует.

Какого рода инструменты?

Гради: Осциллоскопы, мониторы. Можно взять ПК, подключить к его порту USB какие-нибудь АЦП, преобразователи параллельных сигналов в последовательные и так далее, после чего работать с реальным оборудованием. Теперь можно строить на ПК виртуальные инструменты простым вычерчиванием схем. Может быть, вам нужен самописец, или осциллоскоп, или нужно преобразовать каким-то образом данные и представить их в интересном виде в каких-то измерительных приборах – в Lab View это запросто. Невероятно здорово.

В том же направлении пытается работать Чарльз Симони со своей компанией Intentional. По его идее – и он собрал там несколько прототипов, – если я инженер-электрик, то пользуюсь схемами и могу вводить их в свою систему. Я не видел все остальные, но это один из классических примеров. Я выбираю ту визуализацию, которая уместна в моей предметной области.

А как насчет обычных бизнес-приложений? Есть сервер базы данных. Есть бизнес-объекты. Есть уровень представления.

Гради: Большинство промышленных систем в архитектурном отношении не интересны из-за относительной простоты. Они разочаровывают. Нужно принять ряд конструктивных решений, и эти решения известны. Вся сложность в том, что появляется огромное количество технологий, и это проблема выбора. Есть слишком много способов сделать то, что мне нужно.

То есть, допустим, я решаю строить с нуля банковскую систему. Вероятно, есть ряд стандартов, о которых я должен сначала подумать, но количество способов решения этой задачи конечно, хотя и очень велико. Вот тут, я думаю, и возникает множество проблем и неразбериха. Возьмем самую изменчивую часть сегодняшних бизнес-приложений. Это уровень представления. Как мне представлять информацию клиентам? Десятилетиями люди довольствовались печатными документами, но сейчас открылись замечательные способы доступа через Сеть и мобильные устройства, и теперь в этой области появляется множество инноваций, потому что не устоялись модели того, как люди хотят работать с этими системами. Отсюда такая карусель.

Следующая по степени непостоянства область – это бизнес-правила. Проблема пришла из прошлого, когда наши машины так ограничивали нас, что приходилось засовывать бизнес-правила в какие-то жуткие места, в хранимые процедуры, и вводить их в броузере, а они раскиданы повсюду. Мы понимаем, что это в некотором роде проблема, потому что нам трудно реагировать на изменения, а недолговечные бизнес-правила меняются очень быстро.

Можно наблюдать, как для многих систем предприятий проводят рефакторинг, извлекая бизнес-правила. Есть способы устроить так, чтобы система предприятия просматривала это хранилище бизнес-правил и реагировала на них. С другой стороны, мы не могли этого предвидеть. Мы строили первые системы, не зная, какие их части будут наиболее подвержены изменениям, и это далеко не все изменения, происходящие сейчас в этой области.

Возможна ли здесь визуализация? Несомненно. Мы видим возможность того, что будут найдены новые языки для описания бизнес-правил. По моему личному мнению, UML было вполне достаточно, и мы видели интересные политические события, приведшие к созданию BPEL (Business Process Execution Language) как вполне независимого от UML явления.

Microsoft могла бы утверждать, что такой попыткой является M.

Гради: Совершенно верно, что возвращает нас к сказанному мной ранее: будет ли M успешным? Все решит рынок.

Люди, возможно, даже не понимают, что им нужен отдельный уровень для бизнес-правил.

Гради: Совершенно верно. Есть замечательная статья исследователей из IBM под названием «The Diary of a Datum» (Дневник элемента данных).[1] Мы можем добавлять новые уровни, которые отчасти помогают нам обдумывать и визуализировать эти системы, но в итоге это приводит к появлению немыслимого числа уровней в исполнительной части наших систем. В «Дневнике» описано, как мы смотрим на простой элемент данных, не думая о многочисленных преобразованиях, записи в поток, чтении из потока, кэшировании, в результате чего с ним произошло нечто реальное. Такова цена добавления в систему уровней абстракции.

Мне кажется, что средняя сложность и размеры ПО растут год от года. Может ли ООП помочь в борьбе с этим?

Гради: Если под ООП вы имеете в виду конкретные классы языков, то можно утверждать, что указанные языки действительно обладают большими выразительными возможностями, чем другие языки, и потому предоставляют более высокий уровень абстракции (и требуют меньшего количества строк кода для представления). Если вы подразумеваете под ООП просто философию декомпозиции – в противоположность алгоритмическим или функциональным абстракциям, – то мы возвращаемся к одному из положений «Категорий» Аристотеля: для описания сложных вещей совершенно необходимы множественные виды декомпозиции.

Замечу, что не доказана изоморфность сложности размеру/количеству строк кода. Есть другой вид сложности, который лучше измерять тем, что я называю *семантической плотностью* (отношение смысла выражения к количеству семантических связей). Растет ли семантическая плотность? Думаю, да, но мне кажется, что она ортогональна средствам выражения – объектным или нет, – используемым для описания такой семантики.

Сегодня широко обсуждается параллелизм ПО.

Гради: Эта проблема не нова. Моделирование параллельности (многозадачность) на отдельном процессоре – старая концепция, а в тот день, когда к первому на земле компьютеру добавился второй, люди задумались, как заставить их работать вместе. Сегодня существуют зоны вычислений, но чаще вы сталкиваетесь со слабо связанной распределенной параллельностью (например, Сеть) либо с тесной параллельностью

[1] Mitchell, Nick et al. «The Diary of a Datum: Modeling Runtime Complexity in Framework-Based Applications», IBM Research (2007).

(многоядерные процессоры в компьютерах с массовым параллелизмом). Короче, эта тема всегда «широко обсуждалась», и надо признать, что это действительно сложная проблема. Средний разработчик не умеет строить распределенные, параллельные и защищенные системы, потому что для них нужны системные решения.

А как насчет параллельности в разработке? Мы когда-нибудь сможем ускорять разработку, добавляя новых работников?

Гради: Я опять вынужден оспорить исходные посылки вашего вопроса.

«Ускорение разработки» – это одна из возможных выгод, но желательными результатами добавления новых работников можно также считать «улучшение качества», «рост функциональности», «уменьшение сложности» и другие. Трудно отрицать, что добавление новых работников увеличивает количество рабочих циклов, но при этом также растут шум, затраты на обмен данными и стоимость памяти проекта. Реальность такова, что наиболее экономически перспективные преимущественно программные системы требуют немалого количества людей, а самые перспективные из них – сотни, если не больше, основных разработчиков. Так что ваш вопрос в некотором отношении не интересен. :-)

Что ограничивает эффективность совместной работы при разработке ПО?

Гради: Мой опыт показывает, что есть ряд моментов в повседневной деятельности разработчика, которые по отдельности или совокупно влияют на эффективность команды:

- Стоимость начальной и текущей организации рабочего пространства
- Неэффективность совместной работы над продуктом
- Поддержание эффективного обмена данными в группе, включая знания и опыт, состояние проекта и память проекта
- Потери времени при переключении между задачами
- Переговоры с заинтересованными лицами
- Неработающие средства

Обо всем этом я писал в статьях для CDE, которые есть здесь: *http://www.booch.com/architecture/blog.jsp?part=Papers*.

Как вы узнаете простоту в системе?

Гради: Этот самый вопрос мне задавал Дэйв Парнесс. Я недавно написал статью для IEEE в своей колонке по архитектуре и начал ее словами: «Взгляните на огромный многотонный кусок гранита. Он огромен, но очень прост. Взгляните на спираль ДНК. Она маленькая, но очень сложная». Дэйв написал мне по этому поводу возражение, потребовав, чтобы я доказал, почему я так считаю. В ответ я сослался на работу

Герберта Саймона «The Sciences of the Artificial» (Науки об искусственном), изд. MIT Press. Саймон отмечает, что если рассматривать сложные системы, то те системы, которые мы можем понять, обычно имеют ряд общих характеристик. Обычно они иерархические. Они так или иначе делятся на уровни. Обычно в них огромное число повторяющихся элементов.

Саймон в свое время рассматривал это в терминах структурного повторения. Я хочу уточнить его наблюдение и сказать, что это не просто повторение. Мы видим в системе также повторение шаблонов конструкции. Мне довелось работать с великим множеством проектов едва ли не во всех областях, какие можно себе представить, и мне встречались совершенно неряшливые системы, но я видел и красивые системы. Для тех систем, которые красивее и проще, характерно наличие шаблонов проектирования, расположенных выше системы, охватывающих многие отдельные компоненты и обеспечивающих их исключительную простоту.

На самом деле, если посмотреть, чем заняты исследователи ДНК, то это будет поиск тех самых повторений. Мы высокомерно решили, найдя в ДНК «лишние» гены, что это мусор, оставшийся в результате эволюции, который не имеет никакого значения. При более глубоком рассмотрении оказывается, что этот мусор может быть совсем не мусором. Мы назвали это мусорной ДНК, потому что не поняли ее функций. Это проблема «бога пробелов», но я не буду в связи с этим мучить вас метафизикой. Когда стали разбираться с прекрасным, утонченным изяществом этих систем, там обнаружились удивительные шаблоны. Вот где я нахожу простоту и красоту.

Что вы имели в виду под «шаблонами проектирования, расположенными выше системы»?

Гради: Расскажу про одну систему с Уолл-стрит. Не буду называть ее. Мы там недавно проводили археологические изыскания, пытаясь раскопать, на каких решениях она основана. Это гигантская система. Миллионы строк кода на всех мыслимых языках, от ассемблера до самых современных. Если взглянуть на общую архитектуру системы, то заметны основные принципы, весьма обычные. В системе уделено внимание проблемам безопасности. Вы же не хотите, чтобы кто-то пришел и незаметно для вас стал состригать по одной тысячной процента с каждой операции. А когда у вас операций на триллион долларов в день, никто этого и не заметит.

Как бороться с такими вещами? С помощью очень простого бизнес-правила. Все, что изменяет состояние, должно находиться в базе данных в хранимой процедуре, чтобы невозможно было – потому что это проверяется с помощью сквозного контроля и ряда формальных меха-

низмов – невозможно было инжектировать код, изменяющий состояние. Вот что я имел в виду.

Шаблоны проектирования – это, в конечном счете, названия для сообществ классов, действующих совместно и гармонично. Глядя на отдельную строку кода, вы этого не увидите. В этом, кстати, проблема исследования архитектуры, потому что код – это еще не вся истина, и умения находить эти вещи и открывать шаблоны. Они часто скрыты в головах людей.

Тогда почти наверняка вы не сможете, взглянув на систему, понять ее, увидеть ее простоту или внутреннюю организацию, если вам не известно, какими были начальные ограничения, правила, конструктивные решения.

Гради: Совершенно верно. Приходится также выяснять путем логического вывода. Смотрите на кучу всех этих вещей, и начинают проступать шаблоны. Невозможно взять отдельный экземпляр и сказать: «О! Я вижу тут паттерн!» – потому что это противоречит самой природе шаблонов. Это одна из тех областей, где я стараюсь чаще использовать UML: для визуализации архитектуры уже построенных систем. Конечно, есть разработки «с нуля» («green field» – «новое месторождение»), но в целом большинство разрабатываемых в мире программ – это, говоря словами Криса Уинтера, «старое», или «коричневое месторождение» («brown field»). Звучит мерзко и плохо пахнет. Так оно отчасти и есть, но большинство создаваемых нами интересных систем – это адаптации или усиление уже существующих.

Может быть, стоит принять термин «разработка старых месторождений»?

Гради: Это было бы замечательно. В этом нет ничего плохого. Вот тут для меня вступает в игру UML, потому что он позволяет увидеть то, что код сам по себе не покажет.

Эти шаблоны могут стать заметными только на более высоком уровне абстракции. Наверное, рисуя диаграммы или создавая артефакты UML для существующей системы, вы испытываете чувство глубокого удовлетворения, когда удается объединить концепты.

Гради: Совершенно верно.

Вы (и многие другие участники интервью) указывали на ОО-разработку как на главный элемент правильной разработки. Насколько это принципиально?

Гради: Возможно, и странно услышать это от меня, но ООП является, IMHO, следствием, а не первопричиной. Имея возможность изучить многие сложные системы в различных прикладных областях, я при-

шел к выводу, что лучшие из них – самые полезные, самые красивые, самые изобретательные, самые-самые в любом другом отношении – в своей разработке имели ряд общих черт, а именно: стремление к четким абстракциям, разделение интересов и взвешенное распределение ответственности. Абстракция – это в значительной мере проблема классификации, а объектно-ориентированные механизмы особенно хорошо подходят для классификации окружающей действительности.

Лет 20 назад, на OOPSLA, Уорд Каннингем и Кент Бек впервые предложили шаблоны проектирования, основываясь на работе архитектора Кристофера Александера. Было ли это успехом – скромным успехом?

Гради: По моему личному мнению – не кого-либо из здравствующих или почивших, моей компании или кого-то из еще не родившихся, – работа Александера интересна и стала источником вдохновения для многих, проникнув в сферу ПО. Это принесло пользу небольшой группе людей, но не большинству сообщества. Несомненно также, что язык шаблонов, хотя я и встречаю его во многих местах, не приобрел того влияния, которое мог или должен был приобрести.

Говоря о «языке шаблонов» вы имеете в виду нечто близкое к тому, что говорил Александер, – «Давайте составим словарь, с помощью которого мы сможем обсуждать повторяющиеся в проектировании идеи»?

Гради: Абсолютно верно. Во многих отраслях, которые я затрагиваю, и во многих проектах, с которыми сотрудничаю, о шаблонах «банды четырех» часто знают из литературы, но практического применения в создании систем и архитектур они не находят. Есть места, где они сильно изменили картину, но в основной массе не пустили таких глубоких корней, как мне хотелось бы. Дело не в специфике. Дело в способе мысленного представления задачи.

Что могло бы исправить положение?

Гради: Что я обычно делаю, сотрудничая с проектом, так это помогаю им разработать язык шаблонов. Они должны понять. Они должны получить практическое представление о том, что такое шаблоны. Затем я подталкиваю их к тому, чтобы они находили шаблоны и давали названия тем, которые разработали сами для своих систем, и начали их документировать, делая шаблоны своим достоянием.

Какие-то из этих шаблонов могут отражать бизнес-правила и ограничения.

Гради: А какие-то – нет. Все зависит от особенностей предметной области. Какие уникальные способы добавления стоимости, какие новаторские методы решили их проблемы? Это и есть их шаблоны проектирования.

В настоящее время я работаю со спутниковым проектом. Там всего около 50 000 строк Ады. Там есть возможности для создания замечательных шаблонов, существующих в головах этих разработчиков. И проблема в поиске для них хороших названий, чтобы им было удобно общаться друг с другом. Они блестящие архитекторы, но пока вещь не назовешь, трудно говорить о ней, а вот назвав ее, можно манипулировать ею и передавать другим.

Возьмем APL. Там некоторые потоки данных не всегда транслируются непосредственно в другие языки.

Гради: Верно. Ада – во многих отношениях удивительный язык. Взгляните на его возможности, вошедшие в Java, C++ и другие языки. Встроенные средства параллельной обработки, механизмы исключений, механизмы генериков, абстрактные типы данных – Ада опередила свое время.

15

Perl

Поклонники Perl расшифровывают название языка как «Pathologically Eclectic Rubbish Lister» («патологически эклектичный язык для распечатывания чепухи») и «швейцарская армейская бензопила», бравируя лозунгом «Есть больше одного способа сделать это!». Создатель языка Ларри Уолл иногда говорит о нем как о связующем языке, который был задуман как золотая середина между оболочкой UNIX и Си, чтобы помочь людям в работе. Он построен на лингвистических принципах и конструктивных решениях UNIX (и может похвастаться, пожалуй, крупнейшим среди всех языков хранилищем библиотек – CPAN). Многие программисты с нетерпением ждут выхода долгожданной новой версии Perl 6 – языка, рассчитанного минимум на 20 лет.

Язык революции

Как вы определите Perl?

Ларри Уолл: Perl – это непрерывный эксперимент по включению в компьютерный язык некоторых принципов естественного языка, но не на поверхностном синтаксическом уровне, как в Коболе, а на гораздо более глубоком практическом уровне. Вот вам весь список базовых принципов естественного языка:

- Выразительность важнее легкости изучения.
- Детский лепет – это нормально для ребенка.
- Языком можно пользоваться, даже если вы не изучили его целиком.
- Часто есть несколько хороших способов высказать примерно одно и то же.
- Каждое высказывание в языке обретает смысл сразу в нескольких контекстах.
- Язык не знает, какой контекст для вас сегодня оптимален.
- Язык не придерживается какой-то конкретной парадигмы, исключающей остальные.
- Эффективное общение требует определенного уровня сложности языка.
- Семантические сети обычно плохо отображаются в ортогональные пространства.
- Множество сокращений; часто встречающиеся выражения должны быть короче, чем редко встречающиеся.
- Не все можно легко описать; кое-что описать трудно, но можно.
- В языках естественно употреблять такие слоты, как глаголы, существительные, прилагательные, наречия и так далее.
- Человек легко разрешает синтаксическую неоднозначность, если слоты очевидны.
- В языках естественно выделение паузами, интонацией, ударением, темпом и так далее.
- Языки используют местоимения, когда тема разговора очевидна.
- В идеале языки должны описывать решения, а не обсуждать собственные конструкции.
- Для успеха языка здоровая культура важнее, чем конкретная технология.

- Основная задача языка – общение с другими людьми, а не с самим собой.
- Акцент не беда, если вас понимают.
- У субкультур есть свои проблемы, и они часто порождают полезные диалекты или субъязыки.
- Люди могут научиться «сменять фрейм», столкнувшись с различиями в диалекте или акценте.
- Смена фрейма проходит успешнее, если можно сообщить, с каким субъязыком вы имеете дело.
- Любой язык со временем эволюционирует, чему невозможно помешать.
- При общении обычно допустим подход «чем хуже, тем лучше», но иногда предпочтительнее «чем лучше, тем лучше».
- Для восприятия письменных документов особенно важен исторический контекст.

Каждый из этих принципов оказывал глубокое влияние на Perl на протяжении многих лет. Безусловно, любой из них можно развернуть в абзац, главу или диссертацию. Даже если не принимать во внимание компьютерные языки, лингвистика остается огромной дисциплиной со многими специализациями.

Напротив, большинство этих принципов обычно игнорируется в других компьютерных языках. В силу разных исторических причин многие разработчики языков склонны допускать, что программирование – это деятельность, которая ближе к аксиоматическому математическому доказательству, чем к негарантированным попыткам установить межкультурное общение.

Конечно, есть и обратная тенденция: сосредоточенность на этих лингвистических принципах иногда приводила к тому, что я игнорировал важные идеи компьютерных наук. Сейчас мы работаем над восполнением некоторых таких упущений.

Что это были за идеи?

Ларри: Одна из главных точек, где мы несколько оплошали в начале разработки Perl – если честно, это я «оплошал», – в правильном определении областей видимости. Да, в Perl 5 есть лексические области видимости переменных, но во многих случаях область видимости устанавливается не так, как следовало бы. Кроме того, сохраняется масса глобальных переменных.

В Perl 5 также существенна роль дальнодействия (action-at-a-distance). В Perl сделана ошибка, аналогичная той, которую сейчас делают языки

вроде Ruby, разрешающие заниматься «monkey typing», т.е. залезать внутрь объекта и курочить его внутренности, что может привести к непостижимым действиям на расстоянии.

Мы поняли, что есть несколько правильных областей видимости. В классическом случае информация относится к объекту или к лексической области или к динамической области, но есть также область видимости файла, процесса, потока, типа, метакласса, ролей, прототипов, событий, грамматик, транзакций и чего угодно. Можно даже представить себе уровни приоритетов как необычные области видимости. У каждой точки в пространстве и времени есть нечто, естественным образом ассоциированное или связанное с ней, и возникает несоответствие, если это связано с другим местом. Вот это я медленно начинаю понимать. Некоторым кажется, что слишком медленно.

Другой принцип – понимание того, какие структуры данных можно изменять, а какие – нет. Это станет еще более важным в ближайшие годы из-за параллелизма: нельзя успешно справляться с параллельностью, если невозможно уследить, что и когда может меняться. Это как раз то, что Perl в прошлом не хотел признавать. Мы все считали изменяемым.

Иногда изменяемым даже в результате наблюдения.

Ларри: Да. Такой образ мышления годится для маленьких программ и наивных пользователей – он не противоречит ожиданиям новичков, которые не слишком искушены.

С другой стороны, он затрудняет переход к крупным программам, где нужно внимательно следить за тем, что можно изменять, а что – нет, различать общие и локальные данные. Этим вещам придается в Perl все большее значение, особенно в Perl 6.

В то же время мы хотим сохранить прежний дух и, где это возможно, скрыть претенциозные понятия, чтобы новички могли по-прежнему не обращать на них внимания. То, что они скрыты, не означает, что их нет: если Perl будет незаметно делать правильные вещи, то можно хотя бы надеяться, что если что-то пойдет не так, это обнаружится, и надеяться, что можно будет, если нужно, распределить работу по нескольким ядрам, не очень опасаясь, что порядок выполнения окажется неверным. Для этого нужно отслеживать зависимость между данными. А для этого необходимо знать, когда можно рассматривать указатель как число, а когда с ним нужно обращаться как с объектом.

Perl появился как набор инструментов для обработки текста и облегчения системного администрирования. Что он представляет собой сейчас?

Ларри: Сейчас это фактически две разные вещи. Во-первых, в виде Perl 5 это очень надежный образец того, с чего все началось: язык, связующий API, хорошо пригодный для обработки текста (дополненный огромным количеством расширений, любовно созданных невероятно доброжелательной культурой). Это причина широкого применения Perl в Сети: HTML – это текст, а людям нужно было писать HTML, в который вставлялись бы данные из разных источников, в том числе из баз данных. Для этого имелись готовые расширения или их легко можно было написать.

Но Perl – это еще и Perl 6, где мы попытались исправить все, что было плохо в Perl 5, не разрушив того, что в Perl 5 было хорошо. Мы понимаем, что это невозможно, но все равно стараемся. Мы полностью перепроектируем язык, сохраняя базовые конструктивные принципы.

Даже в нынешней незаконченной форме Perl 6, по мнению многих, является замечательным языком, а когда работа будет завершена, мы надеемся получить самоописательность и автопарсинг с помощью грамматик с высокой выводимостью и, таким образом, оптимальность для плавного развития в любом направлении, которое покажется желательным через 20 лет. Там будут регуляторы для настройки разных параметров, а также возможность отключить их, если они вам не нужны в силу выбора парадигмы для решения конкретной задачи.

По крайней мере, такие у нас мечты...

Как вы перешли от создания инструмента для обработки текста и упрощения жизни системных администраторов к созданию полного языка программирования? Это было намеренное решение или постепенный переход?

Ларри: Одно другого не исключает, и это хорошо, потому что я охарактеризовал бы процесс как намеренный плавный переход. Работать над языком очень интересно, и мне с самого начала было ясно, что я буду и дальше развивать его в соответствии со своими новыми потребностями. Но такой процесс обязательно должен быть постепенным, независимо от его преднамеренности, хотя бы по той причине, что должно пройти время, прежде чем выяснится, какие у вас возникли новые потребности.

С другой стороны, в какой-то момент я понял, что Perl – это не только простой способ делать простые вещи, но и возможность делать сложные вещи. Perl 2 мог работать только с текстовыми данными, поэтому я подумал: «Perl – всего лишь язык для обработки текста; если я научу его обрабатывать двоичные данные, куда это может завести?» Затем я понял, что в мире есть великое множество задач, в основном текстовых, которым нужно иногда работать с двоичными данными; если добавить

решение для этого класса задач, то область применения Perl заметно расширится, даже если обработка двоичных данных будет элементарной. В результате Perl 3 стал обрабатывать двоичные данные, и кто знает, куда это может завести?

Кроме того, я стал рассматривать это как продолжение моих прежних представлений о том, что Perl не должен иметь произвольных ограничений, которых было полно в ранних утилитах UNIX. Обрывать строку только потому, что в ней встретился нуль, немногим лучше, чем обрывать ее, потому что у вас слишком маленький буфер. Можно даже сказать, что процесс обобщения языка – это просто снятие разных произвольных ограничений, на том или ином уровне.

Что вам больше нравится – свобода или порядок? Вы предпочитаете иметь единственный способ достичь цели или тысячу таких способов?

Ларри: В этом вопросе немного смысла, если вы не определите, что имеется в виду под «способом» и «достичь», какие разрешены оптимизации и как можно комбинировать и переставлять различные параметры. Естественный язык похож на старосветский город, где улицы очень редко пересекаются под прямым углом и где обычно есть несколько более или менее удобоваримых маршрутов, чтобы добраться в нужное место. Очевидно, что если считать не только лучшие, но все возможные маршруты, их окажется гуголплекс или больше.

Даже в совершенно ортогональном городе, построенном по строго прямоугольной решетке, достичь нужного места можно бесчисленными разными способами, если не ввести то ограничение, что измерения упорядочены, как в математическом векторе пространственных координат, и вам нужен кратчайший путь. Но передвижение по городу человека нельзя описать простым вектором. Перемещаясь по городу или языку, вы оптимизируете свой путь в зависимости от многих внешних факторов. Единственного лучшего решения может не существовать или оно будет зависеть от времени суток. Может быть, вы хотите по пути побывать во всех парках или держаться подальше от реки. Или наоборот, поближе к ней.

Так что в ответ на ваш вопрос нужно сказать, что каждая из этих крайностей не оптимальна. Полагаю, что фрактальная размерность естественного языка несколько больше 1, но значительно меньше 1000. Если у вас действительно только один способ сказать что-либо, то вас можно заменить роботом-программистом. Если у вас действительно есть тысяча разных способов сделать что-то и разницы между ними не видно, то всегда можно придумать какой-то метод отсечения вариантов, чтобы свести их приемлемому числу.

Язык

Многие хвалят Perl за очень и очень хорошие средства обработки текста. Связано ли это каким-то образом с теми лингвистическими соображениями, которыми вы руководствовались при разработке языка?

Ларри: О, хороший вопрос. Так и хочется ответить на него «да», но это, пожалуй, была бы неправда.

Уликой станет то, что хотя было задумано, чтобы Perl работал на определенных уровнях, как это делают естественные языки, он был непригоден для парсинга кода Perl. (Для этого использовался yacc.) Я сильно подозреваю, что если бы ответ на ваш вопрос был положительным, то Perl 1 был бы гораздо ближе к тому типу парсинга, который сейчас задуман для Perl 6. Но этого не было. Можно назвать это психической фрагментацией, но для Perl 1 предполагалась гораздо более ограниченная форма обработки текста, чем та, которую осуществляет наш мозг в отношении естественного языка.

Несколько лет назад я просматривал комплект тестов для Perl 1, и даже тогда код проявлял характерные признаки Perl, даже учитывая, что многие понятия Perl, такие как контекст, развились не сразу. Какие неотъемлемые черты Perl планировались вами с самого начала?

Ларри: Конечно, понятие контекста было важным с самого начала, и в Perl 4 идея контекста была уже вполне проработана. Все же я думаю, что важность контекста осозналась мной лишь постепенно. Как лингвист, я всегда помнил о контексте: согласно тагмемике – одной из моих любимых лингвистических теорий, – огромное значение имеет множественность уровней контекста. Лексическая классификация слова совершенно отличается от того, как оно используется. Вы знаете, что «глагол» является существительным, даже если слово используется как глагол. Поэтому у меня всегда витала мысль, что каждую конструкцию можно использовать разными способами в зависимости не только от ее собственной структуры и типа, но и от ее семантического и культурного контекста. Думаю, в раннем Perl замысел контекста сильнее всего проявляется не в синтаксисе низкого уровня, вроде разницы между скаляром и списком, а в той идее, что, пытаясь выполнить задачу, вы программируете в контексте того, что вы хотите сделать за рамками программы. Поэтому полезно, когда в языке есть несколько способов выразить одно и то же, чтобы выбирать из них оптимальный в зависимости от внешних обстоятельств.

Иными словами, ваша программа должна скорее представлять контекст задачи, а не пытаться описать задачу в контексте вашего языка.

Ларри: Да, вопрос в том, что из этого главнее, вот и все. Обычно находится несколько способов представить конкретную задачу, особенно если есть несколько парадигм программирования. Одни более естественно подходят к одному классу задач, другие – к другому классу. Если ваша задача кажется вам чем-то вроде математического доказательства, то может подойти что-нибудь вроде функционального программирования, что-то более описательное, с универсальными, неизменными значениями – где концепты, с которыми вы непосредственно работаете, включают в себя неизменяемое состояние. Если ваша задача ближе к моделированию, то лучше представлять ее с помощью объектов, изменяющихся во времени. Для той задачи, которую вы пытаетесь решить, эти представления могут быть изоморфны, но разные парадигмы программирования могут потребовать, чтобы вы помещали изменяемое состояние в разные места. Где в объекте находится это состояние – очевидно, для того объекты и существуют. Не столь очевидно, где это состояние, в функциональном программировании: состояние неявно присутствует в стеке и в организации различных монад и вызовов функций.

Одна из форм контекста – это ваше представление о том, как на это смотрит программист.

Можно рассматривать это как ослабленную гипотезу Сепира–Уорфа. В ее сильную версию я не верю…

Вам было предопределено не верить в сильную версию.

Ларри: Да. Я выбрал предопределенность. А может быть, мой мозг устроен не лингвистически, и поэтому я часто думаю без посредства языка. В результате мне кажется, что язык не управляет моим мозгом. Как бы то ни было, меня устраивает слабая форма гипотезы: язык, который вы выбрали, чтобы что-то описать, конечно влияет на то, как вы его опишете. Поэтому если вы на самом деле хотите иметь один язык, импеданс которого хорошо подходит для задач разных типов, то вам нужен язык, который не будет вынуждать вас думать каким-то одним особым образом.

Есть несколько тонких контекстов: однострочник UNIX, которым можно записать полезную, работающую программу Perl; контекст сценария оболочки, в котором у вас будет нечто, похожее на сценарий оболочки, со всей мощью Perl 5; контекст отдельной программы CGI для одной страницы. В какой-то мере это разные контексты для выполнения работы. Оба воспринимаются как явно Perl и как явно данный стиль. Это однострочник Perl, а это сценарий оболочки, написанный на Perl.

Ларри: В лингвистике это называется «прагматики» – один-два шага от семантики в сторону социологии. Некоторые лингвисты склонны сосредоточиваться на фонологии низкого уровня или синтаксисе. Такую же узость взглядов можно обнаружить во многих компьютерных языках: разработчики не подумали о том, как высказывания используются в практической работе. Если взять однострочники и сценарии оболочки, то по существу то же самое можно наблюдать в естественных языках. Есть фразы, которые можно произнести где-нибудь на автобусной остановке, и это общение с помощью однострочников. В английском есть множество лаконичных фраз. С другой стороны, есть самые разнообразные литературные жанры, которые можно сравнить с парадигмами программирования. Есть разные уровни дисциплины, которую можно соблюдать при сочинении литературного произведения. У этих жанров есть свои правила, которые разрешается время от времени нарушать. И иногда вы по глупости нарушаете их, но сам язык не пытается навязать вам какой-то определенный стиль.

Естественные языки нейтральны в этом отношении. Язык подчинен поэту: это художественный материал, из которого художник пытается создать нечто более высокое. Это более высокое управляет всем процессом, как и должно быть.

В определенном смысле, естественные языки отличаются чрезвычайной покорностью. Они не указывают вам, как вы должны говорить. В начальных классах учитель указывает вам, как нужно говорить, но большинство не обращает на это внимания, и это правильно.

Для компьютерных языков есть эквивалент такого учителя, и для некоторых типов высказываний нужно следовать правилам, если только вы не нарушаете их с определенной целью. При всем при том компьютерные языки должны быть понятны компьютерам. Это налагает дополнительные ограничения. В частности, здесь нельзя просто использовать естественный язык, потому что, говоря на естественном языке, мы, как правило, предполагаем высокий уровень интеллекта на приемной стороне, а тот, в свою очередь, предполагает высокий уровень интеллекта на передающей стороне. Если вы предположите такой же уровень интеллекта у компьютера, то будете глубоко разочарованы, потому что мы пока не умеем программировать компьютеры до такого уровня.

Perl – это первый постмодернистский компьютерный язык, но компьютеры очень плохо понимают иронию.

Ларри: Воистину. Они совсем не понимают, когда им следует задать вопрос. Это оттого, что они не способны испытывать неуверенность. Заметьте, есть множество людей, которые тоже не чувствуют неуверен-

ности, но разница в степени. Компьютеры очень быстро движутся по служебной лестнице, достигая уровня своей некомпетентности.

Вы часто высказывались в том смысле, что разработчики языков должны больше слушать лингвистов, чем математиков, потому что лингвисты лучше понимают, как нужно общаться с людьми.

Ларри: Скажем так: они лучше знают, как общаются обычные люди.

Математики знают, как общаться друг с другом, но из этого не следует, что мы должны считать их обычными людьми; наверняка математики сумеют, опираясь на теорию множеств, доказать, что они обычные люди. Все же я считаю, что лингвисты уделяют несколько больше внимания психологии и прагматике, чем это обычно делают математики. Поэтому лингвисты могли бы помочь компьютерам поумнеть в таких вопросах.

Вы согласны с тем, что концепция небольшой, строгой и доказуемой модели языка программирования может оказаться нежизнеспособной на практике?

Ларри: Всегда найдется группа людей, склонных мыслить крайне узко. Если вам удастся найти таких людей, то они будут успешно применять такой язык в решении того класса задач, для которого он предназначен, избегая катастроф.

Не факт, что верно противоположное. Достаточно широкий язык может по ряду причин оказаться неудачным. Возможно, его слишком трудно будет реализовать. Возможно, слишком трудно будет изучить ту его часть, которая необходима, чтобы начать практическую работу. Думаю, это две главные причины, мешающие распространению больших языков.

Тем не менее все изобретенные к настоящему времени компьютерные языки далеко уступают по сложности естественным языкам. А люди наглядно показывают – пожалуй, кроме американцев, – большинство людей, не говорящих по-английски, явно демонстрируют стремление знать несколько языков.

Я пытался изучать несколько естественных языков, и они действительно трудны. Лексическая сложность, странные грамматические правила, особые в каждом языке, разный порядок, в котором языки заставляют думать, возможность сказать в одном языке то, чего нельзя сказать в другом. Выучить естественный язык трудно. Вопрос в том, можно ли создать достаточно сложный язык, чтобы люди захотели преодолевать сложность изучения. Можно ли сделать такой язык, чтобы для него получился полезный пиджин? Мы же видим, как возникают англо-туземные и креольские языки, когда люди достаточно мотивированы, чтобы найти общий язык.

Можно ли перенять такую динамику? Можно ли сконструировать язык, который не более сложен, чем это требуется в действительности? Это хорошие вопросы, на которые все разработчики языков отвечают по-своему. В Perl есть иерархия пространств имен, но в других языках бывают плоские пространства имен, и все помещается в один словарь. Пожалуй, это ситуация для английского. Особенно если вас зовут Webster или Johnson.

В английском есть понятие жаргона и формы имплицитной коммуникации.

Ларри: Конечно, есть. С этим связано понятие лексической видимости, играющее важную роль для управления разновидностями языка в Perl 6. В каждой точке лексической области видимости очень важно знать, на каком конкретно языке мы говорим. Не столько потому, что это нужно человеку: нужно, но человек может сам это сообразить. Компьютер, скорее всего, окажется не столь умен, поэтому совершенно необходимо, чтобы компилятор знал, какой язык он анализирует в данный момент.

Если мы не позволим ошибиться компилятору, то и человек, надеюсь, тоже не ошибется. Разные лексические области, разные отрывки текста в литературе – разные фреймы, как говорят в психолингвистике, – используют разные языки. Люди постоянно сменяют фреймы. Они знают, когда можно говорить небрежно, а когда – правильным языком. Они постоянно перестраиваются, не думая об этом. Это часть инструментов естественного языка.

Как вы решаете, что для вашей задачи нужен особый инструмент или язык?

Ларри: Граница здесь очень нечеткая: понятие отдельного инструмента сливается с понятием набора инструментов, и язык может рассматриваться как набор инструментов, поэтому провести различие нелегко. Швейцарский армейский нож – это один инструмент или несколько?

Вероятно, важнее, как инструменты из набора работают вместе. Видимо, швейцарский армейский нож – это удобный набор инструментов, но попробуйте одновременно воспользоваться несколькими инструментами из набора!

Я бы сказал, что язык отличается от инструмента или даже набора инструментов тем, насколько он *может* быть общим (хотя, конечно, есть специальные языки, а есть универсальные). Хотя язык можно рассматривать просто как инструмент, языки обладают великолепными возможностями для комбинирования идей неожиданными способами. Если для вашей задачи нужны такие возможности композиции и ей не слишком помешает линейность языка, то определение нового языка

может оказаться хорошим решением для вашей конкретной задачи или какого-то ее существенного подмножества.

Кроме решения этой первой проблемы, ваш язык может также стать для вас механической мастерской, где вы с легкостью изготовите другие инструменты, и если это так, вы значительно облегчите себе жизнь и на будущее. Иногда в расчете на это можно пойти путем изобретения языка, даже если это не самый быстрый способ решения первой задачи, для которой вы это делаете. Но если вы по-настоящему ленивы, то сообразите, что ваши лишние усилия окупятся при дальнейшем использовании языка.

Как меняется язык, когда он покидает область своего специального применения? Ранние версии Perl вполне можно считать очищенным диалектом UNIX, играющим роль API для склеивания разных вещей. Как меняется язык, когда он переходит от таких специфических вещей к чему-то более общему?

Ларри: Как я считаю, в таком проблемно-ориентированном языке есть разные конструкции, которые выглядят очень естественно, но все они определены неким специфическим образом. Они кажутся более разумными, чем на самом деле.

Это способ, которым человеческий мозг пытается устанавливать связи с объектами?

Ларри: Да, и когда это происходит, люди поспешно делают выводы, которые могут быть ошибочными. В результате появляется длинный список «часто задаваемых вопросов». Если взглянуть на FAQ для Perl, то там обнаружится много вопросов, которые можно рассматривать как результат такого «ложного обобщения». Пытаясь сделать язык более универсальным, нужно посмотреть на все такие места и спросить себя: «Почему люди делают такие обобщения? Должен ли язык это поддерживать? Какими минимальными изменениями внутреннего механизма можно добиться, чтобы результат стал таким, каким его ждут, а не таким, как сейчас?»

Это как раз тот процесс, который, как вы знаете, применяется в разработке Per 6. Это наш взгляд, установившийся несколько лет назад.

Можете привести пример?

Ларри: В Perl 5 `$var` – скаляр, `@var` – массив, а`%var` – ассоциативный массив. Новички часто делают из этого вывод, что индексированные элементы нужно записывать как `@var[$index]` и`%foo{$key}`, но в силу исторических причин в Perl 5 это неверно. Документация исхитряется объяснить, почему все не так, как того ждут. В Perl 6 мы решили, что лучше исправить язык, а не пользователя.

В Perl 5 многие функции принимают по умолчанию аргумент $_ (текущее значение), если аргументы не заданы, и, в сущности, нужно запомнить список функций, которые так себя ведут. Иначе вы можете ошибочно распространить это правило на другие функции. В Perl 6 мы решили, что не будет функций с таким поведением по умолчанию, чтобы устранить опасность ложных обобщений. Вместо этого появился простой и явный синтаксис для вызова метода с текущим значением.

Во всех случаях, когда язык требует, чтобы вы запомнили некий произвольный список, кто-нибудь ошибется и будет считать элемент входящим в список, когда его там нет, и наоборот. Иногда это постепенно проникает в вашу конструкцию. Когда в UNIX изобрели синтаксис регулярных выражений, в нем было всего несколько метасимволов, которые легко было запомнить. Когда возможности поиска по шаблону стали расширяться, в качестве метасимволов добавлялись все новые и новые символы ASCII либо использовались ранее запрещенные более длинные их последовательности, чтобы сохранить обратную совместимость. Неудивительно, что в результате возникла неразбериха. Во многих программах можно встретить ошибочное обобщение; испуганные пользователи защищают обратной чертой все символы в регулярном выражении, потому что не помнят, какие символы в действительности являются метасимволами. В Perl 6 при переработке синтаксиса поиска по шаблону мы поняли, что большинство символов ASCII уже стали метасимволами, поэтому мы зачислили в метасимволы все символы, кроме букв и цифр, чтобы программисту нужно было меньше помнить. Списка метасимволов больше нет, а синтаксис стал гораздо понятнее.

Конструкцию Perl я описываю словом «синкретичная». Вы берете удачные вещи то здесь, то там и пытаетесь объединить их. Как вы балансируете между синкретизмом с одной стороны и общностью идей и согласованностью идей и функций – с другой?

Ларри: Плохо, как считают многие.

Незавидная участь разработчиков языков состоит в том, что, пытаясь примирить одно с другим, они могут полагаться лишь на интуицию.

История предоставляет мало возможностей для экспериментов.

Ларри: В целом, согласен. Что касается Perl, то у нас была редкая возможность провести успешный эксперимент под названием Perl 5, который позволил нам провести еще один эксперимент под названием Perl 6.

На этот раз мы активно стараемся установить новые пропорции исходя из накопленного опыта. Конечно, мы принимали много плохих решений – если не в отношении функций языка, то в выборе поведения

по умолчанию для отдельных функций. Теперь мы хотим исправить прежние ошибки и сделать новые.

Интересно, что из этого получится. Когда мы учили людей работать с прежними версиями Perl, приходилось много объяснять, почему все должно быть так, как оно есть. Теперь мы должны вернуться и сказать: «Тогда наш ход мыслей был неправильным. Это значит, что сейчас ваш ход мыслей – неправильный». Интересная культурная проблема: можно ли и как вывести людей оттуда, куда вы их завели. Пользуясь библейской метафорой, вы завели их в Египет, а теперь хотите вернуть их в Землю обетованную.

Кто-то последует за вами, а кто-то предпочтет горькую зелень.

Сообщество

Вы всегда поощряли участие сообщества в разработке и реализации. Это было так необходимо для вас? Это отражение вашего стиля работы или вашего эстетического чувства?

Ларри: Я уверен, что это сочетание ряда факторов.

Конечно, мотивом создания сообщества служит желание услышать как положительную, так и отрицательную оценку своей работы. Лучше положительную, но отрицательная тоже полезна.

С точки зрения лингвистики, язык, на котором говорит мало людей, имеет слабые перспективы. На начальном уровне было достаточно очевидно, что сообщество пользователей явно способствует жизнеспособности проекта.

Кроме того, я просто хотел, чтобы результатами моей работы пользовалось больше людей, потому что я люблю помогать людям. По мере своего развития проект становился достаточно крупным, и мне также потребовалась помощь. Где-то к концу Perl 4 я заметил, что все расползается по разным направлениям и разные люди компилируют свои версии исполняемого модуля Perl. Стало очевидно, что Perl нуждается в механизме модульного расширения, и что за разные модули должны отвечать разные люди.

Вскоре после выхода Perl 5 выяснилось, что даже сам Perl слишком велик, чтобы его интеграцией (не говоря о разработке) длительное время мог управлять один человек, не истощив при этом все свои силы. Эту работу нужно было передать другим. По существу, на ранних этапах Perl 5 я понял, что должен научиться делегировать свои полномочия. Увы, моя большая проблема – отсутствие управленческой косточки.

Я не умею делегировать, поэтому я даже делегировал делегирование, что прошло вполне удачно. Я не заставляю кого-то что-то делать: появились другие люди, которые устремляют свои управленческие усилия в нужном направлении, предлагают списки задач и координируют работу остальных участников. Любопытно было заметить, что несмотря на мою неспособность к микроадминистрированию – или благодаря ей, – сообщество в некоторых отношениях оказалось более крепким.

Возможно, у меня все-таки есть одно ценное качество руководителя: я не задумываясь выкладывал людям, что они должны сделать, но обычно в таком абстрактном виде, что они совершенно не понимали, о чем я говорю.

Вы сказали, что вам неинтересно быть менеджером и у вас нет для этого соответствующих способностей. Несмотря на это вы по-прежнему во главе сообщества Perl. Это для вас дело всей жизни или вы просто хотите, чтобы люди договаривались между собой, и делаете для этого все возможное?

Ларри: Видимо, я хочу, чтобы люди договаривались между собой не первобытными способами, как это произошло с Родни Кингом. Моя задача, как мне кажется, в том, чтобы по мере сил исправлять отклонения от правильного курса. Мне редко приходится вмешиваться и принимать решительные действия: был лишь один случай, когда мне пришлось позвонить некоему члену сообщества Perl по телефону и резко одернуть его.

Надо сказать, это подействовало.

Конечно, мне кажется, что если бы я был более искусен в руководстве, то смог бы справиться с некоторыми губительными течениями, которые действительно проникли в отдельные части культуры Perl. В городе Perl есть кварталы, куда не следует заходить с наступлением темноты, но даже самый великодушный, всеведущий и всемогущий правитель не должен решать, что дозволено или не дозволено делать в каждый данный момент.

Пожалуй, даже Всевышний не пытается действовать так по отношению к нам. Как говорит Он в «Бандитах времени», «Думаю, это одно из проявлений свободы воли».

Одним из наивысших достижений Perl 5 является CPAN, и мне кажется, что это главный вид расширяемости. Эта вещь появилась в результате определенных конструктивных решений или счастливого случая?

Ларри: Ну, как обычно с такими историческими вопросами: «да, это моя работа» и «нет, я тут ни при чем».

Из конструкции системы модулей не было очевидно, что разные люди смогут предлагать модули и публиковать их, и даже что возможны хранилища для таких модулей. В других языках уже существовали хранилища для многократно используемого ПО разных типов. Чего я, конечно, не мог предвидеть, так это масштабов данного явления и совершенно невероятной широты применения Perl.

С самого начала я стремился обеспечить в Perl поддержку как можно большего количества API, будь то оболочка, или переменные окружения, или непосредственно операционная система, или терминал. Я был первым, кто ввел парсер для XML. Мне всегда и на разных этапах было важно, чтобы Perl был языком, который пытается не включить в себя все, что можно, а соединиться с внешним миром всеми возможными способами. В этом суть связующего языка. В противоположность такому языку, как Icon, где была сделана попытка определить все внутри изолированным образом.

Не хочу чрезмерно упрощать: в той или иной мере все языки изобретают заново по крайней мере часть колеса.

От вас всегда будут требовать, чтобы ваши решения на все 100% основывались на Perl, или на Java, или на чем-то еще. Так проще, например, тестировать и конфигурировать. Так легче распространять приложения и нанимать программистов. С другой стороны, если ваш язык и ваша культура полностью соответствуют этому принципу, то это явно вредоносная форма высокомерия.

Должна существовать гармония. Perl всегда готов был ошибаться, поддерживая слишком много API, а не слишком мало. Но лучше всего поддерживать оба эти подхода.

Я встроил в язык философию поддержки связи и прагматики. Я не ожидал, что такое новое явление, как Всемирная паутина, достигнет такого масштаба. Perl достиг успеха, на который я никогда не рассчитывал, но в какой-то мере он был в нем заложен.

Вы намеренно заложили механизмы, позволившие этому счастливому случаю осуществиться или при возможности воспользоваться им.

Ларри: В смысле некоторого определения намеренности, которое может относиться к левому или правому полушарию, задней или лобной коре, или среднему мозгу. Обычно довольно рассеянного по мозгу. Значительная часть – «внемозговая» активность, поскольку с самого начала значительный вклад в Perl был внесен не моими, а чужими мозгами.

Хранилища библиотек имеются во многих языках, но у CPAN есть то преимущество, что его разработчики (Ярко, Андреас и другие) построили инфраструктуру, которая поощряет разработку, не ограничивая

ее. Вы способствовали развитию каких-то характеристик языка или сообщества, сделавших это возможным?

Ларри: Я не могу взять на себя ответственность за правильный выбор авторов CPAN (как и за неправильный), но мне кажется, что CPAN – это золотая середина для применения Закона Старджона («90% чего угодно – полная чушь»). Особенно когда вы создаете прототип чего-то нового, легко переусердствовать, пытаясь уйти от этих 90% кода, а в результате потеряв и большую часть остальных 10%. Конечно, мы видели, как этот дрянной код развивается и становится не столь уж плохим, поэтому иногда оправданно хранить спокойствие и считать, что чем хуже, тем лучше. А некоторые просто долго запрягают. Все мы учимся по ходу дела.

Что касается того, как это стало возможным в языке, то это, вероятно, было главной задачей при проектировании Perl 5: дать каждому возможность расширять язык через модули. Если взглянуть назад с точки зрения Perl 6, то я портил конструкцию модулей Perl 5 разными способами, но она была Достаточно Хороша (Good Enough), и результатом стал CPAN. Думаю, что время от времени я поощрял такие попытки со стороны сообщества, хотя и в самом общем виде. В целом, однако, все сводится к тому, что большинство разработчиков Perl отличается невероятной готовностью помогать другим. Я только немного помахал в начале флагом, чтобы процесс пошел, а те, кто заглянул сюда, остались, потому что нашли тут родственные души. И я с удовольствием вижу, что дух сотрудничества распространяется и на другие сообщества.

Случалось ли, что вклад сообщества вызывал у вас удивление?

Ларри: Не знаю, было или не было что-то такое в действительности. Возможно, меня несколько удивила культура обязательной строгости и предупреждений в целом – люди хотели больше дисциплины, чем им предлагалось по умолчанию. Это оказалось настолько приемлемым для культуры, что мы просто решили встроить это в Perl 6. Это меня несколько удивило.

Самое большое удивление постигло меня, когда мы стали перепроектировать Perl для версии Perl 6. Мы запросили предложения в виде RFC (Request For Change), и я рассчитывал получить их штук 20. Получено было 361. Частично удивление было вызвано тем, сколько было вложено труда, – примерно в 15 раз больше, чем я рассчитывал. Двойной удар был нанесен количеством разных вещей, которые, как казалось людям, можно исправить независимо от остального, чего на самом деле нельзя было сделать. Так что потребность в системной реконструкции оказалась глубочайшим удивлением, которое я получил от сообщества за 20 лет. Меня также удивил (в хорошем смысле) успех некоторых хаков

в культуре. Первоначальная уловка с двойным лицензированием для Perl 3 поставила Perl в очень хорошее положение как среди хакеров, которые боятся корпоративной культуры, так и среди корпоративной культуры, которая боится хакерских сообществ.

Оба сообщества нашли успокоение в таком подходе, причем мне не пришлось никому навязывать решение того, какой же лицензии он следовал, – GPL или Artistic License.

Я очень редко встречал тех, кто действительно выбрал бы одну из них.

Ларри: Да. Это была квантовая суперпозиция лицензий, чего люди просто не заметили. Пожалуй, неприятно меня удивило то, что мы породили для этого термин «двойственное лицензирование», которое впоследствии стали воспринимать как необходимость для пользователя выбрать ту или иную лицензию. Никогда не удается придумать термин так, чтобы в нем видели тот смысл, который вы в него вкладываете. Вечно возникает разрыв между тем, что хочется получить, и тем, что получаешь на самом деле.

Эволюция и революция

Какой у вас подход к проектированию и разработке: эволюционный или революционный?

Ларри: Я в каком-то смысле медведь с опилками в голове, поэтому предпочитаю эволюционный подход, когда пишу программы. При разработке программы на Perl я обычно делаю в ней изменение и запускаю, потом делаю новое изменение, и цикл длится, наверное, секунд 30. На отладку у меня уходит мало времени, потому что обычно сразу становится ясно, правильным или неправильным было последнее изменение. Время от времени я занимаюсь рефакторингом, но даже тогда подход остается эволюционным: вношу изменение и убеждаюсь, что ничего не изменилось.

Что касается конструирования языка, то я всегда придерживался аналогичного подхода: действовать эволюционным методом, но наращивать скорость мутаций, поэтому если сравнить два мгновенных снимка, сделанных с достаточным промежутком, изменения покажутся революционными.

Любителям UNIX Perl 1 казался радикальным отходом от awk, sed и оболочки, но на самом деле в тот момент Perl впитал в себя значительную часть культуры UNIX, чтобы не оттолкнуть пользователей. Для каждого нового языка встает проблема перехода на него пользователей, поэтому новые языки часто осуществляют широкие заимствования из уже существующих. (Впоследствии нам пришлось пожалеть о некото-

рых заимствованиях; в частности, синтаксис регулярных выражений становился с годами все противнее, и в Perl 6 мы надеемся это исправить.)

Perl 5 многим казался революцией в сравнении с Perl 4, но на самом деле реализация прошла через несколько промежуточных версий, невидимых для окружающего мира. В какой-то момент одни коды операций интерпретировались старым «стековым» интерпретатором, а другие – новым «бесстековым». Perl 5 был также очень консервативен в отношении обратной совместимости: фактически, он до сих пор правильно выполняет большинство скриптов, написанных на Perl 1.

В Perl 6 мы наконец-то пошли на значительное нарушение обратной совместимости, как бы «отбросив прототип» и стремительно развивая синтаксис и семантику, в то же время стараясь сохранить то «ощущение», которое всегда было характерно для Perl. Но на этот раз я лучше справился с привлечением сообщества к процессу инкрементной реконструкции.

После объявления о начале проекта Perl 6 мы получили те самые 361 RFC, и в большинстве из них предполагалось инкрементное изменение Perl 5 и никаких других изменений. В некотором смысле Perl 6 оказывается просто результатом суммирования, упрощения, унификации и рационализации этих инкрементных предложений, но действительный скачок от Perl 5 к Perl 6 несомненно покажется революционным каждому, кто не участвовал в процессе разработки. Тем не менее большинство программ на Perl 6 должны казаться вполне аналогичными тому, что вы писали на Perl 5, потому что основной ход мыслей примерно тот же. В то же время Perl 6 позволит гораздо легче перейти в функциональный или ОО-режим мышления. Кому-то это покажется революцией.

По-моему, представление о революции возникает, когда люди не замечают всех промежуточных ступеней. Perl разрабатывается так, чтобы помочь людям пройти промежуточные ступени как можно стремительнее, чтобы они могли почувствовать себя революционерами, а это увлекательно.

О какой революции речь?

Ларри: Я говорю здесь о личных революциях. О революции, которая случается, когда кто-то незнакомый с Perl подходит к Perl-программисту и говорит: «Мне кое-что нужно, но я не знаю, как это сделать». «Так это просто», – отвечает программист и пишет маленькую программу, очевидную, быструю и делающую нужное дело, и человек говорит: «Круто!» Когда кто-то говорит «круто!», это маленькая революция.

В известном смысле, между эволюцией и революцией все развивается непрерывно, и все зависит от того, насколько громко вы скажете «круто!» – или «дерьмо!», если вы из правящего класса. При хорошей революции чаще слышится «круто!», чем «дерьмо!».

Я вполне уверен, что бывают хорошие революции. Возможно, это также связано с теологией, по крайней мере на личном уровне. Я верю, что при надлежащем внешнем воздействии люди могут резко и быстро менять свои взгляды.

Хотят они того или нет.

Ларри: Да, это как разница между современными учеными и греческими философами, пытавшимися вывести все из первопричин. Они извлекли какие-то крупицы знания, но без эмпирической проверки не бывает непреднамеренных научных открытий, когда по счастливой случайности что-то падает вам на голову, заставляя полностью изменить ход мыслей, который незадолго до того вполне вас удовлетворял.

Например, Perl 5 в Сети.

Ларри: Perl 5 в Сети. А может, и многие промежуточные формы жизни, которые выползали на землю или заползали в норы, когда падали астероиды, или в других непредвиденных обстоятельствах, с которыми сталкивались наши предки. Есть общие принципы, для которых вы должны быть открыты: мелкие последовательные усовершенствования и крупные прозрения.

Эти большие прозрения индуктивны? Как лямбда-исчисление, где от четырех-пяти отдельных принципов путем рассуждений и дедукции выходишь в мир полноты по Тьюрингу.

Ларри: Да, но действительность часто наносит нам удары непредсказуемого масштаба. Допустим, я оптимизировал температуру своего тела для жизни в умеренном климате, а тут вдруг падает астероид огромного размера. Индукция полезна для постепенных открытий. Она может оказаться бесполезной для подготовки к событиям, которые выбьют большинство исходных оснований у вас из-под ног. Вот когда вам потребуется большой генофонд. Гены – это инструменты из вашего ящика, и их нужно много, особенно среди астероидов.

Вы хотите предоставить людям инструменты, с помощью которых они смогут приспособиться к новым обстоятельствам, оказавшись в такой ситуации?

Ларри: Это прежняя история о том, что меняется и что не меняется. Индукция основана на предположении, что ваши исходные посылки не меняются.

Индукция или дедукция?

Ларри: То и другое. Думаю, это две стороны одной и той же монеты, причем иногда монета встает на ребро. Теория вероятности считает, что вероятность этого равна нулю. Но это не так: у меня самого это получалось. Это было совершенно удивительно. В детстве я играл в футбол с соседом и мы бросали монетку. Он сказал: «Загадывай». Я сказал: «Ребро». И она стала на ребро, потому что ее зажало между жесткими листьями травы. Иногда контекст побеждает теорию вероятности.

Думаю, всю свою жизнь я загадывал «ребро» и часто оказывался прав.

На чем основана ваша идея возможности нескольких конкурирующих или сотрудничающих реализаций Perl 6?

Ларри: Для этого есть несколько возможных оснований. Мы уже говорили, что необходим большой генофонд. Для здорового генофонда требуется широкий обмен генами, а это может быть интересно. Другая причина в том, что несколько разных реализаций не позволят соврать друг другу. Естественно, что разные люди по-разному прочтут спецификацию, и если в ней есть неоднозначности, они, скорее всего, будут обнаружены. Тогда начнутся переговоры между разными реализациями с целью выяснить, что же действительно говорится в спецификации.

Похоже на то, как родители говорят ребенку: «Подели пирожное пополам, а потом твой брат выберет тот кусок, который ему понравится».

Ларри: Да. Это попытка повесить часть проектной работы на создателей реализации, что просто необходимо для такого слабого проектировщика, как я.

Другая сторона стратегии нескольких реализаций связана с тем, что у людей бывают разные интересы. Они сначала захотят сделать прототипы для разных частей реализации. Чтобы не получилось так, что вы напишете половину проекта, после чего обнаружите, что другую половину очень трудно реализовать при сделанных вами допущениях, потому что никто никогда и не пытался это сделать, лучше, чтобы разные люди поработали с прототипами разных частей проекта и рассказали друг другу о встреченных ими опасностях.

Как раз недавно мы видели это на примере SMOP, где с реализацией пока недалеко продвинулись, но очень детально обдумывается, как списки, сигнатуры и прочие важные понятия будут сосуществовать на нижнем уровне и как должны действовать семантически отложенные вычисления, итераторы и массивы.

Это типичное в науке упрощение: мы отвлечемся от остальной части проблемы и изучим один ее конкретный аспект. Думаю, это очень важ-

но, чтобы заставить нас продумывать проект в тех местах, которые мы еще не продумывали.

Из-за того, что они должны стать полными реализациями, или из-за того, что кто-то делает прототипы отдельных частей?

Ларри: Мне это почти безразлично. Если кто-то делает прототип части реализации, ему решать, насколько далеко он хочет идти: сколько у него сил, сколько еще разработчиков он может привлечь к проекту. Это еще один пример того, как я делегирую делегирование другим. Открыт новый континент, и нужно исследовать его во всех направлениях.

Не отдавая предпочтения какой-то одной из будущих реализаций, вы поощряете эксперименты.

Ларри: Да, мы поощряем алгоритм затопления, который может быть не вполне приемлемым, скажем, для корпорации. Там у вас проект с одной главной целью, фиксированными среднемесячными расходами и точной датой завершения. Но у нас есть замечательный параллельный процессор под названием «сообщество open source», и алгоритмы затопления лучше масштабируются для параллельного оборудования, чем для последовательного. Но это не примитивный алгоритм затопления, это больше похоже на колонию муравьев, разыскивающую себе пропитание. Так что в действительности мы оптимизируем процесс для того движка, на котором будем работать. Это не симметричный мультипроцессор, скорее, кластер Beowulf хакеров. У хакеров архитектуры тоже разные.

Общая память тоже составляет проблему, особенно когда состояние в такой степени неизменяемо.

Ларри: Не только память: сообщество open source – это архитектура, где неоднородно *все*, но я думаю, что для нас важнее его сильные, а не слабые стороны. Для муравьев проблема общей памяти не кажется острой.

Накладывает ли какая-либо конкретная реализация Perl 6 ограничения на то, что происходит на более высоких семантических уровнях?

Ларри: Даже если вы станете создавать новояз, никакой язык не может осуществлять полный контроль надо всем, и реализации относятся к тем вещам, которыми язык не может управлять полностью. В той мере, в какой мы пытаемся осуществлять контроль, определение языка действительно сводится к тому, что мы решили включить в комплект тестов или убрать из него, и мне кажется, что такого рода решения должны приниматься главным образом в результате переговоров среди тех, кто осуществляет реализацию, в которые изредка может вмешиваться создатель языка. Это одна из причин, по которым мы считаем

важным иметь несколько реализаций: они помогают друг другу сглаживать шероховатости.

Есть ли у вас какие-то опасения относительно доступности ресурсов?

Ларри: Думаю, разные точки сосредоточения усилий во всяком случае могут самоограничиться в отношении разумной степени масштабирования. Когда в команде оказывается больше шести или двенадцати программистов, она все равно делится на подпроекты. Когда люди начинают изобретать одно и то же колесо, не видя, что делается вокруг, это становится только тратой ресурсов.

Если они изобретают колесо, но по-разному – что ж, это интересный эксперимент по расширению генофонда. Конечно, у любого подхода есть неэффективные стороны. Они могут амортизироваться одновременно всем миром программистов или сказаться позднее, потому что вы не могли сделать сразу все, но как бы вы ни взялись за проект, неэффективность всегда обнаружится.

Некоторые утверждают, что при распространении неэффективности по всему кругу хакеров больше шансов на скорейшее ее исправление, чем при последовательном выпуске. При этом всегда предполагается, что есть достаточно хакеров, способных заняться проблемой, что проблема допускает распараллеливание, и что можно организовать общение добровольцев друг с другом.

Добровольцам не скажешь, что они должны делать. Можно, но бесполезно.

Ларри: Это еще одна большая проблема. Они все равно будут делать то, что хотят. Поэтому, как говорится, если не можешь исправить ошибку, скажи, что так и было задумано.

Мне понравилось ваше высказывание: «Говорят, „чем хуже, тем лучше“, но мы надеемся пройти с Perl 6 еще один цикл „чем лучше, тем лучше“».

Ларри: Верно. Я еще раз поставил на «ребро». Надеюсь на ваше нетерпение.

16

PostScript

PostScript – это конкатенативный язык программирования, чаще всего применяемый для описания документов в издательских системах. PostScript был разработан Джоном Уорноком и Чарльзом Гешке, основавшими в 1982 году компанию Adobe Systems. В 1985 году Apple выпустила LaserWriter со встроенным PostScript, сделав возможной допечатную подготовку на компьютере. PostScript быстро стал стандартом де-факто для обмена документами. Впоследствии его сменил в этом качестве его последователь PDF.

Проектирование на века

Как вы определяете PostScript?

Чарльз Гешке: PostScript – это язык программирования, основная задача которого обеспечить описание на высоком уровне содержимого (печатных) страниц независимым от устройства образом.

Джон Уорнок: PostScript – это интерпретируемый язык программирования, который моделирует простую стековую виртуальную машину. Помимо обычных операторов, имеющихся в большинстве языков программирования, PostScript располагает богатым набором операторов для вывода изображений, графики и шрифтов. Приложение может подавать независимые от разрешения команды PostScript, определяющие вид страницы на печати или экране.

Думаю, успех PostScript связан с его гибкостью и четко определенной базовой моделью изображения. Другие протоколы печати того времени стремились статически определять страницы с помощью структур данных. Такие протоколы неизменно терпели неудачу при описании некоторых интуитивно простых страниц.

Что заставило вас создать язык PostScript, а не формат данных? Его приходится интерпретировать принтеру. В чем фактически разница между языком и форматом данных?

Джон: Когда мы стали заниматься этими вещами в Xerox PARC, там был язык под названием JaM. Мы вели исследования в области графики, и нам нужен был интерпретируемый язык, чтобы можно было быстро проводить эксперименты, иметь интерфейс к аппаратной части Alto и к программным интерфейсам Alto, чтобы эти эксперименты не требовали долгого цикла компиляции и всех этих прогонов, компиляции, ассемблирования, загрузки и так далее. Мы воспользовались интерпретируемым языком, и он оказался очень эффективен для опробования массы новых идей.

Чарльз: Это дает вам также возможность, если со временем выяснится, что некоторые важные вещи в вашем языке работают недостаточно эффективно в режиме интерпретации, сделать из них набор расширений, введя новые операторы и реализовав их на более низком уровне, чтобы повысить эффективность.

Идея использовать язык возникла потому, что мы не знали, какие в будущем появятся устройства, среды и новые возможности и позволят ли они нам управлять выводом печатаемой страницы так, как мы того хотим. Язык давал нам предельную гибкость, которой мы не могли получить от структур данных.

В нем есть те же операции и управляющие элементы, что и в других языках, полных по Тьюрингу, но, конечно, согласно тезису Черча–Тьюринга его нельзя доказать математически.

Джон: В нашей области есть много примеров протоколов управления, не являющихся полными языками. Мы тогда рассудили, что поскольку протоколы всегда используются свободно и непредсказуемо, полный язык программирования позволит нам по-своему запрограммировать то, что мы могли упустить или неожиданно обнаружить. Эта «полнота» обеспечила такую долгую жизнь PostScript, которой мы не ожидали.

Одно из преимуществ конкатенативных языков состоит в том, что необязательно кодифицировать новую функцию: на одних платформах можно сделать ее аппаратной, а на более старых эмулировать ее программно. Если вы определяете новое слово, можно определить его через другие словарные примитивы языка, чтобы сделать доступным на старых машинах, но если нужно, чтобы поддержка этого слова была в интерпретаторе, можно добавить ее в последующие версии. Это правильно?

Чарльз: Да.

Джон: На самом деле, мы ввели возможность переопределения даже простейших операторов. В PostScript можно переопределить команду `add`, заставив ее делать все, что вам угодно. Такая гибкость фактически сделала возможным появление PDF, поскольку мы определили элементарные графические операторы так, что они могут взять свой стек операндов и поместить его в статическую структуру данных, а не сохранять программную природу PostScript.

Было ли это сделано отчасти для борьбы с возможными ошибками в ROM?

Джон: Совершенно верно.

Чарльз: В момент своего появления LaserWriter располагал самым большим по размеру программным обеспечением, когда-либо записывавшимся в ПЗУ.

Полмегабайта?

Джон: Да.

Это было тогда обычной практикой?

Джон: Это был огромный программный код для записи в масочное ПЗУ. На случай ошибок желательно было иметь какие-то обходные пути. Иметь язык программирования, позволяющий программным способом обходить ошибки, было очень удобно.

Чарльз: В любом проекте такой степени сложности нужно считаться с тем, что ошибки были и будут. Нельзя надеяться только на удачу.

В сущности, мы встроили механизм, позволяющий исправлять ошибки, потому что когда у вас десятки или сотни тысяч принтеров, невозможно каждый месяц выкладывать новый комплект ПЗУ.

Джон: С учетом тогдашней стоимости масочных ПЗУ было немыслимо организовать их рассылку.

Принимали ли вы тогда в расчет аппаратное обеспечение – помимо того, что оно должна быть?

Джон: Нет, первоначально язык возник в компании Evans & Sutherland. Мы строили огромные тренажеры с графическими проекторами, и когда появились спецификации проекта, аппаратура была еще не готова, а нам нужно было строить для нее базы данных. Нам пришлось оставить себе как можно больше возможностей для позднего связывания. Позднее связывание было столь важно потому, что у нас не было представления о том, как будет развиваться вся система и какой будет целевая машина.

Чарльз: Джон, можно немного рассказать об этой базе данных. Это был весь нью-йоркский порт.

Джон: Это были все здания Манхэттена – не Манхэттен целиком, а его очертание на фоне неба, Статуя Свободы, – и задача была в том, чтобы построить тренажер для ввода танкеров в порт Нью-Йорка. Это был удивительный проект, потому что он был выполнен примерно за год.

Это большой объем данных.

Джон: Это большой объем данных, особенно если учесть, что работа велась в основном на PDP 15. Это были крошечные машины. У них было 32 Кбайт памяти.

Приятно было, наверное, перейти на лазерный принтер.

Джон: Даже лазерный принтер был в то время самым большим процессором из всех, ранее созданных Apple. Там он считался мощным процессорным оборудованием. Графические интерфейсы для Мака уже были.

Фактически с помощью языка PostScript мы создали ответный интерфейс к имеющемуся графическому интерфейсу, чтобы брать структуры данных и делать такие же вещи на PostScript, для чего были написаны программы PostScript.

В то время это был Мак с 256 Кбайт или с 512 Кбайт памяти, и возможности Mac Draw или Write были очень ограниченными. LaserWriter

фактически загружал целый комплект программ, чтобы интерпретировать их команды.

В тот момент Мак генерировал PostScript?

Джон: Он генерировал PostScript, но это была некая версия PostScript для QuickDraw.

Чарльз: Да, он генерировал PostScript, но через набор подпрограмм PostScript, которые принимали QuickDraw и выводили PostScript. Эти макросы или подпрограммы фактически управляли принтером.

Вы вносили в PostScript изменения, чтобы поддерживать новые устройства?

Джон: Поддержка лазерных принтеров далась нам нелегко. В конце концов мы перенесли PostScript на матричный принтер.

Чарльз: Это был не самый приятный проект.

Джон: Да, уж.

Трудно ли мыслить в двух измерениях (графически) с самого начала создания языка?

Чарльз: Необходимо заложить механизмы двумерных преобразований, которые позволят программисту действовать в собственной системе координат, но в конечном счете преобразуют это пространство к системе координат фактического устройства.

Джон: Думаю, мыслить в двух измерениях легко, если представлять себе каждую двумерную конструкцию как вычерчиваемую (или отображаемую) посредством подпрограммы. Одно из достижений PostScript – возможность заключить каждый объект или группу объектов в собственную систему координат. Благодаря этому можно создавать на странице экземпляры объектов разных размеров и ориентации, не беспокоясь об их внутренних деталях. Благодаря этой простой идее легко думать о том, как заполнить страницу или ее часть.

По образованию вы математики. Какую роль сыграла ваша подготовка в разработке PostScript?

Чарльз: В отношении логики трансформаций модели изображений и мне, и Джону было очевидно, что именно так она и должна действовать, потому что механизм линейных преобразований должен быть неотъемлемой частью основания модели изображений. Это не особенность конкретного языка.

Не знаю, насколько большое значение это имело, но по какому-то стечению обстоятельств в моем случае помимо математики было также очень тесное знакомство со всем типографским процессом. Мои дед

и отец были фотограверами. Я многое знал о практике типографского дела, в частности о передаче полутонов и тому подобном. Какая-то часть всех этих знаний помогла в разработке PostScript, но я бы не рискнул сказать, что математика стала главным фактором. Однако знакомство с математическими инструментами крайне полезно для работы с абстрактными определениями.

Как у вас была организована работа? Были ли разногласия? Как вы принимали общие решения?

Чарльз: Я взял Джона на работу в 1978 году, так что мы работаем вместе уже 30 лет. За все это время не было случая, чтобы мы разрывали отношения из-за недовольства друг другом. Каждый из нас всегда испытывал достаточное уважение к идеям другого, и если случалось, что наши мысли расходились, то мы сразу старались выяснить, почему это произошло и какая из идей лучше, или нельзя ли каким-то образом их объединить.

Насколько помню, до ссор никогда не доходило. Такое сотрудничество достаточно уникально. Очень немногим доводится испытать это в своей профессиональной деятельности. Подобные отношения часто возникают в дружбе, в браке и в других случаях, но такой уровень взаимного уважения в рабочих условиях достаточно редок, как и умение быстро объединиться вокруг своих идей.

Многие сравнивают PostScript с Фортом, потому что тот и другой — стековые языки. Он оказал на вас какое-то влияние?

Джон: Нет. На самом деле, когда работа над языком в E&S была закончена, мы наткнулись на разные обсуждения Форта, и Форт оказался весьма похожим, но во многих отношениях очень и очень иным языком. Так что особого влияния не было. Чистое совпадение.

Чарльз: Полностью согласен.

Я всегда подозревал наличие связи, но теперь еще больше уверен, что сходные требования приводят к сходным конструкциям.

Джон: Замечательная особенность PostScript состоит в том, что для его реализации нужен очень и очень небольшой объем программирования, потому что он эмулирует аппаратную среду. Очень легко построить основные механизмы, а потом по мере надобности добавлять операторы.

Чарльз: Базовый интерпретатор PostScript занимал всего несколько килобайт.

Джон: Довольно мало.

Чарльз: Меньше не бывает.

Какие проблемы решала стековая конструкция? Как пострадал бы PostScript, окажись он в большей мере декларативным языком?

Джон: Стековая организация PostScript облегчает реализацию и позволяет быструю интерпретацию и выполнение. На тех машинах, для которых предназначался PostScript (Motorola 68000), простота и эффективность имели очень большое значение.

Мне кажется, что для определения PostScript наибольшее значение имели следующие решения:

- PostScript – законченный язык программирования со всеми переменными, условными операторами, циклами и прочим.
- Операторы PostScript можно переопределить средствами самого языка. Благодаря этому мы смогли реинтерпретировать имеющиеся файлы PostScript, создав файлы Acrobat (PDF). Это также позволило исправлять ошибки реализации, записанной в память, доступную только для чтения.
- Модель изображения позволяла выделить подструктуры и графически манипулировать ими, а в дальнейшем их можно было включать в более крупные инструменты. Например, PostScript позволяет взять описание страницы, уменьшить его и включить в другую страницу. Такой гибкости и легкости применения не было ни в одном из других протоколов печати.
- PostScript давал пользователю возможность обращаться с изображением как с любым другим графическим элементом: изменять масштаб, поворачивать и преобразовывать как всякий другой графический элемент. Такая возможность появилась впервые в PostScript.
- Несмотря на то что первые принтеры PostScript были черно-белыми, конструкция PostScript допускала использование цвета.

Накладывались ли ограничения на динамический стек?

Джон: Кажется, стек исполнения был ограничен 256 уровнями и все стеки имели ограниченную длину в байтах.

Чарльз: У машины, на которой все выполнялось внутри LaserWriter, была большая для того времени память – полтора мегабайта ОЗУ, из которых один мегабайт отводился под кадровый буфер. Для исполнения отводилось полмегабайта ОЗУ.

И еще у вас было полмегабайта ПЗУ?

Чарльз: Да.

Джон: Да.

И вы копировали ПЗУ в ОЗУ, а потом, если нужно, патчили его?

Джон: Нет, мы просто помещали оператор с загруженным кодом в ОЗУ, и он либо заменял встроенный оператор – как я говорил, операторы можно было переопределять, – либо добавлял новую функциональность.

А как насчет формальной семантики? Некоторые разработчики фиксируют семантику языка, а потом проверяют небольшой набор базовых функций.

Джон: У нас были словари для поиска имен и символов. У нас были массивы. У нас было все для работы с числами. У нас была очень простая стековая машина, которая могла принять до 256 операторов, если потребуется, но это была очень простая машина, и если работает базовый интерпретатор, отлаживаться становится легко.

Пожалуй, мы были уверены, что это надежная конструкция. Кажется, мы не занимались никакими формализмами.

Вы рассчитывали, что код PostScript будут писать вручную?

Джон: Нет. Но многие писали вручную. К этому можно приспособиться. Сейчас я предпочитаю JavaScript.

Чарльз: Все первые брошюры, на которых мы хотели продемонстрировать возможности PostScript, были сделаны дизайнерами – им пришлось научиться программировать.

Думаю, один из залогов долгого и успешного существования языка – размер интерпретатора, о чем я уже говорил. Он весьма невелик. Если бы захотелось, можно было взять и за пару выходных портировать его на любую платформу.

Джон: Пары выходных могло и не хватить.

Чарльз: Да, если заняться тонкостями обработки графики, все становится несколько сложнее.

Джон: Еще одна причина популярности PostScript – его текстовая часть, для которой мы действительно решили ряд проблем. Важнейшей из них стало масштабирование шрифтов по контурам. Благодаря ему из контуров получались довольно красивые символы, а никто больше не умел это делать.

Некоторые считали, что это невозможно.

Джон: Мы сами не были уверены, что это возможно.

Чарльз: Да, это было несколько рискованно. Мы решили пойти по этому пути, потому что альтернативой было набрать кучу людей, чтобы они вручную корректировали растровые изображения. Это противоречило всей философии механизмов произвольных линейных преоб-

разований PostScript, потому что при повороте символа на несколько градусов пришлось бы заново делать новый растр.

Джон: Было несколько ключевых идей, благодаря которым это все стало возможным. Мы первые это сделали.

Чарльз: Думаю, еще одна область, в которой PostScript весьма отличился, это совершенно уникальное качество передачи полутонов. В этой области работали многие, но, мне кажется, то, что было достигнуто Adobe к концу 1980-х, явно не уступало или превосходило возможности электро-механических систем, которые тогда применялись.

Вы связываете это со своими прежними исследованиями в области графики?

Чарльз: Это сделали не только мы вдвоем.

Джон: Не мы вдвоем. На самом деле, мы сделали первую систему передачи полутонов, просто эмулируя работу реальных механических устройств, но потом пригласили математиков, и человек по имени Стив Шиллер (Steve Schiller) проделал большую работу по передаче полутонов, разобравшись в этом на гораздо более фундаментальном уровне.

Чарльз: Мы с Джоном оба много занимались печатью. Мои отец и дед были фотограверами. Когда я приносил домой некоторые наши первые результаты, отец смотрел на них и говорил: «Нда, не очень-то».

Потом подключились Шиллер и другие. В какой-то момент даже отец стал одобрять нашу работу. Это было здорово.

Вы говорите только о вращении растровых шрифтов?

Чарльз: О произвольном масштабировании и разрешении. Это не столь важно при очень высоком разрешении и относительно нормальном размере точки, но для лазерного принтера или, боже упаси, экрана, это не годилось.

Джон: Базовая идея была в том, что люди пытались брать прототип контура символа и определять, какие биты нужно включить. Мы пошли другим путем.

Мы отмечали частоту растрового изображения и преобразовывали контуры, чтобы выровнять их по битовой карте, а потом включали очевидные биты. Благодаря этому все основные штрихи получались одинаковой толщины. Все засечки были одинаковыми. Выделение полужирным получалось правильным, и сама идея была очень проста, но никто за нее так никогда и не взялся.

Чарльз: Вполне терпимо, когда есть некоторая разница в толщине, если не считать случая повторения элемента внутри символа, когда глаз чувствует разницу.

Джон: Очень небольшую разницу.

Как вы справились с кернингом и лигатурами?

Джон: Помещаете начало на границу растра и выбираете ближайшее положение. Парный и более сложный кернинг осуществляется одинаково. Выравниваете символ по границе растра, а потом выбираете расстояние до следующего символа с точностью до пиксела. Все получается как надо. Лигатуры – это просто особые символы.

Чарльз: Потом еще нужно все это сделать для китайского и японского иероглифического письма.

Джон: У нас работал парень по имени Билл Пакстон. Когда мы занялись китайскими символами, оказалось, что важны не только промежутки между штрихами, но и дырочки и маленькие прямоугольники – они не должны исчезнуть или разорваться. Он построил очень сложный набор правил подгонки символа к частоте растра, которые должны были сохранить все важные части символа.

И вам пришлось определять эти важные части для каждого символа?

Джон: В значительно мере, да.

Чарльз: Но это нужно сделать лишь один раз. На самом деле, конечно, создание описания PostScript для китайского шрифта требует гораздо большей работы, чем для романского шрифта, просто ввиду огромного количества символов, но проделав это один раз, дальше можно не беспокоиться.

Может ли эта информация быть использована для других шрифтов?

Джон: Нужно разработать стратегию, а потом можно в значительной мере автоматизировать работу и просто сказать: вот здесь такая ситуация, и действовать нужно так.

Когда создавалась система TrueType, использовалась та же стратегия, но к каждому символу она применялась особо. Стратегия же, использованная в PostScript, в значительной степени распространялась на весь алфавит. Иными словами, на строчной *h* нужно идентифицировать левый штрих и правый штрихи. Уже определенные левый и правый штрихи действуют и для строчной *n*, они действуют везде. Высота строчной буквы без выносных элементов (`x-height`) одинакова для всех букв шрифта.

Нам нужно было определить эти свойства, а потом с ними должны работать алгоритмы. Если вы измените начертание *n*, у вас будет значительно меньше работы. Строить шрифты PostScript было гораздо легче, чем TrueType.

Современные принтеры могут интерпретировать PostScript (а также PDF) без промежуточной трансляции. Это стало возможным благодаря гибкости и красоте конструкции PostScript?

Джон: Очень важно вспомнить, что в то время, когда был впервые реализован PostScript, машины располагали очень маленькой памятью. Первоначально в LaserWriter было 1,5 Мбайт памяти и 0,5 Мбайт масочной памяти, доступной только для чтения. Один мегабайт ОЗУ выделялся в качестве буфера страницы, поэтому у нас было только полмегабайта всей рабочей памяти. Масочное ПЗУ хранило реализацию PostScript.

PostScript проектировался так, чтобы хранить в памяти как можно меньше. Как правило, не хватало памяти даже для того, чтобы записать всю программу PostScript. Это означало, что программа обрабатывала и печатала страницы, встретившиеся в программе PostScript, по мере считывания ее принтером. Такая стратегия позволила нам печатать очень сложные задания, не располагая большой памятью.

Acrobat устроен иначе. Все страницы располагаются в конце файла, поэтому принтеры с поддержкой Acrobat не могут начать печать, не прочтя весь файл. Сейчас, когда счет памяти идет на гигабайты, это не проблема.

При этом у PostScript и PDF одинаковые графические модели печати, и потому они тесно связаны.

Есть ли у предметной области PostScript особенности, затрудняющие написание понятных PostScript-программ?

Джон: Все было прекрасно в области графики, потому что стек, вложенные преобразования и рекурсивные вызовы чудесно действуют при работе с графикой. Можно начертить изображение, потом применить к нему в целом преобразования, и оно позаботится обо всем, что нужно сделать внутри, и сделает это правильно.

Чарльз: Не могу не рассказать здесь одну историю из эпохи Xerox PARC. Там всегда было много споров между сторонниками очень структурированного языка Mesa, между любителями Лиспа и теми, кто вполне был удовлетворен достаточно примитивным BCPL.

Один из наших программистов предложил устроить соревнование, в котором нужно было одну и ту же задачу запрограммировать на всех трех языках, чтобы посмотреть, какой вариант окажется самым коротким, самым быстрым и самым, по его мнению, элегантным.

В результате мы лишь выяснили, что все зависит от способностей программиста, а самым толковым оказался Боб Спраул, который сделал

все на BCPL, и никакие хитрые средства ему не понадобились. В конечном счете умение программиста имеет настолько большее значение, что выбор конкретного языка не столь существенен.

В PostScript Level II добавлены такие функции, как сборка мусора. Допускаете ли вы дальнейшее развитие языка?

Джон: В JavaScript все еще нет графических интерфейсов, с помощью которых можно напечатать страницу. Если я хочу напечатать документ, то пишу его на PostScript и с помощью distiller создаю файл PDF. Что ж, я полжизни не мог без этого обойтись.

Чарльз: Если вы говорите конкретно о PostScript, то подозреваю, что на данном этапе его существования никакого его быстрого развития не будет. Оно идет в направлении встраивания модели графического изображения PostScript в другие среды, например во Flash.

Джон: Да, чтобы Flash могла полноценно работать с текстом, там должны быть обычные текстовые механизмы Adobe. В конце концов все это окажется в телефонах.

Apple использует PDF для описания графических аспектов рабочего стола Mac OS X. Я читал, что фактически был проект использования в этих целях PostScript.

Чарльз: В 1983 году мы заключили с Apple договор, в рамках которого предоставили ей лицензионные права на Display PostScript. Стив Джобс очень хотел, чтобы это было в контракте. Когда Стив ушел из Apple, там решили пойти своим путем и бросить это дело, но Стив понимал, что ему нужна одна и та же модель изображения как для экрана, так и для печатной страницы, чтобы не отрывать одно от другого. Когда он организовал NeXT, мы заключили с ним контракт, и Display PostScript стал моделью графического изображения для компьютерной среды NeXT.

Представляю, насколько полезно это с точки зрения настольных издательских систем, где требуется максимальное соответствие между разными устройствами отображения.

Джон: С общесистемной точки зрения, когда мы смогли масштабировать шрифты в соответствии с разрешением экрана – и это тоже была работа Билла Пакстона, – мы действительно получили очень последовательную модель графики, в которую были заложены большие возможности.

Чарльз: Представьте, насколько интереснее могла быть Сеть, если бы графической моделью для HTML был выбран PostScript и не пришлось заниматься всеми нынешними глупостями для ее моделирования.

Джон: Это действительно любопытно. Сейчас Adobe развивает Flash, добавляя туда работу со шрифтами. Это интересно. В Flash никогда не было ни приличной поддержки шрифтов, ни приличного графического движка. Теперь Flash движется в сторону значительного усиления в этих областях.

Чарльз: Думаем, по сути это будет Display PostScript для Сети.

Как вы считаете, Flash может мигрировать на принтеры?

Джон: Нет.

Чарльз: Нет.

Будет ли что-то зависеть от принтеров?

Джон: Все в меньшей степени.

Чарльз: Сейчас наблюдается явление, когда многие принтеры фактически являются PDF-принтерами, а не PostScript-принтерами, а это просто означает, что интерпретирование вернулось на компьютер.

Джон: Но компьютеры сейчас стали несколько иными, чем были тогда.

Чарльз: То, что сейчас можно со своего компьютера выводить на струйный принтер со скоростью 20 страниц в минуту, говорит о многом.

Исследования и образование

Что в развитии программного и аппаратного обеспечения после 1970-х стало для вас настоящим сюрпризом? Многие из применяемых сегодня идей появились в 1970-х в PARC.

Чарльз: Думаю, важно понимать, как появился PARC и чем он был. История началась во времени перехода власти в США от администрации Эйзенхауэра к администрации Кеннеди.

Эйзенхауэр отвел Кеннеди в сторону и сказал, что некоторые умнейшие люди в американской оборонной промышленности высказали ему свое мнение, что если США хотят расширять свое военное присутствие в мире, им нужно заменить аналоговые средства связи цифровыми. Вряд ли действовавший президент Эйзенхауэр или избранный президент Кеннеди понимали, в чем смысл этой рекомендации, но Кеннеди воспринял ее со всей серьезностью.

Он отвел в сторону самого технически продвинутого человека в своем кабинете, Макнамару, и передал ему эти слова Эйзенхауэра. «Я хочу, чтобы вы этим занялись. У вас огромный бюджет. Я хочу, чтобы вы взяли из него столько, сколько нужно, чтобы начать это дело, но не слишком много, чтобы не пришлось долго отвечать на вопросы Кон-

гресса, потому что я хочу, чтобы эта программа заработала быстро и эффективно».

Макнамара, в свою очередь, нашел подходящего человека в MIT. Его звали Джозеф Карл Робнетт Ликлайдер, и он был связан с исследовательскими лабораториями MIT, где стали исследовать, как использовать компьютерные технологии не для вычислений, а для связи. Он узнал, что ученые MIT установили связи с другими академическими и рядом промышленных исследовательских лабораторий по всей стране, и, объехав их все, обнаружил, что там работают чрезвычайно одаренные люди. Это были такие места, как Калтех, Университеты UCLA, Стэнфорда, Беркли, Юты, Мичигана и, конечно, группа заведений на восточном побережье – технологические институты Массачусетса и Карнеги и ряд других.

Он решил взять несколько десятков миллионов долларов и распределить их небольшими суммами среди примерно десятка университетов и нескольких исследовательских лабораторий, таких как Bolt Beranek, Newman и RAND Corporation. Он сказал: «Я не буду заниматься мелочным контролем. Я дам вам эти деньги, и их вам хватит на несколько лет, чтобы начать эту работу. Ведите исследования так, чтобы когда Конгресс заинтересуется ими и захочет задать нам вопросы, мы могли указать, на что потрачены деньги. Но самое главное, – и особенно это касается академических учреждений, – я хочу, чтобы вы подготовили кадры специалистов в этой области».

Если вы посмотрите на подготовку мою, Джона, то мы все прошли через ARPA. На самом деле, если вы изучите генеалогию Кремниевой долины, то почти все основатели компаний и ведущие исследователи в этой области получили образование через ARPA (Advanced Research Project Agency – Агентство передовых исследовательских проектов) министерства обороны в ту эпоху, когда, как я сказал, мелочным контролем не занимались.

Вот откуда в PARC пришли главные люди, потому что Xerox сделала следующее: она взяла на службу того, кто принял дела у Дж. К. Р. Ликлайдера, человека по имени Роберт У. Тэйлор, а он знал, где мы все учились. Он привел всех этих людей в PARC. PARC стал первой промышленной демонстрацией качества людей, которых сформировало ARPA за предшествующий десяток лет. То, что все эти люди вместе стали работать в одной организации, PARC, оказало огромное влияние на нашу отрасль.

Чарльз, вы создали в Xerox PARC лабораторию графических исследований и руководили ее научной работой. Вы можете дать совет, как руководить исследовательской группой?

Чарльз: Самое нерушимое и важное правило – это набрать самых ярких специалистов, каких вы только сможете найти. Лучшим, наверное, моим кадровым решением в жизни было принять Джона на работу в этой лаборатории, плюс там уже было несколько очень талантливых исследователей. Когда ядро для исследовательской работы состоит из таких людей, они обычно привлекают других высококлассных специалистов, особенно молодых, обычно с учеными степенями, если речь идет об исследованиях. Мы смогли собрать очень сильную команду.

С самого начала большая часть наших исследований не ограничивалась рамками группы, созданной в лаборатории, а охватывала также другие подразделения Xerox и, в какой-то мере, простиралась на сообщество академических исследователей. Такая интеграция очень полезна для исследований, потому что она привносит разнообразие точек зрения.

Как вы определяете хорошего исследователя?

Чарльз: Его нельзя определить с помощью какого-нибудь теста. Если это люди, которые какое-то время работают в этой области, нужно посмотреть, каковы результаты их работы. Я знал, что Джон хорошо проявил себя после получения им высшего образования, но мы никогда не работали вместе и не встречались формально, пока я не пригласил его на интервью. По тому, что он сделал у Evans и Sutherland, я знал, что в нем сочетаются творческий ум и способность доводить дело до конца.

Он из тех, кто не просто кидает идею, предоставляя дорабатывать ее кому-то другому, но доводит ее до конечной реализации. Я всегда считал очень ценным качеством исследователя способность довести идею до фактической разработки ее высококачественного представления.

Может быть, это и составляет разницу между теоретическими и прикладными исследованиями.

Чарльз: Не думаю, что разница в этом. Я считаю, что для тех и других действует один и тот же критерий. За годы своей работы я пришел к выводу, что как исследователи, так и инженеры, которые помимо определенных интеллектуальных способностей обладают способностью довести начатое дело до конца, оказываются значительно продуктивнее и имеют значительно большее влияние.

Как вы определяете, что проект является многообещающим?

Чарльз: В частности, это могут быть мнения коллег: вызывает ли проект у них интерес и хотят ли они присоединиться к работе над ним. Помню, когда мы начали работать над проектом InterPress для Xerox, где разрабатывался предшественник PostScript, у нас были участники не только из Xerox, которые хотели работать над этим.

У нас был профессор из Стэнфорда, профессор из Карнеги-Мелона, исследователь из Xerox, который самостоятельно работал на восточном побережье. Это действительно интересно. За исключением меня и Джона, шестеро участников проекта никогда не работали в одном месте до самого конца проекта. Все делалось с помощью электронной почты и ARPAnet, через которую мы обменивались информацией в течение всего времени.

Моя стратегия всегда заключалась в том, чтобы задать генеральное направление. Люди понимали, какова конкретная цель лаборатории, и, сообразуясь с ней, они творчески решали, что им делать. Моя задача как руководителя была в том, чтобы помочь им сформировать и представить то, что они хотели сделать, а потом найти необходимые им ресурсы.

Если вы занимаетесь преимущественно разработкой, то обычно уделяете значительно больше внимания тому, чтобы получить нечто, имеющее рыночные перспективы, и мы пытаемся составить план в соответствии с рядом критериев, дающих возможность данному проекту оказаться успешным на рынке.

Я спросил вас о руководстве исследовательской лабораторией, потому что знаю о проблемах, возникших при создании продукта из InterPress.

Чарльз: Проблемы были не с руководством исследованиями. Проблемы были с тем, что не было сделано попытки реально определить, как идеи, полученные в результате исследований, можно передать разработчикам компании для эффективной реализации, и здесь действительно был допущен промах. Это была неудача на более высоком уровне управления, чем просто исследования.

Хотя, если честно, то нам, исследователям, казалось, что если у нас появились замечательные идеи, то люди из разработки должны просто наброситься на них и все реализовать. Это возвращает нас к моему замечанию, сделанному ранее. Настоящий исследователь – в молодости я этого не понимал – должен упорно сопровождать свою идею до ее окончательного воплощения, если он действительно хочет добиться успеха.

Чем лидер отличается от менеджера?

Чарльз: Лидер – это человек, который знает, чего он хочет, и довольно хорошо представляет себе, как этого достичь, и он должен уметь найти и заинтересовать людей, чтобы они работали для достижения этой цели. Вот что такое лидер.

Менеджер в основном занят обеспечением поддержки в виде бюджета, обмена информацией и прочего, что требуется для существования про-

екта, но ему не обязательно иметь представление о том, куда движется проект.

В любой сложной организации нужны и те и другие, но если не понимать между ними разницы, возможны катастрофические последствия, потому что иногда лидера и менеджера совмещают в одном лице. Если вы считаете, что взяли на работу лидера, а в действительности у него преобладают навыки менеджера, то результаты могут вас разочаровать. Наоборот, если вам нужно управлять большой организацией, и вы выбрали для этого лидера, который большую часть времени живет будущим, обдумывая свои гениальные идеи, он может оказаться неудачным менеджером.

Здесь требуются разные способности, которые важны сами по себе, но нельзя путать их, когда вы ставите задачу перед человеком. У человека есть качества либо лидера, либо менеджера, а их совмещение встречается очень редко.

Как вы определяете хорошего программиста?

Чарльз: Обычно по опыту работы вместе с ним, рядом с ним, на него – в результате активного сотрудничества в работе. Какого-то абстрактного способа я не знаю.

Мне посчастливилось работать с разными людьми. Есть один человек – помимо Джона, – которого я не могу забыть, и о котором вы, возможно, никогда не слышали. Это Эд Тафт. Я взял Эда к себе на работу в Xerox PARC в 1973 году. Могу сказать, что это лучший программист, с которым мне когда-либо доводилось работать. Он очень старательно относится к описанию задачи, которую нужно решить, и стремится сделать это на таком уровне, который оставляет вам максимальную свободу изменить свое решение относительно того, как на самом деле это должно работать.

Можно сказать, что он применяет позднее связывание, примерно как обсуждавшееся нами в PostScript, но он доводит работу до конца. Когда он предъявляет код и говорит, что тот готов, это значит, что код абсолютно надежен и отшлифован. От уровня концепции до полного завершения – он обеспечивает все. Если бы Эда можно было клонировать! Можно было бы создать великолепную организацию.

Есть ли какая-нибудь особая тема в компьютерных науках, на которую студентам колледжей следовало бы обратить особое внимание?

Чарльз: У меня достаточно консервативное отношение к тому, как должно быть построено начальное преподавание компьютерных наук. Думаю, на младших курсах нужно как можно больше изучать математику и физику, а компьютерные науки оставить, по возможности, для программы получения степени мастера или доктора, но это всего лишь

мое мнение. Очевидно, университеты следуют желаниям своих студентов, и я понимаю, почему они дают диплом бакалавра по компьютерным наукам.

Помимо серьезной подготовки по математике и естественным наукам, очевидно, что если вы хотите заниматься аппаратной частью, то у вас должны быть еще какие-то знания химии и физики. Таково мое предвзятое мнение.

Для общего образования, по моему глубокому убеждению, большое значение имеют гуманитарные науки. Что толку, если в вашей голове рождаются замечательные идеи, когда вы не можете их эффективно представить и убедить других последовать за вами? Для достижения успеха совершенно необходима способность к письменному и устному общению. Если у вас ее нет, ваше образование нельзя считать законченным, и вы будете работать менее эффективно.

Настоятельно рекомендую уделять в основной программе высшего образования как можно больше внимания гуманитарным наукам, наряду с естественными науками и математикой.

Какой урок из опыта вашей разработки PostScript мог бы оказаться полезным другим?

Чарльз: Одна из замечательных идей, положенных в основу PostScript, это возможно более позднее связывание со всякими конкретными вещами. Говоря иначе, нужно выполнять расчеты и прочие действия на достаточно высоком уровне абстракции. Только в последний момент, когда нужно решить, какие биты в растровом изображении должны быть включены, а какие выключены, вы выбираете конкретные алгоритмы, которые это сделают.

Оставаясь на таком более высоком уровне абстракции, вы можете построить достаточно общее описание изображения, которое нужно изготовить, что открывает возможности его импорта на широкий круг устройств. Это такая философия графической модели, которая позволяет делать то, о чем мы уже говорили, а именно, создавать одну графическую модель, которая выполняется не только на вашем персональном компьютере, но и в Сети, и на всех ваших цифровых графических устройствах от телевизоров до телефонов и различных интернет-гаджетов.

Вот в чем прелесть PostScript. Вы можете описывать с его помощью то, что вам нужно, на более высоком уровне, и лишь в последний момент привязать его к конкретному устройству.

Интерфейсы для долголетия

Что должен учитывать разработчик универсального языка программирования, заботящийся о его долголетии? Есть ли какие-то специальные меры, которые он должен принять?

Джон: Многие языки решают конкретные задачи. Вспомните HyperCard Аткинсона. Он сделал самую распространенную, как мне кажется, ошибку, когда вместо полного языка программирования сделал ограниченный. У вас должны быть управляющие структуры, у вас должны быть условные операторы, у вас должны быть циклы, у вас должна быть математика и все, что требуется для полного языка, иначе в какой-то момент вы упретесь лбом в стену.

Нам говорили: «Зачем вам все эти тригонометрические функции? Что вы будете с ними делать?» – и всем им нашлось применение. Взявшись за разработку языка, важно с самого начала понять, что он должен быть полным. Вам нужен доступ к файловой системе. Вам нужно все, что сделает его полным. Думаю, это очень важно.

Прошло 25 лет, и есть машины PostScript, которые все еще работают, – те же самые машины, и они продолжают выполнять PostScript. Он существенно развился, но базовая программа по-прежнему выполняется.

Чарльз: У меня есть LaserWriter второго поколения, которым я продолжаю пользоваться, потому что у него лучшая ручная подача из всех устройств, которые я видел. Он по-прежнему работает. Canon решила, что принтер можно выкинуть после 100 000 отпечатанных листов. Они явно заложили в него лишний запас прочности.

В чем различие между разработкой языка для потребления человеком и для потребления машиной, или для производства человеком и производства машиной? Учитываются ли при разработке какие-то факторы, когда вы соглашаетесь сделать что-то определенным образом, потому что никто не станет сидеть и прокручивать целиком весь этот цикл?

Джон: Крупный недостаток PostScript – большая сложность его отладки. Написав однажды код, вы выкидываете его из головы. Когда через полгода вам приходится вернуться к нему, это малоприятно. В то же время обычные языки с инфиксной нотацией, где не так много текущих состояний, гораздо проще читать и гораздо проще отлаживать.

Не нужно держать в голове состояние всего стека.

Джон: Верно.

Вы рассматривали это противоречие, разрабатывая PostScript?

Джон: Поскольку было очень легко добавлять новые операторы, возникало стремление писать очень короткие подпрограммы и практиковать как можно более четкое разделение функциональности, но синтаксис, конечно, не располагает к программированию вручную.

Вы говорили, что у вас были дизайнеры, которые писали на PostScript программы для изготовления брошюр. В 1980-е были секретари-референты, которые писали вручную документы LaTeX. Каково вам было обучать дизайнеров программированию для изготовления брошюр?

Джон: Интересный вопрос. Малоизвестный факт, о котором Adobe никому не сообщала: у всех наших приложений есть базовый интерфейс к JavaScript. Можно писать скрипты для InDesign. Можно писать скрипты для Photoshop. Можно писать скрипты для Illustrator на JavaScript.

Я постоянно работаю с Photoshop с помощью программ на JavaScript. Как я сказал, это весьма малоизвестный факт, но интерфейсы для скриптов очень полные. Они дают в случае InDesign настоящий доступ к модели объектов, если кому-то это понадобится.

Чарльз: Это испытание не для слабонервных.

Использование JavaScript для управления моделью документа HTML тоже обычно не для слабонервных.

Джон: Да, это так. Я занимаюсь этим постоянно, и это не для слабонервных, особенно ввиду различия наборов символов, да и почти все различается в этих двух средах.

Но, тем не менее, этим пользуются. Если вам нужно автоматизировать создание документов, лучше всего для этого взять JavaScript и запрограммировать потрясающий типографский механизм InDesign или потрясающий графический механизм Photoshop. Можно автоматически и довольно просто создавать огромные количества документов.

Это тоже похоже на фактор долголетия. Сделайте язык универсальным и дайте все эти операции и возможности, но при этом разрешите писать циклы и управлять потоком выполнения.

Джон: Это действительно так. Есть такие проекты, и я связан с некоторыми из них. У меня есть один сайт с 90 000 страниц. Если бы я не автоматизировал его создание, с ним бы ничего не вышло. Слишком много страниц HTML.

Об этом просили дизайнеры или кто-то показал им, как это делается?

Джон: С помощью поддержки и расширения Photoshop, расширения InDesign и расширения многих других вещей; например, многие не зна-

ют, что Bridge – программа для работы с файлами и просмотра графики в Illustrator, Photoshop и InDesign – полностью написана на JavaScript.

Вы просто осуществляете трансляцию между объектными моделями в обоих направлениях.

Джон: Верно. Это большой объем JavaScript, но то, что все так хорошо подходит друг к другу, просто поразительно. Все полностью переносимо.

Как, с вашей точки зрения, должны разработчики учитывать аппаратное обеспечение? Является ли ПО лидером инноваций?

Чарльз: Думаю, это как инь и ян. В те времена, когда все эти «инновации» возникали в PARC, все происходило на очень слабых машинах с относительно маленьким дисковым пространством, с весьма скромными сетевыми возможностями и так далее. В тех жестких условиях мы проявляли большую изобретательность, чтобы заставить наши программы делать вещи, довольно удивительные для пользователей этих компьютеров, потому что они никогда такого не видели.

Сегодня мы находимся в ситуации, когда среда резко изменилась и аппаратная часть продолжает развиваться такими быстрыми темпами, что объем памяти исчисляется гигабайтами при весьма умеренной стоимости, а процессоры работают с молниеносной скоростью.

Любопытно, как мне кажется, что когда сняты все ограничения относительно низкой скорости и относительно малого объема памяти, легко обленится и исходить из возможности полного использования всех аппаратных средств. В результате мы сталкиваемся с очень сложной проблемой, когда совершенно неожиданно аппаратные ограничения снова становятся проблемой и для решения определенных задач требуется творческий и изощренный подход к разработке ПО.

Между аппаратным обеспечением и программной средой всегда устанавливается некоторое равновесие. Что касается приложений, то Vista представляется мне хорошим примером того, как обленившиеся разработчики решили, что новые машины справятся с неэффективностью программ и некоторыми неудачными решениями. Не вышло.

Сейчас они одумались и несколько снизили свои ожидания возможностей аппаратуры, что, возможно, приведет к появлению лучшей версии Windows, чем та, которой оказалась Vista.

Легче ли сегодня добиться популярности языка?

Чарльз: Фактически, нет. Мне кажется, что когда люди первоначально осваивают разработку и программирование, они это делают в условиях определенной среды, которая в значительной степени формирует их представления о том, как нужно разрабатывать программы. Им очень трудно разорвать связь со своим прежним опытом, когда появляются

новые инструменты. Для того чтобы новый язык программирования действительно смог стать популярным, нужно найти среду, где он стал бы ранним опытом для многих людей, которые свяжут с ним будущее. В некоторой степени это больше подходит для учебной среды, чем для независимой организации, пытающейся вывести новый язык на рынок.

Например, такое развитие мы сейчас наблюдаем для понятия облачной (рассеянной) обработки (cloud computing), при которой вычисления и доступ к информации широко распределяются между машиной пользователя, Интернетом, серверами и разными прочими местами.

Мне это хорошо понятно, и я не вел в этой области никакой работы, но тут открываются возможности для языка, который, может быть, позволит лучше учитывать такое разнообразие среды, чем любой из нынешних языков. Я говорю «может быть», потому что не знаю, действительно ли это проблема языка. Конечно, если взять такие организации, как Google, Microsoft и, в какой-то мере, Adobe, которые действительно уделяют много внимания тому, чтобы обеспечить своим клиентам комплексные условия работы в своей среде, то можно обнаружить, что со временем инструменты начинают этому препятствовать. Это может быть собственно язык или, что важнее, язык вместе со средой программирования, которая всегда его сопровождает, но они должны быть подняты выше уровня существующих сегодня инструментов.

Далее, проблема в том, что естественных сред обитания не так много. Нет больше прежних Лабораторий Белла, исследовательского центра IBM или, скажем, Xerox PARC. У нас нет каких-либо «отраслевых лабораторий», где можно вести такого рода исследования и разработки. Есть, конечно, ряд академических сред очень высокого уровня, но значительная часть их финансируется очень узкими исследовательскими проектами, поддерживаемыми преимущественно правительством США через NSF или DARPA.

Когда-то существовала среда, в которой была разработана система Berkeley UNIX, или та, где я работал над диссертацией с Уильямом Вулфом, когда мы разработали язык высокого уровня BLISS для системного программирования. Мы получили финансирование благодаря тем способам, которыми ARPA управляла своими средствами в те времена. Сегодня получить финансирование в исследовательской среде не так легко. Затрудняюсь предположить, где могут произойти подобного рода разработки.

Корпорации, нацеленной на получение дохода и прибыли, очень трудно выделить такие инвестиции, если у нее нет какой-либо побочной организации, чтобы можно было независимо профинансировать исследова-

ния. У большинства корпораций, особенно в области программирования и Интернета, сегодня нет таких внутренних организаций.

Может быть, организовать проект open source?

Чарльз: Может быть, но open source, с моей точки зрения, имеет ту проблему, что эта форма хорошо работает, когда идея уже достаточно хорошо разработана, достигнуты некоторая структурная целостность и хорошее понимание. Если взять чистый лист бумаги, сказать, что это open source, и попытаться что-то начать, думаю, стронуться с места будет очень тяжело.

Не считаете ли вы нужным хотя бы создать открытый стандарт?

Чарльз: В наше время этим нужно заниматься. Нужно, чтобы он был открытым, и, честно говоря, нужно, чтобы у людей была возможность добавлять туда свои средства. Возможно, вы этого не знаете, и я могу говорить только об Adobe, поскольку тут я более компетентен, но у всех продуктов Adobe есть открытый интерфейс в стиле JavaScript, благодаря чему сторонние разработчики могут строить очень изощренные дополнительные модули для всех наших продуктов, включая InDesign, Photoshop, Acrobat и так далее, и делать это платформо-независимым образом с помощью скриптов. Множество компаний и отдельных групп постоянно этим занимается.

Это не open source – в том смысле, что у вас нет Си-кода для начинки Photoshop, но это способ сохранить целостность базовой составляющей, предоставив при этом значительную свободу сторонним разработчикам для экспериментирования и повышения эффективности.

Обычные пожелания

Какие еще важнейшие проблемы стоят в области программирования и компьютерных наук?

Джон: Мир сейчас так устроен, что у меня на полке стоит три или четыре десятка учебников, имеющих отношение к Сети. Все это толстые книжки, и все они противоречат друг другу. Я был бы счастлив, если бы удалось навести порядок в графических моделях, в программных средах и во всем, из чего сегодня состоит Сеть, потому что нет никакой нужды во всех лишних вещах.

Нужно избавиться от кошмара множества разных броузеров и связанных этим проблем, вызванных разными реализациями HTML.

Что касается Flash, то мы пытаемся расширить его и сделать более надежным, чтобы хотя бы иметь один платформо-независимый язык

и переходить с одной платформы на другую без того, чтобы каждый раз страдать от изменения семантики.

Чарльз: Полностью согласен. Удручает, что по прошествии стольких лет мы все еще находимся в условиях, когда нам говорят: хочешь, чтобы это работало, тогда используй Firefox. Давно пора с этим покончить! Весь смысл универсализма Сети в том, чтобы не было таких различий, но мы по-прежнему миримся с ними.

Всегда интересно смотреть, как много времени требуется для отмирания некоторых архаизмов. Помещая их в броузеры, вы способствуете закреплению их вечного существования, а это глупо.

Вы рассматриваете это как слабость существующего процесса стандартизации?

Джон: Процесс стандартизации направлен не столько на решение проблем, сколько на закрепление истории.

Чарльз: По моему личному мнению, работа по стандартизации часто сводится к тому, чтобы окропить святой водой минувший праздник. Как сказал Джон, она не имеет целью создание чего-либо нового. Ее задача – в закреплении исторически создавшегося положения.

Если нет активной, энергичной организации, которая владеет стандартом и наводит порядок или делает реализацию этого стандарта настолько легко и просто доступной, что никто другой больше не думает сделать это самостоятельно, никакого стандарта не будет. Это дилемма, которая стояла перед нами в начале работы над PostScript. Если бы клоны смогли лишить нас контроля над PostScript, мы никогда не добрались бы до PostScript 3. К тому моменту образовалось бы столько несовместимостей, что весь его смысл был бы потерян.

Джон: То же самое относится к PDF. Мы все-таки добились его принятия Национальным управлением архивами в виде подмножества того, что существует сейчас, но хотя бы у нас есть спецификация.

Создавая Acrobat, мы твердо решили, что эти файлы должны жить долго, и мы обязуемся обеспечивать полую обратную совместимость, чтобы Acrobat мог читать и самые старые файлы. Это большой труд. Acrobat – огромная программа, но он играет такую роль в Сети, что я не могу представить себе, как бы она могла без него существовать.

Нужно ли, чтобы эти стандарты управлялись одной главной реализацией, или они могут возникать в результате приблизительного согласия?

Джон: В случае PostScript стандарт был действительно определен на основе нашей реализации, и примерно то же случилось с Acrobat.

Затруднения возникают, если у вас есть Netscape, а потом появляется Microsoft, и у Microsoft нет никакой мотивации обеспечивать совместимость. Это прискорбно.

Чарльз: Точно так же они поступили с Java.

Стала бы Сеть удобнее, если бы место HTML и JavaScript занял PostScript?

Чарльз: Мы работаем над своей новой платформой для Сети, которая сейчас называется Adobe AIR. Adobe Internet Runtime – это способ приблизить качество графики в Сети к уровню обычных приложений и уровню ядра модели изображений PostScript, организовав это так, чтобы можно было создавать приложения, в которых стирается грань между происходящим на настольной машине и в веб-приложении. Полагаем, что так мы сможем разместить в Сети графику и модель изображения типа PostScript, для которых в HTML нет достаточно эффективной поддержки.

HTML создает две проблемы. Во-первых, он ориентирован на представление преимущественно растровой информации. Во-вторых, он не является стандартом. Я хочу этим сказать, что если вы возьмете самые популярные броузеры и откроете в них некоторую HTML-страницу, то все они покажут разное. Для меня это неприемлемо, потому что если требуется построить достаточно сложный сайт, придется программировать с учетом специфики броузеров, чтобы этот сайт одинаково выглядел независимо от того, в каком броузере его просматривают. Это путь назад, к дурным старым временам.

Все произошло из-за того, что HTML оставили в виде некоего «открытого стандарта». Полагаю, здесь заложено противоречие. Можно сделать открытой реализацию стандарта, но сам стандарт должен быть очень хорошо проработан, чтобы не возникало таких несоответствий.

Помню, как Sun впервые представила Java. В конце концов они с Microsoft заключили сделку, и первое, что сделала Microsoft, – внесла в Java изменения. Различные реализации не были согласованы. Это противоречит самому понятию стандарта. Если вам нужен стандарт, так пусть это будет действительно стандарт, чтобы все его соблюдали. Обычно для этого нужна одна достаточно автократическая организация, которая будет поддерживать этот стандарт, а не его реализации.

Мы полагаем, что есть реальная возможность поднять качество всего, что делается в Сети, на совершенно иной уровень. Мы работаем над этим. Есть множество интересных приложений, сделанных сторонними разработчиками на платформе AIR, и мы будем и впредь уделять ей большое внимание. Она делает практически несущественным, в какой

операционной системой и на какой платформе вы работаете. Это становится не важным.

Можно понять, почему Apple и Microsoft в какой-то степени рассматривают это как вызов себе, ведь они хотят, чтобы вы выбрали их решение для осуществления интеграции с Сетью. А мы говорим, что это не должно иметь значения. Это облако должно быть доступно с любого компьютера и для любого типа информации, размещенной в Сети.

17

Eiffel

Eiffel – это объектно-ориентированный язык, разработанный в основном Бертраном Мейером (Bertrand Meyer) в 1985 году. Развитием этого языка сейчас управляет комиссия по стандартизации в Ecma International, выпустившая стандарт ISO в 2006 году. Он включает в себя набор современных и широко распространенных функций: сборка мусора, обобщенное программирование и безопасность типов. Вероятно, главным вкладом Eiffel стала идея «контрактного программирования» (design by contract), в соответствии с которой язык обеспечивает предусловия, постусловия и инварианты интерфейсов, что способствует надежности и повторному использованию компонентов. Влияние Eiffel очевидно в таких языках, как Java, Ruby и C#.

День вдохновения

Почему вы решили создать язык программирования?

Бертран Мейер: Очень редко языки программирования создаются из чистого интереса. Eiffel появился в силу необходимости. Я разработал язык программирования, потому что мне нужно было писать программы, а все другие языки меня не удовлетворяли.

Вам нужен был инструмент для реализации контрактного программирования?

Бертран: Это очевидная причина, но в целом мне нужен был объектно-ориентированный язык. Позвольте описать вам обстановку. В 1985 году мы открыли компанию Interactive Software Engineering. Сейчас она называется Eiffel Software. Мы собирались делать инструменты для программной инженерии. Одна японская компания выделила нам средства для создания редактора программ или редактора, управляемого синтаксисом, который был нами создан и имел некоторый успех.

Это была очень маленькая компания. Я все еще преподавал в Калифорнийском университете Санта-Барбары, так что это был несколько побочный бизнес. У нас были рабочие станции UNIX, полученные от японских заказчиков, для которых они были одним из видов продукции. Шел 1985 год, и я программировал в объектно-ориентированном стиле уже почти десять лет. В 1970-х годах мне повезло натолкнуться на язык Симула-67, которым я сразу увлекся. Я был уверен, что программировать нужно именно так.

На наших машинах не было компилятора Симулы, а я очень любил этот язык. Как сказал Тони Хоар об Алголе, он стал усовершенствованием многих своих наследников. Тем не менее в Симуле не было ни множественного наследования, ни универсальности – а к тому времени я понял, что нужно и то, и другое. Я обосновал это в докладе «Genericity versus Inheritance» (Универсальность и наследование), подготовленном для первой конференции OOPSLA. Мы решили посмотреть, что было доступно в то время. Имелся C++; я открыл книгу и быстро закрыл ее, потому что мне нужно было нечто иное – идея сделать объектность привлекательной для программистов Си была интересна, но явно могла быть только временным решением на пути к чему-то более цельному. Был также Objective-C, но он был слишком тесно связан со Smalltalk и плохо поддерживал те принципы программной инженерии, которые нас интересовали. То же самое относится к самому Smalltalk. Smalltalk – отличная разработка, но она никак не согласовывалась с нашими стремлениями. Eiffel появился как комбинация объектно-ориентированной технологии с принципами и практикой программной инженерии, разработанными в предшествующее десятилетие.

Smalltalk обладал отчетливыми чертами экспериментального программирования, что представлялось неуместным для тех задач, которыми мы собирались заниматься; например, отсутствие статических типов уже никуда не годилось для нас. Там была масса замечательных идей, но ничего из того, что нам было нужно.

Тогда я написал доклад. Это был доклад UC–Santa Barbara, который фактически описывал не язык, а библиотеку структур данных и алгоритм, потому что я был сильно увлечен повторным использованием и хотел, чтобы стандартная библиотека охватывала основные структуры компьютерной науки, которые я иногда называю «Knuthware». Позднее ее назвали EiffelBase, а тогда – просто библиотекой структур данных (Data Structures Library). В моем докладе описывались массивы, связанные списки, стеки, очереди и так далее. Там была использована особая нотация, и я заявил, что мы собираемся это реализовать. Я считал, что на реализацию уйдет три недели. Мы занимаемся ею до сих пор. Но это уже был Eiffel.

Язык не был самоцелью. Целью были многократно используемые компоненты, и я понимал, что для хороших многократно используемых компонентов нужны классы, нужна обобщенность, которую мы включили туда с самого начала, нужно множественное наследование и нужна взвешенная комбинация обобщенности и множественного наследования, что продемонстрировал мой доклад для OOPSLA. Нужны абстрактные классы. Нужны, конечно, контракты, что для меня было самым естественным делом. Все так переживают из-за них, а мне до сих пор непонятно, как можно программировать без контрактов. Еще я знал, что нужен хороший механизм потоков или сериализации, и работу над ним мы не откладывали на конец. Я понял это благодаря языку SAIL (Stanford Artificial Intelligence Language), который был очень хорошо спроектирован – не объектно-ориентированный, но очень интересный, – и с которым я работал в Стэнфорде десятью годами ранее. Принципиально важно было все это иметь с самого начала. Очевидно, конечно, что нужен был механизм сборки мусора.

Как вы пришли к этой философии? Ваш опыт практического программирования позволял определить, как можно усовершенствовать создание ПО?

Бертран: Отчасти опыт, отчасти, разумеется, чтение литературы. Будучи студентом Стэнфорда, я прочел в 1973 году книгу Даля, Дейкстры и Хоара «Structured Programming»[1]. На самом деле, это три монографии под одной обложкой. Первая, написанная Дейкстрой, – это знаме-

[1] Даль У., Дейкстра Э., Хоор К. «Структурное программирование». Серия: Математическое обеспечение ЭВМ. – Пер. с англ. – М.: Мир, 1975.

нитая книга о структурном программировании. Вторая, написанная Хоаром, рассказывает о структурах данных и тоже замечательна, но там была еще третья часть. Один из важных уроков, усвоенных мною из жизни, состоит в том, что люди читают начало книги, поэтому – фактически это совет тем, кто пишет книги, – очень тщательно выбирайте, что поместить на первые 50 страниц своей книги, потому что 90% читателей на этих 50 страницах и остановятся, даже если книга очень хорошая. Большинство людей прочитывает первую часть «Структурного программирования», написанную Дейкстрой. Некоторые читают вторую часть, написанную Хоаром. И, думаю, немногие добрались до третьей части, написанной Оле-Йоханом Далем при поддержке Хоара и названной «Иерархические структуры программ». В действительности это было введение в Симулу и ООП. Я был серьезным студентом: мне было сказано прочесть эту книгу, и я прочел ее от начала и до конца. Мне понравились первая и вторая части, а третью я счел разъяснительной.

Этим также объясняется, почему, когда через несколько лет вышло на сцену ООП и большинство заявляло, что оно заменит структурные методы, эти заявления казались мне бессмысленными. ООП с самого начала было частью структурных методов. Структурное программирование относилось к более низкому уровню, а объектно-ориентированное – к более высокому, но никакой пропасти между ними не было. Прочтя текст Даля и Хоара, я понял, что именно так нужно программировать. Когда в середине 1970-х я начал работать в компьютерной индустрии, мне повезло, что начальник разрешил мне купить компилятор Симулы, потому что он был весьма дорог, но очень хорош. Я активно использовал его в той фирме, где работал, и написал довольно интересные программы. Мне было совершенно очевидно, что других разумных способов писать программы нет, хотя многие считали меня чокнутым. В то время с объектно-ориентированным программированием было очень мало ясности.

В середине 1970-х, сразу после окончания университета, я со своим другом Клодом Бодуэном (Claude Baudoin) написал на французском языке книгу «Méthodes de Programmation» (изд. Eyrolles) – в некотором роде компендиум всего, что мы знали, чему научились в Стэнфорде и других местах. Книга пользовалась большим успехом. Фактически она печатается до сих пор, что несколько странно для книги 1978 года выпуска. Она стала учебником, без преувеличения, для пары поколений французских программистов – и русских, потому что была переведена тогда на русский язык в Советском Союзе[1]. В России она тоже имела большой

[1] Мейер Б., Бодуэн К. «Методы программирования».– Пер. с англ. – М.: Мир, 1982, т. 1, 2.

успех: бывая в России, я до сих пор встречаю людей, которые говорят, что изучали по ней программирование. Для объяснения приемов программирования, алгоритмов и структур данных там использовался псевдокод.

Я показал книгу Тони Хоару, и он сказал, что хотел бы включить ее в свою знаменитую серию по компьютерным наукам, издаваемую Prentice Hall, в английском переводе. Я согласился, и тогда он сказал: «Ты же говоришь немного по-английски, так почему бы тебе самому не сделать перевод?» Я сдуру согласился, вместо того чтобы найти того, кто сможет ее перевести. Это было самой большой глупостью в моей жизни: естественно, переводя книгу, я ее переделывал, потому что с момента первой публикации прошло уже три или четыре года. Я набрался опыта, и у меня возникли новые идеи. Я назвал книгу «Методология прикладного программирования», и она никогда не была опубликована, потому что я не смог ее завершить. Я переделывал каждое предложение. Это было очень непродуктивно, но при этом я улучшил псевдокод, которым пользовался в первой книге. В частности, я почувствовал, что не могу правильно описать программы или алгоритмы без контрактов. Нотация Eiffel для контрактов началась с этой работы.

Было еще одно важное событие. Я работал в промышленности, но взял академический отпуск в Калифорнийском университете в Санта-Барбаре (UCSB) и как гостю мне дали некоторые курсы, которые никто не хотел читать. Там были последовательные курсы 130A и 130B, «Структуры данных и алгоритмы», игравшие очень интересную роль, потому что в действительности у них было три цели. Официальной целью было преподавание структур данных и алгоритмов. Но были еще две скрытые и важные цели. Во-первых, курс должен был быть достаточно труден, чтобы отсеять изрядное количество студентов и оставить тех, которых стоило обучать компьютерным наукам. Во-вторых, в этом курсе студенты должны были изучить Си, потому что знание Си требовалось для других курсов.

Это был полный абсурд, потому что как бы ни был хорош Си, он плохо подходит для описания алгоритмов, не говоря уже об их изучении. Впечатление было ужасное, потому что вместо преподавания того, о чем я хотел рассказать в этом курсе, мне приходилось помогать студентам устранять ошибки в их программах, связанные с указателями Си и аналогичными его особенностями. Из этого я сделал два вывода. Во-первых, никогда больше не соприкасаться с Си в качестве языка, используемого человеком. Си – неплохой язык, если его генерирует компилятор, но идея писать на нем вручную совершенно абсурдна. Во-вторых, я понял, что единственный способ представить основные структуры данных и алгоритмы – это оснастить их в полной мере инвариантами циклов, вариантами циклов, пред- и постусловиями. Отчасти

поэтому, когда в конце того года мне нужно было разработать нотацию для нашей собственной работы в компании, которую мы тогда только что основали, я выбрал язык, которым хотел бы воспользоваться в том курсе, который читал в UCSB. Если шире, то это был результат изучения обширной литературы, поглощенности в течение многих лет работой по развитию программной инженерии, которую вели Даль, Дейкстра, Хоар, Вирт, Харлан Миллз, Дэвид Гриз, Барбара Лисков, Джон Гуттаг, Джим Хорнинг и подобные им люди, и, по сути, отслеживания эволюции языков программирования. Для меня выбор был очевиден. По существу, можно сказать, что Eiffel был спроектирован – я хотел сказать «в течение одного дня», но даже это будет неверно. Eiffel был спроектирован за 15 минут. Все было абсолютно очевидно.

Это Eiffel привел вас к идее контрактного программирования?

Бертран: Нет, путь был скорее обратным. То есть, концепция сложилась раньше. Язык только отражает ее. Для меня это бессмысленный вопрос, точнее, его нужно задавать тем, кто не пользуется контрактным программированием, – это они должны ответить, почему. Я просто не понимаю, как можно писать какие-то элементы программы, не потрудившись обрисовать, для чего эти элементы нужны. Это вопрос, который нужно задать Гослингу, Страуструпу, Алану Кэю или Хейлсбергу. Как они могут писать программы или разрабатывать язык, с помощью которого люди будут писать программы, и не обеспечить такой механизм? Просто не понимаю, как можно без этого написать хотя бы пару строк кода. Спрашивать человека, зачем он применяет контрактное программирование, – это все равно что спрашивать, зачем он пользуется арабскими цифрами. Пусть те, кто использует для умножения римские цифры, объясняют, зачем они это делают.

Я слышал, что контрактное программирование в ОО-языке реализует принцип подстановки Лисков. Вы согласны с этим?

Бертран: Я никогда не понимал, что такое принцип подстановки Лисков. Мне видится, что это просто полиморфизм.

Пожалуй, с этим я согласен, но загвоздка в том, что возможность подстановки должна быть полной. Нельзя, например, ограничить наследуемый тип так, чтобы он делал меньше, чем его родитель. По существу, вы должны выполнить тот же контракт, что и родительский класс.

Бертран: Мне кажется, это то, что появилось в Eiffel в 1985 году, – идея ослабить предусловие и усилить постусловие при переопределении подпрограммы. Если в этом состоит принцип подстановки Лисков, то я, видимо, согласен. Но в Eiffel это было до Барбары Лисков.

Мне нравится мысль о сходимости открытий.

Бертран: Часть работы Барбары Лисков, на которой мы непосредственно основываемся, – это понятие абстрактного типа данных из фундаментальной работы 1974 года. Конечно, была еще работа по языку CLU в MIT, также оказавшая большое влияние. Но принцип подстановки Лисков никогда не казался мне каким-то открытием.

Как контрактное программирование помогает в работе команды разработчиков?

Бертран: Оно позволяет участникам команды знать, чем занимаются их коллеги, не требуя вникать в то, как они это делают. Это дает возможность получать мгновенные снимки продуктов всех команд, исходя из одной лишь спецификации и не привязываясь к конкретному выбору представления. Оно также очень удобно для менеджеров.

Не возникает ли опасность чрезмерной определенности решения?

Бертран: Нет, на самом деле нет. Риск всегда в недостаточности определенности. Контракты редко бывают связаны с излишней специфицированностью. Такой риск возникает, если слишком рано начинают реализацию, покидая уровень спецификаций, но с контрактами этого случиться не может, потому что они описывают намерения, а не реализацию. Проблема спецификаций на основе контрактов противоположная: слишком многое остается недосказанным, потому что трудно специфицировать все полностью.

Я читал, что при использовании контрактов код не должен проверять условия контракта. Вся идея в том, чтобы затруднить возможный отказ кода. Не можете ли вы пояснить это решение?

Бертран: Видимо, многие претендуют на применение принципов контрактного программирования, не осмеливаясь применять это правило. Идея очень проста. Она касается предусловий. Если у вас есть предусловие для процедуры, в котором сказано «вот условия, которые нужно выполнить», то код самой процедуры не должен проверять контракты: вся ответственность за проверку контрактов на этапе исполнения – исходя из того, что в клиентах могут быть ошибки и они не обеспечат выполнение предусловия, – осуществляется в каком-то другом месте. Она возлагается на некий автоматический механизм, который будет использован во время тестирования и отладки. Но если у вас в коде есть и предусловие, и проверка этого условия и что-то неверно, то вы делаете одно и то же дважды; это значит, что вы не можете решить, на кого возложить ответственность за соблюдение условия, ограничения – на клиента или поставщика. Это действительно проверка того, что здесь применяется на самом деле, – контрактное программирование или какая-то разновидность защитного программирования? Готовы ли вы убрать проверки? Мало у кого хватает духа это сделать.

В контрактном программировании есть очень четкое правило, согласно которому предусловие – это ограничение, налагаемое на клиента, на того, кто осуществляет вызов, поэтому если нарушено предусловие, в этом виноват клиент. Подпрограмма за это не отвечает. Что касается постусловия, то за него отвечает поставщик, процедура. Если у вас есть предусловие, подразумевающее ответственность вызывающего, но затем его проверяет сама процедура, значит, вы в нерешительности и собираетесь написать массу бесполезного кода. Это, конечно, очень опасно, особенно потому, что этот код часто не пройдет процесс тестирования и отладки во время разработки. На самом деле, это лишь вопрос серьезного отношения к спецификации.

Насколько важно различие между спецификацией и реализацией?

Бертран: Это действительно хороший вопрос. Это различие очень существенно, но оно относительно. То есть абсолютно невозможно сказать, что это абсолютная спецификация или что это абсолютная реализация. Одна из особенностей ПО заключается в том, что какой бы программный элемент вы ни взяли, он будет спецификацией для чего-то более конкретного и реализацией для чего-то более абстрактного. Возьмите даже конструкцию, которая выглядит совершенно как реализация, скажем присваивание := A+1 или A := B. Большинство скажет вам, что это чистая реализация. Но если вы пишете компилятор, то она для вас станет спецификацией, которая будет развернута в десяток команд машинного языка или Си. Различие важно, но что действительно сложно в ПО, так это то, что при достаточно большом размере для определенного масштаба технология написания реализации становится очень похожей на технологию написания спецификации.

Например, все, кто пишет формальные спецификации, сталкиваются с поразительным явлением: когда пишешь достаточно крупную формальную спецификацию, приходишь к тому, что делаешь вещи и задаешь себе вопросы, удивительно похожие на то, что делаешь и о чем себя спрашиваешь, когда пишешь настоящие программы. Поэтому разница всегда относительна. Причина в том, что в программировании мы не работаем с физическим материалом. Мы не работаем с конкретными, ощутимыми, материальными элементами. Все, с чем мы работаем, – это абстракции, поэтому разницу между реализацией и спецификацией, по существу, составляет один уровень абстрагирования. Поэтому обычно бессмысленно указывать, что нечто является реализацией, а не спецификацией. Но можно сказать, что *X* является спецификацией *Y*; или, по-другому, *Y* является реализацией *X*. Это утверждение, которое имеет смысл: его можно опровергнуть. Но утверждения «*X* является спецификацией» или «*X* является реализацией» не могут быть опровергнуты. На них нельзя точно ответить «да» или «нет».

Как язык программирования связан с конструкцией программ на нем?

Бертран: Одна из действительно уникальных сторон Eiffel – одно из очевидных свойств ПО, которое почти никто не считает ни очевидным, ни просто верным, – это полная непрерывность связи между идеей и реализацией. Мы называем это цельностью разработки (seamless development), и это, возможно, самый важный аспект Eiffel. Все остальное направлено на его поддержку. Из этой идеи, например, следует, что между проектом и реализацией фактически нет разницы. Реализация, если перефразировать известное выражение, – это ведение проектирования другими средствами. Разница только в уровне детализации и уровне абстракции. В частности, Eiffel в той же мере предназначен служить языком анализа и проектирования, в какой быть языком реализации. В общем и целом, это, по существу, метод, а не язык, но та его часть, которая является языком, служит как для анализа и проектирования, так и для реализации.

Те, кто работают с Eiffel, обычно не пользуются UML и подобными ему средствами, которые Eiffel-разработчику представляются больше разговорами, имеющими мало пользы для программирования. Eiffel – это инструмент, предназначение которого – помочь вам, – вы говорите о проектировании, но я бы сказал, что сначала на уровне спецификаций и анализа, затем при проектировании и потом в реализации, – поддержать вас в этом процессе. Но принципиальной разницы между этими задачами нет – согласно моим представлениям и представлениям разработчиков Eiffel в целом.

Еще нужно отметить, что язык должен быть как можно скромнее. Сегодня есть много языков, которые я называю высокомерными, с массой необычных символов и условностей, которые нужно изучить, чтобы быть допущенным во внутренний круг. Например, многие из доминирующих сегодня языков фактически происходят от Си. Они стали результатом ряда добавлений и исключений из Си, и чтобы овладеть ими, нужно разобраться с большим количеством багажа.

Идея Eiffel – не стану утверждать, что Eiffel лишен всякого багажа, но все же его немного – в том, что если вы проектируете, то думаете о проекте, а не о языке. Лучшей похвалой, услышанной мною от пользователей Eiffel, были их слова о том, что, используя этот язык, они думают только о своей задаче и ни о чем больше. Это лучшее влияние, которое язык может оказать на проектирование.

Не пробовали ли вы искать другие решения помимо действий с объектами на уровне языка? Скажем, компоненты типа мелких утилит UNIX, из которых можно строить сложные функции с помощью конвейеров. В конце концов, если вы пишете кому-то письмо и вам нужно

что-то описать, вы же не пользуетесь французскими, итальянскими, английскими или какими-то еще объектами?

Бертран: Конечно, механизмы UNIX очень элегантны, но они слишком мелки для той работы, которая нужна для создания большой программы. Мой опыт показывает, что объекты – это единственный механизм, доказавший возможность своего масштабирования для крупных систем. Единственный другой подход, который я, возможно, стал бы рассматривать, – это функциональное программирование, но я не уверен в его пригодности. Идея заманчивая, очень элегантная, и у нее можно многому поучиться, но на высшем уровне она проигрывает в сравнении с объектами. Объекты – или следует говорить «классы» – значительно эффективнее для описания крупномасштабной структуры системы.

Аналогию нужно проводить не столько с естественными языками, сколько с математикой. Кроме того, это скорее классы, чем объекты. Классы – это не что иное, как перенос в программирование понятия структуры, успешно действующего в математике: группы, поля, кольца и так далее. То есть математика берет объекты, природа которых может быть очень различной, например числа и функции, и показывает, что в обоих случаях у вас одинаковая структура, определяемая операциями с одинаковыми свойствами. Затем вы извлекаете из этого одно абстрактное понятие, скажем группы, моноида или поля. Эта идея успешно применяется в математике последние 200 лет. В этом отношении классы, или объекты, являются не столь уж новым понятием. Это прямой перенос обычного понятия математической структуры.

Многие современные системы состоят из компонентов, размещенных в сети. Должны ли в языке быть отражены такие сетевые аспекты?

Бертран: Это желательно, и в Eiffel можно это делать, но я бы не стал утверждать, что Eiffel в этом сильно преуспел. Сейчас ведется множество разработок в области динамического обновления и параллелизма, результаты которых мы вскоре увидим, но пока их нет. Думаю, это важный вопрос. Можно спорить, к чему он относится – к языку или реализации, но какая-то поддержка в языке необходима.

Какую вы видите связь между объектно-ориентированной парадигмой и параллелизмом?

Бертран: Думаю, параллельная обработка данных имеет чрезвычайно большое значение. Я много об этом писал. В частности, есть много литературы, посвященной SCOOP – разработанной нами модели объектно-ориентированного параллельного программирования. Суть в том, что элементарные подходы оказываются недейственными. Довольно часто говорят, что параллелизм и объекты – это почти одно и то же, это должно замечательно работать, объекты параллельны по своей сути, и счи-

тают, что и делать ничего не нужно. Все совершенно не так. Объединить идеи объектной ориентированности с параллелизмом на элементарном уровне не получится.

Скажу несколько слов о SCOOP. В основе лежит понимание того, что стандартное понятие контракта не может одинаково интерпретироваться в параллельном и в последовательном контекстах. Идея SCOOP в том, чтобы взять модель последовательного ООП и построить для нее минимальное расширение, которое будет поддерживать параллелизм. Этот путь весьма отличен от тех, которыми идут все остальные. Например, если взять исчисление процессов, то там выбран прямо противоположный подход: выяснить, какая система параллельности лучше, а потом устроить поверх нее программирование. Получается нечто, сильно отличающееся от обычных способов программирования. SCOOP учитывает, что человеку трудно думать параллельным образом, последовательно он думает гораздо эффективнее, поэтому SCOOP стремится скрыть сложность параллельности в реализации, в модели. В результате появляется возможность программировать параллельным способом, который очень близок к последовательному программированию и позволяет сохранить привычный образ мышления.

На каком уровне нужно решать проблему параллелизма? Например, JVM почти прозрачным образом управляет некоторыми ее аспектами.

Бертран: Очевидно, потоки Java очень удобны во многих приложениях, но они не очень хорошо согласуются с объектно-ориентированной природой вещей. По сути, это семафоры в понимании Дейкстры. На самом деле, в Eiffel есть библиотека EiffelThreads, которая делает примерно то же самое. Думаю, каждому должно быть ясно, что такие решения годятся на какое-то время, но не масштабируются. Слишком много сохраняется возможностей для возникновения гонки и взаимной блокировки. Задача в том, чтобы программисты были автоматически защищены от таких проблем, а это требует работы на более высоком уровне описания. Как вы сказали, это означает, что все большую часть работы нужно переложить на реализацию.

Многократное использование и универсальность

Как в Eiffel решается проблема модификации и развития программ?

Бертран: Расширяемость, наряду с многократным использованием, с самого начала была одной из наших главных целей. До некоторой степени это причина продолжения работы над Eiffel, потому что, как я говорил, мы сначала разрабатывали Eiffel как инструмент для внут-

реннего использования, а не как товар на продажу. К тому, чтобы пересмотреть нашу позицию и сделать Eiffel более широкодоступным, нас привели высказывания разработчиков о том, что в отличие от тех языков, которыми они пользовались раньше, им теперь гораздо проще менять свои решения, не страдая от своей нерешительности.

Думаю, здесь важны несколько аспектов. Во-первых, в Eiffel тщательно скрывается информация, чтобы одни модули не видели изменений в других. Для потомков информация не скрывается, потому что в этом нет смысла, но от клиентов скрывается. Например, совершенно поразительно и несколько неприятно видеть, что в новейших объектно-ориентированных языках все еще можно непосредственно присваивать значение атрибуту – полю объекта. В Eiffel вам такое не позволено, потому что это нарушает правило скрытия информации. Для программ это катастрофа.

Далее, механизм наследования очень гибок и позволяет писать программы путем модификации существующих шаблонов. Универсальность создает еще один уровень гибкости.

Отсутствие в языке механизма, который выше класса, дает возможность комбинировать классы очень гибким образом. Контракты также играют положительную роль, потому что когда вы вносите в программу изменения, очень важно знать, что́ вы меняете, в частности, касаются ли изменения спецификации или только реализации. При внесении в программу изменений вы должны решить, будут ли эти изменения чисто внутренними, не затрагивающими контрактов, – и тогда вы знаете, что они никак не повлияют на клиентов, – или же изменения коснутся также контрактов, и тогда, конечно, нужно в точности знать, каким образом. Тем самым устанавливается уровень детализации для контроля размера изменений.

Как должны относиться к многократному использованию разработчики? Я задаю этот вопрос, потому что ряд тех, у кого я брал интервью, по сути сказали, что даже если вы строите классы, об их повторном использовании лучше позабыть, потому что оно требует слишком много труда, и что задумываться о многократном использовании стоит лишь тогда, когда обнаружится, что вы регулярно используете некий класс в контекстах разного типа. Вот тогда и стоит потратить лишнее время, сделав класс многократно используемым.

Бертран: Думаю, это относится к тем случаям, когда вы не сильны в вопросах многократного использования или вообще новичок. Верно, если у вас нет опыта повторного использования, то при попытке сделать свою программу более общей, чем того требуют текущие технические требования, вы потратите массу времени и можете не достичь успеха. Но я утверждаю, что если вы освоили повторное использование, если

у вас большой опыт работы с повторно используемыми компонентами других разработчиков и есть собственный опыт создания повторно используемого ПО, то вы успешно справитесь с задачей.

Мне кажется, что многие допускают ошибку, поскольку не понимают, что в повторном использовании есть два аспекта и один должен предшествовать другому. Это аспект потребления и аспект изготовления. Как потребитель, вы просто используете существующее ПО в собственных приложениях; многие делают это в основном для экономии времени. В аспекте изготовления вы делаете свое собственное ПО более удобным для повторного использования. Если вы решите сразу стать производителем повторно используемого ПО, вас ждет неудача, потому что такое производство действительно требует специальных технологий. Вы потратите массу времени, делая свою программу более универсальной, но вам придется гадать, в каком направлении она может быть обобщена позднее, и ваши догадки окажутся, скорее всего, неверны.

Однако если начать с более скромной позиции потребителя – изучить высококачественные библиотеки для многократного использования, способы их создания и проектирования, имеющиеся у них API, – тогда можно применить изученный вами стиль к своим собственным программам. Нечто очень похожее есть в мире Eiffel. Люди учатся программировать на Eiffel, изучая стандартные библиотеки, такие как EiffelBase или графическая библиотека EiffelVision. Это библиотеки высокого качества, которые могут служить моделью для хороших программ. Изучив их, вы сможете применить те же принципы в собственных программах и сделать их гораздо лучше, в частности более пригодными для многократного использования. Вот в таком направлении нужно идти: начать как потребитель и на основе полученного опыта стать производителем. Мой опыт показывает, что этот путь может быть успешным. Эту мысль я развил в своей книге «Reusable Software» (Повторное использование программного обеспечения), изд. Prentice-Hall, 1994.

При таком подходе можно сделать свое ПО многократно используемым. С точки зрения ускоренного и экстремального программирования о повторном использовании заботиться не нужно, потому что это пустая трата времени. Думаю, это относится только к тем, кто не умеет писать многократно используемые программы, потому что не потрудился изучить по хорошим образцам, как это правильно делается.

Возможно, это также зависит от используемого языка программирования.

Бертран: Безусловно зависит – тут меня не нужно долго уговаривать. При проектировании Eiffel решалось три основных задачи. Первой была, конечно, корректность или, более общо, надежность. Второй была расширяемость, простота модификаций, а третьей – многократ-

ное использование кода. Поэтому повторное использование учтено всюду. Например, совершенно поразительно, что обобщенные классы у нас появились с самого начала. Это было совершенно необходимо для многократного использования, но долгие годы над нами постоянно смеялись. На первой конференции OOPSLA в 1986 году у нашей компании был стенд с соответствующим плакатом, и подходя к стенду люди смеялись, видя слово «genericity», которого, по их словам, просто нет в английском языке. Никто не понимал, о чем идет речь.

Через несколько лет в C++ появились шаблоны. Когда в 1995 году появилась Java, там не было генериков, и все говорили, что они не нужны, что это только запутывает объектно-ориентированные языки. Как бы то ни было, спустя десять лет генерики появились, но сделано это было сложным и, на мой взгляд, не вполне удовлетворительным способом – не столько из-за плохого проектирования, сколько из-за требований совместимости. Когда вышел C#, я не поверил глазам – генерики опять отсутствовали, несмотря на опыт, полученный с Java, но, конечно, обобщенные классы были добавлены – лет семь спустя.

Все это было предусмотрено в Eiffel с самого начала в расчете на многократное использование. Конкретные детали механизма наследования, конкретное сочетание переименования, переопределения, отмены определения, имеющиеся в Eiffel, механизм множественного наследования – все это оправдано и мотивировано многократным использованием. Конечно, важную роль играют контракты. Я уже сказал, что не понимаю, как можно программировать без контрактов, но что еще труднее понять – как могут существовать предположительно многократно используемые компоненты без четкой спецификации того, что эти предположительно многократно используемые компоненты делают.

Понемногу люди стали это понимать. Было объявлено, что в .NET 4.0 будут библиотека контрактов Code Contracts и все основные библиотеки; Mscorelib должна быть заново задокументирована и спроектирована с контрактами. Всего лишь через 23 года после Eiffel. Наконец-то начинают понимать, что не может быть многократного использования без контрактов. Потребовалось время. Разумеется, за это время Eiffel продолжал вводить новые идеи, чтобы оставаться лидером.

Когда и как вы поняли, что универсальность столь же важна, как классы?

Бертран: Это представление и понимание, как мне кажется, возникли скорее, в академическом контексте, а не вследствие потребностей производства. Я в своей профессиональной деятельности больше времени провел в промышленности, но немного потрудился и на академической ниве. Пару раз, в 1984 и 1985 годах, я читал в университете Санта-

Барбары курс «Современные концепции в языках программирования». Его содержание было весьма вольным.

Я хотел посмотреть, что происходило в то время на передовой языков программирования. Я включил в свой курс Аду, о которой много спорили, и Симулу, о которой никто не спорил, но я видел, что объектно-ориентированным идеям этого языка принадлежит будущее.

Эта проблема не могла не возникнуть при чтении данного курса, потому что одну неделю я рассказывал об универсальности, а другую – о наследовании, или наоборот. Естественно, возник вопрос, как эти вещи соотносятся между собой. Не помню, спросил ли об этом кто-то из студентов, хотя могло быть и так. Помню, как спрашивал себя сам: «Что же, я так и буду одну неделю доктором Универсальность, а другую неделю – мистером Наследование?»

Это вынудило меня задаться следующим вопросом: «Чего такого нельзя сделать с помощью одного, что можно сделать с помощью другого?» – и наоборот. Конечно, это было результатом многих предшествующих обсуждений и размышлений, но в обсуждении языков программирования общество разделилось на два лагеря: тех, кто считал, что Ада удовлетворяет все мыслимые потребности языков программирования в гибкости, и небольшую группу открывших для себя ООП и наследование.

В рабочих группах велись обычные споры: «На моем языке это получается лучше» – «Нет, на моем». Насколько могу судить, никто не сделал следующий шаг и не попытался выяснить, какая связь имеется между двумя механизмами и как точно сравнить силу их выразительности.

В своем курсе я провел для них некоторый сравнительный анализ. Потом поступило предложение представить доклад для первой конференции OOPSLA, и естественным было выбрать эту тему.

Я сел и написал, что думаю по этому поводу, и получился доклад «Универсальность и наследование» в трудах первой конференции OOPSLA.

Вы опубликовали этот доклад еще до того, как эта проблема была выявлена в наиболее распространенных ОО-языках. Даже в Smalltalk она никак не рассматривается.

Бертран: В Smalltalk это безразлично, потому что в Smalltalk есть динамическая типизация, и ничего этого не нужно. На самом деле, это было очень странно. Вспоминаю слова Шопенгауэра – что-то вроде «сначала они над вами смеются, а потом они...»

...Сначала они вас не замечают, потом они над вами смеются. Да.

Бертран: На самом деле, буквально так и происходило. На первой конференции OOPSLA я представил свой доклад под вывеской Универси-

тета, так что это была действительно научная работа, а у компании был также стенд, очень кустарный, потому что у нас совсем не было денег. Плакаты были самодельными, некоторые рукописными.

К нашему стенду подходили люди и, насмеявшись, приводили друзей, чтобы они тоже посмеялись. Как я уже говорил, одним из поводов для смеха для них было слово genericity (универсальность). Там были большие парни из HP – действительно большие, я не шучу. Некоторые пиджаки из HP подходили по несколько раз, приводя все новых приятелей, чтобы показать это слово и им. «Как это произносится? По-французски, что ли?» Они громко зачитывали его вслух на разные лады: «Наверно, это generisissyty!» – и так далее.

Таков был дух времени. А через 20 лет мне присылают на отзыв статьи, где говорится, что универсальность появилась благодаря Java. Жизнь удивительна.

Проверка языков

Вы, я слышал, говорите по меньшей мере на трех языках: английском, французском, конечно, и немецком. Ваша многоязычность повлияла на вас как на разработчика языков?

Бертран: Если коротко, то да. На самом деле, немецким я владею не очень хорошо. Мой родной язык французский. С английским стараюсь изо всех сил. Я довольно бегло говорю по-русски. У меня даже есть диплом магистра по русскому языку, хотя я говорю далеко не на том уровне, который должен быть у человека с дипломом магистра. Я прилично говорю по-итальянски. Я без особого труда читаю лекции по-русски. По-итальянски я могу читать лекцию в течение 15 минут, но потом у меня начинается перегрев мозга. Это было сказано для большей точности, но ответом на ваш вопрос определенно будет «да».

В компьютерную науку я попал частично из-за своего интереса к языкам. Понимание того, что одно и то же можно сказать по-разному, что нет взаимно однозначного соответствия, что к вещам можно подходить по-разному, что иногда правильным будет употребить существительное, а иногда нюансы лучше передает глагол, – все это оказало определенное влияние и сильно помогло мне. Я также полагаю, что тот, кто знает хотя бы один иностранный язык, лучше говорит на своем родном языке.

Кроме того, обычно во всяких технических начинаниях, в частности в программировании, вы не просто пишете программы, но и пишете на английском или каком-то другом естественном языке. Некоторые уси-

лия, потраченные на улучшение навыков письма или изучение одного или нескольких языков, хорошо окупаются.

Вы подходите к программированию с точки зрения математики или лингвистики либо как-то их комбинируете?

Бертран: Мне следовало бы подходить к программированию более математически, чем я это делаю. Я убежден, что через 50 лет программирование станет разделом математики.

Есть люди, уже давно продвигающие математический подход к программированию. На практике он не утвердился, за исключением отдельных областей, где строят небольшие, жизненно важные системы и другого выхода не остается. В конце концов, программирование – это действующая математика: математика, которая может быть интерпретирована машиной. Думаю, в будущем характер программирования станет еще более математическим.

Что касается моего личного подхода, то я охарактеризовал бы его как сочетание того, что называется лингвистическим подходом, более самопроизвольного, творческого, дискурсивного подхода, и старания соблюдать строгость и математичность. Конечно, в Eiffel влияние математики сильнее, чем в большинстве других существующих языков, исключая, пожалуй, функциональные, вроде Haskell.

Я спрашивал у многих разработчиков, как они относятся к тому, чтобы начать с маленького строгого базового языка и затем развивать его, – возьмите лямбда-исчисление. Можно организовать любые вычисления, если вы можете применять функции. Как вам нравится такой подход?

Бертран: Не думаю, что он очень удачен. Проблема программирования в том, чтобы сочетать науку и технику. С одной стороны, программирование существенно опирается на науку, и, как я сказал, в принципе, программирование – это математика. Но другая сторона – техническая. Некоторые из имеющихся сегодня программ сложнее едва ли не любого артефакта, когда-либо созданного человечеством. Размер больших операционных систем, таких как дистрибутивы Linux, Vista, Solaris, исчисляется десятками миллионов строк кода, иногда больше 50 миллионов. Это невероятно сложные инженерные конструкции. Многие из проблем, с которыми они сталкиваются, являются по существу инженерными проблемами.

Разница между наукой и техникой, если несколько упростить дело, состоит в том, что в науке вам нужно лишь несколько толковых идей; в технике приходится заботиться об огромном количестве деталей, большинство из которых не очень сложны, но их очень много. Таким

образом, с одной стороны немного остроумных идей, а с другой – множество не очень сложных вещей. Для программирования характерно, что требуется и то, и другое. Может показаться, что я противоречу тому, что только что сам сказал, а именно, что через несколько десятилетий программирование станет по существу математическим, но, думаю, здесь нет противоречия. Объясню почему.

В основе своей программирование – это всего лишь математика, но нужно обратить внимание на слова *в основе*. На практике программирование включает в себя также всякие инженерные задачи, которые приходится решать. Если вы пишете операционную систему, то должны гарантировать, что тысячи драйверов для различных устройств, написанных неискушенными программистами, не вызовут краха вашей операционной системы. Вам нужно позаботиться о том, чтобы диалоговые окна, задействованные в работе, использовали многочисленные естественные языки. Нужно создать очень сложный набор механизмов для пользовательских интерфейсов: основные идеи могут быть просты, но детали неисчислимы.

Сложность программирования имеет двойственный характер: это научная сложность и техническая сложность. Наличие очень сильной математической основы, например лямбда-исчисления, поможет вам справиться с первой частью, но не со второй, и поможет с той частью, которая на сегодняшний день лучше всего понята. Лямбда-исчисление очень хорошо подходит для моделирования базовой части языка программирования на уровне, скажем, Паскаля или Лиспа, но современные языки программирования гораздо более претенциозны.

В конечном счете, мы должны все свести к очень простым математическим принципам, но одних математических принципов недостаточно для решения задач, встающих при создании крупных современных программ.

Это то, что отличает академические языки программирования от промышленных языков программирования?

Бертран: Совершенно верно. Когда создавалась Ада, этот язык критиковали за огромный размер и сложность, и его создатель Жан Ишбиа сказал в одном интервью: маленькие языки решают маленькие задачи. В этом есть известная доля правды. Думаю, он, в сущности, отвечал на критику таких людей, как Вирт, который глубоко верил, что «малое прекрасно», но он прав и в целом.

Есть ли что-то между структурным программированием и ООП? Вы сказали, что считаете структурное программирование хорошим способом организации маленьких программ, а ООП – хорошим способом

организации больших программ. Попадает ли какая-то часть программ по размеру в промежуток между тем и другим?

Бертран: Нет, я бы не стал пользоваться ничем, кроме ООП. В принципе, я изучал то и другое одновременно и не вижу причин использовать в своих разработках технологии не-ООП, за исключением, может быть, каких-то маленьких одноразовых скриптов. «Объектная ориентированность» означает просто применение к программам математического понятия структуры, против чего нет веских аргументов.

Программы бывают либо маленькими, либо большими.

Бертран: Нет веских причин не использовать классы. Я точно не знаю, что об этом думал Дейкстра. Он не был большим сторонником ООП, но мне также неизвестно, чтобы он его критиковал: когда ему что-то не нравилось, он мог высказаться очень откровенно и резко.

Вы сказали, что в Eiffel вам был очень нужен механизм поточной сериализации. Можно поинтересоваться зачем?

Бертран: Первым сделанным нами приложением был, как я уже сказал, Smart Editor, выпущенный под торговым названием ArchiText. От редактора всегда требуется возможность работы с небольшой структурой данных, находящейся в памяти, с ее последующим сохранением. Можно каждый раз проводить парсинг текста, а потом обратную операцию, но это абсурдный способ. Допустим, вы редактируете текст и восстановили для текста небольшую структуру, и у вас есть абстрактное синтаксическое дерево или другое эффективное внутреннее представление: зачем вам превращать его в текст, а в следующий раз заново создавать это дерево? Нужно постоянно использовать эту абстрактную структуру. Если нужно сохранить ее, нажмите кнопку, и поточный механизм все сделает за вас.

Так было с первым приложением, но и все последующие приложения оказывались такими же. Если вы пишете компилятор, происходит то же самое. Допустим, ваш компилятор выполняет несколько проходов. Каждый проход берет структуру, полученную при предыдущем проходе, декорирует ее, немного манипулирует с данными и записывает результат на диск. Нет надобности каждый раз делать эту запись каким-то особым способом. Нужно просто нажать на кнопку. Есть десятки приложений, в которых требуется такая вещь.

Можно вставлять промежуточные этапы.

Бертран: Верно. Вас не ограничивает какая-то одна конкретная структура обработки.

Вы использовали выражение «цельность разработки» и сказали, что это одна из принципиальных идей в Eiffel. Что это такое?

Бертран: Идея в том, чтобы обеспечить единство на протяжении всего процесса создания ПО: единый набор задач, единый набор решений этих задач и единая нотация для описания результатов – к чему и стремится язык Eiffel.

Это полностью противоречит тенденциям в развитии нашей отрасли за последние 20 лет, которые я в данном случае не одобряю. Это тенденции к разделению, потому что оно выгодно для бизнеса, поскольку вынуждает людей покупать инструменты для анализа, инструменты для проектирования и IDE, а также на каждом уровне пользоваться услугами консультантов.

Кроме того, как мне кажется, люди не понимают, что, скажем, спецификация по существу в такой же мере ПО, в какой им является реализация. Исторически сложилось так, что языки программирования относились к очень низкому уровню, поэтому мысль, что можно думать на языке программирования, была абсурдна, но с нашими сегодняшними возможностями нет причин сохранять прежние расхождения.

Людям трудно отказаться от мировоззрения, сформировавшегося в эпоху перфокарт, когда сегодня отдаешь свою программу как пакетное задание, а завтра приходишь за результатом, и горе тебе, если при компиляции возникала ошибка. Поневоле приходилось очень тщательно продумывать все заранее.

Бертран: Верно. Нет ничего плохого в том, чтобы подумать заранее, но это не значит, что на разных этапах у вас должны быть совершенно разные системы мышления, а значит, разные инструменты и языки.

Люди относятся к этому едва ли не привлекая моральные критерии. Есть подспудное представление, что анализ высок и благороден, а реализация грязна и презренна. До некоторой степени так и было, когда приходилось программировать на языке ассемблера или даже на таких языках, как Фортран. Фортран был замечательным достижением для своего времени, но это не тот язык, на котором у кого-то могло возникнуть желание думать. Отсюда эта идея, что первоначальное обдумывание – дело благородное, а потом кто-то – возможно, другой человек – осуществит окончательную реализацию, засучив рукава и сделав низкую работу – все равно что открыв капот и повозившись в грязи.

Так было до какого-то времени, но для современных языков, включая, конечно, Eiffel, такое положение не обязательно. Мы не пытаемся сделать наши методы анализа и проектирования более ориентированными на реализацию, а подходим с другого конца. Мы отталкиваемся от программирования и делаем свой язык программирования настолько выразительным, настолько красивым, настолько способствующим продуктивному стилю мышления, что можем с его помощью провести всю

необходимую работу. Первые версии программы получатся абстрактными и описательными, но последующие станут более функциональными и в конце концов будут исполняться. Нет необходимости разрывать процесс на отдельные части.

Такой подход противоположен, например, разработке, управляемой моделью, которая требует создать модель, а потом нечто совершенно отличное от нее – программу.

В таком случае ваша модель, будь она видима или невидима, сродни исходному коду, поскольку именно его вы генерируете, а исходный код – это, в некотором роде, запоздалая мысль.

Бертран: Это хорошо, пока у вас есть полная гарантия, что никто и никогда не тронет исходный код – ни для отладки, к примеру, ни для модификации. Так всегда говорят вначале: что вы, мы работаем только с моделью, а исходный код никто не тронет, но на практике часто происходит совсем иное.

Исходный код – это артефакт проектирования.

Бертран: Возникает вопрос: а что вы отлаживаете? Если вы действительно отлаживаете модель, вас не в чем упрекнуть. Но тогда это значит, что то, что вы называете моделью, на самом деле является программой. Пусть на каком-то очень высоком уровне, но это программа. Вы разработали язык программирования очень высокого уровня, а потом вам нужно построить для него законченную среду разработки.

С другой стороны, если отлаживаемая вами программа сгенерирована, то возникают все сложности раздельной разработки. Вот главный вопрос, который нужно задать тем, кто заявляет, что ведет разработку, управляемую моделью: какую версию вы отлаживаете? Какую версию вы изменяете, когда клиенту нужна новая функция?

Можно ли доказать правильность программы или контракты существуют только для тестирования?

Бертран: Механизм контрактов был включен в Eiffel с самого начала, и идея всегда была в том, что в долгосрочном плане контракты станут использоваться для доказательства программ, но в краткосрочном плане они будут использоваться для тестирования. Одним из результатов этого является возможность включить контроль контрактов во время исполнения, и тогда при нарушении контракта будет генерироваться исключение.

Следующим шагом, на который у нас ушло много времени, было сказать: положим это в основу полностью автоматизированного тестирования. Это исследование, которым мы занимались несколько последних лет с моей группой в ETH, и теперь оно интегрировано в инструменты.

Главный вопрос был: трудно ли тестировать?

Во-первых, мы должны автоматизировать процесс тестирования. Эту задачу решает JUnit и все замечательные инструменты, которыми теперь пользуются для автоматизации этого процесса.

Есть еще две вещи, которые нужно автоматизировать и которые, по существу, никто больше не автоматизировал, но они самые трудные. Первая из них – это генерация контрольных примеров (test cases), потому что их нужно как-то создавать, а их могут быть тысячи, десятки тысяч, сотни тысяч. Третья и последняя, даже если вы сделали первые две, – это проблема прогона этих десятков тысяч тестов, когда кто-то должен решать для каждого из них, пройден он или нет, и это тоже нужно автоматизировать с помощью оракулов тестирования (test oracles). Что можно сделать с помощью контрактов, так это устроить эти оракулы тестирования, просто сказав: если это постусловие или инвариант выполнены, то тест прошел; если это постусловие или инвариант нарушены, тест не прошел. Это делается автоматически.

Остается генерация тестов, для чего мы используем метод, который поначалу кажется очень глупым, но на самом деле работает замечательно: случайное или квазислучайное генерирование. Этот инструмент создает объекты почти случайным образом и затем вызывает все подпрограммы, все методы соответствующих классов с почти случайными аргументами. Мы просто ждем. Мы называем это «обеденным тестированием». Нажимаем на кнопку, запускаем тестирование и уходим на обед, а вернувшись через час, смотрим, где нарушены постусловия.

Действует замечательно, поскольку по сути вам ничего не нужно делать. Вы просто ждете, пока механизм автоматической генерации испытывает вашу программу. Но это можно делать только для языка со встроенными контрактами, а иначе придется добавлять контракты и механизм их проверки. Если у вас есть поддержка контрактов, то это действительно замечательный способ тестировать ваши программы.

Что вы думаете о доказательстве правильности программ? Есть ли в нем польза? Или это несбыточные мечты?

Бертран: Это все больше становится реальностью. Что касается научной работы, которую я веду, то на это направлена значительная часть наших сил. Результаты довольно скудные, потому что основные идеи возникли уже почти 40 лет назад, после публикации Тони Хоаром в 1969 году статьи об аксиоматической семантике. Практическая реализация очень сильно затянулась, но это не несбыточные мечты.

Конечно, в последние 5 или 10 лет был достигнут значительный прогресс. Очень интересна работа с Spec# в исследовательском центре Microsoft. Такую работу мы ведем с Eiffel в ETH, о которой мало извест-

но, потому что мы поставили перед собой очень большие задачи и должны решить множество проблем, чтобы действительно удивить мир. Но я считаю ее перспективной.

Есть еще работа над SPARK. Это очень интересная разработка. Они действительно могут создавать программы, правильность которых доказуема. Загвоздка в том, что это язык, на котором никто не хочет программировать. Они считают его подмножеством Ады. На самом деле, это подмножество Паскаля с модулями. Плата за это – отказ от всех радостей жизни. Никаких классов, динамического создания объектов, универсальности, наследования и указателей. Взяв такой сокращенный язык, они смогли создать эффективные инструменты для доказательства. Это существенное достижение, потому что оно позволяет создавать важные системы, например для военной и аэрокосмической областей, и доказывать их корректность. Вряд ли вы захотите программировать на этом языке. И я не захочу, и 99% всех программистов не захотят. И все же это большое достижение.

Наш проект по доказательству в ETH аналогичен, но относится к языку, на котором люди захотят программировать. Трудность в том, чтобы включить все механизмы языка программирования, которые нам полюбились. Конечно, как только вы включаете, скажем, указатели, так появляются алиасы, а это сразу существенно усложняет задачу. Проблема уже не в том, чтобы доказывать программы. Проблема в том, чтобы доказывать правильность программ, написанных на реалистичном современном языке. И она будет решена.

Нужно понимать, что в сущности мы занимаемся нерешаемыми проблемами. В конечном счете, в программах всегда останутся части, доказать правильность которых мы не сможем, и потому мы также занимаемся другим вопросом – тестированием. Здесь развитие идет гораздо быстрее. Это явно самое замечательное, что нам удалось сделать за последние два или три года, и теперь это полностью интегрировано в среду. Так что одно из преимуществ контрактов состоит в том, что тестирование можно сделать полностью автоматическим. Система тестирования Testing Framework сейчас полностью интегрирована в Eiffel Studio, и тестирование запускается, по существу, простым нажатием кнопки. Не нужно писать контрольные примеры. Не нужно писать оракулы тестирования. Вызываете Testing Framework, и она создает экземпляры классов, вызывает методы этих классов и ждет, пока какой-нибудь контракт окажется невыполненным. Другая замечательная функция Testing Framework – синтез тестов: если в программе возникает аварийная ситуация, этот инструмент автоматически создает из нее контрольный пример, который можно запускать при исправлении ошибки и включить в комплект для регрессионного тестирования.

В отношении вашего вопроса скажу, что есть доказательства и есть тесты, и обе эти стороны важны. Но доказательства определенно становятся возможными.

Доказательства и контракты — это как разные отметки на скользящей шкале.

Бертран: Как я говорил, мы начали с динамической проверки контрактов, но конечной целью всегда считалось доказательство того, что класс удовлетворяет своему контракту. До недавнего времени такой возможности в производстве не было. Люди просто не готовы к доказательствам.

Можно ли рассчитывать на появление этого в Eiffel в ближайшие несколько лет?

Бертран: Вполне. Отчасти это еще область исследований, поэтому указать точный срок трудно. Исследования по тестированию были начаты года четыре назад, и сейчас первые результаты уже включены в среду. Что касается доказательств, то появления первых результатов можно ожидать в течение примерно такого же срока – думаю, года через три.

Значит, для вас это определенно производственная задача?

Бертран: Конечно.

Вы говорили о недоказуемости участков кода. В Haskell есть понятие монад для отделения функционально чистого кода от нечистого — того, у которого есть побочные эффекты. Нельзя ли с помощью аналогичного механизма изолировать недоказуемый код?

Бертран: Нам придется отделить части, которые мы можем доказать, от тех, которые не можем, но вряд ли для этого будет использован механизм монад. Монады – очень интересная идея. В контексте доказательства их можно использовать иначе: они позволяют осуществить инкрементный подход, если определить базовый язык, поддерживающий доказуемость, а потом добавлять к нему более сложные конструкции вроде исключений и прочих некоторым инкрементным образом.

Управление ростом и развитием

Вы сказали, что сконструировали Eiffel примерно за день, но на реализацию своего видения у вас ушло около 20 лет.

Бертран: Главные идеи действительно очень просты, а все остальное, как говорится, комментарии. В известной мере, этим мы и занимались. Мы развивали базовую идею в течение 20 лет.

Возьмите классы, наследование, особенно множественное наследование. Возьмите универсальность, контракты и несколько принципов

языка, например очень важный принцип Eiffel, согласно которому для решения любой задачи нужно предоставить один хороший способ. Или идея высокого отношения сигнал/шум: малый размер – не самоцель для языка, но нужно отбирать функции в соответствии с тем, чтобы они добавляли как можно больше мощи и создавали как можно меньше трудностей. Отбирается два десятка таких идей, одни из которых касаются языка, а другие – метаидеи относительно его конструкции, вот и все, в сущности. Но чтобы превратить это в нечто полезное, с помощью чего можно моделировать систему противоракетной обороны или управлять миллиардами долларов, нужна техническая работа, и она требует много времени.

Ресурсы всегда ограничены, и дело не в том, большая у вас компания или маленькая, потому что новаторские идеи всегда исходят от маленьких групп. Исключение составляют проекты, по существу являющиеся техническими. Чтобы послать человека на Луну, тысячам людей нужно работать несколько лет. Или, скажем, геном человека: вы знаете заранее, как это делать, требуется лишь то, что я называю техникой. Но это исключения.

Что касается действительно новаторских идей, то мне не встречались революционные продукты, над которыми работало бы больше десяти человек, обычно их от двух до пяти. Ресурсы всегда ограничены, поэтому самое важное – решить, что нужно делать, а что нет.

Конечно, мы сделали несколько ошибок. Например, мы вложились в версию для OS/2, что стало совершенно напрасной тратой сил: лучше бы мы совершенствовали основную версию. Такие решения приходится принимать едва ли не каждый день, и иногда они бывают ошибочными.

И за эти 20 лет вы достигли такого состояния, когда вы довольны языком, когда реализация, учет всех нюансов, применимость соответствуют вашим начальным проектным целям?

Бертран: Я бы так не сказал, потому что нужно иметь известную долю нахальства, чтобы говорить такие вещи. В каком-то отношении уже первая реализация была такой, что весь мир мог ею пользоваться. Можно и чересчур поскромничать, сказав, что мы еще не достигли своей цели.

Мы каждый день работаем над совершенствованием реализации и теми вещами, которые считаем абсолютно необходимыми. Совершенства мы никогда не достигнем, во всяком случае, при моей жизни, но вопрос в том, над чем вы работаете в каждый конкретный момент, что вы считаете важным, а что второстепенным. Всегда можно вернуться к прежним решениям и посмотреть, правильно ли был сделан выбор.

Конечно, в реализации всегда были аспекты, подвергавшиеся критике, часто справедливой. С другой стороны, есть люди, которые работают с Eiffel в течение 10, 15 лет, иногда почти с первого появления реализации, и они всегда были довольны.

Поэтому, с одной стороны, я никогда не бываю удовлетворен реализацией, а с другой, думаю, что Eiffel на всем протяжении своего существования оказывался отличным решением, далеко опередившим другие, для тех, кто хотел с ним работать.

Нужно дерзко поверить, что сможешь потратить на проект 20 лет, но приходится и работать над проектом, который надеешься любить все эти 20 лет.

Бертран: Верно. Очень трудно решать, что нужно делать, а что нет. Например, мы были одной из первых коммерческих компаний, выпустивших версию для Linux. В то время такое решение казалось совершенно безумным. Сравните Linux и OS/2. Работа для OS/2 была совершенно напрасной. С другой стороны, кто-то сказал мне, кажется в 1993 году, довольно давно: «Мы тут работаем с этой штукой под названием Linux. Не могли бы вы сделать версию для Linux?» Никто в нашей компании не хотел этим заниматься, но я сказал им, что эта штука претендует быть разновидностью UNIX – у нас были версии для десятков вариантов UNIX, потому что в то время было много разновидностей коммерческой UNIX, и мы старательно трудились над переносимостью технологии, – так что попробуйте скомпилировать систему под Linux и посмотрите, что получится, и если это выльется в месяц работы, то не надо этим заниматься. Если работы на день, то, может быть, стоит попробовать. На самом деле оказалось, что работа тут вообще не нужна. Все скомпилировалось под Linux нажатием одной кнопки. Тот, кто попросил меня сделать версию для Linux, остался очень доволен, и после этого к нам вдруг пошли запросы.

С точки зрения здравого смысла заниматься Linux было глупо, а заниматься OS/2 – умно, а на практике все оказалось наоборот. Нам очень помогло то, что мы быстро выпустили версию для Linux. Такие решения принимать очень трудно, и всегда недостаточно информации, а те, кто дает вам советы, часто не знают, о чем говорят.

Вывод из этого: приходится учитывать все, что слышишь, но в конечном счете нужно хорошо знать, что делаешь, и полагаться только на себя, на собственный разум.

Какие критерии вы применяете при анализе таких решений?

Бертран: Совместимость. Согласуется ли это с идеями организации? Последовательно ли это решение? Уведет ли оно нас в сторону от наших главных идей, области компетенции, лишит удовольствия – потому что

всегда нужно получать удовольствие от того, чем занимаешься, – или это станет полезным опытом, который расширит наши знания?

Как сегодня вы определяете, какие функции добавить в Eiffel? Как вы расширяете язык?

Бертран: До 1998 года, почти до 2000–2001, за развитие языка отвечал по сути я сам, но положение изменилось с появлением комиссии по стандартизации Eiffel в Ecma, что привело к выходу стандарта Ecma в 2005 году и стандарта ISO в 2006. Формально в язык попадают те функции, которые одобрит комиссия. Отвечая более технически, скажу, что мы в крайней степени осторожно подходим к развитию Eiffel; то, что мы сделали как комиссия и как сообщество, порой довольно своеобразно.

Во-первых, у нас в языке есть такой принцип: должен быть один хороший способ решения любой задачи. Это наше главное средство для борьбы с ползучим улучшизмом. Мы не против новых функций, но мы не хотим включать в язык новые функции, которые дублируют существующие механизмы. Критерием является то, что если программисту нужно что-то сделать, у него должен быть для этого один хороший способ. Контрпримером служит наличие в некоторых языках одновременно динамического связывания и массивов указателей на функции. Программисту, особенно новичку в ООП, непонятно, каким из этих методов воспользоваться, если стоит задача вызова разных функций в зависимости от типа объекта. В Eiffel таких дилемм почти никогда не возникает. Нельзя добиться осуществления этого принципа на 100%, но мы близки к этому.

Другой руководящий принцип – максимизация того, что можно назвать соотношением сигнал/шум. Интересно сравнить это с подходом к языкам Никлауса Вирта. Вирт очень любит маленькие языки, и у него просто страх перед их раздуванием. Думаю, многие механизмы Eiffel показались бы ему чрезмерно увеличивающими язык. Мне очень нравится этот взгляд, но Eiffel основывается на несколько ином подходе. Необязательно делать малый размер языка самоцелью; скорее нужно стремиться к росту соотношения сигнал/шум, что означает очень низкий шум. «Шумом» я называю те функции языка, которые не слишком полезны, а лишь осложняют язык, мало увеличивая его выразительность, а «сигнал» означает силу выразительности. Приведу пример. Около 12 лет назад мы ввели в язык понятие «агента», оказавшееся очень удачным. Это разновидность замыканий, развитых лямбда-выражений, и были опасения, что они будут избыточны при имеющихся механизмах, но этого не произошло. Это пример крупного расширения языка, ставшего очень популярным у пользователей и сделавшего возможным то, для чего прежде не было красивых решений. Сплошной сигнал и почти никакого шума.

Третий принцип – это гарантировать, что все наши действия совместимы с целями Eiffel и духом Eiffel, в частности, повышают надежность языка и снижают возможность совершения ошибки программистами. За последние два или три года здесь произошли действительно крупные сдвиги. Думаю, Eiffel – первый коммерческий язык со свойством void-безопасности, т. е. гарантией от разыменования нулевых указателей. Это реализовано в версии 6.4, где полностью переработаны библиотеки. Данная проблема обычна для объектно-ориентированных языков, даже для Си или Паскаля, и состоит в том, что `x.f` может вызвать аварийную ситуацию, если `x` имеет значение `null`, или `void` в терминологии Eiffel. Программирующие на Eiffel теперь совсем не рискуют. Я считаю это крупным достижением, потому что тем самым устраняется большой источник проблем при выполнении программ ООП. Вот такого рода вещи мы собираемся делать, чтобы повысить надежность программных разработок.

Есть и другие принципы, но упомяну еще лишь один, связанный со смелостью комиссии по стандартизации. Мы не стесняемся вносить в язык изменения. В частности, мы без колебаний можем удалить из языка некий механизм, если обнаружим более удачный способ работы. Конечно, это делается крайне осторожно, потому что есть база клиентов, и если у клиента программа объемом в миллион строк, мы не можем позволить себе, чтобы она перестала работать, поэтому старые механизмы обычно продолжают поддерживаться в течение нескольких лет. Мы предлагаем инструменты для миграции и различную помощь, но если в какой-то момент приходим к выводу, что есть некий путь A для решения задачи и есть путь B, который приводит к тому же результату, но сам по себе лучше (в смысле простоты, надежности, расширяемости и так далее), то мы удаляем имеющийся механизм и помещаем на его место другой, который кажется нам лучше.

Как вы справляетесь с проблемами обратной и прямой совместимости?

Бертран: Это наши главные заботы. Когда у вас есть продукт, который держится на рынке достаточно долго, это вскоре становится одной из главных ваших трудностей и отнимает невероятно много времени.

Это крайне сложный вопрос, если вы не являетесь ведущим игроком на рынке. Ведущие игроки могут делать все, что угодно. Все крупные имена в нашей отрасли время от времени так поступают. Они резко все меняют, и клиентам остается только подчиниться.

Сообщество Eiffel в этом отношении имеет некоторые особенности, поскольку более открыто инновациям, чем большинство других – в частности, языковых – сообществ. Люди согласны с тем, что изменения необходимы, но поскольку пользователи Eiffel обычно смотрят в будущее

и заинтересованы в добротных, красивых и творческих решениях, они соглашаются с изменениями, даже затрагивающими миллионы строк кода. Но чего не любят ни они, ни кто-либо другой, так это когда вам приставляют к виску пистолет и говорят: немедленная модификация или смерть. Если вы так подходите к изменениям, то пользователи едва ли будут любить вас.

Стратегия, выбранная в комиссии Ecma, состоит в рассмотрении всех возможных выходов из ситуации и учете всех факторов. Если мы решили, что нужно что-то изменить, мы меняем это. Мы не отказываемся от изменений на том основании, что годами делалось иначе. Если нужны изменения, они будут введены, но при тщательном планировании. Я упрощаю картину, потому что изменения бывают в языке, в библиотеках, в инструментах, и стратегия необязательно должна быть одинаковой во всех случаях, но вот перечень основных правил:

- Провести предварительную подготовку и полностью убедиться в необходимости данного изменения.
- Составить план.
- Объяснить, зачем вы вносите это изменение. Это очень важно. Исходите из того, что вы разговариваете с разумными людьми. Если вы действительно подготовились, тщательно обдумали основания для изменения и способны рассказать о нем и убедить своих ближайших коллег, то сможете убедить и других разумных людей.
- Дать людям время. Почти всегда существенные изменения в языке проводятся в два или даже три этапа. Сначала появляется версия, в которой новый механизм факультативен, а старый установлен по умолчанию, но можно опробовать новый механизм как вариант, обычно отдельно для каждого класса, поэтому можно опробовать его на частях системы. Затем выходит версия, где установки по умолчанию меняются местами.

  Очень редко мы удаляем что-то целиком. Даже устаревшие механизмы сохраняются в качестве варианта. К сожалению, это не всегда возможно, потому что иногда старый и новый механизмы оказываются несовместимы.
- Обеспечить помощь в миграции: инструменты, библиотеки и все, что поможет перейти со старого механизма на новый.

Мы проходили через это много раз, особенно за последние 5–6 лет. Если рассмотреть историю языка Eiffel, то в ней обнаружатся два крупных переворота. Первая версия вышла в 1985–1986 годах. Вторая вышла в 1988 году, но это были по сути дополнения, поэтому проблем совместимости не возникало. Затем между 1990 и 1993 годами мы перешли

на Eiffel 3, и это явно было крупным переворотом, но преимущества были столь велики, что особых протестов не было.

Язык мало менялся, пока в 2001 году не начался процесс стандартизации. Стандарт был опубликован в 2005 году и стал стандартом ISO в 2006 году. Он ввел в язык существенные изменения, реализация которых заняла несколько лет – сейчас она почти завершена. Сегодня мы находимся в середине решения одного из наиболее тяжелых случаев данной проблемы, и это неспроста. Дело касается механизма прикрепленных типов, решающего проблему нулевых указателей, о которой я говорил выше, гарантируя, что ни один вызов `x.f` не будет выполнен, когда `x` равен `null`. Компилятор находит такие случаи и отклоняет программу, если в ней может возникнуть вызов `void` (разыменование нулевого указателя). Но этот механизм влечет несовместимость с существующим кодом, хотя бы по той причине, что в существующем коде есть случаи, где действительно может быть вызов `void`. Фактически этот механизм был реализован для 6.2, а завершающие мазки сделаны в 6.3; полная переработка библиотек для void-безопасности – это большой труд, и она запланирована для 6.4. (Мы работаем по графику, выпуская в год две версии, весной и осенью.)

Конвертирование существующего кода оказалось тонкой проблемой. Мы не можем себе позволить, чтобы существующий код оказался устаревшим, – ни в коем случае. Мы можем только сказать пользователям: если вы хотите воспользоваться этим новым механизмом, который теперь стал доступен, то должны сделать то-то и то-то; мы сами это проделали и знаем, что дело того стоит, но знаем также, каких трудов это требует. Мы поделимся с вами всем нашим опытом и предоставим все инструменты, которые потребуются.

Чему можно научиться на вашем опыте?

Бертран: Не обращать внимания на моду и выбирать то решение, которое правильно с точки зрения разума.

Послесловие

Мое удовольствие от работы над этим проектом можно описать одним словом – восторг. В каждой беседе можно почерпнуть глубокие знания, исторические сведения и практическое понимание предмета. Заразительным также оказался энтузиазм интервьюируемых по поводу проектирования языка, его реализации и дальнейшего развития.

Например, Андерс Хейлсберг и Джеймс Гослинг снова заставили меня восторгаться C# и Java. Чак Мур и Эдин Фалкофф убедили меня заняться изучением Форта и APL – языков, придуманных еще до моего рождения. Ал Ахо соблазнил меня описать свой класс компиляторов. У каждого из тех, кто дал нам интервью, я почерпнул столько идей, что времени не хватит их осмыслить!

Я искренне благодарен этим людям не только за то, что они уделили нам с Федерико столько времени, но и за то, что они осветили нам всем путь к богатейшему полю новых открытий. Вот главные уроки, которые я извлек из этого предприятия:

- Простоту конструкции и реализации нельзя переоценить. Сложность всегда можно добавить потом. Мастер от нее избавляется.
- Отдавайтесь своей любознательности со страстью. Многие из величайших открытий и изобретений появились благодаря тому, что кто-то оказался в нужное время в нужном месте с правильным ответом наготове.
- Изучайте настоящее и прошлое своего любимого дела. Каждый из участников наших интервью работал с другими умными и трудолюбивыми людьми. Очень важно передавать знания друг другу.

Модные языки непрерывно сменяют друг друга, но те проблемы, с которыми сталкивался каждый из их пионеров, по-прежнему преследуют нас, и их решения по-прежнему актуальны. Как сопровождать ПО? Как найти лучшее решение задачи? Как удивить и порадовать пользователей? Как удовлетворить неугомонное стремление все переделать и в то же время сохранить работоспособность действующих решений?

Теперь я лучше знаю, как ответить на эти вопросы. Надеюсь, эта книга помогла вам в обретении собственной мудрости.

Шейн Уорден

Об авторах

Федерико Бьянкуцци – независимый интервьюирующий журналист. Его интервью появляются в таких сетевых изданиях, как ONLamp.com, LinuxDevCenter.com, SecurityFocus.com, NewsForge.com, Linux.com, TheRegister.co.uk и ArsTechnica.com, а также в польском журнале «BSD Magazine» и итальянском журнале «Linux&C».

Шейн Уорден имеет десятилетний опыт разработки бесплатного программного обеспечения, в том числе ядра Perl 5, проекта Perl 6 и виртуальной машины Parrot. В свободное время ведет раздел художественной литературы независимого издательства Onyx Neon Press. Является соавтором «The Art of Agile Development» (O'Reilly).

Участники интервью

Альфред В. Ахо (Alfred V. Aho) – именной профессор факультета компьютерных наук Колумбийского университета. Возглавлял этот факультет с 1996 по 1997 г. и весной 2003 г. Получил степень бакалавра по прикладной физике в университете Торонто и степень доктора по электротехнике и информатике в Принстонском университете. Получил в 2003 г. награду «Great Teacher Award» (за заслуги в преподавании) от Общества выпускников Колумбийского университета. Награжден медалью Джона фон Неймана IEEE и является членом Национальной инженерной академии США и Американской академии наук и искусств. Почетный доктор университетов Хельсинки и Ватерлоо, член Американской ассоциации содействия развитию науки, ACM, Bell Labs и IEEE. Широко известен своими многочисленными статьями и книгами по алгоритмам и структурам данных, языкам программирования, компиляторам и основам компьютерных наук. Соавторами его книг были Джон Хопкрофт, Брайан Керниган, Моника Лэм, Рави Сети, Джефф Ульманн и Питер Вайнбергер.

Профессор Ахо – это буква «A» в AWK, распространенном языке для поиска по шаблонам; «W» представляет Питера Вайнбергера, а «K» – Брайана Кернигана. Алгоритм поиска строк Ахо–Корсак применяется во многих программах для библиографического поиска и анализа генома. Он также автор первых версий программ для поиска строк по шаблону egrep и fgrep, впервые появившихся в UNIX.

В настоящее время в круг научных интересов профессора Ахо входят языки программирования, компиляторы, алгоритмы, программная инженерия и квантовые компьютеры. Профессор Ахо был председателем Специальной группы ACM по алгоритмам и теории вычислимости, председателем Консультативного совета Управления по науке и технике Национального научного фонда. Также является соредактором отдела авторских статей журнала «Communications of the ACM».

До того как занять нынешнее место в Колумбийском университете, профессор Ахо был вице-президентом Исследовательского центра компьютерных наук в Bell Labs, где были изобретены UNIX, Си и C++. Также был техническим специалистом, начальником отдела и директором этого центра. Профессор Ахо был также директором Исследователь-

ской лаборатории информационных наук и технологий в Bellcore (ныне Telcordia).

Гради Буч (Grady Booch) всемирно известен своими новаторскими работами в области программной архитектуры, программной инженерии и сред для совместной разработки. Посвятил жизнь развитию искусства и науки разработки ПО. Был главным научным специалистом Rational Software Corporation с момента ее основания в 1981 г. и до ее приобретения IBM в 2003 г. Сейчас работает в Исследовательском центре IBM Томаса Дж. Уотсона в качестве главного научного специалиста по программной инженерии, продолжая работать над Учебником программной архитектуры, а также руководя несколькими проектами в программной инженерии, не ограниченными требованиями немедленного выпуска продукта. Поддерживает связи с клиентами, решающими практические задачи, устанавливает тесные связи с учебными и исследовательскими организациями в разных концах света. Он один из авторов унифицированного языка моделирования (UML) и участвовал в начальной разработке ряда продуктов Rational. Участвовал в качестве архитектора и архитектурного наставника в создании многочисленных сложных систем, активно использующих ПО, во всех мыслимых предметных областях и в разных частях света.

Гради – автор шести бестселлеров, таких как «UML Users Guide and the seminal» (Язык UML. Руководство пользователя) и фундаментальный труд «Object-Oriented Analysis and Design with Applications» (Объектно-ориентированный анализ и проектирование с примерами приложений), опубликованных издательством Addison-Wesley. Ведет регулярную колонку по архитектуре в IEEE Software. Опубликовал несколько сотен статей по программной инженерии, в том числе ряд статей в начале 1980-х, породивших термин и практику объектно-ориентированного проектирования (OOD), а также статьи в начале 2000-х, породившие термин и практику среды для совместной разработки (CDE).

Гради – член Ассоциации по вычислительной технике (ACM), член Американской ассоциации содействия развитию науки (AAAS) и «Компьютерные профессионалы за социальную ответственность» (CPSR), а также старший член Института инженеров по электротехнике и электронике (IEEE). Также является научным сотрудником IBM, ACM, World Technology Network, визионером Software Development Forum и лауреатом награды «Dr. Dobb's Excellence in Programming» и трех наград «Jolt Award». Является членом правления-основателем Agile Alliance, Hillside Group и Worldwide Institute of Software Architects, а сейчас также работает в консультативном комитете Международной ассоциации программной архитектуры. Кроме того, Гради входит в совет

Теологической школы Илиффа и Музея компьютерной истории. Также является членом редакционной коллегии *IEEE Software*. Гради помогал организовать в Музее компьютерной истории работу по сохранению классических программ и сделал там несколько устных докладов для таких светил, как Джон Бэкус, Фред Брукс, Линус Торвальдс.

Гради получил степень бакалавра в 1977 г., окончив Академию военно-воздушных сил США, а степень магистра по электротехнике получил в 1979 г. в Калифорнийском университете в Санта-Барбаре.

Дон Чемберлен (Don Chamberlin) совместно с Рэем Бойсом разработал SQL – самый известный в мире язык запросов к базе данных. Также был одним из руководителей System R – исследовательского проекта, осуществившего первую реализацию SQL, и разрабатывал основы технологий, на которых построено семейство продуктов баз данных IBM. Также является соавтором предложения «Quilt», ставшего основой языка XQuery. Был представителем IBM в рабочей группе W3C XML Query во время разработки XQuery и редактировал спецификацию языка XQuery. В данное время Дон – адъюнкт-профессор Калифорнийского университета в Санта-Крузе. Также является заслуженным научным сотрудником IBM в исследовательском центре IBM Almaden Research, где проработал много лет. В последние 11 лет также участвовал в качестве судьи и составителя задач в международных соревнованиях ACM по программированию. У Дона степень бакалавра по инженерному делу от Harvey Mudd College и степень доктора по электротехнике от Стэнфордского университета. Он член ACM и Национальной инженерной академии. Также получил награду ACM Software Systems за вклад в проектирование и создание реляционных баз данных.

Д-р Брэд Кокс (Brad Cox) в данное время работает главным архитектором Accenture, где специализируется на безопасности SOA, взаимодействии, стандартах и разработке компонентных систем для клиентов в государственных органах и промышленности. Работал в университете Джорджа Мейсона по программе социального и организационного обучения (PSOL), которая объединяет разные дисциплины и сосредоточена на преодолении трудностей, связанных с изменениями, развитием и обучением при вхождении фирм в глобальную экономику, интенсивно использующую информационные технологии. Занимался применением Интернета, телевидения и средств поддержки групповой работы для облегчения эмпирического и совместного обучения. В число курсов входили «Преодоление электронных рубежей», «Интернет-грамотность» и «Передовые объектные технологии».

Был соавтором книги «Object-Oriented Programming: An Evolutionary Approach» (Объектно-ориентированное программирование: эволюционный подход), изд. Addison-Wesley, которую часто связывают с возникновением нынешнего энтузиазма по поводу объектных и компонентных технологий. Его вторая книга «Superdistribution: Objects As Property on the Electronic Frontier» (Суперраспространение: объекты как свойство на электронном рубеже), изд. Addison-Wesley, предлагает техническое решение для покупки, продажи и владения собственностью, состоящей из битов, а не атомов, как это было испокон веков.

Был сооснователем Stepstone Corporation, где разработал язык программирования Objective-C и библиотеки Software-IC. В исследовательском центре Schlumberger-Doll применил искусственный интеллект, ООП, UNIX и рабочие станции для исследования нефтяных месторождений приборами на кабеле.

В центре программных технологий ITT применил UNIX и объектно-ориентированные технологии для разработки System 1240 – крупной системы коммутации с высокой степенью распределенности.

Получил степень доктора в Чикагском университете за теоретическую и экспериментальную работу в нейрофизиологии в области, которая стала известна как нейронные сети. Свои аспирантские экспериментальные исследования проводил в Национальном институте здравоохранения и в Морских биологических лабораториях Woods Hole.

Эдин Д. Фалкофф (Adin D. Falkoff) – бакалавр химической технологии, Городской колледж Нью-Йорка, 1941; магистр математики, Йель (1963) – до службы в ВМФ США во время Второй мировой войны занимался разработкой материалов и методов для массового производства точных оптических инструментов. Впоследствии занимался разработкой антенн для военных самолетов, прежде чем поступить в 1955 г. на работу в IBM, где руководил научными публикациями в период становления отдела научных исследований IBM. Начал работу в разных областях компьютерных наук в конце 1950-х, и после учебы в Йельском университете по программе стипендий IBM сконцентрировал внимание на компьютерных исследованиях, в том числе APL. В течение нескольких лет входил в число приглашенных преподавателей в Институте исследования систем IBM и был приглашенным лектором по компьютерным наукам в Йельском университете. С 1970 по 1974 год Эдин Фалкофф был организатором и руководителем Научно-исследовательского центра IBM в Филадельфии, с 1977 по 1987 год руководил группой разработки APL в исследовательском центре Томаса Уотсона. Он получил награды IBM за выдающийся вклад за разработку APL и разработку APL\360, за вклад в APL стал первым получателем премии Айверсона,

присуждаемой ACM. Является автором или соавтором таких публикаций, как «Алгоритмы параллельного поиска в памяти», «Формальное описание System 360», «Устройство APL», «Замечание о поиске по шаблону: как найти пустой вектор», «Красочная функция форматирования», «Нотация с использованием точки с запятой и скобок: скрытый ресурс APL», «Семейство систем APL от IBM» и многих других. Эдин Фалкофф зарегистрировал патенты на материалы и методы для производства точных оптических инструментов и проектирования компьютерных систем.

Луис Энрике де Фигейреду (Luiz Henrique de Figueiredo) имеет степень доктора математики от IMPA, Национального института чистой и прикладной математики в Рио-де-Жанейро, где является младшим научным сотрудником и работает в лаборатории визуализации и графики. Также работает консультантом по геометрическому моделированию и программным средствам в Tecgraf, группе компьютерных графических технологий университета PUC-Rio, где участвовал в создании Lua.

Помимо работы над Lua в круг его интересов входят вычислительная геометрия, геометрическое моделирование и методы интервалов в компьютерной графике, особенно аффинная арифметика.

Был постдокторантом в Университете Ватерлоо в Канаде и Национальной лаборатории научных вычислений в Бразилии. Член редколлегии «Journal of Universal Computer Science».

Джеймс Гослинг (James Gosling) получил степень бакалавра в области компьютерных наук в 1977 г. в канадском университете Калгари. Получил докторскую степень по компьютерным наукам в университете Карнеги – Меллон в 1983 г. Его диссертация называлась «Алгебраические операции над ограничениями». В настоящее время – вице-президент и научный сотрудник Sun Microsystems. Разработал системы получения данных со спутников, многопроцессорную версию UNIX, несколько компиляторов, почтовые системы и оконные менеджеры. Также создал текстовый редактор в стиле WYSIWYG, графический редактор на базе ограничений и текстовый редактор Emacs для UNIX. В Sun начинал как ведущий инженер оконной системы NeWS. Создал первоначальный проект языка программирования Java и реализовал для него первый компилятор и виртуальную машину. Участвовал в создании спецификации реального времени для Java, занимался исследованиями в Sun Labs, где его основным интересом были средства разработки ПО. Был главным специалистом по технологиям в подразделении Sun

Developer Products Group, а сейчас главный специалист по технологиям в подразделении Sun Client Software Group.

Чарльз (Чак) Гешке (Charles (Chuck) Geschke) стал в 1982 г. сооснователем Adobe Systems Incorporated. Будучи одним из лидеров программной индустрии в течение более чем 35 лет, ушел в отставку с поста президента Adobe в 2000 г. и продолжает возглавлять совет директоров вместе с другим сооснователем Adobe Джоном Уорноком.

Принимает активное участие в советах ряда учебных заведений, некоммерческих организаций, технологических компаний и художественных организаций. В 1995 г. был избран членом Национальной академии инженерных наук США. В 2008 г. избран членом Американской академии наук и искусств. Недавно завершился его срок пребывания на посту председателя совета попечителей университета Сан-Франциско. Член правления Симфонического оркестра Сан-Франциско и общественной организации Commonwealth Club of California. Также участвует в работе консультативного совета по компьютерным наукам Университета Карнеги – Меллон, правления Egan Maritime Foundation, правления National Leadership Roundtable On Church Management, совета директоров Tableau Software и правления Nantucket Boys and Girls Club.

Прежде чем основать Adobe Systems, создал в исследовательском центре Xerox в Пало-Альто (PARC) лабораторию графических исследований (1980), где руководил работой в области компьютерных наук, графики, обработки изображений и оптики. С 1972 по 1980 г. был руководителем научных исследований в лаборатории компьютерных наук в Xerox PARC. До того как поступить в 1968 г. в очную аспирантуру, был преподавателем математического факультета университета Джона Кэрролла в Кливленде (Огайо).

Руководители промышленности и бизнеса, включая Ассоциацию по вычислительной технике (ACM), Институт инженеров по электротехнике и электронике (IEEE), Университет Карнеги – Меллон, Национальную ассоциацию компьютерной графики и Рочестерский технологический институт, отметили технические и административные достижения Гешке. Он получил премию регионального предпринимателя года (1991) и премию общенационального предпринимателя года (2003). В 2002 г. принят в члены Музея компьютерной истории, а в 2005 г. удостоен награды за выдающуюся общественную деятельность Национальной ассоциации христиан и иудеев Кремниевой долины. Получил медаль за заслуги от Американской ассоциации электроники (AeA) в 2006 г. Он и Джон Уорнок были первыми лидерами программной отрасли, полу-

чившими эту награду. В 2007 г. получил премию Джона У. Гарднера за лидерство. В 2000 г. был отмечен седьмым в публикуемом журналом «Graphic Exchange» списке тех, кто оказал наибольшее влияние на графику за прошедшее тысячелетие.

Имеет докторскую степень по компьютерным наукам от Университета Карнеги–Меллон, а также степень магистра математики и бакалавра искусств в латыни от Университета Ксавье.

Андерс Хейлсберг (Anders Hejlsberg) – технический сотрудник отдела серверов и инструментов Microsoft. Признанный и пользующийся влиянием создатель инструментов разработки и языков программирования. Главный проектировщик языка программирования C# и ведущий участник разработки Microsoft .NET. Вслед за своим появлением в 2000 г. язык программирования C# получил широкое распространение и сейчас стандартизирован ECMA и ISO.

До прихода в Microsoft в 1996 г. Андерс был одним из ведущих работников в Borland International Inc. В качестве главного инженера стал автором Turbo Pascal – революционной интегрированной среды разработки – и главным архитектором ее преемницы Delphi.

Соавтор книги «The C# Programming Language» (Язык программирования C#), изд. Addison-Wesley, получил много патентов в программировании. В 2001 г. получил престижную награду «Dr. Dobbs Excellence in Programming», а в 2007 г. он и его команда получили награду Microsoft «За выдающиеся технические достижения».

Изучал инженерное дело в Техническом университете Дании.

Пол Худак (Paul Hudak) – профессор факультета компьютерных наук в Йельском университете. Преподает в Йеле с 1982 г., в 1999–2005 гг. возглавлял факультет. Степень бакалавра в области электротехники получил в Университете Вандербильта (1973), степени магистра в области электротехники и в области компьютерных наук – в MIT (1974), а степень доктора компьютерных наук защитил в Университете штата Юта (1982).

Научные интересы профессора Худака сконцентрированы на проектировании, теории и реализации языков программирования. Он был в числе организаторов и руководителей комиссии по Haskell, которая в 1988 г. выпустила первую версию Haskell, чисто функционального нестрогого языка программирования. Был соавтором первого доклада по Haskell и написал для этого языка популярные руководство и учебник.

Его ранние работы были также посвящены параллельному функциональному программированию, абстрактной интерпретации и декларативным подходам к состоянию.

В недавнее время профессор Худак занимался разработкой проблемно-ориентированных языков для целого ряда прикладных областей, включая мобильную и гуманоидную робототехнику, графику и анимацию, синтез звука и музыки, графические интерфейсы пользователя и системы реального времени. Также разработал технологию встраивания таких языков в Haskell, в том числе использование таких абстрактных моделей вычислений, как монады и стрелы. В самое последнее время занимался применением Haskell в синтезе компьютерной музыки и звука в области образования и исследований.

Профессор Худак опубликовал больше 100 статей и одну книгу. Он главный редактор «Journal of Functional Programming» и член-основатель рабочей группы 2.8 IFIP по функциональному программированию. Удостоен, среди прочего, звания члена ACM, награды IBM за педагогические достижения и «Президентской награды молодому исследователю» Национального научного фонда.

Джон Хьюз (John Hughes), родившийся в 1958 г. в Северном Уэльсе, провел решающий год (1974–1975) между школой и университетом в качестве программиста в научно-исследовательской группе покойного Кристофера Стречи в Оксфордском университете. Во время собеседования помог Стречи установить модем, но главное, там Джон познакомился с функциональным программированием, страсть к которому не угасла у него с годами. Во время изучения математики в Кембридже он стал соавтором, вероятно, первого компилятора для GEDANKEN, мысленного эксперимента Джона Рейнольдса в проектировании языков программирования. Вернулся в Оксфорд в 1980 г. и в 1983 г. закончил там докторскую диссертацию, посвященную технологиям реализации функциональных языков. Там он познакомился со своей женой Мэри Ширан, работавшей с ним в одной группе исследователем.

1984–1985 гг. Джон провел в постдокторантуре Технического университета Чалмерса в Гётеборге (Швеция), где велась новаторская работа по компиляции ленивых языков, таких как Haskell. Он полюбил как обстановку научных исследований, так и природу западной Швеции. По прошествии этого года Джон недолго преподавал в Оксфорде, а в 1986 г. получил кафедру в Университете Глазго (Шотландия). Университет Глазго в то время сильно расширялся, и Джон смог организовать Группу функционального программирования Глазго, ставшую впоследствии одной из лучших в мире и включавшую Фила Уодлера

и Саймона Пейтон-Джонса. Ежегодные семинары этой группы стали широко известны и в итоге превратились в симпозиум «Тенденции в функциональном программировании», проводимый до сих пор.

Но в 1992 г. Джону была предложена кафедра в Университете Чалмерса, и он воспользовался возможностью вернуться в Швецию. Там он продолжает заниматься функциональным программированием, а с 1999 г. занимается тестированием с помощью автоматизированного средства QuickCheck. В 2006 г. он основал компанию Quviq, занимающуюся маркетингом и разработкой QuickCheck, и теперь половину своего времени отдает этой компании.

Джон получил шведское гражданство и постарался освоить шведский язык и лыжи – последнее сначала не предвещало ничего хорошего! У него двое сыновей, один из которых слеп и страдает аутизмом.

Роберто Иерусалимский (Roberto Ierusalimschy) – адъюнкт-профессор компьютерных наук в PUC-Rio (Понтификальный католический университет в Рио-де-Жанейро), где занимается проектированием и реализацией языков программирования. Ведущий архитектор языка программирования Lua и автор книги «Programming in Lua» (Программирование на Lua), выдержавшей два издания и переведенной на китайский, корейский и немецкий языки.

Имеет степени магистра (1986) и доктора (1990) в области компьютерных наук, полученные в PUC-Rio. Был приглашенным исследователем в Университете Ватерлоо (Канада, 1991), ICSI (Калифорния, 1994), GMD (Германия, 1997) и UIUC (Израиль, США, 2001–2002). Как преподаватель в PUC-Rio был наставником нескольких студентов, ставших впоследствии влиятельными членами сообщества Lua. В последнее время разрабатывает LPEG, пакет поиска по шаблонам для Lua.

Д-р Айвар Якобсон (Ivar Jacobson) родился в г. Истаде (Швеция) 2 сентября 1939 г. (Его полное имя Айвар Ялмар Якобсон, но второе имя он никогда не использует.) Получил степень магистра в области электротехники в Технологическом институте Чалмерса в Гётеборге (1962). Получил докторскую степень в Королевском технологическом институте в Стокгольме (1985), защитив диссертацию на тему «Языковые конструкции для больших систем реального времени». Был приглашенным исследователем в группе функционального программирования и архитектур потоков данных в MIT в 1983–1984 гг. 3 мая 2003 г. получил медаль Густава Далена от Ассоциации выпускников Чалмерса.

Основал в Швеции компанию Objectory AB, которая в 1995 г. вошла в Rational. Работал в Rational во время ее выдающегося роста вплоть до 2003 г., когда компанию приобрела IBM. После этого перестал быть служащим компании, но больше года, до мая 2004 г., был ее главным техническим консультантом.

Параллельно с работой в Rational занимался разными интересными делами. Одним из них была работа в Jaczone AB, компании, которую он основал в апреле 2000 г. вместе с дочерью Агнетой Якобсон. Jaczone продвигает старую идею: сделать программный процесс активным, а не пассивным. Активный процесс выполняется и помогает разработчику работать над своим проектом.

Айвар также считает, что сообщество разработчиков ПО крайне нуждается в совершенствовании методов разработки. В 2004 г. он основал Ivar Jacobson International с целью способствовать применению командами разработчиков во всем мире правильных практик разработки. Ivar Jacobson International сейчас оперирует через отдельные компании в шести странах: Великобритании, США, Швеции, Китае, Австралии и Сингапуре. В 2007 г. его новая компания приобрела Jaczone, так что обе компании теперь объединились.

Саймон Пейтон-Джонс (Simon Peyton-Jones) – магистр искусств, член Британского компьютерного общества, дипломированный инженер. Окончил Тринити-Колледж в Кембридже (1980). Проработав два года в промышленности, несколько лет читал лекции в Университетском колледже Лондона, 9 лет был профессором университета Глазго, прежде чем в 1998 г. перейти на работу в Microsoft Research (Кембридж).

Главные научные интересы – функциональные языки программирования, их реализация и их применение. Руководил рядом исследовательских проектов в области проектирования и реализации систем функционального программирования промышленного качества для однопроцессорных и параллельных машин. Внес основной вклад в создание ставшего стандартом функционального языка Haskell и является ведущим проектировщиком широко известного Glasgow Haskell Compiler (GHC). Написал два учебника по реализации функциональных языков.

Помимо этого интересуется проектированием языков, системами с богатым набором типов, архитектурами на основе программных компонентов, генерацией кода, исполнительными системами, виртуальными машинами и сборкой мусора. Особый интерес вызывает у него непосредственное применение принципиальной теории к практическому проектированию и реализации языков, поэтому он с такой любовью относится к функциональному программированию.

Брайан Керниган (Brian Kernighan) получил степени бакалавра в университете Торонто (1964) и доктора в области электротехники в Принстоне (1969). До 2000 г. работал в Исследовательском центре компьютерных наук в Bell Labs, а сейчас работает на факультете компьютерных наук в Принстоне.

Автор восьми книг и нескольких технических статей, получил четыре патента. В 2002 г. был избран членом Национальной инженерной академии. Область научных интересов: языки программирования, инструменты и интерфейсы, облегчающие работу с компьютерами, в том числе для неспециалистов. Также интересуется техническим образованием нетехнических аудиторий.

Томас Е. Курц (Thomas E. Kurtz) родился близ Чикаго (Иллинойс), 22 февраля 1928 г. Окончил колледж Нокс в Иллинойсе (1950). Затем учился в Принстонском университете, где получил докторскую степень по математике (1956). Проработал в Дартмутском колледже с 1956 г. до выхода в отставку в 1993 г., преподавая там статистику, численный анализ и, в конце, компьютерные науки. В 1963–1964 гг. вместе с Джоном Кемени (ставшим позже президентом Дартмутского колледжа) разработал язык программирования Бейсик, который благодаря революциям, произведенным системами разделения времени и персональными компьютерами, десятки лет оставался самым распространенным языком программирования в мире. Был директором вычислительного центра в Дартмуте с 1966 по 1975 г.

Участвовал в многочисленных советах и комиссиях, вместе с Кемени написал несколько книг по программированию. После ухода из Дартмута участвовал в работе True BASIC, Incorporated, занимавшейся разработкой и маркетингом компьютерного языка Бейсик и других образовательных программных продуктов для персональных компьютеров.

Том Лав (Tom Love) получил докторскую степень по когнитивистике в Вашингтонском университете, где изучал когнитивные особенности успешных компьютерных программистов. После окончания учебы поступил на работу в General Electric, где занимался проектированием интерфейса для фирменной машины текстового поиска – Google in a box. Через несколько месяцев Управление морских исследований связалось с ним и поинтересовалось, не хочет ли он продолжить свои докторские исследования. Это привело к появлению в GE группы по психологии программирования.

Из GE ушел в ITT, где ему предложили создать научно-исследовательскую группу по ПО. Именно в этой группе Брэд Кокс придумал и разработал первое объектно-ориентированное расширение языка Си. Эта группа в ITT также изучала ПО для коллективной работы, распределенные вычисления и интерактивные среды разработки – в 1982 году! На базе этого опыта Том стал впоследствии первым коммерческим пользователем Smalltalk (1982).

В 1983 г. Том и Брэд Кокс основали Stepstone – первую компанию по производству объектно-ориентированных продуктов. В Stepstone они развивали объектные технологии, выдвинули идею Software-IC и выпустили на рынок первый самостоятельный набор многократно используемых классов, IC-pak 201. Другим их достижением было то, что они смогли убедить Стива Джобса использовать Objective-C в качестве языка системного программирования для компьютера NeXT (что позднее стало основой операционной системы Apple OS X). Том также выдвинул идею и организовал группу добровольцев, создавших конференцию ACM OOPSLA.

Проработав пять лет самостоятельным консультантом, Том пришел в IBM Consulting и основал там Object Technology Practice – организацию для разработки приложений, осуществлявшую большие программные проекты для крупных клиентов IBM. Ее успех привел его в «Морган Стэнли», где он установил переработанную корпоративную систему управления рисками за два дня до скандала с банком «Бэрингс».

В 1997 г. Том объединился с Джоном Вутеном, основав ShouldersCorp. В ShouldersCrop руководил более чем десятком успешных 100-дневных проектов, в том числе крупнейшим из известных Agile-проектов, завершенным в 2001 г. Его многообразный опыт применения объектных технологий нашел отражение в выпущенной издательством Cambridge University Press книге «Уроки объектов» (1993).

Бертран Мейер (Bertrand Meyer) – профессор инжиниринга программного обеспечения в Швейцарском федеральном технологическом институте ETH (Цюрих) и главный архитектор в Eiffel Software (Санта-Барбара, Калифорния). Выступает во многих качествах, в том числе как руководитель программных проектов (контролировал разработку инструментов и библиотек общим объемом в несколько миллионов строк кода), программный архитектор, преподаватель, исследователь, автор книг и консультант.

Опубликовал 10 книг, в том числе такие бестселлеры, как «Object-Oriented Software Construction»[1] (изд. Prentice Hall, Jolt Award, 1998),

[1] Мейер Б. «Объектно-ориентированное конструирование программных систем». – Пер. с англ. – Русская редакция, 2005.

и «Eiffel: The Language, Object Success, and Introduction to the Theory of Programming Languages» (Eiffel: язык, успех объектов и введение в теорию языков программирования), изд. Prentice-Hall PTR. Его последняя книга, представляющая собой введение в программирование с полным использованием объектной технологии и контрактов («Touch of Class: An Introduction to Programming Well with Objects and Contracts», Springer-Verlag, 2009), – результат шести лет чтения начального курса программирования в ETH.

Как исследователь опубликовал свыше 200 статей по разным вопросам программирования. Основной вклад внес в программную архитектуру и проектирование (контрактное программирование), языки программирования (Eiffel, ныне стандарт ISO), тестирование и формальные методы. В настоящее время главные области научных исследований его и его группы в ETH – это безопасное и простое программирование для параллельных и многоядерных структур (SCOOP), автоматизированное тестирование (AutoTest), доказательство программ, педагогические инструменты (Trucstudio), педагогика компьютерных наук, среды разработки (EiffelStudio, Origo), разработка на основе компонентов и многократного использования, программные процессы и устойчивые объекты.

Награжден премией ACM Software System (2006) и первой премией Даля–Нюгорда за объектные технологии (2005), является членом ACM и Французской технологической академии.

Робин Милнер[1] (Robin Milner) окончил Кембриджский университет (1958). После непродолжительного пребывания в разных должностях в 1973 г. начал работать в Эдинбургском университете, где в 1986 г. стал сооснователем Laboratory for Foundation of Computer Science. Избран членом Королевского общества (1988), награжден ACM премией Тьюринга (1991). Вернулся в Кембридж в 1995 г., где 4 года возглавлял лабораторию вычислительной техники, уйдя в отставку в 2001 г. Его научные достижения (часто совместные) включают: систему LCF, на которой основаны многие последующие системы автоматизированного доказательства теорем; Standard ML, строго определенный язык программирования промышленного масштаба; исчисление общающихся систем (CCS); пи-исчисление.

В последнее время работал над Bigraphs, топографической моделью мобильных интерактивных систем. Эта модель сочетает в себе мощь пи-исчисления, делающего акцент на возможности мобильных агентов модифицировать свои связи, с мощью исчисления окружений (mobile

[1] Скончался 20 марта 2010 г. в Кембридже.

ambients) Карделли и Гордона, делающего акцент на их перемещении во вложенном пространстве. В комбинации этих двух функций они рассматриваются как независимые: «то, где ты находишься, не влияет на то, с кем ты можешь общаться». Отсюда обобщенная модель, которая не только включает многие исчисления процессов, но и может дать строгую платформу для проектирования «вездесущих» вычислительных систем, которые будут доминировать в XXI веке.

Чарльз Х. Мур (Charles H. Moore) родился в 1938 г.; вырос в штате Мичиган; получил степень бакалавра по физике в MIT; женат на Уинфред Беллис, есть сын Эрик. Сейчас живет в Инклайн-Виллидж, на чудесном озере Тахо; водит WRX; ходит горными туристическими маршрутами Тахо-Рим и Пасифик-Крест; много читает. Получает удовольствие, находя простые решения, для чего может изменить задачу.

В 1960-е годы работал как независимый программист, пока не изобрел в 1968 г. Форт (Forth) – простой, эффективный и гибкий компьютерный язык, которым очень гордится. Занимался программированием телескопов для Национальной радиоастрономической лаборатории (NRAO). В 1971 г. стал сооснователем фирмы Forth, Inc., занимающейся программированием систем реального времени.

В 1983 г., недовольный имевшимся аппаратным обеспечением, стал сооснователем Novix, Inc. Разработал в ней микропроцессор NC4000. Позднее он превратился в Harris RTX2000, который подошел для использования в космосе и теперь вращается вокруг Сатурна на «Кассини».

В фирме Computer Cowboys с помощью специального программного обеспечения разработал ShBoom, Mup20, F21 and i21 – микропроцессоры с архитектурой Forth. Очень гордится этими маленькими, быстрыми и экономичными процессорами.

В этом столетии стал сооснователем IntellaSys и придумал colorForth для программирования конструкторских инструментов для многоядерного чипа. В 2008 г. IntellaSys производила и продавала 40-ядерную версию процессора. В настоящее время Чарльз Мур переносит свои инструменты на этот удивительный чип.

Джеймс Рамбо (James Rumbaugh) получил степень бакалавра физики в MIT, степень магистра астрономии в Калтехе и степень доктора компьютерных наук в MIT. Работу над докторской диссертацией в MIT проводил в группе вычислительных структур профессора Джека Денниса, которая занималась передовыми исследованиями в области фундаментальных вычислительных моделей. В его диссертации были пред-

ставлены язык и архитектура компьютера, управляемого потоком данных, – максимально распараллеленная архитектура.

25 лет проработал в Центре исследований и разработок General Electric в Скенектади (Нью-Йорк), занимаясь различными исследовательскими проектами, среди которых: одна из первых многопроцессорных операционных систем, алгоритмы реконструкции томографических изображений, система проектирования СБИС, одна из первых систем графических интерфейсов пользователя и объектно-ориентированный язык. Вместе со своими коллегами разработал технологию объектного моделирования (OMT) и написал книгу «Object-Oriented Modeling and Design» (Объектно-ориентированное моделирование и проектирование), изд. Prentice Hall, популяризировавшую OMT. Шесть лет вел популярную ежемесячную колонку в журнале ООП (JOOP).

В 1994 г. пришел в Rational Software Corporation в Купертино (Калифорния), где он и Гради Буч объединили свои методы моделирования, создав унифицированный язык моделирования (UML), в который потом внесли свой вклад Айвар Якобсон и сотрудники группы объектного моделирования (OMG). Стандартизация UML, проведенная OMG, способствовала широкому распространению UML как ведущего языка моделирования ПО. Книги, написанные Рамбо, Бучем и Якобсоном, представили UML широкой публике. Участвовал в руководстве дальнейшим развитием UML и пропагандировал применение правильных инженерных принципов в разработке ПО. После того как Rational была приобретена IBM, в итоге вышел в отставку (2006).

Джеймс – опытный лыжник, слабый игрок в гольф и турист выходного дня. Посещает оперу, театр, балет и художественные галереи. Любит хорошую кулинарию, путешествия, фотографию, садоводство и иностранные языки. Любит читать о космологии, эволюции и когнитивнике, эпическую поэзию, фэнтези, об истории и общественных делах. Они с женой живут в Саратоге (Калифорния). У них два сына, которые учатся в колледже.

Бьерн Страуструп (Bjarne Stroustrup) разработал и реализовал C++. За последнее десятилетие C++ стал наиболее широко используемым языком с поддержкой ООП благодаря тому, что сделал технологию абстракции доступной и управляемой для большинства проектов. Используя в качестве инструмента C++, Страуструп стал пионером использования технологий объектно-ориентированного и обобщенного программирования в тех прикладных областях, где ценится эффективность: общее системное программирование, коммутация, моделирование, графика, интерфейсы пользователя, встроенные системы и научные расчеты.

Влияние C++ и популяризированные им идеи отчетливо просматриваются далеко за пределами сообщества C++. Многие языки, включая Си, C#, Java и Fortran99, предоставляют возможности, впервые предложенные для общего использования C++, как и системы типа COM или CORBA.

Его книга «The C++ Programming Language» (Язык программирования C++), изд. Addison-Wesley, 4 издания: 1985, 1991, 1997, 2000 – «специальное» издание, – самая читаемая книга такого рода, переведенная на 19 языков. Более поздняя книга «The Design and Evolution of C++» (Дизайн и эволюция языка C++), изд. Addison-Wesley, 1994, заложила основу нового способа описания того, как идеи, идеалы, проблемы и практические ограничения формируют язык программирования. Еще одна книга, «Programming Principles and Practice Using C++»[1] призвана сыграть роль первого знакомства с программированием и C++. Кроме 6 книг Страуструп опубликовал больше 100 научных и популярных статей. Принял активное участие в создании стандарта ANSI/ISO для C++ и продолжает работать над сопровождением и пересмотром этого стандарта.

Родился в Орхусе (Дания), получил степень магистра в области математики и компьютерных наук в Орхусском университете. Докторскую степень получил в Англии, в Кембриджском университете, где исследовал распределенные вычисления. С 1979 по 2002 г. работал в качестве исследователя, а потом менеджера в Bell Labs и AT&T Labs в Нью-Джерси. Был главой Отдела исследования программирования со времени его создания в AT&T до 2002 г., когда он соединился с Отделением научных исследований Техасского университета A&M. Член Национальной инженерной академии США, член ACM и IEEE. Получил множество профессиональных наград.

Гвидо ван Россум (Guido van Rossum) – создатель Python, одного из важнейших языков, используемых как в Сети, так и вне ее. Сообщество Python присвоило ему титул BDFL (Добровольный пожизненный диктатор), который мог бы происходить из комедийного сериала «Монти Пайтон», если бы он там был.

Гвидо вырос в Голландии и долгое время работал в Амстердаме в CWI, где и появился на свет Python. В 1995 г. он переехал в США, где поселился в Северной Вирджинии, женился и обзавелся сыном. В 2003 г. семья переехала в Калифорнию, где Гвидо теперь работает в Google, тратя

[1] Страуструп Б. «Программирование. Принципы и практика использования C++». – Пер. с англ. – Вильямс, 2010.

половину своего времени на открытый проект Python, а остальное время – на проекты в интересах Google.

Филип Уодлер (Philip Wadler) любит применять теорию к практике, а практику к теории. Два примера реализации теории на практике: GJ, основа новой версии Sun Java, происходит от квантификаторов в логике второго порядка. Его работа над XQuery знаменует одну из первых попыток применения математики для формулирования промышленного стандарта. Пример применения практики к теории: Featherweight Java специфицирует ядро Java с помощью правил объемом меньше страницы. Он один из главных разработчиков языка программирования Haskell, внесший в него две основные новации: типы классов и монады.

Является профессором теоретических компьютерных наук в Эдинбургском университете. Стипендиат Королевского общества Wolfson Research, член Королевского общества Эдинбурга и ACM. Прежде учился или работал в Avaya Labs, Bell Labs, Глазго, Чалмерсе, Оксфорде, CMU, Xerox Parc и Стэнфорде, читал лекции в качестве приглашенного профессора в Париже, Сиднее и Копенгагене. Занимает 70-е место в списке CiteSeers наиболее цитируемых авторов по компьютерным наукам, получил награду POPL за наиболее важную статью, работал главным редактором *Journal of Functional Programming*, участвовал в исполнительном комитете специальной группы ACM по языкам программирования. Автор статей «Listlessness is better than laziness» (Безразличие лучше лени), «How to replace failure by a list of successes» (Как заменить провал списком достижений) и «Theorems for free» (Бесплатные теоремы), соавтор книг «XQuery from the Experts» (Эксперты об XQuery), изд. Addison-Wesley, 2004, и «Java Generics and Collections» (Генерики и коллекции в Java), изд. O'Reilly, 2006. Разъезжает с выступлениями по всему свету – от Айдзу до Цюриха.

Ларри Уолл (Larry Wall) учился во многих местах, включая Кронуоллскую музыкальную школу, Сиэттлский молодежный симфонический оркестр, Библейскую школу Малтномы, Летний институт лингвистики, университет в Беркли и университет в Лос-Анджелесе. Несмотря на то что его образование по большей части относится к музыке, химии и лингвистике, последние 35 или около того лет Ларри работает с компьютерами.

Более всего он знаменит тем, что написал _rn_, _patch_ и язык программирования Perl, но предпочитает представлять себя культурным хакером, призвание которого – внести немного радости в унылое существование программистов. В разных значениях выражения «работать

на» Ларри работал на Seattle Pacific, MusiComedy Northwest, System Development Corporation, Burroughs, Unisys, NSA, Telos, ConTel, GTE, JPL, NetLabs, Seagate, Тима О'Рейлли, Perl Foundation и самого себя. В настоящее время работает в NetLogic Microsystems в Маунтин-Вью (Калифорния). Чтобы попасть на работу, ему нужно пройти мимо Музея компьютерной истории и Googleplex, в чем должен быть заключен какой-то смысл. Скорее всего, абсурдный.

Джон Э. Уорнок (John E. Warnock) – сопредседатель совета директоров Adobe Systems, Inc., компании, которую он основал в 1982 г. вместе с Чарльзом Гешке. Был президентом Adobe первые 2 года работы там, а остальные 16 лет – председателем и CEO. Руководил разработкой всемирно признанных технологий для графики, издательского дела, Интернета и электронных документов, которые произвели революцию в издательском деле и передаче визуальной информации. Получил шесть патентов.

Успех предпринимательской деятельности Уорнока отмечен рядом наиболее влиятельных изданий в мире бизнеса и компьютерной индустрии. Он получил многочисленные награды за технические и административные достижения. Вот частичный список его наград: «Предприниматель года» от Ernst & Young, Merrill Lynch и Inc. Magazine; награда «Выдающемуся выпускнику университета штата Юта»; награда Software Systems Ассоциации по вычислительной технике (ACM); премия за выдающиеся технические достижения Национальной ассоциации графики; первая Международная премия за исключительные достижения в искусстве и дизайне Школы дизайна штата Род-Айленд. Д. Уорнок получил также премию Эдвина Х. Лэнда Американского оптического общества, Бодлианскую медаль Оксфордского университета и медаль Лавлейс Британского компьютерного общества. Является почетным членом Национальной инженерной академии и членом Американской академии искусства и науки. Обладатель почетных степеней Университета штата Юта и Американского института кинематографии.

Был членом совета директоров Adobe Systems Inc., Knight-Ridder, Octavo Corporation, Ebrary Inc., Mongonet Inc., Netscape Communications и Salon Media Group. Бывший председатель Технологического музея инноваций в Сан-Хосе. Также член совета попечителей Американского института кинематографии и член правления Sundance Institute.

До того как основал Adobe Systems, Уорнок был ведущим исследователем в научно-исследовательском центре Xerox Пало-Альто (PARC). До прихода в Xerox занимал важные должности в Evans & Sutherland Computer Corporation, Computer Sciences Corporation, IBM и Университете штата Юта.

Имеет степени бакалавра и магистра математики и докторскую степень по электротехнике (все получены в Университете штата Юта).

Питер Вайнбергер (Peter Weinberger) работает в Google New York с середины 2003 г., занимаясь различными проектами, связанными с хранением больших объемов данных. До этого (с момента разделения AT&T и Lucent) работал в Renaissance Technologies, сказочно удачливом хедж-фонде (успехи которого он не ставит себе в заслугу), где вначале был исполнительным директором, ответственным за вычислительную технику, ПО и информационную безопасность. Примерно год назад все это бросил и стал заниматься трейдинговой системой (для бумаг, обеспеченных закладными).

До разделения AT&T и Lucent работал в исследовательском центре Bell Labs в Мюррей-Хилл. Прежде чем окончательно уйти в менеджмент, работал над базами данных, AWK, сетевыми файловыми системами, компиляцией, эффективностью, профилированием и прочими UNIX-штучками. Затем занялся административной деятельностью, где достиг высшего чина «вице-президента по исследованиям в информатике» (только крупная корпорация может придумать такой титул). Под руководством Питера проводилась примерно треть всех исследований включая математику и статистику, компьютерные науки и речь. Свой последний год в AT&T он провел в службе дальней связи, пытаясь предугадать будущее.

До работы в Bell Labs преподавал математику в Университете штата Мичиган в Энн-Арборе, опубликовав ряд статей (последняя из которых была отвергнута в 2002 г.), не представляющих даже академического интереса.

Получил степень бакалавра в Суортмор-колледже в Суортморе (Пенсильвания) и докторскую степень по математике (теория чисел) в Калифорнийском университете в Беркли.

Алфавитный указатель

A

Acrobat, программа, 520
Adobe AIR, платформа, 521
agile, принцип, 329, 406
API-проектировние, 63, 373, 393
APL, язык, 69
  и параллелизм, 80, 84
  история проектирования, 70, 78
  коллекции, 82
  конструкция, 74
  набор символов для, 72, 80
  обобщенные массивы, 80
  обработка файлов, 83
  обучение программированию с помощью, 75
  общие переменные, 81
  пространства имен, 83
  реализация в карманных устройствах, 73
  синтаксис для, 71, 76, 81
  стандартизация, 73
  трудность изучения, 72
  удачные стороны, 86
  уроки проектирования, 85
  числа с произвольной точностью на основе алгебраической нотации, 71, 76
  что следовало сделать иначе, мнение разработчика, 86
  эффективное использование ресурсов, 72
APL\360, язык, 70
APT (Automatic Programmed Tools), 127
ArchiText, торговая марка, 541
ASP.NET, 378
AWK, язык сценариев, 139, 140
  большие программы, 199
  в сравнении с SQL, 184
  знания, необходимые для работы, 144
  крупные приложения, методы создания, 140
  начальные проектные идеи, 183
  подгонка скриптов, 192
  применение, 140, 144, 146, 164
  программирование
    по примерам, 205, 210
    рекомендации, 164
  размер обрабатываемых данных, 141
  что следовало сделать иначе, мнение Питера Вайнбергера, 188
AWT, библиотека, 357

B

BASIC, язык, 113
BCPL, язык, 507
Berkeley UNIX, 518
BPEL (Business Process Execution Language), 465
BPEL, язык, 339
Bridge, программа, 517
BSD, язык для ядра, 24
Buddha, отладчик, 239

C

C, язык
  применение стека, 368
C#, язык, 375
  Java как первоисточник, 368
  долголетие, 388
  как замена C++, 388
  команда разработчиков, управление, 389
  отзывы пользователей, 382, 389
  отладка, 392
  развитие, 388
  стандартизация ECMA, 391
  формальные спецификации, 392

C++, язык
  C# в качестве замены, 388
  абстрагирование данных, 18
  безопасность типов и защита, 23
  ближе к аппаратной части, принцип проектирования, 20
  будущие версии, 31
  версия 2.0, 31
  влияние производительности на проектирование, 21
  в системном программном обеспечении, 23
  для встроенных приложений, 23
  как расширение Си, 16
  методы обобщенного программирования, 18
  многопоточность, проблемы, 369
  обратная совместимость с Си, 176
  ОО в качестве одной из парадигм, 24
  отладка, 21
  перенос кода с Си, причины, 25
  поддержка параллелизма, 29
  поддержка различных парадигм, 17, 24, 26
  поддержка семантики значений, 20
  популярность, 315
  почему ядро не пишут на, 24
  развитие, 384
  сложность, 17, 315
  сравнение с
    Objective-C, 314, 334
    Си, 25
  тестирование, 21
  требования совместимости, 40
  указатели
    в сравнении с Java, 18
    проблемы, 368
  управление ресурсами, 20
  уроки изобретения, 33
C++0x, язык, 29, 31
C&A (сертификация и аккредитация), 342
CCS (исчисление общающихся систем), 266
CDE, среда, 555
cloud computing, принцип, 518
CLU, язык, 529
CMMI, 408
Codeplex, 382, 394
colorForth, 93
COM, модель, 171
Console.Writeline, 392
COQ, 264
CPAN, Perl, 487, 488

D

DISA (Defense Information Systems Agency), 341
Display PostScript, 508
DMCA, 349
DODAF, 354
DOM, 354
do-синтаксис, 249
DSEL, 250
DSL, 414
DSM (Data Structure Manager), 445

E

ECMA, 549
  стандартизация C#, 391
EE5, 357
Eiffel, язык, 523
  добавление функций, принятие решений, 549
  доказательство, 546
  история, 524, 528
  многократное использование, 535
  поточная сериализация, 541
  развитие, 546
  расширяемость, 533
  скрытие информации, 534
  совместимость
    обратная, 550
    прямая, 550
Erlang, язык, 398
EUML (Essential Unified Modeling Language), 414
Excel, в сравнении с реляционными БД, 306

F

Featherweight Java, 256
Flash, 509, 520
Forth, язык, 89
  colorForth, 93
  многократно используемые концепты смысла, 268
for цикл, в Lua, 222

G

GCC, 358
GEDANKEN, 561
GRM (General Resource Management), 20
GHC (Glasgow Haskell Compiler), 231, 239, 256, 259
Good Enough, 489
GOTO оператор, в Бейсике, 114, 122
GP (generic programming) *см.* обобщенное программирование, 24

H

Haskell, язык, 231
  влияние на другие языки, 249
  конкурирующие реализации, 259
  конструкция, влияние на будущие системы, 261
  ленивые вычисления, 235, 248
  монады, 280
  обобщенные типы, 249
  развитие, 259, 261
  разработка в команде, 232, 235
  система классов Haskell, 243
  система типов, 245, 248, 276
  (см. также функциональное программирование), 243
  списочные выражения, 249
  функциональные замыкания, 249
HOL, 264
HOPL-III
  доклад о языке Emerald, 29
HotSpot, 358, 366
HTML, PostScript как альтернатива для, 521
HyperCard, язык, 515

I

Icon, язык, 488
IC-pak, набор классов 201, 565
IDUG, группа пользователей, 304
Illustrator, программа, 516
InDesign, программа, 516
InterPress, проект, 512
Isabelle, язык, 264

J

J2EE, язык, 353
JaM, язык, 498
Java, язык, 355
  AWT и, 357
  JIT, 358
  виртуальная машина, обоснование, 361
  генерики, влияние Haskell, 245
  добавление функций в язык или внешние библиотеки, 373
  как первоисточник для C#, 368
  обратная совместимость с, 357
  отзывы пользователей, 371
  ошибки, предотвращение и локализация, 362, 368
  платформонезависимость, влияние, 373
  проверка индексов массивов, 359
  раскрытие кода, последствия, 372
  сравнение с C++
    сложность, 17
    указатели, 18
  ссылки, 368
  уменьшение сложности, проблемы, 356
  функции высшего порядка, 380
Java Community Process, 355
Javadoc, инструмент, 370
Java EE, 357
JavaScript, 516, 521
JBI (Java Business Integration), 339, 341
JBoss, 344
JIT технология, 213, 358
JOOP, 568
JUnit, 370, 544
JVM, язык
  новые языки на базе, 366
  популярность, отражение в конструкции, 362
  сборка мусора в, 367
  удовлетворенность проектировщика, 361

L

Lab View, 464
LINQ (Language-integrated query), 386
LCF, 265
  доказательство теорем с помощью, 264
  ограничения, 264

lex, язык
  что нужно знать для работы, 144
LINQ, 249, 378, 381
LINT, 369
Linux, ядро
  почему не на C++, 24
Lisp, язык
  многократно используемые концепты смысла, 268
Lua, язык, 211, 212
  for цикл, 222
  асимметричные сопрограммы в, 215
  виртуальная машина
    влияние на отладку, 227
    выбор ANSI C для, 227
    на базе регистров, 227
  влияние независимости от платформы на отладку, 227
  влияние среды на конструкцию Lua, 226
  диалекты, созданные пользователями, 225
  замыкания в, 214
  используемые ресурсы, 224
  конструкция, влияние на будущие системы, 221
  обновление во время разработки, 222
  обработка чисел, 213
  ограничения, 213
  особенности тестирования, 224
  ошибки проектирования, 216
  параллелизм и, 214
  парсер для, 228
  полный набор функций, 223
  применение, 212
  проблемы дробления, 225
  программирование, рекомендации, 212
  простота, влияние на пользователей, 224
  разделение кода, 225
  расширяемость, 226
  сборка мусора в, 213
  сообщения об ошибках в, 228
  средства безопасности, 212
  таблицы в, 213
  учет мнений пользователей, 223
  что следовало сделать иначе, мнение разработчика, 216

M

make, утилита, 179
Mesa, язык, 507
ML, 263
  доказательство теорем с помощью, 264
  значение, возможность выражения на других языках, 265
  назначение, 280
  описание логики, 265
  проблемы разработки, 279
  система типов, 276, 279
  формальные спецификации, 279
  функции высшего порядка, 265
M, язык, 458, 465
MUMPS, 319

N

National Instruments Lab View, 464
NCES (Net-Centric Enterprise Services), 341
NetBeans, 370

O

OASYS, нотация, 339
Objective-C, язык, 313
  как расширение Си и Smalltalk, 314, 317, 334
  конфигурируемость, 336
  наследование
    множественное, недопустимость, 335
    одиночное, необходимость, 335
  применение, 315
  пространства имен, 335
  протоколы, 335
  размер кода, в сравнении с Си, 326
  сравнение с
    C++, 314, 334
    Smalltalk, 314
  уроки проектирования и разработки, 339
Objective-C, язык, 524
Objectory, методология, 404
OCaml, язык, 242
OMG (Object Management Group), 426
OMT, технология, 454
OOD, технология, 555

open source
  модель, 349, 350
  проекты, успешность, 519
OpenSSL, 343
Oracle, 295
OSGI, 349

P

PARC, проект, 509
patch, утилита, 179
PDF, формат, 499
PEP (Python Enhancement Proposal), 42
Perl, язык, 473
  CPAN, 487, 488
  вариативность, 478
  версия 6, 491, 493, 494, 495
  влияние
    APL, 84
    законов естественных языков, 474, 479
  контекст, 479
  множественность реализаций, 494
  области видимости, ограничения, 475
  переход от текстового инструмента к полному языку, 477
  применение, 476
  прозвища, 473
  развитие, 479, 484, 490
  роль сообщества, 486
  синкретичная конструкция, 485
Photoshop, 516
PostScript, язык, 497
  аппаратное обеспечение, учет возможностей, 500, 509
  баги в ROM, способы обойти, 499
  в сравнении с Фортом, 502
  графические модели печати, в сравнении с PDF, 507
    Apple, графическая модель, 508
    NeXT, графическая модель, 508
  дальнейшее развитие, 508
  двумерные конструкции, 501
  интерфейс JavaScript, 516
  кернинг, 506
  конструктивные решения для, 503
  лигатуры, 506
  написание вручную, 504
  отладка, трудности, 515
  отсутствие формальной семантики, 504
  передача полутонов, 505
  преимущества как конкатенативного языка, 499
  применение, 498
    в Сети, 521
  уроки разработки, 514
  шрифты
    масштабирование, 504
    растровые, 505
    создание, 506
  язык вместо формата данных, 498
Prelude, 261
Python, язык
  безопасность, 52
  версии, 66
    Python 3.0, 66
    Python 3000, 56
  восходящая разработка против нисходящей, 52
  динамический характер, 52
  добавление функций в, 40, 45
  изучение, 53
  легкость сопровождения, 50
  макросы, 57
  множественность парадигм, 47
  множественность реализаций, 60
  обработка чисел, 45
  параллелизм, 60
  поиск в большой базе кода, 61
  применение в проектировании, 51
  простой синтаксический анализатор, 56
  прототипирование, 51
  реализации, 58
  сборка мусора, 58
  строгость форматирования, 57
  типы программистов, влияние на проектирование, 48
  требования для новых версий, 44
  уроки проектирования, 65
  философия элегантности, 46, 54
  функции
    для новичков, 48
    для опытных программистов, 48
Python Enhancement Proposal (PEP), 42
Pythonic, значение слова, 41

Q

QA, 391
QuickCheck, 239, 562
Quill, сравнение с реляционными базами данных, 306

R

RAD, визуальное программирование, 33
  быстрая разработка приложений, 380
RAII (Resource Acquisition Is Initialization), 20
Rational Unified Process, 404
Rendezvous, проект, 307
RIAA (Recording Industry Association of America), 349
RISC-процессор, 340
Ruby on Rails, фреймворк, 395
RUP, методология, 410

S

SaaS (Software as a Service), 348, 351
SAIL (Stanford Artificial Intelligence Language), язык, 335, 525
SCA, 339
Scala, 365, 366
SCA (Software Component Architecture), 341
Scrum, 406
SCOOP, модель, 532
SDL, 415, 416, 417
Second Life, 451
SEQUEL (Structured English Query Language), 293
simplexity, 383
Simula, включение классов в C++, 17
Sketchpad, 427
Smalltalk, язык, 524
  броузер для, 395
  включение в Objective-C, 317, 334
  применение, 314
Smart Editor, 541
SMOP, 493
SOA, *см.* архитектура, ориентированная на сервисы, 339, 341
Software-IC, 340
SPARK, 545
Spec#, 544
Spence, Ian, 403
SQL, язык, 291, 292, 295
  атака инжекции кода, 305
  видимость снаружи, 301
  влияние на разработку языков в будущем, 299
  в сравнении с AWK, 184
  декларативный характер, 296
  знания, необходимые для работы, 307
  конструктивные принципы, 299
  масштабируемость, 303
  обновление индексов, 298
  отзывы пользователей, 304
  пользователи, преимущественно программисты, 306
  популярность, 301
  представления, применение, 297
  проблема
    параллельного доступа к данным, 297
    Хеллоуина, 298
  сложность, 305
  стандартизация, 309
  тесты юзабилити, 304
STM (Software Transactional Memory), 241
Swing, 357
System R, проект, 295

T

Taskmaster, библиотека, 340
Tcl/Tk, 180
Testing Framework, 545
True BASIC, 117, 118, 129
TrueType, 506

U

UCSB, 527
UML (унифицированный язык моделирования), 354, 414, 449
  SDL, влияние на усовершенствование, 416
  возможные модификации, 414
  доля
    используемая всегда, 455
    используемая постоянно, 450
  использование для генерации кода, 417, 427, 449
  как набор DSL, 414, 416
  как помощь в общении, 429
  обратная совместимость с, 435, 455

применение, 429, 449
проектирование, 414
размер проекта, 432
оптимальный, 418
реконструкция, возможность, 435, 437, 450
семантические определения, проблема, 414
сложность, 414, 415, 450
стандартизация, 456, 459
необходимость, 437
убеждение в преимуществах, 417, 418, 432, 449
удаление некоторых элементов, 415
упрощение, 431
уроки разработки, 425, 451
UNCOL (UNiversal Computer Oriented Language), 274
Unified Process, 414
UNIX, 340
переносимость, 150

V

VDM, метод, 214, 416
Vista, 517
Visual Basic, 180
как объектно-ориентированный язык, 129
ограничения, 129

W

WORA, 355
«write once, run anywhere», принцип, 355
WYSIWYG-редакторы, влияние на программирование, 133

X

XML, 308, 384
парсер, 488
XQuery, язык, 257, 308, 339
X Window система, долголетие, 176

Y

yacc, язык, 479
как трансформирующие технологии, 177
что нужно знать для работы, 144
Yahoo! Pipes, 172

A

абстрагирование данных
в C++, 18
в SQL, 300
абстракция
в функциональном программировании, 235
указатели внутри, в C++, 19
автоматическая проверка кода, 369
агенты, 549
в Интернете, 286
Ада, язык, 415, 471
Айверсон, Кеннет, 69, 70, 295
алгебраический язык, Бейсик как, 116
алгебра процессов, 533
Алгол, 524
Алгол-60, 428
алгоритмы
затопления, 494
имитации отжига, 456
Хиндли–Милнера, 263
алиасы, 545
анализ кода, в динамических языках, 47
аналогия с кирпичами (*см.* компоненты), 342
анонимные делегаты, 249
аппаратное обеспечение
PostScript, учет в, 500, 509
влияние
доступности, 140, 162, 209
на программирование, 220
на развитие Бейсика, 120
использование вычислительных мощностей, 100
как стимул инноваций, 517
прогноз, 315, 316
ресурс или ограничение, 99
требования для параллелизма, 100
эффективность, релевантность, 146
архитекторы, определение, 423
архитектура, ориентированная на сервисы (SOA), 338, 341, 343, 344, 347, 351, 352, 442
асимметричные сопрограммы, в Lua, 215
асинхронная работа, в Форте, 102
аспектная ориентированность, 404

ассемблер, 542
аудиоприложения, языковая среда для, 129
Ахо, Альфред
  AWK
    где применять, 140, 144, 146
    знания, необходимые для работы, 144
    размер обрабатываемых данных, 141
    роль в разработке, 140
  lex, что нужно знать для работы, 144
  yacc, что нужно знать для работы, 144
  аппаратное обеспечение,
    влияние на программирование, 140
    значение, 146
  безопасность и степень формализации, 148
  графические интерфейсы, ограничения, 147
  документация как помощь
    в проектировании программ, 154
    в разработке программ, 151
  исследования в компьютерных науках, 155
  командная строка, ограничения, 147
  крупные приложения, методы создания, 140
  курс по компиляторам, 151, 154
  мастерство программиста, 143
  математика, роль в компьютерных науках, 156
  облегчение отладки, проектирование языка для, 145
  обработка по шаблонам
    параллелизм, 157
    эволюция, 141
  обучение
    программированию, 143
    отладке, 154
  отладка, роль компилятора и языка, 156
  переносимость UNIX, 150
  практичность языка программирования, 143
  программирование, возобновление после перерыва, 155
  разработка языка программирования, учет требований пользователей, 142
  сложные алгоритмы, возможность понимания, 142
  творчество, роль в программировании, 143
  теория автоматов, 156
  теория и практика как мотивация, 151
  формализация семантики, 147
  язык для предметной области, 147
  языки программирования, долголетие, 149
  языки, специализированные для предметной области, 145
Ахо–Корасик алгоритм, 142

Б

баги
  доказательство отсутствия, 265
  при проектировании языка, 276
  проблема 2000 года, уроки, 278
  см. также ошибки, 265
башня из моделей, 270
безопасность ПО
  безопасность типов, 23
    в C++, 23
  в Lua, 212
  в Python, 52
  в динамических языках, 53
  влияние
    многоуровневой интеграции, 345
    языка, 196
    формализации языка, 148
  оценка важности, 447
  подходы к, 22
Бейсик, 113
  GOTO оператор, 114, 122
  True BASIC, 117, 118
  аппаратное обеспечение, влияние на, 120
  библиотеки, создание, 135
  большие программы, пригодность для, 117
  игнорирование пробельных символов, 132
  инкапсуляция, 118
  комментарии, 127
  компилятор
    однопроходный, 116, 122
    двухпроходный, 118

конструкция, 122
долголетие, 130
что учитывалось, 114
нечувствительность к пробельным символам, 117
номера строк в, 114, 133
обработка чисел, 115, 121
обучение программированию с помощью, 115, 117
отсутствие обязательности объявления переменных, 123
производительность, 118
уроки разработки, 130
библиотеки
как метод расширения языков, 149
конструкция, формализмы для, 148
определение содержимого, 134
разработка, в сравнении с разработкой языка, 381
создание в Бейсике, 135
бизнес-правила, 465
Биттнер, Курт, 403
«ближе к аппаратной части», принцип проектирования для C++, 20
Бойера–Мура алгоритм, 142
Бойс, Рэймонд, 291, 292
Буч, Гради
UML
генерация кода реализации из, 449
доля, используемая всегда, 455
доля, используемая постоянно, 450
как убедить в преимуществах, 449
конструкция, влияние на групповую работу, 450
обратная совместимость с, 455
применение, 449
реконструкция, возможность, 450
сложность, 450
стандартизация, 456, 459
уроки разработки, 451
Ада, 471
бизнес-правила, 465
влияние конструкции языка на конструкцию программ, 452
корпус литературы для программирования, 460
обучение программистов, 459, 462
ограничения как стимул инноваций, 463
ООП, влияние на правильность разработки, 469
параллелизм, 466
паттерны проектирования, 468, 470
прагматизм и творчество, конфликт, 458
простота, распознание, 467
«разработка старых месторождений», 469
разработка языка
в сравнении с программированием, 453
источник вдохновения, 462
сложность и ООП, 466
устаревшее программное обеспечение, методы работы с, 462
эффективность команд, 467
языки визуального программирования, 464
быстрая разработка приложенийt (RAD), 33
Бэкус, Джон, 415

В

Вайнбергер, Питер, 193, 197
AWK
большие программы на, усовершенствования, 199
в сравнении с SQL, 184
начальные проектные идеи, 183
файлы журналов, обработка, 184
что следовало сделать иначе, 188
аппаратное обеспечение, влияние на программирование, 209
безопасность, зависимость от языка, 196
влияние
реализации на разработку языка, 194
языка на производительность, 195
Лисп, уровень успеха, 201
локальные обходные пути и глобальные исправления, 189
маленькие программы, обработка нетекстовых данных, 185
математика, роль в компьютерных науках, 186

обучение
отладке, 188
программированию, 190
через Интернет, 186
объекты в сравнении с системными компонентами, 195
поиск проблем в программном обеспечении, 210
программирование
на примерах, 192, 205, 210
повышение мастерства, 188
переработка программ, 202
производительность, измерение, 207
простота, рекомендации к достижению, 190
разработка языка программирования, 197
математиками, 200
ревоюция, 199
стиль разработки, 142
Си, знак числа, 203
сообщения об ошибках, качество, 198
стимулирование творчества программистов, 189
успех, определение, 204
учет отладки, 198
функциональное программирование, полезность, 187
языки
расширяемые, 200
универсальные, 200
ван Россум, Гвидо
Python
безопасность, 52
изучение, 53
макросы, 57
обработка чисел, 45
отладка, 61
множественность парадигм, 47
множественность реализаций, 60
параллелизм, 60
применение в проектировании, 51
прототипирование, 51
реализации, 58
сборка мусора, 58
синтаксический анализатор, простота, 56
сопровождение, 50
тестирование кода, 61
строгость форматирования, 57
требования для новых версий, 44
уроки проектирования, 65
Python Enhancement Proposal (PEP), 42
Pythonic, смысл слова, 41
восходящая разработка против нисходящей, 52
динамические языки
анализ кода, 47
достоинства, 52
тенденция к, 66
инструменты для рисования, 61
отладка, 40
поиск в большой базе кода, 61
программисты
возобновление работы после перерыва, 62
определение хороших, 49
прием на работу, 49
учет при проектировании, 48
функции разных уровней, 48
мастерство, 50
проектирование интерфейса или API, 63
типизация
гибридная, 47
динамическая, 46
статическая, 46
философия элегантности, 46, 54
языки программирования
добавление функций, 40, 45
распространение, препятствия, 66
расширяемость, 65
варианты тестирования, 404
ввод/вывод, в Форте, 101
вездесущие системы, 287, 341
видеоприложения, языковая среда для, 129
визуальные языки программирования, 464
виртуальная машина (ВМ), 357, 378
висячие ссылки, 281
возврат на инвестиции, 412
восходящая разработка
в C++, 21
в Python, 52
в Форте, 91

встроенные приложения
для C++, 23
применение Форта, 96
выделение памяти, выполнение компилятором, 125
вызов с остаточными вычислениями, 380
вычислительная наука, преподавание
обучение языкам, 376

Г

Гамма, Эрих, 434
генерики, 372, 378, 536
в C#, 246
в Java, 245
генофонд, 492, 493, 495
Гешке, Чарльз, 497
open source, успешность проекта, 519
PostScript
как язык, а не формат данных, 498
кернинг и лигатуры, 506
команда разработчиков, управление, 502
масштабирование шрифтов, 504
передача полутонов, 505
применение в Сети, 521
растровые шрифты, 505
управление стеками, 502
уроки разработки, 514
аппаратное обеспечение, учет возможностей, 500, 509, 517
баги в ROM, способы обойти, 499
важность мастерства программиста, 507
влияние математической подготовки на разработку, 501
двумерные конструкции, поддержка, 501
достоинства конкатенативных языков, 499
история эволюции программного и аппаратного обеспечения, 509
компьютерные науки, необходимые для преподавания темы, 513
научно-исследовательские группы, управление, 510, 513
открытые стандарты, 519
создание лаборатории графических исследований, 510
стандартизация и связанные с ней проблемы, 520
языки программирования
долголетие, 515
популярность, трудность достижения, 517
гибридная типизация, 47
гипотеза Сепира–Уорфа, 452, 480
Гослинг, Джеймс
API, разработка, 373
AWT, 357
C++
многопоточность , проблемы, 369
указатели, проблемы, 368
C#, 368
Java
виртуальная машина, обоснование, 361
добавление функций и внешние библиотеки, 373
обратная совместимость, 357
отзывы пользователей, 371
ошибки, предотвращение и локализация, 362, 368
проверка индексов массивов, 359
раскрытие кода, последствия, 372
сложность, проблемы снижения, 356
ссылки, 368
Java EE, 357
JIT, 358
JVM
конструкция и популярность, 362
основа для новых языков, 366
удовлетворенность, 361
Scala, 365, 366
автоматическая проверка кода, 369
влияние конструкции языка на конструкцию программ, 369
влияние сетей на разработку языка, 360
документирование, 370
закон Мура, 364
использование Си-стека, 368
компьютерные науки, проблемы преподавания, 370
ООП,
правильное применение, 363
успех, 363

отладка, учет при разработке, 369
параллелизм, 364, 366
платформонезависимость, влияние Java, 373
портативные устройства, языки для, 359
программисты, рекомендации для, 370
производительность, практические последствия, 360
простота и мощь, связь между ними, 356
разработка языка
для личного использования, 367
для системного программирования, 358
сборка мусора, 367
сложность, разные уровни в системе, 356
формальные спецификации, 370
языки,
рост, 372
взаимовлияние, 360
графический интерфейс
в сравнении с командной строкой, 172
ограничения, 147, 186
гуголплекс, 478

Д

дальнодействие, 475
Даль, Оле-Йохан, 526
Дана Скотт, 265
Дейкстра, Эдсгер Вибе, 525
деловые правила, 465
Деннис, Джек, 443
денотационная семантика, 415
деревья DOM, 354
детерминизм, важность, 307
Джоахим Парроу, 267
Джонсон, Ральф, 434
Джошуа Блок, 182
дзен Питона, 41, 46, 54
динамическая типизация, 46
динамические языки
анализ кода в, 47
безопасность и, 53
преимущества, 52
тенденция к, 66, 395
доказательство теорем, 264, 277, 280
доказуемость, 270, 273
документирование
программного обеспечения
Javadoc инструмент, 370
важность, 210
в помощь разработке, 154
комментарии, 110, 219, 393
помощь разработке программ, 151
содержание, 110, 278
составление, 393
языка программирования
дробление стилей, 225
дружественность пользователю, 136
Дэвид Парк, 267
Дэвид Уокер, 267

Е

единообразие, в SQL, 300

Ж

жизненный цикл, модель, 404

З

закон Мура, 364
закон Старджон, 489
замыкания, 214, 222, 249, 397
в Lua, 214
в SQL, 299
затирание типов, 378
затопления алгоритм, 494

И

Иерусалимский, Роберто
Lua
for цикл, 222
асимметричные сопрограммы, 215
виртуальная машина
выбор ANSI C для, 227
регистровая, 227
влияние среды на конструкцию, 226
диалекты, 225
замыкания, 214
обновление во время разработки, 222
минимализм конструкции, 216
обработка чисел, 213

ограничения, 213
отзывы пользователей, 223
отладка, 227
параллелизм, 214
парсер, 228
полный набор функций, 223
применение, 212
проблемы дробления, 225
простота, влияние на пользователей, 224
разделение кода, 225
расширяемость, 226
сборка мусора, 213
сообщения об ошибках, 228
средства безопасности, 212
таблицы, 213
тестирование, 224
функции первого класса, 214
что следовало сделать иначе, 216
аппаратное обеспечение, влияние на программирование, 220
влияние конструкции языка на конструкцию программ, 228
влияние реализации на разработку языка, 228
документирование программ, 219
комментарии, роль, 219
компьютерные науки, 217
математика, роль в, 218
локальные обходные пути и глобальные исправления, 224
обучение
отладке, 219
программированию, 229
программисты, повышение мастерства, 217
разработка языка программирования, 229
при ограничениях на ресурсы, 224
тестирование кода, 219
успех, определение, 216
изоморфизм КарриГоварда, 247
инжекция кода SQL, 305
инженерная работа, 405
инкапсуляция, 339, 340, 442
в Бейсике, 118
преимущества, 131
инструменты для рисования, 61
интеграция, 339
на уровне чипов, 338
интеллектуальный анализ данных, 141
Интернет
влияние на разработку языка, 360
как представление агентов, 286
информатика
определение, 285
связь с техникой, 284
исключения, 362
искусственный интеллект (ИИ), 287
искусство, польза изучения для программистов, 460
исчисление общающихся систем (CCS), 266
исчисление процессов, 533

K

каноническая форма, 279
КарриГоварда изоморфизм, 247
Керниган, Брайан, 385
AWK, применение, 164
C++, обратная совместимость с Си, 176
Tcl/Tk, 180
Visual Basic, 180
аппаратное обеспечение, влияние на программирование, 162
важность реализации для разработки языков программирования, 174
изучение языков программирования, 158
код, облегчающий тестирование, 179
командная строка, возрождение для Интернета, 177
малые языки, развитие, 177
написание текста, связь с программированием, 160
обновление, учет условий, 181
обратная совместимость против инноваций, 177
обучение отладке, 161
ООП, польза, 171
основания для включения функций, 181
переписывание программ, периодичность, 175
построение крупных систем, 171

проблемно-ориентированные языки (DSL), 163
программирование
начальный интерес, 158
повышение мастерства, 161
практика, 161
тщательная ревизия перед поставкой, 182
Си, долголетие, 175
стиль разработки языка, 142
трансформирующие технологии, 177
успех, определение, 163
учет требований пользователей, 160
языки программирования,
проектирование, 174
разработка, 164
руководства по, 159
кернинг, в PostScript, 506
Кинг, Родни, 487
классы, моделирование и разработка, 330
см. также объектно-ориентированное программирование, 330
клеточные автоматы, 443
Кобол, 474
Кодд, Э. Ф., 291, 292
Кокс, Брэд, 313
Objective-C
в сравнении с C++, 334
как расширение Си и Smalltalk, 334
множественное наследование, отсутствие, 335
одиночное наследование, необходимость в, 335
пространства имен не поддерживаются, 335
протоколы, 335
open source, модель, 349, 350
архитектура, ориентированная на сервисы (SOA), 338, 341, 343, 344, 352
безопасность ПО, 345
доверие к ПО, 344
инкапсуляция, 340
качество ПО, улучшение, 348, 351
компоненты, 336, 339, 347
компьютерные науки, 351, 354
проблемы, 343
конфигурируемость, 336
облегченные потоки, 341
обучение программированию, 354
ООП увеличивает сложность приложений, 338
параллелизм и ООП, 337
полученное образование, 351
разделение труда в программировании, 345, 346, 353
сборка мусора, 335, 336
суперраспространение, 347, 353
уроки проектирования, 339
уроки разработки, 339
экономическая модель ПО, 343, 344, 348, 351
коллекции
большие неструктурированные, обработка в APL, 82
операции над каждым элементом, 81
последствия для проектирования, 80
команда разработчиков языка программирования, 189, 232, 235, 372, 389
командная строка
AWK, 146
возрождение, для Интернета, 177
в сравнении с графическим интерфейсом, 172
инструменты, ограничения, 147
объединение программ с помощью конвейера, 147
ограничения, 186
команды
программистов, 109
проектировщиков, 450, 502
комбинаторы парсеров, 252
комментарии, 110, 278
в C#, 393
в Бейсике, 127
значение, 219
компиляторы
качество кода, 109
написание, 98, 125, 151, 154
компонентность, отражение в языке, 532
компоненты, 336, 339, 347, 385
компьютерные науки
исследования, 155, 286
наука ли это, 217, 351, 354
проблемы, 282, 343

развитие в будущем, 519
роль математики, 74, 156, 218, 284
текущие проблемы, 394
преподавание
более сложные темы, 270
групповая работа, 151, 154
изучение нескольких языков, 128, 158
изучение путем преподавания, 143
необходимые предметы, 405, 418, 422
начала программирования, 36, 48, 75, 114, 125
необходимые предметы, 323, 332, 354, 370, 398, 513
обучение языкам, 115, 117, 118
подходы, 190, 252
примеры кода в учебниках, 37
сложные концепции, 322
функциональные языки, роль в, 252
успешность, 138
конвейеры
в математическом формализме, 147
для объединения программ, 147
консультативные советы клиентов, 304
контекст, 479
контрактное программирование, 523, 524, 528, 529
контракты, 525, 543
контрольные примеры как сценарии использования, 404
контроль типов, 246
и ошибки, 96
концепты
в C++, 246
смысла, 268
кооперативная многопоточность, в Форте, 103
Корасик, Маргарет, 157
косвенный шитый код, в Форте, 94
Крухтен, Филипп, 449
Курц, Томас Э.
True BASIC, 117, 118
Visual Basic
как объектно-ориентированный язык, 129
ограничения, 129
Бейсик
GOTO оператор, 114, 122
большие программы, пригодность, 117
игнорирование пробельных символов, 117, 132
как алгебраический язык, 116
комментарии, 127
конструкция, 114, 122
номера строк, 114, 133
обработка чисел, 115, 121
объявления переменных, необязательность, 123
однопроходный компилятор, 122
производительность, 118
анализ перед реализацией, 127
аппаратное обеспечение, 120
библиотеки, 134, 135
видео и аудиоприложения, языковая среда для, 129
влияние конструкции языка на конструкцию программ, 127
выбор используемых слов в зависимости от предметной области, 133
изучение программирования, 114
инкапсуляция, преимущества, 131
компиляторы, написание, 125
математический формализм в проектировании языков, 131
наследование, необходимость и задачи, 124
обучение
отладке, 134
программированию, 115, 117, 125, 138
языкам, в сравнении, 118
ООП, полезность, 128
повторное использование кода, необходимость и задачи, 125
полиморфизм, требование интерпретации на этапе исполнения, 123
программисты,
учет при разработке языка, 131
продуктивность, 136
редакторы WYSIWYG, влияние на программирование, 133
уроки разработки, 130
успех в программировании, определение, 135

учет требований пользователей при программировании, 135, 137
языки программирования
изучение, 128
простота, какова цель, 119

Л

Лав, Том, 313
C++
популярность, 315
сложность C++, 315
Objective-C
в сравнении с C++, 314
в сравнении с Си, 326
как расширение Си и Smalltalk, 314, 317
применение, 315
Smalltalk, сфера применения, 314
аппаратное обеспечение, прогноз, 315, 316
классы, моделирование и разработка, 330
ООП, ограниченность применения, 317
оценка простоты конструкции, 332
преподавание сложных технических понятий, 322
программирование
необходимость практического опыта, 323
обучение, 323, 324
прогноз, 315
программисты
как распознать хороших, 324, 327
рекомендации для, 332
продуктивность
зависимость от качеств программиста, 328
повышение, 331
распределенные команды, организация, 327
сопровождение ПО, численность программистов, 325, 326
успешность ПО, оценка, 333
устаревшее программное обеспечение, реинжиниринг, 328
языки
количество используемых, 320
новые, необходимость в, 321
понимание руководителями, 330
развитие, 316, 322
расширяемость, снижающая потребность в новых, 322
лексическая область видимости, 483
ленивые (отложенные) вычисления, 235, 248
Лига плюща, 324
лигатуры, в PostScript, 506
Ликлайдер, Джозеф Карл Робнетт, 510
Лисп, 201
литература по программированию, 460
лямбда-выражения, 249
лямбда-исчисление, 217, 256, 267, 397, 540
лямбда-функции, 398

М

малые языки, обобщение, 385
масштабирование шрифтов в PostScript, 504
математика
важность изучения, 191
роль в компьютерных науках, 74, 156, 186, 218, 284
(*см. также* теоремы), 264
математики, как разработчики языков, 200, 218
математический формализм
в проектировании языков, 131
использование конвейеров, 147
машина фон Неймана, 267
Мейер, Бертран, 523
Eiffel
добавление функций, принятие решений, 549
история, 524, 528
обратная и прямая совместимость, 550
поточная сериализация, 541
расширяемость, 533
скрытие информации, 534
SCOOP, модель, 532
анализ перед реализацией, 542
влияние знания иностранных языков на разработку языка программирования, 531, 538
доказательнство правильности программ, возможность, 544

компонентность, отражение в языке, 532
контрактное программирование, 528, 530
малые и крупные программы, различие методов, 540
математическая и лингвистическая точки зрения на программирование, 539
многократное использование, 534
параллелизм и ООП, 532
работа с объектами вне языка, 531
развитие, 546
разработка,
управляемая моделью, 543
начиная с ядра, 539
уроки, 552
цельная, 541
структурное программирование против ООП, 540
универсальность (genericity), 536
философия программирования, 525
менеджер структур данных, 445
метасимволы, 485
метаязыки для моделей, 268
микропроцессоры, исходный код в, 106
Милнер, Робин, 263
CCS и пи-исчисление, 266
LCF, ограничения, 264
ML
описание логики, 265
назначение, 280
проблемы разработки, 279
баги
в структуре языка, 276
доказательство отсутствия, 265
«вездесущие системы», 287
влияние конструкции языка на конструкцию ПО, 274
влияние парадигм на программистов, 275
доказательство теорем с помощью LCF и ML, 264
доказуемость, 273
исследования в компьютерных науках, 286
каждому программисту свой язык, 275
комментарии и документация, 278
компьютерные науки
математика, роль в, 284
проблемы, 282
метаязыки для моделей, 268
многократно используемые концепты смысла, 268
модели для систем, 267, 273, 282
неразрешимость в моделях низкого уровня, 269
обучение теоремам и доказуемости, 270
общение между агентами, 286
определение исследователя в области информатики, 285
параллельные системы, анализ, 266
проблема 2000 года, уроки, 278
программисты, повышение мастерства, 278
программы, способность компьютера определить смысл, 277
связь информатики с техникой, 284
системы типов
разрешимость, 270
создаваемые ограничения, 276
структурные проблемы в программах, как избегать, 278
теория смысла, 288
уровни моделей, 268
физические процессы, влияние на модели, 269
функции высшего порядка, необходимость, 265
языки
оберегающие пользователей от ошибок, 288
проверка, 283
пересмотр, 281
разработка, определение, 275
минимализм, в структуре Форта, 93
многозадачность, 466
многократное использование, 534
и SOA, 442
ООП, 439, 440
многократно используемые концепты смысла, 268
многопоточность
как предшественник параллельной обработки, 95
кооперативная, в Форте, 103

математические программы и, 364
примитивы синхронизации, 366
проблемы в C++, 369
фреймворки Java, 364
многоядерные компьютеры, 103
многоязычные виртуальные машины, 377
множественные парадигмы
в Python, 47
Могенс Нильсен, 267
модальная логика, 269
модели
COM, 171
данных, для разработки языка, 296
жизненного цикла, 404
систем, 267, 273, 282
монады, 546
Haskell, 280
музыка, связь со способностями к программированию, 325, 388, 460
Мура закон, 364
Мур, Чарльз Х., 89
colorForth, 93
аппаратное обеспечение
использование вычислительных мощностей, 100
как ресурс или ограничение, 99
влияние конструкции языка на конструкцию программ, 106
возобновление работы после перерыва, 98
восходящее проектирование, 91
групповая работа в программировании, 109
документирование программ, 110
компиляторы,
написание, 98
качество кода, 109
кооперативная многопоточность, 103
микропроцессоры, 106
минимализм конструкции, 93
операционные системы, 98, 101
параллельная обработкка данных, 95
возможные подходы, 100
патентование приложений, 111
портирование, 101
приложения для малых компьютеров в сети, 103
причины ошибок и их обнаружение, 96, 107
программисты, как распознать хорошего, 109
проектирование
приложений, 103
управляемое предметной областью, 92
стек, глубина, 103
уроки
проектирования, 99
разработки, 94
устаревшее программное обеспечение, проблемы, 104
Форт
асинхронное выполнение, 102
вызов подпрограмм на основе стека, 91
и встроенные приложения, 96
как инструментальный набор, 90
конструкция, долголетие, 92
косвенный шитый код, 94
легкость сопровождения, 105
легкость чтения, 92, 97
программирование, рекомендации, 108
программисты восприимчивость, 91
простота, 108
слова, образующие синтаксис, 90, 92, 105
средства ввода/вывода, 101
управление стеками, 107
условные операторы, 107
циклы в Форте, 108
элегантность решения, определение, 97
мэйнфрейм, 229
мэшапы, 172

Н

набор инструментов, 90
набор символов, для APL, 72, 80
наследование, необходимость и задачи, 124
научно-исследовательские группы, управление, 510, 513
неизменяемость, 382

нечувствительность к пробелам, в Бейсике, 117, 132
нисходящая разработка
в C++, 21
в Python, 52
номера строк в Бейсике, 114, 133
нормальная форма, 279

О

обеденное тестирование, 544
Оберон, язык программирования, 376
области видимости, в Perl, 475
облачные вычисления, 518
облегченные потоки, 340
обобщенное программирование
в C++0x, 32
как альтернатива ООП, 18, 27
как парадигма C++, 24, 25
уменьшение сложности, 17
обобщенные типы в Haskell, 249
обработка по шаблонам
алгоритмы, задействующие параллелизм, 157
развитие, 141
образование (*см.* компьютерные науки, преподавание), 36
обратная совместимость, 258
в Java, 357
в JVM, 378
в UML, 455
для будущей ревизии UML, 435
общее управление ресурсами, 20
общение
значение в информатике, 285
между интерактивными агентами, 286
программирование как вид, 11
общие переменные, в APL, 81
объектно-ориентированное программирование (ООП), 24
в Visual Basic, 129
в Lua, 226
влияние на правильность разработки, 469
инкапсуляция, 442
как парадигма C++, 24
масштабируемость, для сложных программ, 439, 466
многократное использование, 440
обобщенное программирование как альтернатива, 27
ограниченность применения, 317
параллелизм, 28, 337, 442, 532
полезность, 128, 171
применение, в сравнении со структурированным программированием, 540
проблемы, 398
работа с объектами вне языка, 531
сложность, 17, 26, 27
увеличение приложений, 338
усилия правильного применения, 363
успех, 363
объекты, в сравнении с системными компонентами, 195
объем данных, рост, 209
объявления, отсутствие в APL, 84
ограничения
области видимости в Perl, 475
как стимул инноваций в программировании, 463
операционные системы, 98, 101
(*см. также* ядра), 24
оракулы тестирования, 544
ортогональность, в SQL, 299
открытые стандарты, 519
отладка кода, 40, 145
C#, 392
C++, 21
Lua, 227
PostScript, трудности, 515
Python, 61
влияние конструкции языка, 198, 386
обучение, 134, 154, 161, 188, 219
роль компилятора, 156
трудоемкость, 173
учет при разработке, 369
функциональное программирование и, 239
ошибки
влияние конструкции языка на уменьшение, 197
обнаружение в Форте, 96, 107
обработка в функциональном программировании, 238
при разработке языка, 197
(*см. также* баги), 265

П

Пайк, Роб, 161
парадигмы
  в C++, 17, 24, 26
  влияние на программистов, 275
  множественные
    в Python, 47
параллелизм, 95, 466
  анализ параллельных систем, 266
  в APL, 80, 84
  в C++, 29
  в C++0x, 32
  в Lua, 214
  в Python, 60
  в SQL, 297
  влияние на разработку языка, 364, 397
  добавление в язык, 382
  и ООП, 28, 337, 398, 532
  функциональные языки и, 444
  подходы к, 100
  применение, 347
    в обработке по шаблонам, 157
  проблемы, 394, 396
  распределение по сети, 30
  технические требования для отдельных областей, 364
  функциональное программирование, 237
парсер, 354
  XML, 488
  Lua, 228
парсинг, 479
патенты на программы, 111
паттерны, 423, 468, 470
  движение за, 424, 433
Пейтон-Джонс, Саймон
  Haskell
    конкурирующие реализации, 259
    конструкция, влияние на будущие системы, 261
    развитие, 259, 261
    система классов, 243
  влияние конструкции языка на конструкцию программ, 250
  команда разработки языка, 232, 235
  обратная совместимость, 258
  преподавание компьютерных наук, 253
  формальная семантика, бесполезность, 254
  функциональное программирование, 235, 243
передача знаний, 404, 413, 419, 422, 424
передача полутонов в PostScript, 505
пи-исчисление, 266
Питерс, Тим (дзен Питона), 41, 46, 54
платформонезависимость, влияние Java, 373
ПО
  доверие, 344
  качество, улучшение, 348, 351
  сложность и ООП, 338, 466
  разделение труда, 345, 346, 353
  устаревшее, реинжиниринг, 328
  экономическая модель, 343, 344, 348, 351
повторное использование кода, необходимость и задачи, 125
подобие, 267
позднее связывание, 513
полиморфизм, требование интерпретации на этапе исполнения, 123
полнота, в SQL, 299
пользователи
  учет мнений о Lua, 223
  учет при программировании, 135, 137, 160
  учет при разработке языка, 142
пользовательские и встроенные элементы языка, различие обработки, 79
портативные устройства, язык для, 359
постфиксные операторы, в Форте, 96, 97
потоки
  косвенный шитый код, 94
  облегченные, 340, 341
  параллелизм, 337
  (*см. также* многопоточность), 94
прагматизм и творчество, конфликт, 458
прагматики, 481
приложения (*см.* программы), 210
примеры кода
  в учебниках по программированию, 37, 159

принцип
подстановки Лисков, 528
разрешения Робинсона, 264
пробелы, игнорирование в Бейсике, 117, 132
проблема 2000 года, уроки, 278
проблемно-ориентированные языки (DSL), 384
Lua, 226
недостатки, 382, 384
переход к универсальным, 484
реконструкция UML как набор, 414, 416
программы в качестве, 77
рост, 163
существование, 394
программирование, 11
анализ перед реализацией, 127, 542
будущее, 519
влияние аппаратного обеспечения, 140, 209, 220
возобновление после перерыва, 98, 155
как вид общения, 11
как инженерная работа, 405
компоненты в, 336, 339, 347
не делай, если не можешь сделать хорошо, 210
ограничения как стимул инноваций, 463
отладка (*см.* отладка кода), 36
по примерам, 192, 205, 210
преподавание (*см.* компьютерная наука), 36
прогноз, 315
подход в разных странах, 408
совершенствование методов и процессов, 410
умные агенты как партнеры, 419
с математической точки зрения, 539
сравнение
с доказательством математических теорем, 187, 208
с написанием текста, 160
с разработкой языка, 194
с точки зрения лингвистики, 539
творчество, 143
тестирование (*см.* тестирование кода), 21
учет требований пользователей, 135, 137, 160
характер, изменение со временем, 100
цельная разработка, 541
программисты
важность мастерства в сравнении с конструкцией языка, 507
влияние парадигм, 275
выявление хороших, 109
знания, привязанные к языку, 418, 424
как распознать хорошего, 219, 324, 327
команды
важность, 109
в классе, 151, 154
необходимые навыки, 325
обучение, 370
организация распределенных, 327
продуктивность, 136
рост размеров, 170
стимулирование творчества, 189, 310
численность, 310, 495
эффективность, 467
мастерство, 143
необходимость практического опыта, 323
обучение отладке, 324
обычные люди как, 396
определение хороших, 49
повышение мастерства, 161, 188, 217, 278, 394
прием на работу, 49
организация, 409
обучение отладке, 459, 462
пользователи в качестве, 420, 425
творчество (*см.* творчество), 462
продуктивность, 386
производительность, 136, 195, 207
типы
восприимчивые к Форту, 91
учет при проектировании, 48, 71
функции разных уровней, 48
программные патенты, 111

программы, 210
доказательство правильности, возможность, 544
документация (*см.* документирование программ), 110
как проблемно-ориентированные языки, 77
легкость сопровождения, 189, 202, 325, 326
локальные обходные пути и глобальные исправления, 189, 224
маленькие, обработка нетекстовых данных, 185
написанные в 1970-х, переработка, 100
оценка успешности, 333
переписывание, периодичность, 175, 202
построение крупных систем, 171
прелесть элегантности, 143
проблемы, обнаружение, 210
производительность, 209
размер, неуклонный рост, 170
сложность, и ООП, 439
способность компьютера определить смысл, 277
структурные проблемы, как избегать, 278
теория и практика как мотивация, 151
тщательная ревизия перед поставкой, 182
продуктивность
пользователей, рост благодаря SQL, 297, 301, 302
программистов
влияние качества программиста, 328
влияние языка программирования, 195, 386
измерение, 207
повышение, 331
при работе в одиночку, 136
проектирование
интерфейса, 63, 181
программная инженерия, 11
управляемое предметной областью, 92
языка программирования, 174
важность реализации для, 165
влияние на конструкцию программ, 78
личный подход к, 77
расширение существующего, 16
формализм математический, 131
производительность
C++, влияние на проектирование, 21
Бейсик, 118
практические последствия, 360
простота
в SQL, 300
в разработке языка, обнаружение, 383
распознавание, 421, 432, 467
связь с мощью, 356
советы по достижению, 190
Форт, 108
языки, задачи, 119
пространства имен
в APL, 83
в Objective-C, отсутствие поддержки, 335
протоколы, в Objective-C, 335
профили, 435
публикации, 24

P

равенство типов, 245
разделение труда, в разработке ПО, 345, 346, 353
разработка
управляемая моделью, 543
управляемая предметной областью, 133, 145, 147, 364
разработка языка программирования, 164, 221, 229
баги в, 276
важность реализации, 174, 194
влияние
SQL, 299
знания иностранных языков, 538
на конструкцию программ, 106, 127, 228, 250, 274, 369, 452
реализации, 228
сетей, 360
среды, 377

возможность научного подхода, 170, 387
в сравнении с программированием, 194, 453
в сравнении с разработкой библиотеки, 381
выбор синтаксиса, 198
для портативных устройств, 359
для системного программирования, 358
источник вдохновения, 462
как оценить простоту, 332, 383
команды, 189, 232, 235, 450
демократическая сущность, 372
управление, 389, 502
математики, 200, 218
модели данных для, 296
научный подход, 197
начиная с базового набора функций, 319, 539
обратная совместимость против инноваций, 177
определение, 275
практичность, 143
предпочтения разработчика, 197
проблемно-ориентированного, 145
программисты, 131
прототипы для, 168
революция, 199
связь с реализацией, 376
совершенствование процессов, 169
сокращение числа ошибок, 166, 197
уроки, 373
учет
отладки, 198
требований пользователей, 142
формализация, преимущества, 147
формализмы, 295
четкость проекта, 207
разыменование нулевого указателя, 552
Рамбо, Джеймс
SOA, 442
UML
генерация кода реализации, 427
применение, 429
реконструкция, 435, 437
убеждение в преимуществах, 432
упрощение, 431
ООП
масштабируемость, 439
многократное использование, 439, 440
архитекторы, определение хороших, 423
безопасность, оценка важности, 447
важность учета будущих модификаций, 448
движение за паттерны, 424, 433
инкапсуляция, 442
компьютерные науки, необходимые предметы, 422
облегчение общения, 429
параллелизм, 442
передача знаний, 422, 424
подготовка, влияние на разработку ПО, 421
программирование пользователями, 425
простота, распознание, 432
разделение труда в программировании, 437
размер проекта, 432
симметричные взаимоотношения, 444, 448
стандартизация, необходимость, 437
умение программировать, связь с языками, 424
универсальная модель/язык, 430
уровни абстракции, 430
уроки разработки, 425
эмергентные системы, 434
юзабилити языков, 428
растровые шрифты, обработка в PostScript, 505
расширения Glasgow, 260
расширяемость, в SQL, 300
регрессионное тестирование, 545
регулярные выражения (*см.* обработка по шаблонам), 142
редактор связей, 118
Рейзнер, Филлис, 304
реляционные базы данных, влияние APL, 84
ресурсы
разработка при ограничениях, 224
управление, 20
рефакторинг в C++, 27

С

Сазерленд, Айвен, 427
сборка мусора, 335, 358, 387
  PostScript Level II, 508
  в JVM, 367
  в Lua, 213
  в Objective-C, 336
  в Python, 58
семантика значений, 20
семантическая плотность, 466
Сепира–Уорфа гипотеза, 452, 480
сетевые ресурсы
  C++0x FAQ, 32
  комиссия по стандартизации C++, 32
сети
  SOA и, 341
  влияние на разработку программ, 350
  из малых компьютеров, приложения для, 103
  использование PostScript вместо HTML и JavaScript, 521
  примитивы синхронизации, 366
  распределение, параллелизм и, 30
  суперраспространение, 347
сеть Петри, 267
Си
  C++
    качество программ, в сравнении с, 25
    перенос кода на, 25
  Objective-C
    как расширение, 314
    размер кода, в сравнении с, 326
  долголетие, 175
  знаки, 203
  как язык системного программирования, 358
  почему ядро пишут на, 24
  производительность, важность, 359
  система типов, 16
симметричные взаимоотношения, 444, 448
Симула-67, язык, 524
синий экран смерти, 359
синкретичная конструкция Perl, 485
синтаксический анализатор ссылок, 309
синтез тестов, 545
синхронизация связи, 267
система классов, в Haskell, 243
системы
  модели для, 267, 273
  шире не быстрее, 364
системы типов
  в ML, 279
  налагаемые ограничения, 276
  разрешимость, 270
Си, язык, 527
слабые таблицы, 222
словарь пользователя, использование в Форте, 92
сложные алгоритмы, возможность понимания, 142
слоты, 474
сообщения об ошибках
  в Lua, 228
  качество, 198
социальная инженерия, 410
Спенс, Ян, 403
специализация в программировании, 285, 437
спецификации
  формальные (*см.* формальные спецификации), 370
списочные выражения
  в Haskell, 249
  в Python, 249
ссылки в Java, вместо указателей, 18
стандартизация
  APL, 73
  C#, 391
  UML, 437, 456, 459
  проблемы, 520
статическая проверка интерфейсов, недостатки, 27
статическая типизация, 46
стек
  для вызова подпрограмм в Форте, 91, 103
  управление в Форте, 107
стековая организация, в PostScript, 502
Страуструп, Бьерн
  C++0x, сетевой FAQ, 32
  C++
    будущие версии, 31
    перенос кода с Си, причины, 25
    поддержка параллелизма, 29
    поддержка различных парадигм, основания для поддержки, 17

почему это не просто объектно-ориентированный язык, 24
принципы и практика применения, 36
сложность в сравнении с Java, 17
указатели в сравнении с Java, 18
уроки изобретения, 33
безопасность программ, 22
встроенные приложения, 23
научные интересы, 36, 38
общее управление ресурсами, 20
о создании нового языка, 32
отладка кода, 21
параллелизм и распределение по сети, 30
параллельность и связь с ООП, 28
почему ядро не пишут на C++, 24
примеры кода в учебниках, 37
работа в промышленности, 36
расширение существующего языка, основания для, 16
семантика значений, 20
системное программное обеспечение, использование, 23
сложность ООП, 26
тестирование кода, 21
структурное программирование, в сравнении с ООП, 540
суперраспространение, 347, 353
сценарии
возникновение концепции, 403
оболочки AWK, 146
телефонной нагрузки, 403

Т

таблицы, в Lua, 213
тагмемика, 479
Тафти, Эдвард, 452
творчество
важность, 405
возможность применения, 419
в программировании, есть ли оно, 143
и прагматизм, конфликт, 458
как функция программиста, 387
конфликты, преимущества, 459
необходимость, 462
стимулирование программистов, 189, 310
тезис Черча–Тьюринга, 499
теоремы, доказательство
LCF и ML, 264
в сравнении с программированием, 187, 208
как задача, 280
преподавание в компьютерных науках, 270
с помощью системы типов, 277
теория автоматов, 156
тестирование, 219
C++, 21
Lua, 224
Python, 61
контрольные примеры, 404
методом черного ящика, 179
написание вспомогательного кода, 179
техника, связь с информатикой, 284
технология на базе правил, 420
тонкие клиенты, 347
трансформация, технологии, 177

У

Уилтамут, Скотт, 390
указатели
в C++, в сравнении с Java, 18
компилятор, 125
не используются в APL, 84
умная пыль, 95
умные агенты для программирования, 419
универсальные массивы, в APL, 80
универсальные языки, 384
компьютерно-ориентированные, 274
Уодлер, Филип
Haskell
влияние на другие языки, 249
система классов, 245
система типов, 246
влияние конструкции языка на конструкцию программ, 251
обобщенные типы, 249
списочные выражения, 249
формальная семантика, бесполезность, 256
функциональные замыкания, 249

Уолл, Ларри
CPAN, 487, 488
Perl
контекст в, 479
переход от текстового инструмента к полному языку, 477
множественность способов достижения цели, 478
применение, 476
развитие, 479, 490
влияние на языки программирования законов естественных языков, 474, 479
лингвистики, 482
команды программистов, численность, 495
лексическая область видимости, 483
множественность реализаций, 494
развитие универсального языка из специализированного, 484
роль сообщества, 486
сложность языков, 356
уменьшение, 482
успех экспериментов, 485
языки в сравнении с инструментами, 483
Уорнок, Джон, 497
JavaScript, интерфейс, 516
PostScript
графические модели печати в сравнении с PDF, 507
как язык, а не формат данных, 498
конструктивные решения, 503
написание вручную, 504
отсутствие формальной семантики, 504
аппаратное обеспечение, учет возможностей, 500, 509
баги в ROM, способы обойти, 499
будущее компьютерных наук и программирования, 519
двумерные конструкции, поддержка, 501
долголетие универсальных языков программирования, 515
достоинства конкатенативных языков, 499
кернинг и лигатуры, 506
масштабирование шрифтов, 504
отладка, трудности, 515
растровые шрифты, 505
создание шрифтов, 506
стандартизация и связанные с ней проблемы, 520
управление стеками в PostScript, 502
уровни абстракции, 430
условные операторы, в Форте, 107
устаревшее (legacy) ПО
как избегать проблем, 329, 346
подходы к, 328
проблемы, 104, 189, 191
подходы к, 411, 462
устойчивость, в SQL, 300

Ф

файлы журналов, обработка в AWK, 184
файлы, обработка в APL, 83
Фалкофф, Эдин, 69, 295
APL
влияние на Perl, 84
влияние на проектирование реляционных баз данных, 84
набор символов, 72, 80
обобщенные массивы, 80
обработка файлов, 83
общие переменные, 81
пространства имен, 83
стандартизация, 73
трудность изучения, 72
удачные стороны, 86
указатели отсутствуют, 84
что следовало сделать иначе, 86
System 360, формальное описание, 70
влияние конструкции языка на конструкцию программ, 78
карманные устройства, 73
коллекции,
влияние на проектирование, 80
неструктурированные, обработка, 82
операции над каждый элементом, 81
компьютерные науки, роль математики, 74
конструкция языка
долголетие, 74
история, 70, 78
необязательность объявлений, 84

обучение программированию, 75
параллелизм, 80, 84
пользовательские и встроенные элементы языка, различие обработки, 79
программисты, учет разных типов при проектировании языка, 71
программы в качестве проблемно-ориентированных языков, 77
проектирование языков, личный подход, 77
синтаксис
на основе алгебраической нотации, 71, 76
простота/сложность, 71, 76, 81
уроки проектирования, 79, 85
эффективное использование ресурсов, 72
Фигейреду, Луис Энрике де
Lua
VM для, выбор ANSI C для, 227
асимметричные сопрограммы, 215
влияние среды на конструкцию, 226
ошибки, 216
парсер, 228
применение, 212
сообщения об ошибках, 228
средства безопасности, 212
таблицы, 213
что следовало сделать иначе, 216
аппаратное обеспечение, влияние на программирование, 220
диалекты пользователей, 225
комментарии, роль, 219
конструкция, влияние на будущие системы, 221
локальные обходные пути и глобальные исправления, 224
математика, роль в компьютерных науках, 218
обучение отладке, 219
программирование, рекомендации, 212
программисты
как распознать хороших, 219
повышение мастерства, 217
разработка при ограничениях на ресурсы, 224
разработка языка программирования, 229
математиками, 218
тестирование кода, 219, 224
успех, определение, 216
физические процессы, зависимость моделей от, 269
философия «меньше значит больше», 64
формальная семантика, 254
в PostScript, 504
полезность, 254, 258
преимущества для разработки языка, 295
формальные спецификации для языков, 454
для C#, 392
необходимость, 370
Форт, 89
асинхронное выполнение, 102
в сравнении с PostScript, 502
выбор слов, 105
вызов подпрограмм на основе стека, 91, 103
для встроенных приложений, 96
использование словаря пользователя, 92
конструкция
влияние на конструкцию ПО, 106
долголетие, 92
кооперативная многопоточность, 103
косвенный шитый код, 94
легкость
сопровождения, 105
чтения, 92, 97
минимализм конструкции, 93
патентование приложений, 111
портирование, 101
постфиксные операторы, 96, 97
причины ошибок и их обнаружение, 96, 107
программирование, рекомендации, 108
программисты, восприимчивость к, 91
проектирование приложений, 103
простота, 90, 92, 108
синтаксис малых слов, 90, 92
средства ввода/вывода, 101
управление стеками, 107

уроки разработки, 94
условные операторы в, 107
циклы, 108
Фортран, 542
фреймворки, изучение, 419
фреймы, 475, 483
функции, 403
высшего порядка в ML, 265, 396
первого класса, в Lua, 214
функциональное программирование, 235, 243, 394
Scala, 365
абстракция в, 235
в учебных планах по компьютерным наукам, 398
долголетие, 243
изучение, 239
ленивые вычисления, 235, 248
обработка ошибок, 238
отладка, 239
отсутствие побочных эффектов, 235, 238
параллелизм, 237, 444
полезность, 187
популярность, 240
функциональные замыкания, в Haskell, 249

X

Хейлсберг, Андерс, 375
API, разработка, 393
C++, развитие, 384
C#
долголетие, 388
комментарии, 393
отзывы пользователей, 382, 389
отладка, 392
развитие, 388
ECMA, стандартизация C#, 391
JVM, обратная совместимость с, 378
безопасность против свободы творчества, 387
динамические языки программирования, 395
документирование ПО, 393
команда разработчиков, управление, 389
компьютерные науки, проблемы, 394
личные мотивы в проектировании языков, 380
многократное использование существующих компонентов, 385
многоязычные виртуальные машины, 377
обучение языкам, 376
ООП, проблемы, 398
отладка, разработка языка для облегчения, 386
параллелизм,
влияние на разработку языка, 397
добавление в язык, 382
обработка средой, 398
проблемы, 394, 396
проблемно-ориентированные языки, 384, 394
программисты, повышение мастерства, 394
простота разработки языка, 383
разработка языка программирования, 376, 383
научный подход, 387
связь между проектированием и реализацией языка, 376
уроки разработки, 399
формальные спецификации, 392
функции высшего порядка, 380, 396
Хеллоуин, проблема в SQL, 298
Хелм, Ричард, 434
Хоар, Тони, 524
Хофштадтер, Дуглас, 431
Худак, Пол
Haskell
влияние на другие языки, 249
конструкция, 261
влияние конструкции языка на конструкцию программ, 250
команда разработки языка, 232, 235
обучение программированию и компьютерным наукам, 252
функциональное программирование, 235, 243
Хьюз, Джон
Haskell, влияние на другие языки, 249
команда разработки языка, 232, 235
ленивые вычисления, 248
функциональное программирование, 235, 243

Ц

цвет, передача полутонов в PostScript, 505
цельная разработка, 541
циклы
  возможные альтернативы, 81
  в Форте, 108

Ч

Чемберлен, Дон, 291
  Excel в сравнении с реляционными базами данных, 306
  Quell в сравнении с реляционными базами данных, 306
  SQL
    влияние на разработку языков программирования, 299
    декларативный характер, 296
    история, 292, 295
    конструктивные принципы, 299
    отзывы пользователей, 304
    популярность, 301
    представления, применение, 297
    сложность, 305
  XML, 308
  XQuery, 308
  атака инжекции кода, 305
  детерминизм, важность, 307
  знания, необходимые для работы, 307
  команды программистов, численность, 310
  масштабируемость, 303
  модели данных для разработки языка, 296
  параллельный доступ к данным, проблемы, 297
  пользователи и программисты, 306
  проблема Хеллоуина, 298
  стандартизация SQL и XQuery, 309
  тесты юзабилити, 304
  успех, определение, 311
  формализм, польза для разработки языка, 295
  эффект видимости снаружи, 301
  языки, вызывающие интерес, 296
Черча–Тьюринга тезис, 499
четвертое поколение компьютерных языков, Форт, 90
числа, обработка
  в Lua, 213
  в Python, 45
    с произвольной точностью, 45
  в Бейсике, 115, 121
читаемость Форта, 97

Ш

шаблоны, в C++, 27
шаблоны проектирования, 433
швейцарский армейский нож, 483
Шиллер, Стив, 505

Ы

Ын, Пан Вей, 404

Э

Эйнштейн, Альберт, 421
экономическая модель ПО, 343, 344, 348, 351
эмергентные системы, 430, 434

Я

ядро, язык для написания, 24
языки, естественные, в сравнении с программными, 11
язык интегрированных запросов, 386
языки программирования, 62
  безопасность против свободы творчества, 387
  влияние лингвистики, 482
  влияние
    на программы, 531
    на продуктивность, 386
    на производительность, 195
  в сравнении с естественными языками, 11
  добавление функций в, 356, 381
  долголетие, 149, 515
  интерфейсы, элегантность, 181
  количество используемых, 320
  личное восприятие программистом, 275
  малые
    возрождение, 177
    развитие, 177
    расширение, 200, 385

налагаемые ограничения, 119
новые, потребность в, 321
обновление, учет условий, 181
обучение языкам, 376
основания для включения функций, 181
отладка, 40
переход от специализированных к универсальным, 484
понимание языков руководителями, 330
популярность языка, трудность достижения, 517
препятствия к распространению, 66
проблемно-ориентированные (*см.* проблемно-ориентированные языки), 163
проверка, 283
размер, 149
распознавание сильных сторон, 167
расширяемость, 65, 149, 200, 322
реализация, контролируемые факторы, 220
ревизия, 281
рост, 372
семейства, 275
сокращение числа ошибок, 288
теория смысла, 288
тестирование новых функций, 169
требования совместимости, 40
уменьшение сложности, 482
универсальные, в идеале, 167, 200
управление развитием, 34, 40, 134, 316, 322
успех экспериментов с, 485
формальные спецификации, 454
юзабилити, 428
язык паттернов, 470
Якобсон, Айвар
Ericsson, 402
Object-Oriented Software Engineering, 403
UML
как набор DSL, 414, 416
как убедить в преимуществах, 417, 418
проектирование, 414
развитие в будущем, 414
сложность, 414, 415
SDL, влияние на усовершенствование UML, 416
аспектная ориентированность, 404
аспектно-ориентированная разработка, 404
генерация кода реализации, 417
изучение сред, 419
команды программистов, формирование, 409
компьютерные науки, преподавание, 405, 418
методы и процессы программирования, совершенствование, 410
передача знаний, 404, 413, 419
программирование
изучение, 402
подход в разных странах, 408
пользователями, 420
связь с естественными языками, 418
простота, распознание, 421
размер проекта, 418
социальная инженерия, 410
сценарии использования, разработка концепции, 403
технология на основе правил, 420
умные агенты в программировании, 419
устаревшее ПО, 411