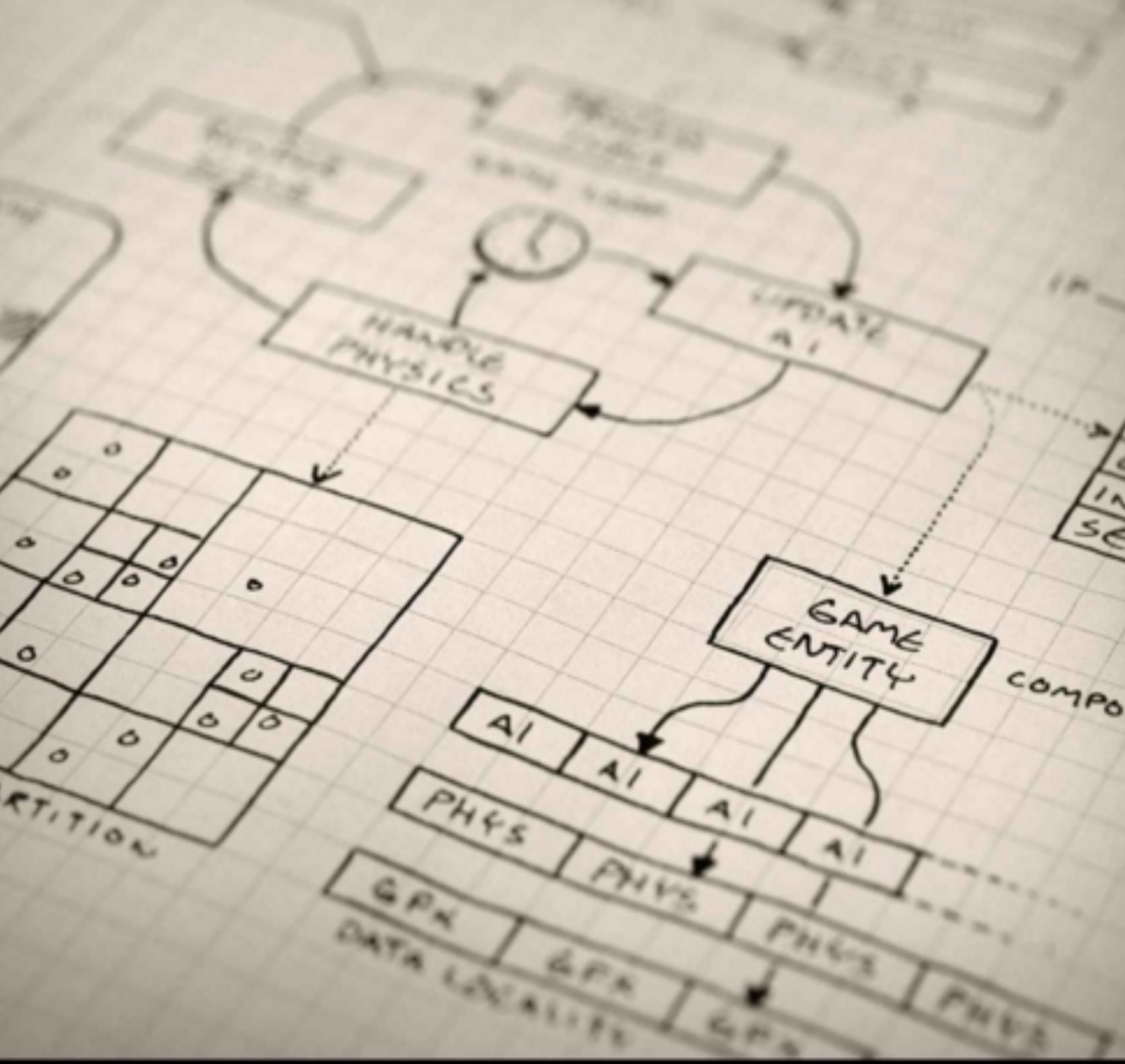


Шаблоны
Игрового Программирования

Роберт Найстром

Содержание

Шаблоны игрового программирования

Приветствую тебя, разработчик игр!

- Борешься с тем, чтобы компоненты кода сливались в единое целое?
- Тяжело вносить изменения с ростом кодовой базы?
- Чувствуешь, что твоя игра как гигантский клубок, в котором все переплетается друг с другом?
- Интересно, как применять шаблоны проектирования в играх?
- Слышал понятия "когерентность кэша" и "пул объектов", но не знаешь, как их применить, чтобы сделать игру быстрее?

Я здесь, чтобы помочь! **Шаблоны Игрового Программирования** — это коллекция игровых паттернов, которые **делают код чище, понятнее и быстрее**.

Этой книги мне не хватало, когда я начинал делать игры, и теперь я хочу, чтобы она была у тебя.

> Данная гит-книга является адаптацией перевода веб версии *Game Programming Patterns* by Robert Nystrom, и была создана для удобного чтения на русском языке в формате электронной книги. Автор данной гит-книги не является автором оригинала и перевода.
>
> **Верстка книги закончена**. Последние изменения 2016-03-12.
>
> На сайте автора http://gameprogrammingpatterns.com/ можно заказать печатную или электронную версию книги на английском языке.

Введение

В пятом классе у нас с другом был доступ к заброшенному классу с парочкой стареньких `TSR-80`. Вдохновил нас учитель, который подарил нам брошюру с простыми `BASIC` программами, чтобы нам было с чем возиться.

Привод для считывания аудиокассет для компьютеров был сломан, поэтому, если нам хотелось запустить какую-нибудь из программ, нам каждый раз приходилось вводить ее вручную. В основном мы предпочитали программы всего из нескольких строк наподобие:

```
10 PRINT "BOBBY IS RADICAL!!!"
20 GOTO 10
```

> Как будто, если компьютер выведет это сообщение достаточное количество раз, утверждение станет правдой.

И даже в этом случае нас подстерегали всевозможные трудности. Мы не имели представления о том как программировать, поэтому любая допущенная синтаксическая ошибка оказывалась для нас непреодолимой преградой. Если программа не работала, что случалось довольно часто, мы просто вводили ее заново.

В самом конце брошюры с кодами программ находился настоящий монстр — программа, которая занимала несколько полных страниц мелкого кода. Нам потребовалось немало времени прежде чем даже осмелиться за нее взяться. Но это все равно было неизбежно, потому что листинг был озаглавлен как *Тоннели и тролли*. Мы не имели ни малейшего представления о том, что она делает, но название явно намекало на то что это игра. А что может быть интереснее чем собственноручно запрограммированная игра? Нам так и не удалось ее запустить, а через год нам пришлось освободить этот класс (только гораздо позже когда я уже постиг основы `BASIC`, я понял что это была всего лишь программа-генератор персонажей для настольной игры, а не сама игра). Тем не менее жребий был брошен — с тех пор я решил для себя что буду разработчиком игр.

Когда я еще был подростком, у нас дома был `Macintosh` с `QuickBASIC` и немного позже с `THINK C`. Я проводил за попытками написания игровых программ все свои летние каникулы. Учиться самому было сложно и даже мучительно. Начинать писать игру всегда было довольно легко — скажем экран с картой или небольшой паззл. Но по

мере добавления новых возможностей программировать становилось все сложнее и сложнее. Как только у меня переставало получаться держать всю игру в голове целиком, все рушилось.

Поначалу задача была просто получить что-то рабочее. Затем становилось понятнее как писать программы, которые не осмыслишь в голове целиком. Вместо того чтобы просто читать книги наподобие *Как программировать на C++*, я начал искать книги о том как организовывать программный код.

> Не одно свое лето я провел за ловлей змеек и черепашек в болотах южной Луизианы. И, если бы там не было так жарко, вполне возможно я бы занимался герпентологией и писал бы другие книги.

Через несколько лет друг дал мне книгу *Паттерны проектирования: Приемы объектно-ориентированного проектирования*. Наконец-то! Это была книга, о которой я мечтал еще с подросткового возраста. Я прочел ее от корки до корки за один присест. У меня все еще были проблемы со своими программами, но мне было приятно видеть, что не только у меня одного возникают подобные сложности и есть люди которые нашли способ их преодолеть. Наконец-то у меня появилось ощущение, что я работаю не просто голыми руками, а у меня появились *инструменты*.

> Тогда мы с ним встретились впервые, и уже через пять минут после знакомства я сел на пол и провел несколько часов за чтением, полностью его игнорируя. Надеюсь, что с тех пор мои социальные навыки хоть немного улучшились.

В 2001-м я получил работу своей мечты: должность инженера-программиста в Electronic Arts. Я не мог дождаться момента, когда смогу увидеть как выглядят настоящие игры и как профессионалы собирают их целиком.

Как им удается создавать такие громадные игры как Madden Football, которые ни у одного человека точно не поместятся в голове? На что похожа их архитектура? Как отделены друг от друга физика и рендеринг? Или как код ИИ (*прим.: искусственный интеллект*) взаимодействует с анимацией? Как, имея единую кодовую базу, добиться ее работы на разных платформах?

Разбираться в исходном коде было одновременно унизительно и удивительно. В графической части, ИИ, анимации, визуальных эффекта был восхитительный код. У нас были люди, умеющие выжать последние такты из ЦПУ (*прим.: центральное процессорное устройство*) и высвободить их для более нужных вещей. Еще до обеда эти люди успевали делать такие вещи, про *возможность* которых я даже не подозревал.

Но вот *архитектура*, соединяющая все эти отличные компоненты вместе, зачастую хромала и делалась в последнюю очередь. Они настолько концентрировались на *функциональности*, что на организацию кода обращалось слишком мало внимания. Высокая связанность (coupling) отдельных модулей была обычным делом. Новый функционал прикручивался к старой кодовой базе как попало. Для моего разочарованного взгляда это выглядело, как работа множества программистов, которые, если и открывали когда-либо *Паттерны проектирования*, то не продвинулись в чтении дальше раздела про Синглтон (Singleton).

Конечно, не все было настолько плохо. Я ведь представлял себе игровых программистов как сидящих в башне из слоновой кости мудрецов, неделями дискутирующих о каждой мелочи в архитектуре игры. Реальность заключалась в том, что я видел код, написанный в условиях жесткого дедлайна для платформы, на которой каждый такт ЦПУ был на вес золота. Люди потрудились на славу и, как я понял впоследствии, они действительно во многом выбрали лучшее решение. Чем больше я тратил времени на работу с этим кодом, тем больше я понимал сколько брильянтов он в себе таит.

К сожалению, тут уместен именно термин "таит". Это были зарытые в коде алмазы, а прямо по ним топталось множество людей. Я наблюдал, как люди в мучениях переизобретали решения, прекрасные примеры которых уже находились в коде с которым они работали. Именно эта проблема и побудила меня на написание данной книги. Я откопал и отполировал для вас лучшие из найденных мной в различных играх шаблоны для того, чтобы вы могли тратить свое время на изобретение чего-то нового, а не на *пере*изобретение уже существующего.

Что есть в магазинах?

Сейчас продаются десятки книг, посвященных игровому программированию. Для чего понадобилось писать еще одну?

Большинство книг по программированию игр, которые я видел, делятся на две категории:

- **Узко-специализированные книги.** Эти книги концентрируются на чем-то одном и глубоко описывают только этот конкретный аспект. Они учат вас 3D графике, рендерингу в реальном времени, симуляции физики, искусственному интеллекту или работе со звуком. Многие разработчики игр вообще специализируются только в определенной области.

- **Книги обо всем.** В противоположность первым, эти книги пытаются охватить все части игрового движка. Они объединяют их вместе и показывают как собрать законченный движок, обычно 3D шутер от первого лица.

Мне нравятся книги из обеих категорий, но я считаю, что все они оставляют слишком много белых пятен. Книги, концентрирующиеся на отдельных аспектах, редко описывают, как данный кусок кода будет взаимодействовать с остальной игрой. Вы можете быть волшебником в области физики или рендеринга, но как правильно связать эти две подсистемы?

Вторая категория охватывает все, но зачастую подход получается очень монолитным и слишком жанрово-ориентированным. Сейчас, в период расцвета казуальных и мобильных игр, создаются игры самых различных жанров. Мы не можем больше просто продолжать клонировать Quake. Книги, показывающие вам создание движка под определенный жанр, не очень помогут вам если *у вас* совсем другая игра.

В отличие от других, моя книга построена по принципу *à la carte* (*франц.: на выбор*). Каждая из глав в этой книге представляет собой законченную идею, которую вы можете использовать в своем коде. Таким образом, вы получаете возможность смешивать их и использовать только то, что лучше всего подходит именно для вашей игры.

> Есть еще один хороший пример принципа *à la carte* — хорошо известная серия *Жемчужины игрового программирования* (Game Programming Gems).

Как все это связано с шаблонами проектирования

Каждая книга, имеющая в заголовке слово "шаблоны", так или иначе связана с классической книгой *Паттерны проектирования: Приемы объектно-ориентированного проектирования*, написанной Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Вличидесом (их еще часто называют "Банда четырех").

Называя свою книгу *Шаблоны игрового программирования*, я вовсе не имею в виду, что книга банды четырех неприменима в играх. Совсем наоборот: вторая часть этой книги как раз обозревает многие из шаблонов, впервые описанных в *Паттернах проектирования*, но с той точки зрения как они могут быть применены в программировании игр.

Более того, я считаю что эта книга будет полезна и для тех, кто не занимается разработкой игр. Использование описанных шаблонов будет уместным во многих не игровых приложениях. Я вообще мог бы назвать книгу *Еще больше шаблонов проектирования*, но на мой взгляд игровые примеры выглядят выразительнее. Или вам интереснее в очередной раз читать книгу о списках сотрудников и банковских счетах (*прим.: часто в качестве примеров применения шаблонов проектирования используют банки, клиентов, счета и пр.*)?

> Паттерны проектирования сами по себе также были написаны под впечатлением от другой книги. Идея создания языка шаблонов, описывающих ничем не ограниченные решения проблем пришла из книги *Язык шаблонов* (A Pattern Language) Кристофера Александера (и его соавторов Сары Ишикавы и Мюррея Силверстейна).

Их книга была посвящена архитектуре (в духе настоящей архитектуры, которая помогает строить дома, стены и т.д.), но они надеялись что и другие смогут использовать подобную структуру для описания решений в других областях. Паттерны проектирования банды четырех стараются применить тот же подход в программировании.

- Вместо того чтобы попытаться низвергнуть *Паттерны проектирования*, я рассматриваю свою книгу как их расширение. Пускай многие представленные здесь шаблоны будут полезны и в других типах программного обеспечения, я считаю что лучше всего они применимы именно к игровым задачам.
- Время и последовательность действий зачастую являются ключевыми частями игровой архитектуры. События должны происходить в правильной последовательности и в нужное время.
- Цикл разработки предельно сжат и множеству разработчиков необходимо иметь возможность быстро внедрять и итерационно менять широкий набор поведения, не наступая друг другу на ноги и не оставляя после себя следов по всей кодовой базе.
- После того как поведение определено, начинается взаимодействие. Монстры кусают героя, зелья смешиваются, а бомбы взрывают врагов и друзей. Все эти взаимодействия должны реализовываться без превращения кодовой базы в спутанный клубок шерсти.
- И наконец, для игр критична производительность. Игровые разработчики постоянно участвуют в гонке за первенство по максимально эффективному использования своей платформы. Небольшой трюк по сбережению нескольких тактов может отделять игру с наивысшим рейтингом и миллионными продажами от проблем с падением `fps` (кадров в секунду) и злыми рецензиями.

Как читать эту книгу

Вся книга разделена на три большие части. Первая — это введение и описание самой книги. Главу из этой части наряду со следующей вы сейчас и читаете.

Вторая часть — Обзор шаблонов проектирования рассматривает несколько шаблонов из книги банды четырех. Относительно каждого я высказываю собственное мнение и описываю его применимость в игровом программировании.

И, наконец, последняя часть — это сама соль данной книги. В ней описаны тринадцать новых шаблонов, которые я нахожу полезными при разработке игр. Она делится еще на четыре части: Последовательные шаблоны (sequencing), Поведенческие шаблоны (behavioral), Шаблоны уменьшения связности (decoupling) и Оптимизационные шаблоны (optimization).

Каждый шаблон внутри раздела описывается в виде стандартизированной структуры так, чтобы вам было проще использовать эту книгу для поиска того, что вам нужно:

- Секция **Задача** представляет собой краткое описание шаблона в терминах задачи, для решения которой он предназначен. Это первое, на что вы будете обращать внимание, когда будете искать в книге решение возникших у вас трудностей.
- Секция **Мотивация** описывает пример проблемы, которую позволяет решить шаблон. В отличие от алгоритма, без приложения к конкретной проблеме шаблон сам по себе не имеет формы. Изучать шаблоны без примеров — это как учиться печь хлеб, не упоминая того, как месить тесто. Этот раздел — тесто, которое мы будем печь дальше.
- Секция **Шаблон** описывает сущность шаблона из предшествующего примера. Если вам нужно формализованное описание шаблона — вы найдете его здесь. Также вам будет полезно заглянуть сюда, если вы уже знакомы с шаблоном, но подзабыли детали.
- Итак, шаблон у нас уже описан на конкретном примере. Но как теперь понять, что шаблон подходит именно для той проблемы, решением которой вы сейчас заняты? Секция **Когда использовать** содержит рекомендацию когда шаблон стоит использовать, а когда его лучше избегать. Секция **Имейте в виду** посвящена последствиям, которые вы получите применив шаблон в своем коде.
- Если вам, как и мне, нужен конкретный пример, как именно чего-либо добиться, тогда секция **Пример кода** для вас. Здесь подробно разбирается реализация шаблона, чтобы вы точно смогли понять как он работает.

- Шаблон — это собственно шаблон решения. Каждый раз, когда вы его используете, вы реализовываете его немного по-другому. Следующая секция — **Архитектурные решения**, раскрывает некоторые варианты применения шаблона.
- В конце находится еще одна коротенькая секция **Смотрите также**, в которой описывается связь шаблона с другими и с первоисточниками из *Паттернов проектирования*. Здесь вы получите более ясную картину о том, какое место занимает шаблон в экосистеме остальных шаблонов.

О примерах кода

Примеры кода в этой книге приводятся на `C++`, но это совсем не значит, что шаблоны можно реализовывать только на этом языке, или что `C++` лучше всех остальных. Для наших целей годится любой ООП язык.

`C++` я выбрал по нескольким причинам. Самая главная из которых заключается в том что на сегодняшний день это самый популярный для написания коммерческих игр язык. Для индустрии это *lingua franca*. Более того синтаксис `C`, на который опирается `C++` является также основой для `Java, C#, JavaScript` и многих других языков. Даже, если вы не знаете `C++`, приведенный в книге код будет понятен без приложения особых усилий.

Цель этой книги не в том, чтобы научить вас `C++`. Примеры наоборот максимально упрощены и даже не демонстрируют хороший стиль использования `C++`. Они предназначены для того, чтобы в них лучше читалась идея, а не просто читался хороший код.

В частности код не использует "модные" решения из `C++11` или более новых реализаций. В нем не используются стандартные библиотеки и довольно редко используются шаблоны. В результате C++ код получился "плохим", но я не считаю это недостатком, потому что таким образом он стал проще и его будет легче понять людям, использующим `C, Objective C, Java` и другие языки.

Чтобы не тратить место на код, который вы уже видели и который не относится к самому шаблону, он иногда будет приведен с сокращениями. В этом случае, пример кода будет сопровождаться пояснением о том, что делает отсутствующая в листинге часть кода.

Представим себе функцию, которая выполняет некоторые действия и возвращает результат. Описываемый шаблон зависит только от возвращаемого значения, а не от того что делает функция. В таком случае, пример кода будет выглядеть следующим

образом:

```
bool update()
{
    // Do work...
    return isDone();
}
```

Куда двигаться дальше

Шаблоны — это постоянно обновляющаяся и расширяющаяся часть программирования. Эта книга продолжает начинание банды четырех в области документирования и демонстрации найденных шаблонов и этот процесс не будет остановлен, когда высохнут чернила на этих страницах.

Именно вы ключевая часть этого процесса. По мере того, как вы будете изобретать новые шаблоны, улучшать (или опровергать!) уже существующие, вы будете приносить пользу всему сообществу разработчиков. Так, что если у вас есть свои суждения, дополнения и другие комментарии касательно написанного, прошу выходить на связь.

Архитектура, производительность и игры

Прежде, чем мы с головой нырнем в кучу шаблонов, я думаю обрисовать для вас общую картину того, что я думаю об архитектуре программного обеспечения в целом и игр в частности. Это поможет вам легче понять остальное содержимое книги. По крайней мере у вас появятся аргументы для непрекращающихся споров о том — хорошая вещь шаблоны или полный отстой.

> Обратите внимание, что я не настаиваю на том, чтобы вы приняли одну или другую сторону в этом противоборстве. Как у любого торговца оружием у меня есть, что предложить всем комбатантам.

Что такое архитектура программы?

Если вы прочтете эту книгу от корки до корки, вы не подчерпнете для себя новых знаний по алгебре, используемой в `3D` графике или вычислениями, используемыми в игровой физике. Вы не увидите реализацию альфа/бета отсечения (alpha/beta pruning) для вашего ИИ или симуляцию реверберации комнаты для звукового движка.

> Вау! Не параграф получился, а просто готовая реклама для книги.

Вместо этого мы уделим внимание коду *между* всем этим. Не столько написанию кода, сколько его *организации*. В каждой программе есть *своя* организация, даже если это просто "джем, размазанный по функции `main()`, чтобы просто посмотреть что получится". Поэтому, я считаю, что гораздо интереснее поговорить о том, как получается *хорошая* организация. Как отличить хорошую архитектуру от плохой?

Я обдумывал этот вопрос не меньше пяти лет. Конечно, как и у каждого из вас, у меня есть интуитивное представление о хорошей архитектуре. Мы все страдаем от настолько плохой кодовой базы, что лучшее, что с ней можно сделать это немного разгрести ее и продолжить страдать дальше.

> Давайте признаем, что большинство из нас хотя бы в какой-то степени за это отвечает.

У некоторых счастливчиков есть противоположный опыт: возможность работать с прекрасно спроектированным кодом. Такой тип кодовой базы ощущается как прекрасно меблированный отель с услужливыми консьержами, следящими за каждым вашим шагом. В чем же заключается разница между ними?

Что такое *хорошая* архитектура программы?

Для меня хорошее проектирование заключается в том, что когда мне нужно внести изменение, вся остальная часть программы как будто специально сделана так, чтобы мне было легко. Я могу добиться желаемого результата с помощью всего нескольких вызовов функций, делающихся так просто, что они не оставляют ни малейшей ряби на глади остального кода.

Звучит прекрасно, но не слишком конкретно. "Пиши такой код, чтобы его изменения не порождали рябь на глади воды". М-да.

Давайте немного углубимся в детали. Первая ключевая особенность архитектуры — это *приспособленность к изменениям*. Кому-то обязательно придется перерабатывать кодовую базу. Если никому больше к коду прикасаться не придется — по причине того, что он совершенен, закончен или наоборот настолько ужасен, что никто не решится открыть его в своем редакторе — проектирование не важно. Проектирование оценивается по простоте внесения изменений. Без изменений — это все равно что бегун, никогда не покидавший стартовой линии.

Как вносить изменения?

Прежде, чем изменять код и добавлять новый функционал или исправлять ошибки или вообще запускать по какой-либо причине свой редактор, вам нужно иметь представление о том, что делает уже существующий код. Конечно, вам не нужно понимать всю программу целиком, но нужно по крайней мере загрузить в свой мозг примата все связанные части.

> Странно об этом говорить, но фактически это оптическое распознавание образов (`OCR`).

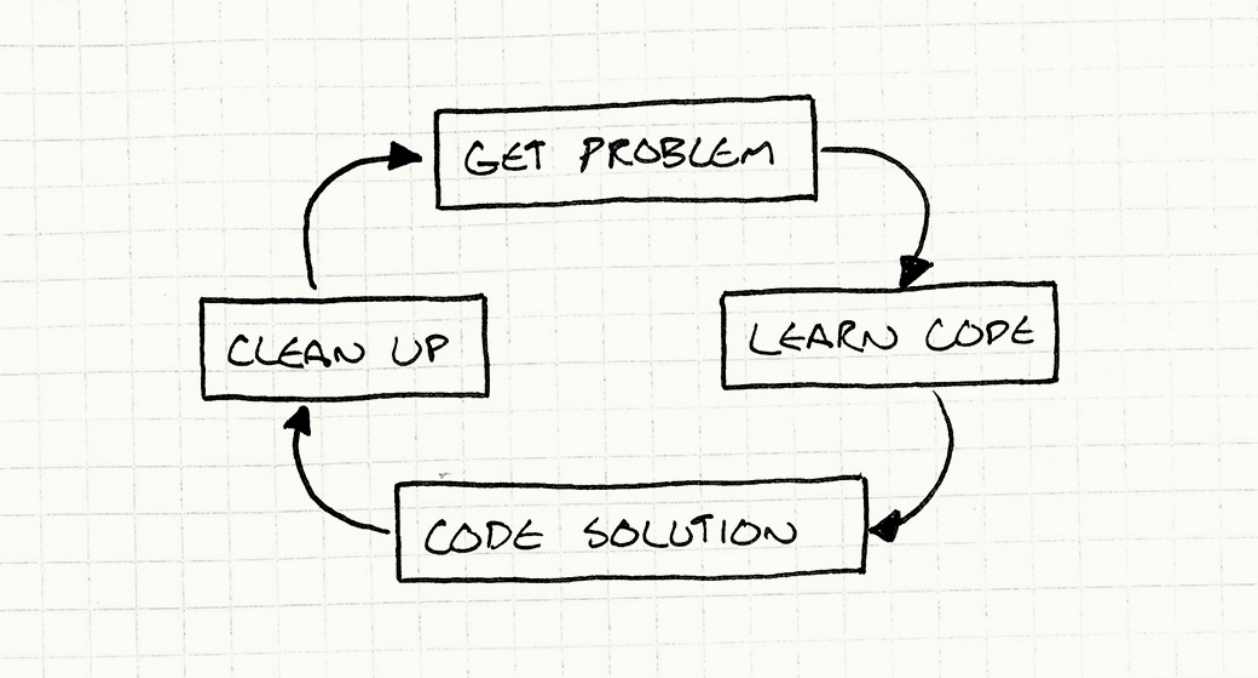
Мы все склонны недооценивать важность этого шага, но зачастую он оказывается самой затратной в плане времени частью программирования. Если вы думаете, что вас тормозит сброс данных из памяти на диск, задумайтесь лучше о скорости работы вашего обезьяньего мозга с оптическими нервами.

Как только вам удается загрузить всю нужную информацию в свою черепушку, вы немного думаете и выдаете решение. Конечно, вам приходится обдумывать некоторые детали, но, в целом, процесс достаточно прямолинеен. Как только вы понимаете проблему и часть кода, которую она затрагивает, сам процесс написания становится тривиальной задачей. Вы начинаете тыкать своими пальчиками в клавиатуру, пока на экране не появляются нужные вам черточки и на этом все, верно? А вот и нет! Прежде чем писать тесты и отсылать код на проверку (review), вам нужно кое-что подчистить.

Вы засунули в игру еще немного кода, но не хотите чтобы следующий кто будет работать с этим кодом спотыкался о следы вашей деятельности. Если это не совсем мелкое изменение, вам нужно предпринять некоторые меры по реорганизации кода, чтобы новый код естественным образом вписывался в уже существующий. Если вы это сделаете, следующий, кто будет после вас работать с этим кодом, даже и не поймет, когда и какая часть кода была написана.

> Я сказал "тесты"? А, да, сказал. Для многих частей кодовой базы игрового кода сложно написать юнит тесты, но некоторая его доля все таки отлично тестируется.
>
> Я не собираюсь лезть на трибуну, но все-таки призываю вас делать больше автоматизированных тестов. Неужели у вас нет более важных дел, чем тестировать одно и то же в ручном режиме?

Упрощенно диаграмма потоков в программировании выглядит следующим образом:



Меня даже несколько пугает, что цикл на диаграмме не имеет выхода.

Как нам может помочь уменьшение связности (decoupling)?

Хотя это и не очевидно, архитектура программы больше всего влияет на фазу изучения кода. Загрузка кода в нейроны настолько мучительно медленна, что стоит предпринимать любые стратегии для уменьшения его объема. В этой книге есть целый раздел, посвященный шаблонам *уменьшения связности* (decoupling) и большая часть книги *Паттерны проектирования* посвящена той же идее.

Уменьшение связности можно определять по-всякому, но лично я считаю два куска кода связанными, если я не могу понять как работает один кусок без понимания работы другого. Если уменьшить их связность (decouple), каждый из них можно будет рассматривать независимо. И это прекрасно, потому что, если к решаемой вами проблеме имеет отношение только *один* кусок кода, вам не придется загружать в свой обезьяний мозг второй кусок.

Для меня это является главной задачей архитектуры программы: **минимизация количества знаний, которые нужно поместить в свою черепушку прежде, чем двигаться дальше**.

Последующие этапы, конечно, тоже вступают в игру. Еще одно определение уменьшения связности состоит в том, что *изменение* одного куска кода не вызывает необходимость изменять другой. Нам обязательно придется *что-то* изменить, но чем меньше у нас связность, тем меньше частей игры это изменение затронет.

Какой ценой?

Звучит здорово, верно? Избавимся от связности и начнем кодить со скоростью ветра. Каждое изменение будет затрагивать всего один или несколько методов и вы будете порхать над кодовой базой, практически не отбрасывая на нее тень.

Именно благодаря этому чувству людей так привлекает абстрагирование, модульность, шаблоны проектирования и вообще архитектура программ. Программа с хорошей архитектурой превращает работу над собой в удовольствие, потому что все любят разработчиков с высокой производительностью. А хорошая архитектура дает *громадный* прирост производительности. Тяжело переоценить получаемый на выходе эффект.

Однако, как и все хорошее в жизни, ничего не дается бесплатно. Хорошая архитектура требует значительных усилий и дисциплины. Каждый раз, когда вы вносите изменения или добавляете новую функциональность, вам нужно прикладывать усилия к тому, чтобы эти изменения изящно интегрировались в остальную часть программы. Вам нужно приложить большие усилия к организации кода и *поддерживать* эту организованность на протяжении тысяч маленьких изменений, которые предстоит совершить на протяжении всего цикла разработки.

> Второй этап этого процесса — поддержка архитектуры — требует особого внимания. Я видел множество примеров того как программисты начинали за здравие с блестящим кодом и заканчивали за упокой, когда насыщали код тысячами хаков "чуть подправить здесь и готово". И так раз за разом.
>
> Также, как в садоводстве, здесь не достаточно просто посадить новые растения. Нужно еще бороться с сорняками и подстригать деревья.

Вам нужно решить связность между каким частями программы вы хотите уменьшить и добавить необходимое абстрагирование. Кроме того, вам нужно предусмотреть пути расширения функциональности, чтобы было проще работать в будущем.

Люди приходят от этого в восторг. Они представляют себе как разработчики будущего (или они сами в будущем) открывают кодовую базу и видят какая она вся понятная, мощная и так и просит себя расширить. Они представляют себе Один Игровой Движок Который Всем Повелевает.

Но вот тут-то и кроется сложность. Добавляете ли вы новый уровень абстракции или предусматриваете место для расширения, вы должны *предполагать*, что эта гибкость понадобится вам в будущем. Вы добавляете код и усложняете игру, тратя при этом время на разработку, отладку и поддержку.

Эти затраты с лихвой окупятся в том случае, если вы угадали и будете изменять код в этом направлении в дальнейшем. К сожалению, предсказывать будущее довольно *сложно* и если модульность вам в дальнейшем не понадобится, вскоре она начнет вам активно вредить. В конце концов, вам просто придется работать с более громоздким кодом.

> Какие-то умники даже придумали термин "YAGNI" — Вам это не понадобится (You aren't gonna need it) — специальную мантру, которая поможет вам бороться со злоупотреблениями в предположениях о том, что может понадобиться вам в будущем.

Когда люди проявляют в этом чрезмерные усилия, в результате получается кодовая база, архитектура которой все больше выходит из-под контроля. У вас повсюду будут сплошные интерфейсы и абстракции. Системы плагинов, абстрактные базовые классы, изобилие виртуальных методов и куча точек для расширения.

Потребуется вечность, чтобы прорваться через завалы всего этого богатства и добраться до настоящего кода, который хоть что-то делает. Конечно, если вам нужно внести какие-то изменения, у вас скорее всего найдется интерфейс, который вам поможет, но вы еще попробуйте его найти. В теории такое уменьшения связности

означает, что вам нужно понимать меньше кода для того, чтобы его расширять, но само по себе нагромождение абстракций закончится тем, что кеш вашего мозга просто переполнится.

Кодовые базы такого типа только *отталкивают* людей от работы над архитектурой программ и от шаблонов проектирования в частности. Зарыться в код довольно просто, только не нужно забывать о том, что мы все-таки занимаемся созданием *игры*. Сладкие песни сирен про расширяемость поймали в свои сети множество игровых разработчиков, которые годами занимаются работой над "движком", даже не понимая какой *конкретно* движок им нужен.

Производительность и скорость

Довольно часто увлечение архитектурой программы и абстракциями критикуют, особенно в игровом программировании за то, что это вредит производительности игры. Многие шаблоны, делающие ваш код более гибким используют виртуальную диспетчеризацию, интерфейсы, указатели, сообщения и другие механизмы, за которые приходится платить производительностью работы приложения.

> Еще один интересный контр-пример — это шаблоны в `C++`. Метапрограммирование на основе шаблонов зачастую позволяет организовать абстрактный интерфейс без ущерба для производительности.
>
> Гибкость здесь довольно высока. Когда вы пишете код вызова конкретного метода в *некотором* классе, вы фиксируете этот класс во время написания — жестко вшиваете в код, какой класс вы вызываете. Когда же вы используете виртуальные методы или интерфейсы, вызываемый класс становится неизвестным до момента *выполнения*. Такой подход достаточно гибкий, но требует накладных расходов в плане производительности.
>
> Метапрограммирование на основе шаблонов представляет собой нечто среднее. Здесь вы принимаете решение о том какой класс вызывать на *этапе компиляции*, когда создается экземпляр шаблона.

Такая критика имеет все основания. Зачастую архитектура программы предназначена для того, чтобы сделать ее более гибкой. Для того, чтобы ее было легче изменять. Это значит, что при кодинге вы допускаете меньше допущений. Вы используете интерфейсы для того, чтобы можно было работать с *любыми* классами их реализующими, вместо того, чтобы ограничиться тем, что необходимо сегодня. Вы используете шаблоны наблюдатель GoG (observer) и сообщения (messaging) для того, чтобы между собой могли легко общаться не только два куска кода сегодня, но и три и четыре в будущем.

Тем не менее производительность предполагает допущения. Искусство оптимизации основывается на введении конкретных ограничений. Можем ли мы предположить, что у нас никогда не будет больше `256` противников? Отлично, значит для `ID` каждого из них достаточно всего одного байта. Будем ли мы вызывать здесь метод конкретного класса? Отлично, значит его можно вызвать статически или использовать `inline` вызов. Принадлежат ли все сущности к одному классу? Замечательно, значит мы можем организовать их в виде непрерывного массива (contiguous array).

При этом не подразумевается вообще никакой гибкости! Мы можем быстро изменять игру, а для того чтобы сделать хорошую игру жизненно важна именно скорость *разработки*. Никто, даже Уилл Райт не способен создать сбалансированный игровой дизайн на бумаге. Необходимы итерации и эксперименты.

Чем быстрее вы сможете попробовать идею и увидеть как она играется, тем больше вы сможете попробовать и с большей долей вероятности придумаете что-то действительно стоящее. Даже после того, как правильная игровая механика найдена, потребуется еще куча времени на ее тюнинг. Даже небольшой дисбаланс способен похоронить всю игру. Ответ здесь простой. Делая свою программу боле гибкой, вы ускоряете процесс прототипирования, но жертвуете производительностью. С другой стороны, любая оптимизация кода делает его менее гибким.

Мой собственный опыт показывает, что проще сделать интересную игру быстрой, чем сделать быструю игру интересной. Компромиссным решением здесь может быть принцип стараться делать код гибким до тех пор, пока дизайн игры более менее не устаканится, а затем избавиться от некоторых уровней абстракции в целях увеличения производительности.

Чем хорош плохой код

Теперь мы переходим к следующему ключевому вопросу относительно стилей кодинга. Большая часть этой книги посвящена тому, как писать легко поддерживаемый чистый код, так что можно считать, что я сторонник "правильного" подхода. Однако неряшливый метод кодинга тоже не стоит забывать.

Написание кода с хорошей архитектурой требует больше усилий и выливается в трату большего количества времени. Более того, *поддержание* кода с хорошей архитектурой на протяжении всей жизни проекта также требует много усилий. Вы должны обращаться со своей кодовой базой также, как порядочный турист, который, покидая стоянку, старается оставить ее в лучшем состоянии, чем нашел.

Это замечательно в случае, если вам придется жить и работать с этим кодом на протяжении долгого времени. Но, как было сказано выше, поддержание игрового дизайна требует проведения множества исследований и экспериментов. Особенно на ранних этапах, когда вы *точно знаете*, что большую часть кода вы просто выкините.

Если вы просто хотите найти наиболее удачное для геймплея решение, следить за красотой архитектуры бессмысленно, потому что так вы потеряете больше времени прежде, чем увидите результат на экране и получите обратную связь. Если то, что было сделано, не заработает, какой вам будет прок от того, что вы потратили столько времени на элегантный код, который вы все равно выбрасываете.

Прототипирование — это просто лепка кода в кучу, достаточно функционального для того, чтобы геймдизайнер мог понять насколько идея хороша — т.е. совершенно легитимная в программировании практика. Здесь главное не забывать о самом важном принципе прототипирования. Если вы пишите код для выкидывания, вы *обязаны* его выкинуть. К сожалению, я раз за разом вижу менеджеров, пренебрегающих этим правилом.

> Босс: "Слушай, есть идея которую нужно опопробовать. Сделай прототип по-быстренькому. Можешь сильно не стараться. Сколько времени тебе нужно?"
>
> Разработчик: "Ну, если совсем по-быстрому, ничего не тестировать и не документировать и с кучей багов, то можно написать временный код за несколько дней."
>
> Босс: "Отлично!"

Через пару дней...

> Босс: "Слушай, прототип классный. Можешь за несколько дней его подлатать и мы возьмем его за основу?"

Вам нужно быть уверенным, что люди использующие код, написанный на выброс понимали бы, что даже если он выглядит рабочим, его *невозможно* поддерживать и его *обязательно* нужно переписать. Если есть хотя бы *малейшая возможность* того, что его придется оставить, вам обязательно нужно, несмотря ни на что, писать его уже правильно.

> Хорошим трюком можно признать привычку написания прототипа на другом языке программирования. Не том, на котором будет писаться игра. В этом случае вам обязательно придется переписать его прежде, чем он попадет в настоящую игру.

Подведем итоги

Как мы увидили, в игру вступает несколько сил:

1. Нам нужна хорошая архитектура для того, чтобы легче было понимать код во время цикла разработки проекта.
2. Нам нужна хорошая производительность.
3. Нам нужно иметь возможность быстро внедрять новую функциональность.

> Я нахожу даже забавным тот факт, что в любом случае нам нужно думать о скорости: скорости разработки в долгосрочной перспективе, скорости работы игры, скорости разработки в короткосрочной перспективе.

Между этими целями наблюдаются некоторые противоречия. Хорошая архитектура ускоряет производительность труда в длительной перспективе, но ее поддержание требует затрат дополнительных усилий после каждого изменения.

Быстрее всего написанная реализация совсем не обязательно самая быстрая в плане производительности. Наоборот, оптимизация требует дополнительного времени разработки. И как только она выполнена, кодовая база сразу начинает костенеть: высокооптимизированный код крайне негибок и его очень сложно менять.

Всегда существует соблазн закончить сегодняшнюю работу сегодня, а завтра заняться чем-то еще. Но, если добавлять функциональность так быстро, насколько это возможно, кодовая база быстро превратится в месиво хаков, багов и противоречий, которые замедлят нашу продуктивность в будущем.

Здесь нет простого ответа, возможны лишь компромиссы. Судя по почте, которую я получаю, многих людей это просто обескураживает. Особенно новичков, которые просто хотят сделать игру. Согласитесь, звучит пугающе, когда слышишь: "Правильного ответа не существует, есть только разные варианты неправильных".

Но на мой взгляд это просто замечательно! Посмотрите на любую другую область человеческой деятельности и, скорее всего, вы увидите в основе набор непреложных истин. В конце концов, если бы существовал простой ответ, все бы только так и делали. Область, в которой мастером можно стать за неделю просто скучна. Вы никогда не услышите потрясающих карьерных историй от копателя канав.

> А может быть и можно. Я не особо задумывался об этой аналогии. Всегда найдутся энтузиасты, которые копают так глубоко, насколько это только возможно. Даже целые субкультуры организуют. Кто я такой чтобы об этом судить?

На мой взгляд, все это очень похоже на сами игры. В игре наподобие шахмат никогда нельзя стать непревзойденным мастером, потому что все части игры отлично сбалансированы. Это значит, что вы можете потратить целую жизнь на перебор всех

возможных стратегий. Игра с плохим дизайном наоборот очень быстро скатывается к одной выигрышной тактике, которой начинает придерживаться игрок, пока она ему не надоест.

Упрощение

Гораздо позднее я открыл для себя еще один метод, смягчающий эти ограничения — *упрощение*. Сейчас я в своем коде стараюсь писать чистое, максимально прямолинейное решение проблемы. Это такой тип кода, после прочтения которого у вас не остается ни малейших сомнений относительно того, что именно он делает и вы не можете представить никакого другого решения.

Я стараюсь выбирать правильные структуры данных и алгоритмы (именно в такой очередности) и в дальнейшем от них отталкиваюсь. При этом я заметил, что чем проще решение — тем меньше кода на выходе. А это значит, что и свою голову мне приходится забивать меньшим количеством кода, когда приходит время его менять.

Зачастую код получается быстрым, потому что не требует слишком больших накладных расходов и сам объем кода невелик. (Конечно, это вовсе не правило. Даже в совсем маленький участок кода можно поместить кучу циклов и рекурсий.)

Однако, обратите внимание, что я не говорю, что *написание* простого кода требует мало времени. Вы могли бы так предположить, потому что в результате кода будет совсем немного, однако хорошее решение — это не просто разрастание кода — это его *дистиллят*.

> Блез Паскаль закончил свое знаменитое письмо следующими словами "Я хотел написать письмо покороче, но мне не хватило времени".
>
> Еще одну интересную мысль можно найти у Антуана де Сент-Экзюпери "Совершенство достижимо, но не тогда, когда уже нечего добавить, а когда уже нечего убавить".
>
> И еще один пример ближе к телу. Каждый раз, когда я просматривал главы этой книги, они становились все короче и короче. Некоторые главы потеряли до 20% объема.

Мы редко сталкиваемся с элегантными проблемами. Вместо этого у нас обычно есть набор вариантов использования. Нам нужно заставить X делать Y когда выполняется условие Z, а W когда выполняется A и т.д. Другими словами все, что у нас есть — это длинный список примеров поведения.

Решение, требующее меньше всего мыслительных усилий — это просто закодировать все эти условия по отдельности. Если вы посмотрите на новичков — они зачастую именно так и делают: они разбиваю решение на большое дерево отдельных случаев.

Никакой элегантности здесь конечно нет и код, написанный в таком стиле имеет тенденцию падать при входных данных, хоть немного отличающихся от тех, на которые рассчитывал программист. Когда мы говорим об элегантном решении, мы чаще всего имеем в виду *обобщенное* решение: небольшой логический блок, который покрывает большую область вариантов использования.

Поиск такого блока похож на подбор нужного шаблона или разгадывание паззла. Требуются большие усилия, чтобы увидеть сквозь разрозненное множество примеров вариантов использования скрытую закономерность, объединяющую их все. И какое же это замечательное чувство, когда разгадка находится.

Просто смиритесь

Большинство людей предпочитают пропускать вступление, так что я поздравляю вас с тем, что вы его одолели. Мне нечем отблагодарить вас за это, кроме нескольких советов, которые я надеюсь будут вам полезны:

- Абстрагирование и уменьшение связности позволяет вашей программе эволюционировать быстрее, но не увлекайтесь этим если не уверены в том, что данный код требует гибкости.
- Во время разработки помните и про дизайн и про производительность, только откладывайте по возможности всяческие тонкие оптимизации на самый конец цикла разработки.

  > Поверьте мне, два месяца до даты релиза — это не тот срок, когда нужно, наконец, приступать к решению проблемы "игра работает, но выдает только 1 FPS".

- Старайтесь исследовать поле дизайнерских решений быстрее, но не настолько, чтобы оставлять после себя месиво в коде. В конце концов, вам ведь еще с этим кодом жить.
- Если собираетесь выбросить код, не тратьте много времени на его совершенствование. Рок звезды ведь именно потому так часто устраивают погромы в номерах отелей, что знают о том, что на следующий день оттуда уедут.
- И самое главное — **если хотите сделать что-то интересное, получайте удовольствие от процесса**.

Обзор шаблонов проектирования

Книге *Паттерны проектирования: Приемы объектно-ориентированного проектирования* уже лет двадцать. Если вы прямо сейчас не читаете книгу через мое плечо, есть вероятность, что, когда вы будете читать эту книгу, *Паттерны проектирования* будут такими старыми, что за это будет стоить выпить. Для такой быстро меняющейся индустрии как программирование — это практически вечность. Неугасающая популярность книги свидетельствует о том, что проектирование — это значительно меньше подверженная влиянию времени вещь, чем большинство технологий, языков и методологий.

И, хотя я до сих пор считаю *Паттерны проектирования* актуальными, за прошедшие десятилетия мы все-таки кое-чему научились. В этом разделе мы пройдемся по самым удачным из оригинальных шаблонов, задокументированных "Бандой Четырех". Смею надеяться, что мне есть, что добавить полезного о каждом из них.

Некоторые шаблоны я считаю слишком переоцененными (Синглтон (Singleton)), а другие наоборот недооцененными (Команда (Command)). Еще парочку я сюда включил, потому что они особенно удачно подходят для применения в играх (Приспособленец (Flyweight) и Наблюдатель (Observer)). И, наконец, на мой взгляд иногда просто интересно посмотреть, как шаблоны путают с другими областями программирования (Прототип (Protorype) и Состояние (State)).

Шаблоны

- Команда (Command)
- Приспособленец (Flyweight)
- Наблюдатель (Observer)
- Прототип (Protorype)
- Синглтон (Singleton)
- Состояние (State)

Команда (Command)

Команда — это один из моих любимых шаблонов. В большинстве программ, которые я писал — и в играх, и в других программах так или иначе находилось применение для этого шаблона. Не раз с его помощью мне удавалось распутать довольно корявый код. Для такого важного шаблона банда четырех приготовила ожидаемо заумное описание:

> Инкапсуляция запроса в внутри объекта, позволяющая пользователю параметризировать клиенты с помощью различных запросов, организовывать в очереди или регистрировать запросы или организовывать поддержку отменяемых операций.

Можно ли согласиться с таким ужасным приговором? Прежде всего он искажает все, что данная метафора способна предложить. За пределами странного мира программ, где слова могут означать что угодно, слово "клиент" означает *личность* — кого-то, с кем вы имеете дело. Причем обычно других людей не принято "параметризировать".

Дальше идет просто перечисление того, где можно применять шаблон. Не слишком очевидно, конечно если ваш вариант использования прямо не указан в этом перечне. *Мое* краткое определение данного шаблона гораздо проще:

Команда — это материализация вызова метода.

> Материализовать (по-английски Reify) происходит от латинского "res", что значит "вещь (thing)" с английским суффиксом "-fy". Т.е. можно было бы использовать слово "овеществить (thingfy)", что, честно, было бы более забавным термином.

Конечно, "краткое" не всегда означает "достаточное", так что толку от этого по прежнему мало. Давайте немного углубимся в суть дела. "Материализовать", чтобы вы знали, означает буквально "сделать реальным". Еще один термин материализации — это объявление чего либо "объектом первого класса".

> *Система отражений* (Reflection systems) в некоторых языках позволяют вам работать с типами в программе императивно. Вы можете создать объект, представляющий собой класс другого объекта и с его помощью понимать, что этот объект может делать. Т.е. мы получаем *овеществленную систему типизации*.

Оба термина означают, что мы возьмем некую *концепцию* и превратим ее в *набор данных* — объект, который можно поместить в переменную, передать в функцию и т.д. Таким образом если мы говорим, что команда — это "материализация вызова метода" — это означает что вызов метода оборачивается некоторым объектом.

Есть много разных названий: "обратный вызов", "функция первого класса", "указатель на функцию", "замыкание (closure)", "частично примененная функция (partially applied function)", в зависимости от языка, к которому вы привыкли. Однако все это одного поля ягоды. Банда четырех немного дальше уточняет:

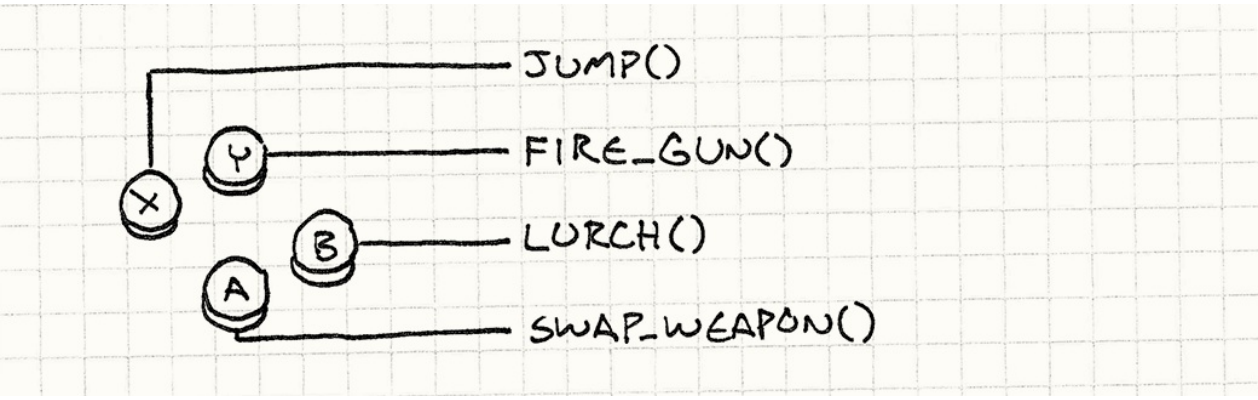
Команда — это объектно- ориентированная замена обратного вызова.

Это уже гораздо полезнее для осмысленного выбора шаблона.

Но пока это все абстрактно и слишком туманно. Я хочу начать главу с чего-то конкретного, и сейчас я это сделаю. Чтобы это сделать, мне понадобится пример, в котором применение команды будет смотреться идеально.

Настройка ввода

Внутри каждой игры есть код, отвечающий за считывание пользовательского ввода — нажатия на кнопки, клавиатурные события, нажатия мыши и т.д. Этот код обрабатывает ввод и преобразует его в соответствующие действия в игре:



Самая примитивная реализация выглядит следующим образом:

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) jump();
  else if (isPressed(BUTTON_Y)) fireGun();
  else if (isPressed(BUTTON_A)) swapWeapon();
  else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

> Совет профессионала: Не нажимайте B слишком часто.

Такая функция обычно вызывается на каждом кадре внутри игрового цикла (Game Loop). Думаю, вам понятно что она делает. Здесь мы видим жесткую привязку пользовательского ввода с действиями в игре. Однако многие игры позволяют пользователям *настраивать* какие кнопки за что отвечают.

Для того, чтобы это стало возможным нам нужно преобразовать прямые вызовы `jump()` и `fireGun()` в нечто, что мы сможем свободно менять местами. "Менять местами" звучит как присвоение значений переменным, поэтому нам нужен объект, который будет представлять игровое действие. И тут в дело вступает шаблон *Команда*.

Для начала определим базовый класс, представляющий запускаемую игровую команду:

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
};
```

> Когда у вас появляется интерфейс с единственным методом, который ничего не возвращает — с большой долей вероятности можно предположить что это шаблон *Команда*.

Теперь создадим дочерние классы для каждой из различных игровых команд:

```
class JumpCommand : public Command
{
public:
  virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
  virtual void execute() { fireGun(); }
};

// Ну вы поняли...
```

В нашем обработчике ввода мы будем хранить указатели на команду для каждой кнопки:

```
class InputHandler
{
public:
  void handleInput();

// Методы для привязки команд...

private:
  Command* buttonX_;
  Command* buttonY_;
  Command* buttonA_;
  Command* buttonB_;
};
```

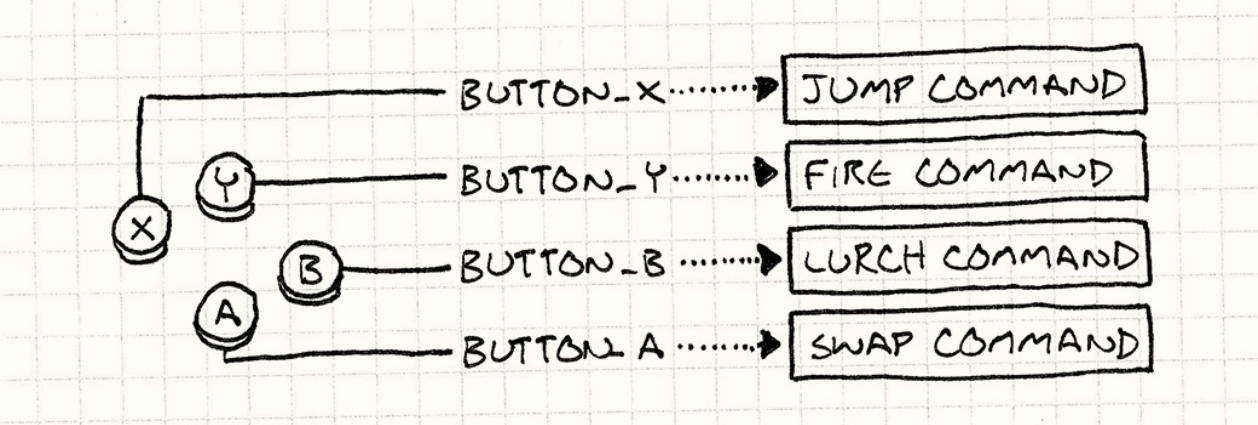
Теперь обработка ввода сводится к делегированию такого вида:

```
void InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) buttonX_->execute();
  else if (isPressed(BUTTON_Y)) buttonY_->execute();
  else if (isPressed(BUTTON_A)) buttonA_->execute();
  else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

Обратите внимание, что проверки на `NULL` здесь нет. Подразумевается, что к каждой кнопке привязана определенная команда.

Если мы и в сам деле хотим иметь кнопку, которая ничего не делает, то нам все равно не нужно добавлять проверку на `NULL`. Вместо этого нам нужно реализовать команду, метод `execute()` которой ничего не делает. И потом вместо установки обработчика кнопки в `NULL` мы будем подставлять указатель на этот объект. Такой шаблон носит название Нулевой объект (Null Object).

Там, где раньше пользовательский ввод напрямую вызывал функции, теперь у нас появился промежуточный слой косвенности:



В этом и заключается сущность шаблона *Команда*. Если вы уже оценили его по-достоинству, оставшуюся часть главы можете рассматривать как бонус.

Указания для актеров

Классы команд, которые мы только что определили, отлично работают для примера выше, но их возможности все-таки сильно ограничены. Проблема в том, что мы предполагаем что у нас уже есть готовые функции высокого уровня `jump()`, `fireGun()` и т.д., которые сами знают, как найти персонаж игрока и заставить его плясать под нашу дудку.

Такое предположение значительно снижает применимость наших команд. Получается что команда `JumpCommand` — это единственное, что способно заставить прыгать только нашего игрока. Давайте избавимся от этого ограничения. Вместо того, чтобы запускать функцию, которая будет сама искать объект для воздействия, мы сами передадим ей объект, которым хотим управлять:

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute(GameActor& actor) = 0;
};
```

Здесь в качестве `GameActor` выступает наш класс "игровой объект", представляющий игрока в игровом мире. Мы передаем его в `execute()` и таким образом изолированная команда получает возможность вызвать метод выбранного нами актера:

```
class JumpCommand : public Command
{
public:
  virtual void execute(GameActor& actor) {
    actor.jump();
  }
};
```

Теперь мы можем использовать этот единственный класс, чтобы заставить прыгать любого в нашей игре. Правда у нас пока еще нет прослойки между обработчиком ввода и командой, которая собственно получает команду и применяет ее к нужному объекту. Для начала мы изменим `handleInput()` таким образом, чтобы она возвращала команду:

```
Command* InputHandler::handleInput()
{
  if (isPressed(BUTTON_X)) return buttonX_;
  if (isPressed(BUTTON_Y)) return buttonY_;
  if (isPressed(BUTTON_A)) return buttonA_;
  if (isPressed(BUTTON_B)) return buttonB_;

  // Если ничего не передано, то ничего и не делаем.
  return NULL;
}
```

Функция не может выполнить команду немедленно, потому что не знает какого актера ей передать. Зато мы можем воспользоваться тем преимуществом команды, что это материализованный вызов — мы можем *отложить* выполнение.

Теперь нам нужен код, который получит команду и передаст в нее актера, представляющего игрока. Нечто наподобие:

```
Command* command = inputHandler.handleInput();
if (command)
{
  command->execute(actor);
}
```

Предполагая, что `actor` указывает на персонажа игрока, мы получаем корректную реализацию того, чего добивались, т.е. мы вернулись к тому же поведению, что и в самом первом примере. Добавив слой косвенности между командой и актером, который ее выполняет, мы получили еще одну приятную способность: *теперь мы можем позволить игроку управлять любым актером в игре, просто подменяя актера, к которому применяется команда*.

На практике такая возможность используется не слишком часто. Но похожий вариант использования все равно часто всплывает. До сих пор мы упоминали только управляемых игроком персонажей. А что насчет остальных? Тех, которые управляются игровым ИИ. Мы можем использовать тот же самый шаблон в качестве интерфейса между движком ИИ и актерами: код ИИ просто будет вызывать объекты `Command`.

Уменьшение связности в данном случае, когда ИИ выбирает команду, а код актера ее выполняет, дает нам дополнительную гибкость. Мы получаем возможность использовать разные модули ИИ для разных актеров. Или же мы можем их смешивать и выстраивать ИИ для разных стилей поведения. Вам нужен более агрессивный противник? Просто подключите более агрессивный ИИ, чтобы им управлять. На самом деле мы можем даже передать на попечение ИИ персонаж *игрока*, что довольно удобно для демо-режима, когда игра работает на автопилоте.

Делая команду, управляющую актером объектом первого класса, мы избавляемся от жесткой привязки прямого вызова методов. Вместо этого можете думать об этом, как об очереди или потоке команд.

> Более подробно о такой очередности можно почитать в Очереди событий (Event Queue).

> И почему, интересно, мне захотелось изобразить для вам такой "поток"? И почему он имеет форму трубки?

Некоторый код (обработчик ввода или ИИ) генерирует команды и добавляет их в поток. Другой код (диспетчер или сам актер) поглощает команды и вызывает их. Поместив такую очередь в середину, мы уменьшили связность между производителем с одной стороны и потребителем с другой.

> Если мы сделаем такую команду *сериализуемой*, мы сможем пересылать их очередность по сети. Сможем взять пользовательский ввод, передать его по сети и воспроизвести на другой машине. Именно такой механизм лежит в основе многопользовательских игр.

Отмена и повтор

Последний пример — это самый известный способ применения данного шаблона. Если объект команда может *выполнять* действия, значит мы уже сделали маленький шаг к тому, чтобы получить возможность их *отменять*. Отмену можно встретить в некоторых стратегических играх, когда вы имеете возможность отменить последнее не понравившееся вам действие. Такая функциональность *обязательно присутствует* и в инструментах, которые используются для *создания* игр. Лучший способ заставить гейм-дизайнера ненавидеть вас — это выдать ему инструментарий, в котором нельзя отменить того, что он наворотил своими толстенькими пальчиками.

> Это я могу утверждать на собственном опыте.

Без шаблона *Команда*, реализация отмены довольно сложна. С ним — пара пустяков. Для примера предположим, что мы разрабатываем однопользовательскую пошаговую игру и хотим разрешить игроку отменять ходы, чтобы он мог больше сосредоточиться на стратегии, а не на угадывании.

Мы уже оценили удобство использования команды для абстрагирования пользовательского ввода, поэтому каждый ход игрока у нас уже инкапсулирован в команду. Например, движение юнита может выглядеть следующим образом:

```
class MoveUnitCommand : public Command
{
public:
  MoveUnitCommand(Unit* unit, int x, int y)
  : unit_(unit), x_(x), y_(y)
  {}

  virtual void execute() {
    unit_->moveTo(x_, y_);
  }

private:
  Unit* unit_;
  int x_, y_;
};
```

Обратите внимание на небольшое отличие от нашей предыдущей команды. В предыдущем примере мы хотели *абстрагировать* команду от актера, на которого она действует. В этом же случае мы специально хотим *привязать* ее к актеру, которого она двигает. Экземпляр этой команды — это не обобщенная операция "перемещающая что либо", которую можно применить в самом разном контексте, а конкретный отдельный шаг в очереди шагов игры.

Это показывает насколько вариативным может быть применение данного шаблона. В некоторых случаях, как наша первая парочка примеров, команда — это многоразовый (reusable) объект, представляющий *действие, которое можно выполнить*. Наш первый пример обработки ввода сводился к единственному вызову метода `execute()` по нажатию нужной кнопки.

А вот более специфическая команда. Она описывает вещи, которые можно сделать в определенный момент. Это значит, что код обработчика ввода будет *создаваться* каждый раз, когда игрок решит двинутся. Выглядеть это будет следующим образом:

```
Command* handleInput()
{
  // Выбираем юнит...
  Unit* unit = getSelectedUnit();

  if (isPressed(BUTTON_UP)) {
    // Перемещаем юнит на единицу вверх.
    int destY = unit->y() - 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  if (isPressed(BUTTON_DOWN)) {
    // Перемещаем юнит на единицу вниз.
    int destY = unit->y() + 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  // Другие шаги...

  return NULL;
}
```

> Конечно, в языках без сборщика мусора, наподобие `C++`, это означает, что выполняющий команды код также должен заботиться и об освобождении памяти, занимаемой командами.

Тот факт, что команды получаются одноразовыми дает нам определенные преимущества. Чтобы сделать команды отменяемыми, мы определим еще одну операцию, которую должен реализовывать каждый класс команд:

```
class Command
{
public:
  virtual ~Command() {}
  virtual void execute() = 0;
  virtual void undo() = 0;
};
```

Метод `undo()` возвращает игру в то состояние, в котором она была до выполнения соответствующего метода `execute()`. Вот наша последняя команда, дополненная поддержкой отмены:

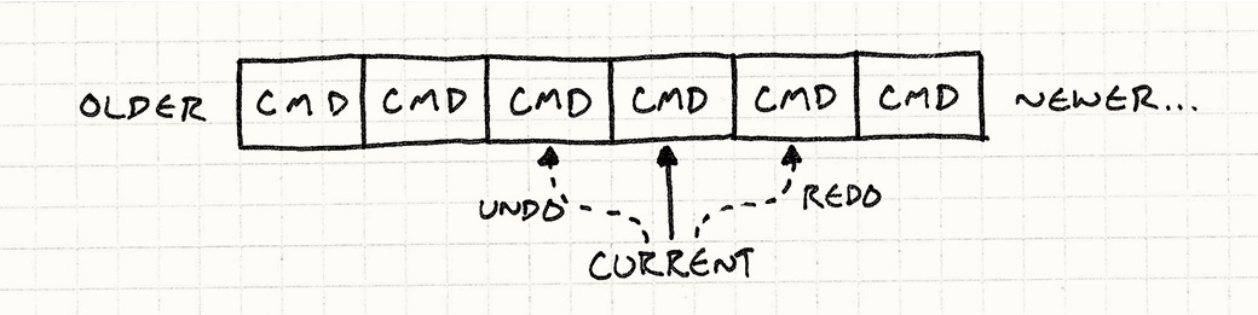
```
class MoveUnitCommand : public Command
{
public:
  MoveUnitCommand(Unit* unit, int x, int y)
  : unit_(unit), xBefore_(0), yBefore_(0),
  x_(x), y_(y)
  {}

  virtual void execute() {
    // Запоминаем позицию юнита перед ходом
    // чтобы потом ее восстановить.
    xBefore_ = unit_->x();
    yBefore_ = unit_->y();

    unit_->moveTo(x_, y_);
  }

  virtual void undo() {
    unit_->moveTo(xBefore_, yBefore_);
  }

private:
  Unit* unit_;
  int xBefore_, yBefore_;
  int x_, y_;
};
```

Обратите внимание, что мы добавили в класс больше состояний. После того, как мы переместили юнит, ему неоткуда узнать, где он был раньше. Чтобы иметь возможность отменить перемещение, нам нужно запомнить предыдущую позицию самостоятельно. Вот для этого мы и добавляем в команду `xBefore_` и `yBefore_`.

> Похоже здесь хорошо смотрелся бы шаблон Хранитель (Memento pattern) GoF, но мне не удалось заставить его работать эффективно. Так как целью команды является изменение только небольшой части состояния объекта, сохранение и всех остальных данных тоже — напрасная трата памяти. Дешевле просто вручную хранить только необходимые биты, которые вы меняете.
>
> Еще один вариант — Постоянные структуры данных (Persistent data structures). При этом при каждом изменении объекта возвращается новый объект, а старый остается неизменным. В разумной реализации эти новые объекты разделяют данные со старыми и в результате, такой подход гораздо разумнее, чем клонирование объектов целиком.
>
> При их использовании каждая команда хранит ссылку на объект перед выполнением команды и отмена означает просто возврат к старому объекту.

Чтобы позволить игроку отменить движение, нам нужно сохранить последнюю выполненную им команду. И потом, когда мы жмакнем `Ctrl-Z`, мы просто вызовем метод `undo()`. (Если мы уже выполнили отмену, то по нажатию на ту же кнопку можно выполнить команду повтор и выполнить команду снова.)

Поддержка множественной отмены не намного сложнее. Вместо того, чтобы просто запоминать последнюю команду, мы будем хранить список команд и ссылку на "текущую". Когда игрок выполняет команду, она добавляется в список команд и помечается как "текущая".



Когда игрок выбирает "Отмена", мы отменяем текущую команду и сдвигаем указатель на одну позицию назад. Когда мы выполняем повтор, мы перемещаем указатель на позицию вперед и выполняем команду. Когда игрок после отмены выполняет новую команду, все содержимое списка после текущей команды выбрасывается.

Когда я первый раз реализовал это в редакторе уровней, я почувствовал себя волшебником. Я был изумлен насколько прямолинейным является это решение и насколько хорошо оно работает. Конечно вам потребуется некоторая дисциплина, чтобы оформить все модификации в виде команд, но как только вы с этим справитесь — дальше все будет просто.

> Повтор встречается в играх не часто, а вот повторное проигрывание — очень часто. Прямолинейная реализация могла бы записывать состояние всей игры целиком на каждом кадре. Но такой подходи потребует слишком много памяти.
>
> Вместо этого многие игры записывают набор команд, выполненных на каждом кадре. Чтобы проиграть игру заново, движок просто запускает игру в обычном режиме и выполняет предварительно записанные команды.

Круто или бесполезно?

Как я говорил раньше, команды похожи на функции первого класса или замыкания, однако во всех примерах мы использовали определение классов. Если вы знакомы с функциональным программированием, вам наверное интересно — где же функции?

Я написал примеры таким образом, потому что поддержка функций первого класса в `C++` весьма ограничена. Указатели на функции не имеют состояния, функторы — странные и все равно требуют определения классов, а лямбды в `C++11` сложны в работе из-за ограничений ручного управления памятью.

При этом я *не* утверждаю, что вы не можете использовать для реализации шаблона *Команда* функции в других языках. Если вам доступна роскошь в виде языка с поддержкой настоящих замыканий — используйте их конечно! В некоторых случаях шаблон *Команда* вообще используется в языках, не поддерживающих замыкания для их эмуляции.

> Я говорю о некоторых способах, потому что разработка настоящих классов или структур для команд может быть полезна даже в языках, поддерживающих замыкания. Если в вашей команде поддерживается много операций (как в отменяемых командах), привязка их к единственной функции будет смотреться неуклюже.
>
> Определение настоящего класса с полями дает возможность читающему код явно видеть какие данные содержит команда. Замыкания прекрасны в своей немногословности для оборачивания состояния, но они могут быть настолько автоматизированными, что будет сложно понять что собственно они хранят.

Например, если мы пишем игру на `JavaScript`, мы можем написать команду движения следующим образом:

```
function makeMoveUnitCommand(unit, x, y) {
  // эта функция представляет собой объект команды:
  return function() {
    unit.moveTo(x, y);
  }
}
```

С помощью пары замыканий мы можем реализовать отмену (undo):

```
function makeMoveUnitCommand(unit, x, y) {
  var xBefore, yBefore;
  return {
    execute: function() {
      xBefore = unit.x();
      yBefore = unit.y();
      unit.moveTo(x, y);
    },
    undo: function() {
     unit.moveTo(xBefore, yBefore);
    }
 };
}
```

Если вам комфортно работать в функциональном стиле, такой способ покажется для вас естественным. Если нет, я надеюсь эта глава вам немного помогла. Для меня осознание полезности шаблона *Команда* стало важным шагом в понимании полезности всей парадигмы функционального программирования в целом.

Смотрите также

- Вы можете наплодить достаточно большое количество классов команд. Чтобы упростить их определение, можно создать общий базовый класс с кучей удобный высокоуровневых методов, которые наследующие его классы могут комбинировать для формирования своего поведения. В таком случае главный метод команды `execute()` превращается в Подкласс песочница (Subclass Sandbox).
- В наших примерах, мы явно указывали какой актер должен выполнять команду. В некоторых случаях, особенно когда модель объекта организована иерархически, все может быть не столь очевидно. Объект может ответить на команду, а может перепоручить ее выполнение какому либо другому подчиненному объекту. Если вы это сделаете, вы получите Цепочку ответственности (Chain of Responsibility) GoF.
- Некоторые команды представляют собой прямолинейное поведение как в примере с `JumpCommand`. В этом случае иметь больше одного экземпляра класса — пустая трата памяти, потому что все экземпляры идентичны. В такой ситуации вам пригодится класс Приспособленец (Flyweight).

Можно применять и Синглтон (Singleton), но друзья не позволяют создавать друзьям синглтоны.

Приспособленец (Flyweight)

Туман рассеивается, открывая нашему взгляду величественный старый лес. Бесчисленные кедры образуют над вами зеленый свод. Ажурная листва пронизывается отдельными лучиками света, окрашивая туман в желтые цвета. Меж гигантских стволов виден бесконечный лес вокруг.

О таких сценах внутри игры мы как игровые разработчики и мечтаем. И именно для таких сцен как нельзя лучше подходит скромный шаблон с именем *Приспособленец (Flyweight)*.

Лес для деревьев

Я могу описать целый лес всего несколькими предложениями, но *реализация* его в настоящей игре — совсем другая история. Если вам захочется вывести на экране весь лес из индивидуальных деревьев целиком, любой графический программист сразу увидит миллионы полигонов, которые придется обработать видеокарте на каждом кадре.

Мы говорим именно о тысячах деревьев, геометрия каждого из которых достаточно детализирована и насчитывает тысячи полигонов. Даже, если у вас найдется достаточно *памяти*, чтобы это все уместить, для того, чтобы отрендерить лес целиком, вам нужно будет пропустить это все через шины процессора и видеокарты.

С каждым деревом связаны следующие данные:

- Полигональная сетка, описывающая его ствол, ветви и листву.
- Текстура коры и листьев.
- Положение и ориентация в лесу.
- Индивидуальные настройки, такие как размер и оттенок, благодаря которым каждое дерево в лесу будет выглядеть индивидуально.

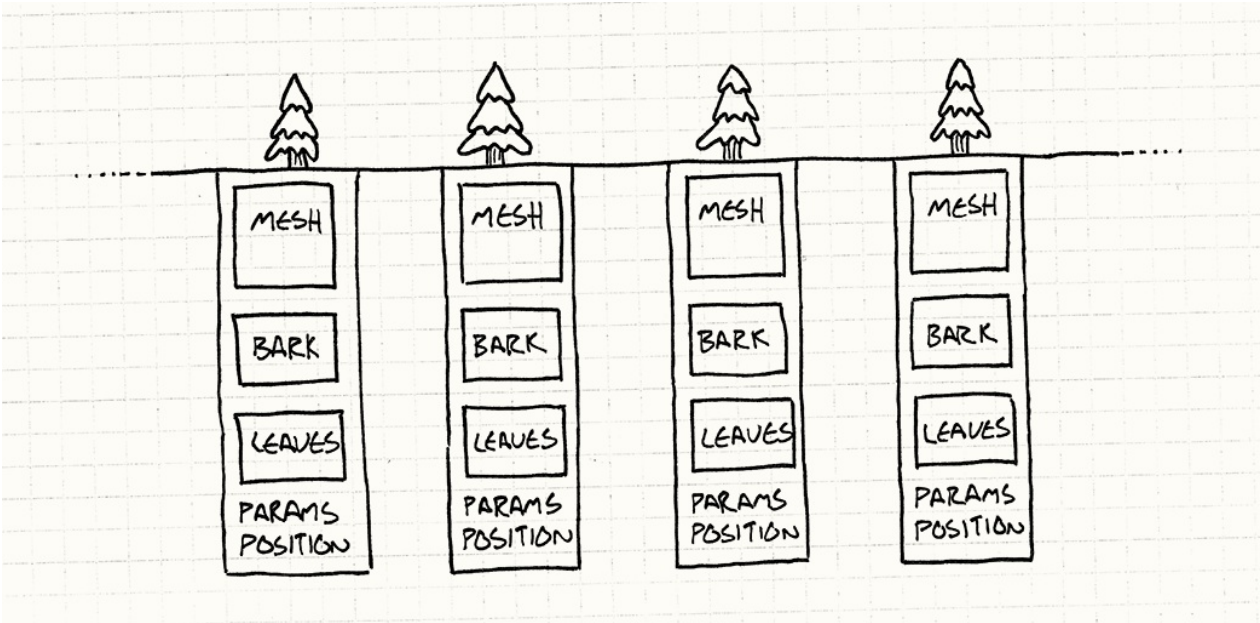
Если набросать описывающий все это код, получится нечто подобное:

```
class Tree
{
private:
  Mesh mesh_;
  Texture bark_;
  Texture leaves_;
  Vector position_;
  double height_;
  double thickness_;
  Color barkTint_;
  Color leafTint_;
};
```

Целая куча данных и здоровенные полигональная сетка и текстура. Весь лес определенно не получится запихнуть целиком в видеокарту на каждом кадре. К счастью есть проверенный временем способ обойти это ограничение.

Дело тут в том, что даже несмотря на то, что деревьев в лесу тысячи, выглядят они довольно похоже. Потому что все используют одни и те же сетку и текстуру. Это значит, что большинство полей в объекта идентичны для всех его экземпляров.

> Я бы посчитал безумцем или миллиардером того, кто стал бы делать для каждого дерева отдельную модель.



> Обратите внимание, что данные в прямоугольнике одинаковы для всех деревьев.

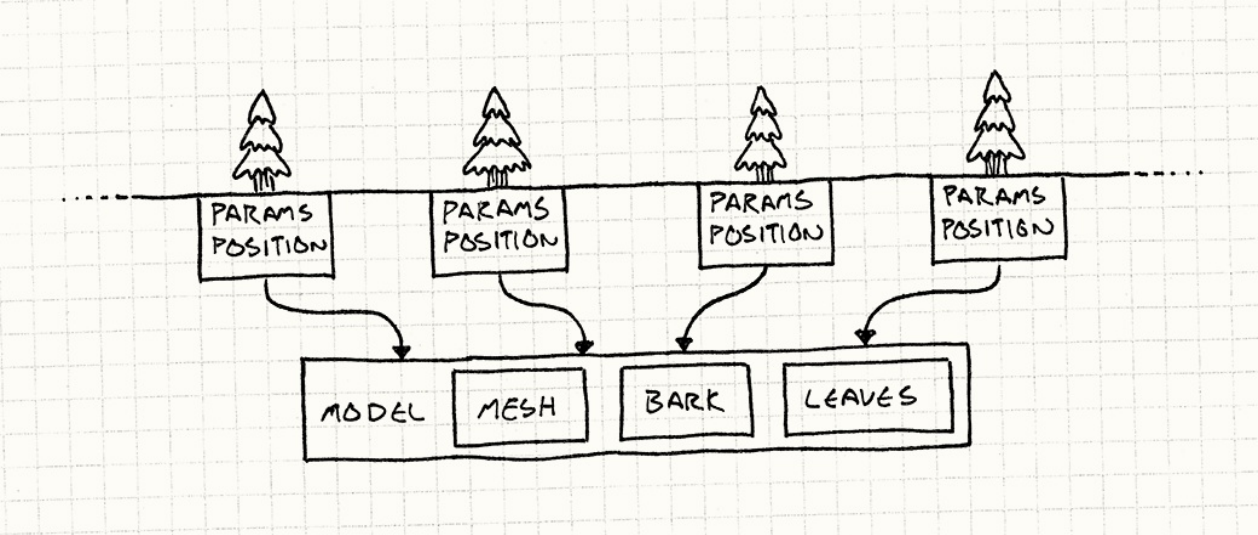
Мы можем это явно смоделировать путем разделения объекта пополам. Мы вытаскиваем общие для всех деревьев данные, и перемещаем их в отдельный класс:

```
class TreeModel
{
private:
  Mesh mesh_;
  Texture bark_;
  Texture leaves_;
};
```

Игре нужен только один экземпляр этого класса, потому что нам незачем иметь тысячи копий одной и той же сетки и текстуры. Поэтому каждый экземпляр дерева в мире должен иметь *ссылку* на общий `TreeModel`. Остальные данные являются индивидуальными для каждого экземпляра:

```
class Tree
{
private:
  TreeModel* model_;

  Vector position_;
  double height_;
  double thickness_;
  Color barkTint_;
  Color leafTint_;
};
```

Результат можно изобразить следующим образом:



> Это очень напоминает шаблон Объект тип (Type Object)]. Оба основаны на делегировании части состояния объекта другому объекту, разделяемому между многими экземплярами. Только области применения у этих шаблонов разные.
>
> Область применения шаблона *Объект тип* — это минимизация количества классов, которые вам нужно определять при добавлении "типов" в свою модель объектов. В качестве бонуса вы получаете разделение памяти. А вот шаблон *Приспособленец* в первую очередь предназначен для увеличения эффективности использования памяти.

Это конечно все хорошо и экономит нам кучу памяти, но как это поможет нам в рендеринге? Прежде чем лес появится на экране, его нужно пропустить через память видеокарты. Нам нужно организовать наше разделение ресурсов таким образом, чтобы это было понятно видеокарте.

Тысяча экземпляров

Для того чтобы минимизировать количество данных, передаваемых видеокарте, мы будем передавать общие данные, т.е. `TreeModel` только *один раз*. После этого мы будем по отдельности передавать индивидуальные для каждого экземпляра данные — позицию, цвет и размер. А затем просто скомандуем видеокарте "Используй эту модель для отрисовки всех этих экземпляров".

К счастью, `API` современных видеокарт такую возможность поддерживает. Детали конечно гораздо сложнее и выходят за рамки рассмотрения этой книги, однако и в `Direct3D` и в `OpenGL` присутствует возможность рендеринга экземпляров (instanced rendering).

> Сам по себе этот `API` видеокарты свидетельствует о том, что шаблон *Приспособленец* — единственный из шаблонов банды четырех, получивший аппаратную реализацию.

В обеих `API` вы формируете два потока данных. Первый — это общие данные, которые используются много раз — сетки и текстуры, как в нашем примере. Второй — список экземпляров и их параметры, которые позволяют варьировать данные из первой группы данных во время отрисовки. Весь лес появляется после единственного вызова отрисовки.

Шаблон приспособленец

Теперь, когда у нас есть хороший пример, я могу рассказать вам о самом шаблоне. *Приспособленец*, как следует из его имени, вступает в игру когда нам требуется максимально облегченный объект, обычно потому что нам нужно очень много таких объектов.

При использовании метода рендеринга экземпляров (instanced rendering) дело даже не в том, что нужно много памяти, а в том, что требуется слишком много *времени*, чтобы прокачать данные о каждом дереве через шину видеокарты. Главное, что базовая идея общая.

Шаблон решает эту проблемы с помощью разделения данных объекта на два типа: первый тип данных — это неуникальные для каждого *экземпляра* объекта данные, которые можно иметь в одном экземпляре для всех объектов. Банда четырех называет их *внутренним* (intrinsic) состоянием, но мне больше нравится название "контекстно-независимые". В нашем примере это геометрия и текстура дерева.

Оставшиеся данные — это *внешнее* (extrinsic) состояние, все что уникально для каждого экземпляра. В нашем случае это позиция, масштаб и цвет каждого из деревьев. Также, как и в приведенном выше фрагменте кода, этот шаблон предотвращает перерасход памяти с помощью разделения одной копии внутреннего состояния между всеми местами, где оно появляется.

То, что мы до сих пор видели, выглядит как простое разделение ресурсов и вряд ли заслуживает того, чтобы называться шаблоном. Частично это вызвано тем, что в нашем примере присутствует очевидное *свойство* (identity), выделяемое в качестве разделяемого ресурса — `TreeModel`.

Я считаю этот шаблон менее очевидным (и поэтому более хитрым), когда он используется в случае, где выделить свойства для разделяемого объекта не так легко. В таком случае возникает впечатление, что объект магическим образом оказывается в нескольких местах одновременно. Давайте я продемонстрирую это на примере.

Место где можно пустить корни

Земля, на которой растут наши деревья, тоже должна быть представлена в игре. Это могут быть участки травы, грязи, холмов, озер и рек и любой другой типа местности, который вы только сможете придумать. Мы сделаем нашу землю *на основе тайлов* (tile - плитка, *прим. перев.*): вся поверхность будет состоять из отдельных маленьких плиток. Каждая плитка будет относиться к какому-либо типу местности.

Каждый из типов местности имеет ряд параметров, влияющих на геймплей:

- Стоимость перемещения, определяющая скорость с которой игроки могут по ней

двигаться.
- Флаг, означающий что местность залита водой и по ней можно перемещаться на лодке.
- Используемая для рендеринга текстура.

Так как все мы — игровые программисты параноики в плане эффективности, мы точно не станем хранить все эти данные для каждого тайла в игровом мире. Вместо этого обычно используется перечисление для описания типов местности:

> В конце концов мы усвоили наш урок с этим лесом.

```
enum Terrain
{
  TERRAIN_GRASS,
  TERRAIN_HILL,
  TERRAIN_RIVER
  // другие типы местности...
};
```

А сам мир хранит здоровенный массив этих значений:

```
class World
{
private:
  Terrain tiles_[WIDTH][HEIGHT];
};
```

> Я использую для хранения 2D сетки многомерный массив. В C++ это эффективно, потому что все элементы упакованы в одном месте. В Java и других managed языках, мы бы получили просто массив строк, каждый из элементов которого был бы ссылкой на массив элементов столбика. Т.е. особой эффективностью в работе с памятью здесь не пахнет.
>
> В любом случае в реальном коде реализацию `2D` сетки лучше спрятать. В примере такой подход выбран исключительно ради простоты.

Чтобы получить полезную информацию о тайле, используется нечто наподобие:

```
int World::getMovementCost(int x, int y)
{
  switch (tiles_[x][y])
  {
    case TERRAIN_GRASS: return 1;
    case TERRAIN_HILL: return 3;
    case TERRAIN_RIVER: return 2;
    // другие типы местности...
  }
}

bool World::isWater(int x, int y)
{
  switch (tiles_[x][y])
  {
    case TERRAIN_GRASS: return false;
    case TERRAIN_HILL: return false;
    case TERRAIN_RIVER: return true;
    // другие типы местности...
  }
}
```

Ну вы поняли идею. Такой подход работает, но я нахожу его уродливым. Когда я думаю о скорости перемещения по местности, я предполагаю увидеть *данные* о местности, а они вместо этого внедрены в код. Еще хуже то, что данные об одном типе местности размазаны по целой куче методов. Хотелось бы видеть все это инкапсулированным в одном месте. В конце концов именно для этого и предназначены объекты.

Было бы здорово иметь настоящий *класс* для местности наподобие такого:

```
class Terrain
{
public:
  Terrain(int movementCost,
          bool isWater,
          Texture texture)
  : movementCost_(movementCost),
    isWater_(isWater),
    texture_(texture)
  {}

  int getMovementCost() const { return movementCost_; }
  bool isWater() const { return isWater_; }
  const Texture& getTexture() const { return texture_; }

private:
  int movementCost_;
  bool isWater_;
  Texture texture_;
};
```

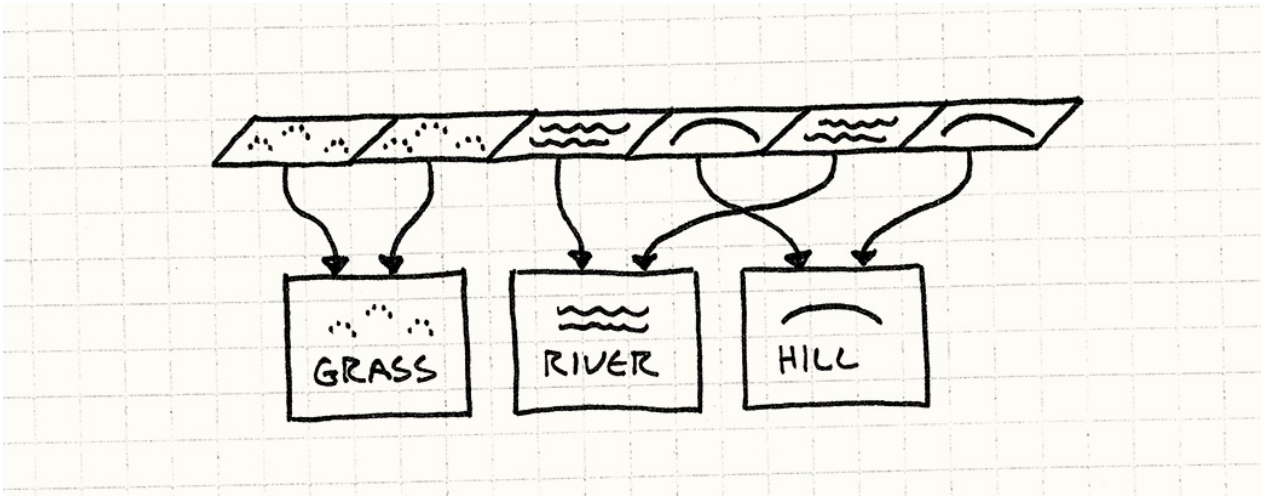
> Вы можете заметить, что все методы здесь объявлены как ``const```. Это не совпадение. Так как один и тот же объект используется в различном контексте, если мы его изменим, изменения одновременно произойдут и во всех остальных местах.
>
> Возможно это не то, что вам нужно. Разделение объектов в целях экономии памяти — это оптимизация, не влияющая на видимое поведение приложения. Вот поэтому объект *Приспособленец* практически всегда делают неизменным (immutable).

Но мы не можем согласиться с тем, что нам придется платить производительностью за роскошь иметь экземпляр данного класса для каждого тайла. Если вы внимательно посмотрите на класс, вы обратите внимание на то, что здесь *нет* вообще никакой специфической информации, определяющей *где* находится тайл. Т.е. в терминах приспособленца, все состояние местности является внутренним (intrinsic) или "контекстно-независимым".

Принимая это во внимание, у нас нет причин иметь больше одного объекта для каждого типа местности. Каждый тайл травы идентичен любому другому. Вместо того, чтобы иметь массив перечислений или объектов `Terrain`, мы будем хранить массив *указателей* на объекты `Terrain`:

```
class World
{
private:
  Terrain* tiles_[WIDTH][HEIGHT];

  // Другие вещи...
};
```

Каждый из тайлов, относящихся к одному типу местности указывает на один и тот же экземпляр местности.



Так как экземпляры местности используется в нескольких местах, управлять их временем жизни будет сложнее, чем если бы создавали их динамически. Вместо этого мы будем прямо хранить их в мире:

```
class World
{
public:
  World()
  : grassTerrain_(1, false, GRASS_TEXTURE),
    hillTerrain_(3, false, HILL_TEXTURE),
    riverTerrain_(2, true, RIVER_TEXTURE)
  {}

private:
  Terrain grassTerrain_;
  Terrain hillTerrain_;
  Terrain riverTerrain_;

  // Другие вещи...
};
```

Теперь мы можем использовать их для отрисовки земли следующим образом:

```
void World::generateTerrain()
{
  // Заполняем землю травой.
  for (int x = 0; x < WIDTH; x++) {
    for (int y = 0; y < HEIGHT; y++) {
      // Добавляем немного холмиков.
      if (random(10) == 0) {
        tiles_[x][y] = &hillTerrain_;
      } else {
        tiles_[x][y] = &grassTerrain_;
      }
    }
  }

  // Добавляем рекую
  int x = random(WIDTH);
  for (int y = 0; y < HEIGHT; y++) {
    tiles_[x][y] = &riverTerrain_;
  }
}
```

> Могу сказать, что это не самый лучший в мире алгоритм процедурной генерации местности.

Дальше, вместо методов в `World` для доступа к параметрам местности, мы можем просто возвращать объект `Terrain` напрямую:

```
const Terrain& World::getTile(int x, int y) const
{
  return *tiles_[x][y];
}
```

Таким образом `World` больше не перегружен различной информацией о местности. Если вам нужны некоторые параметры тайла, вы можете получить их у самого объекта:

```
int cost = world.getTile(2, 3).getMovementCost();
```

Мы снова вернулись к приятному `API` работы с реальным объектом и добились этого практически без дополнительных накладных расходов: указатель обычно занимает не больше места чем перечисление.

Что насчет производительности?

Я говорю "почти", потому что для точной оценки производительности нужно проводить точное сравнение по сравнению с использованием перечислений. Обращение к местности через указатели предполагает косвенное обращение. Чтобы получить данные о местности, например скорость перемещения, вам сначала потребуется получить указатель в массиве местностей, чтобы через него получить объект местности и только потом получить у него скорость перемещения. Попытка обращения через массив может привести к промаху кеша, а стало быть к общему замедлению работы.

> Более подробно о прыганье по указателям (pointer chasing) и про промахи кеша (cache misses) можно почитать в главе Локализация данных (Data Locality).

Как всегда, при оптимизации стоит пользоваться золотым правилом — *сперва выполняем профилирование*. Современное железо настолько сложно, что производительность уже нельзя оценивать на глазок. Мои собственные тесты показали, что применение шаблона *Приспособленец* не дает падения производительности по сравнению с применением перечислений (enum). Более того, *Приспособленец* даже быстрее. Но тут стоит учитывать и то, насколько часто данные выгружаются из памяти.

В чем я *точно* уверен, так это в том, что использование объекта приспособленца скидывать со счетов не стоит. Он дает вам преимущества объектно-ориентированного стиля без расходов на кучу дополнительных объектов. Если вы ловите себя на создании множества последовательностей, а затем организовываете по ним выбор с помощью `switch`, попробуйте использовать шаблон. Ну, а если боитесь за производительность, то по крайней мере проверьте свои опасения профайлером, прежде чем приводить свой код в менее поддерживаемую форму.

Смотрите также

- В примере с тайлами мы сразу создали экземпляры для каждого типа местности и сохранили их в `World`. Таким образом, мы получили возможность повторно использовать и разделять экземпляры. Во многих других случаях у вас не будет желания создавать сразу *всех* возможных приспособленцев.

  Если вы не можете предугадать какие из них вам понадобятся, возможно лучше создавать их по запросу. Чтобы воспользоваться преимуществом разделения ресурсов вы можете организовать проверку, не загружен ли уже нужный вам экземпляр. Если загружен, вы просто возвращаете этот экземпляр.

Обычно это значит, что вам следует инкапсулировать их создание внутри некоторого интерфейса, который вначале будет производить поиск уже загруженных объектов. Пример такого сокрытия конструктора демонстрирует шаблон Фабричный метод (Factory Method)[GoF].

- Чтобы иметь возможность возвратить ранее созданного приспособленцы, вам нужно хранить пул уже загруженных объектов. Если называть имена, то для их хранения можно использовать Пул объектов (Object Pool).
- Когда вы используете шаблон Состояние (State), у вас часто возникает объект "состояние", который не имеет никаких полей, специфичных для машины, на которой это состояние используется. Для этого вполне достаточна сущность и методы состояния. В этом случае вы можете легко применять этот шаблон и переиспользовать один и тот же экземпляр состояния одновременное во множестве машин состояний.

Наблюдатель (Observer)

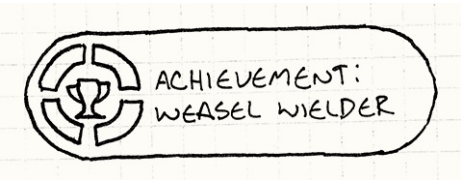
Мы не сможете бросить камень в винчестер и не попасть в приложение, построенное с использованием архитектуры Модель-Вид-Контроллер (Model-View-Controller), в основе которой как раз и лежит шаблон *Наблюдатель*. *Наблюдатель* настолько распространен, что даже включен в `Java` в библиотеку ядра (java.util.Observer), а в `C#` вообще является частью *языка* (ключевое слово event).

> Как и многое другое в программировании, `MVC` была изобретена последователями языка `Smalltalk` в 70-х. Вполне возможно что любители `Lisp` пришли к этому еще в 60-е, но не посчитали нужным это задокументировать.

Наблюдатель — один из самых используемых и широко известных шаблонов Банды четырех, но так как по какой-то непонятной причине мир игровой разработки зачастую обособлен, возможно для вас он станет новинкой. Ну а если вы не только что покинули монастырь, у меня для вас припасен мотивирующий пример.

Достижение разблокировано

Допустим, вам нужно добавить в игру систему достижений. Это будут дюжины значков, которые игрок может заработать "Убив 100 демонических обезьян", "Упав с моста" или "Пройдя уровень, имея на руках только дохлого хорька".



> Готов поклясться, что рисовал эту картинку без всяких задних мыслей.

Реализовать такую систему не так уж просто, потому что к разблокированию достижения может вести самое различное поведение. Если вы не будете осторожны, корни вашей системы достижений расползутся во всему остальному коду. Потому что достижение "Упасть с моста" явно будет связано с работой физического движка, но уверены ли вы, что хотите видеть функцию `unlockFallOffBridge()` прямо в самой гуще алгоритмических вычислений?

Чего мы как обычно хотим, так это того, чтобы весь код, касающийся одного аспекта игры, был максимально сконцентрирован в одном месте. Сложность в том, что на получения достижений влияют самые различные объекты геймплея. Но как этого достичь не внедряя код достижений повсюду?

> Это гипотетический вопрос. Ни один уважающий себя программист физических движков не позволит вам влезать в свой прекрасный математический код с такой приземленной вещью как *геймплей*.

Здесь то нам и поможет шаблон наблюдатель. Он позволяет коду объявлять что произошло нечто интересное *не заботясь о том, кто получит уведомление*.

Например, если у вас есть физический код, который занимается симуляцией гравитации и определяет какие тела лежат на плоскости, а какие стремительно несутся к ней в падении. Чтобы реализовать упомянутое достижение "Упасть с моста", вам нужно внедрить сюда код получения достижения. Но в результате мы получим месиво в коде. Вместо этого мы поступим немного иначе:

```
void Physics::updateBody(PhysicsBody& body)
{
  bool wasOnSurface = body.isOnSurface();
  body.accelerate(GRAVITY);
  body.update();

  if (wasOnSurface && !body.isOnSurface()) {
    notify(body, EVENT_START_FALL);
  }
}
```

Всё, что этот код делает — это говорит "Ну, мне как бы все равно кому это интересно, но этот объект только что упал. Можете делать с этим фактом что хотите."

> Физический движок просто решает какие уведомления посылать. Поэтому назвать его полностью несвязанным (decoupled) нельзя. Но, к сожалению, при построении своей архитектуры мы стремимся к тому, чтобы сделать ее *лучше*, а не *совершенной*.

Система получения достижений регистрирует себя таким образом, чтобы физический код мог посылать сообщения, а система достижения их получала. Затем она может проверить, что упавшим телом был наш несчастный герой, и если до этого он находился на мосту, выдаст значок с достижением. При этом мы увидим как достижение разблокируется с салютом и фанфарами и все это без вмешательства в физический код.

> Теперь при необходимости вы можете полностью поменять всю систему достижений не трогая ни строчки кода в физическом движке. Он по прежнему посылает свои сообщения и не имеет никакого значения, что эти сообщения никто не принимает.

Конечно, если вам нужно будет полностью убрать достижения и необходимость в генерации уведомлений внутри физического движка исчезнет, вам придется удалять и код этих уведомлений. Но в то время, когда игра еще эволюционирует, такая гибкость не помешает.

Как это работает

Если вы еще не в курсе как реализовывается этот шаблон, из вышеприведенного описания вы наверное уже обо всем догадались. Но чтобы еще сильнее упростить понимание, я еще раз коротко опишу детали.

Наблюдатель

Начнем мы с самого любопытного класса, который хочет знать обо всем интересном, что делает другой объект. Этого можно добиться следующей реализацией:

```
class Observer
{
public:
  virtual ~Observer() {}
  virtual void onNotify(const Entity& entity, Event event) = 0;
};
```

> Параметры, передаваемые в `onNotify()`, я оставляю на ваше усмотрение. Все-таки наблюдатель — это *шаблон*, а не "готовый код, который вы просто можете вставить в свою игру". Обычно передается объект, который послал уведомление и обобщенный параметр "data", в котором описываются детали.
>
> Если вы используете язык, поддерживающий дженерики (generics) или шаблоны, вы может использовать их, но все таки неплохо использовать здесь объект, специфичный для варианта использования. В своем примере я жестко закодировал передачу сущности и перечисления (enum), описывающего что произошло.

Каждый конкретный класс, реализовывающий это становится наблюдателем. В нашем примере это система достижений, примерно следующего вида:

```
class Achievements : public Observer
{
public:
  virtual void onNotify(const Entity& entity, Event event) {
    switch (event) {
    case EVENT_ENTITY_FELL:
      if (entity.isHero() && heroIsOnBridge_) {
        unlock(ACHIEVEMENT_FELL_OFF_BRIDGE);
      }
      break;

    // Обработка остальных событий и обновление heroIsOnBridge_...
    }
  }

private:
  void unlock(Achievement achievement) {
    // Разблокирование если не было разблокировано раньше...
  }

  bool heroIsOnBridge_;
};
```

Объект

Метод уведомления запускается объектом, над которым ведется наблюдение. Банда четырех называет его в своей манере "объектом" (subject). Он делает две вещи. Во-первых, он хранит список наблюдателей, которые терпеливо ожидают интересных им сообщений:

```
class Subject
{
private:
  Observer* observers_[MAX_OBSERVERS];
  int numObservers_;
};
```

> В настоящем коде вы скорее всего будете использовать динамические наборы данных, а не массив фиксированной длины. Я использую здесь массив, чтобы не смущать людей, использующих отличные от `C++` языки и не знакомых со стандартной библиотекой `C++`.

Обратите внимание, что у объекта есть *открытое (public)* `API` для изменения этого списка:

```
class Subject
{
public:
  void addObserver(Observer* observer) {
    // Добавление в массив...
  }

  void removeObserver(Observer* observer) {
    // Удаление из массива...
  }

  // Другие вещи...
};
```

Таким образом, сторонний код может управлять тем, кто получает уведомления. Объект общается с наблюдателями, но не *связан* с ними. В нашем примере, ни в одной строчке кода физики достижения не упоминаются. И, тем не менее, система достижений уведомления получает. Это была самая хитрая часть шаблона.

Также крайне важно то, что в объекте хранится *список* наблюдателей, а не просто один из них. Таким образом, мы добиваемся того, что наблюдатели *между собой* даже косвенно не связаны. Возьмем, например, звуковой движок, который тоже будет наблюдать за падением и проигрывать соответствующий звук. Если объект будет поддерживать только одного наблюдателя, то для того, чтобы звуковой движок смог в нем зарегистрироваться, ему придется предварительно *отменить* регистрацию системы достижений.

Это значило бы, что две системы начнут между собой взаимодействовать, причем не самым лучшим образом, так как одна будет мешать работе другой. Поддержка списка наблюдателей позволяет каждому наблюдателю действовать независимо от всех остальных. Каждый из них будет работать таким образом, как будто он единственный в мире.

Теперь объекту остается только передать уведомления:

```
class Subject
{
protected:
  void notify(const Entity& entity, Event event) {
    for (int i = 0; i < numObservers_; i++) {
      observers_[i]->onNotify(entity, event);
    }
  }

  // Другие вещи...
};
```

> Обратите внимание, что этот код не предполагает изменение наблюдателем списка в методе `onNotify()`. В более правильной реализации возможность таких изменений будет предотвращена или будет обрабатываться специальным образом.

Наблюдаемая физика

Теперь нам осталось еще подцепить сюда физический движок так, чтобы он мог посылать уведомления, а система достижений могла на них подписаться. Будем придерживаться оригинального рецепта из *Паттернов проектирования* и наследуем объект `Subject`:

```
class Physics : public Subject
{
public:
  void updateBody(PhysicsBody& body);
};
```

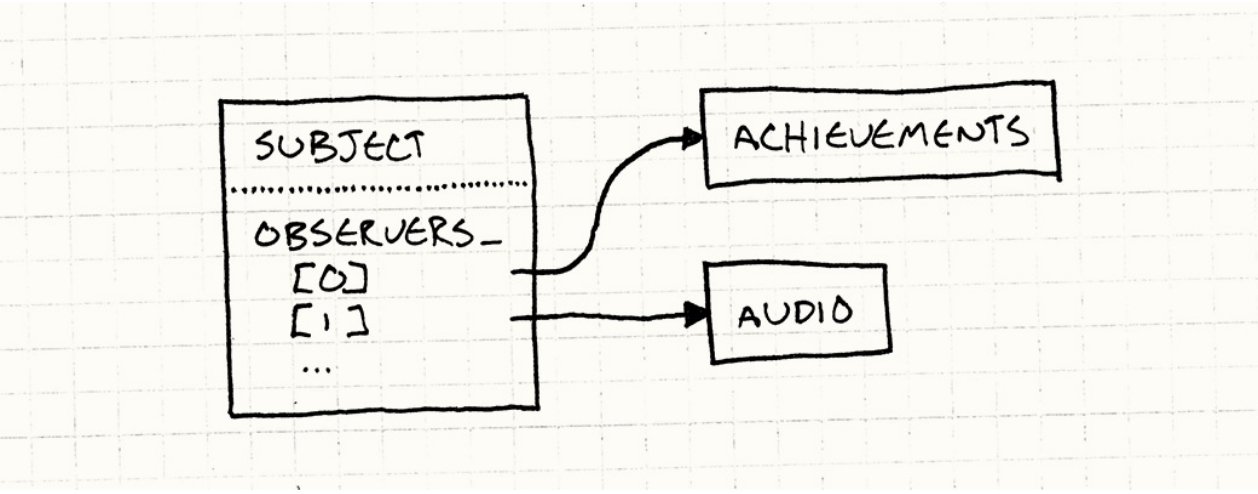
Это позволит нам вызвать `notify()` из `Subject`. Таким образом, унаследованный класс может вызывать `notify()` для отправки сообщений, а внешний код — нет. В то же время, `addObserver()` и `removeObserver()` публичные. Поэтому любой может наблюдать за подсистемой физики.

> Если бы это был реальный код, я бы не стал использовать здесь наследование. Вместо этого я позволил бы Physics иметь экземпляр Subject. Вместо наблюдения за всем физическим движком, объект наблюдателя стал бы отдельным объектом "уведомления о падении". Наблюдатели могут регистрироваться с помощью конструкции наподобие:

```
physics.entityFell()
.addObserver(this);
```

> На мой взгляд в этом и заключается разница между системами "наблюдатель" и "события". Первая наблюдает за всем что может быть интересным. Вторая наблюдает за объектом, описывающим что произошло нечто интересное.

Теперь, когда физический движок выполняет нечто стоящее внимания, вызываем `notify()` как в первом мотивирующем примере. Дальше выполняется обход списка наблюдателей и они уже делают все остальное.



Довольно легко, верно? Всего один класс, содержащий список указателей на экземпляры определенного интерфейса. Даже трудно поверить, что такая прямолинейная конструкция является основой бесконечного количества программ и фреймворков.

Впрочем, не обходится и без критики. Когда я начал расспрашивать игровых программистов что они думают об этом шаблоне, я услышал множество нелестных отзывов. Попытаюсь их перечислить.

"Это слишком медленно"

Я это часто слышу, особенно от тех программистов, которые не разбираются в деталях. Все дело в том, что как только они слышат хоть что-то о шаблонах проектирования, они сразу представляют себе кучу классов, непрямых связей и безумного перерасхода драгоценных процессорных тактов.

Шаблон наблюдатель получил такую плохую репутацию потому, что в связи с ним слишком часто упоминаются имена таких темных личностей как "события" (events), "сообщения" (messages) и даже "привязка данных" (data binding). Некоторые из этих систем *могут* быть медленными (иногда это делается сознательно и на то есть причины). Виной тому очереди и динамическое выделение памяти при каждом уведомлении.

> Вот поэтому я и считаю, что шаблоны стоит документировать. Когда терминология размывается, мы теряем способность общаться кратко и однозначно. Вы говорите "Наблюдатель", а кто-то слышит "События" или "Сообщения", потому что никто не удосужился записать и потом прочитать о разнице.
>
> Вот для чего я и написал эту книгу. Чтобы не быть голословным — глава об Очереди событий (Event Queue) в книге тоже имеется.

Но теперь, когда вы видели как шаблон реализуется на самом деле, вы знаете как обстоит дело. Отсылка уведомления — это всего лишь проход по списку и вызов виртуального метода. Естественно — это немного медленнее, чем статический метод диспетчеризации, но цена настолько незначительна, что может себя проявить только в самом требовательном к производительности коде.

Я считаю, что применять этот шаблон все-таки лучше не в самой горячей части кода, а там, где вы можете себе позволить применить динамическое выделение памяти. В остальном можно сказать, что никаких накладных расходов нет. Мы не создаем объекты для сообщений. У нас нет очередей. Все что у нас есть — это косвенность вместо синхронных вызовов методов.

Слишком *быстро*?

Не забывайте, что шаблон Наблюдатель работает синхронно. Объект вызывает наблюдателя напрямую, а это значит, что он сам не продолжит работу, пока наблюдатель не вернет свой метод уведомления. Медленный наблюдатель может вообще блокировать выполнение объекта.

Звучит пугающе, но на практике далеко не конец света. Просто следует иметь это в виду. У программистов `UI`, которые занимается событийным программированием годами, даже есть свой лозунг: "никаких потоков в `UI`".

Если вы реагируете на событие синхронно, вам нужно завершить свои дела и передать управление обратно как можно скорее, чтобы не возникла блокировка `UI`. Если вам нужно выполнить что-то долгое — просто поместите это в другой поток или рабочую очередь.

Нужно быть осторожным, если вы планируете использовать наблюдателей вместе с потоками и настоящими блокировками. Если наблюдатель пытается перехватить блокировку в объекте, вы можете разблокировать игру. В движке с активным использованием потоков лучше использовать асинхронные сообщения с помощью Очереди событий (Event Queue).

"Здесь слишком много динамического выделения памяти"

Целые кланы программистов, включая многих игровых программистов, перешли на языки, оснащенные сборщиком мусора и динамическое выделение памяти превратилось для них в пустую страшилку. Однако для программ, для которых критична скорость выполнения, в том числе и игр, управление выделением памяти по

прежнему остается серьезной заботой даже в управляемых языках. Динамическое выделение памяти требует времени для переназначения участков памяти, даже если это делается автоматически.

> Многие игровые разработчики беспокоятся не столько о выделении памяти, сколько о ее фрагментации. Если ваша игра должна работать без падений дни напролет чтобы пройти сертификацию, слишком большая фрагментация памяти может не позволить вам выпустить игру.
>
> Глава Пул объектов (Object Pool) посвящена самой этой проблеме и способам ее решения.

В примере выше я использовал массив фиксированной длины для того, чтобы максимально упростить код. В реальной реализации, список наблюдателей обычно представляет собой динамическую коллекцию и во время работы увеличивается и уменьшается по мере добавления и удаления наблюдателей. Многих пугает такое обращение с памятью.

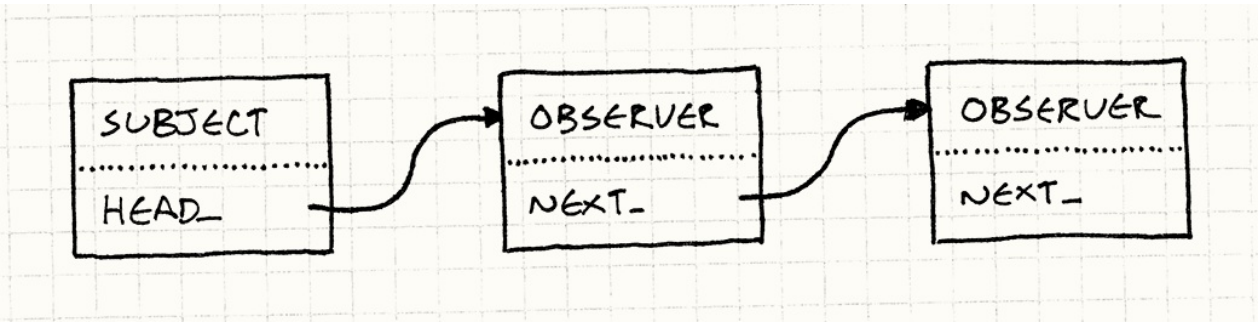
Главное, что нужно понять — это то, что выделение памяти происходит только при настройке наблюдателей. *Посылка* уведомлений вообще не требует выделения памяти — это просто вызов метода. Если вы выполните настройку наблюдателей в начале игры и не будете их больше трогать, количество выделений памяти будет минимальным.

Однако проблема все еще здесь, поэтому я покажу вам способ реализации при котором динамически выделять память вообще не придется.

Связанные в цепочку наблюдатели

В коде, который был у нас до сих пор, `Subject` хранил в себе список на всех `Observer`, за которыми он наблюдал. У самого класса `Observer` не было ссылки на этот список. Это чисто виртуальный интерфейс. Интерфейсы предпочтительнее чем конкретные, постоянные классы, так что это не так уж плохо.

Но если мы перенесем часть состояния в сам `Observer`, мы сможем решить нашу проблему с выделением памяти, передавая объект *по цепочке самих наблюдателей*. Вместо объекта, хранящего набор указателей, наблюдатели станут узлами связанного списка:



Чтобы это реализовать, мы уберем массив из `Subject` и заменим его указателем на голову списка наблюдателей:

```
class Subject
{
  Subject() : head_(NULL)
  {}

  // Методы...
private:
  Observer* head_;
};
```

Сам `Observer` мы дополним указателем на следующего наблюдателя в списке:

```
class Observer
{
  friend class Subject;

public:
  Observer() : next_(NULL)
  {}

  // Другие вещи...
private:
  Observer* next_;
};
```

Еще мы сделаем `Subject` другом. У объекта есть `API` для добавления и удаления наблюдателей, но теперь мы будем управлять этим списком изнутри самого класса `Observer`. Самый простой способ позволить ему работать с этим списком — это сделать его другом.

Чтобы зарегистрировать нового наблюдателя, нужно просто добавить его в список. Просто добавим его в начало списка:

```
void Subject::addObserver(Observer* observer)
{
  observer->next_ = head_;
  head_ = observer;
}
```

Можно выбрать другой вариант и добавить его в конце списка. Такой вариант получается более сложным: `Subject` должен проходить весь список от начала до конца или хранить еще один указатель `tail_`, который будет указывать на последний элемент.

Добавление в начало списка проще, но имеет один побочный эффект. Когда мы проходим список чтобы отослать уведомления каждому наблюдателю, самые *последние* из зарегистрированных наблюдателей получают уведомления *первыми*. Так, что если вы зарегистрировали наблюдателей в последовательности A,B и C, они получат уведомления в последовательности C,B,A.

В теории вас вообще не должна интересовать последовательность. При использовании наблюдателей хорошим тоном считается написание такого кода, который не зависит от очередности уведомлений. А вот если очередность обработки имеет значение, это значит, что между двумя наблюдателями существует связь, которая может больно вам аукнуться.

Давайте посмотрим как выглядит удаление:

```
void Subject::removeObserver(Observer* observer)
{
  if (head_ == observer) {
    head_ = observer->next_;
    observer->next_ = NULL;
    return;
  }

  Observer* current = head_;
  while (current != NULL) {
    if (current->next_ == observer) {
      current->next_ = observer->next_;
      observer->next_ = NULL;
      return;
    }

    current = current->next_;
  }
}
```

> Удаление узла из связанного списка обычно требует обработки неприглядного особого случая для удаления самого первого узла как в данном примере. Более элегантным решением будет использование указателя на указатель.
>
> Я не сделал этого здесь, потому что обычно когда я показываю такой код людям, он вводит в замешательство примерно половину из них.

Так как у нас есть единственный связанный список, нам нужно пройтись по нему, чтобы найти нужного наблюдателя и удалить его. С обычным массивом мы поступали бы точно также. А вот если мы используем *двухсвязный* список, где каждый из наблюдателей есть указатель на следующего и предыдущего наблюдателей, мы можем удалить наблюдателя за одну операцию. В настоящем коде я бы так и поступил.

Единственное что осталось реализовать — это уведомление. Это также просто как перемещение по списку:

```
void Subject::notify(const Entity& entity, Event event)
{
  Observer* observer = head_;
  while (observer != NULL) {
    observer->onNotify(entity, event);
    observer = observer->next_;
  }
}
```

> Здесь мы проходимся по всему списку и уведомляем каждый наблюдатель в нем. Таким образом мы добиваемся одинакового приоритета для всех наблюдателей и их независимость друг от друга.
>
> Мы можем изменить этот механизм таким образом, чтобы когда наблюдатель получал уведомление, он возвращал флаг, означающий что объект должен продолжать прохождение по списку наблюдателей или остановиться. Если вы это сделаете, вы вплотную приблизитесь к шаблону Цепочка ответственности (Chain of Responsibility) GoF.

Не так уж плохо, верно? Объекту может принадлежать любое количество наблюдателей без малейшего динамического использования памяти. Регистрировать и отменять регистрацию также быстро и просто, как и при работе с обычным массивом. Придется пожертвовать только одним маленьким удобством.

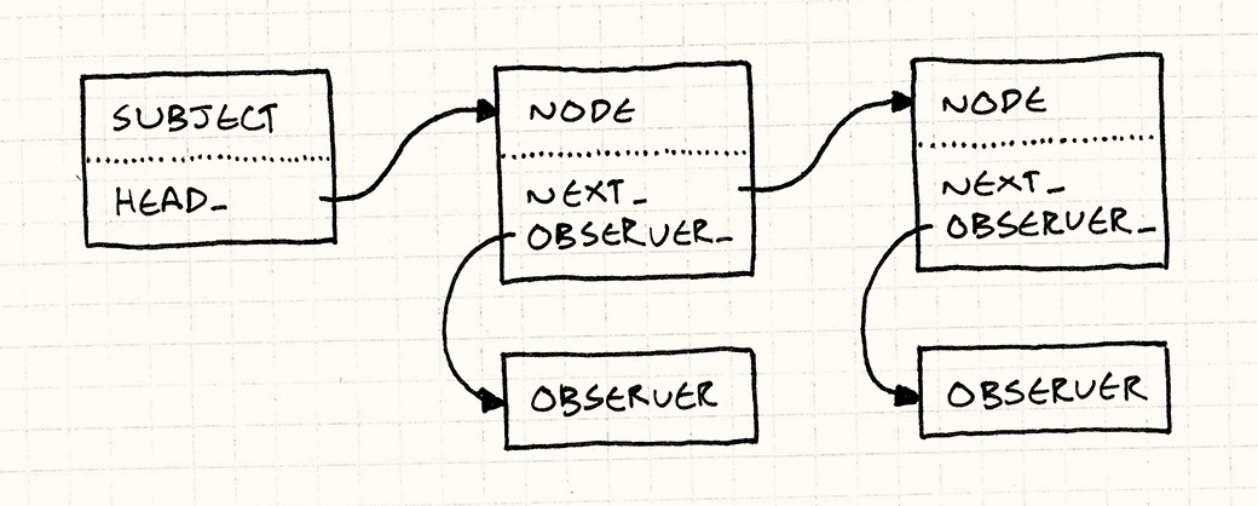
Так как мы используем самого наблюдателя в качестве узла списка, мы предполагаем, что он сам и есть часть списка наблюдателей. Другими словами, наблюдатель может наблюдать только за одним объектом в каждый момент времени. В более

традиционной реализации, когда каждый объект хранит свой собственный список, каждый наблюдатель может находиться сразу в нескольких списках одновременно.

Вам придется смириться с этим ограничением. Я нахожу более естественной ситуацию, когда у объекта есть много наблюдателей, а не наоборот. Если для вас это проблема — существует более сложное решение, которое также не использует динамическое выделение памяти. Оно слишком большое, чтобы целиком поместиться в эту главу, но я покажу вам небольшой набросок, а в остальном вы разберетесь сами.

Пул списков узлов

Как и раньше, каждый объект содержит связанный список наблюдателей. Однако сами эти списки не будут состоять из объектов наблюдателей. Вместо этого они сами будут представлять из себя маленький "узел списка", в котором есть указатель на наблюдатель и указатель на следующий узел в списке.



Так как несколько узлов могут указывать на один и тот же наблюдатель, это означает, что один и тот же наблюдатель может одновременно находиться в списках нескольких объектов. И, таким образом, к нам вернулась способность наблюдать за несколькими объектами одновременно.

> Связанные списки бывают двух типов. Тот тип, который вы изучали в школе, состоит из узлов, хранящих данные. В нашем предыдущем примере связанных наблюдателей все было наоборот: *данные* (в нашем случае наблюдатель) содержат *узел* (т.е. указатель `next_`).
>
> Этот последний тип называется интрузивным (intrusive) связанным списком, потому что использует объекты в списке, вторгающиеся в определение того, что такое представляет из себя сам объект. Интрузивные связанные списки конечно не такие гибкие, но как вы сами видите более эффективные. Их можно часто увидеть например в ядре `Linux`, где понятное дело самое важно — это производительность.

Способ, позволяющий избавиться от динамического выделения памяти прост: так как все узлы у нас одного размера и типа, мы можем предварительно создать для них Пул объектов (Object Pool). Вы получаете для работы список узлов фиксированного размера и можете использовать и повторно использовать их как вам нужно без необходимости дополнительного выделения памяти.

Оставшиеся проблемы

Я думаю мы избавились от трёх главных страшилок, отпугивавших людей от этого шаблона. Как мы увидели, он простой, быстрый и может хорошо работать без сложных манипуляций с памятью. Но значит ли это, что нужно применять наблюдателей везде и всюду?

Теперь другой вопрос. Как и любой другой шаблон — наблюдатель не панацея от всех бед. Даже если он реализован корректно и эффективно, он может не быть правильным решением. Плохая репутация у некоторых шаблонов возникает потому, что люди пытаются применить хороший шаблон к неправильной проблеме и только ухудшают свое положение.

Осталось еще две сложности, одна технического плана и вторая, скорее относящаяся к сложности поддержки. Начнем с технической, потому что такие как правило проще.

Удаление объектов и наблюдателей

Основной код, который мы с вами рассмотрели хорош, но что насчет сопроводительного кода: что происходит, когда мы удаляем объект или наблюдателя? Если просто применить `delete` к любому из наблюдателей, на него останется ссылка

в объекте. Теперь это опасный указатель на удаленный объект. И если этот объект попытается послать уведомление... вот почему некоторые люди начинают ненавидеть `C++`.

> Не хотелось бы ни на кого указывать пальцем, но *Паттерны программирования* не говорят об этой проблеме вовсе.

Уничтожать объект проще потому, что в большинстве наших реализаций у нас нет ссылок на него. Но даже в этом случае отправка объекта в мусор может доставить определенные проблемы. Наблюдатели могут ожидать получения уведомлений в будущем и никак не смогут узнать, что этого больше не произойдет. И теперь они вовсе не наблюдатели, как они сами о себе думают.

Справиться с этим можно несколькими способами. Проще всего решить проблему напрямую. Пусть сам наблюдатель и разбирается с тем, чтобы при удалении отменить повсюду свою регистрацию. Чаще всего наблюдатель обладает информацией о том, за какими объектами он наблюдает, и остается только добавить вызов `removeObserver()` в деструктор.

> Как всегда самое сложно — это не сделать, а *не забыть* сделать.

Если у вас нет желания оставлять наблюдателей висеть в системе, когда объекты превратятся в призраков, это можно легко исправить. Для этого можно заставить каждый объект перед своим уничтожением послать всем финальное уведомление — "последний вздох". Таким образом, все наблюдатели его получат и смогут предпринять соответствующие действия.

> Оплакать, цветочки послать и т.д.

Люди, даже те из нас, которые провели достаточно много времени в обществе машин и обрели некоторые их свойства, по своей природе крайне ненадежны. Вот почему мы придумали компьютеры: они делают ошибки гораздо реже.

Самый правильный ответ состоит в том, чтобы заставить наблюдателей самостоятельно отменять свою регистрацию в объектах при своем удалении. Если вы реализуете эту логику единожды в базовом классе, все остальные, кто будет ее использовать, уже не должны будут заботиться об этом самостоятельно. Сложность при этом конечно увеличивается. Получается, что каждый *наблюдатель* должен хранить список *объектов*, которые за ним следят. Получится двухсторонняя связь с помощью указателей.

Без паники, у нас есть сборщик мусора

Сейчас все модники, любящие современные языки со сборщиками мусора наверняка самодовольно наблюдали за происходящим. Думаете, если вы напрямую не управляете памятью, вам не нужно ей управлять? Подумайте еще раз.

Представьте себе, что у вас есть экран интерфейса, на котором отображается статистика игрока типа его здоровья и т.д. Когда игрок вызывает этот экран, вы создаете новые экземпляр объекта. Когда экран закрывается, вы просто забываете о нем и дальше его удаляет сборщик мусора.

Каждый раз когда игрок получает удар в лицо (или еще куда либо), он посылает уведомления. Экран интерфейса наблюдает за этим и обновляет полосу здоровья. Отлично. А теперь, что произойдет, если игрок закроет окно, а вы не отмените регистрацию наблюдателя?

Интерфейс больше не виден, но сборщик мусора не может его удалить, потому что на него до сих пор есть активная ссылка в наблюдателе. Каждый раз, когда будет загружаться экран, будет создаваться его новый экземпляр.

И все время, пока игрок будет играть, бродить по миру, сражаться, его персонаж будет отсылать уведомления всем экземплярам экрана. Их не видно на экране, но они есть в памяти и тратят такты процессора на обработку невидимых элементов интерфейса. А если, например, при этом издаются звуки, вы получите и реальное подтверждение неправильного поведения.

Эта проблема в системах уведомления настолько распространена, что даже имеет собственное имя: *проблема бывших слушателей (lapsed listener problem)*. Так как объекты остались подписанными на своих слушателей, у вас образовался интерфейс-зомби, занимающий память. Мораль здесь простая — соблюдайте дисциплину при отмене регистрации.

> О важности этой проблемы свидетельствует даже отдельная статья в видипедии.

Что происходит?

Еще одна более глубокая проблема с шаблоном Наблюдатель — это прямое следствие его основного назначения. Мы используем его для того, чтобы избавиться от связности между двумя частями кода. Это позволяет объекту не напрямую общаться с наблюдателем без статической связи между ними.

Это большая победа, когда вам нужно сосредоточиться на поведении объектов и когда лишние связи будут только раздражать и отвлекать вас от главного. Если вы ковыряетесь с физическим движком, вы не хотите забивать свой редактор и свой мозг тоже информацией о каких-то достижениях.

С другой стороны, если ваша программа не работает и баг распространяется на несколько звеньев наблюдателя, разобраться с такой организацией потоков гораздо сложнее. В сильносвязанном коде достаточно просто заглянуть в вызываемый метод. Для вашей среды разработки это детский лепет, потому что связь статическая.

А вот если связность организована через список наблюдателей, единственным способом узнать, кто получил уведомление является просмотр списка во *время выполнения* (runtime). Вместо того, чтобы иметь возможность *статически* видеть структуру коммуникаций в программе, нам приходится изучать ее *императивное, динамическое* поведение.

Мой совет о том, как с этим справиться предельно прост. Если вам часто приходится размышлять об *обоих* участниках коммуникации для того, чтобы понять работу программы, не используйте для реализации этой связи шаблон Наблюдатель. Выберите нечто более явное.

Когда вы копаетесь в какой-то большой программе, вы обычно имеете дело с большими кусками, которые нужно заставить работать вместе. Для этого есть много терминов "разделение внимания" и "когерентность и сцепка" и "модульность", но все сводится примерно к "эти штуки собираются вместе и не хотят работать вот с этими штуками".

Шаблон наблюдатель является отличным способом позволить несвязанным кускам кода общаться друг с другом без объединения в еще больший кусок. *Внутри* одного куска, посвященного одному конкретному аспекту, он уже не так эффективен.

Вот почему он так хорошо вписывается в наш пример: достижения и физика являются наименее связанными областями, зачастую даже реализуемые разными людьми. Нам нужно добиться минимума общения между ними и чтобы ни одной из них не нужны были дополнительные знания о другой.

Наблюдатели сегодня

Паттерны проектирования вышли еще в 90-е (1994). Тогда объектно-ориентированное программирование было *горячей* парадигмой. Каждый программист в мире надеялся "Выучить ООП за 30 дней", а менеджеры среднего звена платили зарплату в зависимости от того, кто сколько классов создал. Крутость программиста определялась глубиной иерархии наследования, которую он наворотил.

> Примерно в тоже самое время Ace of Base записали *три* своих главных хита. Это я к тому, чтобы вы могли судить о вкусах того времени.

Популярность шаблона Наблюдатель пришлась как раз в духе того времени. Поэтому не удивительно, что его классы тяжеловесны. Правда в наше время программисты чувствуют себя уже гораздо комфортнее в функциональном программировании. Реализовывать только ради получения уведомлений целую системы интерфейсов — это не вписывается в сегодняшнее понятие об эстетике.

Сегодня такой подход кажется громоздким и негибким. Он и правда громоздкий и негибкий! Например, вы не можете иметь единственный класс, использующий разные методы уведомления для разных объектов.

> Вот почему объект обычно передает себя наблюдателю. Так как у наблюдателя только один метод уведомления `onNotify()`, если он наблюдает за несколькими объектами, ему нужно иметь возможность указать какой из них его вызвал.

Более современный вариант реализации "наблюдателя" — это просто ссылка на метод или функцию. В языках с поддержкой функций первого класса и особенно с поддержкой замыканий, так чаще всего наблюдатель и реализуют.

> В наше время замыкания есть в *любом* языке. `C++` преодолел ограничения на замыкания в языке без сборщика мусора и даже `Java` в `JDK8` смогла с ними подружиться.

А, например, в `C#` вообще существует "событие" (event), интегрированное в сам язык. С его помощью вы регистрируете наблюдателя — "делегата", который в терминах данного языка является ссылкой на метод. В системе сообщений `JavaScript` наблюдателями *могут* быть как объекты, поддерживающие специальный протокол `EventListener`, так и обычные функции. Чаще всего используют последнее.

Если бы я разрабатывал систему наблюдателя сегодня, я бы выбрал основанную на функциях, а не на классах систему. Даже в `C++` я стремлюсь к системе, которая позволяет регистрировать указателей на функции-члены в качестве наблюдателей, вместо экземпляров некоего интерфейса `Observer`.

> Вот интересный пост в блоге о их реализации на C++.

Наблюдатели завтра

Система событий и другие подобные наблюдателю шаблоны сейчас чрезвычайно распространены. Это довольно избитый пусть. Как только у вас появится опыт их использования в нескольких крупных приложения, вы непременно кое-что заметите. Большое количество кода в ваших наблюдателях выглядит одинаково. Обычно он делает примерно следующее:

1. Получает уведомление о произошедшем изменении.
2. Заставляет какой либо участок `UI` отобразить это новое состояние.

В духе "О, здоровье у нас теперь равно 7? Давайте значит установим длину для полоски здоровья в 70 пикселей." Через некоторое время это становится довольно нудным. Академики компьютерных наук и простые программисты *всегда* старались бороться с этой нудностью. Их попытки известны нам под названиями "программирование потоков данных" (dataflow programming), "функциональное реактивное программирование" (functional reactive programming) и т.д.

Несмотря на некоторые успехи, особенно в отдельных областях, таких как обработка звука или проектирование микрочипов, Святой Грааль до сих пор не найден. Сейчас наиболее популярны наименее амбициозные варианты. Многие современные фреймворки используют "привязку данных" (data binding).

В отличие от радикальной модели, привязка данных не стремится к полному устранению императивного кода и не пытается построить всю архитектуру вокруг гигантского графа потока данных. Все что она делает — это автоматизирует бесполезную работу по настройке интерфейса или пересчету свойств, отражающих изменение какого либо значения.

Как и другие декларативные системы, привязка данных пожалуй слишком медленная для того, чтобы внедрять ее в игровой движок. Но я не удивляюсь, если увижу ее в каких либо менее критичных в плане производительности областях, таких как интерфейс.

И тем временем, старый добрый Наблюдатель по прежнему с нами и ждет нас. Конечно сейчас он уже не выглядит таким захватывающим, как свежие технологии, объединяющие в своем названии слова "функциональные" и "реактивные", зато он крайне прост и точно работает. А для меня это зачастую главные критерии выбора.

Прототип(Prototype)

Первый раз я узнал о существовании слова "прототип" из *Паттернов проектирования*. Сейчас это слово достаточно популярно. Но обычно его используют без привязки к шаблону проектирования GoF. Мы еще к этому вернемся, но для начала я хочу показать вам другое, более интересные области, где можно встретить термин "прототип" и стоящую за ним концепцию. А для начала давайте рассмотрим оригинальный шаблон.

> Я умышленно не пишу здесь "оригинальный". *Паттерны проектирования* цитируют легендарный проект Sketchpad *1963-го* года за авторством Ивана Сазерленда, который можно считать первым примером применения шаблона в природе. Когда все остальные слушали Дилана и Битлз, Сазерленд был занят всего-навсего изобретением базовых концепций `CAD`, интерактивной графики и объектно-ориентированного программирования.
>
> Можете посмотреть демо и впечатлиться.

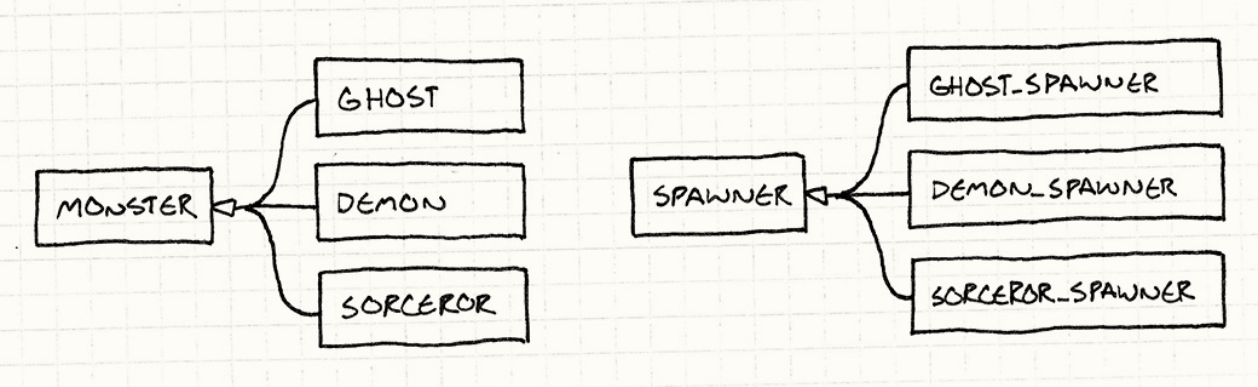
Шаблон проектирования прототип

Давайте представим, что мы делаем игру в стиле Gauntlet. У нас есть всякие существа и демоны, роящиеся вокруг героя и норовящие откусить кусочек его плоти. Эти незваные сотрапезники появляются через "спаунеры" (spawners, тут, источники-генераторы создания существ, прим.пер.) и для каждого типа врагов есть отдельный тип спаунера.

Для упрощения примера давайте сделаем предположение, что для каждого типа монстра в игре имеется отдельный тип. Т.е. у нас есть `C++` классы для `Ghost`, `Demon`, `Sorcerer` и т.д.:

```
class Monster
{
  // Stuff...
};

class Ghost : public Monster {};
class Demon : public Monster {};
class Sorcerer : public Monster {};
```

Спаунер конструирует экземпляры одного из типов монстров. Для поддержки всех монстров в игре мы *можем* использовать прямолинейный подход и заведем класс спаунер для каждого класса монстра. В результате получится следующая иерархия:



> Я должен был выкопать пыльную книгу по UML, чтобы сделать эту схему. ⇦ означает "наследуется".

Реализация будет выглядеть так:

```
class Spawner
{
public:
  virtual ~Spawner() {}
  virtual Monster* spawnMonster() = 0;
};

class GhostSpawner : public Spawner
{
public:
  virtual Monster* spawnMonster() {
    return new Ghost();
  }
};

class DemonSpawner : public Spawner
{
public:
  virtual Monster* spawnMonster() {
    return new Demon();
  }
};

// Ну вы поняли...
```

Если вам конечно не платят за каждую строчку кода, использовать такой подход совсем не весело. Куча классов, куча похожего кода, куча избыточности, куча дублей, куча самоповторов...

Шаблон Прототип предлагает решение. Ключевой мыслью является *создание объекта, который может порождать объекты, похожие на себя*. Если у вас есть один призрак, вы можете с его помощью получить кучу призраков. Если есть демон, можно сделать больше демонов. Любого монстра можно трактовать как *прототипируемого* монстра, используемого для генерации новых версий его самого.

Для реализации этой идеи, мы дадим нашему базовому классу `Monster` абстрактный метод `clone()`:

```
class Monster
{
public:
  virtual ~Monster() {}
  virtual Monster* clone() = 0;

  // Другие вещи...
};
```

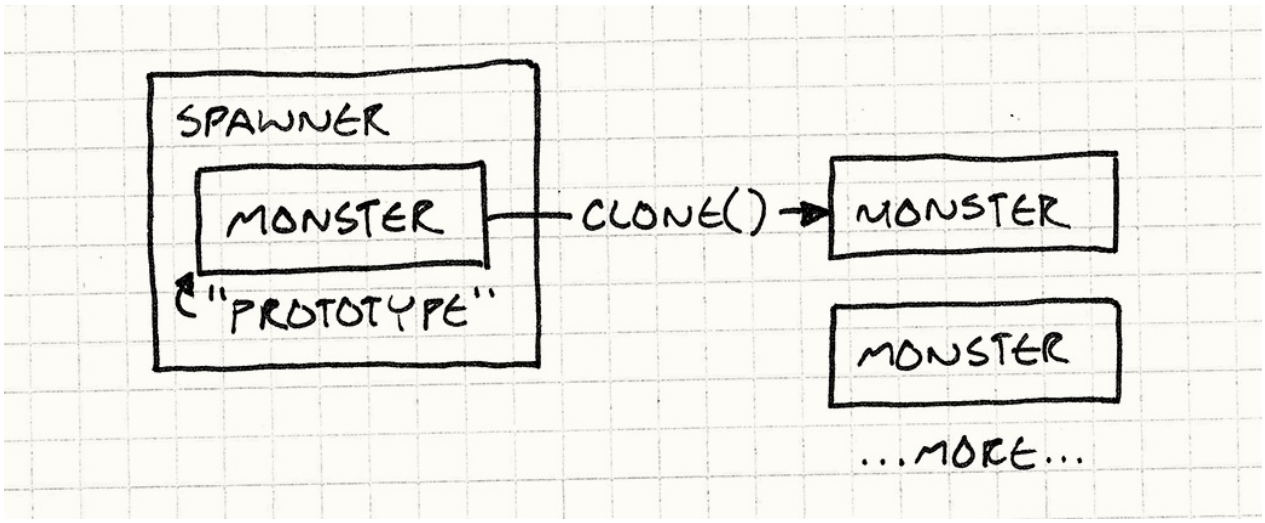
Каждый подкласс монстра предоставляет свою реализацию, которая возвращает объект, идентичный по классу и состоянию ему самому. Например:

```
class Ghost : public Monster
{
public:
  Ghost(int health, int speed)
    : health_(health),
      speed_(speed)
  {}

  virtual Monster* clone() {
    return new Ghost(health_, speed_);
  }

private:
  int health_;
  int speed_;
};
```

Как только все монстры будут его поддерживать, нам больше не нужен будет отдельный класс спаунер для каждого класса монстров. Вместо этого мы обойдемся всего одним:

```
class Spawner
{
public:
  Spawner(Monster* prototype)
    : prototype_(prototype)
  {}

  Monster* spawnMonster() {
    return prototype_->clone();
  }

private:
  Monster* prototype_;
};
```

Внутри себя он содержит монстра, скрытого извне, который используется спаунером в качестве шаблона для штамповки новых монстров ему подобных. Получается нечто наподобие матки пчел, никогда не покидающей своего улья.



Для создания спаунера призраков, мы просто создаем прототипируемый экземпляр призрака и затем создаем спаунер, который будет хранить этот прототип:

```
Monster* ghostPrototype = new Ghost(15, 3);
Spawner* ghostSpawner = new Spawner(ghostPrototype);
```

Интересна одна особенность этого шаблона заключается в том, что он не просто клонирует класс прототипа, но и клонирует его состояние. Это значит, что мы можем сделать спаунер для быстрых призраков, для слабых, для медленных, просто создавая соответствующего прототипируемого призрака.

На мой взгляд этот шаблон одновременно и элегантен и удивителен. Я не могу представить, чтобы дошел до него своим умом, но теперь я просто не могу себе представить, что я мог бы о нем не знать.

Насколько хорошо он работает?

Итак, нам не нужно создавать отдельный класс спаунер для каждого монстра и это хорошо. Но при этом нам нужно реализовывать метод `clone()` в каждом классе монстров. Кода там примерно столько же сколько и в спаунере.

К сожалению, если вы попытаетесь написать корректную реализацию `clone()`, вы быстро наткнетесь на несколько подводных камней. Должен это быть глубокий клон или приблизительный? Другими словами, если демон держит вилы, должен ли клонированный демон тоже держать вилы?

Это не просто выглядит как надуманная проблема, это действительно *надуманная проблема*. Нужно принять как должное то, что у нас есть отдельные классы для каждого монстра. В наше время так игровые движки писать не принято.

Большинство из нас не раз убеждались на собственном опыте, что поддержка такой организации иерархии классов крайне болезненна, поэтому вместо этого для моделирования различных сущностей без отведения под каждую отдельного класса мы используем шаблоны наподобие Компонент (Component) или Тип объекта (Type Object).

Функции спаунера

Даже если у нас для каждого типа монстра имеется свой класс, есть другой способ "поймать кота". Вместо того, чтобы делать отдельный *класс* спаунер для каждого монстра, можно организовать *функцию* спаунер:

```
Monster* spawnGhost()
{
  return new Ghost();
}
```

Это уже не настолько примитивный подход, как создание отдельного класса для каждого нового типа монстров. Теперь единственный класс-спаунер может просто хранить указатель на функцию:

```
typedef Monster* (*SpawnCallback)();

class Spawner
{
public:
  Spawner(SpawnCallback spawn)
    : spawn_(spawn)
  {}

  Monster* spawnMonster() {
    return spawn_();
  }

private:
  SpawnCallback spawn_;
};
```

И для создания спаунера призраков нужно будет всего лишь вызвать:

```
Spawner* ghostSpawner = new Spawner(spawnGhost);
```

Шаблоны (Templates)

Сейчас большинство `C++` разработчиков знакомы с концепцией шаблонов. Нашему классу спаунеру нужно создать экземпляр определенного класса, но мы не хотим жестко прописывать в коде определенный класс монстра. Естественным решением этой задачи будет воспользоваться возможностями шаблонов и добавить параметр типа:

```
class Spawner
{
public:
  virtual ~Spawner() {}
  virtual Monster* spawnMonster() = 0;
};

template <class T>
class SpawnerFor : public Spawner
{
public:
  virtual Monster* spawnMonster() { return new T(); }
};
```

Я не могу утверждать, что программисты `C++` научились их любить или что некоторых они настолько пугают, что люди просто отказываются от `C++`. В любом случае, все кто сегодня использует `C++`, используют и шаблоны тоже.

Применение выглядит следующим образом:

```
Spawner* ghostSpawner = new SpawnerFor<Ghost>();
```

> Класс `Spawner` в данном коде не интересуется, какой тип монстра он будет создавать. Он просто работает с указателем на `Monster`.
>
> Если бы у нас был только класс `SpawnerFor<T>`, у нас не было бы ни одного экземпляра супертипа, разделяемого между шаблонами так что любому коду, работающему со спаунерами разных типов монстров, тоже пришлось бы принимать в качестве параметров шаблоны.

Класс первого типа

Предыдущие два решения требовали от нас иметь класс `Spawner`, параметризируемый типом. В `C++` классы в общем не являются объектами первого класса, так что это требует некоторых усилий. А вот если вы используете язык с динамическими типами наподобие `JavaScript`, `Python` или `Ruby`, где классы — это просто обычные объекты, которые можно как угодно передать, задача решается гораздо проще.

> В некотором роде шаблон Объект тип (Type Object) — это очередной способ обхода проблемы отсутствия класса первого типа. В языке с таким типом он тоже может быт полезен, потому что позволяет вам самостоятельно определять что такое "тип". Вам может пригодится семантика отличная от той, что предоставляют встроенные классы.

Если вам нужно соорудить спаунер — просто передайте ему класс монстра, которых он должен клонировать, т.е. по сути обычный объект, представляющий класс монстра. Проще пареной репы.

Имея столько возможностей, я не могу припомнить случай, в котором *паттерн проектирования* Прототип был бы лучшим вариантом. Может ваш опыт немного отличается от моего, но давайте лучше перейдем к следующей теме: прототипу как *языковой парадигме*.

Прототип, как языковая парадигма

Многие думают, что "объектно-ориентированное программирование" — это синоним слова "классы". Определения `ООП` напоминают кредо совершенно противоположных религий. Единственным бесспорным фактом является признание того факта, что *ООП позволяет вам определять "объект", объединяющий данные и код в единое целое*. По

сравнению со структурированными языками наподобие `C` и функциональными языками типа `Scheme`, ключевой особенностью `ООП` является способность связки состояния и поведения.

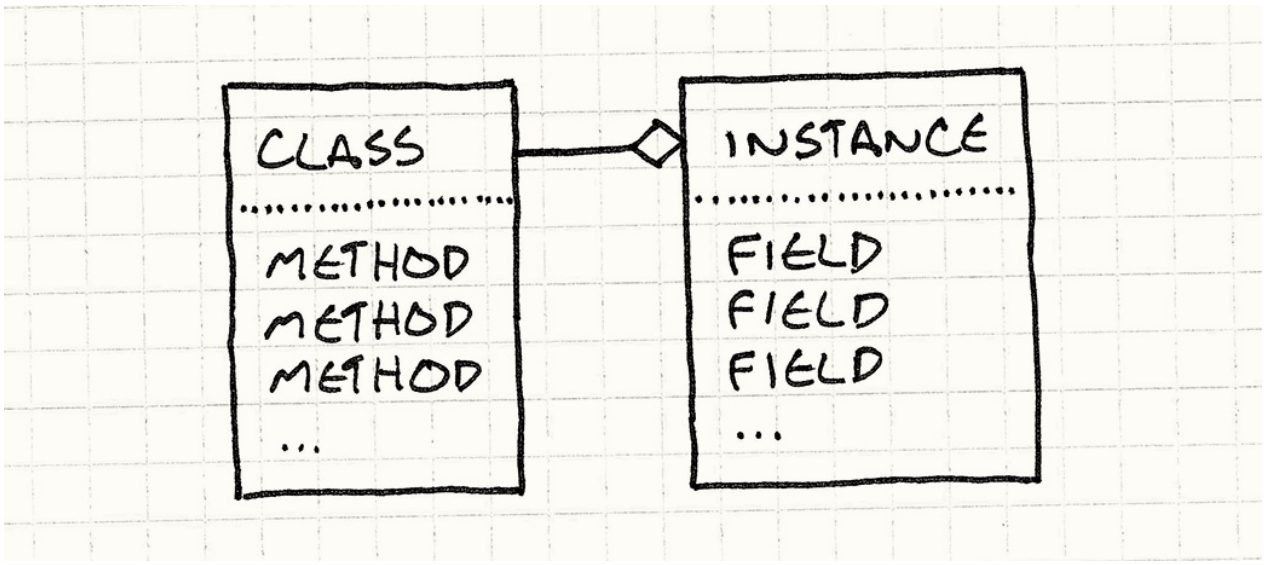
Вам может показаться что единственным способом это осуществить является использование классов, но некоторые люди, включая Дейва Унгара и Ренделла Смита думают иначе. Еще в 80-е они создали язык `Self`. Несмотря на то, что это `ООП` язык, классов в нем нет.

Self

На самом деле `Self` даже *более* объектно-ориентированный, чем языки с классами. Под `ООП` мы подразумеваем неразлучность состояния и поведения, а в языках с классами между ними на самом деле есть большое разделение.

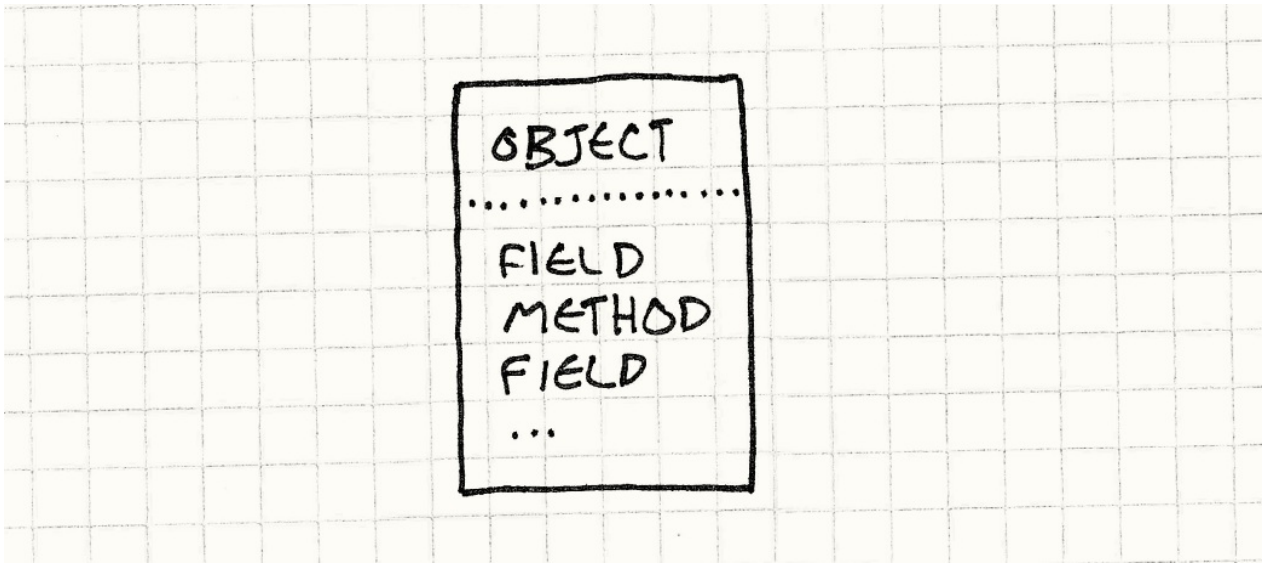
Вспомните семантику своего любимого языка с классами. Чтобы получить доступ к состоянию объекта, вы ищете в памяти его экземпляр. Состояние *содержится* в экземпляре.

Для вызова метода вы сначала ищете класс экземпляра и затем ищете метод *в нем*. Поведение содержится в *классе*. Всегда присутствует этот уровень косвенности для доступа к методу, отделяющий поля от методов.



> Например, чтобы вызвать виртуальный метод в `C++`, вы ищете его через указатель на экземпляр в виртуальной таблице и затем уже ищете в нем метод.

`Self` убирает это различие. Чтобы найти *что-угодно*, вы просто ищете это в объекте. Экземпляр может хранить как состояние так и поведение. Вы можете иметь отдельный объект с совершенно уникальным для него методом.

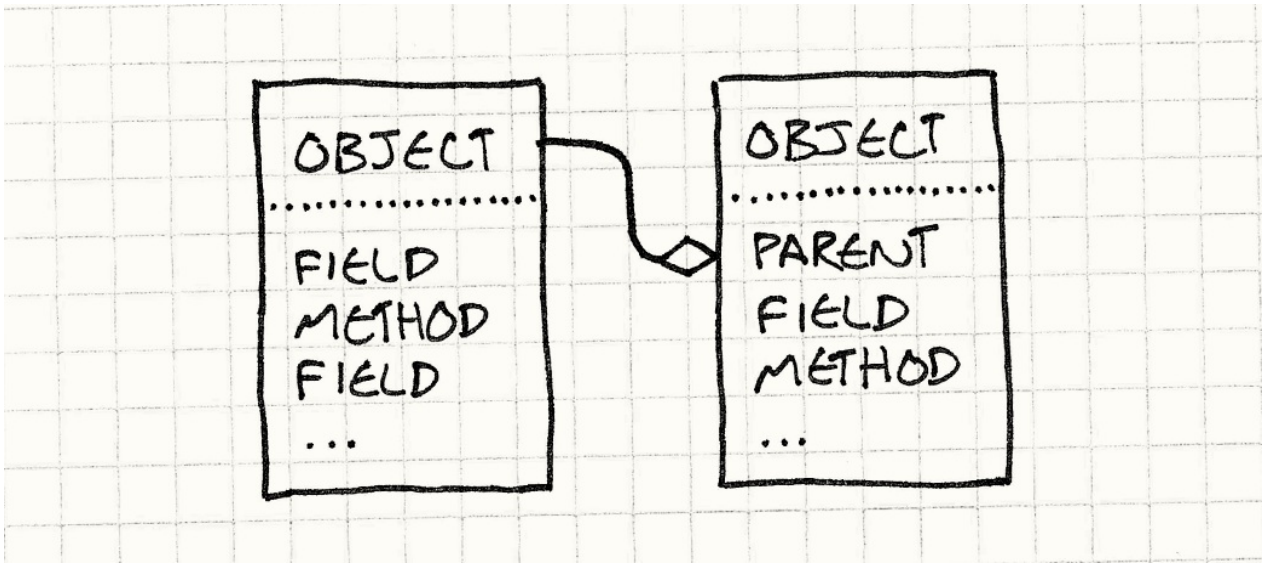


> Никто из людей не остров, кроме этого объекта (No man is an island, but this object is. Отсылка к сериалу Девочки Гилмор)

Если бы это было все, что делает `Self`, пользоваться им было бы довольно сложно. Наследование в языках с классами, несмотря на свои недостатки, дает вам удобный механизм для полиморфного повторного использования кода и избегания дублирования. Для получения подобных результатов в `Self` есть *делегирование*.

Чтобы получить доступ к полю или вызвать метод определенного объекта, мы сначала должны получить доступ к самому объекту. Если получилось — дальше все просто. Если нет — мы ищем *родителя* объекта. Это просто ссылка на другой объект. Если не удалось найти свойство у самого объекта, мы попробуем его родителя, и родителя родителя и т.д. Другими словами, неудавшийся поиск *делегируется* родителю объекта.

> Здесь допущено небольшое упрощение. `Self` помимо всего прочего поддерживает еще и несколько родительских объектов. Родители — это всего лишь специальным образом помеченные поля, дающие вам возможность использовать штуки типа наследования родителей или изменять их во время работы. Такой подход называется *динамическим наследованием* (dynamic inheritance).



Родительский объект дает нам возможность повторно использовать поведение (и состояние!) между несколькими объектами, так что мы уже перекрыли некоторую функциональность классов. Еще одна ключевая особенность классов заключается в том, что они позволяют нам создавать экземпляры классов. Когда вам нужен новый `ThingamaBob`, вы просто пишете `new Thingamabob()` ну или нечто подобное, если используете другой язык. Класс — это фабрика экземпляров самого себя.

Как можно создать нечто без класса? А как мы на самом деле делаем обычно новые вещи? Также, как и в рассмотренном нами шаблоне проектирования, `Self` делает это с помощью *клонирования*.

В `Self` *каждый* из объектов поддерживает шаблон проектирования Прототип автоматически. Любой объект можно клонировать. Чтобы наделать кучу одинаковых объектов нужно просто:

1. Привести один из объектов в нужное вас состояние. Можно просто взять за основу встроенный в систему базовый объект `Object` и дополнить его нужными полями и методами.
2. Клонировать его и получить столько... э-э... клонов, сколько вам нужно.

Таким образом, мы получаем элегантность шаблона Прототип, но без необходимости писать реализацию `clone()` для каждого класса самостоятельно. Он просто встроен в систему.

Это настолько прекрасная, разумная и минималистская система, что как только я узнал об этой парадигме, я сразу принялся за написание языка на основе прототипов, просто чтобы разобраться в парадигме получше.

> Я пришел к выводу, что написание языка с нуля — не лучший способ что-либо выучить, но это одна из моих странностей. Если вам любопытно, язык называется `Finch`.

И как оно?

Играться с языком на базе прототипов было замечательно, но как только мой собственный язык заработал, я обнаружил один малоутешительный факт: программировать на нем было не особо весело.

> С тех пор я часто слышу что многие программисты на `Self` приходят к тому же выводу. Впрочем это не означает, что проект был совсем провальный. `Self` был настолько динамичен, что для того, чтобы работать с нормальной скоростью, ему реально необходимы все современные инновации в области виртуализации.
>
> Изобретенные ими идеи относительно компиляции на ходу, сборщика мусора и оптимизации вызова методов — это именно те технологии, которые сделали (зачастую усилиями тех же самых людей) многие современные языки с динамическими типами достаточно быстрыми для того, чтобы писать на них популярные приложения.

Конечно язык был простым для реализации, но только потому, что я переложил всю сложность на плечи пользователя. Как только я начал пробовать им пользоваться, я обнаружил, что мне очень не хватает структурированности, которую дают классы. Я закончил тем, что стал пытаться компенсировать их отсутствие в самом языке написанием специальной библиотеки.

Возможно, все дело в том, что я слишком привык пользоваться языками с классами и мой мозг слишком привык к этой парадигме. Но у меня есть большое подозрение, что многим людям такой "порядок вещей" нравится.

И в продолжение истории ошеломительного успеха языков на основе классов. Посмотрите как много игр страдают от избытка классов персонажей, полного перечня различных типов врагов, предметов, навыков, каждый из которых старательно подписан. Не думаю, что вы найдете много игр, где каждый монстр представляет собой уникальную снежинку в духе "нечто среднее между троллем и гоблином и небольшой примесью змея".

Несмотря на то, что прототипы — это действительно очень мощная парадигма, и я хочу, чтобы об этом узнало как можно больше людей, я рад, что большинство из нас все таки не использует ее в повседневной работе. Потому что тот код с реализаций прототипов, что я видел, представлял из себя настолько ужасное месиво, что я так и не смог его понять.

> Также это говорит о том, что на самом деле существует очень *мало* кода, написанного в стиле прототипирования. Я смотрел.

А что насчет JavaScript?

Ну хорошо, если языки на основе прототипов настолько недружественны, то как я могу объяснить существование `JavaScript` ? Ведь это язык с прототипами, которым ежедневно пользуются миллионы людей. Код `JavaScript` выполняет больше компьютеров, чем код на любом другом языке в мире.

Брендан Эйх, создатель `JavaScript` , черпал вдохновение прямиком из `Self` и поэтому большая часть семантики `JavaScript` основана на прототипах. Каждый объект может иметь произвольный набор свойств, которые в свою очередь могут быть как полями, так и "методами" (которые на самом деле просто функции, хранящиеся в виде полей). У каждого объекта может быть другой объект, называемый его "прототипом", к которому происходит делегирование если нужное поле не найдено.

Для разработчика языка привлекательной особенностью прототипов является то, что реализовывать их легче, чем классы. Эйх тоже этим пользовался: первая версия `JavaScript` была написана всего за десять дней.

И все-таки, несмотря на все это, я считаю что на практике у `JavaScript` гораздо больше общего именно с языками на основе классов, чем с основанными на прототипах. Это заметно уже хотя бы потому, что в `JavaScript` предпринято большое отступление от `Self` — ключевой операции любого языка на основе прототипов — *клонирования* — нигде не видно. В `JavaScript` не существует метода для клонирования объекта.

Самая близкая по смыслу операция из существующих — это `Object.create()` , позволяющая вам создать новый объект, делегирующий к уже существующему. И даже эта возможность появилась только в спецификации `ECMAScript 5` , через четырнадцать лет после выхода `JavaScript` . Давайте я покажу, как обычно определяют типы и создают объекты в `JavaScript` вместо клонирования. Начинается все с *функции конструктора* (constructor function):

```
function Weapon(range, damage) {
  this.range = range;
  this.damage = damage;
}
```

С ее помощью создается новый объект и инициализируются его поля. Вызов выглядит следующим образом:

```
var sword = new Weapon(10, 16);
```

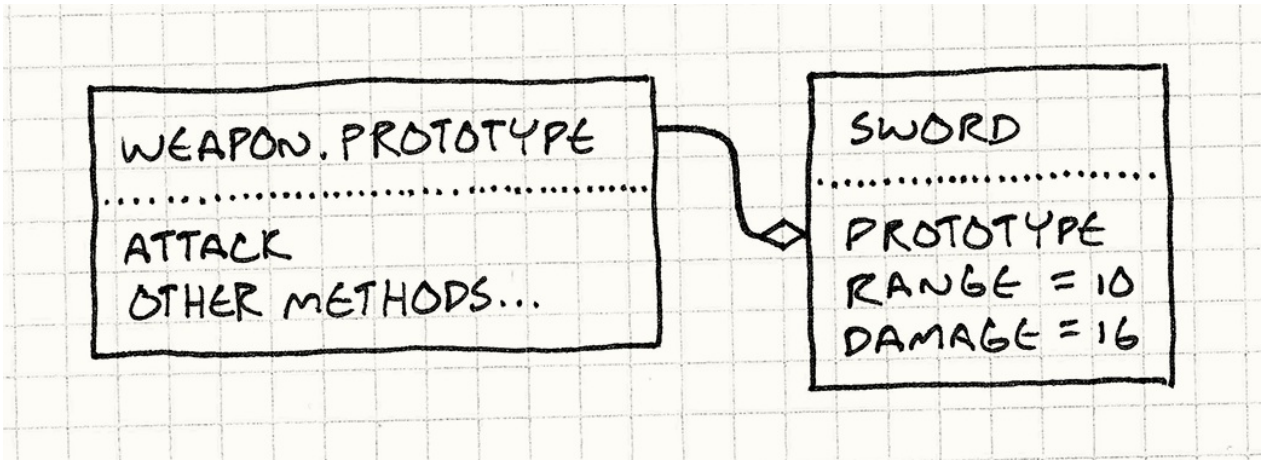
В этом коде `new` вызывает тело функции `Weapon()`, внутри которой `this` связано с новым пустым объектом. Внутри тела функции к объекту добавляется куча полей, а потом новосозданный объект автоматически возвращается.

Оператор `new` делает за вас еще одну вещь. Когда он создает чистый объект, он сразу делает его делегатом объекта-прототипа. Доступ к объекту прототипу можно получить через `Weapon.prototype`.

Так как состояние добавляется в теле конструктора, для определения *поведения* вы обычно добавляете методы к прототипу объекта. Примерно таким образом:

```
Weapon.prototype.attack = function(target) {
  if (distanceTo(target) > this.range) {
    console.log("Out of range!");
  } else {
    target.health -= this.damage;
  }
}
```

Здесь мы добавляем прототипу оружия свойство `attack`, значением которого будет функция. И так как каждый объект, возвращаемый `new Weapon()`, делегируется к `Weapon.prototype`, вы можете теперь сделать вызов `sword.attack()` и он вызовет нужную нам функцию. Выглядит это примерно так:



Давайте еще раз:

- Новые объекты вы создаете с помощью операнда `new`, который вы вызываете используя объект, представляющий собой тип — функцию-конструктор.
- Состояние хранится в самом экземпляре.

- Поведение задается через уровень косвенности — делегирование к прототипу и хранится в виде отдельного объекта, представляющего собой набор методов, разделяемый между всеми объектами данного типа.

Вы можете назвать меня психом, но это крайне похоже на мое определение классов, которое я привел выше. Вы *имеете* возможность писать код в стиле прототипов в `JavaScript` (без клонирования), но синтаксис и идиоматика языка предполагают подход, основанный на классах.

Я лично считаю, что это хорошо. Как я уже сказал, я убедился на собственном опыте, что прототипы усложняют работу с кодом, так что мне нравится то, как `JavaScript` оборачивает свое ядро в более похожую на классы форму.

Прототипы для моделирования данных

Итак, я продолжаю перечислять вещи, за которые я *не люблю* прототипы. Депрессивная глава получается. Я задумывал эту книгу скорее как комедию, а не как трагедию, так что покончим с этим и перейдем к областям, где *на мой взгляд* прототипы или, говоря конкретнее, *делегирование* может быть полезным.

Если вы посчитаете все байты в игре, приходящиеся на код и сравните с объемом остальных данных, вы увидите, что с момента появления игр, доля данных постоянно увеличивается. Ранние игры практически все генерировали процедурно и, как следствие, могли поместиться на дискетку или картридж. В большинстве современных игр код — это всего лишь "движок", который позволяет игре работать, а сама игра полностью определена в данных.

Это конечно здорово, но перемещение контента в файлы данных вовсе не означает, что мы избавляемся от организационных сложностей большого проекта. Скорее наоборот, усложняем себе жизнь. Одной из причин почему мы используем языки программирования является то, что они предоставляют нам инструменты по снижению сложности.

Вместо того, чтобы копировать и вставлять кусок кода в десяти местах, мы помещаем его в отдельную функцию и вызываем ее по имени. Вместо того, чтобы копировать метод в кучу классов, мы просто помещаем его в отдельный класс, а остальные классы от него наследуем.

Когда объем данных в игре достигает некоторого предела, вам сразу начинает хотеться обладать подобными возможностями. Моделирование данных — это слишком большая область, чтобы обсуждать ее на поверхностном уровне, но я хочу показать

вам одну из возможностей, которая пригодится вам в вашей игре: использование прототипов и делегирование для повторного использования данных.

Давайте представим себе, что мы определяем модель данных для бессовестного клона `Gauntlet`, о котором я писал выше. Геймдизайнеру нужны какие-то файлы, в которые он сможет поместить описание атрибутов монстров и предметов.

> Я имею в виду полностью оригинальную игру, никоим образом не напоминающую хорошо известную ранее многопользовательскую аркадную игру с видом сверху. Так что не подавайте на меня в суд пожалуйста.

Можно использовать `JSON`: сущности данных будут представлены в виде `maps` или `мешков со свойствами` (property bags) или еще дюжиной терминов, потому что программисты просто обожают придумывать для одного и того же разные имена.

> Мы так часто их переизобретаем, что Стив Йегге решил назвать их Универсальным шаблоном проектирования (“The Universal Design Pattern”).

Итак, гоблин в игре описан следующим образом:

```
{
  "name": "goblin grunt",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}
```

Довольно прямолинейный подход и даже не любящие писать дизайнеры могут справиться. Можно, например, добавить еще парочку сестринских описаний в славном семейном дереве зеленых гоблинов:

```
{
  "name": "goblin wizard",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"],
  "spells": ["fire ball", "lightning bolt"]
}
```

```
{
  "name": "goblin archer",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"],
  "attacks": ["short bow"]
}
```

Если бы это был обычный код, наше чувство прекрасного уже заставило бы нас беспокоиться. У этих сущностей слишком много общей дублирующейся информации, а хорошо натренированные программисты это просто *ненавидят*. Данные занимают слишком много места и требуют слишком много времени на написание. Даже для того, чтобы выяснить одинаковые ли *это* данные, вам нужно тщательно их прочитать. Их поддержка — настоящая головная боль. Если мы захотим сделать всех гоблинов в игре сильнее, нам нужно будет не забыть обновить значение здоровья для них всех. Плохо, плохо, плохо.

Если бы это был код, мы могли бы создать абстракцию "гоблин" и использовать ее между всему типами гоблинов. Но тупой `JSON` ничего об этом не знает. Давайте попробуем сделать его чуточку умнее.

Определим для каждого объекта поле "`prototype`" и поместим туда имя объекта, к которому он делегирует. Любые свойства, отсутствующие у первого объекта нужно будет смотреть в прототипе.

Это позволит нам упростить описание нашей оравы гоблинов:

> Таким образом "`prototype`" переходит из разряда обычных данных в метаданные. У каждого гоблина есть бородавчатая кожа и желтые зубы. У него нет прототипа. Прототип — это свойство *объекта данных, описывающего гоблина*, а не самого гоблина.

```
{
  "name": "goblin grunt",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}
```

```
{
  "name": "goblin wizard",
  "prototype": "goblin grunt",
  "spells": ["fire ball", "lightning bolt"]
}
```

```
{
  "name": "goblin archer",
  "prototype": "goblin grunt",
  "attacks": ["short bow"]
}
```

Так как и лучник, и чародей имеют в качестве прототипа пехотинца, нам не нужно указывать заново здоровье, сопротивляемости и уязвимости для каждого из них. Добавленная нами в данные логика предельно проста — мы просто добавили простейшее делегирование и сразу смогли избавиться от кучи повторов.

Хочу обратить ваше внимание на то, что мы не стали добавлять четвертого "базового гоблина" в качестве *абстрактного* прототипа, к которому будут делегировать остальные три. Вместо этого, мы просто взяли одного из гоблинов, который является простейшим и делегируем к нему.

Такой подход является естественным для систем на основе прототипов, где каждый объект можно использовать для клонирования нового объекта с уточненными свойствами и смотрится натуральным и здесь. Применительно к игровым данным такой подход тоже удобен, потому что здесь часто приходится создавать объекты, лишь немного отличающиеся от остальных.

Подумайте о боссах и уникальных предметах. Очень часто они являются лишь немного измененной версией обыкновенных игровых объектов и прототипирование с делегированием очень хорошо подходит для их описания. Магический `Sword of Head-Detaching` (Меч-голова с плеч) можно описать как длинный меч с определенными бонусами:

```
{
  "name": "Sword of Head-Detaching",
  "prototype": "longsword",
  "damageBonus": "20"
}
```

Такие дополнительные возможности для описания данных могут облегчить жизнь вашим дизайнерам и добавить больше вариативности предметам и популяции монстров в игре, а это именно то, что может понравиться игрокам.

Синглтон (Singleton)

Эта глава — настоящая аномалия. Все остальные главы демонстрируют как нужно использовать шаблоны. Эта глава показывает как их использовать не нужно.

Несмотря на благородные намерения, с которыми шаблон Синглтон (Singleton pattern)[GoF] создавался Бандой Четырех, обычно вреда от него больше, чем пользы. Несмотря на их призыв не злоупотреблять этим шаблоном, это послание как-то потерялось, прежде чем попало в игровую индустрию.

Как и в случае с любым другим шаблоном, использование его в неподходящем месте также уместно как лечение пулевого ранения пластырем. Так как используется он слишком часто, большая часть главы будет посвящена тому, где стоит *избегать* использование синглтона. Но для начала познакомимся с самим шаблоном.

> Когда большая часть индустрии перебралась с `C` на объектно-ориентированное программирование, основной проблемой была "как мне получить доступ к экземпляру?". Нужно было вызвать определенный метод, но не было экземпляра объекта, который реализовывал этот метод. Решением был Синглтон (или другими словами использование глобальной области видимости). (У этого шаблона есть еще одно название — Одиночка)).

Шаблон Синглтон

Паттерны программирования определяют Синглтон следующим образом:

> Обеспечивает существование единственного экземпляра класса и обеспечивает глобальный доступ к нему.

Давайте проведем разделение на этом "и" и рассмотрим обе половины по отдельности.

Ограничение экземпляров класса до одного

Иногда класс может работать корректно только, если существует его единственный экземпляр. Зачастую, это необходимо, если класс взаимодействует с внешней системой, которая поддерживает собственное глобальное состояние.

Представим себе класс-обертку над низкоуровневым `API` файловой системы. Так как файловые операции занимают определенное время, наш класс выполняет операции асинхронно. Это значит, что несколько операций могут выполняться конкурентно и

следовательно им нужно координировать свою работу. Если мы начнем с вызова для создания файла и затем попытаемся удалить тот же самый файл, наш класс-обертка должен знать об обеих вызовах, чтобы они друг другу не мешали.

Чтобы это сделать, наш класс-обертка должен иметь доступ ко всем предыдущим операциям. Если пользователь может свободно создавать любое количество экземпляров этого класса, каждый из этих экземпляров не будет значить ничего об операциях, выполняемых другими экземплярами. Вот здесь то и нужен Синглтон. Он позволяет обеспечить еще на этапе компиляции возможность существования только одного экземпляра класса.

Обеспечение глобальной области видимости

Наш класс-оболочку над файловой системой использует несколько других систем в игре: система отчетов, загрузка контента, сохранение состояния игры и т.д. Если эти системы не смогут создать собственный экземпляр обертки над файловой системой, то как они смогут ее использовать?

Синглтон решает и эту проблему. Он не только создает единственный экземпляр класса, но и делает его глобально видимым. Таким образом кто угодно и где угодно может наложить на него свою лапу. Целиком классическая реализация выглядит следующим образом:

```
class FileSystem
{
public:
  static FileSystem& instance() {
    // Ленивая инициализация.
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

Статический член `instance_` хранит экземпляр класса, а приватный конструктор обеспечивает то что этот экземпляр единственный. Публичный и статический метод `instance()` предоставляет доступ к экземпляру для всей остальной кодовой базы. А еще он отвечает за создание экземпляра синглтона методом ленивой инициализации, т.е. в момент первого вызова.

Современный вариант выглядит следующим образом:

```
class FileSystem
{
public:
  static FileSystem& instance() {
    static FileSystem *instance = new FileSystem();
    return *instance;
  }

private:
  FileSystem() {}
};
```

`C++11` гарантирует, что инициализация локальной статической переменной происходит только один раз, даже в случае конкурентного доступа. Поэтому, при условии что у вас есть поддерживающий `C++11` компилятор, такой код является потоково-безопасным. А вот первый пример — нет.

> Конечно потоковая безопасность самого вашего класса синглтона — это совсем другое дело! Мое замечание касается только его *инициализации*.

Зачем мы его используем

Похоже у нас есть победитель. Наша обертка над файловой системой доступна всегда и везде и ее не нужно передавать с помощью каких либо ухищрений. Сам класс достаточно умен для того чтобы не позволить нам по ошибке устроить месиво из кучи его экземпляров. А вот и еще несколько приятных особенностей:

- **Экземпляр не будет создан если его никто не захочет использовать.** Экономия памяти и циклов процессора — это всегда хорошо. Благодаря тому что синглтон инициализируется при первом вызове, его экземпляр не создастся если никто в игре к нему не обратится.
- **Инициализация во время выполнения.** Очевидной альтернативой Синглтону является класс ос статическими переменными членами. Мне нравятся простые решения, да и использование статических классов вместо синглтона возможно. Однако у статических членов есть одно ограничение: автоматическая инициализация. Компилятор инициализирует статические переменные до вызова `main()`. Это значит, что они не могут использовать информацию, которая будет известна только после того как программа запустится и начнет работать (например, когда будет загружена файл настроек). А еще это значит, что они не могут полагаться друг на друга — компилятор не гарантирует очередности в которой относительно друг друга статические переменные будут инициализированы.

Ленивая инициализация решает обе эти проблемы. Синглтон будет инициализирован настолько поздно, насколько возможно, так что ко времени его создания нужная информация уже будет загружена. И если это не приводит к циклической зависимости, один синглтон может ссылаться при инициализации на другой.

- **У вас может быть подкласс синглтон.** Это очень мощная, но редко используемая возможность. Скажем например, что мы хотим сделать свою обертку над файловой системой кросс-платформенной. Чтобы это заработало нам нужно сделать интерфейс файловой системы абстрактным с подклассами, которые будут реализовывать интерфейсы для каждой платформы. Вот эти базовые классы.

```
class FileSystem
{
public:
  virtual ~FileSystem() {}
  virtual char* readFile(char* path) = 0;
  virtual void writeFile(char* path, char* contents) = 0;
};

class PS3FileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path) {
    // Файловая система Sony IO API...
  }

  virtual void writeFile(char* path, char* contents) {
    // Файловая система Sony IO API...
  }
};
```

```
class WiiFileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path) {
    // Файловая система Nintendo IO API...
  }

  virtual void writeFile(char* path, char* contents) {
    // Файловая система Nintendo IO API...
  }
};
```

А теперь превращаем `FileSystem` в синглтон:

```
class FileSystem
{
public:
  static FileSystem& instance();

  virtual ~FileSystem() {}
  virtual char* readFile(char* path) = 0;
  virtual void writeFile(char* path, char* contents) = 0;

protected:
  FileSystem() {}
};
```

Хитрость здесь заключается в создании экземпляра:

```
FileSystem& FileSystem::instance()
{
  #if PLATFORM == PLAYSTATION3
    static FileSystem *instance = new PS3FileSystem();
  #elif PLATFORM == WII
    static FileSystem *instance = new WiiFileSystem();
  #endif

  return *instance;
}
```

С помощью простого переключателя компилятора мы связываем обертку файловой системы с конкретным типом. И теперь вся наша кодовая база может получать доступ к файловой системе через `FileSystem::instance()` не привязываясь ни к какому платформозависимому коду. Вместо этого получившаяся связность инкапсулируется внутри реализации самого класса `FileSystem`.

Итак мы подошли к тому пределу, до которого мы обычно доходим при решении подобных проблем. У нас есть обертка вокруг файловой системы. Она работает. Она доступна глобально в любом месте где может понадобиться. Пришло время проверить код, налить себе любимый напиток и начать праздновать.

Почему мы можем пожалеть, что стали его использовать

В близкой перспективе шаблон Синглтон довольно дружественен. Как и в случае со многиими другими архитектурными решениями, платить придется в долгой перспективе. Как только мы добавим в замороженный код несколько лишних синглтонов, у нас начут появлятся новые проблемы:

Это глобальная переменная

Когда игры еще писались несколькими парнями в гаражах, борьба с железом была гораздо важнее, чем башни из слоновой кости программной архитектуры. Кодеры старой школы на `C` и `assembler` свободно использовали глобальные и статические переменные и создавали отличные игры. Но, как только игры стали больше и сложнее, архитектура и сложность ее поддержки стали превращаться в бутылочное горлышко. Создавать игры стало сложнее не из-за аппаратных ограничений, а из-за производственных ограничений.

Мы перебрались на языки наподобие `C++` и стали постигать мудрость, накопленную нашими программистами-предшесвтенниками. Важнейшим усвоенным уроком насчет глобальных переменных было то, что использовать глобальные переменные — плохо. По нескольким причинам:

- **Они делают код менее понятным.** Допустим мы ищем баг в функции, написанной кем-то другим. Если функция не затрагивает никаких глобальных состояний, мы можем ограничиться рамками функции и просто изучить какие аргументы она получает.

  > Компьютерная наука называет функции, не получающие доступ и не изменяющие глобальные состояния "чистыми (pure)" функциями. Чистые функции легче понять, легче оптимизировать на этапе компиляции и с ними можно делать такие удобные вещи как изоляция памяти, когда мы можем кешировать и повторно использовать результаты выполнения таких функций.
  >
  > И хотя на практике не всегда возможно соблюдать истинную чистоту, преимущества этих функций настолько соблазнительны, что существуют даже целые язык типа `Haskel`, где разрешены только чистые функции.

  А теперь представьте что прямо в середине функции происходит обращение к `SomeClass::getSomeGlobalData()`. Теперь для того, чтобы понять что происходит, нам нужно прошерстить всю кодовую базу и выяснить, кто еще работает с этими глобальными данными. И у вас конечно нет причин ненавидеть глобальные переменные, пока однажды вам не придется просмотреть миллион строк кода в три часа утра в поисках того глючного вызова, который все таки записывает в статическую глобальную переменную некорректное значение.

- **Они усиливают связность.** Новый программист в вашей команде конечно еще не знаком с вашей прекрасной, легко поддерживаемой архитектурой игры, но ему нужно выполнить задание: добавить проигрывание звуков, когда камни падают на землю. Мы с вами хорошо понимаем, что нам нужно любой ценой не допускать

лишних связей между физической и аудио подсистемами, но новичок просто хочет выполнить свое задание. К нашему несчастью экземпляр `AudioPlayer` имеет глобальную область видимости. И вот после добавления всего одного `#include` вся ранее возведенная архитектура рушится.

Если бы экземпляр аудио плеера не был бы объявлен глобальным, добавление `#include` с его заголовочным файлом так ничего бы и не дало. Этот факт сам по себе четко сказал бы новичку, что эти модули не должны ничего знать друг о другие и ему нужно найти другой способ решения проблемы. *Управляя доступом к экземпляру вы управляете связностью*.

- **Они не конкурентно-дружественны.** Деньки, когда игра работала на одноядерном процессоре уже сочтены. Современный код должен по крайней мере корректно работать в многопоточной системе, даже если он не использует все ее преимущества. Когда мы делаем что либо глобальным, у нас образуется область памяти, видимая всеми потоками. И каждый поток может к ней обратится не зная, что возможно с этой памятью уже работает кто-то еще. Это приводит к блокировкам (deadlocks), гонкам за доступ (race conditions) и другим страшным багам синхронизации.

Все эти ужасы должны отпугнуть нас от использования глобальных переменных в целом и шаблона Синглтон в частности, но они никак не приближают нас к понимают того, как нам *нужно* проектировать игру. Как строить архитектуру игры без глобального состояния?

На этот вопрос можно отвечать только развернуто (большая часть книги как раз этому и посвящена) и этот ответ совсем не тривиален и не очевиден. А еще нам ведь нужно выпустить игру. Шаблон Синглтон выглядит как панацея. А у нас книга об объектно-ориентированных шаблонах проектирования. Так что он *должен* казаться вполне архитектурным, верно? Ведь он позволяет нам проектировать программы также как и десятилетия раньше.

К сожалению, это скорее плацебо, а не лекарство. Если вы еще раз просмотрите список проблем, порождаемых глобальными переменными, вы заметите, что Синглтон ни одну из них не решает. А все потому, что синглтон *всего лишь* глобальное состояние, инкапсулированное внутри класса.

Он решает две проблемы, даже если у вас всего одна

Слово "и" в описании Синглтона от Банды Четырех выглядит немного странным. Решает этот шаблон одну или сразу две проблемы? Что, если у нас только одна из этих проблем? Обеспечение наличия всего одной копии экземпляра может быть полезно,

но кто сказал, что мы хотим, чтобы кто угодно мог пользоваться этим экземпляром? И наоборот, глобальная видимость может быть полезной, но хочется иметь возможность иметь множество экземпляров.

Вторая из проблем — удобство доступа — обычно и побуждает нас к использованию шаблона Синглтон. Представим себе класс журналирования — логгер. Большинство модулей в игре только выиграют от возможности легко добавлять в лог диагностическую информацию. В то же время передача единственного экземпляра логгера в каждый класс будет только перегружать код и отвлекать от его основного назначения.

Очевидным решением является преобразования класса `Log` в синглтон. В результате каждая функция может получить экземпляр класса напрямую. Но в то же самое время у нас появляется новое ограничение. Внезапно мы лишаемся возможности иметь больше одного класса логгера.

По началу это и не проблемы вовсе. Мы пишем всего в один лог файл и нам все-равно не нужно больше одного экземпляра. А дальше, в процессе разработки у нас появляются новые трудности. Каждый член команды использует логгер в своих нуждах и вскоре лог файл превращается в месиво данных. Программистам приходится пролистывать полотна сообщений, чтобы найти интересующее.

Мы могли бы решить эту проблему добавив возможность писать в несколько лог файлов. Для этого можно завести отдельный логгер для всех областей игры: сети, пользовательского интерфейса, аудиоподстистемы, геймплея. Но мы не можем этого сделать. Класс `Log` не только не позволяет нам больше иметь несколько экземпляров класса, но и налагает такое ограничение на все свои вызовы:

```
Log::instance().write("Some event.");
```

Теперь, чтобы вернуть себе возможность иметь одновременно несколько экземпляров логгера (как было изначально), им нужно изменить не только сам класс, но и все участки кода, где он используется. Наш удобный доступ перестал быть удобным.

> Может возникнуть ситуация и похуже. Представьте себе, что класс `Log` находится в библиотеке, общей для *нескольких* игр. И теперь, чтобы изменить его архитектуру, нам нужно скоординировать изменение между несколькими группами людей, у большинства из которых нет ни времени ни мотивации для подобных изменений.

Ленивая инициализация отнимает у вас контроль над происходящим

В мире настольных `PC`, где полно виртуальной памяти и мягкие системные требования, ленивая инициализация — благо. Игры — это другое дело. Инициализация системы требует времени: на выделение памяти, на загрузку ресурсов и т.д. Если инициализация аудио подсистемы требует несколько сотен миллисекунд, нам нужно иметь возможность контролировать, когда эта инициализация произойдет. Если мы позволим ей лениво инициализироваться при первом проигрывании звука — это произойти в середине напряженной игры и вызовет неслабое падение `FPS` и заикание геймплея.

Кроме этого, играм обычно требуется надежный контроль размещения объектов в куче чтобы избежать фрагментации памяти. Если аудио система выделяет при инициализации область в памяти из кучи, нам нужно понимать *когда* произойдет инициализация, чтобы проконтролировать, *где* в куче будет размещена эта область.

> Подробнее про фрагментацию памяти написано в главе Пул Объектов (Object Pool).

Из-за двух этих проблем, большинство игр, которые я видел, не полагаются на ленивую инициализацию. Вместо этого они реализуют Синглтон следующим образом:

```
class FileSystem
{
public:
  static FileSystem& instance() { return instance_; }

private:
  FileSystem() {}

  static FileSystem instance_;
};
```

Таким образом, мы решаем проблему ленивой инициализации, но делаем это за счет потери некоторых особенностей синглтона, которые делают его лучше, чем простая глобальная переменная. После добавление статического экземпляра мы больше не можем использовать полиморфизм и класс должен стать конструируемым во время статической инициализации. И еще мы теперь не можем освободить память, занимаемую экземпляром когда он станет не нужным.

Вместо создания синглтона, все что мы получили — это простой статический класс. Это не обязательно плохо, но если нам нужен просто статический класс, то почему бы вообще не избавиться от метода `instance()` и пользоваться статической функцией напрямую? Вызов `Foo::bar()` проще чем `Foo::instance().bar()` и к тому же яснее показывает что мы имеем дело со статической памятью.

> Обычный аргумент для выбора синглтона над статическими классами является то, что если вы решите изменить статический класс в нестатический позже, вам нужно исправить все места вызова. В теории, вы не должны делать это с синглетоном, потому что вы могли бы передавая экземпляр вокруг и назвав его как обычный метод экземпляра.
>
> Обычным аргументом в пользу выбора синглтона вместо статического класса является то, что для последующей замены на нестатический класс вам придется менять каждый вызов класса в коде. Теоретически, при использовании синглтона вам это делать не придется, потому что в него можно передать экземпляр класса извне и обращаться через него к обычному методу экземпляра.
>
> На практике я не припомню, чтобы когда либо видел нечто подобное. Все просто записывают в одну строку `Foo::instance().bar()`. И если потом вы сделаете Foo не синглтоном, нам всеравно придется править каждый вызов. Поэтому лично я предпочту и класс попроще и вызовы с более простым синтаксисом.

Что можно сделать вместо этого

Если я хорошо справился со своей задачей, вы теперь дважды подумаете, прежде чем в следующий раз вытащить Синглтон из ящика с инструментами для решения своей проблемы. Но у вас по прежнему осталась неразрешенная проблема. А каким же инструментом тогда следует пользоваться? В зависимости от того, что вам нужно сделать, у меня есть для вас несколько вариантов, но сначала...

Подумайте нужен ли вам класс вообще

Большинство синглтонов, которые я видел в играх были "менеджерами": это были просто такие туманные классы, созданные только для того, чтобы нянчиться с другими объектами. Я помню кодовые базы, где практически у каждого класса был свой менеджер: Монстр, Менеджер монстров, Частица, Менеджер частиц, Звук, Менеджер звуков, Менеджер менеджеров. Иногда в имени встречались слова "система" или "движок", но сама суть не менялась.

Иногда классы смотрители конечно полезны, но зачастую они просто демонстрируют незнание `ООП`. Полюбуйтесь на эти два надуманных класса:

```
class Bullet
{
public:
  int getX() const { return x_; }
  int getY() const { return y_; }

  void setX(int x) { x_ = x; }
  void setY(int y) { y_ = y; }

private:
  int x_, y_;
};

class BulletManager
{
public:
  Bullet* create(int x, int y) {
    Bullet* bullet = new Bullet();
    bullet->setX(x);
    bullet->setY(y);

    return bullet;
  }
```

```
  bool isOnScreen(Bullet& bullet) {
    return  bullet.getX() >= 0 &&
            bullet.getX() < SCREEN_WIDTH &&
            bullet.getY() >= 0 &&
            bullet.getY() < SCREEN_HEIGHT;
  }

  void move(Bullet& bullet) {
    bullet.setX(bullet.getX() + 5);
  }
};
```

Может этот пример и довольно туп, но я видел достаточно много кода, архитектурная сущность которого после небольшого упрощения сводилась к тому же. Если посмотреть на код выше, вполне логично предположить что `BulletManager` можно сделать синглтоном. В конце концов все где встречаются пули, будет нуждаться и в их менеджере, так что сколько еще экземпляров менеджера нам нужно?

Правильный ответ — *нисколько*. И вот как мы решим проблему "синглтона" для нашего класса менеджера:

```
class Bullet
{
public:
  Bullet(int x, int y) : x_(x), y_(y) {}

  bool isOnScreen() {
    return  x_ >= 0 && x_ < SCREEN_WIDTH &&
            y_ >= 0 && y_ < SCREEN_HEIGHT;
  }

  void move() { x_ += 5; }

private:
  int x_, y_;
};
```

Вот и все. Нет менеджера — нет проблемы. Плохо спроектированные синглтоны — это чаще всего просто "вспомогательные" классы, добавляющие функциональность к другим классам. Везде где это возможно просто переносите поведение внутрь самого класса. В конце концов ООП именно о том чтобы позволить классу заботиться о себе самостоятельно.

Вне менеджера, тем не менее, могут существовать и другие проблемы, которые можно решить синглтоном. И для каждой из этих проблем есть альтернативные решения, которые нужно иметь в виду.

Ограничение класса единственным экземпляром

Это ровно половина того, что дает нам шаблон Синглтон. И как в нашем примере с файловой системой, такое ограничение на наличие единственного экземпляра класса может быть критичны. Однако из этого совсем не следует, что мы обязательно желаем предоставлять *публичный*, *глобальный* доступ к экземпляру. Нам может быть удобнее ограничить доступ к некоторым областям кода или даже сделать его приватным для отдельного класса. В таком случае предоставление глобального доступа только ослабляет архитектуру.

> Например, нам может понадобиться поместить нашу обертку над файловой системой внутрь *другой* абстракции.

Нам может понадобиться обеспечить существование единственного экземпляра без предоставления глобального доступа. Есть несколько вариантов решения. Вот один из них:

```
class FileSystem
{
public:
  FileSystem() {
    assert(!instantiated_);
    instantiated_ = true;
  }

  ~FileSystem() { instantiated_ = false; }

private:
  static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

Класс позволяет кому угодно вызвать его конструктор, но по прежнему содержит проверку и выдает ошибку если вы попытаетесь создать второй экземпляр. И когда правильный код создаст экземпляр первым, мы можем быть уверены что никто больше другой экземпляр не создаст и доступа к нему не получит. Класс обеспечивает существование себя в единственном экземпляре, но не навязывает нам того как с ним нужно обращаться.

Недостатком этого примера является то, что проверка на существование экземпляра выполняется во *время выполнения*. А вот шаблон Синглтон самой своей сущностью гарантирует существование только одного экземпляра класс уже на этапе компиляции.

Удобство доступа к экземпляру

Удобство доступа — одна из причин, почему мы выбираем синглтон. С его помощью очень просто получить доступ к объекту, который может понадобиться нам в самых различных местах. К сожалению эта простота имеет свою цену: доступ к объекту слишком просто получить даже оттуда, откуда нам *не* хотелось бы.

Общее правило такое: нам желательна как можно более узкая видимость переменных, но без ущерба для функциональности. Чем уже видимость объекта, тем реже нам нужно задумываться о его влиянии на другие части кода в работе. Прежде чем применять подход дробовика в виде объекта синглтона с *глобальной* областью видимости, давайте посмотрим какие еще варианты доступа кодовой базы к объекту у нас есть:

> Подход дробовика — стратегия в маркетинге (противоположность подхода винтовки), заключается в покрытии максимальной аудитории.

- **Прямая передача.** Простейшее решение и в некоторых случаях самое лучшее. Мы просто передаем объект в качестве аргумента в функцию, которая в нем нуждается. Не стоит сгоряча отбрасывать этот подход за его громоздкость.

  > Некоторые используют для описания этого способа термин "инъекция зависимости (dependency injection)". Вместо того, чтобы позволить коду выбираться наружу и становиться зависимым от каких либо глобальных вызовов, зависимости передаются внутрь кода через параметр. А еще иногда используется инверсная "инъекция зависимости" для более сложных зависимостей кода.

  Представьте себе функцию рендеринга объекта. Чтобы отрендерить объект, нужно получить доступ к объекту, представляющему собой устройство видеовывода и поддерживающему состояния рендера. Очень часто все это передается целиком в функции рендеринга через параметр с именем `context` или подобным.

  С другой стороны, некоторые объекты не соответствуют контексту метода. Например, функции управляющей `AI` полезно иметь возможность писать в лог, но это совсем не основная ее задача. Поэтому передача в эту функцию объекта `Log` в качестве аргумента будет смотреться по крайней мере странно и в таком случае лучше выбрать другой вариант.

  > Вещи наподобие логгинга, раскиданные по всей кодовой базе тоже обозначаются своим термином "сквозное связывание(cross-cutting concern)". Борьба со сквозным связыванием встречается в архитектуре постоянно, особенно в статически типизированных языках.
  >
  > Аспектно-ориентированное программирование (Aspect-oriented programming) разработано как раз для такой концепции.

- **Получение из базового класса.** Во многих играх архитектура представляет собой неглубокую, но достаточно ветвистую иерархию. Зачастую всего с одним уровнем наследования. Например, у вас может быть базовый класс `GameObject`, от которого наследуются классы для каждого врага или объекта в игре. При такой архитектуре большая часть игрового кода обитает в "листьях" унаследованного класса. Это значит что у всех классов есть доступ к одному и тому же базовому классу `GameObject`. Мы можем воспользоваться этим преимуществом:

```
class GameObject
{
protected:
  Log& getLog() { return log_; }

private:
  static Log& log_;
};

class Enemy : public GameObject
{
  void doSomething() {
    getLog().write("I can log!");
  }
};
```

При этом никто за пределами `GameObject` не может получить доступ к его объекту `Log`, а любая другая унаследованная сущность могут, с помощью `getLog()`. Этот шаблон позволяет полученным объектам реализовывать себя в терминах защищенных методов, которые описаны в главе подкласс Песочница (Subclass Sandbox).

> Возникает вопрос "Как `GameObject` может получить экземпляр `Log`?" Проще всего будет иметь базовый класс и создавать его статический экземпляр.
>
> Если же вы не хотите, чтобы ваш базовый класс играл такую активную роль, вы можете использовать функцию инициализации, передав туда экземпляр `Log`, который будет использовать наш `GameObject`. Или использовать шаблон Локатор службы (Service Locator).

- **Получить через другой объект, который уже является глобальным.** Цель — убрать вообще все глобальные состояния, конечно похвальна, но навряд ли практична. Большинство кодовых баз обязательно имеет хотя бы несколько глобальных объектов, например объекты `Game` или `World`, представляющие общее состояние игры.

  Вы можете обратить это себе на пользу и уменьшить количество глобальных объектов, нагрузив уже существующие. Вместо того чтобы делать синглтон из `Log`, `FileSystem` и `AudioPlayer` можно поступить таким образом:

```
class World
{
public:
  static World& instance() { return instance_; }

  // Функции для указания log_, и всего остального ...

  Log& getLog() { return *log_; }
  FileSystem& getFileSystem() { return *fileSystem_; }
  AudioPlayer& getAudioPlayer() { return *audioPlayer_; }

private:
  static World instance_;

  Log *log_;
  FileSystem *fileSystem_;
  AudioPlayer *audioPlayer_;
};
```

Здесь глобальным объектом является только `World`. Функции могут получить доступ к другим системам через него:

```
World::instance().getAudioPlayer().play(VERY_LOUD_BANG);
```

> Пуристы могут заметить, что я нарушаю законы Деметры. И все таки я продолжаю утверждать, что это лучше, чем куча отдельных синглтонов.

Если позднее архитектуру придется изменить и добавить несколько экземпляров `World` (например для стримминга или тестовых целей) `Log`, `FileSystem` и `AudioPlayer` останутся незатронутыми — они даже разницы не заметят. Есть и недостаток — в результате гораздо больше кода будет завязано на сам класс `World`. Если классу просто нужно проиграть звук, наш пример все равно требует от него знания о `World` для того, чтобы получить аудио плеер.

Выходом может быть гибридное решение. Код, уже знающий о `World` может получать через него прямой доступ к `AudioPlayer`. А код, который о нем не знает может получать доступ к `AudioPlayer` с помощью другого решения, о которых мы уже говорили.

- **Получение через Локатор службы (Service Locator).** До сих пор мы предполагали что глобальный класс — это обязательно какой-то конкретный класс наподобие `World`. Но есть еще и вариант, при котором мы определяем класс, весь смысл которого будет заключаться в предоставлении глобального доступа к объектам. Этот шаблон называется Локатор службы (Service Locator) и мы его обсудим в отдельной главе.

Что же остается на долю Синглтона

Наш вопрос так и остался без ответа. Где же *стоит* применять шаблон Синглтон? Честно говоря, я никогда не использовал в игре чистую реализацию от Банды Четырех. Для обеспечения единственности экземпляра я предпочитаю использовать статический класс. Если это не работает, я использую статический флаг для проверки во время выполнения что экземпляр класса уже создан.

Еще нам могут помочь некоторые другие главы книги. Шаблон подкласс Песочница (Subclass Sandbox) дает нескольким экземплярам доступ к разделяемому состоянию, не делая его глобально доступным. Локатор службы (Service Locator) не только делает объект глобально доступным, но еще и предоставляет вам дополнительную гибкость в плане настройки объекта.

Состояние (State)

Пришло время исповедаться: я немного перестарался с этой главной. Предполагалось, что она посвящена шаблону проектирования Состояние (State) [GoF]. Но я не могу говорить о его применении в играх, не затрагивая концепцию *конечных автоматов (finite state machines)* (или "FSM"). Но как только я в нее углубился, я понял, что мне придется вспомнить *иерархическую машину состояний (hierarchical state machine)* или *иерархический автомат* и *автомат с магазинной памятью (pushdown automata)*.

Тематика получается слишком обширной, поэтому чтобы сократить главу до минимума, я буду опускать некоторые очевидные фрагменты примеров кода и вам придется заполнить некоторые пропуски самостоятельно. Я надеюсь, это не сделает их менее понятными.

Не нужно расстраиваться, если вы никогда не слышали про конечные автоматы. Они хорошо известны разработчикам `ии` и компьютерным хакерам, но малоизвестны в других областях. На мой взгляд они заслуживают большей известности, так что я хочу продемонстрировать вам несколько проблем, которые они решают.

> Все это отголоски старых ранних деньков искусственного интеллекта. В 50-е и 60-е искусственный интеллект в основном фокусировался на обработке языковых конструкций. Многие используемые в современных компиляторах технологии были изобретены для парсинга человеческих языков.

Все мы там были

Допустим мы работаем над небольшим платформером сайд-скроллером. Наша задача заключается в моделировании героини, которая будет аватаром игрока в игровом мире. Это значит, что она должна реагировать на пользовательский ввод. Нажмите `в` и она прыгнет. Довольно просто:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B) {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

Заметили баг?

Здесь нет никакого кода, предотвращающего "прыжок в воздухе"; продолжайте нажимать `B` пока она в воздухе и она будет подлетать снова и снова. Проще всего решить это добавлением булевского флага `isJumping_` в `Heroine`, который будет следить за тем когда героиня прыгнула:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B) {
    if (!isJumping_) {
      isJumping_ = true;
      // Прыжок...
    }
  }
}
```

> Нам нужен еще и код, который будет устанавливать `isJumping_` обратно в `false`, когда героиня снова коснется земли. Для простоты я опускаю этот код.

Дальше мы захотели добавить героине игрока возможность пригибаться, когда она находится на земле и вставать когда кнопка отжимается:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B) {
    // Прыгаем если уже не прыгнули...
  } else if (input == PRESS_DOWN) {
    if (!isJumping_) {
      setGraphics(IMAGE_DUCK);
    }
  } else if (input == RELEASE_DOWN) {
    setGraphics(IMAGE_STAND);
  }
}
```

А здесь баг заметили?

С помощью этого кода игрок может:

1. Нажать `вниз` для приседания.
2. Нажать `B` для прыжка из сидячей позиции.
3. Отпустить `вниз`, находясь в воздухе.

При этом героиня переключится на графику стояния прямо в воздухе. Придется добавить еще один флаг...

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B) {
    if (!isJumping_ && !isDucking_) {
      // Прыжок...
    }
  } else if (input == PRESS_DOWN) {
    if (!isJumping_) {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  } else if (input == RELEASE_DOWN) {
    if (isDucking_) {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

Теперь будет здорово добавить героине способность атаковать подкатом, когда игрок нажимает вниз, находясь в воздухе:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B) {
    if (!isJumping_ && !isDucking_) {
      // Прыжок...
    }
  } else if (input == PRESS_DOWN) {
    if (!isJumping_) {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    } else {
      isJumping_ = false;
      setGraphics(IMAGE_DIVE);
    }
  } else if (input == RELEASE_DOWN) {
    if (isDucking_) {
      // Стояние...
    }
  }
}
```

Снова ищем баги. Нашли?

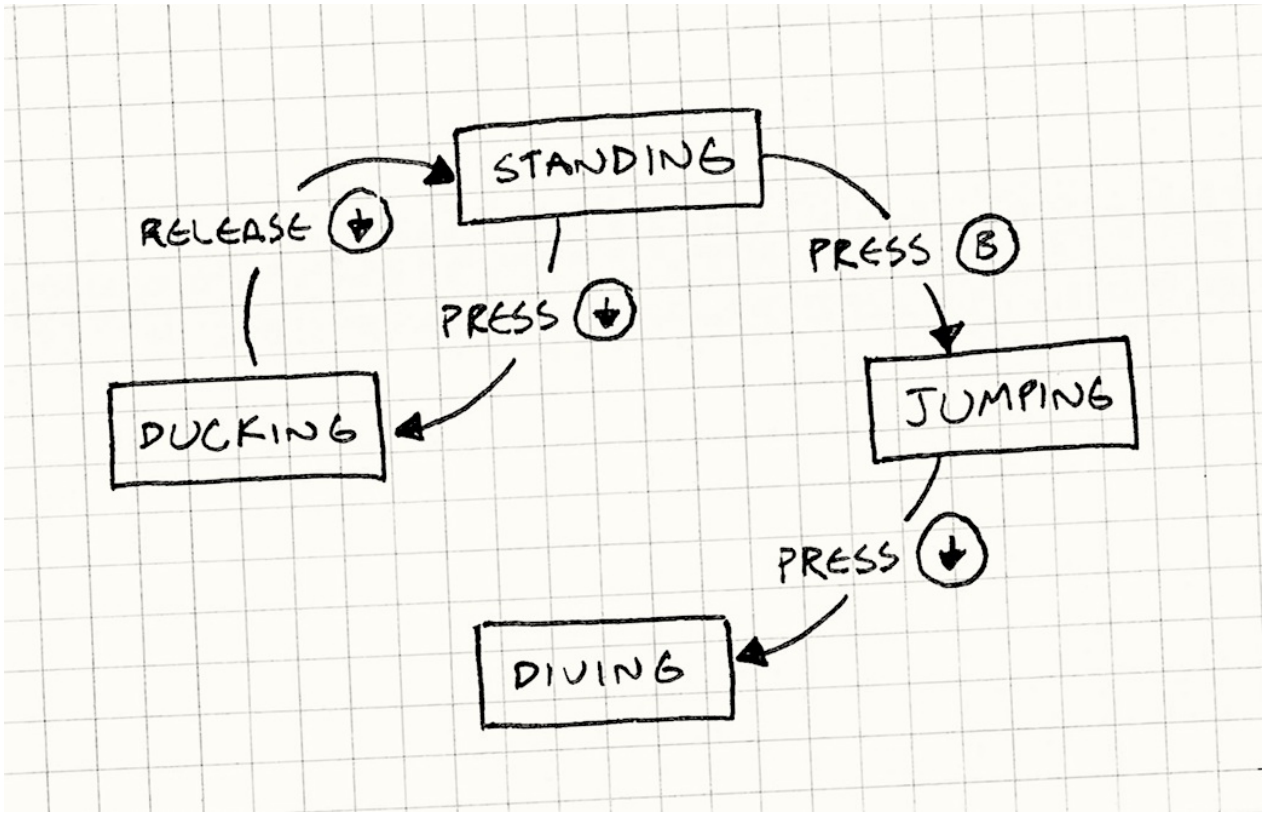
У нас есть проверка на то, чтобы было невозможно прыгнуть в воздухе, но не во время подката. Добавляем еще один флаг...

Есть в этом подходе что-то неправильное. Каждый раз, когда мы касаемся кода, у нас что-то ломается. Нам понадобится добавить еще кучу движения, у нас ведь еще даже *ходьбы* нет, но при таком подходе нам придется преодолеть еще кучу багов.

> Программисты которых мы все идеализируем и которые создают отличный код на самом деле совсем не супермены. Они просто развили в себе чутье на угрожающий появлением ошибок код и стараются по возможности его избегать.
>
> Сложное ветвление и изменяющиеся состояния — это как раз и есть те типы кода, которых стоит избегать.

Конечные автоматы — наше спасение

В порыве разочарования, вы убираете со стола все, кроме карандаша и бумаги и начинаете чертить блок-схему. Рисуем прямоугольник для каждого действия, что может совершить героиня: стояние, прыжок, приседание и подкат. Чтобы она могла реагировать на нажатие клавиш в любом из состояний, рисуем стрелки между этими прямоугольниками, подписываем над ними кнопки и соединяем между собой состояния.



Поздравляю, вы только что создали *конечный автомат* (*finite state machine*). Они пришли из области компьютерных наук, называемой *теория автоматов* (*automata theory*), в семейство структур которой также входит знаменитая машина Тьюринга. `FSM` - простейший член этого семейства.

Суть заключается в следующем:

- **У нас есть фиксированный набор *состояний*, в которых может находиться автомат.** В нашем примере это стояние, прыжок, приседание и подкат.
- **Автомат может находиться только в *одном* состоянии в каждый момент времени.** Наша героиня не может прыгать и стоять одновременно. Собственно для того чтобы это предотвратить FSM в первую очередь и используется.
- **Последовательность *ввода* или *событий*, передаваемых автомату.** В нашем примере это нажатие и отпускание кнопок.
- **Каждое состояние имеет *набор переходов*, каждый из которых связан с вводом и указывает на состояние.** Когда происходит пользовательский ввод, если он соответствует текущему состоянию, автомат меняет свое состояние на то куда указывает стрелка.

  Например, если нажать вниз в состоянии стояния, произойдет переход в состояние приседания. Нажатие вниз во время прыжка меняет состояние на подкат. Если в текущем состоянии никакой переход для ввода не предусмотрен — ничего не происходит.

В чистой форме это и есть целый банан: состояния, ввод и переходы. Можно изобразить их в виде блок-схемы. К сожалению, компилятор таких каракулей не поймет. Так как же в таком случае *реализовать* конечный автомат? Банда Четырех предлагает свой вариант, но начнем мы с еще более простого.

> Моя любимая аналогия `FSM` —это старый текстовый квест `Zork`. У вас есть мир, состоящий из комнат, которые соединены между собой переходами. И вы можете исследовать их, вводя команды типа "идти на север".
>
> Такая карта полностью соответствует определению конечного автомата. Комната, в которой вы находитесь —это текущее состояние. Каждый выход из комнаты — переход. Навигационные команды — ввод.

Перечисления и переключатели

Одна из проблем нашего старого класса `Heroine` заключается в том, что он допускает некорректную комбинацию булевских ключей: `isJumping_` и `isDucking_`, они не могут быть истинными одновременно. А если у вас есть несколько булевских флагов, только один из которых может быть `true`, не лучше ли заменить их все на `enum`.

В нашем случае с помощью `enum` можно полностью описать все состояния нашей `FSM` таким образом:

```
enum State
{
  STATE_STANDING,
  STATE_JUMPING,
  STATE_DUCKING,
  STATE_DIVING
};
```

Вместо кучи флагов, у `Heroine` есть только одно поле `state_`. Также нам придется изменить порядок ветвления. В предыдущем примере кода, мы делали ветвление сначала в зависимости от ввода, а потом уже от состояния. При этом мы группировали код по нажатой кнопке, но размывали код, связанный с состояниями. Теперь мы сделаем наоборот и будем переключать ввод в зависимости от состояния. Получим мы вот что:

```
void Heroine::handleInput(Input input)
{
  switch (state_) {
    case STATE_STANDING:
      if (input == PRESS_B) {
        state_ = STATE_JUMPING;
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
      } else if (input == PRESS_DOWN) {
        state_ = STATE_DUCKING;
        setGraphics(IMAGE_DUCK);
      }
      break;
```

```
    case STATE_JUMPING:
      if (input == PRESS_DOWN) {
        state_ = STATE_DIVING;
        setGraphics(IMAGE_DIVE);
      }
      break;

    case STATE_DUCKING:
      if (input == RELEASE_DOWN) {
        state_ = STATE_STANDING;
        setGraphics(IMAGE_STAND);
      }
      break;
  }
}
```

Выглядит довольно тривиально, но тем не менее этот код уже гораздо лучше, чем предыдущий. У нас остались некоторые условные ветвления, но зато мы упростили изменяемое состояние до единственного поля. Весь код, управляющий единственным состоянием собран в одном месте. Это самый простой способ реализации конечного автомата и иногда его вполне достаточно.

> Теперь героиня уже не сможет быть в *неопределенном* состоянии. При использовании булевых флагов некоторые комбинации были возможны, но не имели смысла. При использовании `enum` все значения корректны.

К сожалению, ваша проблема может перерасти такое решение. Допустим, мы захотели добавить нашей героине специальную атаку, для проведения которой героине нужно присесть для подзарядки и потом разрядить накопленную энергию. И пока мы сидим, нам нужно следить за временем зарядки.

Добавляем в `Heroine` поле `chargeTime_` для хранения времени зарядки. Допустим у нас уже есть метод `update()`, вызываемый на каждом кадре. Добавим в него следующий код:

```
void Heroine::update()
{
  if (state_ == STATE_DUCKING) {
    chargeTime_++;

    if (chargeTime_ > MAX_CHARGE) {
      superBomb();
    }
  }
}
```

> Если вы угадали, что это шаблон Метод обновления (Update Method), вы выиграли приз!

Каждый раз, когда мы приседаем заново, нам нужно обнулять этот таймер. Для этого нам нужно изменить `handleInput()`:

```
void Heroine::handleInput(Input input)
{
  switch (state_) {
    case STATE_STANDING:
      if (input == PRESS_DOWN) {
        state_ = STATE_DUCKING;
        chargeTime_ = 0;
        setGraphics(IMAGE_DUCK);
      }
      // Обработка оставшегося ввода...
      break;

    // Другие состояния...
  }
}
```

В конце концов, для добавления этой атаки с подзарядкой, нам пришлось изменить два метода и добавить поле `chargeTime_` в `Heroine`, даже если оно используется только в состоянии приседания. Хотелось бы иметь весь этот код и данные в одном месте. Банда Четырех может нам в этом помочь.

Шаблон состояние

Для людей, хорошо разбирающихся в объектно-ориентированной парадигме, каждое условное ветвление — это возможность для использования динамической диспетчеризации (другими словами, вызова виртуального метода в `C++`). Думаю нам нужно спуститься в эту кроличью нору еще глубже. Иногда `if` — это все что нам нужно.

> Этому есть историческое обоснование. Многие из старых апостолов объектно-ориентированной парадигмы, такие как Банда Четырех со своими *Паттернами программирования* и Мартин Фулер с его *Рефакторингом* пришли из `Smalltalk`. А там `ifThen` — это всего лишь метод, которым вы обрабатываете условие и который реализуется по разному для объектов `true` и `false`.

В нашем примере мы уже добрались до той критической точки, когда нам стоит обратить внимание на что-то объектно-ориентированное. Это подводит нас к шаблону Состояние. Цитирую Банду Четырех:

Позволяет объектам менять свое поведение в соответствии с изменением внутреннего состояния. При этом объект будет вести себя как другой класс.

Не очень то и понятно. В конце концов и `switch` с этим справляется. Применительно к нашему примеру с героиней шаблон будет выглядеть следующим образом:

Интерфейс состояния

Для начала определим интерфейс для состояния. Каждый бит поведения, зависящий от состояния — т.е. все что мы раньше реализовывали при помощи `switch` — превращается в виртуальный метод этого интерфейса. В нашем случае это `handleInput()` и `update()`.

```
class HeroineState
{
public:
  virtual ~HeroineState() {}
  virtual void handleInput(Heroine& heroine, Input input) {}
  virtual void update(Heroine& heroine) {}
};
```

Классы для каждого из состояний

Для каждого состояния мы определяем класс, реализующий интерфейс. Его методы определяют поведение героини в данном состоянии. Другими словами берем все варианты из `switch` в предыдущем примере превращаем их в класс состояния. Например:

```
class DuckingState : public HeroineState
{
public:
  DuckingState() : chargeTime_(0)
  {}

  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN) {
      // Переход в состояние стояния...
      heroine.setGraphics(IMAGE_STAND);
    }
  }

  virtual void update(Heroine& heroine) {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE) {
      heroine.superBomb();
    }
  }

private:
  int chargeTime_;
};
```

Обратите внимание, что мы перенесли `chargeTime_` из класса самой героини в класс `DuckingState`. И это очень хорошо, потому что этот кусок данных имеет значение только в этом состоянии и наша модель данных явно об этом свидетельствует.

Делегирование к состоянию

Дальше мы даем `Heroine` указатель на текущее состояние, избавляемся от здоровенного `switch` и делегируем его работу состоянию.

```
class Heroine
{
public:
  virtual void handleInput(Input input) {
    state_->handleInput(*this, input);
  }

  virtual void update() {
    state_->update(*this);
  }

  // Другие методы...
private:
  HeroineState* state_;
};
```

Чтобы "изменить состояние" нам нужно просто сделать так, чтобы `state_` указывал на другой объект `HeroineState`. В этом собственно и заключается шаблон Состояние.

> Выглядит довольно похоже на шаблоны Стратегия (Strategy) GoF и Объект тип (Type Object). Во всех трёх у нас есть главный объект, делегирующий к подчиненному. Различие в *назначении*.
>
> - Цель Стратегии заключается в *уменьшении связности* (decouple) между главным классом и его поведением.
> - Целью Объект тип (Type Object) является создание некоторого количества объектов, ведущих себя одинаково с помощью разделения между собой общего объекта типа.
> - Целью Состояния является изменение поведения главного объекта через изменение объекта к которому он делегирует.

А где же эти объекты состояния?

Я вам кое-что не сказал. Чтобы изменить состояние, нам нужно присвоить `state_` новое значение, указывающее на новое состояние, но откуда этот объект возьмется? В нашем примере с `enum` думать не о чем: значения `enum` — это просто примитивы наподобие чисел. Но теперь наши состояния представлены классами и это значит, что нам нужны указатели на реальные экземпляры. Существует два самых распространенных ответа:

Статические состояния

Если объект состояния не имеет никаких других полей, единственное, что он хранит — это указатель на внутреннюю виртуальную таблицу методов, для того чтобы эти методы можно было вызвать. В таком случае, нет никакой необходимости иметь больше одного экземпляра класса: каждый из экземпляров все равно будет одинаковым.

> Если у вашего состояния нет полей и только один виртуальный метод, можно еще сильнее упростить шаблон. Заменим каждый *класс* состояния *функцией* состояния — обычной функцией верхнего уровня. И соответственно поле *state_* в нашем главном классе превратится в простой указатель на функцию.

Вполне можно обойтись единственным *статическим* экземпляром. Даже если у вас целая куча `FSM`, находящихся одновременно в одном и том же состоянии, они могут указывать на один и тот же статический экземпляр, потому что ничего специфичного для конкретного конечного автомата в нем нет.

> Получился шаблон Приспособленец (Flyweight Pattern) GoF

Куда вы поместите статический экземпляр — это уже ваше дело. Найдите такое место, где это будет уместно. Давайте поместим наш экземпляр в базовый класс. Без всякой причины.

```
class HeroineState
{
public:
  static StandingState standing;
  static DuckingState ducking;
  static JumpingState jumping;
  static DivingState diving;

  // Остальной код...
};
```

Каждое из этих статических полей — экземпляр состояния, используемого игрой. Чтобы заставить героиню подпрыгнуть, состояние стояния сделает нечто вроде:

```
if (input == PRESS_B)
{
  heroine.state_ = &HeroineState::jumping;
  heroine.setGraphics(IMAGE_JUMP);
}
```

Экземпляры состояний

Иногда предыдущий вариант не взлетает. Статическое состояние не подойдет для состояния присядки. У него есть поле `chargeTime_` и оно специфично для героини, которая будет приседать. Это еще худо бедно сработает в нашем случае, потому что у нас всего одна героиня, но если мы захотим добавить кооператив для двух игроков, у нас будут большие проблемы.

В таком случае, нам следует создавать объект состояния, когда мы переходим в него. Это позволит каждому `FSM` иметь собственный экземпляр состояния. Конечно, если мы выделяем память под *новое* состояние, это значит нам следует *освободить* занимаемую память текущего. Мы должны быть осторожны, так как код, который вызывает изменения находится в методее текущего состоянии. Мы не хотим, чтобы удалить `this` из-под себя.

Вместо этого, мы позволим `handleInput()` в `HeroineState` опционально возвращать новое состояние. Когда это произойдет, `Heroine` удалит старое состояние и поменяет его на новое, например, так:

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL) {
    delete state_;
    state_ = state;
  }
}
```

Таким образом, мы не удаляем предыдущее состояние, пока мы не вернулись из своего метода. Теперь, состояние стояния может перейти к состоянию нырок путем создания нового экземпляра:

```
HeroineState* StandingState::handleInput(Heroine& heroine, Input input)
{
  if (input == PRESS_DOWN) {
    // Other code...
    return new DuckingState();
  }

  // Stay in this state.
  return NULL;
}
```

Когда у меня получается, я предпочитаю использовать статические состояния, потом что они не занимают память и такты процессора, выделяя объекты при каждом изменении состояния. Для состояний, которые не представляют из себя нечто большее, чем просто *состояния* — это как раз то что нужно.

> Конечно, когда вы выделяете память под состояние динамически, вам стоит подумать о возможной фрагментации памяти. Помочь может шаблон Пул объектов (Object Pool).

Действия для входа и выхода

Шаблон Состояние предназначен для инкапсуляции всего поведения и связанных с ним данных внутри одного класса. У нас довольно неплохо получается, но остались некоторые невыясненные детали.

Когда героиня изменяет состояние, мы также переключаем ее спрайт. Прямо сейчас, этот код принадлежит состоянию, с *которого* она переключается. Когда состояние переходит от нырка в состояние стояния, то нырок устанавливает ее образ:

```
HeroineState* DuckingState::handleInput(Heroine& heroine, Input input)
{
  if (input == RELEASE_DOWN) {
    heroine.setGraphics(IMAGE_STAND);
    return new StandingState();
  }

  // Other code...
}
```

То, что мы действительно хотим, каждое состояние контролировало свою собственную графику. Мы можем добиться этого, добавив в состояние *входное действие* (*entry action*):

```
class StandingState : public HeroineState
{
public:
  virtual void enter(Heroine& heroine) {
    heroine.setGraphics(IMAGE_STAND);
  }

  // Other code...
};
```

Возвращаясь к `Heroine`, мы модифицируем код, добиваясь, чтобы изменение состояния сопровождалось вызовом функции входного действия нового состояния:

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL) {
    delete state_;
    state_ = state;

    // Вызов входного действия нового состояния.
    state_->enter(*this);
  }
}
```

Это позволит упростить код состояния DuckingState:

```
HeroineState* DuckingState::handleInput(Heroine& heroine, Input input)
{
  if (input == RELEASE_DOWN) {
    return new StandingState();
  }

  // Other code...
}
```

Все это делает переключение в стояние и состояние стояния заботится о графике. Теперь наши состояния действительно инкапсулированы. Еще одной приятной особенностью такого входного действия является то, что оно запускается при входе в состояние независимо от состояния, в *котором* мы находились.

На большинстве графов состояний из реальной жизни присутствует несколько переходов в одно и то же состояние. Например, наша героиня может стрелять из оружия стоя, сидя или в прыжке. А это значит, что у нас может появиться дублирование кода везде, где это происходит. Входное действие позволяет собрать его в одном месте.

Можно по аналогии сделать и *выходное действие* (*exit action*). Это будет просто метод, который мы будем вызывать для состояния, перед тем, как *покидаем* его и переключаемся на новое состояние.

И чего же мы добились?

Я столько времени потратил, чтобы продать вам `FSM`, а теперь собираюсь вырвать коврик из под ног. Все, что я до сих пор говорил — правда и отлично решает проблемы. Но, так уж вышло, что самые главные достоинства конечных автоматов, одновременно являются и их самыми большими недостатками.

Состояние автомата помогает вам серьезно распутать код, организовав его в крайне строгую структуру. Все, что у нас есть — это фиксированный набор состояний, единственное текущее состояние и жестко запрограммированные переходы.

> Конечный автомат не обладает полнотой по Тьюрингу (Turing complete). Теория автоматов описывает полноту через серию абстрактных моделей, каждая из которых сложнее предыдущей. Машина Тьюринга — одна из самых выразительных.
>
> "Полнота по Тьюригну" означает систему (обычно язык программирования), обладающую достаточной выразительностью для реализации машины Тьюринга. В свою очередь это означает что все полные по Тьюрингу языки примерно одинаково выразительны. FSM недостаточно выразительны чтобы войти в этот клуб.

Если же вы попробуете использовать машину состояний для чего-либо более сложного, как например игровой `AI`, вы сразу уткнетесь в ограничения этой модели. К счастью, наши предшественники научились обходить некоторые препятствия. Я закончу эту главу несколькими такими примерами.

Машина конкурентных состояний

Мы решили добавить нашей героине возможность носить оружие. Хотя она теперь вооружена, она по прежнему может делать все, что делала раньше: бегать, прыгать, приседать и т.д. Но теперь, делая все это, она еще может и стрелять из оружия.

Если мы захотим вместить такое поведение в рамки `FSM`, нам придется удвоить количество состояний. Для каждого из состояний нам придется завести еще одно такое же, но уже для героини с оружием: стояние, стояние с оружием, прыжок, прыжок с оружием.... Ну вы поняли.

Если добавить еще несколько видов оружия, количество состояний увеличится комбинаторно. И это не просто куча состояний, а еще и куча повторов: вооруженное и безоружное состояния практически идентичны за исключением части кода, отвечающей за стрельбу.

Проблема здесь в том, что мы смешиваем две части состояния — что она *делает* и что *держит в руках* — в один автомат. Чтобы смоделировать все возможные комбинации, нам нужно завести состояние для каждой *пары*. Решение очевидно: нужно завести два отдельных конечных автомата.

> Если мы хотим объединить n состояний действия и m состояний того, что держим в руках в один конечный автомат — нам нужно $n \times m$ состояний. Если у нас будет два автомата — нам понадобится $n + m$ состояний.

Наш первый конечный автомат с действиями мы оставим без изменений. А в дополнение к нему создадим еще один автомат для описания того, что героиня держит. Теперь у `Heroine` будет две ссылки на "состояние", по одной для каждого автомата.

```
class Heroine
{
  // Остальной код...

private:
  HeroineState* state_;
  HeroineState* equipment_;
};
```

> Для иллюстрации мы используем полную реализацию шаблона Состояние для второго конечного автомата, хотя на практике в данном случае хватило бы простого булевского флага.

Когда героиня делегирует ввод состояниям, она передает перевод обеим конечным автоматам:

```
void Heroine::handleInput(Input input)
{
  state_->handleInput(*this, input);
  equipment_->handleInput(*this, input);
}
```

> Более сложные системы могут иметь в своем составе конечные автоматы, которые могут поглощать часть ввода таким образом чтобы другие автоматы его уже не получали. Это позволит нам предотвратить ситуацию, когда несколько автоматов реагируют на один и тот же ввод.

Каждый конечный автомат может реагировать на ввод, порождать поведение и изменять свое состояние независимо от других автоматов. И когда оба состояния практически не связаны между собой это отлично работает.

На практике вы можете встретить ситуацию, когда состояния взаимодействуют друг с другом. Например, она не может выстрелить в прыжке или например выполнить атаку с подкатом когда вооружена. Чтобы обеспечить такое поведение и координацию автоматов в коде, вам придется вернуться к той же самой грубой проверке через `if` *другого* конечного автомата. Не самое элегантное решение, но по крайней мере работает.

Иерархическая машина состояний

После дальнейшего оживления поведения героини, у нее наверняка появится целый букет похожих состояний. Например, у не могут быть состояния стояния, ходьбы, бега и скатывания со склонов. В любом из этих состояний нажатие на `B` заставляет ее подпрыгнуть, а нажатие `вниз` — присесть.

В простейшей реализации конечного автомата мы дублировали этот код для всех состояний. Но конечно было бы гораздо лучше, если бы нам нужно было написать код всего один раз и после этого мы могли бы использовать его повторно для всех состояний.

Если бы это был просто объектно-ориентированный код, а не конечный автомат, можно было бы использовать такой прием разделения кода между состояниями, как наследование. Можно определить класс для состояния "на земле", который будет обрабатывать подпрыгивание и приседание. Стояние, ходьба, бег и скатывание для него наследуется и добавляет свое дополнительное поведение.

> Такое решение имеет как хорошие, так и плохие последствия. Наследование — это мощный инструмент для повторного использования кода, но в то же время оно дает очень сильную связность между двумя кусками кода. Молот слишком тяжел, чтобы бить им бездумно.

В таком виде получившаяся структура будет называться *иерархическая машина состояний* (или *иерархический автомат*). А у каждого состояния может быть свое *суперсостояние* (само состояние при этом называется *подсостоянием*). Когда

наступает событие и подсостояние его не обрабатывает, оно передается по цепочке суперсостояний вверх. Другими словами, получается подобие переопределения унаследованного метода.

На самом деле, если мы используем оригинальный шаблон Состояние для реализации `FSM`, мы уже можем использовать наследование классов для реализации иерархии. Определим базовый класс для суперкласса:

```
class OnGroundState : public HeroineState
{
public:
  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == PRESS_B) {
      // Подпрыгнуть...
    } else if (input == PRESS_DOWN) {
      // Присесть...
    }
  }
};
```

А теперь каждый подкласс будет его наследовать:

```
class DuckingState : public OnGroundState
{
public:
  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN) {
      // Встаем...
    } else {
      // Ввод не обработан. Поэтому передаем его выше по иерархии.
      OnGroundState::handleInput(heroine, input);
    }
  }
};
```

Конечно, это не единственный способ реализации иерархии. Но, если вы не используете шаблон Состояние Банды Четырех, это не сработает. Вместо этого вы можете смоделировать четкую иерархию текущих состояний и суперсостояний с помощью *стека* состояний вместо единственного состояния в главном классе.

Текущее состояние будет находится вверху стека, под ним его суперсостояние, дальше суперсостояние для *этого* суперсостояния и т.д. И когда вам нужно будет реализовать специфичное для состояния поведение, вы начнете с верха стека спускаться по нему вниз, пока состояние его не обработает. (А если не обработает — значит вы его просто игнорируете).

Автомат с магазинной памятью

Есть еще одно обычное расширение конечных автоматов, также использующее стек состояния. Только здесь стек представляет совершенно другую концепцию и используется для решения других проблем.

Проблема в том, что у конечного автомата нет концепции *истории*. Вы знаете в каком состоянии вы *находитесь*, но у вас нет никакой информации о том, в каком состоянии вы *были*. И соответственно нет простой возможности вернуться в предыдущее состояние.

Вот простой пример: Ранее мы позволили нашей бесстрашной героине вооружиться до зубов. Когда она стреляет из своего оружия, нам нужно новое состояние для проигрывания анимации выстрела, порождения пули и сопутствующих визуальных эффектов. Для этого мы создаем новое `FiringState` и делаем в него переходы из всех состояний, в которых героиня может стрелять по нажатию кнопки стрельбы.

> Так как это поведение дублируется между несколькими состояниями, здесь как раз можно применить иерархическую машину состояний для повторного использования кода.

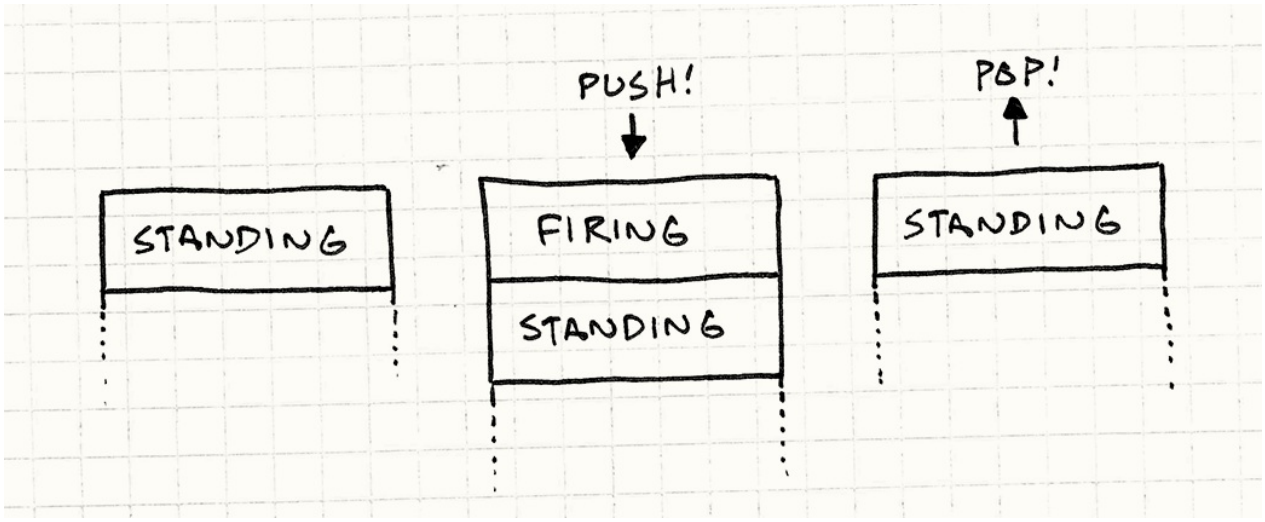
Сложность здесь в том, что нужно каким либо образом понять в какое состояние нужно перейти *после* стрельбы. Героиня может выстрелить всю обойму, пока она стоит на месте, бежит, прыгает или приседает. Когда последовательность стрельбы закончена, ей нужно вернуться в состояние, в котором она была до стрельбы.

Если мы привязываемся к чистому `FSM`, мы сразу забываем в каком состоянии мы были. Чтобы за этим следить, нам нужно определить множество практически одинаковых состояний — стрельба стоя, стрельба в беге, стрельба в прыжке и т.д. Таким образом, у нас образуются жестко закодированные переходы, переходящие в правильное состояние по своему окончанию.

Что нам на самом деле нужно — так это возможность хранить состояние, в котором мы находились до стрельбы и после стрельбы вспоминать его снова. Здесь нам снова может помочь теория автоматов. Соответствующая структура данных называется Автомат с магазинной памятью (Pushdown Automaton).

Там, где в конечном автомате у нас находится единственный указатель на состояние, в автомате с магазинной памятью находится их *стек*. В `FSM` переход к новому состоянию заменяет собой предыдущий. Автомат с магазинной памятью тоже позволяет это делать, но добавляет сюда еще две операции:

1. Вы можете *поместить* (*push*) новое состояние в стек. Текущее состояние всегда будет находиться вверху стека, так что это и есть операция перехода в новое состояние. Но при этом старое состояние остается прямо под текущим в стеке, а не исчезает бесследно.
2. Вы можете *извлечь* (*pop*) верхнее состояние из стека. Состояние пропадает и текущим становится то что находилось под ним.



Это все что нам нужно для стрельбы. Мы создаем *единственное* состояние стрельбы. Когда мы нажимаем кнопку стрельбы, находясь в другом состоянии, мы *помещаем* (*push*) состояние стрельбы в стек. Когда анимация стрельбы заканчивается, мы *извлекаем* (*pop*) состояние и автомат с магазинной памятью автоматически возвращает нас в предыдущее состояние.

Насколько они реально полезны?

Даже с этим расширением конечных автоматов, их возможности все равно довольно ограничены. В `AI` сегодня преобладает тренд использования вещей типа *деревьев поведения* (behavior trees) и *систем планирования* (planning systems). И если вам интересна именно область `AI`, вся эта глава должна просто раздразнить ваш аппетит. Чтобы его удовлетворить, вам придется обратиться к другим книгам.

Это совсем не значит, что конечные автоматы, автоматы с магазинной памятью и другие подобные системы полностью бесполезны. Для некоторых вещей это хорошие инструменты для моделирования. Конечные автоматы полезны когда:

- У вас есть сущность, поведение которой изменяется в зависимости от ее внутреннего состояния.
- Это состояние жестко делится на относительно небольшое количество конкретных вариантов.

- Сущность постоянно отвечает на серии команд ввода или событий.

В играх конечные автоматы обычно используются для моделирования `AI`, но их можно применять и для реализации пользовательского ввода, навигации в меню, парсинга текста, сетевых протоколов и другого асинхронного поведения.

Последовательные шаблоны

Видеоигры прекрасны по большей мере потому, что позволяют нам побывать где-то еще. На несколько минут (а если быть более честным, гораздо на дольше) мы становимся обитателями виртуального мира. Создание такого мира — это самое прекрасное, что есть в игровом программировании.

Один из аспектов создания таких игровых миров — это *время*: искусственный мир живет и дышит в своем собственном ритме. Как строители миров, мы должны самостоятельно изобретать время и шестеренки, управляющие работой часов игры.

В данном разделе собраны шаблоны, которые могут нам в этом помочь. Игровой цикл (Game Loop) — это центральная ось, на которую опирается игровое время. Объекты слышат его тиканье через методы обновления (Update Methods). Мы можем спрятать последовательную сущность компьютера за фасадом снимков отдельных моментов времени с помощью двойной буферизации (Double Buffering) и в результате обновление игрового мира будет казаться плавным.

Шаблоны

- Двойная буферизация (Double Buffering)
- Игровой цикл (Game Loop)
- Методы обновления (Update Methods)

Двойная буферизация (Double Buffering)

Задача

Дать возможность ряду последовательных операций выполняться мгновенно или одновременно.

Мотивация

В своем сердце компьютер отсчитывает последовательность ударов. Его мощь заключается в способности разбивать громадные задания на мелкие шаги, которые можно выполнять один за другим. Однако пользователю зачастую нужно видеть как вещи выполняются за один шаг или несколько задач выполняются одновременно.

> В потоковой и многоядерной архитектуре это уже не совсем верно, но даже при наличии нескольких ядер, всего только несколько операций могут выполняться в конкурентном режиме.

Типичный пример, встречающийся в любом игровом движке — это рендеринг. Когда игра отрисовывает мир, видимый пользователем, она делает это отдельными кусочками: горы вдали, крутые холмы, деревья, все по очереди. Если пользователь *увидит* этот процесс отрисовки в таком инкрементном режиме, иллюзия когерентности мира теряется. Сцена должна обновляться плавно и быстро, образуя последовательность законченных кадров, появляющихся мгновенно.

Двойная буферизация решает эту проблему, но чтобы понять как, нам нужно для начала вспомнить как компьютер показывает графику.

Как работает компьютерная графика (коротко)

Видео дисплей, как и компьютерный монитор, рисует пиксель за пикселем. Они обновляются один за другим, слева направо в каждом ряду, а затем происходит переход вниз к следующему ряду. Когда нижний правый угол достигнут, происходит переход к левому верхнему углу и процесс начинается заново. Происходит это так быстро — по крайней мере 60 раз в секунду, что наш глаз этого пробегания по рядам не замечает. Для нас все выглядит, так как будто сменяются статичные картинки на весь экран.

Можно думать об этом как о крошечном шланге, из которого мы поливаем экран пикселями. Отдельные цвета подводятся к этому шлангу и распыляются на экран по одному биту цвета за раз. Но каким образом этот шланг знает куда какой цвет направлять?

> Такое объяснение конечно несколько... упрощено. Если вы хорошо разбираетесь в работе железа — можете спокойно пропустить следующий раздел. У вас уже есть все необходимые знания для понимания оставшейся части главы. А если вам это *незнакомо*, я дам вам необходимый минимум знания для понимания шаблона.

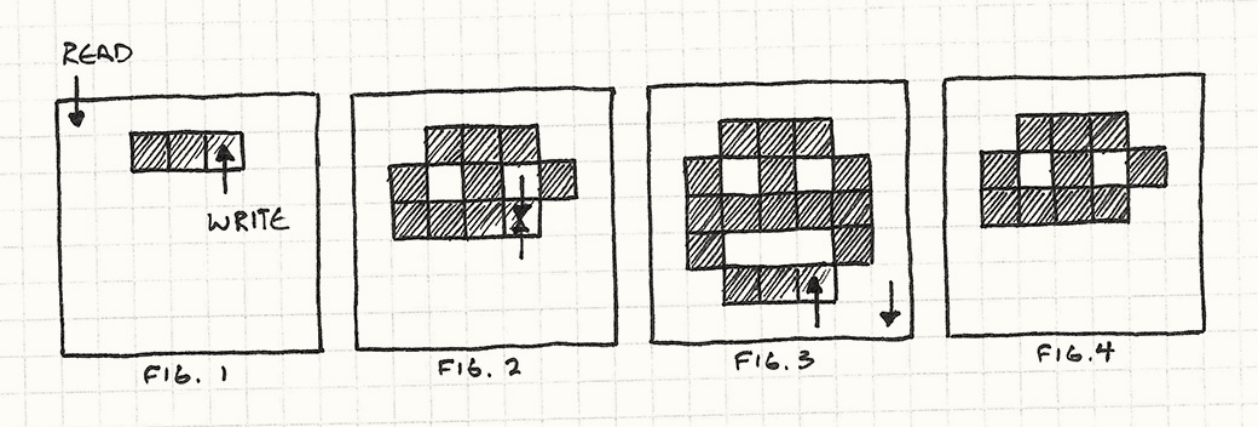
В большинстве компьютеров ответ кроется в применение *буфера кадра* (framebuffer). Буфер кадра — это массив пикселей в памяти, участок RAM, где каждые несколько байтов представляют собой отдельный пиксель. И в то время когда шланг распыляет пиксели по экрану, он считывает значения цветов из массива по одному байту за раз.

Для того чтобы наша игра появилась на экране, все что мы на самом деле предпринимаем — это просто записываем значения в этот массив. Все самые изощренные графические алгоритмы сводятся к этому: установке значения байтов в буфере кадра. Но есть тут одна проблема.

Как я сказал ранее, компьютеры работают последовательно. Если машина выполняет кусок нашего кода рендеринга, мы не ожидаем что делаем в тот же самый момент что-то еще. В целом это верно, но некоторые вещи все-таки происходят во время выполнения нашей программы. Один из таких процессов — это постоянное считывание информации из буфера кадра. И это уже может быть проблемой.

> Соответствие между значениями байтов и цветами описывается *форматом пикселей* (pixel format) и *глубиной цвета* (color depth) системы. В большинстве современных игр используется 32 битный цвет: по восемь бит на красный, зеленый и синий канал и еще восемь для специального дополнительного канала.

Предположим что мы хотим отобразить на экране смайлик. Наша программа начинает в цикле двигаться по буферу кадра, окрашивая пиксели. Что мы до сих пор не уяснили — так это то, что видео драйвер производит считывание из буфера кадра в то же самое время, когда мы ведем в него запись. И когда он проходит по записанным пикселям, на экране начинает появляться смайлик. А потом он нас обгоняет и считывает данные из тех пикселей, куда мы еще ничего не записали. Результат паршивый: результатом будет баг, когда картинка отрисовывается только в верхней части экрана.



> Мы начинаем отрисовывать пиксели из буфера кадра также как видео драйвер (Рис. 1). Видеодрайвер настигает рендер и попадает туда, куда пиксели еще не записывались (Рис. 2). Дале мы заканчиваем отрисовку (Рис. 3), но драйвер этого уже не видит.
>
> Вот результат, который пользователь увидит на экране (Рис. 4). Название "разрыв" возник потому что нижняя часть картинки как будто оторвана.

Вот здесь нам и пригодится наш шаблон. Наша программа рендерит пиксели по одному за раз, но драйверу нам нужно передавать всю картинку целиком — один кадр без смайлика и один кадр со смайликом. Именно для этого и нужна двойная буферизация. Попробую подобрать понятную аналогию.

Акт первый. Сцена первая

Представьте, что мы смотрим нашу собственную пьесу. Как только заканчивается первая сцена и начинается вторая, нам нужно сменить декорации. Если мы просто начнем перетаскивать реквизит, иллюзия когерентности пространства пропадет. Мы конечно можем просто приглушить свет на этот период (так в театре тоже делают), но аудитория по прежнему будет понимать что что-то происходит. Мы же хотим чтобы между сценами не было провалов.

В реальности мы можем прибегнуть к оригинальному решению: Построим две декорации таким образом, что они обе будут видны публике. У каждой сцены свое освещение. Назовем их A и B. Первая сцена демонстрируется в декорации A. В это время декорация B затемнена и работники сцены готовят ее к показу сцены два. Как только сцена первая завершается, мы выключаем свет в декорации A и включаем его в декорации B. Внимание публики сразу переключается к декорации B, где уже начинается сцена вторая.

> С помощью полупрозрачного зеркала и очень маленького макета, можно добиться того чтобы зрители видели обе сцены одновременно в одном и том же месте. Как только освещение поменяется, они будут смотреть уже на другую сцену не меняя при этом направление взгляда. Предлагаю вам провести такой эксперимент самостоятельно.

В это время наши работники сцены занимаются затемненной декорацией А, подготавливая ее для сцены три. Как только сцена два закончится, мы снова переключим свет на декорацию А. Этот процесс будет повторяться на протяжении всей пьесы, используя затемненную декорацию для подготовки следующей сцены. Для перехода между сценами мы просто затемняем одну и освещаем другую. В результате наша публика получает возможность видеть спектакль без задержек между сценами. И никогда не видит работников сцены.

Вернемся к графике

Точно также работает и двойная буферизация и именно такой процесс скрывается за системой рендеринга практически любой современной игры. Вместо единственного буфера кадра у нас есть *два*. Один из них представляет текущий кадр — аналогию декорации А. Это то место, откуда считывает данные видеодрайвер. `GPU` может производить из него считывание сколько угодно и когда угодно.

> Хочу заметить, что не все игры и консоли пользуются таким методом. Старые и самые простые из консолей были настолько ограничены в плане памяти, что вынуждены были синхронизировать отрисовку с обновлением картинки. Довольно хитрая задача.

В это время наш код рендеринга пишет в *другой* буфер. Это наша затемненная декорация В. Когда код рендеринга заканчивает отрисовку сцены, мы переключаем свет *подменяя* (swapping) буфера. Этим самым мы говорим видеобуферу, чтобы он теперь считывал данные из второго буфера вместо первого. И пока мы будем выполнять переключение в конце обновления экрана, никаких разрывов мы не увидим и сцена будет отображаться целиком.

А в это самое время наш старый буфер кадра становится готовым к использованию. Мы начинаем рендерить в него новый кадр. Вуаля!

Шаблон

Класс буфера инкапсулирует **буфер** — часть состояния, которое можно изменить. Буфер изменяется постепенно, но мы хотим чтобы внешний код увидел изменение как единый атомарный процесс. Чтобы это стало возможным, класс хранит *два* буфера: **следующий** и **текущий**.

Когда требуется *считать информацию из* буфера — всегда используется *текущий*. А когда информация *записывается* — используется *следующий* буфер. Когда изменения закончены, операция **обмена** (swap) мгновенно меняет местами следующий и текущий буферы, так что новый буфер становится видным публично. Старый текущий буфер теперь доступен для повторного использования в качестве следующего буфера.

Когда его использовать

Это шаблон из тех, про который вы сами поймете, когда его нужно будет использовать. Если у вас есть система, в которой не хватает двойной буферизации, это обычно заметно (как в случае с разрывом) или приводит к некорректной работе. Но просто сказать "Вы поймете когда он вам пригодится" — недостаточно. Если говорить конкретнее, этот шаблон стоит применять если справедливо одно из следующих утверждений:

- У нас есть состояние, изменяющееся постепенно.
- К состоянию есть доступ посередине процесса его изменения.
- Мы хотим предотвратить код, считывающий состояние от чтения незаконченного изменения.
- Мы хотим иметь возможность считывать состояние, не дожидаясь когда оно будет изменено.

Имейте в виду

В отличие от больших архитектурных шаблонов, двойная буферизация существует на низкоуровневом слое реализации. Поэтому последствия на всю кодовую базу в целом не слишком велики — большая часть игры даже не заметит разницы. Но и здесь не обошлось без подводных камней.

Переключение само по себе требует времени

Двойная буферизация требует этапа *переключения* (swap), как только изменение будет закончено. Само это переключение должно быть атомарным — остальной код не должен иметь доступ во время этой операции *ни к одному* из состояний. Чаще всего переключение выполняется также быстро как переназначение указателя. А вот если переключение требует больше времени чем собственно изменение состояния, то толку от шаблона не будет никакого.

Нам нужно иметь два буфера

Втрое следствие применения шаблона — увеличение потребления памяти. Как явственно следует из названия нам нужно постоянно держать именно *две* копии состояния в памяти. На устройствах с ограниченным объемом памяти это довольно дорогая цена за применение шаблона. Если вы не можете позволить себе иметь два буфера, вам стоит присмотреться к другим способам обеспечения недоступности состояния для чтения во время изменения.

Пример кода

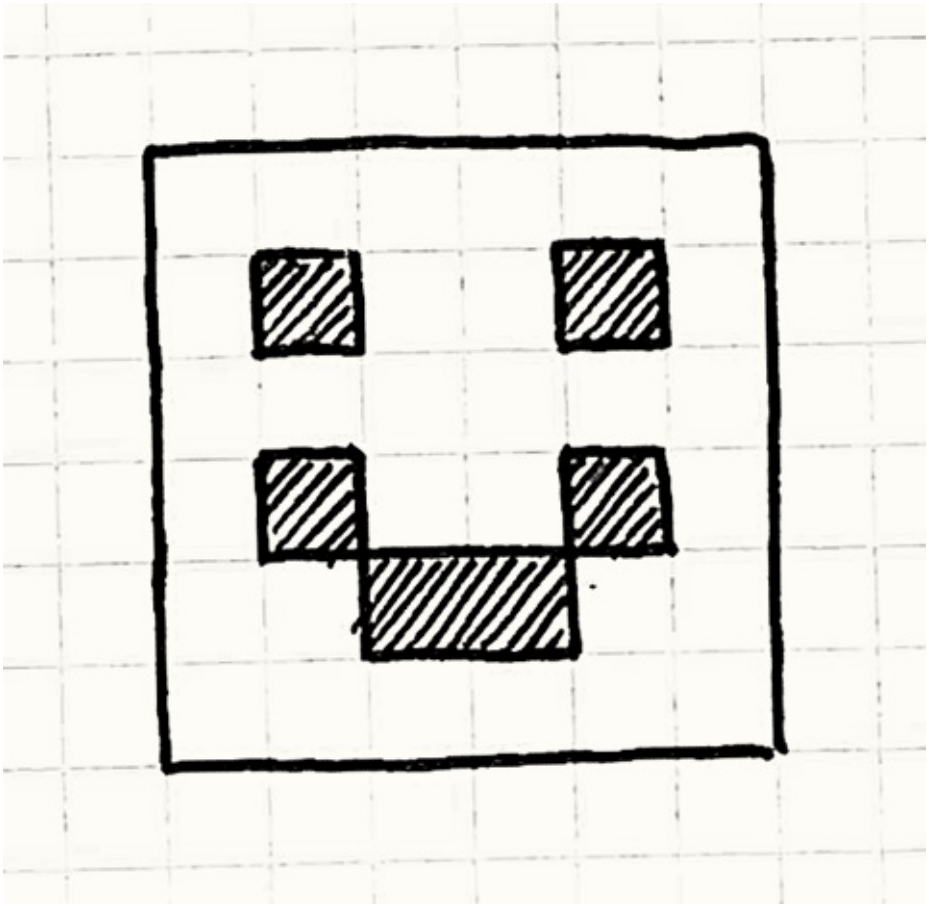
Теперь, когда мы разобрались с теорией, давайте перейдем к практике. Мы напишем очень приблизительную графическую систему вывода пикселей в буфер кадра. В большинстве консолей и `PC` всю эту низкоуровневую графическую работу делает видеодрайвер, однако ручная реализация позволит нам лучше разобраться в происходящем. Для начала сам буфер:

```
class Framebuffer
{
public:
  Framebuffer() { clear(); }

  void clear() {
    for (int i = 0; i < WIDTH * HEIGHT; i++) {
      pixels_[i] = WHITE;
    }
  }

  void draw(int x, int y) {
    pixels_[(WIDTH * y) + x] = BLACK;
  }

  const char* getPixels() {
    return pixels_;
  }

private:
  static const int WIDTH = 160;
  static const int HEIGHT = 120;

  char pixels_[WIDTH * HEIGHT];
};
```

У него есть базовые операции очистки всего буфера в указанный цвет и установки цвета отдельного пикселя. Еще у него есть функция `getPixels()`, скрывающая за собой массив сырых данных в памяти, хранящий данные пикселей. Мы не увидим ее в примере, но видеодрайвер будет часто использовать такую функцию для пересылки содержимого буфера на экран.

Обернем этот сырой буфер классом `Scene`. Его задача заключается в отрисовке чего-либо с помощью вызовов `draw()` своего буфера:

```
class Scene
{
public:
  void draw() {
    buffer_.clear();

    buffer_.draw(1, 1);
    buffer_.draw(4, 1);
    buffer_.draw(1, 3);
    buffer_.draw(2, 4);
    buffer_.draw(3, 4);
    buffer_.draw(4, 3);
  }

  Framebuffer& getBuffer() { return buffer_; }

private:
  Framebuffer buffer_;
};
```

> Конкретно этот код рисует вот такой замечательный шедевр:
>
> 

На каждом кадре игра командует сцене отрисоваться. Сцена очищает буфер и затем отрисовывает один за другим кучу пикселей. Еще она предоставляет доступ к внутреннему буферу через `getBuffer()`, чтобы видеодрайвер мог получить нужные ему данные.

Выглядит достаточно прямолинейно, но если оставить все как есть, у нас будут серьезные проблемы. Проблема в томб что видеодрайвер может вызвать `getPixels()` у буфера в *любое* время, даже здесь:

```
buffer_.draw(1, 1);
buffer_.draw(4, 1);
// <— Здесь видеодрайвер считывает пиксели!
buffer_.draw(1, 3);
buffer_.draw(2, 4);
buffer_.draw(3, 4);
buffer_.draw(4, 3);
```

Когда такое происходит, пользователь увидит глаза на лице, а рот на один кадр пропадет. На следующем кадре отрисовка прервется в какой-либо еще точке. В результате у нас получится ужасно моргающая графика. Исправить это можно добавлением второго буфера.

```
class Scene
{
public:
  Scene()
  : current_(&buffers_[0]),
    next_(&buffers_[1])
  {}

  void draw() {
    next_->clear();

    next_->draw(1, 1);
    // ...
    next_->draw(4, 3);

    swap();
  }

  Framebuffer& getBuffer() { return *current_; }

private:
  void swap() {
    // Just switch the pointers.
    Framebuffer* temp = current_;
    current_ = next_;
    next_ = temp;
  }

  Framebuffer  buffers_[2];
  Framebuffer* current_;
  Framebuffer* next_;
};
```

Теперь у `Scene` есть два буфера, хранящиеся в массиве `buffers_`. Мы не ссылаемся на них из массива напрямую. Вместо этого у нас есть два члена класса `next_` и `current_`, указывающие на массив. Когда мы рисуем, мы выполняем отрисовку на следующий буфер, на который ссылается `next_`. А когда видеодрайверу нужно считать значение пикселя, он всегда обращается к другому буферу через `current_`.

Таким образом видеодрайвер никогда не видит буферб с которым мы в данный момент работаем. Единственный оставшийся кусочек пазла — это вызов `swap()` после тогоб как сцена заканчивает отрисовывать кадр. Он меняет местами два буфера, просто обменивая между собой указатели в `next_` и `current_`. В следующий раз, когда видеодрайвер вызовет `getBuffer()`, он обратится к новому буферуб в который мы только что закончили рисовать и выведет последний кадр на экран. И никаких больше разрывов и неприятных глитчей.

Не только графикой единой

Суть проблемы, решаемой двойной буферизацией заключается в доступе к состоянию во время его модификации. На это есть две причины. Первую мы упоминали в примере с графикой: код из другого потока или прерывания напрямую получает доступ к состоянию.

А вот еще одна распространенная причина: когда код *производит модификацию*, он получает доступ к тому же состоянию, которое изменяет. Это встречается во множестве областей, особенно в физике и `ии` (Искусственный интеллект) , где сущности друг с другом взаимодействуют. Двойная буферизация поможет и в этом случае.

Искусственный интеллект

Давайте представим себеб что мы разрабатываем поведенческую систему для игры по мотивам гротескной буффонады (http://ru.wikipedia.org/wiki/Буффонада или http://en.wikipedia.org/wiki/Slapstick). В игре имеется сцена, в которой участвует куча актеров, творящих всякие шутки и трюки. Вот базовый актер:

```
class Actor
{
public:
  Actor() : slapped_(false) {}

  virtual ~Actor() {}
  virtual void update() = 0;

  void reset()      { slapped_ = false; }
  void slap()       { slapped_ = true; }
  bool wasSlapped() { return slapped_; }

private:
  bool slapped_;
};
```

На каждом кадре игра отвечает за то чтобы вызвать `update()` актера. Таким образом он может что-либо сделать. С точки зрения игрока критически важно, чтобы *обновления всех актеров выглядели одновременными*.

> А это уже пример шаблона Метод обновления (Update Method).

Кроме этого, актеры могут взаимодействовать друг с другом и под "взаимодействовать" я понимаю "давать друг другу пощечины". Во время обновления актер вызывает метод `slap()` другого актера, чтобы дать ему пощечину и вызывает

`wasSlapped()`, чтобы определить получил ли пощечину сам.

Актерам потребуется декорация, в которой они будут взаимодействовать:

```
class Stage
{
public:
  void add(Actor* actor, int index) {
    actors_[index] = actor;
  }

  void update() {
    for (int i = 0; i < NUM_ACTORS; i++) {
      actors_[i]->update();
      actors_[i]->reset();
    }
  }

private:
  static const int NUM_ACTORS = 3;

  Actor* actors_[NUM_ACTORS];
};
```

`Scene` позволяет нам добавлять актеров и предоставляет единый вызов `update()`, обновляющий всех актеров. Несмотря на то, что для зрителя актеры выглядят действующими одновременно, на самом деле они обновляются один за другим.

Единственное, о чем нужно упомянуть — это то, что каждое состояние "получил пощечину" очищается сразу после обновления. Это сделано для того, чтобы каждый актер реагировал на пощечину только один раз.

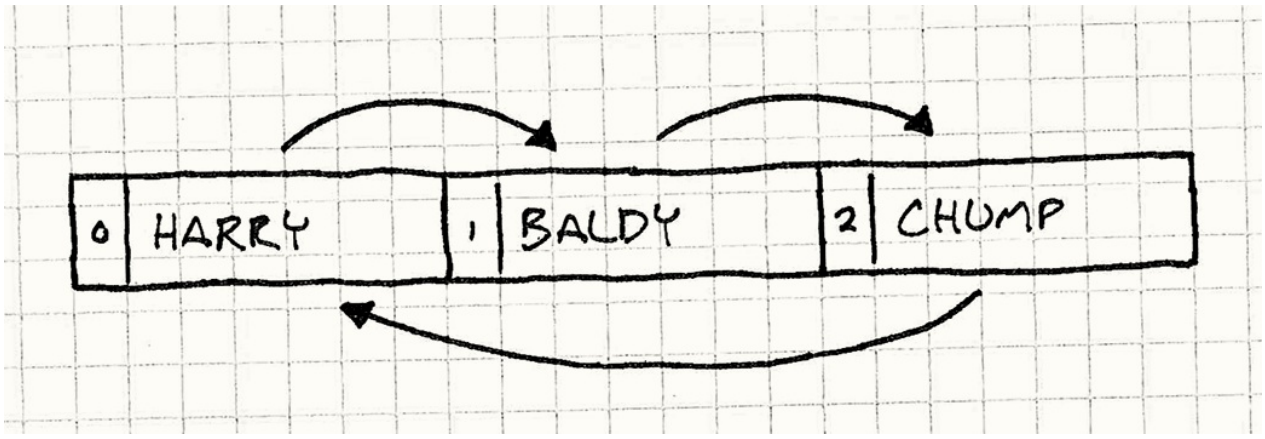
Чтобы все заработало, давайте определим конкретный подкласс актера. Наш комедиант довольно прост. Он находится напротив другого актера. Когда он получает пощечину (от кого угодно) — он реагирует на пощечину актера, который находится перед ним.

```
class Comedian : public Actor
{
public:
  void face(Actor* actor) { facing_ = actor; }

  virtual void update() {
    if (wasSlapped()) facing_->slap();
  }

private:
  Actor* facing_;
};
```

Теперь запустим в декорацию несколько комедиантов и посмотрим, что получится. Добавим трех комедиантов, каждый из которых смотрит на следующего. Последний смотрит на первого, замыкая получившийся круг:

```
Stage stage;

Comedian* harry = new Comedian();
Comedian* baldy = new Comedian();
Comedian* chump = new Comedian();

harry->face(baldy);
baldy->face(chump);
chump->face(harry);

stage.add(harry, 0);
stage.add(baldy, 1);
stage.add(chump, 2);
```

Получившаяся декорация выглядит следующим образом. Стрелки показывают кто на кого смотрит, а номера обозначают индекс в массиве декорации.



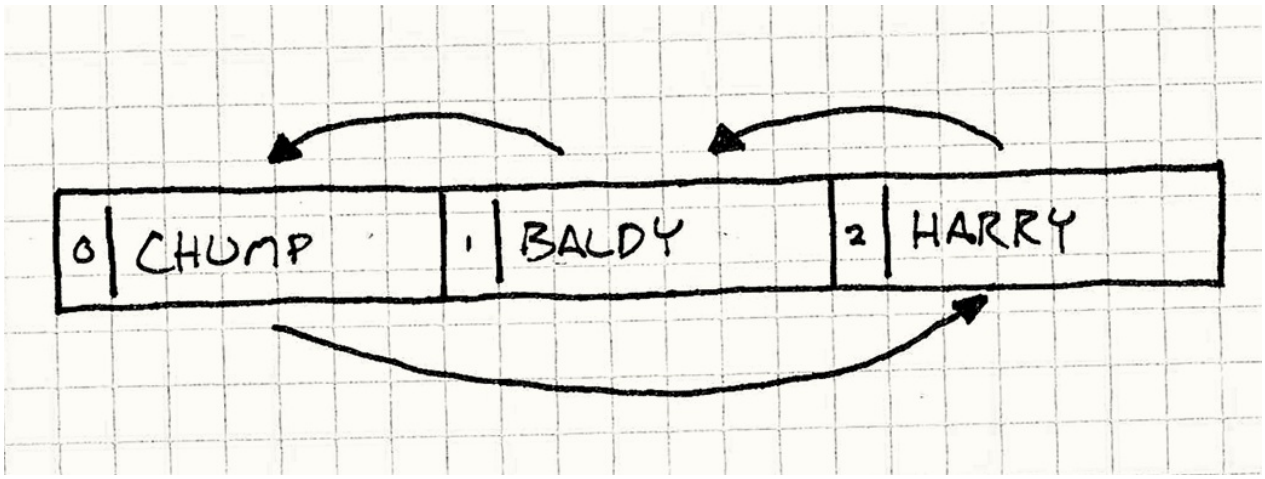
Дадим пощечину Гарри и посмотрим что из этого получится, когда мы запустим обновление.

```
harry->slap();

stage.update();
```

Помните, что функция декорации `update()` обновляет актеров по-очереди, так, что если мы проследим, что происходит в коде, мы обнаружим следующее:

```
Stage updates actor 0 (Harry)
Harry was slapped, so he slaps Baldy
Stage updates actor 1 (Baldy)
Baldy was slapped, so he slaps Chump
Stage updates actor 2 (Chump)
Chump was slapped, so he slaps Harry
Stage update ends
```

Итак, в единственном кадре наша начальная пощечина Гарри прошлась по всем комедиантам. Теперь, чтобы немного разнообразить ситуацию, мы поменяем актеров в массиве декорации местами, но смотреть они будут друг на друга по-прежнему.



Не будем трогать остальную часть декорации, а просто заменим код с добавлением актеров на следующий:

```
stage.add(harry, 2);
stage.add(baldy, 1);
stage.add(chump, 0);
```

Давайте посмотрим что произойдет, когда мы запустим наш эксперимент снова:

```
Stage updates actor 0 (Chump)
  Chump was not slapped, so he does nothing
Stage updates actor 1 (Baldy)
  Baldy was not slapped, so he does nothing
Stage updates actor 2 (Harry)
  Harry was slapped, so he slaps Baldy
Stage update ends
```

Ух ты! Совсем другое дело. Проблема очевидна. Когда мы обновляем актеров, мы изменяем состояние "получил пощечину", т.е. то же самое состояние, которое мы читаем во время обновления. Из—за этого сделанные вначале процесса обновления изменения начинают влиять на то, что происходит дальше в процессе того же самого шага обновления.

В конце концов получается так, что актеры начинают реагировать на пощечину либо на том же самом кадре, либо на следующем только на основании того, в какой очередности они находятся в декорации. Это нарушает наше главное требование — обеспечение видимости одновременного действия всех актеров: порядок, в котором расположены актеры, не должен влиять на результаты обновления в каждом кадре.

> Если вы и дальше продолжите наблюдение за происходящим — вы заметите что пощечины распространяются каскадно, одна пощечина за кадр. На первом кадре Гарри бьет Балди. На следующем Балди бьет Чампа и т.д.

Буферизация пощечин

К счастью, нам может помочь наш шаблон *Двойной буферизации*. На этот раз, вместо того, чтобы заводить две монолитные копии "буферизуемого" объекта, мы буферизуем гораздо более мелкую сущность — состояние "получил пощечину" у каждого из актеров:

```
class Actor
{
public:
  Actor() : currentSlapped_(false) {}

  virtual ~Actor() {}
  virtual void update() = 0;

  void swap() {
    // Swap the buffer.
    currentSlapped_ = nextSlapped_;

    // Clear the new "next" buffer.
    nextSlapped_ = false;
  }

  void slap()       { nextSlapped_ = true; }
  bool wasSlapped() { return currentSlapped_; }

private:
  bool currentSlapped_;
  bool nextSlapped_;
};
```

Вместо единственного состояния `slapped_`, у каждого актера будет два. Как и в предыдущем графическом примере, текущее состояние используется для считывания, а следующее — для записи.

Функция `reset()` заменяется функцией `swap()`. Теперь прямо перед очисткой подменяемого состояния, оно копирует следующее состояние в текущее и делает его новым текущим. Для этого требуется внести небольшое изменение в `Stage`:

```
void Stage::update()
{
  for (int i = 0; i < NUM_ACTORS; i++) {
    actors_[i]->update();
  }

  for (int i = 0; i < NUM_ACTORS; i++) {
    actors_[i]->swap();
  }
}
```

Функция `update()` теперь обновляет всех актеров и только после этого подменяет все их состояния.

В конце концов, у нас получится такая система, в которой актер видит пощечину только на следующем кадре после того, как она была нанесена. И в этом случае актеры будут действовать одинаково, вне зависимости от порядка нахождения в массиве. Для пользователя или внешнего кода обновление актеров будет выглядеть одновременным.

Архитектурные решения

Двойная буферизация — шаблон очень прямолинейный, поэтому рассмотренные нами примеры покрывают большую часть вариантов использования. Самое главное в реализации шаблона — это принять решение по двум следующим вопросам.

Как мы будем переключать буферы?

Операция переключения — наиболее критичная часть процесса, потому что во время ее выполнения оба буфера блокируются как для чтения так и для записи. Чтобы добиться наилучшей производительности, эта операция должна происходить настолько быстро, насколько это только возможно.

- **Переключение указателей, ссылающихся на буферы:**

  Именно так работает пример с графикой и это самое распространенное решение двойной буферизации графики.

  - *Это быстро.* Неважно какого размера сам буфер, потому что все переключение представляет собой простое переключение указателей. Сложно придумать что-то более быстрое и более простое.
  - *Внешний код не может хранить постоянный указатель на буфер.* Это основное ограничение. Так как мы не переносим непосредственно сами данные, все что мы делаем — это просто указываем кодовой базе искать буфер в новом месте, как в нашем первом примере с декорациями. Это значит, что оставшаяся кодовая база не может хранить указатели на данные внутри буфера, потому что через некоторое время он будет указывать на неправильный адрес.

    В системах, где видеобуфер должен находиться в строго определенном месте в памяти, это может быть большой проблемой. В этом случае у нас есть несколько вариантов.

- *Находящиеся в буфере данные будут отставать на два кадра, а не относиться к последнему кадру*. Удачные кадры рисуются в альтернативный буфер без копирования данных между ними следующим образом:

  ```
  Frame 1 drawn on buffer A
  Frame 2 drawn on buffer B
  Frame 3 drawn on buffer A
  ...
  ```

  Вы можете заметить, что когда мы отрисовываем третий кадр, данные, уже находящиеся в буфере, относятся к первому кадру, а не к более свежему второму. В большинстве случаев это не проблема — мы просто очищаем весь буфер перед отрисовкой. Но если мы хотим использовать часть данных повторно, важно учитывать то, что данные могут быть на кадр старше, чем мы рассчитываем.

> Одно из классических применений старого буфера кадра — это эффект размытия движения. Текущий кадр смешивается с частью ранее отрендеренного кадра таким образом чтобы получилось подобие того, что фиксирует реальная камера.

- **Копирование данных между буферами:**

  Если мы не можем перенаправить пользователей на другой буфер, единственным выходом остается полное копирование данных из следующего кадра в текущий. Именно так работают наши драчливые комедианты. В этом случае мы выбрали такой вариант потому, что состояние — это всего лишь булевский флаг и его копирование занимает не больше времени, чем указателя на сам буфер.

  - *Данные в следующем буфере отстают только на один кадр*. Это преимущество копирования данных над простым перебрасыванием двух буферов. Если нам понадобится доступ к предыдущим данным из буфера, они будут более свежими.
  - *Обмен может занять больше времени*. Это конечно основной недостаток. Наша операция обмена теперь означает копирование в памяти всего буфера. Если буфер достаточно большой, как например весь буфер кадра целиком, на это может потребоваться слишком много времени. Так как во время обмена никто не может ни читать буфер, ни писать в него — это серьезное ограничение.

Какова дробность самого буфера?

Еще один вопрос — это организация самого буфера: является ли он единым монолитным куском данных или разрозненным набором объектов. Наш графический пример — это первое, а пример с актерами — второе.

Чаще всего ответ кроется в природе того, что вы буферизуете, но пространство для маневра все-равно остается. Например, наши актеры могут хранить все свои сообщения в едином блоке сообщений и обращаться к ним по индексу.

- **Если буфер монолитный:**
  - *Обмен проще*. Так как у нас есть всего пара буферов, нам достаточно одной операции обмена. Если обмен осуществляется переназначением указателей, мы можем осуществить его всего нескольким операциями, вне зависимости от его размера.
- **Если у многих объектов есть свой кусочек данных:**
  - *Обмен медленнее*. Чтобы его выполнить, нам нужно обойти всю коллекцию объектов и выполнить обмен для каждого.

    В нашем примере с комедиантами, это нормально, потому что нам все равно нужно очищать состояние следующей пощечины — каждый кусочек буферизованного состояния нужно трогать на каждом кадре. Если же нам не нужно затрагивать старый буфер, мы можем применить простую оптимизацию, чтобы получить для разделенного на множество кусочков состояния такую же производительность, как и для монолитного буфера.

    Идея заключается в том, чтобы взять концепцию "текущего" и "следующего" указателей и применить ее к каждому объекту, превратив их в *смещение* относительно объекта. Примерно таким образом:

```
class Actor
{
public:
static void init() { current_ = 0; }
static void swap() { current_ = next(); }

void slap()        { slapped_[next()] = true; }
bool wasSlapped()  { return slapped_[current_]; }

private:
static int current_;
static int next()  { return 1 - current_; }

bool slapped_[2];
};
```

Актеры получат свое состояние пощечины, используя `current_` для индексации в массиве состояния. Следующее состояние всегда будет в другом индексе массива, так что мы можем вычислить его через `next()`. Обмен состояния — это просто альтернатива индексу `current_`. Весьма разумно сделать `swap()` *статичной* функцией: ее нужно вызывать всего один раз и состояния всех актеров сразу будут обменены.

Смотрите также

- Шаблон *Двойная буферизация* можно найти практически в любом графическом `API`. В `OpenGL` есть `swapBuffers()`, в `Direct3D` — "цепочки обмена" (swap chains). А фреймворк XNA от Microsoft выполняет обмен буферов с помощью функции `endDraw()`.

Игровой цикл(Game Loop)

Задача

Устранить зависимость игрового времени от пользовательского ввода и скорости процессора.

Мотивация

Если бы меня спросили, без какого шаблона из этой книги я не смог бы жить, я вспомнил бы именно об этом. Игровой цикл — это квинтэссенция примера "шаблона в игровом программировании". Он есть практически в каждой игре и двух одинаковых практически нет. И при этом в не играх он встречается крайне редко.

Чтобы увидеть насколько он полезен, давайте вспомним прошлое. В старые времена компьютерного программирования у всех были бороды, а программы работали как посудомоечные машины: вы загружали в нее код, нажимали кнопку, ждали и забирали результат. Готово. Это называлось *пакетным режимом* (batch mode): как только работа была сделала, программа останавливалась.

> У Ады Лавлейс и контр-Адмирала Грейс Хоппер бороды были почетными.

Такое можно увидеть и поныне. Слава богу хотя бы с перфокартами больше заморачиваться не нужно. Скрипты терминалов, консольные программы и даже маленькие скрипты на `Python` для форматирования кода в этой книге — это все примеры работы в пакетном режиме.

Интервью с процессором

В конце концов программисты поняли, что отправка программы в вычислительный центр и получение результатов через несколько часов- не самый лучший способ работы и отлова багов в программах. Хотелось немедленного отклика. И тогда появились *интерактивные* программы. Одними из первых интерактивных программ были игры:

```
YOU ARE STANDING AT THE END OF A ROAD BEFORE A SMALL BRICK
BUILDING . AROUND YOU IS A FOREST. A SMALL
STREAM FLOWS OUT OF THE BUILDING AND DOWN A GULLY.

> GO IN
YOU ARE INSIDE A BUILDING, A WELL HOUSE FOR A LARGE SPRING.
```

> Это Colossal Cave Adventure — первый текстовый квест.

Теперь вы могли общаться с программой вживую. Она ожидала вашего ввода и затем реагировала на него. Затем наступала ваша очередь реагировать — ну прямо как в детском саду учат. Когда была ваша очередь действовать — программа ничего не делала. Примерно вот так:

```
while (true)
{
  char* command = readCommand();
  handleCommand(command);
}
```

Циклы событий

Современные программы с графическим пользовательским интерфейсом, если снять с них оболочку, поразительно напоминают старые текстовые квесты. Ваш текстовый процессор обычно просто сидит и ничего не делает до тех пор, пока вы не нажмете какую-либо клавишу:

```
while (true)
{
  Event* event = waitForEvent();
  dispatchEvent(event);
}
```

Единственное различие здесь в том, что вместо *текстовых команд*, программа ожидает пользовательского ввода — нажатия мыши и клавиш. Но в основе лежит тоже самое, что и в текстовых квестах, где программа *блокируется* в ожидании пользовательского ввода, что на самом деле является проблемой.

В отличие от других программ, игры продолжают работать даже когда пользователь не предоставляет никакого ввода. Если вы будете просто смотреть на экран, игра не остановится. Анимации продолжат проигрываться. Визуальные эффекты танцуют и блестят. А если вам не повезет, монстр продолжит понемногу грызть вашего героя.

> Большинство циклов событий содержат события "простой (idle)", во время которых можно продолжать работать без пользовательского ввода. Для мигающего курсора или прогресс бара это нормально, но для игры не годится.

Вот мы и подошли к первой ключевой особенности игрового цикла: *он обрабатывает пользовательский ввод, но не ожидает его*. Цикл продолжает крутиться всегда:

```
while (true)
{
  processInput();
  update();
  render();
}
```

Позже мы рассмотрим его подробнее, но в основе лежат именно эти вещи. `processInput()` обрабатывает пользовательский ввод с момента прошлого вызова. `update()` продвигает симуляцию игры на один шаг. Сюда входят `ИИ` и физика (обычно именно в таком порядке). И наконец `render()` рисует игру, чтобы игрок увидел происходящее.

> Как можно догадаться из имени `update()` — это подходящее место для применения шаблона Метод обновления (Update Method)

Мир вне времени

Если этот цикл не блокируется при вводе — возникает резонный вопрос: насколько *быстро* он крутится? На каждом шаге игрового цикла состояния игры немного продвигается вперед. С точки зрения обитателей игрового мира это заставляет их время идти вперед.

> Обычно единица игрового цикла называется "тиком" или "кадром".

При этом у *игрока* время тоже тикает. Если мы измерим скорость выполнения циклов в единицах реального времени, мы получим единицу измерения "кадры в секунду" или `FPS`. Если игровые циклы сменяются быстро, `FPS` высок, а игра работает быстро и плавно. Если медленно — игра подтормаживает и становится похожей на кино в замедленном воспроизведении.

В нашем примитивном игровом цикле, который старается сменяться как можно чаще, частоту кадров определяют два фактора. Первый — это *сколько работы нужно выполнять на каждом кадре*. Сложная физика, куча игровых объектов и детализированная графика могут настолько нагрузить ваши процессор и видеокарту, что на обработку кадра понадобится очень много времени.

Второй фактор — это *производительность платформы*. Быстрые чипы могут перемалывать гораздо больше кода за то же время. Количество ядер, видеокарта, дискретный аудио чип и планировщик ОС — все это влияет на количество действий, которые можно успеть выполнить за один тик.

Секунды в секунду

В каждой игре второй фактор всегда фиксирован. Если вы пишете игру для `NES` или `Apple II`, вы *точно* знаете какой процессор у вас будет и можете на него рассчитывать. Все о чем вам нужно заботиться — это каким объемом работы вы его нагружаете.

Старые игры разрабатывались таким образом, чтобы выполняемая работа позволяла игре работать с нужной скоростью. Но если бы вы попробовали поиграть в ту же самую игру на более быстром или медленном компьютере, сама игра стала бы работать быстрее или медленнее.

Сейчас редко кто из разработчиков может позволить себе роскошь знать на каком железе будет работать их игра. Вместо этого играм приходится адаптироваться под большое разнообразие конфигураций.

> У старых компьютеров для этого даже была отдельная кнопка "turbo". Новые компьютеры были быстрее и не позволяли играть в старые игры, потому что те работали слишком быстро. *Отключение* режима `turbo` замедляло компьютер и игра снова становились нормальной.

Таким образом мы подошли еще к одной важной особенности игрового цикла: *он обеспечивает постоянную скорость игры в независимости от аппаратного обеспечения*.

Шаблон

Игровой цикл работает на протяжении всей игры. На каждом своем цикле игровой цикл **обрабатывает пользовательский ввод** без блокировки, **обновляет состояние игры** и **рендерит игру**. А еще он следит за ходом времени и **управляет скоростью игрового процесса**.

Когда использовать

Использовать неподходящий шаблон — хуже, чем не использовать никаких шаблонов вовсе, так что обычно этот раздел призван предостеречь вас от излишнего энтузиазма. Задачей шаблонов проектирования не является проникновение в вашу кодовую базу в максимальном количестве.

Но этот шаблон не такой как другие. Я с уверенностью могу сказать, что вы *будете* его использовать. Если вы используете готовый движок, вы его не пишете сами, но он все равно присутствует.

> На мой взгляд в этом и заключается отличие "движка" от "библиотеки". Используя библиотеку, вы организуете игровой цикл самостоятельно и используете из него библиотеку. Движок наоборот организует игровой цикл самостоятельно и вызывает из него *ваш* код.

Вам может показаться, что для пошаговой игры он не нужен. Но даже там, несмотря на то, что *состояние игры* не обновляется до того как игрок сделает шаг, *визуальное* и *аудио* состояние все равно продолжают обновляться. Анимации и музыка все равно продолжают проигрываться, даже если игра ждет, пока вы сделаете свой шаг.

Имейте в виду

Цикл, о котором мы все время тут говорим — это самая главная часть кода в вашей игре. Говорят что 90% времени в программе используется 10% кода. Ваш игровой цикл определенно попадает в эти 10%. Заботьтесь о нем и обращайте внимание на его производительность.

> Вот из-за подобной статистики "настоящие" инженеры и механики и не принимают нас всерьез.

Вам придется взаимодействовать с циклом сообщений вашей платформы

Если вы строите вашу игру поверх ОС или платформы с графическим интерфейсом и встроенным циклом сообщений, вашей игре придется взаимодействовать с двумя циклами. Нужно заставить их уживаться между собой.

Иногда у вас есть возможность взять ситуацию под контроль и ограничиться одним циклом. Например, если вы пишете игру и не хотите связываться с почтенным `Windows API`, внутри вашей функции `main()` может находиться игровой цикл. Внутри вы можете вызвать `PeekMessage()` для обработки и удаления сообщений от `ОС`. В отличие от `GetMessage()`, `PeekMessage()` не блокируется в ожидании пользовательского ввода и ваш игровой цикл не будет тормозиться.

Другие платформы не позволяют вам обращаться с циклом сообщений столько вольно. Если вы пишете игру для браузера, цикл сообщений вшит глубоко внутри самого браузера. Так как цикл сообщений заправляет всем самостоятельно, вам придется использовать его и в качестве игрового цикла. Вы вызываете нечто наподобие `requestAnimationFrame()` и он уже через калбек (callback) вызовет ваш код, чтобы игра продолжала работать.

Пример кода

Несмотря на такое длинное вступление, сам код игрового цикла довольно прямолинеен. Мы рассмотрим несколько вариантов и отметим их достоинства и недостатки.

Игровой цикл управляет `ИИ`, рендерингом и другими игровыми системами, но они ни в коей мере не являются частью самого шаблона. Они просто из него вызываются. Так что реализация (непростая!) `render()`, `update()` и остальных оставляется читателю.

Беги, беги настолько быстро, насколько можешь

Мы уже видели самый примитивный игровой цикл:

```
while (true)
{
  processInput();
  update();
  render();
}
```

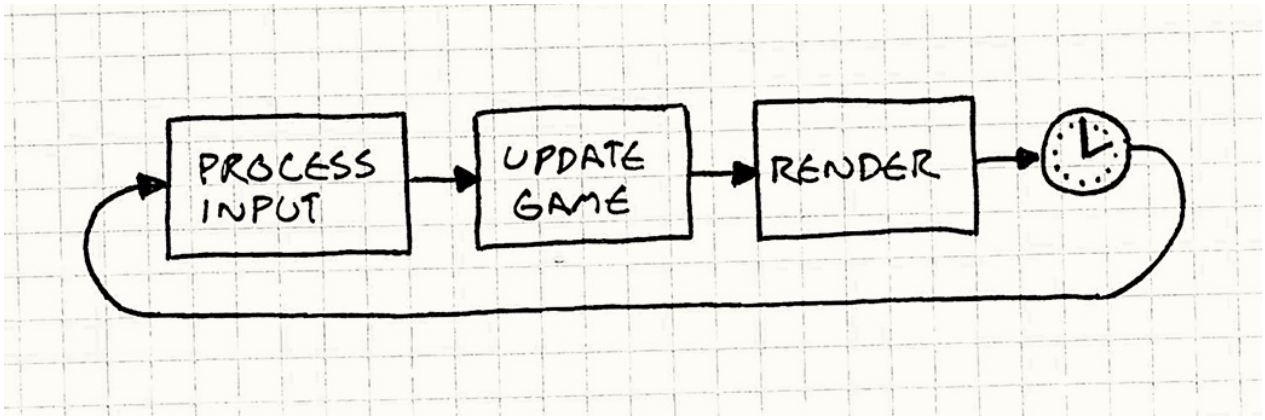
Его главная проблема в том, что он никак не управляет скоростью игры. На быстрой машине он будет работать так быстро, что пользователь даже не разберет, что происходит. На медленной машине игра будет просто тормозить. Если в какой-то части игры у вас есть сложный `ИИ` или физика, игра тоже будет замедляться в этих местах.

Передохнем немного

Первый вариант, который мы рассмотрим добавляет простой фикс. Скажем мы хотим чтобы наша игра работала с `FPS` 60. Это около 16 миллисекунд на кадр.

> 1000мс/FPS = мс на кадр

Пока вы сможете уложить все свои вычисления и рендеринг в это время, у вас будет постоянный фреймрейт. Все что вам нужно будет сделать — это после обработки каждого кадра ожидать, пока наступит время для следующего примерно таким образом:



Код изменится таким образом:

```
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();

  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

`sleep()` здесь контролирует, чтобы игра не работала слишком быстро, если успела обработать кадр раньше времени. Но это *не поможет*, если ваша игра работает слишком *медленно*. Если игра будет тратить на цикл больше 16 мс, время сна станет *отрицательным*. Если бы у нас был компьютер, способный перемешаться во времени в прошлое, все было бы просто, но увы.

Вместо этого игра просто замедляется. Вы можете избежать этого делая меньше работы на каждом цикле: урезать и упростить графику или оглупить `ии`. Но это повлияет на качество игрового процесса даже на быстрых машинах.

Один маленький шаг и один гигантский шаг

Давайте попробуем кое-что посложнее. Проблема, с которой мы столкнулись, разбивается на следующие:

1. На каждом цикле время игры немного продвигается вперед.

2. Этот процесс требует некоторого количества реального времени.

Если шаг второй займет больше времени, чем первый, игра замедлится. Если на то, чтобы продвинуть время в игре на 16 мс придется потратить больше 16 мс — это не годится. А вот если за один шаг мы сможем обработать *больше* чем 16 мс игры, мы сможем обновлять игру уже не так часто.

Идея выбора длины шага основывается на количестве *реального* времени, прошедшего с прошлого кадра. Чем больше времени занял кадр, тем больше шагов делает игра. Мы всегда будем успевать за реальным временем потому что будем выполнять все больше и больше шагов. Такое время шага можно назвать *переменным* или *гибким*. Выглядит это следующим образом:

```
double lastTime = getCurrentTime();
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - lastTime;
  processInput();
  update(elapsed);
  render();
  lastTime = current;
}
```

На каждом кадре мы определяем сколько реального времени прошло с момента последнего обновления игры (`elapsed`). А когда мы обновляем состояние игры, мы передаем это значение внутрь. Далее движок отвечает за то, чтобы обновить игру на переданное количество времени.

Представим, что у нас через весь экран летит пуля. При фиксированной длительности шага, на каждом кадре она будет двигаться в соответствии со своей скоростью. С переменой длительностью шага вам придется *масштабировать скорость в зависимости от прошедшего времени*. Как только шаг увеличивается, пуля летит быстрее, чем на прошлом кадре. Главное, что экран целиком пуля пролетит за *одно* и то же количество *реального* времени, независимо от того, произойдет это за 20 маленьких быстрых шагов или за четыре больших медленных. Похоже на то, что нужно:

- Игра работает с одинаковой скоростью на любом оборудовании.
- Игроки с более мощными компьютерами вознаграждаются более плавным геймплеем.

Но, к сожалению, у нас появляется и одна серьезная проблема: игра получается недетерминированной и нестабильной. Вот пример того, в какую ловушку мы угодили:

Предположим, у нас идет сетевая игра, и Фред играет на ультрамощном PC, а Джордж на старом нетбуке своей бабушки. Упомянутая выше пуля пролетает по экрану у них обеих. На компьютере Фреда игра летает и каждый шаг занимает очень мало времени. У него пуля пролетает экран за секунду и происходит это за 50 кадров. У бедняги Джорджа на весь этот процесс отводится только пять кадров.

Это значит, что на компьютере Фреда физический движок обновляет позицию пули 50 раз в секунду, а у Джорджа — всего пять. Большинство игр используют при расчетах вещественные числа, которые очень подвержены *ошибкам округления*. Каждый раз, когда вы складываете два вещественных числа, результат получается немного другим. Машина Фреда работает в десять раз быстрее и на ней накапливается большая ошибка чем у Джорджа. В результате через секунду полета *пуля* окажется на этих двух машинах в *разных местах*.

> "Детерминированность" означает, что каждый раз когда вы будете запускать программу и подавать на вход один и тот же пользовательский ввод, вы получите один и тот же результат на выходе. Как вы понимаете, в детерминированной программе куда проще искать баги: достаточно найти входные данные, порождающие баг и его можно будет воспроизводить каждый раз.
>
> Компьютеры по своей сути детерминированы: они просто механически следуют инструкциям. Недетерминированность начинается там, где вмешивается внешний мир. Например, сеть, системное время и планировщик потоков опираются на внешние вещи за пределами контроля программы.

Это одна из самых неприятных проблем, возникающих при переменном временном шаге, но есть и другие. Для того, чтобы физический движок работал в реальном времени, правила реальной механики приходится аппроксимировать. Чтобы эти аппроксимации не взорвались, их приходится глушить. Такое приглушение требует крайне чувствительной подстройки к конкретному временному шагу. Стоит изменить этот шаг, и физика станет нестабильной.

> "Взорвались" — это конечно образно. Когда физический движок начинает глючить, объекты получают совершенно неверное ускорение и улетают в небеса.

Вся эта нестабильность настолько плоха, что я упомянул здесь этот пример только в качестве предупреждения и как шаг к гораздо лучшему решению...

Догонялки

Обычно рендеринг — это та часть движка, на которую переменное время *не влияет*. Так как рендеринг происходит в конкретное время, для него не важно сколько времени прошло с момента прошлого рендеринга. Вещи рендерятся в том состоянии, в котором находятся в данный момент.

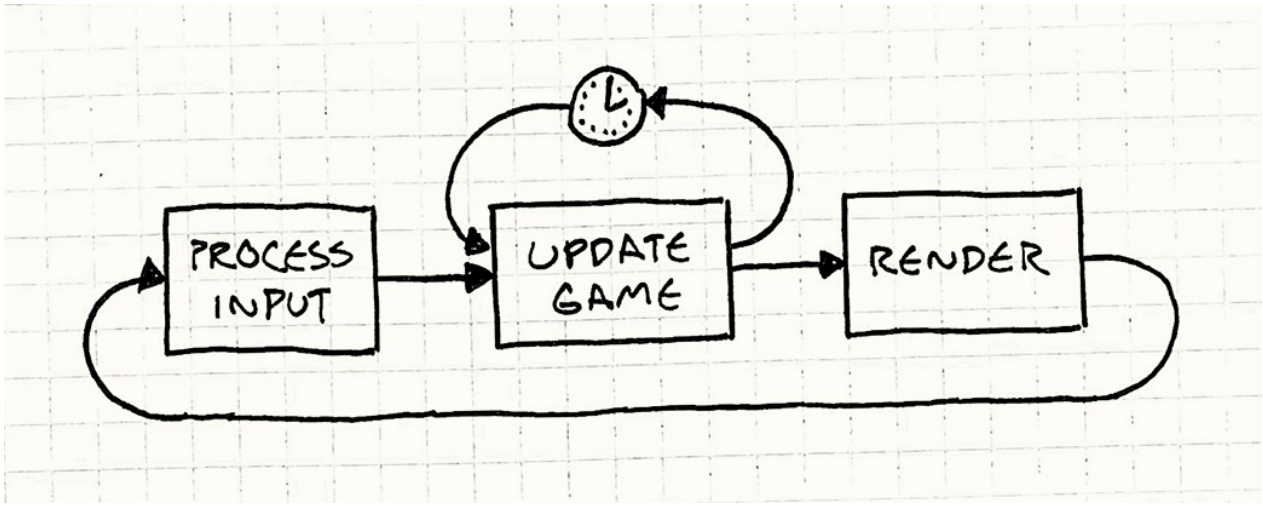
> Это более-менее справедливое утверждение. Эффекты типа размытия движения могут зависеть от результата предыдущего рендеринга. Но если время с момента прошлого рендеринга будет постоянно немного варьироваться, игрок вряд ли заметит разницу.

Мы можем использовать этот факт в свою пользу. *Обновлять* игру мы будем фиксированными шагами, потому что это проще и в плане `ии` и физики гораздо стабильнее. Но вот в плане *рендеринга* мы позволим себе некоторую гибкость для того, чтобы сэкономить немного процессорного времени.

Поступим следующим образом. С момента окончания прошлого игрового цикла у нас прошло некоторое время. Именно столько времени нам нужно просимулировать в игре, чтобы отобразить текущее состояние игроку. Добьемся мы это с помощью *серии фиксированных* временных шагов. У нас получится такой код:

```
double previous = getCurrentTime();
double lag = 0.0;
while (true) {
  double current = getCurrentTime();
  double elapsed = current - previous;
  previous = current;
  lag += elapsed;

  processInput();

  while (lag >= MS_PER_UPDATE) {
    update();
    lag -= MS_PER_UPDATE;
  }

  render();
}
```

Он состоит из нескольких частей. В начале каждого кадра мы обновляем `lag` на основе прошедшего реального времени. Это значение обозначает насколько наше игровое время отстало от реального. Далее мы будем обновлять состояние игры шагами фиксированной длины до тех пор, пока не догоним реальное время. Как только мы его догнали, выполняем рендеринг и начинаем процедуру снова. Визуализировать это можно следующим образом:



Обратите внимание, что временной шаг теперь не соответствует *видимому* фреймрейту. `MS_PER_UPDATE` — это *дробность* наших обновлений игры. Чем короче этот шаг, тем больше процессорного времени нужно для того, чтобы нагнать реальное время. Чем он длиннее — тем грубее геймплей. В идеале он должен быть довольно коротким, чтобы `FPS` был не ниже 60 и игра на быстрых машинах работала плавно.

Но будьте осторожны и не сделайте его *слишком* коротким. Нам нужно быть уверенными, что временной шаг будет больше, чем время, необходимое на обработку `update()` даже на медленном оборудовании. В противном случае игра всегда будет опаздывать и никогда не догонит реальное время.

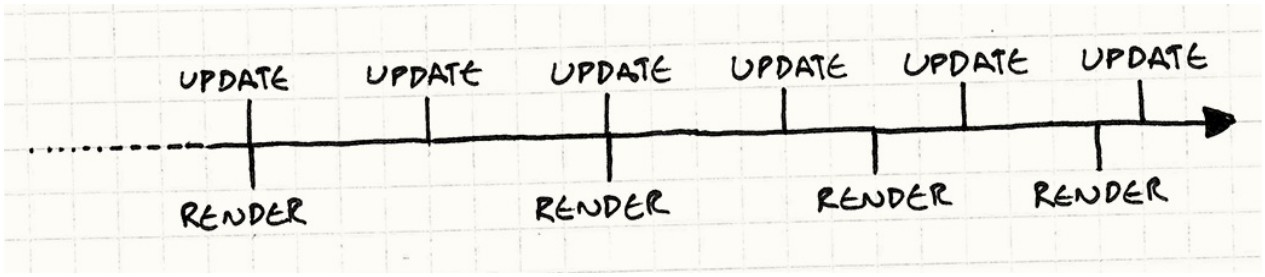
К счастью, теперь у нас есть пища для размышлений. Хитрость здесь в том, что мы *выдернули рендеринг из цикла обновлений*. Это освобождает кучу процессорного времени. В результате сама игра *симулируется* с константной скоростью обновления на самом разном оборудовании. А если игрок видит подтормаживания на слабой машине, то подтормаживает только видимая часть игры.

Я на этом остановлюсь, но вы можете подстраховаться, ограничив количество временных шагов, которые могут выполняться друг за другом некоторым максимумом. Игра замедлится, но по крайней мере не заблокируется насовсем.

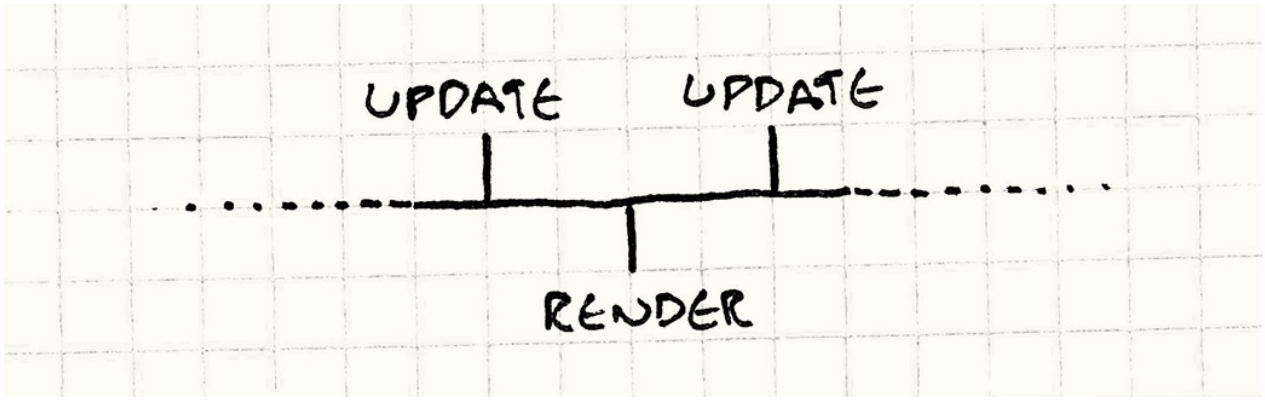
Застрять в середине

У нас осталась одна проблема и это остаточный лаг. Мы обновляем игру фиксированными временными шагами, а рендеринг выполняем в произвольные моменты времени. Это значит, что с точки зрения игрока, игра будет часто рендериться между двумя обновлениями.

Посмотрите на таймлайн:



Как вы видите, мы выполняем обновления довольно частыми, фиксированными интервалами. И время от времени выполняем рендеринг. Он выполняется реже, чем обновления и с не очень стабильным интервалом. И то и другое приемлемо. Не очень хорошо то, что мы не всегда выполняем рендеринг сразу после обновления. Посмотрите на время третьего рендеринга. Он оказался прямо между двумя обновлениями:



Представьте себе летящую через экран пулю. На момент первого обновления она находится слева. На момент второго — справа. Рендеринг произошел ровно посередине между этими двумя событиями, так что игрок может ожидать увидеть пулю прямо в центре экрана. Но в нашей реализации она по прежнему будет находиться слева. И поэтому движение будет смотреться дерганным и запинающимся.

На наше счастье, мы *точно* знаем в какой момент между обновлениями происходит рендеринг: значение хранится в `lag`. Мы пережили предыдущее обновление, с тех пор прошло меньше времени, чем наш шаг обновления и длительность этого времени больше *нуля*. Так где мы оказались? Мы слегка залезли в следующее обновление.

Когда мы попадаем в рендер, мы передаем следующее:

```
render(lag / MS_PER_UPDATE);
```

Наш рендер обладает информацией обо всех объектах *и их текущей скорости*. Скажем пуля находится на 20 пикселей от левого края экрана и двигается со скоростью 400 пикселей за кадр. Если мы на полпути до следующего обновления, мы должны передать в `render()` 0.5. И пулю мы соответственно рисуем в позиции 220 пикселей. Та-дам! Плавное движение.

> Мы разделили здесь `MS_PER_UPDATE` чтобы получить *нормализованное* значение. Значение, передаваемое в `render()` может варьироваться от 0(сразу после прошлого обновления) до 1(прямо перед следующим обновлением), независимо от временного шага обновления. Поэтому рендеру не нужно волноваться о фреймрейте. Он просто оперирует значениями от 0 до 1.

Конечно такая экстраполяция тоже может быть ошибочной. Когда мы на самом деле будем просчитывать следующий кадр, мы можем обнаружить что пуля наткнулась на препятствие или замедлилась чем-то. А мы уже рендерили ее в позиции, полученной интерполяцией предыдущего положения и предполагаемого будущего. Но верно это или нет мы сможем узнать только на следующем полном обновлении физики и `ии`.

Не удивительно, что такие попытки игры в угадывание не всегда заканчиваются успешно. К счастью, они обычно малозаметны. И по крайней мере, они не так заметны, как запинания, которые можно увидеть, если вообще не применять интерполяцию.

Архитектурные решения

Несмотря на длину этой главы, я еще много о чем не рассказал. Если сразу заморачиваться синхронизацией с ходом луча дисплея, мультипоточностью и работой видеокарты, игровой цикл может показаться довольно запутанной штукой. Но на высшем уровне осталось еще несколько вопросов.

Контролируете ли вы игровой цикл сами или это делает платформа?

Зачастую этот выбор уже сделан за вас. Если вы пишете игру для браузера, у вас скорее всего *не получится* написать классический игровой цикл. Событийная модель браузера диктует свои правила. Аналогично, если вы используете готовый движок, вы скорее всего будете использовать встроенный в него игровой цикл, а не писать собственный.

- **Использование цикла событий платформы:**
    - *Это просто*. Вам не придется заботиться о написании и оптимизации собственного игрового цикла.
    - *Хорошо сопрягается с платформой*. Вам не придется заботиться о выделении процессорного времени на нужды платформы, кешировании событий, или наоборот бороться с несоответствием между моделью ввода платформы и вашей.

    - *Вы теряете управление таймингом*. Платформа будет вызывать ваш код по своему усмотрению. И к сожалению, скорее всего это будет происходить не так часто, и не так плавно, как вам хотелось бы. Гораздо хуже то, что большинство циклов событий разработаны без учета нужд игр и обычно работают медленно и грубо.

- **Использование игрового цикла движка:**
    - *Вам не придется его писать*. Написание игрового цикла — дело довольно хитрое. Так как этот код выполняется на каждом кадре, любые баги или проблемы с производительностью будут серьезно влиять на всю вашу игру. Надежный игровой цикл — это одна из причин использовать уже готовый движок.
    - *Вам не удастся его написать*. У предыдущего достоинства есть и обратная сторона. Вам не удастся подстроить игровой цикл под свои нужды, если у вас есть к нему специфические требования.
- **Написание самостоятельно:**
    - *Полный контроль*. Вы можете делать все что хотите. Архитектуру можно специально разрабатывать исходя из нужд вашей игры.
    - *Вам нужен интерфейс для взаимодействия с платформой*. Фреймворки приложения и сама операционная система обычно рассчитывают на периодическое получение отрезка времени для обработки сообщений и другой работы. Если всем управляет только ваш собственный цикл, то ресурсы больше никому не достанутся. Поэтому периодически вам придется отдавать из своих рук управление, чтобы фреймворки нигде не застопорились.

Как это отразится на потреблении энергии?

Еще лет пять назад такой проблемы вообще не существовало. Игры работали на устройствах, постоянно подключенных к сети или специализированных портативных системах. Но теперь, когда так расплодились смартфоны, ноутбуки и вообще мобильный гейминг, вам придется это учитывать. Хорошо работающая игра, но быстро превращающая телефон в печку и сжирающая за полчаса аккумулятор — это не та игра, которая может порадовать людей.

Сейчас вам нужно не только думать о том как сделать игру красивой, но и следить за потреблением процессорного времени. Стоит задуматься о *верхнем* пределе производительности, после которого вы позволите процессору немного поспать, если он сделал все, что было нужно на текущем кадре.

- **Работа с максимальной скоростью:**

  Так мы обычно ведем себя на стационарных компьютерах (даже если игра потом будет запускаться на ноутбуке). Игровой цикл никогда не будет стараться заснуть или передать управление ОС. Вместо этого все свободное время будет посвящено увеличению FPS или улучшению качества графики.

  В результате игра выглядит максимально привлекательно, но использует энергию без всякой меры. Если играть в такую игру на ноутбуке, можно обжечь коленки.

- **Ограничение фреймрейта:**

  Мобильные игры чаще всего концентрируются на геймплее, а не на качестве графики. Многие из этих игр устанавливают предел для фреймрейта (обычно 30 или 60 FPS). Если игровой цикл справляется с обработкой кадра быстрее, чем отведено пределом, он засыпает чтобы передохнуть.

  Таким образом игрок получает "довольно хорошо" работающую игру и не так быстро разряжающуюся батарею.

Каким образом можно управлять скоростью игры?

Игровой цикл состоит из двух частей: неблокирующего пользовательского ввода и обработки прошедшего времени. Ввод — это просто. А вот в работе со временем есть свои хитрости. Существует практически бесконечное количество платформ, на которых могут работать игры и каждая игра работает хотя бы на нескольких. Основная хитрость в том, как справиться с этим разнообразием.

> Похоже игроделанье заложено в человеческую природу, потому что каждый раз когда люди создают вычислительную машину, для нее сразу начинают писать игры. У PDP-1 было всего 4096 слов памяти и процессор 2kHz, а Стив Рассел с компанией все равно написали для нее Spacewar.

- **Фиксированный временной шаг без синхронизации:**

  Это у нас уже было в самом первом примере кода. Вы просто повторяете игровой цикл так быстро, насколько это возможно.

  - *Это просто*. Главное (да и пожалуй единственное) достоинство.
  - *Скорость игры напрямую зависит от скорости железа и сложности самой игры*. Отсюда и главный недостаток — на скорость игры может влиять что угодно. Это такой аналог велосипеда с одной передачей (fixie) в семье игровых циклов.

- **Фиксированные временные шаги с синхронизацией:**

На следующей ступени по степени сложности находится работа игры с фиксированным временным шагом, но с добавлением задержки или точки синхронизации в конце цикла, чтобы игра не работала слишком быстро.

- *Все еще достаточно просто*. Одна дополнительная строка кода — это конечно слишком просто для примера реализации из настоящей игры. В большинстве игровых циклов, вы так или иначе, но все равно занимаетесь синхронизацией. Ведь вы скорее всего используете двойную буферизацию для графики и синхронизируете переключение буфера с частотой обновления монитора.
- *Приемлемо с точки зрения энергосбережения*. Для мобильных игр это на удивление важное преимущество. Нет никакой необходимости убивать батарею пользователя. Позволив игре поспать несколько миллисекунд, вместо того чтобы выгрести всю процессорную мощность, вы сохраняете энергию.
- *Игра больше не работает слишком быстро*. Такой подход решает проблему постоянства скорости работы по сравнению с фиксированным циклом.
- *Игра не может работать слишком медленно*. Если игре придется потратить слишком много времени на обновление и рендеринг, она начнет тормозить. Так как в таком варианте игрового цикла обновление не отделено от рендеринга, он больше подвержен такой проблеме чем более совершенные варианты реализации. Вместо того, чтобы пропустить рендеринг какого-либо кадра и компенсировать отставание, игровой процесс просто замедлится.

- **Переменная длительность временного шага:**

  Я упоминаю здесь этот вариант, но хочу сказать, что все разработчики кого я знаю настоятельно рекомендуют его не использовать. Тем не менее, я считаю, что будет полезно сказать почему это плохо.

  - *Адаптируется как к слишком медленной игре, так и к слишком быстрой*. Если игра не успевает работать в реальном времени, шаги будут становиться все дольше и дольше.
  - *Геймплей становится недетерминированным и нестабильным*. И это уже реальная проблема. Физика и сетевой код при переменой длительности временного шага становятся гораздо сложнее для реализации.

- **Фиксированный временной шаг обновления и непостоянный рендеринг:**

Последнему решению мы посвятили больше всего примеров кода, как самому применимому. При этом игра обновляется фиксированными временными шагами, но может пропускать рендеринг кадров, если это нужно для компенсации отставания.

- *Адаптируется как к слишком медленной игре, так и к слишком быстрой.* Если игра успевает обновляться в реальном времени — отставания не будет. Так что чем более мощный у игрока компьютер, тем плавнее будет работать игра.
- *Этот метод самый сложный.* Основной недостаток его в том, что слишком много чего нужно предусмотреть в реализации. Нужно настроить длительность временного шага таким образом, чтобы он был достаточно коротким для быстрых машин, но и не слишком замедлял слабые машины.

Смотрите также

- Очень рекомендую классическую статью Гленна Фидлера на эту же тему "Fix Your Timestep". Без него эта глава не состоялась бы.
- Немногим хуже статья Виттера про игровой цикл.
- Внутри Unity есть очень сложный игровой цикл, работа которого отлично проиллюстрирована здесь. Оригинальная ссылка не открывается.

Метод обновления (Update Method)

Задача

Симуляция коллекции независимых объектов с помощью указания каждому объекту обработки одного кадра поведения за раз.

Мотивация

Могучая валькирия игрока выполняет квест по краже прекрасных украшений с трупа давно умершего короля-волшебника. Она приближается ко входу величественной усыпальницы и ее атакует... *ничего*. Никаких проклятых статуй, стреляющих молниями. Никаких воинов нежити, патрулирующих вход. Она просто заходит. Забирает лут. Игра окончена. Вы выиграли.

Ну нет. Так не пойдет.

Гробнице нужны стражи — противники, с которыми сможет побороться наша героиня. Для начала нам понадобятся ожившие скелеты воины, патрулирующие вход. Если вы проигнорируете все, что знаете об игровом программировании, простейший код, перемещающий скелетов туда и сюда будет выглядеть так:

> Если королю-волшебнику нужно более интеллектуальное поведение, у него должно остаться хоть что-то от мозгов.

```
while (true) {
  // Патрулируем вправо.
  for (double x = 0; x < 100; x++) {
    skeleton.setX(x);
  }

  // Патрулируем влево.
  for (double x = 100; x > 0; x--) {
    skeleton.setX(x);
  }
}
```

Проблема здесь в том, что хотя скелеты и двигаются туда-сюда, но игрок их не видит. Программа зациклена в бесконечном цикле и никакого игрового процесса тут нет. Чего мы на самом деле хотим добиться — так это того, чтобы скелеты двигались *на*

каждом кадре.

Уберем эти циклы и переложим работу на уже существующий цикл. Это позволит игре реагировать на пользовательский ввод и рендерить врагов во время их перемещения. Вот так:

> Игровой цикл (Game Loop) — это еще один шаблон, описанный в книге.

```
Entity skeleton;
bool patrollingLeft = false;
double x = 0;

// Главный игровой цикл:
while (true) {
  if (patrollingLeft) {
    x--;
    if (x == 0) patrollingLeft = false;
  } else {
    x++;
    if (x == 100) patrollingLeft = true;
  }

  skeleton.setX(x);

  // Обработка игрового ввода и рендеринг игры...
}
```

Я привожу здесь код до и после, чтобы показать вам насколько код усложнился. Патрулирование влево и вправо может быть простым циклом `for`. Они даже косвенно следят за направлением скелетов в зависимости от запущенного цикла. Но теперь нам придется на каждом кадре прерывать выполнение и переходить в игровой цикл, а потом продолжать с того места, на котором мы остановились. А чтобы определять направление движения, нам придется использовать дополнительную переменную `patrollingLeft`.

Этот вариант уже более-менее рабочий. Безмозглый мешок с костями не составит вашей Северной воительнице серьезную конкуренцию, так что следующее, что мы добавим — это зачарованные статуи. Они будут стрелять молниями так часто, что героине придется прокрадываться мимо них на цыпочках.

Продолжая в нашем "максимально простом стиле", получим вот что:

```
// Переменные скелетов...
Entity leftStatue;
Entity rightStatue;
int leftStatueFrames = 0;
int rightStatueFrames = 0;

// Основной игровой цикл:
while (true) {
  // Код скелетов...
  if (++leftStatueFrames == 90) {
    leftStatueFrames = 0;
    leftStatue.shootLightning();
  }

  if (++rightStatueFrames == 80) {
    rightStatueFrames = 0;
    rightStatue.shootLightning();
  }

  // Обработка игрового ввода и рендеринг игры...
}
```

Не могу сказать что мы движемся в сторону кода, который легко и приятно поддерживать. У нас появилась куча новых переменных, а внутри игрового цикла образовалось месиво из кода, каждая часть которого обрабатывает отдельную сущность в игре. Чтобы они все работали одновременно, у нас получилось месиво относящегося к ним кода.

> Каждый раз когда ваш код можно описать словом "месиво" — у вас явно есть проблема.

Шаблон, который мы будем использовать для решения этой проблемы настолько прост, что вы наверное уже и сами до него додумались: *каждая сущность в игре инкапсулирует собственное поведение*. Таким образом наш игровой цикл становится лаконичным и мы получаем возможность обрабатывать любое количество сущностей.

Чтобы это сделать, нам нужен уровень абстракции и мы создаем его, определяя абстрактный метод `update()`. Игровой цикл поддерживает коллекцию объектов, но не знает их конкретный тип. Все что о них нужно знать — это то, что их нужно обновлять. Таким образом мы отделяем поведение каждого объекта от игрового цикла и других объектов.

На каждом кадре игровой цикл проходит по всей коллекции и вызывает `update()` для каждого объекта. Таким образом каждый объект может обновить свое поведение на один кадр. Вызывая его для объектов на каждом кадре, мы получаем их одновременное действие.

> Так как мне обязательно кто-то это припомнит, сознаюсь сразу — да, они не работают в полностью конкурентном режиме. Пока один из объектов обновляется, все остальные этого не делают. Мы еще вернемся к этому позже.

Игровой цикл содержит динамическую коллекцию объектов, так что добавлять и удалять объекты с уровня довольно просто — просто добавляем или удаляем их из коллекции. Больше никакого хардкодинга. Более того, уровень можно наполнить объектами из внешнего файла с данными, т.е. получаем как раз то что нужно геймдизайнерам.

Шаблон

Игровой мир содержит **коллекцию объектов**. Каждый объект реализует **метод обновления**, **симулирующий один кадр** поведения объекта. На каждом кадре игра обновляет каждый объект из коллекции.

Когда использовать

Если шаблон Игровой цикл (Game Loop) можно сравнить с хлебом, то шаблон *Метод обновления* можно назвать маслом. В той или иной форме этот шаблон использует огромное число игр, в которых есть живые сущности. Если в игре есть космодесантники, драконы, марсиане, призраки или атлеты, скорее всего она использует этот шаблон.

Однако, если игра более абстрактная и движущиеся части в ней не намного живее, чем шахматные фигуры, этот шаблон может и не понадобиться. В играх наподобие шахмат вам нет необходимости моделировать поведение всех игровых объектов одновременно и, возможно, вам не придется просить пешку обновить себя на каждом кадре.

> Вам не нужно обновлять их *поведение* на каждом кадре, но даже в настольной игре, вам возможно придется обновлять на каждом кадре *анимацию*. В этом вам шаблон тоже может помочь.

Метод обновления хорошо работает когда:

- В вашей игре есть некоторое количество объектов или систем, которые должны работать одновременно.
- Поведение каждого объекта практически не зависит от остальных.
- Объекты нужно обновлять постоянно.

Имейте в виду

Этот шаблон достаточно прост, чтобы скрывать в себе какие-то неприятные сюрпризы. Тем не мене каждая строка кода имеет последствия.

Разделение кода на срезы отдельных кадров делает его сложнее

Если вы сравните два первые примера кода, вы увидите, что второй уже значительно сложнее. Оба они заставляют скелет просто ходить туда-сюда, но второй делает это с перерывами на передачу управления коду игрового цикла на каждом кадре.

Такое изменение практически всегда необходимо для обработки пользовательского ввода, рендеринга и других вещей, о которых приходится заботиться игре, так что первый пример навряд ли можно назвать типичным. Зато он наглядно демонстрирует насколько усложняется поведенческий код, когда его приходится преобразовывать подобным образом.

> Я говорю "практически", потому что иногда и такое возможно. У вас вполне может быть прямолинейный код, который никогда не отвлекается на поведение объектов и в то же время у вас может быть множество одновременно обновляющихся объектов, работающих в конкурентном режиме и управляемых игровым циклом.
>
> Все что вам нужно — это система, поддерживающая множество "потоков" выполнения, работающих одновременно. Если код объекта можно просто остановить во время выполнения и потом продолжить, вместо того чтобы полностью *выходить* из него, такой код можно писать в более императивной манере.
>
> Настоящие потоки обычно слишком тяжеловесны чтобы хорошо работать, но если ваш язык поддерживает легковесные конкурентные конструкции наподобие генераторов (generators), сопрограмм (coroutines) или нитей (fibers), вы можете использовать их.
>
> Еще одним способом создания потоков выполнения на уровне приложения является использование шаблона Байткод(Bytecode).

Вам нужно сохранять состояние чтобы продолжать с того места где вы прервались

В первом примере кода у нас не было никаких переменных, определяющих налево идет стражник или направо. Это явно следовало из выполняющегося кода.

Когда мы поменяли этот код на код вида кадр за раз, нам пришлось добавить переменную `patrollingLeft`, чтобы за этим следить. Когда мы возвращаемся к нашему коду, предыдущая позиция теряется и нам нужно хранить достаточно информации чтобы восстановить состояние перед следующими кадром.

Здесь нам может помочь шаблон Состояние (State). Машины состояний (state machines) и их разновидности так часто встречаются в играх отчасти потому, что (что следует из их имени) они хранят состояние, которым вы можете воспользоваться после того, как отвлечетесь на что-то.

Объекты симулируются на каждом кадре, но не в настоящем конкурентном режиме

В этом шаблоне игровой цикл перебирает всю коллекцию объектов и обновляет каждый из них. Внутри вызова `update()`, большинство объектов могут получить доступ ко всему остальному игровому миру, включая другие объекты, которые обновляются. Это значит, что порядок, в котором объекты обновляются, начинает иметь значение.

Если обновление A происходит перед B в списке обновления, тогда во время обновления A, оно видит предыдущее состояние B. Но когда обновляется B, оно уже видит A в *новом* состоянии, потому что A на этом кадре уже обновлялось. Даже если с точки зрения игрока все происходит одновременно, ядро игры все равно работает в пошаговом режиме. Просто один полный "шаг" соответствует по длительности одному кадру.

> Если по какой либо причине, вы решили что ваша игра не должна работать в таком последовательном режиме, вам может помочь нечто наподобие шаблона Двойная буферизация (Double Buffer). В таком случае порядок обновления A и B перестанет играть какую-либо роль, потому что *и тот и другой* объект будут видеть предыдущее состояние другого.

Пока логика игры не предполагает тесного связывания это нормально. Обновление объектов в параллельном режиме таит в себе некоторые неприятные сюрпризы. Представьте себе шахматы, где белые и черные ходят одновременно. Обе стороны пытаются переставить фигуру в пустую клетку. Как же можно решить эту проблему?

Последовательное обновление эту проблему решает: каждое обновление постепенно изменяет мир из одного состояния в другое без промежуточного времени, когда вещи находятся в двойственном состоянии и требуют уточнения.

> Еще это помогает при разработке сетевых игр, потому что у вас есть сериализуемый набор шагов, которые можно передавать по сети.

Будьте осторожны при изменении списка объектов во время обновления

Когда вы используете этот шаблон, на метод обновления обязательно завязывается слишком большая часть поведения игры. Обычно сюда относится и код, добавляющий и удаляющий обновляемые объекты в игре.

Например, наш скелет охранник будет дропать предмет после смерти. Новый объект вы можете без проблем просто добавить в конце списка обновляемых объектов. Вы идете по списку объектов дальше, доходите до только что добавленного нового объекта и обновляете и его тоже.

Только это совсем не значит, что новый объект имеет право действовать в том же кадре, когда его добавили, до того как игрок смог его хотя бы увидеть. Если вы не хотите чтобы это произошло, можно применить одну хитрость и кешировать в начале цикла обновления количество объектов в списке и обновлять только такое их количество:

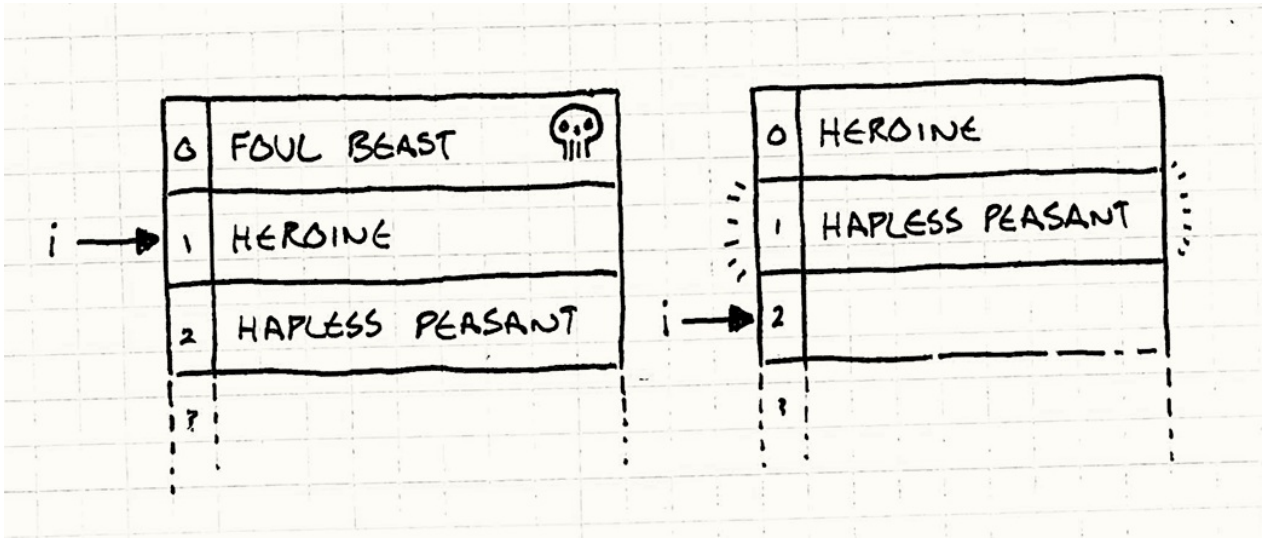
```
int numObjectsThisTurn = numObjects_;
for (int i = 0; i < numObjectsThisTurn; i++) {
  objects_[i]->update();
}
```

Здесь `objects_` — это массив обновляемых объектов в игре, `a numObjects_` — его длина. Когда добавляется новый объект, длина увеличивается на единицу. Мы кэшируем длину в `numObjectsThisTurn` в начале цикла, так что итерация останавливается перед тем, как мы доберемся до объектов, добавленных на текущем кадре.

Проблема усложняется, если мы выполняем *удаление* во время итерации. Вы убиваете глупого монстра и теперь его нужно выкинуть из списка объектов. Если он находится в списке до объекта, который вы сейчас обновляете, вы можете случайно пропустить объект:

```
for (int i = 0; i < numObjects_; i++) {
  objects_[i]->update();
}
```

Этот простейший цикл инкрементирует индекс обновляемого объекта на каждой итерации. В левой части иллюстрации ниже показано как массив выглядит пока мы обновляем героиню.



Так как мы обновляем героиню, `i` равняется 1. Она убивает монстра и он удаляется из массива. Героиня перемещается на позицию 0 и несчастный крестьянин перемещается на позицию 1. После обновления героини, `i` инкрементируется до 2. Как вы можете видеть справа, несчастного крестьянина пропустили и он никогда не будет обновлен.

> Простейшее решение — это двигаться по списку объектов *в обратную сторону*. Таким образом удаление объекта сместит только уже обновленные объекты.

Можно просто быть более аккуратным при удалении объектов и обновлять любые итерационные переменные, учитывая удаление. Или же можно подождать с удалением до тех пор, пока мы не обойдем весь список. Пометьте объект как "мертвый", но сразу не удаляйте. Во время обновления мы пропускаем мертвые объекты. А когда закончили — проходим список еще раз и удаляем все мертвые объекты.

> Если вы обновляете объекты в цикле обновления в многопоточном режиме, вам тем более стоит повременить с любыми изменениями списка объектов, чтобы избежать расходов на синхронизацию во время обновления.

Пример кода

Шаблон настолько прямолинеен, что пример кода просто топчется на месте. Это совсем не значит, что шаблон бесполезен. Он очень *полезен*, потому что он *простой*: это просто решение проблемы без всяких украшений.

Но чтобы говорить более конкретно, давайте пройдемся по нескольким реализациям. Начнем с класса сущности, представляющей скелет или статую.

```
class Entity
{
public:
  Entity() : x_(0), y_(0)
  {}

  virtual ~Entity() {}
  virtual void update() = 0;

  double x() const { return x_; }
  double y() const { return y_; }

  void setX(double x) { x_ = x; }
  void setY(double y) { y_ = y; }

private:
  double x_;
  double y_;
};
```

Я добавил в нее несколько вещей, но это самый минимум того, что понадобится нам в дальнейшем. Смею предположить что в реальном коде будет гораздо больше всяких связанных с графикой или физикой штук. Самое важное для нашего шаблона заключается в том, что у сущности есть абстрактный метод `update()`.

Игра поддерживает коллекцию таких сущностей. В нашем примере мы поместим ее в класс, представляющий игровой мир:

```
class World
{
public:
  World() : numEntities_(0)
  {}

  void gameLoop();

private:
  Entity* entities_[MAX_ENTITIES];
  int numEntities_;
};
```

> В настоящей программе, вы скорее всего будете использовать специальный класс-коллекцию, но для упрощения я использую в своем примере обычный массив.

Теперь, когда все готово, игра реализует шаблон, обновляя на каждом кадре все сущности:

```
void World::gameLoop()
{
  while (true) {
    // Обработка пользовательского ввода...

    // Обновление каждой из сущностей.
    for (int i = 0; i < numEntities_; i++) {
      entities_[i]->update();
    }

    // Физика и рендеринг...
  }
}
```

> Как следует из названия метода — это пример применения шаблона Игровой цикл (Game Loop).

Сущности подклассы?

Уверен, что у некоторых читателей уже пошли мурашки по коже, когда они увидели, как я наследую от главного класса сущности для определения специального поведения. Если вы не видите в этом проблемы, я введу вас в курс дела.

Когда игровая индустрия вышла из первобытного супа ассемблерного кода 6502 и синхронизации с обратным ходом луча в мир объектно-ориентированных языков, разработчики попали в безумный мир архитектуры программного обеспечения. Одной из любимых возможностей было наследование. И была воздвигнута византийская башня наследования, такая высокая, что заслоняла собой солнце.

Это была ужасная идея и никто не мог поддерживать гигантскую гору иерархии классов без того, чтобы быть под ней погребенным. Даже банда четырех уже предупреждала об этом в еще 1994-м году:

Предпочитайте "композицию объектов" "наследованию классов".

> Только между нами, мне кажется маятник слишком далеко качнулся от подклассов. Обычно я их избегаю, но не стоит возводить неиспользование наследования в догму, также как не стоит превращать в догму его использование. Используйте эту возможность осмотрительно и трезво оценивая последствия.

Когда это понимание проникло в игровую индустрию, решение виделось в шаблоне Компонент (Component). Если мы его применим, наш `update()` станет *компонентом* сущности, а не самой `Entity`. Таким образом, нам не нужно будет организовывать сложную иерархию сущностей, чтобы определять и повторно использовать поведение. Вместо этого мы можем смешивать и сопоставлять компоненты.

> Вот такие.

Если бы мы писали настоящую игру, я бы тоже так поступил. Но эта глава не про компонент. Она о методе `update()` и проще всего увидеть как он работает с минимальным количеством дополнительных деталей — это поместить метод непосредственно в ```Entity и унаследовать от нее несколько подклассов.

Определение сущностей

Хорошо, вернемся к нашей задаче. Изначально мы хотели реализовать патрулирование скелета-охранника и работу стреляющих молниями статуй. Начнем с нашего костлявого друга. Чтобы определить его патрульное поведение, создаем новую сущность с соответствующей реализацией `update()`:

```
class Skeleton : public Entity
{
public:
  Skeleton() : patrollingLeft_(false)
  {}

  virtual void update() {
    if (patrollingLeft_) {
      setX(x() – 1);
      if (x() == 0) patrollingLeft_ = false;
    } else {
      setX(x() + 1);
      if (x() == 100) patrollingLeft_ = true;
    }
  }

private:
  bool patrollingLeft_;
};
```

Как вы видите, мы практически выдрали этот кусок кода из игрового цикла, который мы писали раньше и поместили его в метод `update()` класса `Skeleton`. Единственное серьезное отличие только в том, что `patrollingLeft_` превратилось в поле из обычной локальной переменной. Таким образом, значение будет сохраняться между вызовами `update()`.

Давайте теперь перейдем к статуям:

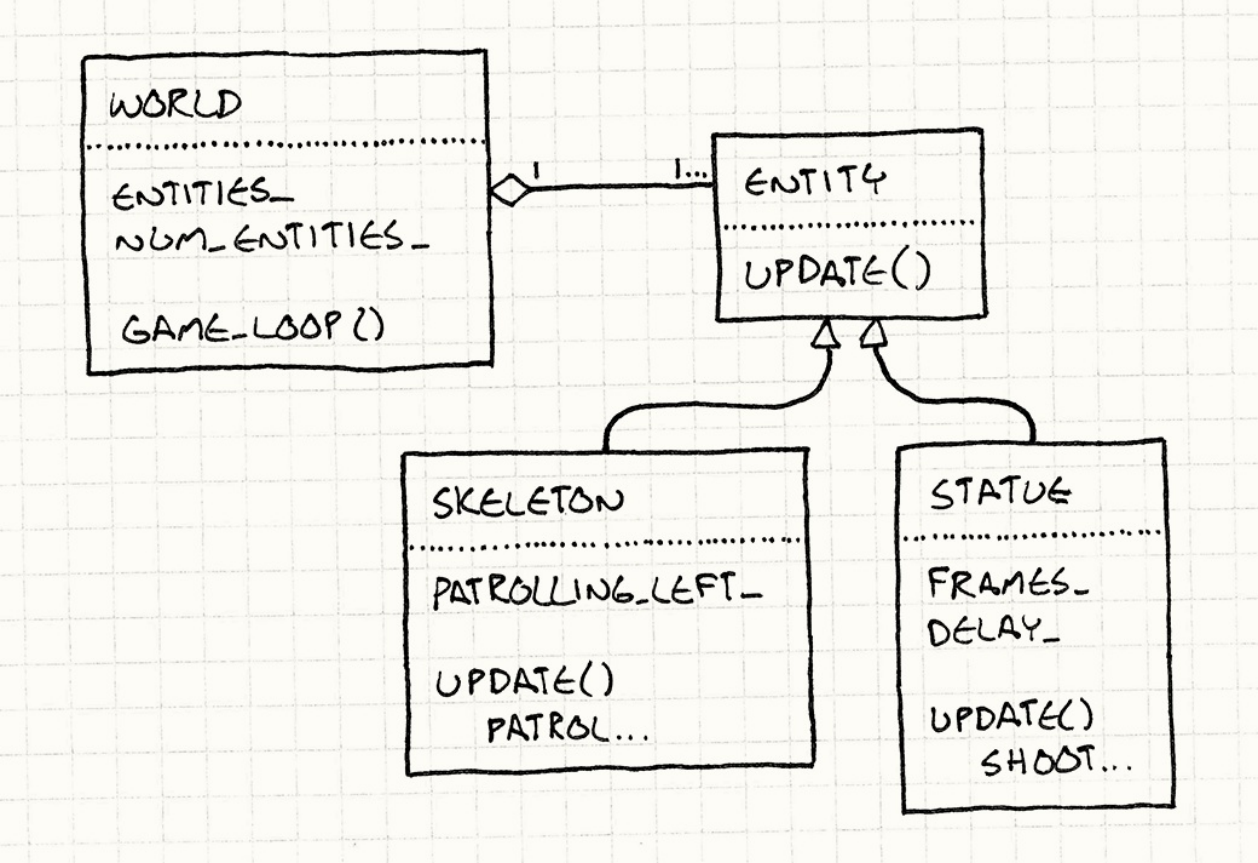
```
class Statue : public Entity
{
public:
  Statue(int delay) : frames_(0), delay_(delay)
  {}

  virtual void update() {
    if (++frames_ == delay_) {
      shootLightning();

      // Сброс таймера.
      frames_ = 0;
    }
  }

private:
  int frames_;
  int delay_;

  void shootLightning() {
    // Стрельба молнией...
  }
};
```

И опять мы практически просто перенесли сюда старый код из игрового цикла и кое-что переименовали. Таким образом, мы просто упрощаем кодовую базу. В оригинальном скученном императивном коде у нас были отдельные локальные переменные и для счетчика кадров статуи и для частоты стрельбы.

Теперь, когда мы перенесли их в сам класс `Statue`, вы можете создать столько переменных сколько нужно и у каждого экземпляра будет собственный таймер. Вот она истинная цель шаблона: теперь нам гораздо проще добавлять новые сущности в игровой мир, потому что каждая из них располагает всем необходимым, чтобы о себе позаботиться.

Шаблон позволяет нам *населять* мир, а не *реализовывать* его. А еще мы получаем дополнительную гибкость за счет наполнения мира через отдельный файл данных или редактор уровня.

> Людям еще интересен `UML`? Если да, то вот что я нарисовал.



Ход времени

Это была суть шаблона, но я хочу показать вам еще несколько вариаций. Пока что мы предполагали что каждый вызов `update()` продвигает состояние игрового мира на фиксированный отрезок времени.

Мне такой подход нравится, однако многие игры используют *переменный временной шаг*. В этом случае игровой цикл симулирует более длинные или более короткие отрезки времени, в зависимости от того, сколько потребовалось времени на обработку и рендеринг предыдущего кадра.

> В главе Игровой цикл (Game Loop) как раз описаны достоинства и недостатки фиксированных и переменных временных шагов.

Это значит что каждому вызову `update()` необходимо знать насколько продвинулись виртуальные часы. Поэтому внутрь передается количество прошедшего времени. Например наш скелет патрульный может обрабатывать переменный временной шаг следующим образом:

```
void Skeleton::update(double elapsed)
{
  if (patrollingLeft_) {
    x -= elapsed;
    if (x <= 0) {
      patrollingLeft_ = false;
      x = -x;
    }
  } else {
    x += elapsed;
    if (x >= 100) {
      patrollingLeft_ = true;
      x = 100 — (x — 100);
    }
  }
}
```

Теперь расстояние, проходимое скелетом увеличивается вместе с увеличением обрабатываемого времени. Как вы видите работа с переменным временным отрезком увеличивает сложность кода. Скелет должен следить за тем чтобы не выйти за рамки зоны патрулирования во время долгого временного отрезка и нам нужно корректно обрабатывать этот случай.

Архитектурные решения

В таком простом шаблоне не слишком много разнообразия, но все таки есть некоторые настройки.

В каком классе будет жить метод обновления?

Самое главное решение, которое вам нужно принять — это внутри какого класса разместить `update()`.

- **Класс сущности:**

  Это простейшее решение, если у вас уже есть класс сущности, потому что вам не нужно будет вводить в игру никаких дополнительных классов. Это может сработать, если у вас не слишком много видов сущностей, но в целом индустрия уходит от такого подхода.

  Создавать подкласс `Entity` каждый раз когда вам нужно новое поведение — довольно хрупкий и болезненный метод, если видов сущностей у вас достаточно много. Вскоре вы обнаружите, что вам хочется повторно использовать куски кода, не слишком хорошо укладывающиеся в стройную иерархию и застрянете на этом.

- **Класс компонент:**

  Если вы уже используете шаблон Компонент (Component), то тут и думать нечего. Он позволяет каждому компоненту обновляться независимо. Подобно тому как работает метод обновления для снижения связности в игре в целом, этот шаблон позволяет вам снизить связность частей отдельной сущности друг от друга. Рендеринг, физика и AI могут позаботиться о себе самостоятельно.

- **Класс делегат:**

  Есть еще один шаблон, который предполагает делегирование части поведения класса другому объекту. Шаблон Состояние (State) именно этим и занимается, так что вы можете менять поведение объекта, подменяя объект к которому он делегирует. Шаблон Объект тип (Type Object) делает это, позволяя вам разделять поведение между несколькими сущностями одного "вида".

  Если вы используете один из этих шаблонов, поместить `update()` в класс, к которому мы делегируем, будет вполне логичным решением. В этом случае у вас все еще может быть метод `update()` в главном классе, но он уже будет не виртуальным, а просто будет вызывать объект делегат. Примерно так:

  ```
  void Entity::update()
  {
    // Forward to state object.
    state_->update();
  }
  ```

  Это позволяет вам определять новое поведение, изменяя объект делегат. Как при использовании компонентов, он дает вам гибкость в изменении поведения без необходимости определять полностью новый подкласс.

Как обрабатываются спящие объекты?

Очень часто у вас в игровом мире присутствуют объекты, которые по какой-либо причине не нужно обновлять. Они могут быть отключенными, находящимися за пределами экрана или еще не разблокированными. Если у вас в этом состоянии находится довольно большое количество объектов, может быть довольно расточительно в плане процессорного времени проходить их все во время обновления каждого кадра.

Одним из решений является создание отдельной коллекции объектов только с "живыми" объектами, которые требуют обновления. Когда объект становится неактивным, он удаляется из этой коллекции. Когда включается снова — добавляется

обратно. Таким образом мы будем проходить только по списку активных объектов.

- **Если вы используете единственную коллекцию интерактивных объектов:**
  - *Вы впустую тратите время.* Для интерактивных объектов вам придется добавлять проверку флага в духе "активен ли я" или вызывать пустой метод.

    > В дополнение к потерянным процессорным тактам на проверку активности объекта, бесполезный перебор объектов приводит к ошибкам обращения к кешу. Процессор спроектирован таким образом, что быстрее читает данные из кеша, а не из памяти. Лучше всего это получается, когда следующий читаемый объект находится сразу за предыдущим.
    >
    > Когда вы переходите к следующему объекту, вы можете пропустить оставшуюся в кеше порцию данных, заставляя процессор загружать следующую порцию данных из обычной памяти.

- **Если вы используете отдельную коллекцию с активными объектами:**
  - *Вы используете дополнительную память для хранения второй коллекции.* В случае если у вас активны все объекты — это будет просто вторая копия первой коллекции. В таком случае она явно избыточна. Но когда скорость важнее памяти (а обычно так и есть) это допустимая потеря.

    Еще одно решение — это перейти к двум коллекциям, но теперь во второй будут храниться только неактивные объекты.
  - *Вам нужно поддерживать синхронизацию коллекций.* Когда объект создается или полностью уничтожается (или временно отключается), вам нужно удалить или изменить его как в основной коллекции, так и в коллекции активных объектов.

Выбрать, что вам больше подходит, можно проанализировав сколько неактивных объектов может у вас быть. Чем их больше, тем полезнее использовать для них отдельную коллекцию, чтобы не отделять их прямо внутри игрового цикла.

Смотрите также

- Этот шаблон вместе с Игровым циклом (Game Loop) и Компонентом (Component) является частью троицы, обычно образующей сердце современного игрового движка.

- Когда вы начинаете задумываться о производительности обновления кучи сущностей или компонентов на каждом цикле, вам может прийти на помощь шаблон Локализация данных (Data Locality).
- Фреймворк Unity использует этот шаблон в нескольких классах, включая `MonoBehaviour`.
- Платформа Microsoft XNA использует этот шаблон в классах `Game` и `GameComponent`.
- Игровой движок на JavaScript Quintus использует этот шаблон в главном классе `Sprite`.

Поведенческие шаблоны

Как только вы сформируете основу игры и украсите ее актерами и декорациями, все что вам остается — это запустить сцену. Для этого нам нужно поведение — сценарий, из которого игровые сущности будут знать что им делать.

Конечно, весь код — это уже "поведение" и вообще, все программы определяют именно поведение, но отличие игр от других программ как раз и заключается в том насколько *широкий* диапазон поведения вам нужно реализовать. И хотя у текстового процессора конечно тоже много функций, это количество просто меркнет перед тем, сколько обитателей, предметов и квестов в средней ролевой игре.

Шаблоны в этом разделе помогут вам определить и улучшить широкий диапазон поведения быстрым и простым для поддержки способом. Объект тип (Type Object) создает категории поведения без необходимости создавать для этого отдельные классы. Подкласс песочница (Subclass Sandbox) предоставляет вам набор примитивов, с помощью которых можно составлять различное поведение. Самый продвинутый метод — это Байткод (Bytecode), который выносит поведение из сущности и помещает его в данные.

Шаблоны

- Байткод (Bytecode)
- Подкласс песочница (Subclass Sandbox)
- Объект тип (Type Object)

Байткод (Bytecode)

Задача

Обеспечить поведению гибкость данных, декодируемых в виде инструкций для виртуальной машины.

Мотивация

Создавать игры бывает весело, но не очень то и легко. Современные игры требуют гигантской сложнейшей кодовой базы. Производители консолей и владельцы магазинов приложений постоянно ужесточают требования к качеству и даже единственный баг может не дать вашей игре выйти на рынок.

> Я работал над играми, в которых было шесть миллионов строк кода на `C++`. Для сравнения код, управляющий марсоходом Mars Curiosity примерно в половину меньше.

В то же время мы хотим выжать из существующего железа всю производительность до последней капли. Игры нагружают железо как никакие другие программы, и нам нужно постоянно заниматься оптимизацией, чтобы просто угнаться за конкурентами.

Чтобы соответствовать этим высоким требованиям стабильности и производительности, мы используем тяжеловесные языки типа `C++`, в которых есть достаточная низкоуровневая выразительность чтобы напрямую работать с железом и богатая система типизации, предотвращающая или хотя бы ограничивающая появление багов.

Мы гордимся своими навыками работы с ними, но у всего есть своя цена. Чтобы стать профессиональным программистом нужно потратить немало лет на учение, после которых вы сможете справиться со сложностью собственной кодовой базы. Время сборки больших игр может варьироваться от "можно пойти попить кофе" до "пожарьте кофе бобы, размелите их вручную, сварите эспрессо, приготовьте молоко и потренируйтесь в приготовлении латте."

Помимо всех этих сложностей в играх обязательно должен присутствовать еще один компонент: *веселье*. Игроки требуют игровой процесс, который будет и новаторским и отлично сбалансированным. Для этого нужно множество итераций, но даже

небольшая настройка требует от программиста залазить в низкоуровневый код и потом ожидать перекомпиляцию, что в результате во многом убивает творческий процесс.

Война заклинаний!

Скажем, мы работаем над магическим файтингом. Двое волшебников борются между собой с помощью заклинаний, пока не останется только победитель. Мы можем определить эти заклинания в коде, но это будет означать что каждый раз когда нужно будет настроить их параметры, придется беспокоить программиста. Если геймдизайнер хочет немного изменить числа чтобы попробовать что получится, для этого придется перекомпилировать всю игру, перезагружать ее и повторять бой заново.

Как часто бывает с современными играми, нам нужно иметь возможность обновлять игру после продажи чтобы исправлять баги и добавлять новый контент. Если заклинание будет жестко закодированным, то для обновления нужно будет патчить сам исполнимый файл игры.

Давайте пойдем дальше и представим себе что мы хотим поддерживать *моды*. Мы хотим дать игрокам возможность создавать собственные заклинания. Если они будут находиться в коде, это значит что каждому моддеру нужно иметь всю цепочку инструментов для сборки игры и нам нужно выкладывать свои исходники. Еще хуже то, что если в заклинании будет баг, он будет способен обрушить всю игру целиком на машине игрока.

Данные > Код

Понятно, что наша реализация на языке программирования нам подходит не очень. Нам нужно выделить для заклинаний отдельную от игры песочницу. Нам нужно, чтобы они легко редактировались и перезагружались, а также были физически отделены от остальной части исполнимого файла.

Не знаю как вам, а мне это напоминает *данные*. Если мы сможем определить наше поведение в отдельном файле данных, которые движок игры будет каким-то образом загружать и "выполнять" — это и будет то что нам нужно.

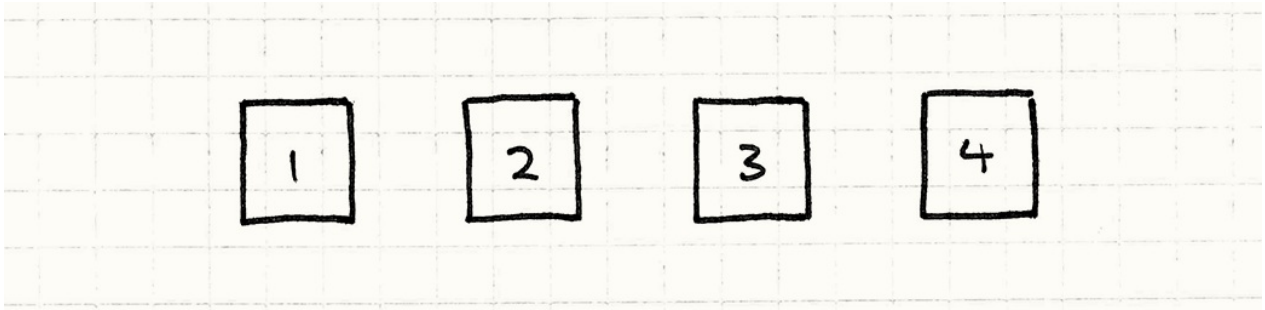
Нужно определиться с тем что значит "выполнить" данные. Как несколько байтов из файла можно превратить в поведение. Есть несколько способов. Я думаю нам легче будет оценить сильные и слабые стороны шаблона, если мы сравним его с другим шаблоном: Интерпретатор (Interpreter)[GoF].

Шаблон Интерпретатор

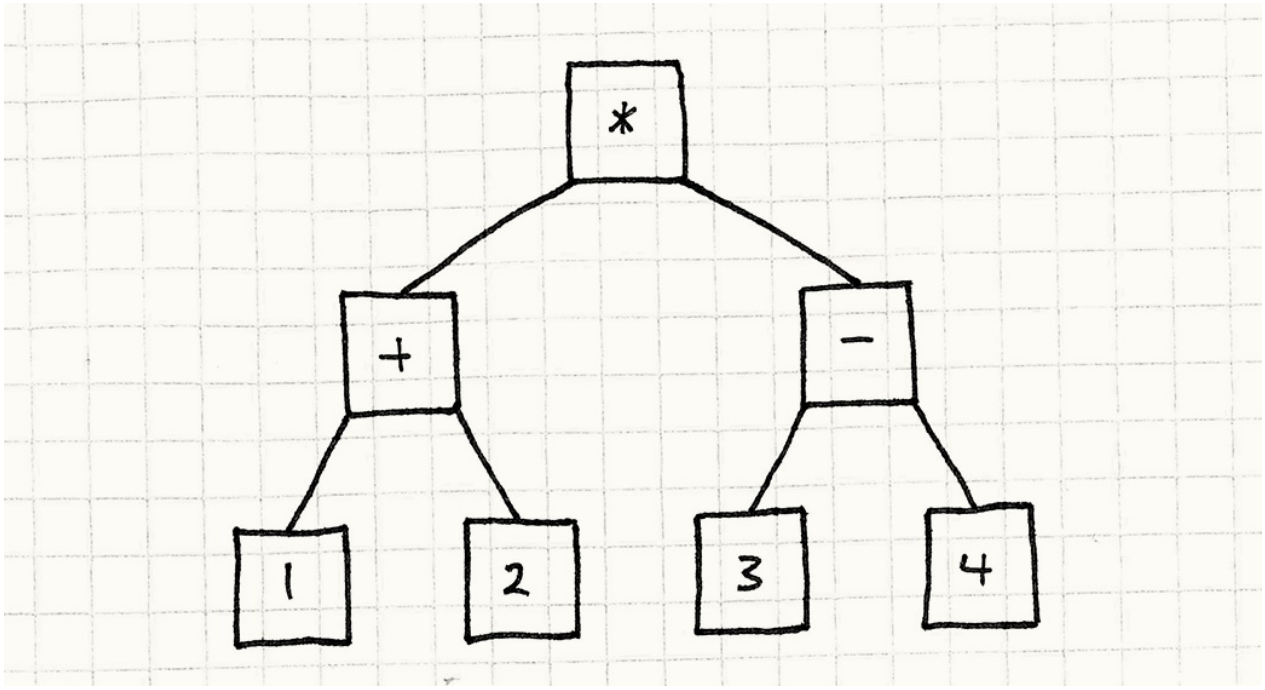
Я мог бы написать об этом шаблоне целую главу, но до меня это сделали четыре известные вам мужика. Вместо этого я ограничусь совсем кратким его описанием. Он начинается с языка — я имею в виду языка программирования — который вы хотите выполнять. Скажем он будет поддерживать математические выражения следующего вида:

```
(1 + 2) * (3 – 4)
```

Мы берем каждый из фрагментов этого выражения и превращаем его в соответствии с правилами языка в *объект*. Числовые литералы будут объектами:



По идее, это просто небольшая обертка над сырыми значениями. Операторы тоже будут объектами и у них будут ссылки на свои операнды. Если учитывать родителей и наследников, это выражение превращается в небольшое дерево объектов:



> Что это за "магия"? Очень просто: *парсинг*. Парсер получает строку символов и превращает ее в *абстрактное синтаксическое дерево*, т.е. коллекцию объектов, представляющих грамматическую структуру текста.
>
> Как только у нас будет парсер — считай пол компилятора готово.

Шаблон интерпретатор не о том как *создавать* это дерево. Он о том как его *выполнять*. Действует он весьма разумно. Каждый объект в дереве — это выражение или подвыражение. В полностью объектно-ориентированной манере мы позволяем выражениям вычислять самих себя.

Для начала определим базовый интерфейс, реализуемый всеми выражениями.

```
class Expression
{
public:
  virtual ~Expression() {}
  virtual double evaluate() = 0;
};
```

Далее определяем класс, реализующий это для всех типов выражений в словаре нашего языка. Самые простые из них — это числа.

```
class NumberExpression : public Expression
{
public:
  NumberExpression(double value)
  : value_(value)
  {}

  virtual double evaluate()
  {
    return value_;
  }

private:
  double value_;
};
```

Выражение числовой литерал вычисляет свое значение. Сложение и умножение немного сложнее, потому что содержат подвыражения. До того как они смогут вычислить свое значение, они должны рекурсивно вычислить все подвыражения. Примерно таким образом:

> Я уверен, что вы догадаетесь как выглядит реализация умножения.

```
class AdditionExpression : public Expression
{
public:
  AdditionExpression(Expression* left, Expression* right)
  : left_(left), right_(right)
  {}

  virtual double evaluate()
  {
    // Вычисляем операнды.
    double left = left_->evaluate();
    double right = right_->evaluate();

    // Складываем их.
    return left + right;
  }

private:
  Expression* left_;
  Expression* right_;
};
```

Довольно изящно, не правда ли? Всего несколько простых классов и теперь у нас есть возможность вычислять достаточно сложные арифметические выражения. Нам просто нужно будет создать правильные объекты и корректно их подключить.

Это отличный, простой шаблон, но имеющий значительные недостатки. Посмотрите на иллюстрацию. Что вы видите? Кучу прямоугольников и кучу стрелок между ними. Код представляется как развесистое фрактальное дерево мелких объектов. Это порождает некоторые неприятные последствия:

> `Ruby` был реализован именно таким образом около 15 лет назад. В версии 1.9 он перешел на байткод по типу того, что описан в этой главе. Смотрите сколько времени я вам сэкономил!

- Загрузка с диска требует создания экземпляров и связывание множества мелких объектов.
- Эти объекты и указатели между ними занимают кучу памяти. На 32-битной машине даже такое маленькое выражение занимает не меньше 68 байт, не считая выравнивания.

> Если вы играете дома, не забудьте принять во внимание виртуальную таблицу указателей (vtable pointers).

- Путешествие по указателям подвыражений убивает данные в кеше. Да и вообще все вызовы виртуальных методов сеют панику в кеше инструкций.

> См. главу Локализация данных (Data Locality), где подробнее описано что такое кеш и как он влияет на производительность.

Если соединить все это вместе, то что получится? S-L-O-W (медленно). Вот почему большинство языков программирования, которые мы используем не основаны на шаблоне *Интерпретатор*. Он просто слишком медленный и требует слишком много памяти.

Машинный код, виртуальный

Вспомним нашу игру. Когда мы ее запускаем, компьютер игрока не пытается построить в реальном времени словарь C++ кода. Вместо этого у нас есть машинный код, который выполняется на процессоре. В чем же особенность машинного кода?

- *Он компактен*. Это цельное, сплошное скопление бинарных данных, в котором зазря не тратится ни единый бит.
- *Он линеен*. Инструкции собраны вместе и выполняются одна за другой. Никаких прыжков по памяти (если вы конечно не управляете порядком выполнения).
- *Он низкоуровневый*. Каждая инструкция выполняет относительно минимальную вещь и более сложное поведение получается путем их комбинирования.
- *Он быстрый*. Следствием всего этого (ну и еще конечно потому, что это реализовано в железе) является то, что машинный код выполняется со скоростью ветра.

Звучит здорово, но ведь мы не собираемся писать настоящий машинный код для нашего заклинания. Позволять пользователю писать машинный код, который мы будем выполнять — это крайне небезопасно. Что нам нужно — так это некий компромисс между машинным кодом и безопасностью шаблона *Интерпретатор*.

Что если вместо загрузки настоящего машинного кода и прямого его выполнения, мы определим наш собственный *виртуальный* машинный код? Далее мы просто напишем небольшой эмулятор для нашей игры. Он будет похож на машинный код — компактный, линейный, сравнительно низкоуровневый, но будет интерпретироваться нашей игрой и значит его можно будет выполнять в режиме песочницы.

> Вот почему многие игровые консоли и iOS не позволяют программам выполнять машинный код, загружаемый или генерируемый во время выполнения программы. Это не очень хорошо, потому что самые быстрые реализации языков именно так и работают. Они содержат в себе компилятор на лету (just-in-time) или JIT, преобразующий язык в оптимизированный машинный код на лету.

Наш маленький эмулятор будет называться *виртуальной машиной* (или "VM" если коротко), а наш синтетический машинный код, который на ней выполняется будет называться *байткодом*. Он обладает гибкостью и простотой использования описания вещей с помощью данных, но обладает лучшей производительностью чем высокоуровневые реализации типа шаблона *Интерпретатор*.

> В программистских кругах "виртуальная машина" и "интерпретатор" — синонимы и я использую эти слова во взаимозаменяемом значении. Но когда мы говорим о шаблоне Интерпретатор от банды четырех, я специально пишу шаблон, чтобы было понятно о чем я говорю.

Звучит довольно сложно. Оставшуюся часть главы я посвящу тому, чтобы продемонстрировать, что если ограничиться только самыми главными функциями, это вполне посильная задача. Даже если вы сами не будете использовать этот шаблон, вы будете лучше понимать `Lua` и другие языки, основанные на этом шаблоне.

Шаблон

Набор инструкций определяет низкоуровневые операции, которые можно выполнить. Они кодируются в виде **последовательности байтов**. **Виртуальная машина** выполняет эти инструкции по одной за раз, используя **стек промежуточных значений**. Комбинируя инструкции можно определить сложное высокоуровневое поведение.

Как использовать

Это самый сложный шаблон из описанных в книге и его не так уж просто вставить в игру. Использовать его стоит только если вам нужно определить очень много разнообразного поведения и язык, на котором вы пишете игру не слишком для этого подходит потому что:

- Он слишком низкоуровневый и программировать на нем сложно или чревато ошибками.
- Итеративная работа с ним получается слишком медленной из-за долгой компиляции или других проблем со сборкой.
- Он слишком доверителен. Если вы хотите быть уверенными что определенное поведение не обрушит игру, вам нужно организовать для него песочницу, отдельную от остальной кодовой базы.

Конечно, все это подходит практически к любой игре. Кто же не хочет более быстрых циклов или большей безопасности? К сожалению, ничто не достается даром. Байткод медленнее настоящего, так что в критических для производительности частях движка игры его применять не стоит.

Имейте в виду

Есть что-то притягательное в идее создания собственного языка или системы в системе. Я приведу здесь минимальный пример, но в реальном мире такие вещи обычно разрастаются подобно виноградной лозе.

> Для меня игровая разработка сама по себе привлекательна. В любом случае я создаю виртуальное пространство для других людей, в котором они могут играть и заниматься созиданием.

Каждый раз, когда я вижу как кто-то реализует язык или систему скриптов, они говорят "Не волнуйся, она будет совсем маленькой". А потом они неизбежно начинают добавлять все новые и новые маленькие возможности, пока не получится полноценный язык. И в отличие от других языков, он растет прямо там, где появился как некий живой организм и в результате его архитектура получается подобной городским трущобам.

> Для примера можете посмотреть любой язык шаблонов.

Конечно, ничего *плохого* в создании полноценного языка нет. Но только если вы делаете это сознательно. Иначе не забывайте контролировать возможности своего байткода. Наденьте на него поводок, прежде чем он от вас не убежал.

Вам нужен интерфейс для байткода (front-end)

У низкоуровневых инструкций байткода впечатляющая производительность, но бинарный формат — это не то что хотели бы видеть ваши пользователи. Одна из причин, почему мы убираем описание поведения из кода — это потому что мы хотим его описывать на *более высоком* уровне. Если `C++` — слишком низкий уровень, позволяющий писать практически на уровне ассемблера — даже ваша собственная архитектура — это еще не улучшение!

Также, как в шаблоне *Интерпретатор* от банды четырех, мы подразумеваем, что у нас есть способ *генерации* байткода. Обычно пользователи описывают поведение на довольно высоком уровне и потом специальная утилита превращает результат в байткод, понятный виртуальной машине. Другими словами компилятор.

> Оспаривать это утверждение берется почтенная игра RoboWar. В этой игре игрок пишет маленькую программу для управления роботом на языке, очень похожем на ассемблер и с набором инструкций, подобным здесь описываемому.
>
> С ее помощью я впервые познакомился с ассемблеро-подобными языками.

Я знаю что звучит пугающе. Вот почему я об этом и упоминаю. Если у вас нет возможности писать дополнительный инструментарий — значит байткод не для вас. Но как вы скоро увидите, все не так уж и плохо.

Вам будет не хватать вашего отладчика

Программирование — это сложно. Мы знаем чего хотим от машины, но не всегда правильно сообщаем ей об этом — т.е. допускаем появление багов. Для того чтобы их устранять мы вооружаемся кучей инструментов, помогающих нам понять что в коде работает неправильно и как это исправить.

У нас есть отладчик, статический анализатор, декомпилятор и т.д. Все эти инструменты разработаны для каких-либо существующих языков: либо машинного кода, либо для чего-то более высокоуровневого.

Когда вы определяете собственную виртуальную машину для байткода, эти инструменты становятся для вас недоступными. Конечно вы можете отлаживать саму виртуальную машину. Но таким образом вы ведь видите только что делает *сама* виртуальная машина, а не байткод, который она интерпретирует. И конечно она не позволит вам увидеть что делается с байткодом в его исходной высокоуровневой форме, из которой он компилировался.

Если поведение, которое мы определяем довольно простое, вы вполне можете обойтись без всяких дополнительных инструментов. Но как только объем контента будет расти, думайте о том что вам придется потратить время на инструменты, помогающие пользователям видеть что их байткод делает. Они могут не поставляться в комплекте с игрой, но они могут быть критичными для того чтобы вы вообще смогли выпустить игру.

> Конечно, если вы хотите чтобы ваша игра была модифицируемой, тогда вам *придется* оставить в игре эти возможности и это только увеличивает их важность.

Пример кода

После предыдущих нескольких разделов вы наверное будете удивлены тем насколько прямолинейна реализация. Для начала нам нужно сформировать набор инструкций для виртуальной машины. Прежде чем начать думать о байткоде и прочем, давайте подумаем о нем как об `API`.

Магический API

Если мы описываем заклинания обычным `C++` кодом, какой `API` нам нужен для его вызова? Что за базовые операции определены в движке игры, с помощью которых можно определять заклинания?

Большинство заклинаний в конце концов меняют одну из статистик волшебника, так что мы начнем с функций для такого изменения:

```
void setHealth(int wizard, int amount);
void setWisdom(int wizard, int amount);
void setAgility(int wizard, int amount);
```

Первый параметр указывает на какого волшебника мы влияем, например `0` для игрока, а `1` для его противника. Таким образом можно, например, лечить своего волшебника и наносить урон противнику. С помощью всего трех этих методов можно описать очень много магических эффектов.

Если заклинание будет просто тихо изменять состояния, игровая логика будет работать, но играть будет скучно до слез. Давайте это исправим:

```
void playSound(int soundId);
void spawnParticles(int particleType);
```

Эти функции не влияют на геймплей, но улучшают *восприятие* игры. Мы можем добавить тряску камеры, анимации и т.д., но для начала хватит и этого.

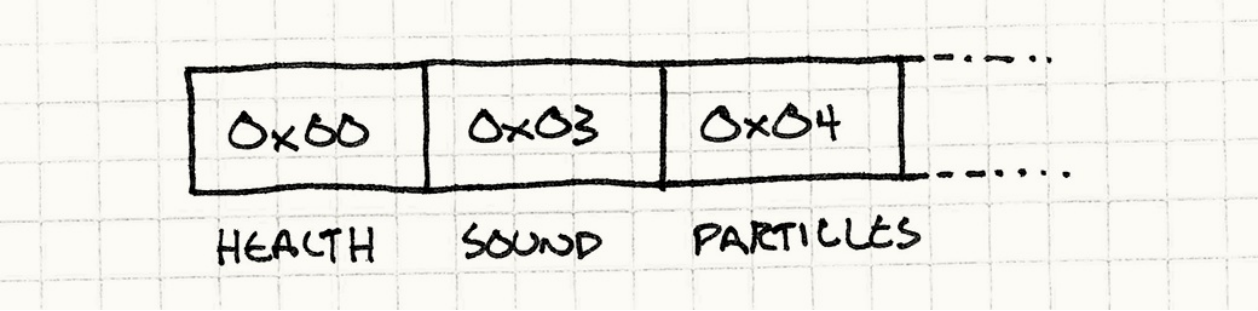
Набор магических инструкций

Теперь посмотрим как мы можем превратить *программный* `API` в нечто, что можно контролировать из данных. Начнем с малого, а потом будем дорабатывать то что получается. Сейчас мы считаем что в методы никаких параметров не передается. Скажем метод `set___()` всегда будет влиять на волшебника игрока и устанавливать параметр в максимальное значение. Аналогично операции спецэффектов (FX) будут проигрывать жестко заданный звук или систему частиц.

Теперь мы можем описать наше заклинание как набор инструкций. Каждая указывает какую операцию нужно выполнить. Перечислим их:

```
enum Instruction
{
  INST_SET_HEALTH = 0x00,
  INST_SET_WISDOM = 0x01,
  INST_SET_AGILITY = 0x02,
  INST_PLAY_SOUND = 0x03,
  INST_SPAWN_PARTICLES = 0x04
};
```

Чтобы закодировать заклинание в данных, мы просто сохраняем массив со значениями enum. У нас всего несколько примитивов так что диапазон значений enum помещается в один байт. Это значит что наше заклинание — это всего лишь список байтов — следовательно "байткод".



> Некоторые виртуальные машины используют больше чем один байт для каждой инструкции и декодируют их по более сложным правилам.
>
> Тем не менее одного байта достаточно для Java Virtual Machine и для Microsoft's Common Language Runtime, на котором основана платформа `.NET` и нас вполне устраивает работа этих машин.

Чтобы выполнить одну инструкцию, мы смотрим что у нас за примитив и вызываем соответствующий метод `API`:

```
switch (instruction)
{
  case INST_SET_HEALTH:
    setHealth(0, 100);
    break;

  case INST_SET_WISDOM:
    setWisdom(0, 100);
    break;

  case INST_SET_AGILITY:
    setAgility(0, 100);
    break;

  case INST_PLAY_SOUND:
    playSound(SOUND_BANG);
    break;

  case INST_SPAWN_PARTICLES:
    spawnParticles(PARTICLE_FLAME);
    break;
}
```

Таким образом, наш интерпретатор формирует мост между мирами кода и данных. Мы можем поместить его в небольшую обертку виртуальной машины, которая выполняет все заклинание следующим образом:

```
class VM
{
public:
  void interpret(char bytecode[], int size) {
    for (int i = 0; i < size; i++) {
      char instruction = bytecode[i];
      switch (instruction)
      {
         // Случаи для каждой инструкции...
      }
    }
  }
};
```

Теперь у нас есть своя виртуальная машина. К сожалению, она не очень то и гибкая. Мы не можем описать заклинание которое действует на персонаж противника или понижает статистику. И звук мы можем играть всего только один!

Чтобы получить что-то похожее на настоящий язык, нам нужно добавить параметры.

Машина стеков

Чтобы выполнить сложное выражение с вложениями, нужно начинать с подвыражений. Вычисляем их и передаем результаты вверх в качестве аргументов содержащих их выражений до тех пор, пока все выражение не будет вычислено.

Шаблон *Интерпретатор* моделирует это явно в виде дерева вложенных объектов, но нас интересует скорость и плоский список инструкций. Мы все равно хотим чтобы результат подвыражений попадал в правильное ограничивающее выражение. Но так как наши данные теперь плоские, чтобы этим управлять нам нужно использовать порядок инструкций. Мы будем делать это также как работает процессор: с помощью стека.

Такая архитектура без всякой фантазии названа стековой машиной. Языки программирования типа `Forth` [1]), `PostScript` [2] и `Factor` [3]) предоставляют эту модель пользователю напрямую.

```
class VM
{
public:
  VM()
  : stackSize_(0)
  {}

  // Other stuff...

private:
  static const int MAX_STACK = 128;
  int stackSize_;
  int stack_[MAX_STACK];
};
```

Виртуальная машина содержит внутренний стек значений. В нашем примере единственный тип значений, с которыми работают наши инструкции — это числа. Поэтому мы можем использовать простой массив целых чисел. Как только любому биту данных нужно будет быть переданным из одной инструкции в другую — это будет происходить через стек.

Как следует из имени, значения могут помещаться и извлекаться из стека. Добавим методы для этих операций:

```
class VM
{
private:
  void push(int value) {
    // Поверяем переполнение стека.
    assert(stackSize_ < MAX_STACK);
    stack_[stackSize_++] = value;
  }

  int pop() {
    // Проверяем что стек не пустой.
    assert(stackSize_ > 0);
    return stack_[--stackSize_];
  }

  // Другие вещи...
};
```

Когда инструкции требуется получить параметры, она берет их из стека:

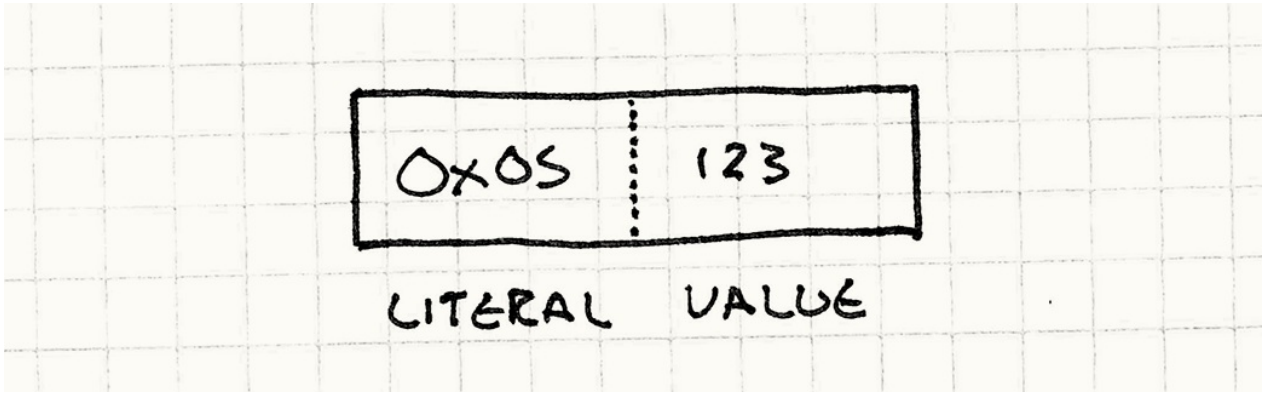
```
switch (instruction)
{
  case INST_SET_HEALTH:
  {
    int amount = pop();
    int wizard = pop();
    setHealth(wizard, amount);
    break;
  }

  case INST_SET_WISDOM:
  case INST_SET_AGILITY:
    // Аналогично, как сверху...

  case INST_PLAY_SOUND:
    playSound(pop());
    break;

  case INST_SPAWN_PARTICLES:
    spawnParticles(pop());
    break;
}
```

Чтобы извлечь значение *на* стек, нам нужна еще одна инструкция: литерал. Она представляет собой сырое целое значение. Но откуда берется *это* значение? Как нам избежать здесь бесконечной регрессии вниз?

Можно воспользоваться тем фактом, что наш поток инструкций — это всего лишь последовательность байт: мы можем добавить число прямо в поток инструкций. Определим еще один тип инструкций для численных литералов:

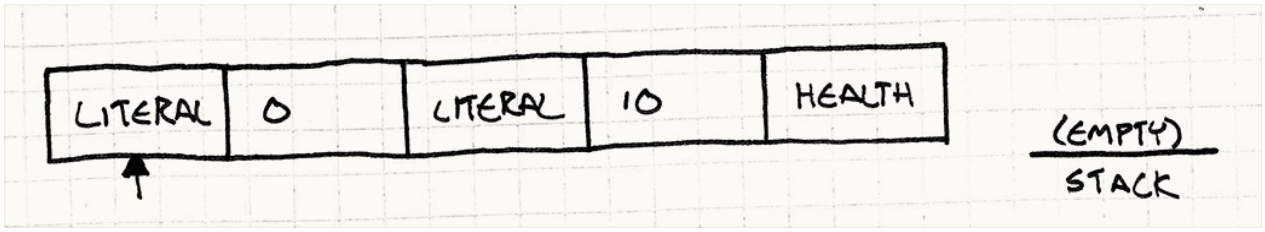
```
case INST_LITERAL:
{
  // Считываем следующий байт из байткода.
  int value = bytecode[++i];
  push(value);
  break;
}
```

> Здесь я просто считываю один байт для получения значения числа чтобы не заморачиваться с кодом, считывающим многобайтные целые значения, но в настоящей реализации вам понадобятся литералы, способные покрывать полный диапазон численных типов.

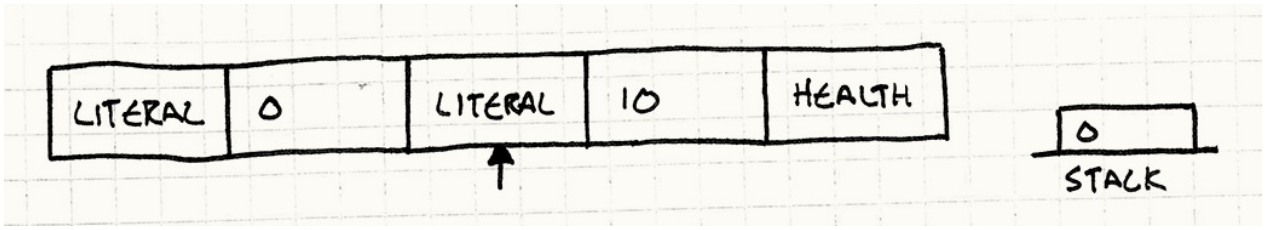


Она считывает следующий байт в потоке байткода *как число* и помещает его на стек.

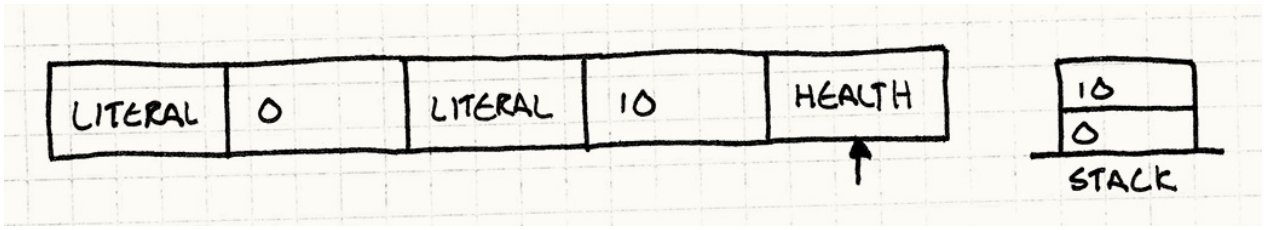
Давайте сформируем строку из нескольких инструкций и посмотрим как интерпретатор будет ее выполнять чтобы понять как работает стек. Начнем с пустого стека и интерпретатора, указывающего на первую инструкцию.



Первой выполняется инструкция `INST_LITERAL`. Она считывает следующий байт из баткода (`0`) и помещает его в стек.



Далее выполняется второй `INST_LITERAL`. Он считывает `10` и тоже помещает в стек.



И наконец вызывается `INST_SET_HEALTH`. Она берет со стека значение `10` и сохраняет его в `amount`. Далее берет со стека `0` и сохраняет в `wizard`. После этого происходит вызов `setHealth()` с этими параметрами.

Та-да! У нас получилось заклинание, устанавливающее здоровье волшебника игрока в значение десять единиц. Мы получили достаточную гибкость для того, чтобы устанавливать здоровье любого из волшебников в любое значение. Аналогично мы можем проигрывать разные звуки и порождать частицы.

Но... это все еще похоже на формат *данных*. Мы не можем, например, установить здоровье волшебника равным половине его мудрости. А нашим дизайнерам было бы интересно указывать *правила* для заклинаний, а не просто *значения*.

Поведение = композиция

Если думать о нашей маленькой виртуальной машине как о языке программирования, все что он поддерживает — это всего несколько встроенных функций и константных параметров для них. Чтобы описывать на байткоде *поведение*, нам не хватает *композиции*.

Нашим дизайнерам нужно иметь возможность создавать выражения, комбинирующие различные значения интересным образом. Простой пример — это возможность модифицировать характеристики персонажа *на* определенное значение, а не устанавливать *в* конкретное значение.

Для этого нам нужно учитывать текущее значение характеристики. У нас есть инструкции для *установки* значений. А теперь нам нужно добавить инструкции для *чтения* значений:

```
case INST_GET_HEALTH:
{
  int wizard = pop();
  push(getHealth(wizard));
  break;
}

case INST_GET_WISDOM:
case INST_GET_AGILITY:
  // Ну вы поняли...
```

Как вы видите, они работают со стеком в обе стороны. Они берут со стека параметр чтобы узнать для какого волшебника нужно получить характеристику, получают это значение и помещают его обратно в стек.

Теперь мы можем написать заклинание, которое будет копировать характеристики. Мы можем создать заклинание, которое будет устанавливать ловкость волшебника равной его мудрости или чары, которые будут приравнивать здоровье одного волшебника здоровью другого.

Уже лучше, но все равно ограничения остаются. Нам нужна арифметика. Пришло время для нашей маленькой виртуальной машины научиться складывать `1+1`. Добавим еще несколько инструкций. Теперь вам уже наверное проще будет понять как они работают. Просто покажу сложение:

```
case INST_ADD:
{
  int b = pop();
  int a = pop();
  push(a + b);
  break;
}
```

Как и остальные наши инструкции, она берет со стека пару значений, выполняет с ними работу и помещает результат обратно в стек. До сих пор каждая новая наша инструкция давала нам фиксированный прирост в выразительности, а теперь мы сделали большой шаг. Это не очевидно, но теперь мы можем обрабатывать любые типы сложных, многоуровневых вычислений.

Перейдем к более сложному примеру. Скажем, мы хотим создать заклинание, которое будет устанавливать значение здоровья волшебника равным среднему арифметическому между его ловкостью и мудростью. Вот его код:

```
setHealth(0, getHealth(0) +
    (getAgility(0) + getWisdom(0)) / 2);
```

Вы можете подумать, что нам нужна инструкция для обработки явной группировки, которая управляет подчиненностью в данном выражении, но стек косвенно уже все это поддерживает. Вот как мы стали бы вычислять значение вручную:

1. Получаем текущее значение здоровья волшебника и сохраняем его.
2. Получаем ловкость волшебника и сохраняем ее.
3. Делаем тоже самое с мудростью.
4. Складываеп последние два значения и запоминаем результат.
5. Делим его на два, сохраняем результат.
6. Вспоминаем значение здоровья волшебника и прибавляем к нему результат.
7. Берем результат и устанавливаем в качестве текущего значения здоровья волшебника.

Обратили внимания на все эти "сохраним" и "вспомним"? Каждое "сохраним" — это помещение в стек (push), а каждое "вспомним" — это получение значения со стека (pop). Это значит что мы достаточно просто можем перевести наши действия в байткод. Например, первая строка для получения текущего значения здоровья волшебника выглядит так:

```
LITERAL 0
GET_HEALTH
```

Этот фрагмент байткода помещает значение здоровья волшебника в стек. Если мы механически транслируем каждую строку таким образом, мы получим фрагмент байткода, вычисляющий наше оригинальное выражение. Чтобы почувствовать как комбинируются инструкции, я продемонстрирую вам это ниже.

Чтобы показать как стек меняется со временем, мы проследим за примером выполнения, в котором у волшебника здоровье будет равно `45`, ловкость — `7` и мудрость — `11`. После выполнения каждой инструкции видно как выглядит стек и приведен небольшой комментарий.

```
LITERAL 0   [0]             # индекс волшебника
LITERAL 0   [0, 0]          # индекс волшебника
GET_HEALTH  [0, 45]         # getHealth()
LITERAL 0   [0, 45, 0]      # индекс волшебника
GET_AGILITY [0, 45, 7]      # getAgility()
LITERAL 0   [0, 45, 7, 0]   # индекс волшебника
GET_WISDOM  [0, 45, 7, 11]  # getWisdom()
ADD         [0, 45, 18]     # сложение ловкости и мудрости
LITERAL 2   [0, 45, 18, 2]  # Divisor
DIVIDE      [0, 45, 9]      # вычисление среднего ловкости и мудрости
ADD         [0, 54]         # добавление среднего к текущему значению
SET_HEALTH  []              # установка здоровья равным результату
```

Если вы посмотрите на состояние стека на каждом шаге, вы увидите каким магическим образом мы управляем потоком инструкций. Мы помещаем в стек `0` в качестве индекса волшебника в начале и он таки будет болтаться в самом низу до тех пор пока не понадобится нам в самом конце для инструкции `SET_HEALTH`.

> Возможно эпитет "магия" здесь даже недостаточен.

Виртуальная машина

Я могу и дальше добавлять новые и новые инструкции, но думаю на этом стоит остановиться. На данный момент у нас уже есть маленькая виртуальная машина, позволяющая довольно свободно определять поведение в простом, компактном формате данных. Хотя "байткод" и "виртуальная машина" и звучат пугающе, вы можете убедиться сами что все что здесь есть — это стек, цикл и управляющие переключатели.

Помните нашу исходную задачу о том что поведение нужно определять в режиме песочницы? Теперь когда вы знаете как вы реализовали виртуальную машину мы эту задачу выполнили. Байткод не может сделать ничего злонамеренного и не может выбраться за пределы своего места в движке, так как общаться с остальным кодом он может только через свои инструкции.

Мы сами контролируем сколько памяти виртуальная машина использует и какой у нее размер стека. Поэтому нам нужно тщательно следить за тем чтобы не было переполнения. Мы даже можем управлять тем сколько времени она использует. В нашем цикле инструкций мы можем подсчитать сколько всего выполнили и прерваться когда достигнем установленного предела.

> Управлять временем выполнения в нашем примере необязательно, потому что у нас нет никаких инструкций для организации циклов. Мы можем ограничить время выполнения только ограничением общего размера байткода. Из этого кстати следует что наш байткод не является полным по Тьюрингу.

Осталась всего одна проблема: само создание байткода. До сих пор мы составляли псевдокод и компилировали его вручную. Если у вас конечно нет *кучи* свободного времени, на практике так работать не получится.

Утилита для волшебства

Нашей изначальной задачей было получение *высоко*уровневого способа задания поведения, но в результате у нас получилось нечто еще более *низко*уровневое чем `C++`. У этого решения есть нужная нам производительность и безопасность, которые мы хотели, но никакого удобства работы для дизайнера.

Чтобы устранить это препятствие нам нужен инструмент. Нам нужна программа, позволяющая пользователям определять поведение для заклинания на высоком уровне, а затем генерировать низкоуровневый байткод для стековой машины.

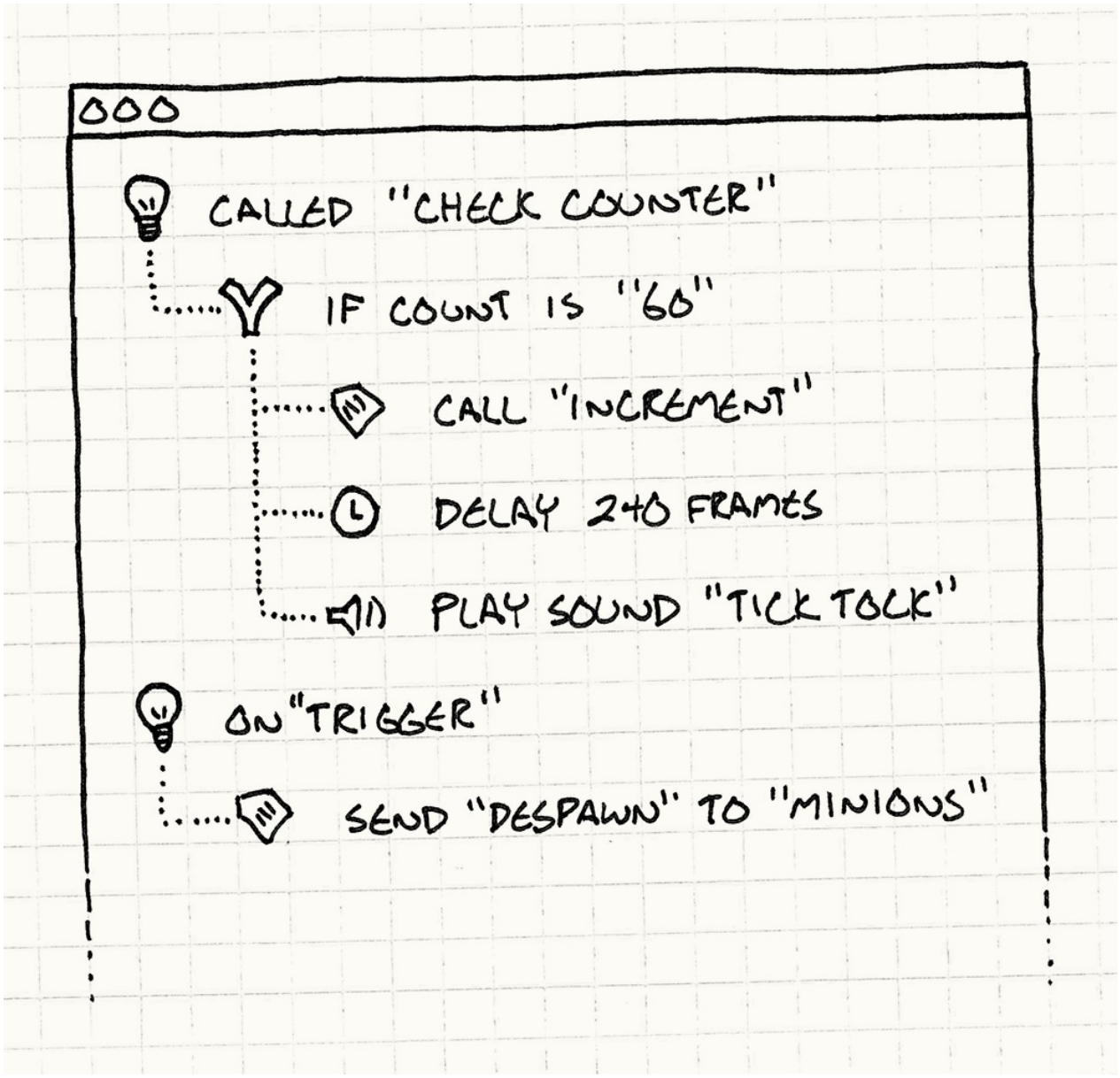
Звучит еще страшнее, чем создание виртуальной машины. Многие программисты изучали компиляторы в качестве дисциплины в колледже и начинают испытывать симптомы посттравматического синдрома от одного вида книжки с драконом или слов "lex)" и "yacc".

> Я говорю конечно о классической книге Компиляторы. принципы технологии и инструментарий

По правде говоря, компиляция текстоподобного языка не настолько страшная задача, но пожалуй чересчур обширная, чтобы говорить о ней подробно в этой главе. Однако вам это может и не понадобиться. Я ведь говорил об *инструменте* — т.е. *компиляторе*, исходниками для которого должен быть *текстовый файл*.

Я скорее призываю вас к созданию инструмента с графическим интерфейсом, в котором люди смогут определить нужное им поведение, даже если они не слишком технически подкованы. Написание текста, свободного от синтаксических ошибок слишком сложно для неподготовленных людей, у которых нет опыта когда на них годами ругается компилятор.

Вместо этого вы можете создавать приложение, которое позволит пользователю "скриптовать", кликая и перетаскивая маленькие кирпичики или выбирая варианты из меню или работать с другими подобными объектами, удобными для определения поведения.



> Скриптовая система, которую я писал для Henry Hatsworth and the Puzzling Adventure работает именно так.

Это хорошо тем, что, пользуясь только пользовательским интерфейсом, невозможно создать "некорректную" программу. Вместо того чтобы извергать на пользователей сообщения с ошибками, вы можете блокировать кнопки или ограничивать вводимые значения чтобы создаваемые скрипты всегда были корректными.

> Хочу еще раз напомнить насколько важна обработка ошибок. Как программисты мы привыкли рассматривать человеческие ошибки как недопустимые недостатки, которые необходимо в себе искоренять.
>
> Чтобы создать систему, с которой пользователю было бы приятно работать, вам придется принять его человечность, включая склонность ошибаться. Люди только тем и занимаются что делают ошибки и это фундаментальная основа творческого процесса. Корректная их обработка с такими полезными функциями как отмена позволяет пользователям быть более креативными и делать работу лучше.

Таким образом, вам не нужно разрабатывать словарь и писать парсер для маленького языка. Но я знаю что не всем по душе программирование пользовательских интерфейсов. Ну значит в таком случае у меня нет для вас хороших новостей.

В конце концов шаблон ведь о том чтобы иметь возможность описывать поведение в дружественной пользователю высокоуровневой форме. Вам нужно разнообразить игровой опыт. Чтобы эффективно выполнять поведение вам нужно преобразовывать результат в низкоуровневую форму. Это серьезная работа, но при должном усилии она окупится.

Архитектурные решения

Я старался сохранить эту главу настолько простой, насколько это возможно, но все, что мы на самом деле сделали — это создали новый язык. Это прекрасное открытое пространство для творчества. Исследовать его — громадное удовольствие, но при этом нужно не забывать и о самой игре.

> Но раз уж эта глава получилась самой длинной в книге — значит с задачей я не справился.

Как инструкции будут получать доступ к стеку?

Виртуальные машины с байткодом делятся на два основных подвида: основанные на стеке (stack-based) и основанные на регистрах (register-based). В виртуальных машинах, основанных на стеках, инструкции всегда работают с верхом стека, как в наших примерах кода. Например, `INST_ADD` берет сверху два значения, складывает их и помещает на верх стека результат.

У виртуальных машин на основе регистров тоже есть стек. Единственное отличие заключается в том что инструкции могут считывать из него значения на более глубоком уровне. Вместо того чтобы `INST_ADD` всегда брала операнды *сверху* стека, у нее в байткоде хранится два индекса, указывающие откуда брать операнды.

- **Виртуальные машины на основе стека:**
  - *Инструкции минимального размера*. Так как каждая инструкция берет аргументы с верха стека, для этого не нужно декодировать никаких данных. Это значит что каждая инструкция довольно мала и обычно занимает один байт.
  - *Генерация кода проще*. Когда вы будете писать компилятор или инструмент для вывода байткода, вы увидите что генерировать байткод для стека проще. Так как каждая инструкция работает с вершиной стека, вам нужно просто вывести инструкции в правильном порядке, чтобы между ними передавались параметры.
  - *У вас будет больше инструкций*. Каждая инструкция видит только верх стека. Это значит что для того чтобы сгенерировать код типа `a = b + c`, вам нужно будет использовать отдельные инструкции для помещения `b` и `c` наверх стека, выполнить операцию и потом поместить результат в `a`.
- **Виртуальные машины на основе регистров:**
  - *Инструкции больше*. Так как инструкциям нужно указывать откуда из стека брать аргументы, сами инструкции будут требовать больше бит для описания. Например, в `Lua` — наверное самой известной виртуальной машине на основе регистров — на инструкцию отводится целых `32` бита. `6` из них используется для типа инструкции, а остальные — для аргументов.

    > `Lua` не определяет собственную форму байткода и меняет ее от версии к версии. То, что я написал выше — справедливо для `Lua 5.1`. Прекрасное введение в `Lua` можно прочитать здесь.
  - *Вам нужно меньше инструкций*. Так как каждая из инструкций может выполнять больше работы, многие инструкции вам будут просто не нужны. Некоторые даже считают, что в результате мы получаем прирост в производительности, потому что нам приходится меньше перетасовывать данные в стеке.

Так что же вам выбрать? Я рекомендую остановиться на варианте виртуальной машины со стеком. Ее легче реализовывать и гораздо проще генерировать код для нее. Виртуальная машина на базе регистров получила репутацию более быстрой после того как на нее перешла Lua, но это все сильно зависит от ваших инструкций и многих других деталей реализации виртуальной машины.

Какие инструкции у нас бывают?

Ваш набор инструкций определяет границы, в рамках которых мы можем выражаться в байткоде и сильно влияет на производительность вашей виртуальной машины. Вот список основных типов инструкций, которые вам могут понадобиться:

- **Внешние примитивы**. Это те, которые выходят за пределы виртуальной машины, попадают в другие части игры и отвечают за видимые пользователем вещи. Они отвечают за типы реального поведения, которое можно выразить в байткоде. Без них ваша виртуальная машина не сможет делать ничего кроме пустого прожигания циклов процессора.
- **Внутренние примитивы**. Это значения, которыми мы манипулируем внутри самой виртуальной машины типа литералов, арифметики, операторов сравнения и инструкций для жонглирования стеком.
- **Порядок управления**. В нашем примере мы такого не видели, но если вам нужно добиться императивного поведения, условного выполнения или контролировать выполнение инструкций с помощью циклов и выполнять инструкции больше одного раза, вам нужны инструкции контроля над порядком выполнения. На самом низком уровне байткода это реализуется очень просто — с помощью переходов (jumps).

  В нашей инструкции цикла будет присутствовать индекс, чтобы следить за тем, где в байткоде мы находится. Все переходы выполняются изменением этой переменной и присваиванием нового значения, где мы теперь находимся. Другими словами это `goto`. С его помощью можно устроить любые условные переходы более высокого уровня.
- **Абстрактные**. Если ваши пользователи начинают объявлять в данных *слишком* много вещей, им может захотеться использовать некоторые фрагменты байткода повторно вместо того чтобы заниматься копипастингом. Т.е. вам могут понадобиться подобия вызываемых процедур.

  В простейшей форме они представляют собой всего лишь переход. Единственная разница заключается в том, что виртуальная машина поддерживает второй *возвращаемый* стек. Когда она выполняет инструкцию "вызова (call)", она помещает индекс текущей инструкции в возвращаемый стек и затем делает переход к вызванному байткоду. Когда она доходит до "возврата (return)", виртуальная машина берет из возвращаемого стека индекс и переходит по нему назад.

В каком виде могут быть представлены значения?

Наша простенькая виртуальная машина работает только с одним типом данных — целыми. Это все сильно упрощает, потому что наш стек — это просто стек *целых (`int`)* чисел. В более совершенной виртуальной машине поддерживаются разные типы данных: строки, объекты, списки и т.д. И конечно нам нужно определиться с тем, как мы будем их хранить.

- **Единый тип данных:**
  - **Это просто**. Вам не нужно беспокоиться о тегах, преобразованиях и проверках типа.
  - *Вы не можете работать с разными типами данных*. Очевидный недостаток. Попытка хранить разные типы данных в едином представлении — например чисел в виде строк — не лучшее решение.
- **Вариант с тегами (tagged):**

  Это обычное представление для языков с динамической типизацией. Каждое значение состоит из двух частей. Первая часть — это тег типа `` `enum ``, означающий какой тип данных внутри хранится. Оставшиеся биты интерпретируются в соответствии с указанным типом:

  ```
  enum ValueType
  {
    TYPE_INT,
    TYPE_DOUBLE,
    TYPE_STRING
  };

  struct Value
  {
    ValueType type;
    union
    {
      int intValue;
      double doubleValue;
      char* stringValue;
    };
  };
  ```

  - *Значения знают свой тип*. Преимуществом этого решения является то, что мы можем проверить тип значения во время выполнения. Это важно для динамической диспетчеризации и проверки того, что вы не пытаетесь выполнить операцию над типом, который ее не поддерживает.

- *Памяти требуется больше*. Каждому значению приходится хранить вместе с собой несколько дополнительных бит для обозначения типа. А на таком низком уровне, как виртуальная машина даже несколько лишних бит значат очень много.

- **Безтеговые объединения (untagged union):**

  Здесь тоже используется `union`, как и в предыдущем случае, но вместе с ним *не* хранится тип данных. У вас может быть кучка бит, представляющих больше, чем один тип и правильная их интерпретация — это уже ваша забота.

  Именно так представляют данные в памяти языки со статическим типизированием. Так как система типов выполняет проверку того, что вы не перепутали типы еще на этапе компиляции, вам не придется заниматься проверками во время выполнения.

  > Еще именно таким образом хранят данные *нетипизированные* языки программирования, такие как ассемблер или `Forth`. Эти языки возлагают ответственность за правильное использование типов на пользователя. Подход явно не для слабонервных!

  - *Этот вариант компактен*. У вас есть возможность забронировать больше битов для хранения самого значения.
  - *Это быстро*. Отсутствие тегов означает, что вам не придется тратить такты процессора на их проверку во время выполнения. Именно поэтому языки со статическим типизированием настолько быстрее языков с динамическими типами.
  - *Это небезопасно*. А вот и расплата. Плохой байткод, перепутавший указатель с целым значением или наоборот может привести к падению вашей игры.

    > Если ваш байткод скомпилирован из языка со статической типизацией, вы можете подумать, что теперь вы в безопасности, потому что компилятор не сгенерирует некорректный байткод. Это может быть и правдой, но помните, что пользователь злоумышленник может сгенерировать зловредный код и без компилятора.
    >
    > Вот поэтому например в `Java Virtual Machine` перед выполнением программы запускается *верификация байткода*.

- **Интерфейс:**

Объектно-ориентированное решение для значения, которое может быть в одном из нескольких состояний с помощью полиморфизма. Интерфейс предоставляет виртуальные методы для различных типов тестов и конвертирования в таком духе:

```
class Value
{
public:
  virtual ~Value() {}

  virtual ValueType type() = 0;

  virtual int asInt() {
    // Можно вызывать только для целых.
    assert(false);
    return 0;
  }

  // Другие методы конвертации...
};
```

Далее мы сделаем конкретные классы для разных типов данных, наподобие:

```
class IntValue : public Value
{
public:
  IntValue(int value)
  : value_(value)
  {}

  virtual ValueType type() { return TYPE_INT; }
  virtual int asInt() { return value_; }

private:
  int value_;
};
```

- *Неограниченные возможности*. Вы всегда можете определить новый класс за пределами виртуальной машины. Главное, чтобы он реализовывал базовый интерфейс.
- *Объектно-ориентированность*. Если вы придерживаетесь принципов `ООП`, такой подход реализует диспетчеризацию типов "правильным" образом, т.е. определяет специфичное для типа поведение, а не занимается чем-то типа переключения поведения в зависимости от тега.

- *Многословность*. Вам нужно определить отдельный класс с полным набором возможностей для каждого типа данных. Обратите внимание, что в предыдущих примерах мы продемонстрировали полное определение всех типов значений. А здесь ограничились только одним!
- *Неэффективность*. Чтобы воспользоваться свойствами полиморфизма, нам нужно выполнить переход по указателю. А это значит, что даже такое простейшее значение как булевское значение или число обернуты в объект, выделенный в куче. Каждый раз при получении значения нужно будет выполнять вызов виртуального метода.

  В таком ответственном коде как ядро виртуальной машины, даже самое незначительное снижение производительности может быть критичным. На самом деле такое решение страдает от тех же проблем, из-за которых мы отказались от шаблона *Интерпретатор* с тем лишь отличием, что проблема у нас теперь в *значении*, а не *коде*.

Вот что я рекомендую: Если вы способны ограничиться единственным типом данных — так и сделайте. В противном случае используйте объединение с тегом. Именно так и работает большинство интерпретаторов в мире.

Как генерируется байткод?

Самый главный вопрос я оставил напоследок. Я показал вам как *применять* и как *интерпретировать* байткод, но создавать что-то, с помощью чего можно *производить* байткод придется вам. Обычно для этого пишется компилятор, но это не единственное решение.

- **Если вы определяете язык на текстовой основе:**
  - *Вам нужно определить синтаксис*. Сложность этой задачи недооценивают как новички, так и бывалые архитекторы языков. Определить словарь, удовлетворяющий парсер — легко. Определить удовлетворяющий пользователя — крайне сложно.

    Дизайн синтаксиса — это дизайн пользовательского интерфейса и этот процесс не становится легче от того, что интерфейс сводится к строкам или символам.
  - *Вам придется реализовать парсер*. Несмотря на репутацию эта часть довольно простая. Вы можете использовать генератор наподобие `ANTLR` или `Bison` или по моему примеру самостоятельно написать рекурсивный спуск.

    - *Вам придется обрабатывать синтаксические ошибки*. Это одна из самых важных и сложных частей процесса. Когда пользователи допускают синтаксические и семантические ошибки, а они будут, это ваша задача наставить их на путь истинный. Выдавать полезную обратную связь не так уж и просто, если известно только то, что парсер наткнулся на какую-то ошибку пунктуации.
    - *Этот способ практически недоступен для технически неподкованных пользователей*. Мы программисты любим текстовые файлы. Вместе с мощными инструментами командной строки мы можем думать о них как о компьютерных LEGO блоках: простые, но легко сочетаемые миллионом способов.

      Большинство непрограммистов так текстовые файлы не воспринимают. Для них это подобно заполнению налоговой декларации для роботизированного аудитора, который ругается на них даже если они забыли поставить единственную точку с запятой.

- **Если вы разрабатываете графический инструмент:**
    - *Вам придется заниматься пользовательским интерфейсом*. Кнопки, нажатия, перетаскивания и все такое. Некоторых от этого коробит, а мне лично даже нравится. Если вы пойдете по этому пути, крайне важно воспринимать пользовательский интерфейс как залог плодотворной работы дизайнера, а не как неприятную обязанность.

      Каждое потраченное на разработку этого инструмента усилие сделает инструмент проще и приятнее в работе, что напрямую влияет на качество контента в вашей игре. Если бы вы могли заглянуть за кулисы ваших любимых игр, вы бы увидели что секрет их успеха зачастую заключается в удобном инструментарии.
    - *У вас будет меньше ошибок*. Так как пользователь выстраивает поведение интерактивно шаг за шагом, ваше приложение может оберегать его от ошибок прямо во время работы.

      В текстовом языке инструмент не видит *ничего* из пользовательского контента, пока не получит весь файл целиком. Поэтому там предотвращать ошибки гораздо сложнее.
    - *Сложность портирования*. Преимущество текстового файла заключается в том, что текстовый файл универсален. Простой компилятор просто считывает один файл и записывает другой. Портировать его на другую ОС — задача тривиальная.

> Если конечно забыть про символы перевода строки. И кодировки.

Когда вы разрабатываете пользовательский интерфейс, вам нужно выбрать для этого определенный фреймворк и многие из них специфичны для конкретной ОС. Есть конечно и кроссплатформенные тулкиты для пользовательского интерфейса, но они обычно платят за кроссплатформенность привычностью: на любой платформе они смотрятся одинаково странными.

Смотрите также

- Этот шаблон очень близок шаблону Интерпретатор (Interpreter) GoF. Они оба позволяют вам выражать сложное поведение с помощью данных.

  В жизни часто происходит так, что вы начинаете использовать *оба* шаблона. Инструмент, который вы используете для генерации байткода будет содержать внутри дерево объектов, представляющее код. А это именно то, чем занимается шаблон *Интерпретатор*.

  Чтобы превратить это в байткод, вы рекурсивно обходите дерево, также как интерпретатор в шаблоне *Интерпретатор*. *Единственное* различие в том, что вместо мгновенного выполнения примитивных фрагментов поведения, вы выводите их в виде байткода для выполнения позже.

- Язык программирования Lua — это самый известный скриптовый язык из используемых в играх. Внутри он реализован как крайне компактная виртуальная байткод машина на основе регистров.

- Kismet — это графический скриптовый инстурмент, встроенный в `UnrealEd` — редактор движка `Unreal engine`.

- Мой собственный скриптовый язык Wren — это простой интерпретатор байткода на основе стека.

Подкласс песочница (Subclass Sandbox)

Задача

Определение поведения в подклассе с помощью набора операций, предоставляемых базовым классом.

Мотивация

Каждый мальчишка хотел в детстве быть супергероем, но к сожалению с космическими лучами у нас на Земле не густо. Игры помогают хотя бы немного почувствовать себя супергероем. Так как никто из наших дизайнеров так и не научился говорить "нет", в *каждой* игре планируются дюжины, если не сотни различных суперспособностей для героев.

Наш план заключается в том, чтобы иметь базовый класс `Superpower`. Далее мы создаем класс наследник для каждой суперсилы. Делим дизайн документ между программистами поровну и начинаем кодить. Когда все закончили у нас появилась сотня классов суперспособностей.

> Когда вы видите, что у вас образуется *слишком много* подклассов, как в этом примере, это обычно свидетельствует о том, что лучше применить подход управления через данные (data-driven approach). Вместо того, чтобы использовать огромное количество *кода* для определения различных суперсил, попробуйте определить их поведение с помощью *данных*.
>
> В этом вам могут помочь шаблоны типа Объект тип (Type Object), Байткод (Bytecode) или Интерпретатор (Interpreter) GoF.

Мы хотим поразить нашего игрока разнообразием нашего мира. Мы хотим, чтобы в игре была каждая способность, о которой он только мог мечтать в детстве. Это значит, что эти подклассы суперсил могут делать практически все: проигрывать звуки, порождать визуальные эффекты, взаимодействовать с ИИ, создавать и уничтожать другие игровые сущности и вмешиваться в работу физики. Нет ни одного уголка кодобазы, до которого они не могли бы добраться.

Предположим, что мы дадим нашей команде волю и позволим заняться написанием классов суперсил. Что в этом случае произойдет?

- *У нас будет куча избыточного кода*. Хоть разные суперсилы и различаются между собой довольно сильно, их действие все равно частично перекрывает друг друга. Многие из них будут проигрывать звуки и запускать визуальные эффекты похожими способами. Замораживающий луч, испепеляющий луч и луч горчицы Джинна — довольно похожи если рассмотреть их подробнее. Если реализующие все это люди не будут между собой взаимодействовать, они потратят кучу лишнего времени и сгенерируют кучу дублирующего кода.

- *Каждый уголок кода будет связан с этими классами*. Не обладая достаточными знаниями, люди будут писать код, вызывающий подсистемы, которые изначально и не предполагалось связывать с классами суперсил. Если наш рендер организован в виде нескольких хитрых слоев, только один из которых предполагается взаимодействующим с кодом за пределами графического движка, мы можем обнаружить что некоторые суперсилы будут обращаться к каждому из его слоев.

- *Когда эти внешние системы потребуется изменить, связанные с ними суперсилы внезапно могут начать работать неправильно*. Как только мы начинаем связывать различные классы суперсил с самыми разными частями нашего движка, не стоит удивляться потом, что их изменение будет влиять на работу классов суперсил. И это совсем не весело, потому что ваши программисты графики, аудио и пользовательского интерфейса совсем не хотят быть еще и программистами геймплея.

- *Сложно определить инварианты, которым подчиняются все суперсилы*. Предположим, что мы хотим, чтобы все звуки, проигрываемые нашими суперсилами, попадали в очередь с правильными приоритетами. Если каждый из сотни ваших классов будет самостоятельно обращаться к звуковой системе, сделать это будет довольно проблематично.

Чего мы на самом деле хотим — так это возможность выдать каждому программисту геймплея набор примитивов, из которых он сможет конструировать суперсилы. Хотите чтобы суперсила проиграла звук? Вот вам функция `playSound()`. Хотите частиц? Вот `spawnParticles()`. Нам нужно удостовериться что эти операции покрывают все что вам нужно и вам не нужно будет беспорядочно прописывать `#include` для заголовков и совать нос во все уголки остальной кодобазы.

Мы добьемся этого сделав эти операции *защищенными методами (protected methods) базового класса* `Superpower`. То, что мы поместили их все прямо в базовый класс, означает, что у всех классов наследников будет простой к ним доступ. Объявление их защищенными (и вероятно не виртуальными) говорит о том, что они предназначены для того, чтобы *вызываться* только из классов наследников.

Теперь, когда у нас есть игрушки для игры, нам нужно место где можно с ними играть. Специально для этого определим метод *песочницу (sandbox method)*: абстрактный защищенный метод, который должны реализовывать подклассы. Таким образом, для реализации новой силы нам нужно:

1. Создать новый класс, унаследованный от `Superpower`.
2. Переопределить метод песочницу `activate()`.
3. Реализовать его тело с помощью вызовов методов, предоставляемых классом `Superpower`.

Проблему избыточности кода мы можем решить, сделав эти операции как можно больше высокоуровневыми. Каждый раз когда мы видим код, дублирующийся в нескольких подклассах, мы всегда можем поместить его в `Superpower` в качестве новой операции, которую смогут использовать подклассы.

С проблемой излишней связности мы боремся, сосредотачивая всю связность в одном месте. `Superpower` в результате сама будет связана с самыми разными системами игры, а вот сотни унаследованных классов — нет. Вместо этого они будут связаны только со своим базовым классом. Когда одна из систем игры изменится, нам придется изменить и `Superpower`, а вот к десяткам подклассов можно будет не притрагиваться.

Этот шаблон подводит нас к архитектуре с неглубокой, но широкой иерархией классов. Цепочка экземпляров *неглубокая*, но у нас есть просто *уйма* классов, завязанных на `Superpower`. Имея один класс со множеством прямых подклассов, мы получаем в нашей кодовой базе точку приложения усилий. Время и усилия, затраченные на `Superpower`, окупятся при создании широкого набора классов в игре.

> В будущем вы наверняка встретите еще множество людей, критикующих наследование в объектно-ориентированных языках. Наследование *сулит* проблемы — не существует связывания сильнее в кодобазе, чем между базовым классом и подклассом. При этом работать проще с *широким*, а не *глубоким* деревом наследования.

Шаблон

Базовый класс определяет абстрактный **метод песочницу** и несколько **предоставляемых операций (provided operations)**. Объявление их защищенными явно означает, что они предназначены только для использования классами наследниками. Каждый унаследованный **подкласс песочницы** реализует метод песочницы с помощью предоставляемых операций.

Когда использовать

Все очень просто. Шаблон легко найти во множестве кодовых баз, даже за пределами игровой индустрии. Если у вас часто встречаются невиртуальные защищенные методы, вы возможно уже используете его подобие. *Подкласс песочница* следует использовать когда:

- У вас есть базовый класс и множество дочерних.
- Базовый класс способен реализовывать все операции, которые нужны для работы дочерним.
- В поведении подклассов наблюдается много совпадений и вы хотели бы упростить кодовую базу за счет повторного использования кода.
- Вы хотите минимизировать связность между этими дочерними классами и остальной программой.

Имейте в виду

"Наследование" во многих программистских кругах сегодня стало чуть ли не ругательством и одна из причин заключается в том, что базовые классы имеют тенденцию обрастать все большим и большим количеством кода. И этот шаблон, как никакой другой, подвержен этой тенденции.

Так как подклассам приходится общаться с остальной игрой через базовый класс, базовый класс оказывается связанным со всеми системами, с которыми вынужден общаться хотя бы один его дочерний класс. Конечно подклассы настолько же сильно связаны со своим базовым классом. Эта паутина связей не даст вам легко изменить кодовую базу без риска что-либо разрушить — классическая проблема хрупкости базового класса.

Обратной стороной монеты является то, что связывание сосредоточено на базовом классе, а классы наследники гораздо более явным образом отделены от остального мира. В идеале основная часть вашего поведения будет сосредоточена в этих подклассах. А это значит, что большая часть вашей кодобазы изолирована и ее легче поддерживать.

Так что если вы видите, что ваша кодовая база превращается в гигантскую миску тушенки, попробуйте выделить часть предоставляемых операций в отдельные классы, с которыми базовый класс сможет частично разделить ответственность. В этом вам поможет шаблон Компонент.

Пример кода

Так как этот шаблон довольно прост, примеров кода не будет слишком много. Это не значит что он бесполезен. Этот шаблон о *намерении*, а не о сложности реализации.

Начнем с базового класса `Superpower`:

```
class Superpower
{
public:
  virtual ~Superpower() {}

protected:
  virtual void activate() = 0;

  void move(double x, double y, double z) {
    // Здесь код...
  }

  void playSound(SoundId sound, double volume) {
   // Здесь код...
  }

  void spawnParticles(ParticleType type, int count) {
   // Здесь код...
  }
};
```

Метод `activate()` — это метод песочница. Так как он виртуальный и абстрактный, подклассы *должны* его переопределять. Таким образом, это будет очевидно для тех, кто будет работать над нашими классами сил.

Остальные защищенные методы `move()`, `playSound()` и `spawnParticles()` — это предоставляемые операции. Это именно их подклассы будут вызывать в своей реализации `activate()`.

Мы не реализуем предоставляемые операции в этом примере, но в настоящей игре здесь был бы реальный код. Именно в этих методах будет проявляться связность `Superpower` с остальными частями игры: `move()` работает с физическим кодом, `playSound()` общается с аудио движком, и т.д. Так как это все находится в *реализации* базового класса, вся связность инкапсулируется внутри самого `Superpower`.

А теперь выпускаем наших радиоактивных пауков и получаем суперсилу. Вот она:

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate() {
    // Взмываем в небо.
    playSound(SOUND_SPROING, 1.0f);
    spawnParticles(PARTICLE_DUST, 10);
    move(0, 0, 20);
  }
};
```

> Ну ладно. Возможно способность *прыгать* — это не слишком *супер*. Я просто не хочу слишком переусложнять пример.

Эта сила подбрасывает супергероя в воздух, проигрывает сопроводительный звук и порождает облачко пыли. Если бы все суперсилы были такими простыми — просто комбинация звука, эффекта с частицами и движения — нам бы вообще шаблон не понадобился. Вместо этого Superpower мог бы просто содержать готовую реализацию activate(), получающую доступ к полям `ID` звука, типу частиц и движения. Но это могло бы сработать, если бы силы работали одинаково, но просто с разными данными. Доработаем его немного:

```
class Superpower
{
protected:
  double getHeroX() {
    // Здесь код...
  }

  double getHeroY() {
    // Здесь код...
  }

  double getHeroZ() {
    // Здесь код...
  }

  // Остальное...
};
```

Здесь мы добавляем несколько методов для получения позиции игрока. Теперь наш подкласс `SkyLaunch` может их использовать:

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate() {
    if (getHeroZ() == 0) {
      // Мы на земле, значит можем прыгать.
      playSound(SOUND_SPROING, 1.0f);
      spawnParticles(PARTICLE_DUST, 10);
      move(0, 0, 20);
    } else if (getHeroZ() < 10.0f) {
      // Невысоко над землей, значит можем делать двойной прыжок.
      playSound(SOUND_SWOOP, 1.0f);
      move(0, 0, getHeroZ() — 20);
    } else {
      // Находимся в воздухе и можем выполнить подкат.
      playSound(SOUND_DIVE, 0.7f);
      spawnParticles(PARTICLE_SPARKLES, 1);
      move(0, 0, -getHeroZ());
    }
  }
};
```

Так как у нас появился доступ к части состояния, теперь наш метод песочница может эффективнее управлять потоком выполнения. Всего несколько простых выражений `if` и вы можете реализовать все что захотите. Когда у вас в качестве метода песочницы будет полноценный метод с необходимым кодом, вас только небо остановит.

> Ранее я предлагал применить для суперсил подход с описанием с помощью данных (data-driven approach). А вот и причина почему *не* стоит этого делать. Если ваше поведение достаточно сложное и императивное, его сложнее будет задавать с помощью данных.

Архитектурные решения

Как вы видите, это довольно "мягкий" шаблон. Он описывает базовую идею, но не слишком акцентируется на деталях механики. Это значит, что каждый раз, когда вы его применяете, вы можете делать интересные решения. Вот над чем стоит поразмыслить.

Какие операции нужно предоставить?

Это самый большой вопрос. От него зависит насколько шаблоном будет удобно пользоваться и насколько он будет полезен. В самом минималистичном варианте, базовый класс вообще не предоставляет *никаких* операций. Все что у него есть — это метод песочница. Чтобы его реализовать, вам нужно вызывать системы за пределами базового класса. Если вы выберете такую тактику поведения, можно сказать, что вы вообще не используете сам шаблон.

Другая крайность — это базовый класс, предоставляющий любые операции, которые могут понадобиться подклассам. Подклассы привязаны только к базовому классу и вообще не общаются с внешними системами.

> В частности, это значит, что в исходнике каждого такого подкласса будет только один `#include`, подключающий базовый класс.

Между этими двумя крайностями лежит некое среднее решение, когда часть операций предоставляется базовым классом, а остальные используются из других систем напрямую. Чем больше операций вы представляете, тем меньше у вас связности подклассов со внешним системами, но зато больше связность с базовым классом. Мы снижаем связность у классов наследников, но увеличиваем ее у самого базового класса.

Это очень выгодно, когда у вас есть множество классов наследников, связанных со внешними системами. Перенося ее в предоставляемые операции, вы концентрируете связность в одном месте: в базовом классе. И чем чаще вы это делаете, тем больше и сложнее для поддержки становится базовый класс.

Так где же провести черту? Вот несколько основных правил:

- Если предоставляемые операции используются только несколькими подклассами, вы не получите большого выхлопа за свои вложения. Вы увеличите сложность базового класса, которая скажется на всем прочем, но от этого снизится связность всего нескольких наследников.

  Так стоит делать, если эти операции пересекаются с уже существующими. Но возможно проще и очевиднее будет просто позволить подклассам обратиться к внешним системам напрямую.

- Когда вы вызываете метод в каком-либо другом месте игры, лучше, если этот метод не изменяет никакого состояния. Связность все равно увеличивается, но это "безопасная" связность, потому что она ничего в игре не ломает.

> Я не зря беру слово"безопасность" в кавычки, потому что технически даже получение данных может добавить вам проблем. Если ваша игра многопоточная, вы можете пытаться читать какое-то значение в то время, когда оно изменяется. И если не будете достаточно осторожны, у вас окажутся некорректные данные.
>
> Еще один хитрый случай — это когда состояние вашей игры строго детерминировано (что практикуют многие онлайновые игры для сохранения синхронизации между игроками). Если вы получаете доступ к чему либо за пределами синхронизированного состояния игры, у вас могут образоваться крайне опасные недетерминированные баги.

Вызовы, которые меняют состояние в свою очередь гораздо сильнее связывают части вашей кодобазы и вам сложнее будет анализировать такие связи. Этот факт делает их хорошими кандидатами на включение в список предоставляемых операций в более видимый для анализа базовый класс.

- Если реализация предоставляемых операций сводится просто к вызову какой-либо внешней системы — большой пользы она не несет. В этом случае может быть проще просто вызвать внешнюю систему напрямую.

  Однако даже простейшее перенаправление может быть полезным: такие методы зачастую обращаются к состояниям, которые нежелательно напрямую видеть классам наследникам. Предположим что `Superpower` предоставляет такую операцию:

  ```
  void playSound(SoundId sound, double volume)
  {
    soundEngine_.play(sound, volume);
  }
  ```

  Это просто обращение к одному из полей `soundEngine_` из `Superpower`. Выигрыш здесь в том, что поле осталось инкапсулированным в `Superpower` и подклассы его не видят.

Следует ли предоставлять методы напрямую или через содержащий их объект?

Сложность этого шаблона заключается в том, что в результате у вас образуется огромное количество методов, сосредоточенное в одном базовом классе. Этого можно избежать, переместив часть методов в отдельные классы. А предоставляемые операции будут просто возвращать один из этих объектов.

Например, чтобы позволить силе проигрывать звук, мы можем добавить такую возможность прямо в `Superpower`:

```
class Superpower
{
protected:
  void playSound(SoundId sound, double volume) {
    // Здесь код...
  }

  void stopSound(SoundId sound) {
    // Здесь код...
  }

  void setVolume(SoundId sound) {
    // Здесь код...
  }

  // Метод песочница и другие операции...
};
```

Но, у нас ведь и так слишком много всего в `Superpower`, а нам хотелось бы этого избежать. Поэтому мы сделаем отдельный класс `SoundPlayer` и перенесем эту функциональность в него:

```
class SoundPlayer
{
public:
  void playSound(SoundId sound, double volume) {
    // Здесь код...
  }

  void stopSound(SoundId sound) {
    // Здесь код...
  }

  void setVolume(SoundId sound) {
    // Здесь код...
  }
};
```

А теперь `Superpower` будет просто предоставлять доступ к этому классу:

```
class Superpower
{
protected:
  SoundPlayer& getSoundPlayer() {
    return soundPlayer_;
  }

  // Метод песочница и другие операции...

private:
  SoundPlayer soundPlayer_;
};
```

Подобный перенос предоставляемых операций во вспомогательные классы имеет следующие преимущества:

- *Уменьшается количество методов в базовом классе*. В нашем примере мы избавились от трех методов за счет одного получателя класса (getter).
- *Код во вспомогательном классе обычно легче поддерживать*. Ключевые базовые классы типа `Superpower`, несмотря на наши лучшие намерения, может быть сложно изменять, потому что от них слишком много всего зависит. Перенос функциональности в другой менее связанный дополнительный класс, упрощает ее изменение без ущерба для других вещей.
- *Снижается связность между базовым классом и остальными системами*. Когда метод `playSound()` находился прямо в `Superpower`, это значило, что наш класс был напрямую связан с `SoundId` и остальным аудио кодом, вызываемым реализацией. Перенос всего этого в `SoundPlayer` снижает связность `Superpower` до одного класса `SoundPlayer`, в котором теперь сосредоточены все остальные зависимости.

Как базовый класс будет получать нужно ему состояние?

Вашему базовому классу часто придется получать данные, которые он хочет инкапсулировать и держать невидимыми для своих подклассов. В нашем первом примере, класс `Superpower` предоставлял метод `spawnParticles()`. Если для его реализации нужен объект системы частиц, то как нам его получить?

- **Передаем в конструктор базового класса:**

  Проще всего передать его в качестве аргумента конструктора базового класса:

```
class Superpower
{
public:
  Superpower(ParticleSystem* particles)
  : particles_(particles)
  {}

  // Метод песочница и другие операции...

private:
  ParticleSystem* particles_;
};
```

В этом случае мы можем быть уверенными, что у каждой суперсилы будет возможность воспользоваться эффектами сразу после создания. Но давайте посмотрим на класс наследник:

```
class SkyLaunch : public Superpower
{
public:
  SkyLaunch(ParticleSystem* particles)
  : Superpower(particles)
  {}
};
```

Здесь видна очевидная проблема. Каждому классу наследнику придется иметь конструктор, вызывающий конструктор базового и передающий в него этот аргумент. А это значит, что каждый класс наследник буде частью состояния, о котором мы хотим чтобы он вообще не знал.

Для поддержки это тоже сплошная головная боль. Если позже мы захотим добавить еще одну часть состояния в базовый класс, нам придется изменить и все конструкторы всех унаследованных от него классов.

- **Выполняем двухшаговую инициализацию:**

  Чтобы не передавать все через конструктор, мы можем разделить инициализацию на два шага. Конструктор не будет принимать никаких параметров и просто создает объект. После этого мы вызываем отдельный метод, объявленный прямо в базовом классе и передаем ему оставшуюся часть необходимых ему данных.

```
Superpower* power = new SkyLaunch();
power->init(particles);
```

Обратите внимание, что раз мы ничего не передаем в конструктор `SkyLaunch`, он не связан ни с чем, что мы хотели бы оставить личным (private) в `Superpower`. Проблема с этим подходом в том, что вам всегда нужно помнить о необходимости вызова `init()`. Если вы когда-нибудь об этом забудете, у вас будет сила, застрявшая в некоем полусозданном состоянии и ничего не делающая.

Чтобы исправить это, мы можем инкапсулировать весь процесс внутри одной функции следующим образом:

```
Superpower* createSkyLaunch(ParticleSystem* particles)
{
  Superpower* power = new SkyLaunch();
  power->init(particles);
  return power;
}
```

> Использовав здесь трюк с приватным конструктором и дружественным классом, вы можете быть уверены что функция `createSkylaunch()` является *единственным* способом создания силы. Таким образом вы никогда не забудете ни об одном этапе инициализации.

- **Сделаем состояние статичным:**

  В предыдущем примере мы инициализировали каждый *экземпляр* `Superpower` системой частиц. Это имеет смысл, если каждая сила нуждается в собственном уникальном состоянии. Но что, если наша система частиц реализована как Синглтон (Singleton) и используется всеми силами совместно.

  В этом случае мы можем сделать состояние приватным для базового класса и даже *статичным (static)*. Игра все равно будет проверять инициализацию состояния, но *класс* `Superpower` придется инициализировать только один раз, а не для каждого экземпляра.

  > Имейте в виду, что при этом у нас появляется множество проблем из-за синглтона: единое состояние оказывается общим для большого множества объектов (всех экземпляров `Superpower`). Система частиц инкапсулирована и *не видна* глобально, что есть хорошо, но она все равно усложняет понимание работы суперсил, потому что они все работают с одним и тем же объектом.

```
class Superpower
{
public:
  static void init(ParticleSystem* particles) {
    particles_ = particles;
  }

  // Метод песочница и другие операции...

private:
  static ParticleSystem* particles_;
};
```

Обратите внимание что здесь статичны и `init()` и `particles_`. Пока игра вызывает `Superpower::init()` перед всем остальным, каждая сила сможет получить доступ к системе частиц. В тоже время экземпляры `Superpower` можно свободно создавать просто вызывая конструктор класса наследника.

Что еще лучше, теперь `particles_` является *статической* переменной и нам не нужно сохранять ее в каждом экземпляре `Superpower`, так что наш класс будет расходовать меньше памяти.

- **Использование поиска службы(service locator):**

  Предыдущий вариант требовал, чтобы внешний код обязательно не забывал о том, чтобы передать состояние в базовый класс, прежде чем его можно будет использовать. Таким образом на окружающий код налагаются определенные обязанности. Еще как вариант, можно позволить базовому классу обрабатывать это, получая нужное состояние самостоятельно. Для этого можно использовать шаблон Поиск службы (Service Locator).

  ```
  class Superpower
  {
  protected:
    void spawnParticles(ParticleType type, int count) {
      ParticleSystem& particles = ServiceLocator::getParticles();
      particles.spawn(type, count);
    }

    // Метод песочница и другие операции...
  };
  ```

Здесь для `` `spawnParticles() `` нам нужна система частиц. Вместо того, чтобы нам ее давали из внешнего кода, мы сами получаем ее через поиск службы.

Смотрите также

- Когда вы применяете шаблон Метод обновления (Update Method), ваш метод обновления часто будет представлять из себя и метод песочницу.
- Роль этого шаблона сходна с ролью шаблона Метод шаблон (Template Method) GoF. В обеих шаблонах вы реализуете метод с помощью набора примитивных операций. В *Методе песочнице*, метод находится в шаблоне наследнике, а операции примитивы в базовом классе. А в *Методе шаблоне*, метод содержится в *базовом* классе, а примитивы операций реализуются в классах *наследниках*.
- Также этот шаблон можно рассматривать как вариацию шаблона Фасад (Facade) GoF. Этот шаблон скрывает несколько различных систем за единым упрощенным `API`. В *Подклассе песочнице* базовый класс работает как фасад, скрывающий весь движок от подклассов.

Объект тип (Type Object)

Задача

Сделать более гибким создание новых "классов" с помощью создания класса, каждый экземпляр которого может представлять собой другой тип объекта.

Мотивация

Давайте представим себе, что мы работаем над фентезийной ролевой игрой. Нам нужно написать код для орд разнообразных монстров, которые рыщут вокруг и желают растерзать нашего героя. У монстров есть несколько разных атрибутов: здоровье, атака, графика, звуки и т.д., но в качестве примера мы будем беспокоиться только о первых двух.

У каждого монстра в игре есть значение текущего здоровья. Начинает он с полным и каждый раз, когда монстр получает ранение, оно уменьшается. Еще у монстра есть строка атаки. Когда он будет атаковать нашего героя, этот текст будет демонстрироваться пользователю. (Сейчас для нас это не важно.)

Дизайнеры сказали нам что монстры бывают разных *родов*. Например "драконы" или "тролли". Каждый род описывает *тип* монстра, существующего в игре и у нас в подземелье одновременно может быть множество монстров одного рода.

Род определяет начальное здоровье монстра: дракон начинает с большим значением, чем тролль и соответственно убить его сложнее. Также он определяет строку атаки: все монстры одного рода атакуют одинаково.

Типичное ООП решение

Получив такой игровой дизайн, мы запускаем текстовый редактор и начинаем кодить. В соответствии с дизайном, дракон — это тип монстра, а тролль — еще один тип и т.д для всех родов. Думая в объектно-ориентированном стиле мы придем к базовому классу `Monster`.

> Это так называемое публичное наследование или "is-a" отношение (один класс является подклассом другого). В рамках традиционного `ООП` мышления, так как дракон "is-a" (является) монстром, мы моделируем это отношение делая `Dragon` подклассом `Monster`. Как мы увидим позже создание подкласса — это единственный способ организации такого отношения в коде.

```
class Monster
{
public:
  virtual ~Monster() {}
  virtual const char* getAttack() = 0;

protected:
  Monster(int startingHealth)
  : health_(startingHealth)
  {}

private:
  int health_; // Текущее значение здоровья
};
```

Публичная функция `getAttack()` позволяет боевому коду получать строку, которую нужно показывать когда монстр атакует героя. Каждый класс рода будет классом наследником, переопределяющим этот класс и предоставляющим свое сообщение.

Конструктор является защищенным и устанавливает начальное значение для здоровья монстра. Для каждого рода у нас есть свой унаследованный класс, предоставляющий свой публичный конструктор, вызывающий базовый и устанавливающий соответствующее роду начальное значение здоровья.

А теперь рассмотрим несколько подклассов родов:

```
class Dragon : public Monster
{
public:
  Dragon() : Monster(230) {}

  virtual const char* getAttack() {
    return "Дракон выдыхает огонь!";
  }
};

class Troll : public Monster
{
public:
  Troll() : Monster(48) {}

  virtual const char* getAttack() {
    return "Троль ударяет дубиной!";
  }
};
```

> Восклицательный знак делает все еще более захватывающим!

Каждый унаследованный от класса `Monster` передает в базовый класс стартовое здоровье и переопределяет `getAttack()`, чтобы возвращать правильную для рода строку. Все работает как надо и довольно скоро мы сможем увидеть нашего героя убивающим всех этих монстров. Мы продолжаем писать код и прежде чем опомниться, у нас будут дюжины подклассов монстров, начиная с кислотных слизней и заканчивая зомби-козлами.

А затем мы начнем вязнуть. Нашим дизайнерам хочется иметь *сотни* родов и если продолжать в том же духе, нам придется всю жизнь прописывать эти семь строчек для нового подкласса и перекомпилировать игру. Или что еще хуже, дизайнерам захочется слегка подкорректировать уже существующие рода монстров. И теперь вся наша продуктивная работа свелась к тому что мы:

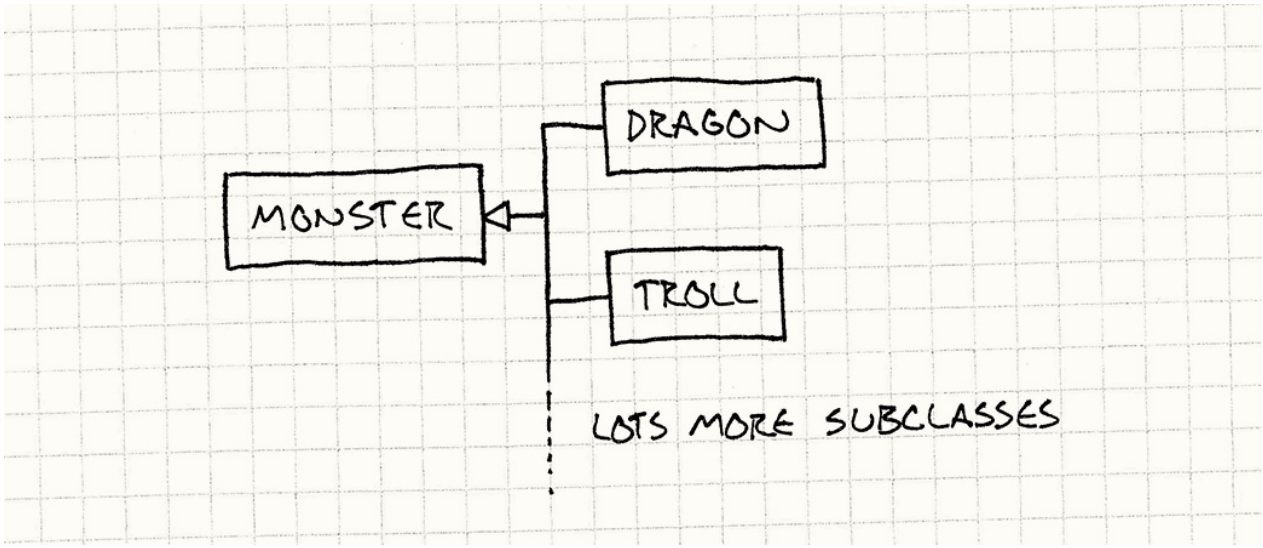
1. Получаем письмо от дизайнера, который просит изменить здоровье с `48` на `52`.
2. Ищем и изменяем `Troll.h`.
3. Перекомпилируем игру.
4. Проверяем изменение.
5. Отвечаем на письмо.
6. Повторяем.

В результате день испорчен. Мы превратились в обезьянку с данными. Наши дизайнеры расстроены тем, что для простого изменения числового значения им приходится ожидать целую вечность. Что нам нужно, так это возможность изменять характеристики рода без перекомпиляции игры каждый раз. Более того, нам бы хотелось, чтобы дизайнеры могли изменять настройки самостоятельно, *вообще* без помощи программиста.

Класс для класса

На самом высшем уровне проблема, которую мы хотим решить, чрезвычайно проста. У нас в игре есть куча монстров и мы хотим сделать часть данных общей среди них. Нашего героя атакует орда монстров и мы хотим, чтобы некоторые из них имели одинаковую строку атаки. Мы будем считать, что все эти монстры относятся к одному "роду", и этот род определяет строку атаки.

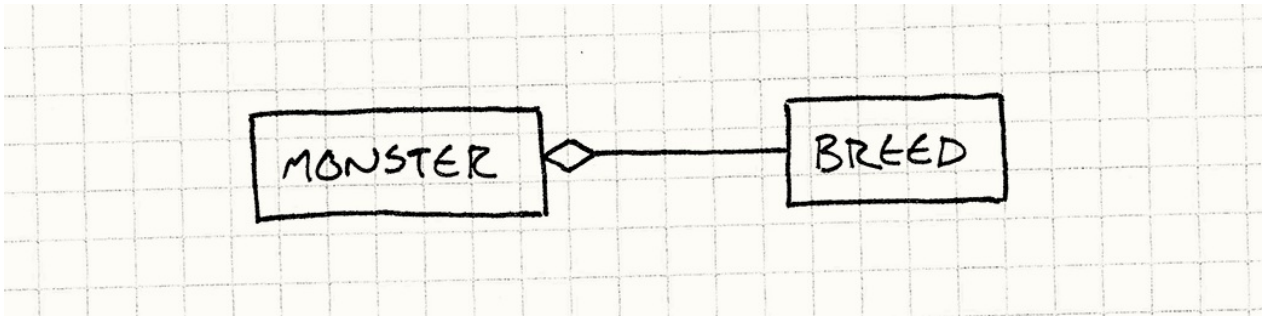
Мы решили реализовать эту концепцию с помощью наследования, так как это хорошо согласуется с нашим интуитивным пониманием ситуации: если дракон — это монстр, значит каждый дракон в игре будет экземпляром "класса" дракон. Объявим каждый род в виде подкласса абстрактного базового класса `Monster` и, как следствие, каждый монстр в игре будет экземпляром этого унаследованного класса. В результате у нас получится примерно такая иерархия:

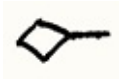


здесь  означает "наследуется от".

Каждый экземпляр монстра в игре будет представлять собой один из унаследованных типов монстров. Чем больше у вас родов, тем больше иерархия классов. Конечно это проблема: добавление новых родов означает добавление нового кода и каждый род должен компилироваться как отдельный тип.

Этот подход работает, но это только один из вариантов. Мы можем спроектировать наш код и таким образом, что каждый монстр будет *содержать* (has-a) род. Вместо того, чтобы создавать подкласс `Monster` для каждого рода, у нас будет простой класс `Monster` и единственный класс `Breed`.



Здесь  означает "имеется ссылка от"

Вот и все. Два класса. Обратите внимание, что здесь нет вообще никакого наследования. В такой системе каждый монстр в игре просто будет экземпляром класса `Monster`. Класс `Breed` содержит информацию, общую для всех монстров одного рода: начальное здоровье и строка атаки.

Чтобы ассоциировать монстра с родом, мы дадим каждому экземпляру `Monster` ссылку на объект `Breed` с информацией об этом роде. Чтобы получить строку атаки, монстр просто вызывает метод своего рода. Класс `Breed` по сути определяет "тип" монстра. Каждый экземпляр рода является *объектом*, представляющим отдельный концептуальный *тип*, как следует из имени шаблона: *Объект тип*.

Что в этом шаблоне самое замечательное, так это то, что теперь мы можем определять новые *типы* без усложнения кодовой базы: мы просто перенесли часть информации о типе из жестко закодированной иерархии классов в данные, которые можно определить во время работы программы.

Мы можем создать сотни разных видов просто создав больше экземпляров `Breed` с разными значениями. Если мы создаем экземпляр рода на основе данных, прочитанных из файла конфигурации, у нас появится возможность определять новый тип монстра полностью в данных. Это так просто, что дизайнерам точно понравится!

Шаблон

Определим класс **объект тип** и класс **типизированного объекта**. Каждый экземпляр объекта типа представляет отдельный логический тип. Каждый типизированный объект хранит **ссылку на объект тип, описывающий его тип**.

Специфичные для экземпляра данные хранятся в экземпляре типизированного объекта, а общие для всех экземпляров этого концептуального типа данные хранятся в объекте типе. Объекты, ссылающиеся на один и тот же объект тип, будут себя вести как если бы они были одного типа. Это позволит нам разделять данные и поведение между набором схожих объектов, примерно в том же духе как это делают подклассы, но без жестко закодированной структуры наследования.

Когда использовать

Этот шаблон полезен везде, где необходимо определять множество различных "видов" вещей, но жесткое их определение с помощью средств типизации вашего языка будет слишком строгим. В частности это полезно в следующих случаях:

Вы не знаете какие типы вам понадобятся. (Что например, если нашей игре понадобится поддерживать загружаемый контент, содержащий новые рода или монстров?)

Вы хотите иметь возможность изменять или добавлять новые типы без перекомпиляции или изменения кода.

Имейте в виду

Этот шаблон посвящен замене определения "типа" с императивного, но жесткого способа с помощью самого языка на более гибкий, но менее поведенческий мир объектов в памяти. Гибкость — это хорошо, но перемещая определение типа в данные, вы кое-что теряете.

За типом объекта придется следить вручную

Одно из преимуществ использования `C++` подобной системы типов в том, что компилятор занимается всей бухгалтерией классов автоматически. Данные, определяющий каждый класс, автоматически компилируются в статические сегменты памяти внутри исполнимого файла и просто работают.

Применяя шаблон *Объект тип*, мы теперь сами отвечаем не только за управление нашими монстрами в памяти, но и за их *типы*: нам нужно следить за тем, чтобы объекты рода были инициализированы и находились в памяти все время, пока они будут нужны монстрам. Когда мы создаем нового монстра, мы должны быть уверены что правильно его инициализировали и указали для него правильный род.

Мы избавились от некоторых ограничений компилятора, но взамен нам придется делать кое-что из того, что он раньше делал за нас.

> Если заглянуть под капот `C++`, виртуальные классы там реализуются с помощью структуры, называемой "таблица виртуальных функций" (virtual function table) или просто "vtable". vtable — это простая структура, содержащая набор указателей на функции, по одному на каждый виртуальный метод в классе. Для каждого класса в памяти хранится отдельная таблица. Каждый экземпляр класса обладает указателем на виртуальную таблицу своего класса.
>
> Когда вы вызываете виртуальную функцию, код сначала выполняет поиск в виртуальной таблице объекта, а потом вызывает функцию, хранящуюся в соответствующем указателе на функцию в таблице.
>
> Знакомо звучит? Виртуальная таблица — это наш объект рода, а наш указатель в виртуальной таблице — это ссылка, с помощью которой монстр ссылается на свой род. Классы `C++` — это шаблон *Объект Тип*, реализованный на `C` и автоматически обрабатываемый компилятором.

Нам будет сложнее определить *поведение* каждого типа

Используя подклассы, вы можете переопределить метод и вообще сделать все что угодно: вычислить значение процедурно, вызвать другой код и т.д. Никаких ограничений. Если у нас возникнет такое желание, мы даже можем определить подкласс монстра, строка атаки которого изменяется на основе фазы луны. (Я думаю для оборотней — это то, что нужно).

Однако когда мы используем шаблон *Объект тип*, мы заменяем переопределенный метод переменной членом. Вместо того, чтобы наш монстр был подклассом, переопределяющим метод для *вычисления* строки атаки с использованием другого *кода*, у нас будет объект рода, который хранит строку атаки в другой *переменной*.

Это значительно упрощает использование объекта типа для определения типо-специфичных *данных*, но усложняет определение типо-специфичного *поведения*. Если, например, различные рода монстров нуждаются в разных алгоритмах ИИ, использование этого шаблона усложняется.

Есть несколько способов избежать эти ограничения. Проще всего иметь фиксированный набор предопределенных вариантов поведения и дальше использовать данные в объекте типа для *выбора* одного из них. Пускай, например, ИИ монстров может быть в состояниях "стоять на месте","преследовать героя", "скулить и

прятаться в страхе" (не все же они могучие драконы). Мы можем определить функции для реализации каждой из этих линий поведения. А дальше мы можем ассоциировать алгоритм ИИ с родом, просто сохранив в нем указатель на нужную функцию.

> Снова звучит знакомо? Теперь мы вернулись к реализации виртуальной таблицы в нашем объекте типа.

Еще более мощное решение — это реализовать поддержку поведения полностью в данных. Шаблоны Интерпретатор (Interpreter) [GoF] и Байткод (Bytecode) позволят нам создавать объекты, представляющие поведение. Если мы прочитаем данные из файла и используем их для создания структуры данных из одного из этих шаблонов, мы перейдем к определению поведения за пределами кода, полностью с помощью контента.

> Со временем игры становятся все больше управляемыми за счет данных (data-driven). Железо становится все более мощным, и мы все больше сталкиваемся с ограничениями в плане количества контента, который мы можем подготовить, чем со стороны железа, которое мы нагружаем. Когда у нас был только картридж размером `64K`, сложно было просто *уместить* в него весь геймплей. А теперь когда у нас есть по крайней мере двухслойный `DVD`, сложность уже состоит в том чтобы *наполнить* этот `DVD` геймплеем.
>
> Скриптовые языки и другие высокоуровневые способы определения поведения в игре могут значительно повысить нашу производительность, но за счет разумной платы в виде снижения производительности во время работы игры. Так как улучшается только железо, а не наши мозги, такая плата становится все менее и менее чувствительной.

Пример кода

Для нашей первой реализации мы начнем с простого и построим базовую систему, описанную в разделе мотивации. Начнем с класса Breed:

```
class Breed
{
public:
  Breed(int health, const char* attack)
  : health_(health), attack_(attack)
  {}

  int getHealth() { return health_; }
  const char* getAttack() { return attack_; }

private:
  int health_; // Начальное здоровье.
  const char* attack_;
};
```

Очень просто. По сути — это контейнер для двух полей данных: начальное здоровье и строка атаки. Посмотрим, как его будут использовать монстры:

```
class Monster
{
public:
  Monster(Breed& breed)
  : health_(breed.getHealth())
  , breed_(breed)
  {}

  const char* getAttack() {
    return breed_.getAttack();
  }

private:
  int health_; // Текущее здоровье.
  Breed& breed_;
};
```

Когда мы конструируем монстра, мы передаем ему ссылку на объект рода. Таким образом мы указываем род монстра вместо того чтобы делать его подклассом, как раньше. Внутри конструктора `Monster` использует род для определения начального здоровья. Чтобы получить строку атаки, монстр просто обращается к своему роду.

В этом простом кусочке кода и заключена вся суть шаблона. Все остальное всего лишь бонус.

Уподобление объекта тип настоящим типам: конструкторы

С помощью того, что у нас сейчас есть, мы конструируем монстра напрямую, и сами отвечаем за то, чтобы передать ему род. Такой метод значительно хуже, чем обычное создание экземпляров объектов в `ООП` языках: обычно мы не выделяем сначала область памяти, а затем *даем* ей класс. Вместо этого мы вызываем функцию конструктор самого класса и она отвечает за то, чтобы создать для нас новый экземпляр.

Этот же шаблон можно применить и к объекту типу:

```
class Breed
{
public:
  Monster* newMonster() { return new Monster(*this); }

  // Предыдущий код Bread...
};
```

> "Шаблон" здесь вполне правильно слово. Потому что мы говорим не много ни мало, а о шаблоне программирования: Фабричный метод (Factory Method) GoF.

В некоторых языках этот шаблон применяется для создания *всех* объектов. В `Ruby`, `Smalltalk`, `Objective-C` и некоторых других языках, где классы являются объектами, вы создаете новые экземпляры, вызывая метод самого объекта класса.

А теперь класс, который его использует:

```
class Monster
{
friend class Breed;

public:
  const char* getAttack() {
    return breed_.getAttack();
  }

private:
  Monster(Breed& breed)
  : health_(breed.getHealth())
  , breed_(breed)
  {}

  int health_; // Текущее здоровье.
  Breed& breed_;
};
```

Основное отличие — это новая функция *newMonster()* в *Breed*. Это наш фабричный метод "конструктор". В нашей первой реализации создание монстра выглядело следующим образом:

> Вот и еще одно небольшое различие. Так как пример кода написан на `C++`, мы можем использовать очень полезную возможность: дружественные классы.
>
> Мы сделали конструктор `Monster` приватным, чтобы быть уверенными в том, что его никто не вызовет напрямую. Дружественные классы обходят это ограничение и *Breed* все равно получает к нему доступ. Это значит, что единственный способ создать монстра — это использовать `newMonster()`.

```
Monster* monster = new Monster(someBreed);
```

А после изменений оно будет выглядеть так:

```
Monster* monster = someBreed.newMonster();
```

Для чего это нужно? Создание объекта состоит из двух шагов: выделения памяти и инициализации. Конструктор `Monster` позволяет нам выполнять всю необходимую инициализацию. В нашем примере это просто сохранение рода, но в настоящем игровом мире мы загружали бы графику, инициализировали ИИ монстров и делали другую работу по настройке.

Однако все это происходит *после* выделения памяти. У нас уже есть участок памяти для хранения нашего монстра еще до того, как вызывается конструктор. В игре мы обычно контролируем этот аспект создания объекта: мы обычно используем вещи типа нестандартных выделителей памяти или шаблон Пул объектов (Object Pool) для управления тем, где будут храниться объекты.

Определяя функцию "конструктор" в `Breed` у нас появляется место, где можно разместить такую логику. Вместо простого вызова `new`, функция newMonster может взять память из пула или нестандартной кучи перед тем как передать управление ``Monster `для инициализации. Помещая эту логику внутрь` Breed``` — единственную функцию, которая имеет возможность создавать монстров, мы можем быть уверены, что все монстры буду создаваться с помощью нужной нам схемы управления памятью.

Разделяемые между экземплярами данные

Все, чего мы на данный момент добились — это довольно полезная, но все-таки довольно базовая система типов объектов. В конце концов, у нас могут появиться сотни родов, каждый с дюжинами атрибутов. Если дизайнер захочет настроить все

тридцать видов троллей, чтобы они стали немного сильнее, ему придется весьма долго заниматься утомительным вводом данных.

Что здесь может помочь, так это возможность разделять атрибуты между множеством родов, также как роды позволяют нам разделять атрибуты между множеством монстров. Также как мы применяли наше `ООП` решение в первый раз, мы можем решить эту задачу наследованием. Только на этот раз вместо использования механизма наследования, предоставляемого языком, мы реализуем его самостоятельно с помощью объекта типа.

Чтобы ничего не усложнять, мы будем поддерживать единичное наследование. Точно также как наш класс может иметь родительский базовый класс, мы можем позволить роду иметь родительский род:

```
class Breed
{
public:
  Breed(Breed* parent, int health, const char* attack)
  : parent_(parent)
  , health_(health)
  , attack_(attack)
  {}

  int getHealth();
  const char* getAttack();

private:
  Breed* parent_;
  int health_; // Начальное здоровье.
  const char* attack_;
};
```

Когда мы конструируем род, мы даем ему родителя, от которого он наследуется. Роду, который не имеет предков мы передаем `NULL`.

Чтобы все это имело смысл, дочерний род должен управлять тем, какие атрибуты наследуются от родителя, а какие переопределяются и указываются им самим. В системе из нашего примера мы указываем, что род переопределяет здоровье монстра ненулевым значением и переопределяет атаку ненулевой строкой. Другими словами атрибуты наследуются от родителя.

Существует два способа это сделать. Один заключается в том, чтобы обрабатывать делегирование динамически, каждый раз при запрашивании атрибута следующим образом:

```
int Breed::getHealth()
{
  // Переопределение.
  if (health_ != 0 || parent_ == NULL)
    return health_;

  // Наследвоание.
  return parent_->getHealth();
}

const char* Breed::getAttack()
{
  // Переопределение.
  if (attack_ != NULL || parent_ == NULL)
    return attack_;

  // Наследование.
  return parent_->getAttack();
}
```

У такого способа есть преимущество в том, что он будет работать даже в том случае, если во время работы изменить род и он больше не будет переопределять все или какое-то один атрибут. С другой стороны, нам потребуется немного больше памяти (чтобы хранить указатели на родителей) и мы немного теряем в скорости. Нам придется обходить всю цепочку наследования каждый раз, когда мы запрашиваем атрибут.

Если мы можем полагаться на то что атрибуты рода меняться не будут, можно выбрать более быстрый вариант и применять наследование во *время создания* (construction time). Такое делегирование называется "копированием вниз", потому что мы *копируем* унаследованные атрибуты *вниз* в производный тип в момент его создания. Это выглядит следующим образом:

```
Breed(Breed* parent, int health, const char* attack)
  : health_(health)
  , attack_(attack)
{
  // Наследование непереопределенных атрибутов.
  if (parent != NULL) {
    if (health == 0)
      health_ = parent->getHealth();
    if (attack == NULL)
      attack_ = parent->getAttack();
  }
}
```

Обратите внимание, что теперь нам не нужно поле для родительского рода. Как только конструктор выполнился, мы можем забыть о родителе, потому что уже скопировали из него нужные нам атрибуты. Чтобы получить доступ к атрибутам рода мы просто возвращаем значение поля:

```
int getHealth() { return health_; }
const char* getAttack() { return attack_; }
```

Просто и быстро!

А теперь предположим, что наш движок выполняет настройку родов, загружая их из `JSON` файла, в котором они определены. Вот как он выглядит:

```
{
  "Troll": {
    "health": 25,
    "attack": "The troll hits you!"
  },
  "Troll Archer": {
    "parent": "Troll",
    "health": 0,
    "attack": "The troll archer fires an arrow!"
  },
  "Troll Wizard": {
    "parent": "Troll",
    "health": 0,
    "attack": "The troll wizard casts a spell on you!"
  }
}
```

У нас будет код, читающий каждую запись о роде и создающий новые экземпляры рода для каждой из них. Как вы можете видеть из значения поля `"parent"`, равного `"Troll"`, рода `Troll Archer` и `Troll Wizard` наследуются от базового рода `Troll`.

Так как у них обеих в поле здоровья стоит ноль, они наследуют его от базового рода `Troll`. Это значит, что теперь дизайнеры могут настраивать здоровье в `Troll`, а все остальные рода тоже будут обновляться. По мере того, как будет увеличиваться количество родов и количество атрибутов в каждом роде, такой подход позволит нам сэкономить много времени. И теперь, в лице небольшого фрагмента кода, у нас есть открытая система, дающая достаточную свободу нашим дизайнерам и сохраняющая им время. А мы тем временем возвращаемся к кодированию другого функционала.

Архитектурные решения

Шаблон Объект тип позволяет создавать систему типов, подобно тому, как если бы мы писали собственный язык программирования. Простор для архитектурных решений достаточно широк и мы можем делать очень много всяких интересных вещей.

На практике вашего внимания заслуживают всего несколько вещей. Время и удобство поддержки ограничивают нас от слишком сложных вещей. Самое главное при разработке объекта типа, чтобы наши пользователи (обычно это не программисты) могли легко понять как с ним работать. Чем проще он будет, тем он для нас полезнее. Поэтому мы будем рассматривать только хорошо проторенные дорожки, а все остальное оставим академикам и исследователям.

Наш Объект тип будет инкапсулирован или открыт?

В нашем примере реализации `Monster` ссылался на род, но публично он не виден. За пределами кода мы не можем напрямую получить род монстра. Для всей остальной кодовой базы, монстры вообще не имеют типа и то что они относятся к определенному роду — не более чем детали реализации.

Мы легко можем это изменить, позволив монстрам возвращать свой род (`Bread`):

> Как и в других примерах в книге, мы следуем соглашению, когда объект возвращается по ссылке, а не в виде указателя, чтобы дать понять пользователю что NULL никогда возвращаться не будет.

```
class Monster
{
public:
  Breed& getBreed() { return breed_; }

  // Существующий код...
};
```

Таким образом, мы изменяем дизайн `Monster` . Тот факт, что монстры имеют род, теперь является публично видимой частью `API` . Оба решения имеют свои преимущества.

- **Если объект тип инкапсулирован:**
    - *Сложность шаблона Объект тип скрыта от остальной кодовой базы.* Она становится деталями реализации, за которую отвечает только сам объект тип.

- *Типизированный объект может частично переопределять поведение из объекта типа.* Предположим, что мы захотели изменить строку атаки для монстра, когда он находится на грани смерти. Так как строку атаки мы всегда получаем через Monster, вполне логично, что мы разместим код именно там:

  ```
  const char* Monster::getAttack()
  {
  if (health_ < LOW_HEALTH) {
    return "The monster flails weakly.";
  }

  return breed_.getAttack();
  }
  ```

  Если бы внешний код вызывал `getAttack()` напрямую из рода, у нас не было бы возможности реализовать такую логику.

- *Нам нужно писать метод перенаправления для всего в чем участвует объект тип.* Это самая утомительная черта такой архитектуры. Если в нашем объекте типе много функций, класс объект должен будет заводить собственные методы для всех из них, кого мы захотим сделать публично видимыми.

- **Если объект тип является видимым:**

  - *Внешний код может взаимодействовать с объектом типом без помощи экземпляра типизированного класса.* Если объект тип инкапсулирован, мы никак не можем использовать его без типизированного объекта, являющегося для него оболочкой. Например, это не дает использовать наш шаблон конструктор, когда новый монстр создается с помощью вызова метода рода. Если пользователи не могут обратиться к роду напрямую, они не смогут его вызвать.

  - *Объект тип теперь является частью публичного `API` объекта.* В целом, узкий интерфейс легче поддерживать, чем широкий: чем меньше вы показываете остальной части кодовой базы, тем меньше сложность и проще поддержка. Делая объект тип видимым, мы расширяем `API` объекта и включаем в него все возможности объекта типа.

Как создается типизированный объект?

Когда мы применяем этот шаблон, каждый "объект" у нас представляется двумя объектами: главным объектом и объектом типом, который он использует. Каким же образом мы можем их создать и связать вместе?

- **Конструируем объект и передаем в него объект тип:**
    - *Выделение памяти контролирует внешний код.* Так как вызывающий код самостоятельно конструирует оба объекта, он может управлять и тем, где в памяти они будут находиться. Если мы хотим, чтобы наши объекты можно было использовать в разных сценариях работы с памятью (различные типы выделения памяти, на стеке и т.д.), в этом случае мы получаем нужную нам гибкость.
- **Вызов "конструирующей" функции объекта типа:**
    - *Объект тип управляет выделением памяти.* Это обратная сторона медали. Если мы не хотим, чтобы пользователь выбирал, где в памяти создавать наши объекты, а хотим управлять этим процессом сами, мы заставляем его пользоваться нашим фабричным методом объекта типа. Это может быть полезным, если мы хотим чтобы все объекты находились в одном Пуле объектов (Object Pool) или хотим еще использовать какие-либо другие способы выделения памяти.

Можно ли менять тип?

До сих пор мы предполагали, что как только объект создается и связывается с объектом типом, эта связь уже никогда не меняется. Объект как создается с одним типом, так с ним и умирает. Но это совсем не обязательно. Мы вполне можем позволить объектам менять свой тип.

Снова вернемся к нашему примеру. Когда монстр погибает, дизайнер может попросить нас о том, чтобы труп превратился в ожившего зомби. Мы можем реализовать это, создавая нового монстра рода зомби, когда монстр погибает. Но есть и другой вариант. Мы можем просто взять нашего монстра и изменить его род на род зомби.

- **Если тип не меняется:**
    - *Такой вариант проще как для программирования, так и для понимания.* На концептуальном уровне, "тип" — это нечто что большинство людей воспринимают как нечто неизменное. И мы закрепляем такое предположение в коде.
    - *Его проще отлаживать.* Если мы попробуем отследить баг, при котором монстр оказывается в каком-то странном состоянии, нам будет гораздо проще разобраться в проблеме, если монстр *сейчас* относится к тому типу, к которому относился всегда.
- **Если тип можно менять:**

- *Мы будем реже создавать объекты.* В нашем примере, если тип не меняется, нам нужно будет потратить циклы процессора на создание нового зомби, копирование в него атрибутов из оригинального монстра, которого он будет представлять и затем удалять его. Если мы теперь сможем просто изменить тип, вся работа сведется к простому переназначению.
- *Нам нужно быть осторожнее с предположениями.* У нас имеется довольно сильная связь между объектом и его типом. Например, род может предполагать что *текущее* здоровье монстра никогда не может превышать начальное здоровье, соответствующее его роду.

  Если мы позволим менять род, нам нужно будет проверять, чтобы новый тип удовлетворял требованиям, предъявляемым к существующему объекту. Когда мы изменяем тип, нам скорее всего придется добавить какой-то код валидации, который будет проверять чтобы объект находился в том состоянии, в котором имеет смысл менять его тип.

Какой тип наследования у нас поддерживается?

- **Никакого наследования:**
  - *Это просто.* Зачастую простота — самый лучший выбор. Если у вас нет кучи данных, которые необходимо разделять между объектами типа, зачем самостоятельно усложнять себе жизнь?
  - *Может привести к дублированию работы.* Стоит еще поискать такую систему генерации контента, в которой дизайнеры *хотели бы* отказаться от наследования. Когда у вас есть пятьдесят различных типов эльфов, настраивать их здоровье, немного изменяя одно и то же значение в пятидесяти разных местах — это *отстой*.
- **Простое наследование:**
  - *Все еще достаточно просто.* Его легко реализовать и, что более важно, легко понять. Если с системой будут работать нетехнические специалисты, чем меньше в ней будет движущихся частей, тем лучше. Не зря большое количество языков программирования поддерживают только единичное наследование. Это вполне удачный компромисс между мощью и простотой.
  - *Поиск атрибутов работает медленнее.* Чтобы получить из объекта типа нужные данные, нам нужно обойти всю цепочку наследования, чтобы найти тот тип, из которого значение нужно брать. Если такое происходит в коде, для которого критична производительность, мы не можем позволить себе тратить на это время.

- **Множественное наследование:**
  - *Мы можем избежать практически любого дублирования данных.* С помощью хорошей системы множественного наследования, пользователи могут построить систему наследования, в которой практически не будет избыточности. И когда придет время настройки чисел, большей части копипаста можно будет избежать.
  - *Это сложно.* К сожалению, выигрыш является более теоретическим, чем практическим. Множественное наследование сложнее понимать и осмысленно использовать.

    В нашем примере тип Зомби Дракон, наследуется и от Зомби и от Дракона, какие атрибуты будут браться из Зомби и какие из Дракона? Для того, чтобы применять эту систему, пользователям нужно понимать работу графа наследования и способность предвидеть какая иерархия будет удачной.

    Большая часть стандартов кодирования на `C++`, которые я сейчас вижу, запрещает использовать множественное наследование, а в `C#` и `Java` оно отсутствует полностью. Так что стоит это признать: оно настолько сложное, что лучше не использовать его вообще. Если вы хорошо об этом подумаете, то поймете что у нас редко возникает реальная необходимость использовать множественное наследование для объектов типов в игре. Так что чем проще, тем лучше.

Смотрите также

- На самом высоком уровне этот шаблон отвечает за разделение данных и поведения между несколькими объектами. Еще один шаблон — Прототип (Prototype) GoF, посвящен решению той же проблемы, но несколько по другому.
- Объект тип — близкий родственник Приспособленца (Flyweight) GoF. Оба позволяют вам разделять общие данные между экземплярами. В случае с *Приспособленцем*, наследование экономит память, а разделяемые данные не обязательно должны концептуально представлять из себя "тип" объекта. В случае с *Объектом типом*, усилия концентрируются на организации и гибкости.
- Есть много общего у этого шаблона и с шаблоном Состояние (State) GoF. Оба шаблона позволяют объекту делегировать часть себя к другому объекту. В случае с объектом типом, мы обычно делегируем, что объект из себя представляет:

инвариантным данным, описывающим объект в широком смысле. В случае *Состояния*, мы делегируем то, чем является объект в данный момент: временные данные, описывающие текущую конфигурацию объекта.

Когда мы обсуждали возможность объекта менять собственный тип, вы можете думать об этом как о частичном дублировании *Объектом типом* функциональности шаблона *Состояния*.

Шаблоны уменьшения связности

Как только вы начинаете разбираться в языке программирования, написание кода, который вам нужен становится достаточно простым. Гораздо сложнее писать код, который будет легко изменять в будущем. Очень редко бывает так, что мы можем предполагать, что произойдет в будущем, когда запускаем наш редактор.

У нас есть мощный инструмент, упрощения изменений — *снижение связности* (decoupling). Когда мы говорим два участка кода "слабо связаны (decoupled)", мы имеем в виду, что изменение одного обычно не требует изменения другого. Когда вам нужно добавить новый функционал в игре, чем меньше частей кода вам придется затронуть — тем лучше.

Шаблон Компонент(Component) снижает связность различных областей вашей игры друг от друга с помощью единой сущности, обладающей всеми их аспектами. Очередь событий (Event Queue) снижает связность двух общающихся друг с другом объектов, как статически, так и *во время работы* (in time). Шаблон Поиск службы (Service Locator) позволяет коду обращаться к объекту, не привязываясь к коду, который его предоставляет.

Шаблоны

- Компонент(Component)
- Очередь событий (Event Queue)
- Поиск службы (Service Locator)

Компонент(Component)

Задача

Позволяет одной сущности охватывать несколько областей, не связывая их между собой.

Мотивация

Предположим, мы создаем платформер. Биография итальянских водопроводчиков и так всем хорошо известна, так что мы возьмем Датского пекаря Бьйорна (Bjørn). Разумеется у нас будет класс, представляющий нашего дружелюбного кондитера и он будет содержать все, что персонаж делает в игре.

> Вот из-за вот таких гениальных идей я программист, а не дизайнер.

Так как Бьерном управляет игрок, это значит, что нужно считывать пользовательский ввод и переводить его в движение. И, конечно, ему нужно взаимодействовать с уровнем, т.е. происходит обработка физики и коллизий. Когда мы закончим, персонажу нужно появиться на экране, что предполагает анимацию и рендеринг. Еще потребуется проигрывание звуков.

Остановимся на минутку и подумаем. Программная архитектура 101 говорит нам, что различные области следует сохранять изолированными друг от друга. Если мы создаем текстовый процессор, то код обрабатывающий печать не должен зависеть от кода, загружающего или сохраняющего документ. В игре конечно нет такого деления, как в бизнес приложении, но общий принцип остается неизменным.

До тех пор пока это возможно, мы не хотим чтобы ИИ, физика, рендер, звук и остальные области не знали друг о друге, но пока что все это перемешано внутри одного класса. Мы знаем куда ведет эта дорога: свалка внутри исходника в 5000 строк, настолько большая, что только самый отважный ниндзя-кодер в вашей команде отважится туда отправиться.

Это отличная гарантия занятости для тех, кто решится этим заняться, и настоящий ад для всех остальных. Такой огромный размер файла свидетельствует о том, что даже сравнительно тривиальное изменение может иметь далеко идущие последствия. И совсем скоро класс начнет быстрее плодить *баги*, чем мы будем успевать пополнять его *функциональностью*.

Гордиев узел

Гораздо хуже проблемы масштабирования проблема увеличения связности. Все отдельные системы нашей игры связаны в гордиев узел кода в духе:

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
  playSound(HIT_FLOOR);
}
```

> Несмотря на то, что подобная связность плоха в *любой* игре, в современных играх, использующих конкурентный режим работы она еще страшнее. Для нашего многоядерного железа жизненно важно чтобы код использовал несколько потоков одновременно. Одним из вариантов разделения выполнения игры по потокам, является разделение по областям: ИИ запускается на одном ядре, звук на другом, рендеринг на третьем и т.д.
>
> Как только вы внедрите у себя подобную систему, отсутствие связности между областями станет для вас критичным, иначе вы рискуете получить блокировки и другие баги конкурентного выполнения. А если у вас будет класс, метод `UpdateSounds()` которого вызывается из одного потока, а метод `RenderGraphics()` из другого, вы очень рискуете столкнуться с подобными багами.

Каждый программист, который попытается изменить подобный код, должен хотя бы немного разбираться в физике, графике и звуке для того чтобы ничего не сломать.

Эти две проблемы дополняют друг друга: класс затрагивает так много областей, что каждый из программистов вынужден с ним работать, но настолько велик, что работать с ним просто ужасно. Если он достаточно плох, кодеры начнут добавлять всякие хаки в другие части кода, лишь бы не связываться с ужасным классом, в который превратился наш `Bjorn`.

Разрубание узла

Проблему можно решить, последовав примеру Александра Великого: с помощью меча. Мы берем наш монолитный класс `Bjorn` и разрезаем его на отдельные части по границам областей. Например, берем весь код обработки пользовательского ввода и переносим его в отдельный класс `InputComponent`. Теперь `Bjorn` будет содержать экземпляр этого компонента. Повторяем процесс для каждой области, которой касается `Bjorn`.

Когда мы закончим, в самом `Bjorn` практически ничего не останется. Все, что останется в его тонкой скорлупке — так это связи между компонентами. Мы решили нашу большую проблему, просто поделив ее на множество мелких классов. Но на самом деле мы добились даже большего.

Свободные концы

Наши классы компонентов теперь мало связаны (decoupled). Несмотря на то, что `Bjorn` содержит `PhysicsComponent` и `GraphicsComponent`, эти двое друг о друге не знают. Это значит, что тот, кто работает над физикой, может изменять относящийся к ней компонент без необходимости трогать графику и наоборот.

На практике компонентам приходится *как-либо* друг с другом взаимодействовать. Например, компоненту ИИ может потребоваться сообщить физическому компоненту куда собирается идти Бьйорн. Однако мы можем ограничить это взаимодействие до того, что *должен* сообщать компонент, а не предоставлять им свободно между собой общаться.

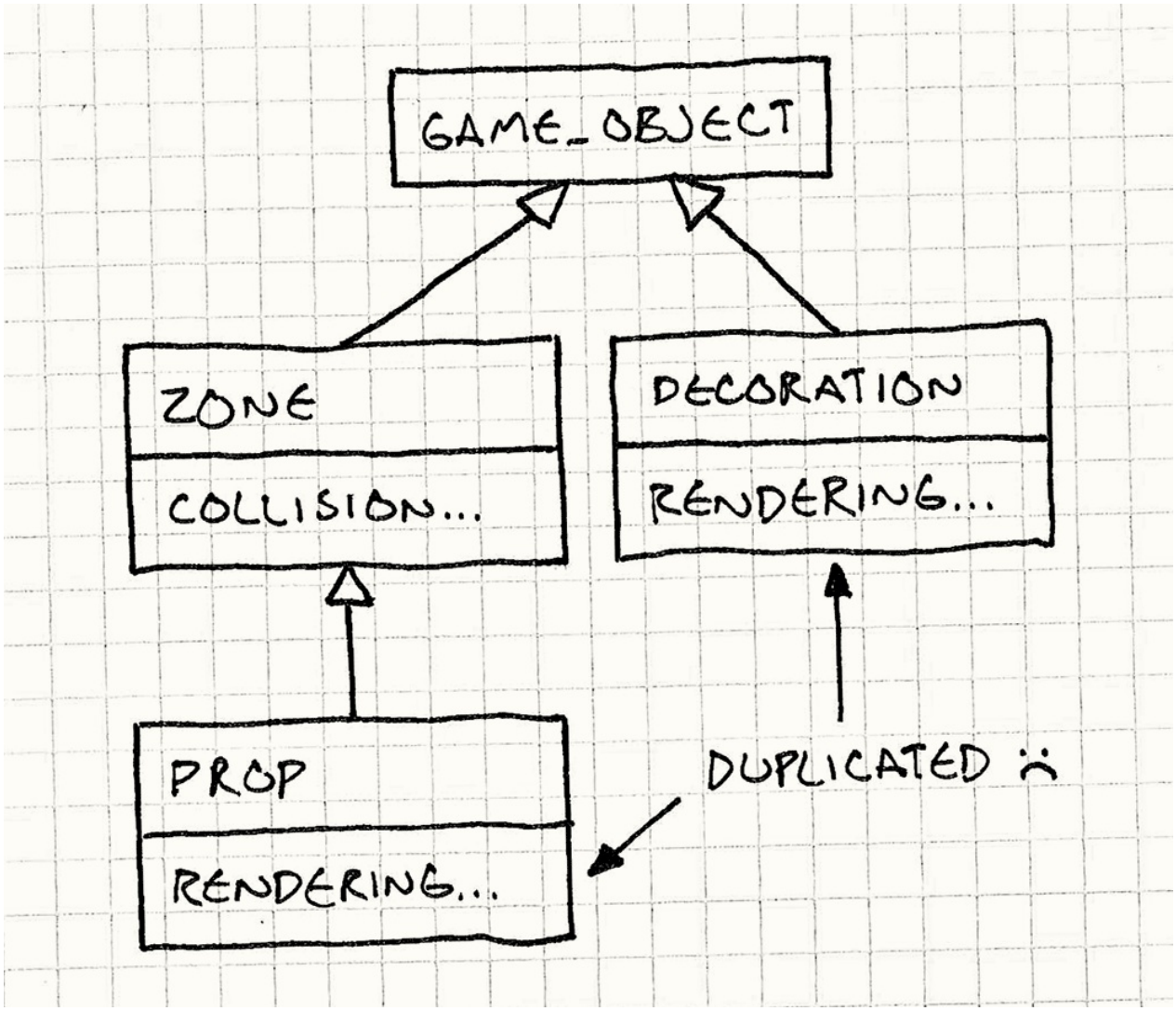
Собираем все вместе

Еще одна возможность данного дизайна заключается в том, что компоненты теперь представляют собой пакеты, пригодные для повторного использования. До сих пор мы фокусировались на нашем пекаре, но давайте рассмотрим еще несколько объектов игрового мира. *Декорации* — это то, каким игрок видит мир, но с чем не взаимодействует: кусты, непролазные дебри и другие видимые детали. *Реквизит* похож на декорации, но мы можем его потрогать: ящики, булыжники и деревья. *Зоны*, в отличие от декораций невидимы, но интерактивны. Они полезны для вещей типа запуска заставок, когда Бьйорн в них входит.

> Когда объектно-ориентированное программирование появилось на сцене впервые, наследование было самым любимым из всех его инструментов. Оно было объявлено ультимативным молотом повторного использования кода и кодеры постоянно им размахивали. С тех пор мы на собственных ошибках убедились в том, что этот молот может быть слишком тяжел. Наследование имеет свое применение, но для повторного использования кода оно обычно слишком громоздко.

Ему на замену в программирование пришел новый тренд: композиция взамен наследования везде, где это возможно. Вместо совместного использования кода двумя классами, которые *наследуются* от какого-то одного класса, мы позволяем им обеим *обладать одним и тем же экземпляром* этого класса.

Теперь подумаем, как мы настроили бы иерархию всех этих классов, если бы не использовали компоненты. Первая попытка может выглядеть таким образом:



У нас есть класс `GameObject`, содержащий такие понятные вещи как позиция и ориентация. `Zone` наследуется от него и добавляет обнаружение коллизий. Аналогично, `Decoration` наследуется от `GameObject` и добавляет рендеринг. `Prop` наследуется от `Zone` и может повторно использовать код обнаружения коллизий `Collizion` оттуда. Однако `Prop` не может *в то же самое время* наследоваться от `Decoration`, чтобы использовать код рендеринга оттуда без превращения в Смертельный бриллиант.

> "Смертельный бриллиант (Deadly Diamond)" случается в иерархии классов при множественном наследовании, когда у нас есть два пути, ведущие к базовому классу. Сама проблема выходит за рамки рассмотрения нашей книги, но думаю вы понимаете, что просто так вещи "смертельными" не называют.

Мы можем сделать по другому, и `Prop` будет наследоваться от `Decoration`, но тогда у нас получится дублирование кода *определения коллизий*. В любом случае, не существует простого способа для повторного использования кода определения

коллизий или рендеринга без использования множественного наследования. Единственная альтернатива — поместить все снова в `GameObject`, но тогда `Zone` будет тратить память на данные для рендеринга, которые ей не нужны, а `Decoration` будет делать тоже самое с данными для физики.

А теперь попробуем повторить тоже самое с компонентами. Наши подклассы полностью исчезли. Вместо этого у нас есть один класс `GameObject` и два класса компонента: `PhysicsComponent` и `GraphicsComponent`. Декорация — это просто `GameObject` с `GraphicsComponent`, но без `PhysicsComponent`. Для зоны все наоборот, а реквизит использует оба компонента. Никакого дублирования кода, никакого множественного наследования и только три класса взамен четырех.

> Хорошая аналогия — это меню ресторана. Если каждый элемент меню — это монолитный класс, то можно представить себе что мы можем заказать только комбо. Для этого нам нужно иметь отдельный класс для каждой возможной комбинации. Чтобы удовлетворить запрос каждого посетителя, нам понадобятся дюжины комбинаций.
>
> Компоненты — это блюда на выбор (à la carte): каждый посетитель может выбрать только те блюда, которые хочет, а меню — это всего лишь список блюд, из которых можно выбирать.

Компоненты — это практически механизм plug-and-play для объектов. Они позволяют нам конструировать сложные сущности с богатым поведением, просто подключая переназначенные для повторного использования объекты компоненты в сокеты сущности. Такой себе программный Вольтрон.

Шаблон

Единая сущность охватывает множество областей. Для сохранения изолированности областей, код для каждой помещается в свой собственный **класс компонент**. Сущность упрощается до простого **контейнера компонентов**.

> "Компонент", как и "Объект" — это одни из слов в программировании, означающее сразу все и ничего. Потому что они использовались для описания нескольких концепций. В бизнес приложениях "Компонент" — это шаблон проектирования, описывающий слабо связанные сервисы, общающиеся по сети.
>
> Я пытался найти для этого отдельного шаблона, применяемого в играх, отдельный термин, но "Компонент" оказался наиболее подходящим. Так как шаблоны проектирования — это документирование существующих практик, у меня нет привилегии выдумывать новые термины. Так, что вслед за `XNA`, `Delta3D` и остальными, будем использовать название "Компонент".

Когда использовать

Компоненты чаще всего можно найти внутри класса-ядра, описывающего сущности в игре, но использовать их можно и в других частях игры. Этот шаблон стоит использовать, если верно что-либо из нижеперечисленного:

- У вас есть класс, затрагивающий множество областей, которые вы хотите оставить несвязанными друг с другом.
- Класс становится слишком массивным и сложным для работы.
- Вы хотите определить множество объектов, разделяющих различные возможности, но при этом не можете использовать наследование, потому что оно не дает вам достаточно свободно подбирать части, которые вы хотите использовать.

Имейте в виду

Этот шаблон добавляет сложности в простой процесс создания класса и наполнения его кодом. Каждый концептуальный "объект" становится кластером объектов, который нужно инстанциировать, инициализировать и корректно увязать вместе. Коммуникация между разными компонентами усложняется и размещение в памяти тоже усложняется.

Для большой кодовой базы ее сложность может оправдывать снижение связности и добавление возможности повторного использования кода. Но будьте осторожны и прежде чем применить шаблон убедитесь, что не пытаетесь применить "решение" к несуществующей проблеме.

Еще компоненты могут пригодиться, когда вам часто приходится прыгать сразу через несколько уровней косвенности, чтобы сделать то, что нужно. Имея объект контейнер, вы сначала выбираете нужный компонент, а *затем* делаете то что вам нужно. Во внутренних циклах, где производительность критичная, такой переход по указателю может плохо отразиться на производительности.

Но есть и обратная сторона. Шаблон *Компонент* зачастую *улучшает* производительность и связность кеша. Компоненты упрощают использование шаблона Локализация данных (Data Locality), помогающего размещать данные так, как удобно процессору.

Пример кода

Одним из самых больших вызовов при написании этой книги для меня стала задача изоляции каждого шаблона. Многие шаблоны проектирования обычно содержат в себе код, который не является частью шаблона. Чтобы очистить шаблон до самой его сути я попытался убрать как можно больше всего лишнего, но на каком-то этапе это будет напоминать попытку объяснить работу двигателя внутреннего сгорания не упоминая топлива или масла.

Шаблон *Компонент* пожалуй в этом смысле самый сложный из всех. Вы не можете оценить его по настоящему, если не будете видеть код каждой из областей, связность между которыми он снижает. Поэтому мне придется привести немного больше кода Бьйорна, чем хотелось бы. Сам шаблон представляет из себя всего лишь *классы* компонентов, но их код поможет понять, для чего эти классы предназначены. Это не настоящий код и обращается он к другим классам, которые здесь не представлены. Но таким образом вы хотя бы получите представление о том что происходит.

Монолитный класс

Чтобы увидеть общую картинку применения этого шаблона, начнем с рассмотрения монолитного класса `Bjorn`, который делает все что нам нужно, но не использует наш шаблон:

> Хочу отметить, что использование имени персонажа в коде обычно плохая идея. У маркетологов есть плохая привычка менять названия незадолго до выпуска продукта. "Фокус группы показали что от 11 до 15 процентов тестеров негативно оценивают имя Бьйорн. Поэтому будем использовать имя Стив".
>
> Вот поэтому многие проекты используют внутри только кодовые имена. Это между прочим еще и веселее. Приятнее ведь говорить людям, что вы работаете над "Большим электрокотом", а не просто над "очередной версией Photoshop".

```
class Bjorn
{
public:
  Bjorn()
  : velocity_(0), x_(0), y_(0)
  {}

  void update(World& world, Graphics& graphics);

private:
  static const int WALK_ACCELERATION = 1;

  int velocity_;
  int x_, y_;

  Volume volume_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

У `Bjorn` есть метод `update()`, который вызывается на каждом кадре игре:

```
void update(World& world, Graphics& graphics)
{
  // Применяем пользовательский ввод к скорости героя.
  switch (Controller::getJoystickDirection())
  {
  case DIR_LEFT:
    velocity_ -= WALK_ACCELERATION;
    break;

  case DIR_RIGHT:
    velocity_ += WALK_ACCELERATION;
    break;
  }

  // Изменение позиции на скорость.
  x_ += velocity_;
  world.resolveCollision(volume_, x_, y_, velocity_);

  // Отрисовка соответствующего спрайта.
  Sprite* sprite = &spriteStand_;
  if (velocity_ < 0) {
    sprite = &spriteWalkLeft_;
  } else if (velocity_ > 0) {
    sprite = &spriteWalkRight_;
  }

  graphics.draw(*sprite, x_, y_);
}
```

Он считывает данные от джойстика и применяет ускорение или торможение. Далее определяется новая позиция с помощью физического движка. И наконец `Bjorn`, отрисовывается на экране.

Реализация в примере предельно проста. У нас нет гравитации, анимации и дюжин всяких других деталей, делающих игру персонажем приятной. И даже сейчас мы уже видим, что в одной единственной функции используется работа нескольких программистов из нашей команды и выглядит это уже довольно запутано. Представьте, что вместо этого у нас есть тысяча строк кода и поймете насколько это усложняет работу.

Разделение на области

Начнем с одной из областей. Возьмем часть `Bjorn` и отправим в отдельный компонент. Начнем с первой области, с которой мы работаем: с пользовательского ввода. Первое, что делает `Bjorn` — это считывает пользовательский ввод и изменяет соответствующим образом скорость. Давайте поместим эту логику в отдельный класс:

```
class InputComponent
{
public:
  void update(Bjorn& bjorn)
  {
    switch (Controller::getJoystickDirection())
    {
    case DIR_LEFT:
      bjorn.velocity -= WALK_ACCELERATION;
      break;

    case DIR_RIGHT:
      bjorn.velocity += WALK_ACCELERATION;
      break;
   }
}

private:
   static const int WALK_ACCELERATION = 1;
};
```

Довольно просто. Мы взяли первую часть метода `update()` из `Bjorn` и поместили в новый класс. Изменения в `Bjorn` тоже будут вполне очевидными:

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);

    // Изменение позиции в зависимости от скорости.
    x += velocity;
    world.resolveCollision(volume_, x, y, velocity);

    // Draw the appropriate sprite.
    Sprite* sprite = &spriteStand_;
    if (velocity < 0) {
      sprite = &spriteWalkLeft_;
    } else if (velocity > 0) {
      sprite = &spriteWalkRight_;
    }

    graphics.draw(*sprite, x, y);
  }

private:
  InputComponent input_;

  Volume volume_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

Теперь у `Bjorn` есть объект `InputComponent`. Также, как раньше мы обрабатывали пользовательский ввод напрямую из метода `update()`, теперь мы делегируем к компоненту:

```
input_.update(*this);
```

Мы только начали, но уже избавились от части связности: главный класс `Bjorn` уже не содержит ссылки на `Controller`. К этому мы еще вернемся.

Отделение остального

Продолжим и проделаем ту же самую работу по копированию и вставке для физического и графического кода. Вот наш `PhysicsComponent`:

```
class PhysicsComponent
{
public:
  void update(Bjorn& bjorn, World& world)
  {
    bjorn.x += bjorn.velocity;
    world.resolveCollision(volume_,
    bjorn.x, bjorn.y, bjorn.velocity);
  }

private:
  Volume volume_;
};
```

Помимо того, что мы вынесли из главного класса `Bjorn` физическое *поведение*, вы можете видеть, что мы перенесли и *данные*: объект `Volume` теперь принадлежит компоненту.

И теперь последнее, но все равно важное изменение. Теперь наш код рендеринга будет жить здесь:

```
class GraphicsComponent
{
public:
  void update(Bjorn& bjorn, Graphics& graphics)
  {
    Sprite* sprite = &spriteStand_;
    if (bjorn.velocity < 0) {
      sprite = &spriteWalkLeft_;
    } else if (bjorn.velocity > 0) {
      sprite = &spriteWalkRight_;
    }

    graphics.draw(*sprite, bjorn.x, bjorn.y);
  }

private:
  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

Мы убрали практически все. Так что же у нас осталось от нашего скромного кондитера? Не так уж много:

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

Класс `Bjorn` теперь делает всего две вещи: хранит набор компонентов, которые его собственно и определяют и хранит состояние, разделенное между несколькими областями. Позицию и скорость мы оставили в ядре `Bjorn` по двум причинам. Во-первых, это общее для всех областей (“pan-domain”) состояние: практически каждый компонент будет его использовать, так что совсем не очевидно, в каком компоненте ему *нужно* находиться, если мы захотим перенести его туда.

Во-вторых, и что более важно, это упрощает коммуникацию компонентов без установления между ними связи.

Робо-Бьйорн

До сих пор мы выносили поведение в отдельные классы компонентов, но не делали поведение *абстрактным*. `Bjorn` все равно знал о вполне конкретных классах, в которых определялось его поведение. Давайте изменим это.

Мы возьмем наш компонент для обработки пользовательского ввода и спрячем его за интерфейсом. Превратим InputComponent в абстрактный базовый класс:

```
class InputComponent
{
public:
  virtual ~InputComponent() {}
  virtual void update(Bjorn& bjorn) = 0;
};
```

Дальше, мы возьмем наш существующий код обработки пользовательского ввода и поместим его в класс, реализующий этот интерфейс:

```
class PlayerInputComponent : public InputComponent
{
public:
  virtual void update(Bjorn& bjorn)
  {
    switch (Controller::getJoystickDirection())
    {
      case DIR_LEFT:
        bjorn.velocity -= WALK_ACCELERATION;
        break;

      case DIR_RIGHT:
        bjorn.velocity += WALK_ACCELERATION;
        break;
    }
  }

private:
  static const int WALK_ACCELERATION = 1;
};
```

Мы изменим `Bjorn` таким образом, чтобы он содержал указатель на компонент ввода, вместо собственного экземпляра:

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  Bjorn(InputComponent* input)
  : input_(input)
  {}

  void update(World& world, Graphics& graphics)
  {
    input_->update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

Теперь, когда мы инстанцируем `Bjorn`, мы можем передать ему компонент ввода, которым он сможет пользоваться:

```
Bjorn* bjorn = new Bjorn(new PlayerInputComponent());
```

Этот экземпляр может быть конкретным типом, который реализует наш абстрактный интерфейс `InputComponent`. Мы платим за это тем, что теперь `update()` у нас представляет собой вызов виртуального метода, что довольно медленно. Что же мы получаем в замен?

Большая часть консолей требует того, чтобы игра поддерживала "демо режим". Если игрок находится в главном меню и ничего не делает, игра начинает играть автоматически и вместо игрока играет компьютер. Таким образом игра не выжжет на вашем телевизоре главное меню и будет лучше смотреться если запустить ее на стенде в магазине.

В этом нам помогает то, что мы спрятали класс компонента ввода за интерфейсом. У нас уже есть конкретный `PlayerInputComponent`, который обычно используется когда играет игрок. Сделаем еще один:

```
class DemoInputComponent : public InputComponent
{
public:
  virtual void update(Bjorn& bjorn)
  {
    // ИИ для автоматического управления Бьйорном...
  }
};
```

Когда игра переходит в демо режим, вместо того, чтобы конструировать Бьорна также, как и раньше, мы связываем его с новым компонентом:

```
Bjorn* bjorn = new Bjorn(new DemoInputComponent());
```

И теперь, просто подменив компонент у нас получился полноценный управляемый компьютером игрок для демо режима. Мы можем использовать повторно и другой код Бьйорна — физика и графика даже не заметит разницы. Может вы и будете считать меня несколько странным, но такие вещи помогают мне просыпаться по утрам в хорошем настроении.

> Ну и кофе конечно. Ароматный, дымящийся, горячий кофе.

Никакого Бьйорна?

Если вы посмотрите на класс `Bjorn` теперь, вы увидите, что в общем никакого "Бьйорна" там уже нет — это просто мешок с компонентами. На самом деле, это хороший кандидат на роль базового класса "игрового объекта", который можно использовать для *любого* объекта в игре. Все, что нам нужно сделать — это передать в него *все* компоненты и мы сможем получить любой нужный нам объект, подбирая части как доктор Франкенштейн.

Давайте возьмем наши оставшиеся компоненты — физический и рендеринга — и спрячем их за интерфейсом как мы уже поступили с вводом.

```
class PhysicsComponent
{
public:
  virtual ~PhysicsComponent() {}
  virtual void update(GameObject& obj, World& world) = 0;
};

class GraphicsComponent
{
public:
  virtual ~GraphicsComponent() {}
  virtual void update(GameObject& obj, Graphics& graphics) = 0;
};
```

Теперь наш `Bjorn` перерождается в обобщенный класс `GameObject` и будет использовать эти интерфейсы:

```
class GameObject
{
public:
  int velocity;
  int x, y;

  GameObject(InputComponent* input,
    PhysicsComponent* physics,
    GraphicsComponent* graphics)
  : input_(input)
  , physics_(physics)
  , graphics_(graphics)
  {}

  void update(World& world, Graphics& graphics)
  {
    input_->update(*this);
    physics_->update(*this, world);
    graphics_->update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent* physics_;
  GraphicsComponent* graphics_;
};
```

> Некоторые системы компонентов идут даже дальше. В них вместо `GameObject` содержащего компоненты используется просто `ID` — номер. И отдельно хранится колекция компонентов, каждый из которых знает `ID` сущности, к которой прикреплен.
>
> Такая система компонентных сущностей (entity component systems) снижает связность компонентов до самого предела и позволяет добавлять компоненты в сущность таким образом, что сущность даже не будет об этом знать. Более подробно это описано в главе Локализация данных (Data Locality).

Наши существующие конкретные классы будут переименованы и теперь будут поддерживать эти интерфейсы:

```
class BjornPhysicsComponent : public PhysicsComponent
{
public:
  virtual void update(GameObject& obj, World& world)
  {
    // Физический код...
  }
};

class BjornGraphicsComponent : public GraphicsComponent
{
public:
  virtual void update(GameObject& obj, Graphics& graphics)
  {
    // Графический код...
  }
};
```

И теперь мы можем создать объект с полностью идентичным оригинальному Бьйорну поведению, не создавая для этого специальный класс следующим образом:

```
GameObject* createBjorn()
{
  return new GameObject(
      new PlayerInputComponent(),
      new BjornPhysicsComponent(),
      new BjornGraphicsComponent()
  );
}
```

> Эта функция `createBjorn()` конечно представляет собой пример из шаблона Фабричный метод (Factory Method) GoF от банды четырех.

Определяя другие функции, инстанцирующие `GameObjects` с другими компонентами, мы можем создавать самые разные типы объектов, которые нам нужны в нашей игре.

Архитектурные решения

Самый главный вопрос, на который вам придется ответить, когда вы будете применять этот шаблон — это "Какой набор компонентов мне нужен?" Ответ будет зависеть от ваших нужд и от жанра вашей игры. Чем больше и сложнее ваш движок, тем более мелко вам захочется нарезать его на компоненты.

Кроме это существует еще несколько специфических возможностей, над которыми следует подумать:

Как объект будет получать свои компоненты?

Как только мы разрежем наш монолитный объект на несколько отдельных компонентов, нам нужно решить как мы будем снова собирать их вместе.

- **Если объект создает собственные компоненты:**
    - *Мы можем быть уверены что объект всегда будет иметь нужные ему компоненты.* Вам никогда не придется волноваться о том, что кто-то забыл подвязать к объекту нужный компонент и уронил игру. Объект контейнер позаботится об этом самостоятельно.
    - *Объект сложнее реконфигурировать.* Одна из основных функций этого шаблона заключается в том, что он позволяет вам создавать новые типы объектов просто рекомбинируя компоненты. Если ваш объект всегда связывает себя с ними самостоятельно, мы не можем использовать такую гибкость.
- **Если компоненты предоставляет внешний код:**
    - *Объект становится более гибким.* Мы можем полностью изменить поведение объекта, передав ему другие компоненты для работы. В своем максимальном виде, наш объект становится обобщенным контейнером компонентов, который мы можем повторно использовать раз за разом для самых разных целей.
    - *Объект может быть отвязан от конкретных типов компонентов.* Если мы позволим внешнему коду передавать внутрь компоненты, вполне вероятно, что мы позволим передавать и производные типы компонентов. На этом этапе, объект знает только об интерфейсах компонентов, а не самих конкретных типах. В результате получается архитектура с очень хорошей инкапсуляцией.

Как компоненты будут общаться друг с другом?

Минимально связанные компоненты с изолированной функциональностью — это скорее идеал, недостижимый на практике. Тот факт, что компоненты уже являются частью *некоего* объекта, т.е. единого целого, означает что им нужна координация. А это означает коммуникацию.

Так каким же образом наши компоненты будут общаться? У нас есть несколько вариантов, но в отличие от других "альтернатив" в этой книге, они не эксклюзивны: вы скорее всего будете использовать в своей архитектуре сразу несколько из них.

- **С помощью изменения состояния объекта контейнера:**
  - *Компоненты остаются несвязанными.* Когда наш `InputComponent` устанавливает скорость Бьйорна и после этого его использует `PhysicsComponent`, эти два компонента даже не знают о существовании друг друга. Все что они знают — это то, что скорость Бьйорна изменяется с помощью какой-то неизвестной черной магии.
  - *Это требует того, чтобы любая информация, которой хочется поделиться компоненту была перенесена в объект контейнер.* Зачастую — это состояние, нужное только части компонентов. Например, компоненты анимации и рендеринга могут разделять графически специфичную информацию. Помещение этой информации в объект контейнер, где к ней может получить доступ любой компонент только загрязняет класс объекта.

    Гораздо хуже то, что если мы используем объект контейнер с несколькими конфигурациями компонентов, может получиться так, что мы будем тратить память на состояние, которое не нужно ни *одному* из компонентов объекта. Если мы помещаем специфичные для рендеринга данные в объект контейнер, любой невидимый объект будет бесцельно тратить на них ценную память.
  - *Общение становится неявным и зависящим от порядка обработки компонентов.* В нашем примере кода, наш оригинальный монолитный метод `update()` содержал очень тщательно организованную очередность операций: пользовательский ввод изменял скорость, которая затем использовалась физическим кодом для определения позиции, которая в свою очередь использовалась кодом рендера для отрисовки Бьйорна в правильной точке. После того, как мы разделили код на компоненты, мы должны быть осторожны, чтобы не нарушить очередность операций.

    Если мы с этим не справимся, у нас появятся неочевидные и трудноотлавливаемые баги. Например, если мы сначала обновили графический компонент, мы ошибочно отрендерим Бьйорна на позиции из прошлого кадра, а не текущего. Если вы представите себе, что у вас есть еще несколько компонентов и гораздо больше кода, вы сможете себе представить как сложно будет отлавливать такие баги.

    > Общие изменяемые состояния такого типа, когда одни и те же данные считывает и изменяет большое количество кода крайне сложно правильно организовать. Вот почему многие академики тратят столько времени на чисто функциональные языки типа Haskel, где вообще нет изменяемых состояний.

- **Обращаясь друг к другу напрямую:**

Идея тут заключается в том, что компоненты, которым нужно пообщаться, буду иметь прямые ссылки друг на друга и в результате им вообще не придется общаться через контейнер.

Представим себе, что мы хотим научить Бьйорна прыгать. Графическому коду необходимо знать — нужно его рисовать с помощью спрайта прыжка или нет. Для этого он может спросить у физического движка, находится ли он сейчас на земле. Проще всего это сделать, если позволить графическому компоненту обратиться к физическому компоненту напрямую:

```
class BjornGraphicsComponent
{
public:
  BjornGraphicsComponent(BjornPhysicsComponent* physics)
  : physics_(physics)
  {}

  void Update(GameObject& obj, Graphics& graphics)
  {
    Sprite* sprite;
    if (!physics_->isOnGround()) {
      sprite = &spriteJump_;
    } else {
      // Существующий графический код...
    }

    graphics.draw(*sprite, obj.x, obj.y);
  }

private:
  BjornPhysicsComponent* physics_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
  Sprite spriteJump_;
};
```

Когда мы конструируем `GraphicsComponent` Бьйорна, мы передаем ему ссылку на его `PhysicsComponent`.

- *Это просто и быстро*. Общение здесь — это прямой вызов метода одним объектом из другого. Компонент может вызывать любой метод из тех, что поддерживаются компонентом, на который у него есть ссылка. Полная свобода для всех.

- *Два компонента крепко связаны.* Негативная сторона такой свободы. Мы практически делаем шаг назад в сторону нашего монолитного класса. Впрочем, это не настолько плохо, как оригинальный единый класс, так как мы по крайне мере ограничили связность до связности двух компонентов, которым нужно взаимодействовать.

- **С помощью пересылки сообщений:**

- Это самая сложная альтернатива. Мы можем построить внутри нашего объекта контейнера небольшую систему сообщений и позволить компонентам передавать через нее друг другу информацию.

  Вот пример реализации. Мы начнем с определения базового интерфейса `Component`, который будут реализовывать все компоненты.

```
class Component
{
public:
  virtual ~Component() {}
  virtual void receive(int message) = 0;
};
```

  У него есть метод `receive()`, который реализуется в каждом классе компоненте для прослушивания входящих сообщений. В данном случае мы используем для идентификации сообщений простое `int`, но в полной реализации мы можем прикреплять к сообщению и другие данные.

  Далее мы добавим объекту контейнера метод для отсылки сообщений:

```
class ContainerObject
{
public:
  void send(int message)
  {
    for (int i = 0; i < MAX_COMPONENTS; i++) {
      if (components_[i] != NULL) {
        components_[i]->receive(message);
      }
    }
  }

private:
  static const int MAX_COMPONENTS = 10;
  Component* components_[MAX_COMPONENTS];
};
```

А теперь, если у нашего компонента есть доступ к своему контейнеру, он может отсылать ему сообщения, которые в свою очередь в широковещательном режиме передаются всем его родственным компонентам. (В том числе и ему самому. Так что будьте осторожны и не попадитесь в ловушку бесконечного цикла сообщений). У такого решения есть несколько последствий:

> Если вы действительно хотите быть модным, вы можете даже заставить эту систему сообщений организовывать сообщения в очередь для последующей обработки. Подробнее это описано в Очереди событий (Event Queue).

- *Родственные компоненты становятся несвязанными.* Используя родительский объект контейнер, как и в случае с разделяемым состоянием, мы можем быть уверены, что наши компоненты не связаны друг с другом.

  > Банда четырех называет такой шаблон Посредником (Mediator) GoF: два или более объектов общаются друг с другом не напрямую, а передавая сообщения через промежуточный объект. В данном случае в роли посредника выступает сам объект контейнер.

- *Объект контейнер остается простым.* В отличие от варианта с разделяемым состоянием, когда сам объект контейнер обладает данными и знает какие компоненты их используют, здесь он просто вслепую передает сообщения. Таким образомб можно организовать очень специфичную передачу информации между двумя компонентами, не впутывая сюда объект контейнер.

Думаю я вас не удивлю, если скажу что единого правильного ответа здесь нет. В конце концовб вы можете обнаружить, что используете их все сразу. Разделяемые состояния полезны для самых базовых вещей, таких как позиция или размер, которые могут быть практически у любого объекта.

Некоторые области разделены, но все равно достаточно родственные. Например, анимация и рендеринг, пользовательский ввод и ИИ, физика и коллизии. Если у вас есть отдельный компонент для каждой половинки этих пар, вы можете прийти к решению что вам будет проще позволить половинкам знать друг о друге.

Сообщения полезны для "менее важного" общения. Их принцип выстрелил и забыл хорош для вещей типа просьбы к звуковому компоненту проиграть звук, когда физический компонент посылает сообщение о том, что объект с чем-либо пересекся.

И как всегда, я рекомендую вам начать с простого, а затем добавлять дополнительные пути коммуникации по мере необходимости.

Смотрите также

- Класс GameObject из фреймворка Unity целиком построен вокруг компонентов.
- Движок с открытым кодом Delta3D содержит базовый класс `GameActor`, реализующий этот шаблон через базовый класс с соответствующим именем `ActorComponent`.
- Фреймворк XNA от `Microsoft` содержит основной класс `Game`. Он содержит коллекцию объектов `GameComponent`. И пусть наш пример использует компоненты на уровне отдельных игровых сущностей, а `XNA` реализует шаблон на уровне самого главного игрового объекта, зато принцип один и тот же.
- Этот шаблон очень похож на шаблон Стратегия (Strategy) GoF от банды четырех. Оба шаблона посвящены тому, чтобы забрать часть поведения объекта и делегировать его к отдельному подчиненному объекту. Разница заключается в том, что в случае с шаблоном стратегия, отдельный объект стратегия обычно не имеет состояния — он инкапсулирует алгоритм, а не данные. Он определяет *как* объект себя ведет, а не *что* он из себя представляет.

  Компоненты несколько в большей степени самодостаточны. Они зачастую хранят состояния, описывающие объект и помогающие определить его истинную сущность. Однако это не обязательно. У вас вполне могут быть компоненты, которым не нужны никакие локальные состояния. В этом случае, вы легко можете использовать один и тот же *экземпляр* компонента для нескольких объектов контейнеров. В таком случае он будет себя вести практически как стратегия.

Очередь событий (Event Queue)

Задача

Позволяет одной сущности охватывать несколько областей, не связывая их между собой.

Мотивация

Если вы не живете до сих под камнем, где нет доступа к интернету, вы наверняка уже слышали об "очереди событий (event queue)". Ну или хотя бы об "очереди сообщений (message queue)", или "цикле событий (event loop)", "конвейере (обработки) сообщений (message pump)" и т.д. Чтобы освежить вашу память, я хочу познакомить вас с несколькими проявлениями шаблона.

> В большинстве других глав я использую "события" и "сообщения" в качестве синонимов. Но когда это будет необходимо, я буду указывать на их различия.

Цикл событий графического пользовательского интерфейса

Если вы когда-либо занимались программированием графического интерфейса, тогда вы наверняка уже знакомы с *событиями*. Каждый раз, когда пользователь взаимодействует с вашей программой — нажимает кнопку, открывает меню или нажимает клавишу — операционная система генерирует событие. Она вбрасывает этот объект в приложение, а ваша забота — это подхватить его и увязать с каким либо интересным поведением.

> Такой стиль программирования является общепринятым. Такая парадигма называется Событийно-ориентированное программирование (event-driven programming).

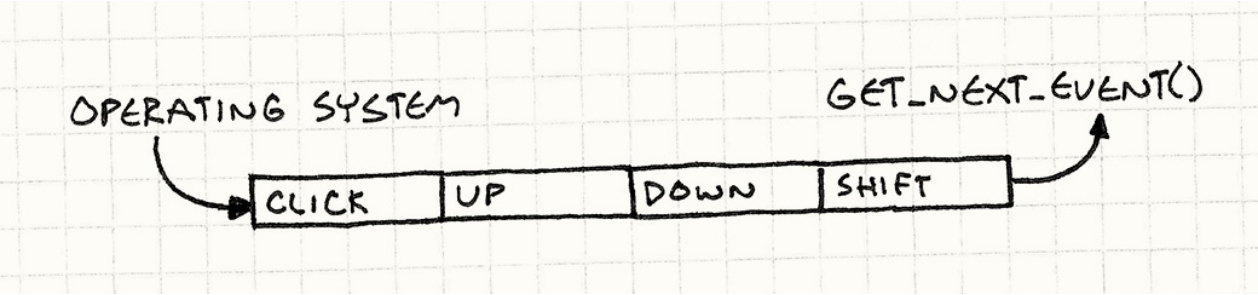
Для того, чтобы получать такие послания, где-то глубоко в недрах вашего кода должен находиться *цикл событий (event loop)*. Выглядит он примерно так:

```
while (running)
{
  Event event = getNextEvent();
  // Обработка события...
}
```

Вызов `getNextEvent()` отправляет порцию необработанного пользовательского ввода в ваше приложение. Вы перенаправляете его в обработчик событий, и ваша программа магическим образом оживает. Самое интересное здесь в том, что приложение *изымает (pulls)* событие когда вам это нужно. Операционная система не сразу перескакивает к определенному коду в вашем приложении, как только пользователь тыкает в периферийное устройство.

> И наоборот, *прерывания* от операционной системы *именно* так и работают. Когда происходит прерывание, операционная система останавливается, чтобы ваша программа не делала, и переходит сразу к обработчику прерываний. Именно из-за такой непредсказуемости с прерываниями так сложно работать.

Это значит, что когда поступает пользовательский ввод, он должен куда-то попасть, так чтобы операционная система не потеряла его между устройством, зарегистрировавшим ввод и следующим вызовом `getNextEvent()` в вашей программе. Это "куда-то" и называется *очередью*.



Когда поступает пользовательский ввод, ОС добавляет необработанные события в очередь. Когда вы вызываете `getNextEvent()`, эта функция вытаскивает самое старое событие из очереди и передает его приложению.

Центральная шина событий

Большинство игр не работает в таком событийно-ориентированном стиле, но для игр является нормальным иметь собственную очередь событий в качестве нервной системы. Обычно ее описывают эпитетами "центральная", "глобальная" или "главная". Используется она для высокоуровневого общения между игровыми системами, остающимися несвязанными.

> Если хотите понять почему не используются — посмотрите в главе Игровой цикл(Game Loop).

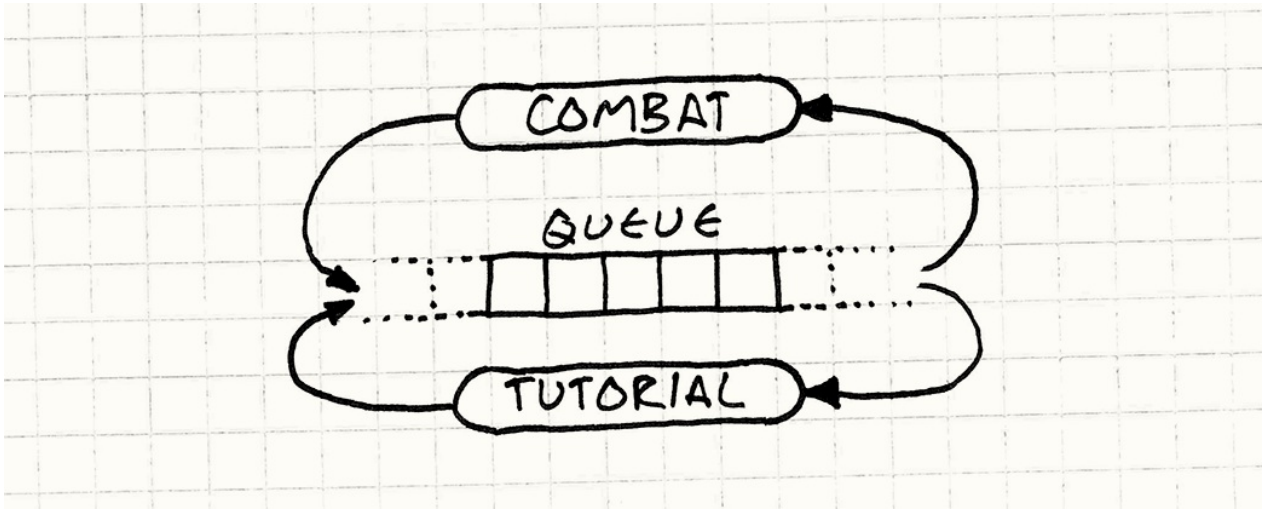
Предположим, что в вашей игре есть система обучения, показывающая окна с подсказками после определенных игровых действий. Например, после того как игрок первый раз побеждает глупое чудовище, вы хотите показать ему маленький овал с надписью "Нажмите X, чтобы собрать добычу!"

> Изящная система обучения мучительна для реализации и большинство игроков практически не обращают на нее внимания, так что можно подумать, что она не стоит ваших усилий. Но, если игроку все-таки *захочется* воспользоваться обучением, оно будет для него неоценимой помощью в освоении вашей игры.

Ваш игровой процесс и боевой код и без того достаточно сложны. Последнее, что вам хотелось бы делать — это помещать внутрь кучу триггеров для запуска обучения. Вместо этого в некоторых играх существует центральная очередь событий. Любая игровая система может посылать в нее события, так что боевая система добавляет в нее событие "враг убит" каждый раз, когда вы расправляетесь с врагом.

В свою очередь, каждая игровая система получает события из очереди. Движок обучения регистрируется в очереди и сообщает о том, что хочет получать события "враг убит". Таким образом, знание о том, что врага убили поступило из боевой системы в движок обучения, при том что эти две подсистемы друг о друге не знают.

> Такая модель, когда у вас есть общее пространство, в котором сущности могут размещать информацию и получать от нее уведомления похожа на систему классной доски (blackboard systems), применяемую в области ИИ.



Я хотел остановиться на этом примере для оставшейся части главы, но я не слишком большой любитель глобальных систем. Это достаточно распространенная техника, но я не хочу, чтобы у вас складывалось впечатление, что очередь событий обязательно должна быть глобальной.

Так о чем мы?

Давайте лучше добавим в нашу игру звук. Люди — скорее визуальные существа, но то, что мы слышим, значительно влияет на наши эмоции и наше ощущение окружающего пространства. Правильно смоделированное эхо может помочь нам представить темную пещеру вместо черного экрана, а вовремя проигранное адажио на скрипке может вызвать у вас учащенное сердцебиение и эмоциональный отклик.

Чтобы наша игра смогла воспроизводить звуки, мы начнем с простейшего подхода и будем его постепенно развивать. Добавим небольшой "аудио движок", позволяющий проигрывать звук с определенным идентификатором и громкостью:

> Несмотря на то, что я очень скептически отношусь к шаблону Синглтон(Singleton) GoF, в данном случае он вполне уместен. Но я буду использовать еще более простое решение и просто сделаю метод статичным.

```
class Audio
{
  public:
  static void playSound(SoundId id, int volume);
};
```

Он отвечает за загрузку нужного аудио ресурса, поиск свободного канала для его проигрывания и запуск. Этот раздел не посвящен какому-то реальному `API` конкретной платформы, так что я буду предполагать что он реализован где-то еще. С его помощью мы напишем собственный метод:

```
void Audio::playSound(SoundId id, int volume)
{
  ResourceId resource = loadSound(id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, volume);
}
```

Мы проверяем его, добавляем внутрь несколько звуковых файлов и начинаем разбрызгивать вызовы `playSound()` по всей кодовой базе подобно аудио фее. Например, в нашем коде пользовательского интерфейса, когда выбранный элемент меню изменяет свое значение, мы проигрываем бип.

```
class Menu
{
public:
  void onSelect(int index) {
    Audio::playSound(SOUND_BLOOP, VOL_MAX);
    // Другие вещи...
  }
};
```

После этого, мы обнаруживаем, что иногда, когда мы переключаем элемент меню, у нас замирает экран на целых несколько кадров. Вот и первая проблема:

- **Проблема 1: `API` блокирует вызывающий код до тех пор, пока аудио движок не закончит свои приготовления.**

Наш метод `playSound()` — *синхронный*. Он не возвращается в вызывающий код до тех пор, пока из динамиков не послышится бип. А если для этого придется сначала подгрузить файл с диска, потребуется еще больше времени.

Пока что проигнорируем это и двинемся дальше. В коде ИИ мы добавляем вызов, позволяющий нам запускать вопль страдания каждый раз, когда враг получает урон от игрока. Ничто так не радует сердце игрока, как причинение страдания виртуальной форме жизни.

Это работает, но иногда, когда наш герой наносит мощный удар, он бьет сразу двух монстров на одном и том же кадре. В таком случае игра запускает звук сразу два раза. А если вы знаете хоть что-то о звуке, вы понимаете что запуск двух звуков одновременно означает смешивание их волновой формы. И когда их форма волны совпадает, это означает что играется тот же самый звук, только *в два раза громче*. Раздражающе громко.

> Я столкнулся с этой проблемой когда работал над Henry Hatsworth in the Puzzling Adventure. Тогда я поступил так, как опишу дальше.

Мы встретимся с той же проблемой и во время битвы с боссом, когда по экрану бегает множество его миньонов. Наше железо может играть ограниченное количество звуков одновременно. Когда мы превышаем этот предел, звук игнорируется или обрезается.

Для обработки таких ошибок нам нужно посмотреть на весь набор вызовов звуков, чтобы агрегировать их и распределить по приоритетам. К сожалению наш аудио `API` обрабатывает каждый вызов `playSound()` по отдельности. Он видит запрос через замочную скважину, по одному за раз.

- **Проблема 2: Запросы невозможно обрабатывать в агрегированной форме.**

Эта проблема выглядит даже более серьезной, чем та с которой мы познакомимся дальше. Но сейчас у нас по всей кодовой базе в самых разных системах распределено множество вызовов `playSound()`. При этом наша игра работает на современном многоядерном железе. Чтобы задействовать все ядра, мы распределяем наши системы по разным потокам — рендеринг в один, ИИ в другой и т.д.

Так как наш `API` является асинхронным, он запускается в потоке *вызывающего* кода. Когда мы вызываем его из другой системы игры, мы сталкиваемся с конкуренцией за наш `API` со стороны нескольких потоков. Посмотрите на код примера. Видите код для потоковой синхронизации? Вот и я нет.

Это особенно очевидно потому, что мы хотим иметь *отдельный* поток для аудио. Он просто будет сидеть там и ожидать всех остальных, пока остальные потоки заняты борьбой и ломкой друг друга.

- **Проблема 3: Запросы обрабатываются в неправильном потоке.**

Обычно эта проблема связана с тем, что аудио движок интерпретирует вызов `playSound()` как "Бросить все и проиграть звук прямо сейчас!". Проблема в этой *немедленности*. Остальные системы игры вызывают `playSound()` когда *им* удобно, но не обязательного в тот момент, когда это удобно аудио движку. Чтобы это исправить, мы отвяжем *получение* запроса от его *обработки*.

Шаблон

Очередь хранит **набор уведомлений или запросов** в порядке первым пришел — первым ушел (`FIFO`). Отсылка уведомлений **ставит запрос в очередь и возвращает управление**. Далее обработчик запросов **обрабатывает элементы очереди**, но несколько позже.

Запросы могут **обрабатываться напрямую** или **перенаправляются заинтересованным сторонам**. Таким образом, мы **снижаем связность отправителя и получателя**, как **статически**, так и **во время** работы.

Когда использовать

Если вы хотите просто отвязать того, кто получает уведомления от их отправителя, проще всего будет воспользоваться помощью шаблонов Наблюдатель (Observer) или Команда (Command). Очередь вам понадобится только, если вы хотите убрать связность между чем-либо *во время* работы.

> Я уже писал об этом в предыдущих главах, но позволю себе повториться. Сложность вас замедляет, поэтому берегите простоту как драгоценный ресурс.

Я думаю об этом в терминах помещения (pushing) и извлечения (pulling). У вас есть некий код A, который хочет, чтобы другой код B выполнил некую работу. Естественным образом для A инициировать это действие будет *помещение* запроса в B.

В то же время, естественным способом обработки такого запроса для B будет *извлечение* запроса в удобное *ему* время, когда наступит *его* время выполниться. Когда у вас с одной стороны образуется модель помещения, а с другой стороны — модель извлечения, вам нужен некий буфер посередине. Это именно то, что предоставляет очередь в отличие от более простых шаблонов.

Очереди передают управление коду, который извлекает из них элементы: получатель может отложить обработку, агрегировать запросы или полностью их удалить. При этом мы *отбираем* управление у отправителя. Все, что он может сделать — это вбросить запрос в очередь и надеяться на лучшее. В этом заключается слабость очередей, если отправителю нужен отклик.

Имейте в виду

В отличие от более современных шаблонов в этой книге, очередь событий довольно сложный шаблон и как правило оказывающий далеко идущие последствия на всю архитектуру игры. Это значит, что вам стоит крепко подумать перед тем, как решить как его применять и применять ли вообще.

Центральная очередь событий — это глобальная переменная

Один из обычных способов применения этого шаблона — это организация Центральной станции, через которую будут проходить сообщения для всех частей игры. Это мощнейший элемент инфраструктуры, но *мощная* не всегда значит *хорошая*.

Пусть это и занимает много времени, но со временем все мы убеждаемся на собственном опыте, что глобальные переменные — это плохо. Когда у вас есть часть состояния, с которой может работать кто угодно, могут возникнуть любые ползучие зависимости. Этот шаблон окружает состояние небольшим удобным протоколом, но он по прежнему глобален, со всеми сопутствующими опасностями.

Состояние мира может изменяться

Предположим, что некий ИИ код отправляет в очередь событие "сущность мертва", когда виртуальный миньон избавился от бренности бытия. Это событие болтается в очереди неизвестно сколько кадров, пока внезапно не оказывается первым и попадает на обработку.

В то же самое время система начисления опыта решает подсчитать количество убитых игроком и вознаградить его устрашающую эффективность. Она получает события "сущность мертва", определяет тип сущности и в зависимости от сложности ее убийства начисляет причитающееся вознаграждение.

Для этого требуется наличие в мире различных частей состояния. Нам нужна умершая сущность, чтобы определить насколько сильной она была. У нас может возникнуть желание посмотреть, что ее окружает и какие препятствия или миньоны находятся рядом. Но если событие было получено гораздо позже, вся эта информация уже пропала. Сущность была удалена, а находящиеся рядом враги разбежались.

Когда вы получаете событие, вы должны быть крайне осторожны с предположением, что *текущее* состояние игрового мира полностью отражает состояние мира в тот момент, *когда событие было порождено*. Это значит что события в очереди должны быть более тяжеловесными, чем в синхронных системах. Мгновенному сообщению достаточно просто сказать что "что-то произошло", а получатель может оглянуться вокруг и получить детали. Когда мы пользуемся очередью, все эти эфемерные детали нужно собрать сразу и отправить вместе с событием, и только в этом случае ими можно будет воспользоваться в дальнейшем.

Вы рискуете застрять в циклах обратного вызова

Все системы событий и сообщений должны опасаться циклов:

1. A Отсылает событие.
2. B получает его и реагирует, отсылая событие.
3. Это событие принадлежит к тому типу, который интересен A. Поэтому A получает его. В свою очередь A отсылает сообщение...
4. Переходим к пункту 2.

Когда наша система сообщений работает *синхронно*, вам будет легко обнаружить циклы: они переполнят стек и обрушат игру. В случае с очередью, асинхронность затрагивает и стек, так что игра будет продолжать работать, несмотря на то, что по ней будут бродить ложные события. Общепринятым способом от этого избавиться является отказ от *отсылки* событий из *обработчиков* событий.

> А еще неплохой идеей будет добавить в вашу систему событий вывод отладочной информации.

Пример кода

У нас уже есть некий код. Он не совершенен, но обладает правильной базовой функциональностью: нужный нам публичный аудио `API` и работающие низкоуровневые аудио вызовы. Все что нам осталось — это исправить проблемы.

Первая — это *блокировка* нашего `API`. Когда фрагмент кода проигрывает звук, он не может ничего сделать пока `playSound()` не закончит загрузку и начнет издавать звуки через динамик.

Мы хотим отложить эту работу на потом, чтобы `playSound()` мог возвращать управление сразу. Чтобы это сделать, нам нужно *реализовать (reify)* запрос на проигрывание звука. Нам потребуется маленькая структура, хранящая детали формируемого запроса, которые понадобятся позже.

```
struct PlayMessage
{
  SoundId id;
  int volume;
};
```

Теперь нам нужно добавить в `Audio` место для отслеживания этих добавляемых сообщений. Профессор по алгоритмам сказал бы вам, что нужно использовать структуры данных типа Кучи Фибоначчи (Fibonacci heap) или списка с пропусками (skip list) или, черт побери, по крайне мере *связанный* список. Но на практике практически всегда хранить однородные структуры данных лучше в старом добром линейном массиве:

> Разработчики алгоритмов зарабатывают на публикации анализа новых структур данных. Они точно не заинтересованы в сохранении основ.

- Никакого динамического выделения памяти.
- Никаких накладных расходов на хранение дополнительной информации или указателей.
- Дружелюбное для кеша последовательное использование памяти.

> Более подробно о том, что такое дружелюбность для кеша написано в главе Локализация данных (Data Locality).

Приступим:

```
class Audio
{
public:
  static void init() {
    numPending_ = 0;
  }

  // Другие вещи...
private:
  static const int MAX_PENDING = 16;

  static PlayMessage pending_[MAX_PENDING];
  static int numPending_;
};
```

Мы можем настроить размер массива из расчета на самый худший сценарий. Чтобы проиграть звук, мы просто помещаем в него в конце новое сообщение:

```
void Audio::playSound(SoundId id, int volume)
{
  assert(numPending_ < MAX_PENDING);

  pending_[numPending_].id = id;
  pending_[numPending_].volume = volume;
  numPending_++;
}
```

В результате `playSound()` может возвращать управление сразу, но нам конечно все равно придется проигрывать звук. Этот код нужно куда-то перенести и этим куда-то будет метод `update()`:

```
class Audio
{
public:
  static void update() {
    for (int i = 0; i < numPending_; i++) {
      ResourceId resource = loadSound(pending_[i].id);
      int channel = findOpenChannel();
      if (channel == -1) return;
      startSound(resource, channel, pending_[i].volume);
    }

    numPending_ = 0;
  }

  // Другие вещи...
};
```

> Как следует из имени — это шаблон Метод обновления (Update Method).

Теперь нам просто нужно вызвать его из удобного для нас места. Это "удобно" зависит от вашей игры. Это может значить как вызов прямо из Игрового цикла (Game Loop) или из выделенного звукового потока.

Все это хорошо работает, но не подразумевает того, что мы можем обработать все звуковые запросы в едином вызове `update()`. Если вы делаете нечто в духе асинхронной обработки запросов после загрузки нужного звукового ресурса, это не сработает. Чтобы `update()` обрабатывал запросы по одному, ему нужно иметь возможность извлекать запросы из буфера.

Кольцевой буфер

Существует куча способов реализации очереди, но мне больше всего нравится называемая *кольцевой буфер*. Он сохраняет все достоинства массивов, но позволяет инкрементно удалять элементы из начала очереди.

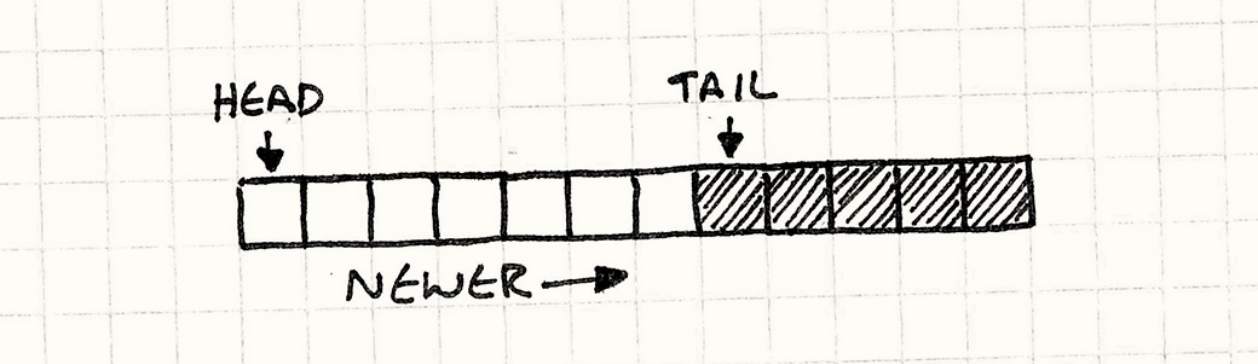
Я знаю, что вы подумали. Если мы удаляем элементы из начала очереди, не приходится ли нам сдвигать все оставшиеся элементы? Не будет ли это слишком медленно?

Вот почему нам советуют учить связные списки: вы можете удалять из них узлы без необходимости передвигать элементы. Вы в конце концов тоже можете реализовать очередь без сдвигов массива. Я через это тоже проходил, но для начала немного терминологии.

- **Голова** очереди — это откуда *считываются* запросы. Голова — это самый старый поступивший запрос.

- **Хвост** — это другой конец очереди. Это слот в массиве, в который *записывается* следующий запрос. Обратите внимание, что он находится *после* конца очереди. Вы можете думать о нем как о полуоткрытом диапазоне, если вам так будет удобнее.

Так как `playSound()` добавляет новые запросы в конец массива, голова начинается с нулевого элемента, а хвост растет вправо.



Давайте это закодируем. Для начала изменим немного наши поля, чтобы сделать эти два маркера явными в классе:

```
class Audio
{
public:
  static void init() {
    head_ = 0;
    tail_ = 0;
  }

  // Методы...
private:
  static int head_;
  static int tail_;

  // Массив...
};
```

В реализации `playSound()`, `numPending_` заменяется на `tail_`, но в целом все то же:

```
void Audio::playSound(SoundId id, int volume)
{
  assert(tail_ < MAX_PENDING);

  // Добавление в конец списка.
  pending_[tail_].id = id;
  pending_[tail_].volume = volume;
  tail_++;
}
```

Гораздо интереснее изменения внутри `update()`:

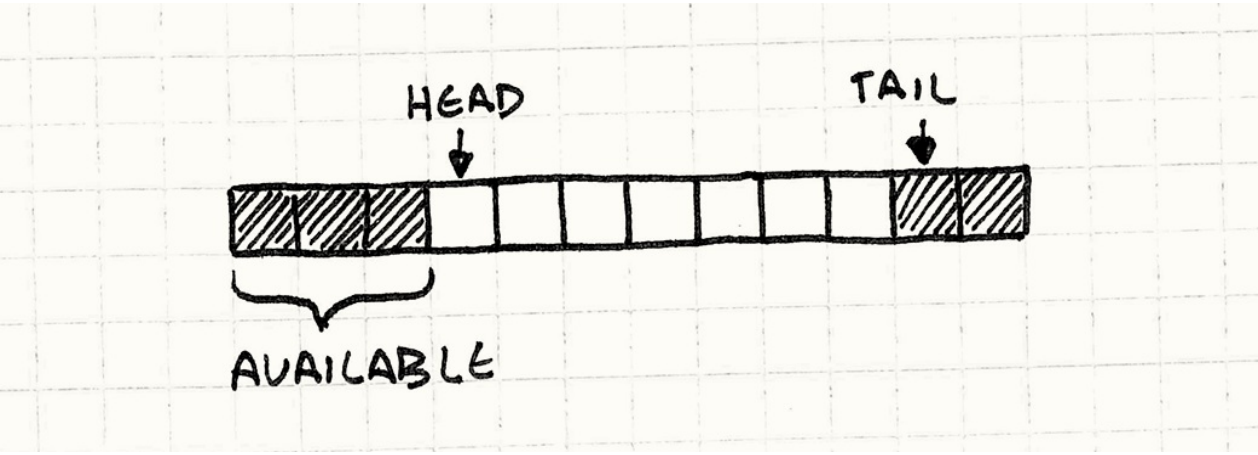
```
void Audio::update()
{
  // Если запросов в очереди нет — ничего не делаем.
  if (head_ == tail_) return;

  ResourceId resource = loadSound(pending_[head_].id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, pending_[head_].volume);

  head_++;
}
```

Мы обрабатываем запросы начиная с головы и затем удаляем их, смещая указатель головы вправо. То, что очередь пустая, мы обнаружим когда у нас не будет никакого расстояния между головой и хвостом.

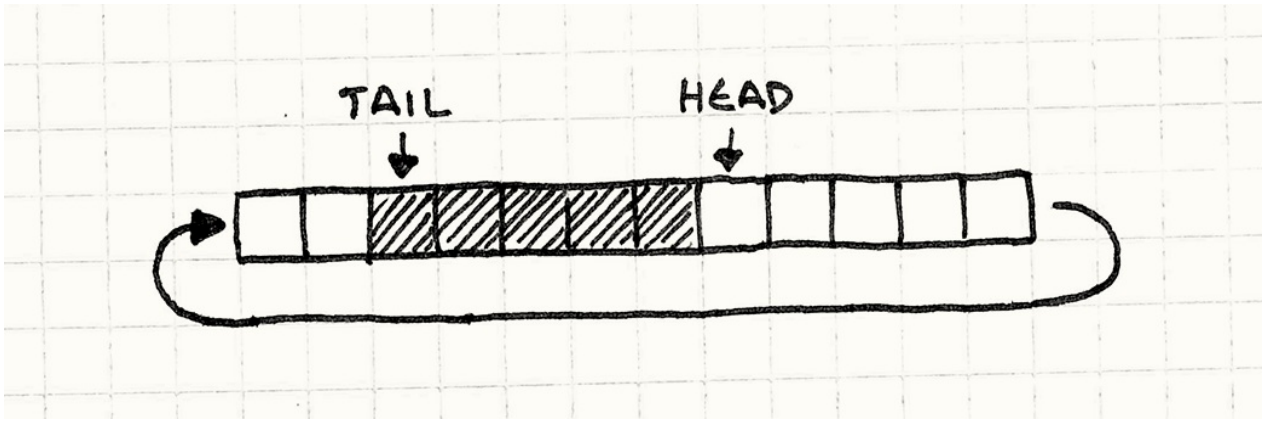
> Вот поэтому хвост и находится *после* последнего элемента. Это и значит, что если очередь будет пустой, индекс головы и хвоста будут совпадать.

Теперь, когда у нас есть очередь, мы можем добавлять элементы в конец и удалять их из начала. Здесь есть очевидная проблема. По мере того, как мы будем идти по очереди запросов, голова и хвост будут смещаться вправо. В конце концов `tail_` достигнет конца массива и вечеринка закончится. И здесь придется подумать.

> Мы ведь не хотим, чтобы вечеринка закончилась? Нет. Конечно нет.



Обратите внимание, что пока хвост ползет вправо, *голова* ползет тоже. Это значит, что у нас в *начале* массива образуются элементы, которые мы больше не используем. И поэтому все, что мы делаем — это переносим хвост назад в начало массива после того как достигли его конца. Вот почему буфер называется *кольцевым*: он работает как циклический массив ячеек.



Реализация максимально проста. Когда мы добавляем в очередь элемент, нам нужно удостовериться, что хвост перейдет на начало массива, когда достигнет его конца:

```
void Audio::playSound(SoundId id, int volume)
{
  assert((tail_ + 1) % MAX_PENDING != head_);

  // Добавление в конец списка.
  pending_[tail_].id = id;
  pending_[tail_].volume = volume;
  tail_ = (tail_ + 1) % MAX_PENDING;
}
```

`tail_++` заменяется на деление инкремента по модулю на размер массива чтобы он переходил в начало. Еще одно изменение — это добавление `assert`. Нам нужно быть уверенными, что очередь не переполнена. Пока у нас в очереди будет меньше запросов чем `MAX_PENDING`, у нас будет оставаться промежуток между головой и хвостом. Если очередь будет полной, этот промежуток исчезнет и у нас получится странный Уроборос. Хвост пересечется с головой и начнет ее перезаписывать. `assert` проверяет чтобы этого не случилось.

Добавляем в `update()` и перенос головы:

```
void Audio::update()
{
  // Если запросов в очереди нет — ничего не делаем.
  if (head_ == tail_) return;

  ResourceId resource = loadSound(pending_[head_].id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, pending_[head_].volume);

  head_ = (head_ + 1) % MAX_PENDING;
}
```

Вот что у нас получилось: очередь с динамическим выделением памяти, никакого копирования элементов и кеш-дружелюбное поведение. И все это с помощью простого массива.

> Если вас беспокоит максимальная вместимость, вы можете использовать растущий массив. Когда очередь переполняется, мы выделяем массив в два раза больше чем предыдущий (или используем другой множитель) и копируем в него все элементы.
>
> Даже если при увеличении массива использовать копирование, добавление элемента все равно имеет константную *амортизированную* сложность.

Агрегация запросов

Теперь, когда мы разобрались с очередью, мы можем перейти к следующей проблеме. Первая заключается в том, что два запроса на проигрывание одного и того же звука приведут к проигрыванию звука один раз, но слишком громко. Так как мы знаем какие запросы ожидают обработки, все что нам нужно сделать — это объединить запросы, которые совпадают с уже обрабатываемым:

```
void Audio::playSound(SoundId id, int volume)
{
  // Обход ожидающих запросов.
  for (int i = head_; i != tail_;
      i = (i + 1) % MAX_PENDING) {
    if (pending_[i].id == id) {
      // Использование большего из двух значений.
      pending_[i].volume = max(volume, pending_[i].volume);

      // Не нужно добавлять.
      return;
    }
  }

  // Предыдущий код...
}
```

Если у нас есть два запроса на проигрывание одного и того же звука, мы объединяем их в один запрос с наибольшей громкостью. "Агрегация" здесь довольно рудиментарна, но мы можем использовать ту же идею и для более интересных серий.

Обратите внимание, что мы объединяем запросы когда запрос *добавляется*, а не *обрабатывается*. Это лучше и для очереди, потому что нам не нужно тратить слоты на лишние запросы, которые придется удалять позже. И для реализации это проще.

К сожалению такое решение увеличивает время обработки вызова. Вызов `playSound()` обходит всю очередь перед тем, как вернуть управление, что может подтормаживать на больших очередях. Поэтому возможно агрегацию лучше перенести в `update()`.

> Еще одним способом избежать O(n) стоимости сканирования очереди является использование другой структуры данных. Если мы будем использовать хеш таблицу с `SoundId` в виде ключа, мы сможем выполнять поиск дубликатов за константное время.

Здесь есть еще одна важная вещь, о которой нужно помнить. Окно "одновременности" запросов, которые мы можем агрегировать, не может превышать размер очереди. Если мы обрабатываем запросы быстрее и размер очереди остается маленьким, у нас остается меньше шансов на то, чтобы объединить ее элементы. И наоборот, если обработка происходит с лагами и наша очередь переполняется, мы сможем объединить больше элементов.

Этот шаблон избавляет отправителя от необходимости знать когда будет обработан запрос, но когда вы трактуете всю очередь как живую структуру данных с которой вы работаете, тогда лаг между запросом и обработкой может иметь видимое влияние на поведение. Так что прежде чем поступать таким образом подумайте — устраивает ли вас такое поведение.

Разделение потоков

И наконец самая страшная проблема. В нашем синхронном `API` в каком потоке вызывалась `playSound()`, в том же потоке запрос и обрабатывается. А это не всегда то, что мы хотим.

На современном многоядерном оборудовании, вам нужно использовать больше одного потока, если вы хотите использовать все возможности своего оборудования. Есть бесчисленное количество вариантов распределения кода по потокам, но общепринятой стратегией является размещение каждой системы в отдельном потоке — аудио, рендер, ИИ и т.д.

> Прямолинейный код работает только на одном ядре. Если вы не используете потоки, то даже если используете модный сейчас асинхронный стиль программирования, вы все равно можете нагрузить только одно ядро, т.е. используете только часть возможностей вашего процессора.
>
> Серьезные программисты компенсируют это разделением приложения на множество независимых *процессов*. Это позволит ОС работать с ним в конкурентном режиме на нескольких ядрах. Игры почти всегда работают в виде одного процесса, так что потоки действительно полезны.

Теперь, когда у нас есть три критических элемента, мы к этому готовы:

1. Код для запроса звука отвязывается от кода, который его проигрывает.
2. У нас есть очередь для общения между этими двумя.
3. Очередь инкапсулирована от остальной программы.

Все, что осталось — это сделать изменяющие очередь методы — `playSound()` и `update()` потоко-безопасными. Обычно я не стал бы это делать с помощью конкретного кода, но так как эта книга посвящена архитектуре, я не хочу углубляться в детали конкретного `API` или механизмов блокировки.

На самом высоком уровне все, что нам нужно сделать — это убедиться в том, что очередь не изменяется в конкурентном режиме. Так как `playSound()` делает совсем немного работы — попросту заполняет всего несколько полей — эта функция может выполнять `lock`, не блокируя обработку надолго. В `update()` нам понадобится нечто наподобие условной переменной, чтобы мы не тратили впустую процессорные циклы, пока не появятся запросы для обработки.

Архитектурные решения

Многие игры используют очередь событий в качестве ключевой части их коммуникационной структуры и вы также можете потратить кучу времени на проектирование различных типов сложных маршрутов и фильтрации сообщений. Но прежде, чем вы решитесь построить нечто столь же громадное, как телефонный коммутатор Лос-Анджелеса. Рекомендую начать с простого. Вот несколько вопросов, над которыми нужно подумать:

Что происходит в очереди?

Я использовал до сих пор слова "событие" и "сообщение" как взаимозаменяемые, потому что разница практически не имела значения. На снижение связности и агрегацию это не влияет.

- **Если вы используете очередь событий:**

  "Событие" или "уведомление" описывает нечто что уже случилось в духе "монстр помер". Вы помещаете их в очередь так что объекты смогут реагировать на события, примерно как в асинхронном шаблоне Наблюдатель(Observer) GoF.

- *Вы можете позволить себе несколько слушателей*. Так как очередь содержит вещи, которые уже случились, отправителя мало беспокоит кто их получает. С этой точки зрения, события находятся в прошлом и уже забыты.
- *Область видимости очереди стремится к расширению*. Очередь событий часто используется для широкополосного вещания (broadcast) событий во все заинтересованные стороны. Чтобы обеспечить максимальную гибкость для всех заинтересованных частей, такую очередь стоит сделать глобальной.

- **Если вы используете очередь сообщений:**

  "Сообщение" или "запрос" описывают действие, которое мы хотим чтобы случилось *в будущем*. Например, "проиграть звук". Вы можете думать об этом, как об асинхронном `API` для сервиса.

  > Еще одно название для "запроса" — это "команда", как в шаблоне Команда(Command) GoF и с ним тоже можно использовать очередь.

  - *У вас скорее всего будет всего один слушатель*. В нашем примере, помещенные в очередь сообщения — это запросы специально для *аудио* `API` для проигрывания звука. Если другие случайные части игрового движка начнут похищать сообщения из очереди, ни к чему хорошему это не приведет.

    > Я говорю "скорее всего", потому что вы можете добавлять сообщения не беспокоясь о том, какой код его обрабатывает до тех пор, пока он будет обрабатываться так, как вы ожидали. В таком случае вы делаете нечто в духе Поиска сервиса (Service Locator).

Кто может читать из очереди?

В нашем примере очередь инкапсулирована и только `Audio` класс может из нее читать. В системе событий пользовательского интерфейса, вы можете регистрировать любых слушателей, каких только ваша душа пожелает. Вы наверняка слышали два разных термина: "направленное вещание" и "широкополосное вещание". Полезны оба стиля.

- **Очередь с направленным вещанием:**

  Такой подход естественен когда очередь является частью `API` класса. Как в нашем примере с аудио, с точки зрения вызывающего кода, видно только доступный для вызова метод `playSound()`.

  - *Очередь становится реализацией деталей читателя*. Все, что может отправитель — так это просто отправлять сообщения.

- *Очередь сильнее инкапсулирована*. При прочих равных условиях, сильная инкапсуляция всегда лучше.
- *Вам не нужно беспокоиться о конкуренции между слушателями*. Когда у вас есть несколько слушателей, вам нужно решить, будут ли они все получать каждый элемент (широкополосное вещание) или каждый элемент в очереди предназначен строго для конкретного слушателя (больше похоже на живую рабочую очередь).

  В любом случае, слушатели могут выполнять лишнюю работу или мешать друг другу и вам нужно тщательно продумывать, какое у них должно быть поведение. Когда у вас всего один слушатель, никаких подобных сложностей не возникает.

- **Очередь с широкополосным вещанием:**

  Именно так работает большинство систем "событий". Если у вас есть десять слушателей когда происходит событие, его увидят все десять слушателей.

  - *События могут упасть на пол*. Из предыдущего вывода следует, что если у вас ноль слушателей, событие получат все ноль. В большинстве широкополосных систем, если по время обработки события у нас нет никаких слушателей, события просто пропадают.
  - *Вам может понадобиться фильтровать события*. Широкополосные очереди обычно видны большей части программы и в результате у вас получится куча слушателей. Перемножая множество событий на множество слушателей, мы получим огромное количество обработчиков событий.

    Чтобы снизить их количество, большинство систем широкополосного вещания позволяют слушателям ограничивать набор принимаемых событий. Например, они могут объявить, что принимают события мыши или события в определенной части пользовательского интерфейса.

- **Рабочая очередь:**

  Как и в случае с широкополосной очередью, здесь у вас тоже будет множество слушателей. Разница в том что теперь каждое событие из очереди отправляется только одному слушателю. Этот шаблон обычно распределяет работу между пулом конкурентно работающих потоков.

  - *Вам потребуется планировщик*. Так как элементы передаются только одному слушателю, очереди нужна логика для определения наилучшего кандидата. Это может быть очевидное решение, случайный выбор или более сложная система приоритетов.

Кто может писать в очередь?

Это обратная сторона предыдущего архитектурного решения. Этот шаблон работает со всеми конфигурациями чтения/записи: один к одному, один к многим, многие к одному или многие к многим.

> Возможно вы слышали название " fan-in " для описания системы коммуникации многие к одному и " fan-out " для системы один ко многим.

- **Когда писатель один:**

  Этот стиль больше всего похож на синхронный шаблон Наблюдатель (Observer) GoF. Вам нужен один привилегированный объект, генерирующий события, которые будут получать другие.

  - *Вы косвенно знаете откуда поступают события*. Так как пополнять очередь может только один объект, все слушатели могут быть уверены, что именно он является отправителем.
  - *Обычно вы разрешаете существовать множеству читателей*. Вы конечно можете создать очередь с одним отправителем и одним получателем, но тогда шаблон уже не будет похож сам на себя и превратится в подобие обычной структуры данных очередь.

- **Когда писателей много:**

  Именно так работает пример с аудио. Так как `playSound()` публичный метод, любая часть кодовой базы может добавлять в очередь запросы. Именно так работает "глобальная" или "центральная" система событий.

  - *Вам нужно быть осторожнее с циклами*. Так как потенциально помещать запросы в очередь может кто угодно, очень просто отправить какой-то запрос прямо в середине обработчика события. И если вы не будете достаточно осторожны, то рискуете получить цикл обратных вызовов.
  - *Вам может понадобиться ссылка на отправителя в самом событии*. Когда слушатель получает событие, он не знает кто его послал, потому что это может быть кто угодно. И если вам нужно это знать, вам нужно добавить такую информацию в событие, чтобы слушатель мог ею воспользоваться.

Какова продолжительность жизни событий в очереди?

В системах синхронных уведомлений, выполнение не возвращается обратно отправителю, пока получатель не закончит обработку сообщения. Это значит, что сообщение может безопасно размещаться в локальной переменной или в стеке. Когда

у нас появляется очередь, сообщение живет дольше, чем породивший его вызов.

Если вы используете язык со сборщиком мусора, вам не нужно слишком об этом беспокоиться. Вы помещаете сообщение в очередь и оно будет там находиться до тех пор, пока не перестанет быть нужным. В `C` или `C++` вы сами должны позаботиться о том, чтобы объект прожил достаточно долго.

- **Передача владения:**

  Это традиционный способ работы при ручном управлении памятью. Когда сообщение попадает в очередь, очередь забирает его себе и отправитель им больше не владеет. После того как оно отправляется на обработку, его забирает себе получатель и теперь он отвечает за удаление сообщения.

  > В `C++` именно такую семантику предоставляет нам `unique_ptr<T>` прямо из коробки.

- **Совместное владение:**

  В наши дни, когда даже `C++` программисты работают с достаточным комфортном со сборщиком мусора, все чаще применяется совместное владение. При этом сообщение остается живым, пока на него хотя бы кто-то ссылается и автоматически удаляется, когда его все забыли.

  > Анаголично в `C++` для этого предусмотрен `shared_ptr<T>`.

- **Принадлежность очереди:**

  Еще одним вариантом является разрешение событию жить в очереди *всегда*. Вместо выделения памяти для события самостоятельно, отправитель запрашивает "свежее" из очереди. очередь возвращает ссылку на сообщение, которое уже выделено и находится в очереди, а отправитель его заполняет. Когда сообщение поступает на обработку, получатель обращается к тому же сообщению из очереди.

  > Другим словами получается резервное хранилище для очереди как в Пуле объектов (Object Pool).

Смотрите также

- Я уже упоминал об этом несколько раз, но во многом этот шаблон является асинхронным родственником хорошо знакомого нам шаблона Наблюдатель (Observer) GoF.

- Как и многие другие шаблоны, очередь событий имеет множество псевдонимов. Один из таких устоявшихся терминов — это "очередь событий". Он обычно ссылается на проявление более высокого уровня. Очередь событий обычно используется для общения *внутри* приложения, а очередь сообщений для общения *между* приложениями.
- Еще один термин "публикация/подписка (publish/subscribe)", часто сокращаемый до "pubsub". Как и очередь сообщений, он обычно применяется для описания распределенных систем, и в меньшей степени для скромных шаблонов программирования, которыми мы здесь занимаемся.
- Конечный автомат, подобно шаблону Состояние (State) GoF от банды четырех требует потока ввода. Если вы хотите реагировать на него асинхронно, вам поможет применение очереди.

  Если у вас есть множество конечных автоматов, обменивающихся сообщениями, у каждого из которых есть небольшая очередь поступающих сигналов (называемая *почтовым ящиком (mailbox)*), значит вы переизобретаете актерскую модель общения.
- В языке Go встроенный тип "канал" представляет из себя очередь сообщений или событий.

Поиск службы (Service Locator)

Задача

Обеспечить глобальную точку доступа к службе без привязки пользователя к конкретному классу, который ее реализует.

Мотивация

Некоторым объектам или системам в игре может понадобиться посещать практически все уголки кодовой базы. Тяжело найти часть игры, которой *не* требуется выделять память, вести журнал, работать с файловой системой или генерировать случайные числа. О таких системах можно думать как о *сервисах*, которые должны быть доступны всей игре.

В нашем примере, мы будем работать с аудио. Это конечно не что-то настолько низкоуровневое, как выделение памяти, но все равно затрагивает кучу игровых систем: Падающий камень ударяется об землю (физика). Снайпер `NPC` стреляет из винтовки (ИИ). Пользователь выбирает элемент меню с бипающим подтверждением (пользовательский интерфейс).

В каждом из этих мест нам нужно иметь возможность обращаться к аудио системе примерно таким образом:

```
// Используем статический класс?
AudioSystem::playSound(VERY_LOUD_BANG);

// Или может быть синглтон?
AudioSystem::instance()->playSound(VERY_LOUD_BANG);
```

Неважно каким путем мы пойдем, потому что в любом случае мы столкнемся с увеличением связности. Каждый участок кода в нашей игре может вызывать напрямую класс `AudioSystem` и для этого можно использовать механизм статического класса или Синглтона (singleton) [GoF].

Эти вызовы естественно должны быть к чему-то привязаны, чтобы звук проигрывался, но позволять всем обращаться к вполне конкретной аудио реализации — это все равно, что давать сотне незнакомцев указания как добраться до вашего дома, чтобы

они могли оставить письмо у вас на ступеньках. Это не просто слишком персонализированное обращение, это крайне неудачная идея, если вам придется потом сообщать всем, как добраться до нового места.

Есть решение получше: телефонная книга. Люди, которым нужно с нами связаться могут найти наше имя и получить наш адрес. Когда мы переезжаем, мы сообщаем об этом телефонной компании. Они обновляют книгу и все снова могут получить правильный адрес. Кроме того мы можем вообще не сообщать реальный адрес. Мы можем просто завести специальный почтовый ящик, который будет нас "представлять". Позволяя нашим респондентам находить нас по книге, мы организуем *единое место, управляющее тем, как нас найти*.

В этом и заключается суть шаблона *Поиск службы*: он отвязывает код, которому нужна служба от того, чем она является (Тип конкретной реализации) и *где* находится (как получить экземпляр).

Шаблон

Класс *служба* определяет абстрактный интерфейс для набора операций. Конкретный *поставщик службы* (service provider) реализует этот интерфейс. Отдельный *поиск службы* (service locator) предоставляет доступ к службе и занимается поиском нужного поставщика и скрывает конкретный тип поставщика и процесс его поиска.

Когда использовать

Каждый раз, когда у вас появляется нечто, доступное для каждой части вашей программы — вы напрашиваетесь на проблемы. Это основная проблема шаблона Синглтон (singleton) GoF и этот шаблон в этом плане ничем от него не отличается. Мой основной совет насчет того, когда использовать шаблон: *пореже*.

Вместо того, чтобы использовать глобальный механизм для предоставления коду доступа к нужному объекту, рассмотрим сначала *вариант передачи им самого объекта*. Это крайне просто и делает связность очевидной. И к том уже удовлетворяет все наши потребности.

Но... бывают ситуации, в которых ручная передача объекта ничем не обоснована или даже затрудняет чтение кода. Некоторые системы, такие как журналирование или управление памятью, не должны быть частью модуля с публичным `API`. Параметры для вашего кода рендеринга должны касаться *рендеринга*, а не вещей типа журналирвоания.

Более того, остальные системы представляют системы, фундаментально независимые по своей сути. Ваша игра скорее всего может общаться только с одним аудио устройством и одной системой видеовывода. Это свойство окружающей среды, поэтому помещать их за десять слоев методов, чтобы к ним имел доступ только глубоко запрятанный метод, не только бессмысленно, но и усложняет весь ваш код.

В подобных случаях этот шаблон может быть полезен. Как мы увидим дальше, он работает более гибким и настраиваемым образом, чем *Синглтон*. При разумном использовании, он придаст вашей кодовой базе значительную гибкость без особых затрат в плане производительности.

> Соответственно если применить шаблон ни к месту, он потянет за собой в качестве багажа весь шаблон *Синглтон*, да еще и с ущербом для производительности.

Имейте в виду

Основная сложность при работе с поиском службы заключается в том, что он берет зависимость — связь между двумя кусками кода — и откладывает их связывание до времени выполнения. Это добавляет нам гибкости, но ценой является усложнение понимания зависимостей при чтении кода.

Служба обязана находиться

Когда мы используем синглтон или статический класс, у нас не может возникнуть ситуации, когда объект недоступен. Вызывающий их код может быть уверен, что он существует. Но так как этот шаблон предполагает *поиск* службы, нам нужно обрабатывать случаи, когда поиск завершится неудачей. К счастью, позже мы рассмотрим стратегию, направленную на то, чтобы гарантировать наличие *хоть какой-то* службы, когда мы ее запрашиваем.

Служба не знает о том кто ее ищет

Так как поиск доступен глобально, любой код в игре может запрашивать службу и использовать ее. Это значит, что служба должна уметь работать в любых обстоятельствах. Например, класс, который предполагается использовать во время части симуляции внутри игрового цикла, но не во время ренденринга не может быть службой — мы не можем гарантировать, что он не будет использован в неподходящее время. Поэтому, если класс предполагается использовать только в определенном контексте, безопаснее избегать демонстрации его внешнему миру с помощью этого шаблона.

Пример кода

Вернемся к нашей проблеме с аудио системой и попробуем организовать ее видимость для остальной кодовой базы с помощью поиска службы.

Служба

Начнем с аудио `API`. Вот интерфейс, который будет предлагать наша служба:

```
class Audio
{
public:
  virtual ~Audio() {}
  virtual void playSound(int soundID) = 0;
  virtual void stopSound(int soundID) = 0;
  virtual void stopAllSounds() = 0;
};
```

Настоящий аудио движок конечно будет более сложным, но мы ограничимся базовой идеей. Что здесь важно, так это то, что интерфейс является абстрактным без привязки к реализации.

Поставщик службы

Сам по себе аудио интерфейс не слишком полезен. Нам нужна конкретная реализация. Эта книжка не посвящена написанию аудио кода, так что представьте себе что за этими функциями стоит реальный код:

```
class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID) {
    // проигрываем звук, используя аудио api консоли...
  }

  virtual void stopSound(int soundID) {
    // останавливаем звук, используя api консоли...
  }

  virtual void stopAllSounds() {
    // останавливаем все звуки, используя api консоли...
  }
};
```

Теперь у нас есть интерфейс и его реализация. Оставшаяся часть поиска службы — класс, связывающий их вместе.

Простой поиск

Следующая реализация представляет собой простейший тип поиска службы:

```
class Locator
{
public:
  static Audio* getAudio() { return service_; }

  static void provide(Audio* service) {
    service_ = service;
  }

private:
  static Audio* service_;
};
```

> Такая техника обычно называется *иньекцией зависимости* (dependency injection) — неуклюжий жаргон для простой идеи. Предположим, у вас есть класс, зависящий от другого. В нашем случае класс `Locator` нуждается в экземпляре службы `Audio`. Обычно, поиск сам отвечает за его создание. Иньекция зависимости наоборот, предполагает что внешний код отвечает за *иньекцию* этой зависимости в объект, которому это нужно.

Статическая функция `getAudio()` выполняет поиск — мы можем вызвать ее откуда угодно из нашей кодовой базы и она вернет нам экземпляр `Audio`, который мы сможем использовать:

```
Audio *audio = Locator::getAudio();
audio->playSound(VERY_LOUD_BANG);
```

Способ "поиска" очень прост: он полагается на внешний код для регистрации поставщика службы, который должен быть выполнен прежде, чем службой можно будет воспользоваться. Когда начинается игра, он вызывает нечто наподобие:

```
ConsoleAudio *audio = new ConsoleAudio();
Locator::provide(audio);
```

Главное, на что стоит обратить здесь внимание — это то, что код, вызывающий `playSound()`, не заботится о конкретном классе `ConsoleAudio`, а требует только абстрактный интерфейс `Audio`. Не менее важно то, что даже класс *поиска* не привязан к конкретной предоставляемой службе. *Единственное* место в коде, которое знает о настоящем конкретном классе находится в функции инициализации, в которой служба регистрируется.

У нас есть еще один уровень снижения связности: интерфейс `Audio` не заботит тот факт, что доступ к нему обычно выполняется через поставщика службы. Все, что он знает — это обычный абстрактный базовый класс. Это полезно, потому что означает, что мы можем применять этот шаблон к существующим классам, которые не были специально для этого разработаны. В этом и заключается главная разница с Синглтон (singleton) [GoF], который непосредственно влияет на архитектуру класса "службы".

Нулевая служба

Пока что наша реализация довольно простая и достаточно гибкая. Но у нее есть одно серьезное ограничение: если мы попробуем использовать службу до того, как она будет зарегистрирована, мы получим `NULL`. Если код вызова не сделает соответствующей проверки, наша игра упадет.

> Я иногда слышу, как это называют "временным связыванием" (temporal coupling): два разных куска кода, которые должны вызываться в определенной последовательности для корректной работы программы. В каждой программе присутствует нечто подобное, но как и в случае со всеми прочими видами связности, чем связности меньше, тем проще работать с кодовой базой.

К счастью, существует еще один шаблон проектирования, называемый "Нулевой объект", который будет здесь уместен. Основная идея заключается в том что там, где у нас возвращается `NULL`, когда мы хотим создать или найти объект, мы вместо него возвращаем специальный объект, который реализует такой же интерфейс, как и желаемый объект. Его реализация обычно ничего не делает, но позволяет нашему коду корректно получать объект и продолжать с ним работу как будто он "настоящий".

Чтобы им воспользоваться, мы определим еще один нулевой поставщик службы:

```
class NullAudio: public Audio
{
public:
  virtual void playSound(int soundID) { /* Do nothing. */ }
  virtual void stopSound(int soundID) { /* Do nothing. */ }
  virtual void stopAllSounds() { /* Do nothing. */ }
};
```

Как вы видите, он реализует интерфейс службы, но на самом деле ничего не делает. Теперь изменяем соответствующим образом наш поиск:

```
class Locator
{
public:
  static void initialize() { service_ = &nullService_; }

  static Audio& getAudio() { return *service_; }

  static void provide(Audio* service) {
    if (service == NULL) {
      // Возвращение к нулевой службе.
      service_ = &nullService_;
    } else {
      service_ = service;
    }
  }

private:
  static Audio* service_;
  static NullAudio nullService_;
};
```

> Вы можете заметить что теперь мы возвращаем службу по ссылке, а не через указатель. Так как ссылки в `C++` (в теории!) никогда не могут быть равны `NULL`, возврат ссылки подсказывает пользователю кода, что он всегда может рассчитывать на получение валидного объекта.
>
> Еще одна вещь, на которую стоит обратить внимание — это то, что мы делаем проверку на `NULL` в функции `provide()`, а не на доступность. Это значит, что нам нужно, чтобы вызов `initialize()` выполнился раньше и поиск стал по умолчанию поставщиком нулевой службы. В свою очередь мы перемещаем туда ветвление из `getAudio()`, что в результате экономит нам несколько циклов при вызове.

Вызывающий код никогда не может знать, какая "настоящая" служба будет найдена, но и не должен волноваться о том, что вернется `NULL`. Он гарантированно получит валидный объект.

Это очень полезно в случае, если мы *намеренно* не находим службу. Если мы хотим временно отключить систему, у нас появился простой способ это сделать: мы просто ее не регистрируем в поставщике службы, и поиск превратится в нулевого поставщика.

> Отключение аудио может быть весьма полезным во время разработки. Так, мы освобождаем немного памяти и процессорных циклов. И что более важно, это избавляет вас от прослушивания громкого скрежета, когда мы попадаем в отладчик во время проигрывания громкого звука. Нет ничего более бодрящего по утрам, чем зацикленный двадцатисекундный скрежет на полной громкости.

Декоратор журналирования

Теперь, когда наша система превратилась в довольно удобную, давайте обсудим еще одно улучшение, которое может предложить нам этот шаблон: декорацию службы. Поясню на примере.

Во время разработки, ведение журнала произошедших интересных событий может существенно помочь в понимании того, что происходит под капотом вашего игрового движка. Если вы работаете над ИИ, вам будет интересно узнать, когда сущность изменяет состояние ИИ. Если вы звуковой программист, вас может интересовать список проигранных звуков, чтобы вы могли проверить, что они играются в правильном порядке.

Типичным решением является замусоривание кода вызовами функции `log()`. К сожалению, это меняет одну проблему на другую: теперь у нас *слишком много* сообщений. Кодеру ИИ не очень интересно какой звук играется, а звуковику не очень интересны изменения состояния ИИ, но сейчас они оба вынуждены продираться через дебри предназначенных не им сообщений.

В идеале, мы хотели бы иметь возможность включать журналирование только тех вещей, которые нас интересуют, а в финальной сборке игры журналирование вообще лишнее. Если для различных систем журналирование выглядит как служба, мы можем применить шаблон Декоратор(Decorator) [GoF]. Определим еще одну реализацию поставщика службы для аудио:

```
class LoggedAudio : public Audio
{
public:
  LoggedAudio(Audio &wrapped)
  : wrapped_(wrapped)
  {}

  virtual void playSound(int soundID) {
    log("play sound");
    wrapped_.playSound(soundID);
  }

  virtual void stopSound(int soundID) {
    log("stop sound");
    wrapped_.stopSound(soundID);
  }

  virtual void stopAllSounds() {
    log("stop all sounds");
    wrapped_.stopAllSounds();
  }


private:
  void log(const char* message) {
    // Код для журналирования сообщений...
  }

  Audio &wrapped_;
};
```

Как вы видите, он представляет собой обертку вокруг другого поставщика службы и предоставляет некий интерфейс. Он не только переправляет все вызовы настоящему поставщику службы, но и записывает в журнал каждый вызов. Если программисту захочется включить журналирование аудио, он может поступить так:

```
void enableAudioLogging()
{
  // Decorate the existing service.
  Audio *service = new LoggedAudio(Locator::getAudio());

  // Swap it in.
  Locator::provide(service);
}
```

Теперь любые вызовы к аудио службе оставят записи в журнале перед тем, как сработать как и раньше. И конечно, такое решение будет отлично работать с нашей нулевой службой, так что вы может *выключить* звук, но в журнал все равно будут

поступать сообщения, *как будто* он работает.

Архитектурные решения

Мы рассмотрели типичную реализацию, но у нас все же есть еще несколько вариантов реализации в зависимости от того, как мы ответим на несколько ключевых вопросов.

Как выполняется поиск службы?

- **Ее регистрирует внешний код:**

  Именно этот механизм использует наш пример для поиска службы и он же чаще всего применяется в играх.

  - *Он быстрый и простой*. Функция `getAudio()` просто возвращает указатель. Компилятор скорее всего превратит ее в `inline` и в результате у нас останется уровень абстракции и никакой потери в производительности.
  - *Мы управляем созданием поставщика*. Представьте себе службу для доступа к игровым контроллерам. У нас есть два конкретных поставщика: один для обычных игр и другой для сетевых. Сетевой поставщик передает ввод с контроллера по сети, так что остальная игра и удаленные игроки видят его также, как и локальный контроллер.

    Чтобы это заработало, конкретный сетевой поставщик должен знать `IP` адресс удаленного игрока. Если поиск сам создает объект, как он узнает что передать внутрь? Класс поиска не знает ничего о том, что такое сеть и еще меньше о том, что такое `IP` адресс пользователя.

    Регистрация поставщика извне решает эту проблему. Вместо того, чтобы класс создавал сам поиск, сетевой код игры создает ориентированный на работу с сетью поставщик и передает внутрь нужный `IP` адресс. Дальше он передает объект поиску, который знает только о существовании абстрактного интерфейса службы.
  - *Мы можем изменить службу во время работы игры*. В готовой игре это можно не использовать, но во время разработки это может пригодиться. Во время тестирования вы можете, например, заменить аудио службу на нулевую службу, о которой мы говорили выше и на время отключить звук прямо во время работы игры.

- *Поиск зависит от внешнего кода*. Это конечно недостаток. Любой код, обращающийся к службе предполагает, что кто-то где-то ее уже зарегистрировал. Если такой инициализации не было, у нас случится либо падение игры, либо произойдет нечто непредвиденное.

- **Привязка во время компиляции:**

  Идея в том, что процесс "поиска" на самом деле происходит во время компиляции с помощью макросов препроцессора. Примерно так:

```
class Locator
{
public:
  static Audio& getAudio() { return service_; }

private:
  #if DEBUG
    static DebugAudio service_;
  #else
    static ReleaseAudio service_;
  #endif
};
```

  Поиск службы в таком случае подразумевает несколько вещей:

  - *Это быстро*. Так как вся настоящая работа выполняется на этапе компиляции, во время выполнения ничего делать не нужно. Компилятор скорее всего сделает вызов `getAudio()` `inline` и в результате вызов получится максимально быстрым.
  - *Вы можете гарантировать наличие службы*. Так как поиск теперь обладает службой, и выбирает его во время компиляции, мы можем предположить, что если игра скомпилировалась, нам не нужно волноваться о недоступности службы.
  - *Вы не можете так просто сменить службу*. Это главный недостаток. Так как связывание происходит во время сборки, каждый раз, когда вам захочется поменять службу, вам придется перекомпилировать игру.

- **Настройка во время работы:**

  В скучных как цвет хаки землях бизнес приложений, если вы скажете "поиск службы (service locator)", будет подразумеваться именно это. Когда служба регистрируется, поиск занимается некоей магией во время работы. чтобы найти нужную зарегистрированную реализацию.

> Отражение (Reflection) — это способность языка программирования работать с системой типов во время выполнения. Например, мы можем найти класс по его имени, найти его конструктор и вызвать его для создания экземпляра.
>
> Языки с динамической типизацией, такие как `Lisp`, `Smalltalk` и `Python` имеют такую способность от рождения, но даже новые статические языки, такие как `C#` или `Java` тоже их поддерживают.

Обычно, это означает загрузку файла настройки , которая описывает какую службу выбирать и затем выполняет отражение экземпляра этого класса во время выполнения. Этот процесс порождает следующие последствия.

- *Мы можем выполнять обмен служб без перекомпиляции*. Это значительно более гибкий подход, чем связывание на этапе компиляции, но недостаточно гибкий как тот, который позволяет смену службы во время выполнения.
- *Службы могут изменять даже непрограммисты*. Это полезно, когда дизайнер хочет запустить игру с некоторыми отключенными возможностями, но ради этого не хочется лезть в код игры. (Точнее говоря *программистам* не хочется этим заниматься).
- *Одна кодовая база может поддерживать несколько конфигураций одновременно*. Так как поиск службы полностью убирается из кодовой базы, мы можем использовать один и тот же код для одновременной поддержки нескольких конфигураций служб.

  Это одна из причин, почему эта модель преобладает в промышленной web-разработке: вы можете разместить одно и то же приложение на разных серверах, лишь немного изменив файл настройки. В играх это не так полезно. Консольное железо обычно стандартизировано и даже `PC` игры создаются в расчете на определенную конфигурацию.
- *Это сложно*. В отличие от предыдущего решения, это довольно тяжеловесное. Вы можете создать систему настройки, возможно написать код для загрузки и парсинга файла и даже *предпринять какие-то меры* для локализации службы. Но только вот время, потраченное на все это — это время, не потраченное на саму игру.
- *Поиск службы требует времени*. И теперь наша улыбка окончательно исчезнет. Переход к конфигурированию во время выполнения означает, что нам придется тратить на это циклы процессора для поиска службы. Это можно минимизировать с помощью кэширования, но при первом

использовании службы игра все равно будет тратить время на ее поиск. Игровые разработчики ненавидят тратить циклы процессора на вещи, не улучшающие впечатления игрока от игрового процесса.

Что произойдет если служба не будет найдена?

- **Пусть это обрабатывает пользователь:**

  Такое решение проще, чем потратить доллар. Если поиск не может найти службу, он возвращает `NULL`. Сюда входит:

  - *Он позволяет пользователю определить, как обрабатывать ошибку.* Некоторые пользователи могут расценивать невозможность найти службу как критическую ошибку, которая должна привести к остановке игры. Другие могут ее проигнорировать и продолжить работу. Если поиск не может определить общую политику для всех случаев, то передача ошибки вниз по дереву наследования позволит каждому решить, как на нее реагировать самостоятельно.
  - *Пользователи службы должны обрабатывать ошибку.* Конечно, из этого следует и то, что каждое место вызова должно выполнять проверку найдена ли служба. Если почти все они будут обрабатывать ошибку одинаково, мы получим очень много дублирующего кода по всей кодовой базе. И если, хотя бы в одном из этих мест, мы забудем вставить проверку, игра может упасть.

- **Остановка игры:**

  Я упоминал, что мы можем *утверждать* (prove), что служба всегда будет доступна во время компиляции, но это еще не значит, что мы можем *заявлять* (declare), что его наличие является частью соглашения с поиском во время выполнения. Проще всего это сделать с помощью `assert`:

```
class Locator
{
public:
  static Audio& getAudio() {
    Audio* service = NULL;

    // Здесь находится код для поиска службы...

    assert(service != NULL);
    return *service;
  }
};
```

Если служба не найдена, игра останавливается перед тем как ее попробует использовать следующий код. Вызов assert() не решает здесь проблему ненайденной службы, но зато показывается где произошла ошибка. Мы как будто утверждаем "Невозможность найти службу является багом поиска".

> Функция `assert()` подробно объясняется в главе Синглтон (singleton)

Так что же это нам дает?

- *Пользователю не нужно обрабатывать ненайденную службу*. Так как одна и та же служба может использоваться в сотне мест, нам нужно написать гораздо меньше кода. Утверждая, что поиск всегда возвращает службу, мы ограждаем пользователя от необходимости этим заниматься.
- *Игра останавливается, если служба не найдена*. Если служба действительно не будет найдена, игра остановится. Это хорошо, потому что заставит нас обратить внимание на баг, не дающий службе найтись (если например код инициализации не был вызван там где следует), но это будет тормозить всех остальных до тех пор, пока ошибка не будет исправлена. Для большой команды разработчиков это может вылиться в длительный перерыв в работе.

- **Возвращение нулевой службы:**

  Мы показали такое усовершенствование в примере реализации. Его использование означает что:

  - *Пользователям не нужно обрабатывать пропавшую службу*. Как и в предыдущем случае, мы проверяем, чтобы у нас всегда возвращалась валидная служба, упрощая код, использующий службу.
  - *Игра будет продолжать работать, если служба не доступна*. Это одновременно и зло и благо. Это будет полезно, если мы хотим, чтобы игра работала, даже когда служба недоступна. Например, это полезно при работе большой командой когда функционал, над которым мы работаем зависит от другой системы, которая еще не готова.

    Недостаток заключается в том, что нам будет сложнее отлаживать *случайно* ненайденную службу. Предположим игра использует службы для доступа к данным и затем на основе этих данных принимает решение. Если мы не смогли зарегистрировать настоящую службу, и код вместо нее получит нулевую службу, игра не будет вести себя так, как мы ожидаем. И понадобится немало работы, чтобы отследить ошибку, заключающуюся в том, что служба была недоступна когда нам понадобится.

> Мы можем облегчить свою участь, если поместим в нулевую службу отладочное сообщение, выводимое при ее использовании.

Среди всех этих вариантов, я чаще всего наблюдаю простое утверждение (asserting) того, что служба найдена. К тому времени, как игра выходит, ее тестируют на самых разнообразных конфигурациях. Шанс того, что служба после этого не будет найдена, крайне мал.

В большой команде я советую вам воспользоваться нулевой службой. Ее не сложно реализовать, и она убережет вас от простоя, когда служба недоступна. А еще вы легко сможете отключить службу на время, если она содержит ошибки или просто вам мешает.

Какова область видимости службы?

До сих пор мы предполагали, что поиск предоставляет доступ к службе *всем*, кому это нужно. Несмотря на то, что это самый распространенный способ применения шаблона, есть и еще один вариант ограничения доступа к самому классу и его наследникам:

```
class Base
{
  // Код для поиска службы и установки service_...

protected:
  // Классы наследники могут использовать службу
  static Audio& getAudio() { return *service_; }

private:
  static Audio* service_;
};
```

Таким образом, доступ к службе ограничивается классами, унаследованными от `Base`. Мы в любом случае в выигрыше:

- **Если доступ глобальный:**
  - *Вся кодовая база имеет возможность воспользоваться службой*. Большая часть служб должна существовать в едином экземпляре. Позволив всей кодовой базе получать доступ к службе, мы можем избежать появления экземпляров поставщиков по всей кодовой базе, потому что они не смогут обратиться к "настоящему".

    - *Мы утрачиваем контроль над тем, где используется служба*. Это очевидная плата за объявление чего либо глобальным: кто угодно может к ней обратиться. В главе Синглтон (singleton) GoF полно страшных историй о том, к чему приводит глобальность.

- **Если доступ ограничен конкретным классом:**

    - *Мы управляем связностью*. Это главное преимущество. Ограничивая доступ к службе до ветви дерева иерархии, мы позволяем несвязанным системам оставаться несвязанными.

    - *Это может привести к дублированию усилий*. Потенциальный недостаток заключается в том, что если нескольким сторонним классам потребуется доступ к службе, им придется устанавливать отдельную с ней связь. Какой бы процесс для поиска или регистрации службы не использовался, в этих классах будет иметь место дублирование.

      (Еще одна альтернатива заключается в изменении иерархии класса так, чтобы сосредоточить все это в базовом классе, но такие сложности не обязательно стоят затраченных на них усилий.)

Моя общая рекомендация заключается в том, что если служба ограниченна в игре какой-то одной областью, область ее видимости стоит этим классом и ограничить. Например, служба получения доступа к сети может быть ограничена сетевыми классами. Службы, используемые в более широком смысле, такие как журналирование, могут быть и глобальными.

Смотрите также

- Шаблон *Поиск службы* во многом является родственником Синглтона (singleton) GoF, так что вам стоит рассмотреть оба варианта и выбрать лучший.
- Фреймворк Unity использует этот шаблон вместе с шаблоном Компонент (Component) в методе `GetComponent()`.
- Фреймворк Microsoft's XNA содержит этот шаблон внутри своего главного класса `Game`. У каждого экземпляра есть объект `GameServices`, который можно использовать для регистрации и поиска любых типов служб.

Шаблоны оптимизации

Тогда как рост производительности железа уже давно удовлетворил потребности большинства программ, игры до сих пор остаются исключением. Игроки всегда желают получить еще более богатый, реалистичный и захватывающий игровой опыт. Игры пытаются любым способом привлечь внимание игрока и те, кто выжимает из железа больше чем остальные, зачастую выигрывают.

Оптимизация для увеличения производительности — это глубокое искусство, затрагивающее все аспекты программирования. Низкоуровневые программисты учатся работать с самыми незначительными особенностями архитектуры железа. В то же время разработчики алгоритмов разрабатывают математические аппараты для повышения их эффективности.

Здесь я затрону несколько среднеуровневых шаблонов, которые часто используются для ускорения работы игры. Локализация данных (Data Locality) познакомит вас с современной иерархией организации памяти и как ее можно использовать в своих целях. Шаблон Грязный флаг (Dirty Flag) поможет избавиться от лишних вычислений, а Пул объектов (Object Pools) поможет избежать лишнего выделения памяти. Разделение пространства (Spatial Partition) ускорит виртуальный мир и размещение в нем его обитателей.

Шаблоны

- Локализация данных (Data Locality)
- Грязный флаг (Dirty Flag)
- Пул объектов (Object Pool)
- Разделение пространства (Spatial Partition)

Локальность данных (Data Locality)

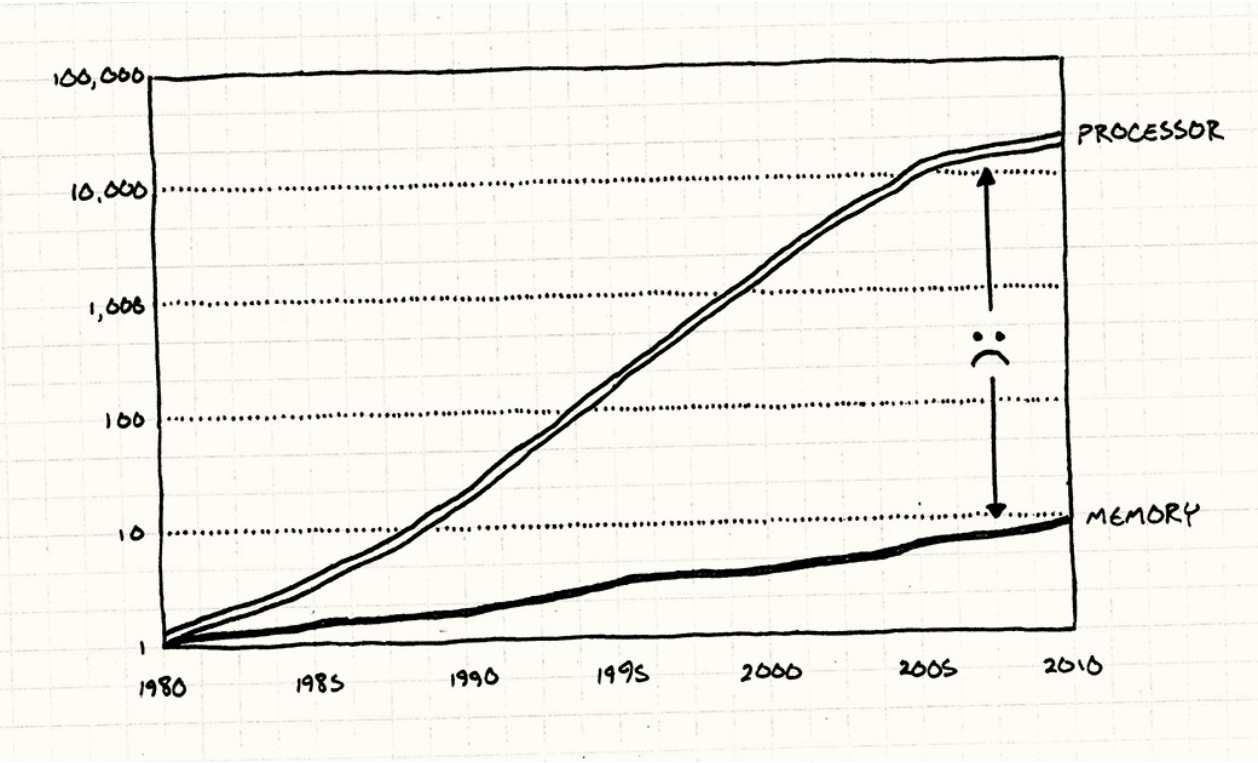
Задача

Ускорение доступа к памяти с помощью более удобного для кэширования процессором размещения данных.

Мотивация

Нам лгали. Нам продолжают демонстрировать графики, где скорость процессоров растет из года в год, как будто закон Мура — это не историческое наблюдение, а закон божий. Не шевеля и пальцем мы, компьютерные люди, видим как некая сила магическим образом позволяет работать нашим программам все быстрее.

Чипы *становятся* быстрее (хотя сейчас мы приближаемся к плато на графике), но производители железа кое-что утаивают. Мы конечно можем теперь *обрабатывать* данные быстрее, но мы не можем *ускорить* сами данные.



> Процессоры и память по отношению к состоянию на `1980` год. Как вы видите, процессоры сделали громадный скачок и уперлись в потолок, а память все еще тащится позади.
>
> Данные взяты из *Computer Architecture: A Quantitative Approach* Джона Л. Хеннеси, Девида А. Паттерсона, Андре С.Арчи-Дюссо по мотивам презентации Тони Альбрехта “Pitfalls of Object-Oriented Programming”.

Для того, чтобы ваш супер быстрый процессор мог начать перемалывать гору вычислений, ему нужно получить данные из оперативной памяти и поместить в регистры. А как вы видите, память до сих пор не смогла угнаться за ростом скорости процессоров. Даже близко не смогла.

На современном оборудовании вам понадобятся *сотни* циклов, чтобы извлечь данные из памяти. И если большинство инструкций будут требовать данных, а их получение будет требовать сотен циклов, то разве не будет наш процессор проводить 99% своего рабочего времени в ожидании данных?

На самом деле, процессоры действительно останавливаются и ожидают память удивительно долго, но не все так плохо. Чтобы во всем этом разобраться, давайте отправимся в землю очень длинных аналогий...

> Это называется "случайный доступ к памяти", потому что в отличие от дисковых носителей, вы можете теоретически получать данные из любого места с одинаковой скоростью. И вам не нужно волноваться о последовательном считывании данных, как при работе с диском.
>
> И в конце концов вы не должны. Как мы сейчас увидим, оперативная память совсем не настолько хорошо обеспечивает случайный доступ.

Хранилище данных

Представьте себе что вы клерк в маленьком офисе. Ваша работа заключается в запросе коробок с бумагами и выполнении с ними какой-то бухгалтерской работы — складывать всякие числа или что-то в таком духе. Вы работаете с коробками подписанными специальным сакральным образом, понятным другим бухгалтерам.

> Наверное не стоило выбирать в качестве аналогии работу, о которой я ничего не знаю.

Благодаря тяжелой работе, врожденным способностям и стимуляторам, вы можете справиться с одной коробкой скажем за минуту. Но есть одна проблема. Все коробки хранятся в хранилище, расположенном в соседнем здании. Чтобы взять коробку, вам

нужно попросить кладовщика принести ее вам. Он идет на склад, садится в грузоподъемник и едет по проходам между стойками, пока не доберется до нужной коробки.

Эта работа может занять у него целый день. И в отличие от вас, его никогда не называли работником месяца. Это значит, что как бы быстро вы не работали, больше чем одну коробку в день вы не получите. Все остальное время вы будете сидеть и думать о тщетности своей жизни, растрачиваемой на такую неблагодарную работу.

Но однажды появляется группа промышленных дизайнеров. Их задача состоит в повышении производительности. В духе ускорения работы сборочных линий. После того, как они понаблюдают несколько дней за вашей работой, они смогут сделать следующие выводы:

- Чаще всего, когда вы заканчиваете работу с одной коробкой, следующая коробка, за которую вы беретесь находится на складе на той же полке, рядом с предыдущей.
- Использовать погрузчик для перевозки всего одной коробки с бумагами — довольно глупая затея.
- У вас в офисе есть никем не используемый угол.

> Технически это называется локальность ссылок (locality of reference).

Оптимизаторы приходят к мудрому решению. Когда вы будете запрашивать у парня с погрузчиком коробку, он будет снимать со стеллажа целый поддон с коробками. Он захватит не только нужную вам коробку, но и находящиеся рядом с ней. Он конечно не знает, понадобятся вам они или нет (и в его обязанности это и не входит); он просто снимает со стеллажа столько, сколько помещается на поддоне.

Нагруженный поддон он привозит вам. Игнорируя технику безопасности, он заезжает на погрузчике в офис и оставляет поддон в углу офиса.

Когда вам понадобится новая коробка, вы теперь первым делом должны проверить, не находится ли она уже на поддоне в вашем офисе. Если, да —то это просто отлично! Вам придется потратить всего несколько секунд, чтобы взять ее и продолжить работу с цифрами. Если на поддоне окажется пятьдесят коробок и они все вам нужны —значит в этот день вы сделаете в пятьдесят раз больше работы.

Но вот если вам нужна коробка, которой на поддоне нет, вы возвращаетесь к старому сценарию. Так как у вас в офисе помещается только один поддон, вашему приятелю со склада придется забрать тот, что лежит у вас в офисе и привезти новый.

Поддон для процессора

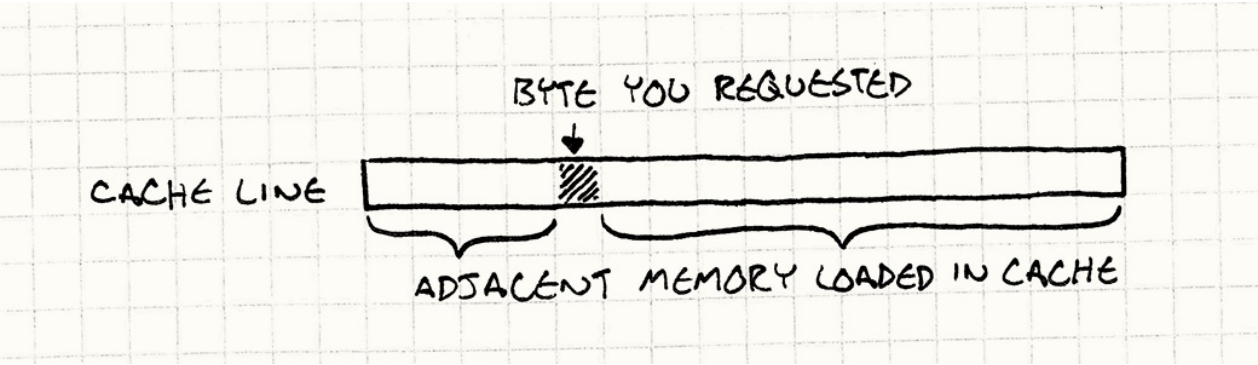
В целом именно так и работают современные процессоры. Если вы еще не поняли — вы выступаете в роли процессора. Ваш стол — это регистры процессора, а коробка с бумагами — данные, которые в него помещаются. Склад — это оперативная память вашего компьютера, а назойливый кладовщик — это шина, по которой данные перемещаются между основной памятью и регистрами.

Если бы эта глава писалась тридцать лет назад, на этом аналогия и заканчивалась бы. Но как только чипы начали стремительно становиться быстрее, а память — *нет*, разработчики аппаратного обеспечения начали искать решение проблемы. Решение, которое они предложили — это *кэш процессора*.

Современные компьютеры имеют небольшую память, размещенную прямо в процессоре. Процессор может получать из нее данные гораздо быстрее, чем из основной памяти. Она не слишком большая по объему, потому что много в чип не поместится, и к тому же быстрая статическая память (static RAM или SRAM) гораздо дороже обычной.

> В современном железе обычно присутствует кэш нескольких уровней. Вы наверняка слышали как их называют “L1”, “L2”, “L3” и т.д. Каждый уровень больше и медленнее предыдущего. В этой главе мы не будем вспоминать о том, что память организована в такую иерархию, но вам все равно стоит об этом знать.

Эта маленькая порция памяти называется *кэш* (точнее говоря, находящийся на чипе *кэш первого уровня*) и в нашей аналогии работает как поддон с коробками. И когда вашему процессору потребуется байт данных из памяти, он автоматически извлекает последовательную порцию данных из памяти — обычно 64 или 128 байт и помещает ее в кэш. Эта порция данных называется блок данных или *строка кэша* (*cache line*).



Если следующий нужный вам байт данных окажется в том же блоке, процессор считывает его прямо из кэша, что *гораздо* быстрее, чем обращение к оперативной памяти. Удачное нахождение порции данных в кэше называется *попаданием в кэш* (*cache hit*). Если их там найти не удается, и нужно обращаться к основной памяти — это *кэш-промах* (*cache miss*).

> В аналогии я приукрасил одну деталь (по крайней мере). В вашем офисе есть место только для одного поддона, т.е. одна строка кэша. Настоящий кэш содержит несколько строк кэша. Детали реализации выходят за рамки рассмотрения книги, но если вам интересно — советую погуглить по запросу "ассоциативность кэша" ("cache associativity").

Когда происходит кэш-промах, процессор *глохнет*: он не может выполнить следующую инструкцию, потому что у него нет для этого данных. И он будет сидеть так и скучать на протяжении нескольких сотен циклов, пока выборка данных не завершится. Наша задача — избежать этого. Представьте себе, что вы хотите оптимизировать критически важный для производительности фрагмент кода, выглядящий следующим образом:

```
for (int i = 0; i < NUM_THINGS; i++)
{
  sleepFor500Cycles();
  things[i].doStuff();
}
```

Что вам хочется изменить в этом коде в первую очередь? Совершенно верно. Убрать этот бесполезный и затратный вызов функции. Этот вызов эквивалентен цене кеш-промаха для производительности. Каждый раз, когда вас выкидывает в основную память, вы вставляете в свой код задержку.

Что? Производительность данных?

Когда я начал работу над этой главой, я потратил некоторое время на написание маленьких подобных игровых программ, в которых можно было увидеть наилучший и наихудший сценарии использования кэша. Мне хотелось получить тест, который демонстрировал бы насколько плачевны результаты разрушения кэша.

Когда примеры заработали, я был крайне удивлен. Я предполагал что разница будет заметна, но даже не предполагал насколько она велика, пока не увидел это своими глазами. Я написал две программы, выполняющие *абсолютно одинаковые* вычисления. Единственное отличие заключалось в количестве допускаемых кэш-промахов. Более медленная программа работала в *пятьдесят раз* медленнее.

> Здесь конечно есть много подводных камней. На самом деле у разных архитектур компьютеров кэш организован по разному и результаты, полученные на моей машине могут отличаться от полученных на вашей, а специализированные игровые консоли вообще сильно отличаются от настольных `PC`, не говоря уже о мобильных устройствах.
>
> Так что оценка может варьироваться.

Это просто открыло мне глаза. Я привык думать о производительности категориями кода, а не данных. Сами байты не бывают быстрыми или медленными — это просто статичные состояния. Но, из-за того, что у нас есть кэш, *способ организации данных напрямую влияет на производительность*.

Сложность заключается в том, чтобы подать это таким образом, чтобы нам хватило одной главы. Оптимизация для кэша — тема обширная. А я ведь даже еще ничего не рассказал о *кэшировании инструкций* (*instruction caching*). Вы ведь помните, что код тоже находится в памяти и должен быть загружен в процессор перед тем как будет выполнен. Кто-то более сведующий в теме мог бы написать об этом целую книгу.

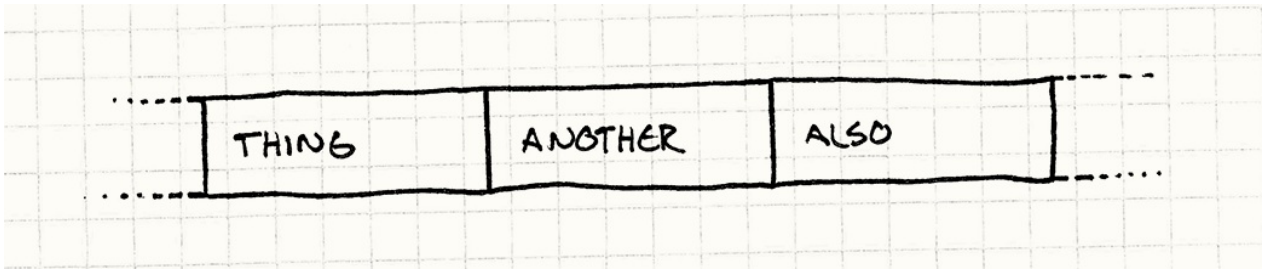
> И такая книга действительно существует: Data-Oriented Design Ричарда Фабиана.

Так как вы уже читаете *эту* книгу, я продемонстрирую вам несколько техник, которые помогут вам начать думать о том, как структуры данных влияют на производительность.

Все сводится к одной простой мысли: когда чип выполняет чтение из памяти, он считывает целую строку кэша. Чем больше вы уместите в эту строку кэша, тем большую скорость получите. Так что наша цель заключается в том, чтобы *организовать структуры данных таким образом, чтобы обрабатываемые вами вещи находились в памяти одна за другой*.

> Здесь есть одно существенное допущение: один поток. Если вы изменяете близкорасположенные данные в разных потоках — вам выгоднее иметь их в *разных* строках кэша. Если два потока пытаются изменить данные в одной строке кэша, обеим ядрам придется заниматься дорогостоящей синхронизацией их кэша.

Другими словами, если код перемалывает сначала `Thing`, потом `Another`, и потом `Also`, вам желательно расположить их в памяти таким образом:



Обратите внимание что это не *указатели* на `Thing`, `Another`, и `Also`. Это сами находящиеся в них данные, расположенные одни за другими. Как только процессор считывает `Thing`, он начинает считывание `Another` и потом `Also` (в зависимости от размера данных и размера нашей строки кэша). Когда вы начинаете работать над следующим объектом, он уже находится в кэше. Чип счастлив и вы счастливы.

Шаблон

Современные процессоры обладают **кэшем для ускорения доступа к памяти**. Доступ к памяти, **находящейся рядом с той, к которой мы только что обращались — значительно быстрее**. Используйте это свойство для ускорения работы с помощью **увеличения локальности данных** — размещение данных в памяти **последовательно, в порядке их обработки**.

Когда использовать

Главное правило при работе с любыми методами оптимизации — это использование их только там, *где есть проблемы с производительностью*. Не тратьте время на оптимизацию редко вызываемых частей вашей кодовой базы. Оптимизация не нужна в том случае, если она будет просто усложнять вам жизнь, потому что результат всегда будет сложнее и менее гибкий.

Что касается конкретно этого шаблона, вам стоит для начала удостовериться, что ваша проблема с производительностью *вызвана именно кэш-промахами*. Если ваш код тормозит по другой причине, этот шаблон вам не поможет.

Проще всего выполнить профилирование вручную, просто вставив в код механизм для замера времени, прошедшего между попаданиями в две точки в коде. Желательно с помощью точного таймера. Чтобы обнаружить кэш-промахи, вам нужно что-то более сложное. Ведь вам нужно увидеть сколько кэш-промахов у вас есть и где они происходят.

К счастью существуют профайлеры, способные нам помочь. Стоит потратить время на их освоение и изучение выдаваемых ими чисел (на удивление сложных), перед тем, как приступать к хирургическому вмешательству в свои данные.

> К сожалению, большинство из этих инструментов совсем не дешевы. Но если вы работаете в команде консольных разработчиков, у вас наверняка уже есть лицензия.
>
> Если нет — я могу порекомендовать отличную бесплатную альтернативу — Cachegrind. Он запускает вашу программу на симуляторе процессора с иерархией кэша и потом показывает, что происходило с кэшем.

Как было сказано выше, кэш-промахи *влияют* на производительность игры. И хотя вам не стоит впустую тратить время на предварительную оптимизацию для работы с кэшем, не забывайте о кэше, когда будете проектировать свои структуры данных.

Имейте в виду

Одним из отличительных признаков архитектуры программного обеспечения является *абстракция*. Большая часть этой книги посвящена шаблонам, снижающим связность между отдельными частями кода, так чтобы их было проще изменять по отдельности. В объектно-ориентированных языках это обычно подразумевает использование интерфейсов.

В `C++` использование интерфейсов включает в себя доступ к объектам через указатели или ссылки. Но переход по указателю означает прыжок в памяти, что в свою очередь приводит к кэш-промаху, которые этот шаблон призывает избегать.

> Вторая часть интерфейса — это *виртуальный вызов методов*. Это требует от процессора заглядывать в виртуальную таблицу и искать там указатель на настоящий вызываемый метод. И опять мы сталкиваемся с указателем, что приводит к кэш-промахам.

Для того, чтобы применить этот шаблон, вам нужно пожертвовать частью вашей драгоценной абстракции. Чем больше вы будете подчинять свою архитектуру локальности данных, тем больше вам придется пожертвовать наследованием и интерфейсами и предоставляемым ими удобством. Это не серебрянная пуля, а разумный обмен. И в этом и заключается весь интерес!

Пример кода

Если вы все таки решитесь спуститься в кроличью нору оптимизации для обеспечения локальности данных, вы обнаружите, что нарезать ваши данные и структуры так, чтобы процессору было их легче считывать, можно бесчисленным количеством способов. Для начала я покажу вам пример каждого из наиболее часто используемых способов организации данных. Мы рассмотрим их в контексте определенной части игрового движка, но (как и в случае с любым другим шаблоном) имейте в виду, что общий принцип применим везде, где он уместен.

Последовательные массивы

Начнем с Игрового цикла (Game Loop), обрабатывающего множество игровых сущностей. Эти сущности подразделяются на различные области — `AI`, физику и рендеринг с помощью шаблона Компонент (Component). Вот класс сущности `GameEntity`:

```
class GameEntity
{
public:
  GameEntity(AIComponent* ai,
             PhysicsComponent* physics,
             RenderComponent* render)
    : ai_(ai), physics_(physics), render_(render)
  {}

  AIComponent* ai() { return ai_; }
  PhysicsComponent* physics() { return physics_; }
  RenderComponent* render() { return render_; }

private:
  AIComponent* ai_;
  PhysicsComponent* physics_;
  RenderComponent* render_;
};
```

Каждый компонент содержит сравнительно небольшое количество состояний, возможно чуть больше, чем несколько векторов или матриц и метод для их обновления. Детали здесь значения не имеют, но можно представить нечто типа:

> Как следует из имени, это пример шаблона Метод обновления (Update Method). Даже `render()` реализует этот шаблон, просто с другим именем.

```
class AIComponent
{
public:
  void update() { /* Обработка и изменение состояния... */ }

private:
  // Цели, настроение, и т.д.. ...
};

class PhysicsComponent
{
public:
  void update() { /* Обработка и изменение состояния... */ }

private:
  // Твердое тело, скорость, масса, и т.д. ...
};

class RenderComponent
{
public:
  void render() { /* Обработка и изменение состояния... */ }

private:
  // Сетка, текстуры, шейдеры, и т.д....
};
```

Игра поддерживает большой массив указателей на все сущности в игровом мире. На каждом обороте игрового цикла мы вынуждены делать следующее в определенном порядке:

1. Обновлять компоненты `AI` для всех сущностей.
2. Обновлять для них физические компоненты.
3. Выполнять их рендеринг с помощью компонентов рендеринга.

Многие игры реализуют это следующим образом:

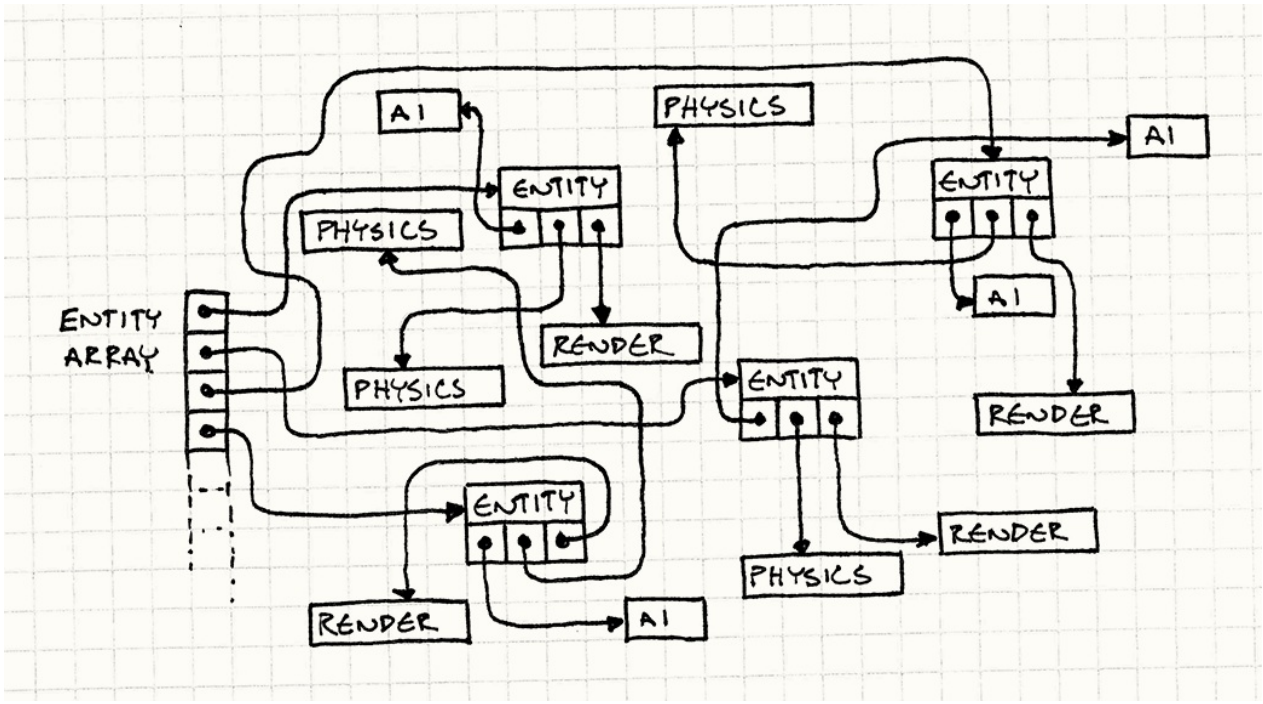
```
while (!gameOver)
{
  // Обрабатываем AI.
  for (int i = 0; i < numEntities; i++) {
    entities[i]->ai()->update();
  }
```

```
  // Обновляем физику.
  for (int i = 0; i < numEntities; i++) {
    entities[i]->physics()->update();
  }

  // Рисуем на экране.
  for (int i = 0; i < numEntities; i++) {
    entities[i]->render()->render();
  }

  // Остальные задачи игрового цикла...
}
```

До того, как вы познакомились с процессорным кэшем, это выглядит совершенно безобидно. Но теперь вы наверняка видите, что что-то здесь не так. Этот код не просто ломает кэш, он просто заходит к нему со спины и избивает до полусмерти. Смотрите что делается:

1. Массив игровых сущностей *указывает* на них, так что к каждому элементу мы должны переходить через указатель. Это кэш-промах.
2. Сама игровая сущность содержит указатели на компоненты. Еще один кэш-промах.
3. Далее мы обновляем компонент.
4. И теперь возвращаемся к шагу один и повторяем *для каждого компонента каждой сущности в игре*.

Самое же страшное заключается в том, что мы не имеем ни малейшего представления о том, как эти объекты располагаются в памяти. Мы полностью полагаемся на милость менеджера памяти. А так как сущности постоянно создаются и удаляются, они скорее всего будут размещаться совершенно хаотично.



> На каждом кадре игровой цикл вынужден переходить по каждой из этих стрелок чтобы добраться к нужным ему данным.

Если вашей целью является организация дешевого безумного тура переходов по адресному пространству игру с названием в духе "256 мегабайт за четыре ночи!" —это отличная идея. Но нашей целью является ускорение работы игры и бесцельное брожение по всей памяти — не тот способ, который нам в этом поможет. Помните функцию `sleepFor500Cycles()` ? Так вот, в этом коде она вызывается *постоянно*.

> Существует специальный термин для бесполезных переходов по указателям —"гонка указателей" (pointer chasing) и, в отличие от названия, само это явление совсем не веселое.

Поступим умнее. Нашим первым наблюдением станет то, что единственная причина, по который мы переходим по указателю к сущности —это немедленный переход еще по *одному* указателю, чтобы получить компонент. Сама `GameEntity` не имеет интересных состояний и полезных методов. Ваш игровой цикл занимается исключительно компонентами.

Вместо громадного созвездия раскиданных по всему адресному пространству сущностей и компонентов, мы попробуем вернуться на Землю. У нас будет большой массив для каждого типа компонентов: плоский массив `AI` компонентов, еще один для физики и еще один для рендеринга.

Примерно так:

```
AIComponent* aiComponents =
    new AIComponent[MAX_ENTITIES];
PhysicsComponent* physicsComponents =
    new PhysicsComponent[MAX_ENTITIES];
RenderComponent* renderComponents =
    new RenderComponent[MAX_ENTITIES];
```

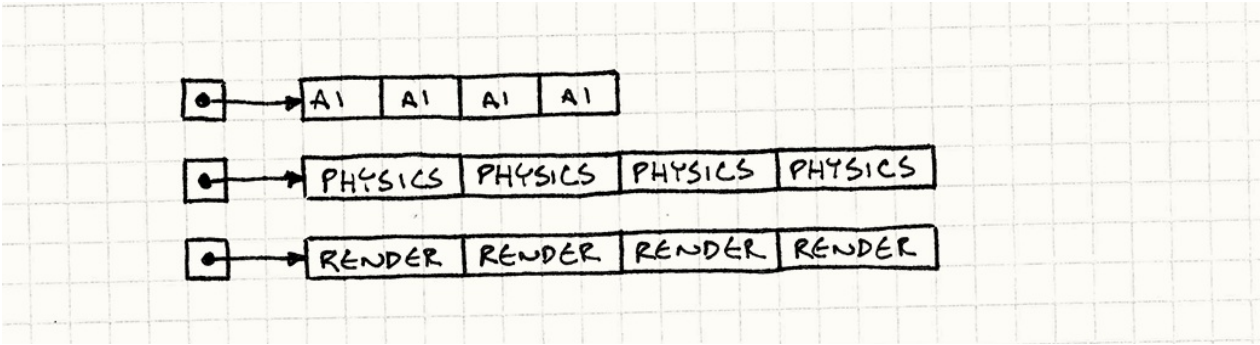
> Моя наименее любимая часть использования компонентов — это то, что слово "компонент" такое длинное.

А сейчас я вас удивлю и скажу, что массив *компонентов* не состоит из *указателей на компоненты*. Здесь у нас находятся именно данные, байт за байтом. Теперь игровой цикл может выполнять их обход напрямую.

```
while (!gameOver) {
  // Обработка AI.
  for (int i = 0; i < numEntities; i++) {
    aiComponents[i].update();
  }

  // Обработка физики.
  for (int i = 0; i < numEntities; i++) {
    physicsComponents[i].update();
  }

  // Отрисовка на экране.
  for (int i = 0; i < numEntities; i++) {
    renderComponents[i].render();
  }

  // Остальные задачи игрового цикла...
}
```

> Составить верное впечатление о том, что мы здесь делаем, можно по минимальному количеству операторов -> в новом коде. Если вы хотите улучшить локальность данных, всегда смотрите от каких операций косвенности вы можете отказаться.

Мы избавились от любых гонок указателей. Вместо перескакивания по памяти, мы выполняем линейный проход по трем последовательным массивам.



В результате в голодную память процессора закачивается цельный поток байт. В моих тестах, такое изменение ускорило цикл обновления в *пятьдесят раз* по сравнению с первой версией.

Интересно, что здесь нам даже не пришлось жертвовать инкапсуляцией. Конечно, теперь наш игровой цикл выполняет обновление компонентов, а не сущностей, но раньше это было необходимо для обеспечения правильного порядка обработки. И даже сейчас сами компоненты достаточно хорошо инкапсулированы. У них есть собственные данные и методы. Мы просто изменили способ их использования.

Это совсем не значит, что мы полностью избавились от `GameEntity`. Мы можем их оставить в качестве наборов указателей на компоненты. Они просто указывают на элементы этих массивов. В других частях игры, куда вам потребуется передать концептуальную "игровую сущность", это все равно полезно. Главное, что критичный в плане производительности игровой цикл, может этим не заниматься и переходит к данным напрямую.

Упакованные данные

Представим себе, что мы делаем систему частиц. Следуя совету из предыдущего раздела, мы собрали наши частицы в одном большом последовательном массиве. Давайте упакуем их в небольшой класс менеджер:

```
class Particle
{
public:
  void update() { /* Гравитация, и т.д. ... */ }
  // Позиция, скорость, и т.д. ...
};
```

```
class ParticleSystem
{
public:
  ParticleSystem() : numParticles_(0)
  {}

  void update();

private:
  static const int MAX_PARTICLES = 100000;

  int numParticles_;
  Particle particles_[MAX_PARTICLES];
};
```

> Класс `ParticleSystem` — это пример Пула объектов (Object Pool), специально построенного для одного типа объектов.

Рудиментарный метод обновления для системы выглядит следующим образом:

```
void ParticleSystem::update()
{
  for (int i = 0; i < numParticles_; i++) {
    particles_[i].update();
  }
}
```

Но нам в свою очередь не нужно обновлять все частицы на каждом кадре. Система частиц имеет пул объектов фиксированного размера, но они не обязательно разбросаны по экрану в полном объеме. Проще всего поступить примерно так:

```
for (int i = 0; i < numParticles_; i++) {
  if (particles_[i].isActive()) {
    particles_[i].update();
  }
}
```

Добавим в `Particle` флаг, следящий за тем, используется она или нет. В нашем игровом цикле мы выполняем такую проверку для каждой частицы. При этом флаги загружаются в кэш вместе с остальными данными частиц. Если частица *не* активна, мы ее пропускаем и переходим к следующей. Оставшаяся часть частицы, которую мы загрузили пропадает.

Чем меньше у нас активных частиц, тем чаще мы буде делать такие пропуски. Чем больше их будет, тем больше кэш-промахов у нас будет между полезной работой по обновлению частиц. Если массив будет достаточно большой, а активных частиц будет много, мы снова вернемся к разрушению кэша.

Организация объектов в последовательный массив не решает нашей проблемы, если обрабатываемые нами объекты расположены не последовательно. Если он замусорен неактивными объектами, нам придется их перескакивать и мы возвращаемся к изначальной проблеме.

> Смекалистые низкоуровневые программисты наверняка заметили еще одну проблему. Выполняя `if` для каждой частицы, мы рискуем получить *ошибку предсказания переходов* (*branch misprediction*) и *остановки конвейера* (*pipeline stall*). На современных процессорах одна "инструкция" обычно занимает несколько циклов. Для того чтобы процессор постоянно был занят, инструкции *попадают на конвейер* (*pipeline*), так что следующая инструкция начинает выполняться прежде, чем закончится предыдущая.
>
> Чтобы это стало возможным, процессору приходится угадывать какая из инструкций будет выполняться следующей. Для последовательно выполняющегося кода это сделать элементарно, но если у нас есть управление потоком, все становится сложнее. Когда мы выполняем инструкции `if`, нам нужно угадать — активна ли частица и нужно ли будет вызывать для нее `update()` или нет.
>
> Чтобы ответить на этот вопрос, чип выполняет *предсказание переходов* (*branch prediction*): он смотрит, какие переходы выполнялись до этого и пытается угадать, сделаете ли вы это снова. А когда ваш цикл постоянно попадает на неактивные частицы, такое предсказание будет неверным.
>
> Когда предсказание оказывается неверным, процессору приходится выбросить инструкции, которые он уже начал обрабатывать (*очистка конвейера*(*pipeline flush*)) и начинает заново. Влияние на производительность на разных машинах может отличаться, но оно все равно достаточно велико для того, чтобы некоторые разработчики вообще старались отказываться от ветвления кода в самых горячих участках кода.

Из названия этого подраздела вы наверное уже угадали ответ. Вместо того, чтобы проверять флаг активности, мы будем использовать его для сортировки. Все активные частицы мы будем держать в начале списка. И если мы будем знать что все эти частицы активны — нам не придется выполнять никаких проверок флага.

Мы можем посчитать количество активных частиц. И тогда наш цикл обновления превратится в такую замечательную конструкцию:

```
for (int i = 0; i < numActive_; i++) {
  particles[i].update();
}
```

Теперь мы *не* перескакиваем через данные. Каждый поступающий в кэш байт используется для обработки частиц.

Конечно, я не утверждаю, что нам нужно применять `quicksort` ко всей коллекции объектов на каждом кадре. Это уничтожит результаты всего, чего мы добились. Все что мы хотим — так это сохранить массив отсортированным.

Предполагая, что массив уже отсортирован — и он таким является, когда все наши частицы неактивны — единственный момент когда он может стать неотсортированным — это тогда, когда частица становится активной или неактивной. Мы можем довольно легко отследить эти два случая. Когда частица становится неактивной, мы переходим в конец активных частиц и меняем ее с первой неактивной:

```
void ParticleSystem::activateParticle(int index)
{
  // Не должна быть активной!
  assert(index >= numActive_);

  // Меняем ее на первую неактивную частицу
  // следующую сразу за активной.
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;

  // Теперь активных на одну больше.
  numActive_++;
}
```

Чтобы сделать частицу неактивной, поступаем наоборот:

```
void ParticleSystem::deactivateParticle(int index)
{
  // Не должна быть неактивной!
  assert(index < numActive_);

  // Теперь активных на одну меньше.
  numActive_--;

  // Меняем ее на последнюю активную частицу
  // находящуюся сразу перед неактивной.
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;
}
```

Многие программисты (включая меня) выработали аллергию на перемещение объектов в памяти. Перемещение порции байт по сравнению с перемещением указателя *ощущается* более тяжеловесным. Но когда сюда добавляется стоимость *переходов* по указателю, оказывается, что интуиция нас здесь подводит. В некоторых случаях дешевле переместить данные в памяти и обеспечить тем самым наполнение кэша.

> Искренне советую вам сначала использовать *профайлер*, перед тем как заниматься подобными вещами.

Вследствие того, что наши частицы теперь отсортированы по состоянию активности, нам теперь не нужен флаг активности для каждой частицы. Его заменяет позиция частицы в массиве и счетчик `numActive_`. В результате, наши частицы стали меньше и в строку кэша их теперь уместится больше, что приведет к еще большему увеличению скорости работы.

К сожалению, не все так радужно. Как вы можете видеть из `API`, мы потеряли на этом частичку объектно-ориентированности. Класс `Particle` больше не управляет собственным состоянием. Вы больше не можете вызывать его метод `activate()`, потому что теперь он даже не знает свой индекс. Вместо этого любой код, который хочет изменить активность частицы должен обращаться к системе частиц.

В данном случае меня вполне устраивает тот факт, что `ParticleSystem` и `Particle` крепко связаны между собой. Я думаю о них как о единой *концепции*, распределенной между двумя физическими классами. Таким образом мы соглашаемся с фактом, что частицы имеют смысл *только* в контексте системы частиц. Также в этом случае порождать и уничтожать частицы будет именно сама система частиц.

Горячее/холодное разделение

Итак, вот еще один пример того, простая техника позволит нам улучшить работу с кэшем. Представим себе `AI` компонент для некоей игровой сущности. У нее есть несколько состояний: текущая проигрываемая анимация, точка назначения куда она движется, уровень энергии и т.д. — все что проверяется и изменяется на каждом кадре игры. Примерно так:

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;
};
```

Помимо этих есть и гораздо реже используемые состояния. Они хранят данные, описывающие, что выпадет в качестве лута, когда игровая сущность окажется перед нашим дробовиком. Данные лута обычно используются всего один раз в жизни сущности, как раз перед смертью.

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  // Предыдущие поля...
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

Предположим, что мы следовали предыдущему шаблону и во время обновления `AI`, выполняли проход по аккуратно упакованному линейному массиву данных. Но в перечень этих данных входят и данные о луте. В результате, каждый компонент увеличивается, а количество помещающихся в строку кэша компонентов уменьшается. У нас будет больше кэш-промахов, потому что нам придется обходить больший участок памяти. Данные о луте будут попадать в кэш для каждого компонента на каждом кадре, даже если мы и не будем их касаться.

Решение этой проблемы называется "горячее/холодное разделение" (*hot/cold splitting*). Идея заключается в разделении данных на две части. Первая хранит "горячие" данные: состояния, которые мы используем на каждом кадре. Вторая хранит "холодные" данные: все остальное, что используется значительно реже.

Горячая часть — это *основа* `AI` компонента. Это то, что мы используем настолько часто, что не хотим допускать гонки указателей для их поиска. Холодную часть компонента можно хранить в стороне, но она все равно будет нам нужна. И поэтому мы помещаем на нее указатель в горячей части данных следующим образом:

```
class AIComponent
{
public:
  // Методы...
private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;

  LootDrop* loot_;
};
```

```
class LootDrop
{
  friend class AIComponent;
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

Теперь, когда мы выполняем обход `AI` компонентов на каждом кадре, единственными данными, загружаемыми в кэш являются те, что мы активно используем (за исключением маленького указателя на холодные данные).

> Мы, конечно, можем попробовать вообще избавиться от указателя, организовав два параллельных массива для горячих и для холодных данных. Когда нам понадобятся холодные данные `AI`, мы сможем найти их во втором массиве по тому же индексу.

Как вы видите, все начинает усложняться. В моем примере достаточно очевидно, какие данные должны быть горячими, а какие холодными, но так бывает не всегда. Что, если у вас есть данные, активно используемые в одном режиме, но не нужные для других? Что, если сущности используют определенную часть данных только тогда, когда находятся в определенной части уровня?

Выполнять такую оптимизацию, словно делать нечто среднее между черной магией и прочисткой канализации. Очень легко позволить затянуть себя в этот бесконечный процесс переноса данных туда-сюда, прежде чем мы добьемся хоть какого-то эффекта. Научиться определять когда результат стоит потраченных усилий можно только на практике.

Архитектурные решения

Этот шаблон на самом деле про образ мышления: он заставляет вас думать о размещении ваших данных в памяти, как ключевой части обеспечения хорошей производительности вашей игры. Конкретные архитектурные решения могут быть самыми разными. Вы можете позволить локальности данных влиять на всю вашу архитектуру или это может быть локализованный шаблон, применяемый только к некоторым ключевым структурам данных.

Самый главный вопрос, на который вам придется ответить — это где и когда вам стоит применять этот шаблон. А вот еще несколько вопросов, о которых тоже стоит вспомнить.

> Знаменитая статья Ноэля Ллописа заставила многих людей задуматься об архитектуре игры с точки зрения использования кэша и получила название "ориентированная на данные архитектура" (*data-oriented design*).

Что вы будете делать с полиморфизмом

До сих пор мы избегали подклассов и виртуальных методов. Предположим у нас есть хорошо упакованный массив *гомогенных* объектов. При этом, мы знаем что все они одного размера. Но ведь полиморфизм и динамическая диспетчеризация тоже хорошие инструменты. Как же согласовать эти два противоречия?

- **Не используйте:**

  Проще всего ответить так: не используете подклассы или по крайней мере избегайте их применения, там где имеет смысл применять оптимизацию работы с кэшем. Культура программирования и так уже постепенно отходит от массированного применения наследования.

  > Одним из способов использовать гибкость полиморфизма без использования подклассов является шаблон Объект тип (Type Object).

  - *Это просто и безопасно.* Вы всегда знаете с каким классом работают все ваши объекты одного размера.

- *Это быстро.* Динамическая диспетчеризация подразумевает поиск метода в виртуальной таблице методов и переход по указателю к самому методу. Несмотря на то, что стоимость этой операции на различном оборудовании серьезно варьируется, вам все равно придется платить за динамическую диспетчеризацию.

  > Как обычно с абсолютной уверенностью можно утверждать только о том, что ничего абсолютного нет. В большинстве случаев компилятор `C++` требует косвенности для вызова виртуального метода. Но в некоторых случаях компилятор может выполнить *развиртуализацию* (*devirtualization*) и вызвать нужный метод статически, если знает конкретный тип получателя. Развиртуализация встречается еще чаще в языках с компиляцией на лету, таких как `Java` или `JavaScript`.

- *Это негибкий подход.* На самом деле динамическая диспетчеризация применяется потому, что она дает нам мощный инструмент настройки поведения между объектами. Если вы хотите, чтобы разные сущности в вашей игре имели собственные стили рендеринга или обладали собственными специальными движениями и атаками, вам в этом помогут именно виртуальные методы. Попытка реализовать нечто подобное с помощью кода без виртуальных методов, у вас образуется громадное месиво конструкций типа `switch`.

- **Используйте отдельный массив для каждого типа:**

  Мы используем полиморфизм, чтобы активизировать поведение объекта, тип которого нам неизвестен. Другими словами, у нас есть мешок с разными штуками и мы хотим, чтобы каждый из этих объектов делал что-то свое, когда мы ему прикажем.

  Но в таком случае сразу возникает вопрос — зачем складывать все в один мешок? Почему не организовать вместо этого отдельные гомогенные коллекции для каждого типа?

  - *Наши объекты будут плотно упакованы.* Так как все объекты в массиве одного типа — у нас не будет заполнений (padding) и других странностей.
  - *Мы можем применять статическую диспетчеризацию.* Как только ваши объекты будут разделены по типу, полиморфизм вам больше не понадобится. Вы можете использовать простые невиртуальные вызовы методов.
  - *Вы можете поддерживать множество коллекций.* Если у вас есть много разных типов объектов, поддерживать для каждого типа отдельный массив может быть сложно и накладно.

- *Вам нужно заботиться о каждом из типов*. Так как у вас для каждого типа имеется отдельная коллекция, вы не сможете убрать связность от набора классов. Часть магии полиморфизма заключается в *открытости*: код, работающий с наследниками, может быть полностью несвязан с потенциально большим набором типов, реализующих данный интерфейс.

- **Используйте коллекцию указателей:**

  Если проблема кэширования вас не волнует — это самое логичное решение. Просто организуйте массив указателей на базовый класс или тип интерфейс. Вы можете полноценно использовать полиморфизм, а объекты могут быть любого размера.

  - *Это гибкое решение*. Работающий с коллекцией код может работать с объектами любого типа, пока они поддерживают нужный вам интерфейс. Полная свобода.

  - *Это решение гораздо менее кэш-дружелюбное*. Основная причина, почему мы обсуждали другие решения — это враждебная для кэша косвенность, создаваемая указателями. Но помните, что если код не является критически важным в плане производительности, то все в порядке.

Как определять игровые сущности?

Если вы используете этот шаблон в тандеме с шаблоном Компонент (Component), у вас вполне может существовать последовательный массив компонентов, представляющий сущности вашей игры. Игровой цикл будет обходить их напрямую, так что объект для игровой сущности не слишком важен, но может быть полезен в других частях кодовой базы, где вам захочется работать с отдельной концептуальной "сущностью".

Вопрос здесь в том — как представлять сущность? Как она будет следить за своими компонентами?

- **Если игровые сущности — это классы с указателями на свои компоненты:**

  Именно так выглядел наш первый пример. Это самое традиционное `ООП` решение. У вас есть класс для `GameEntity` и у него есть указатели на свои компоненты. Так как все это просто указатели — остается только догадываться, где и как на самом деле размещаются эти компоненты в памяти.

  - *Вы можете хранить компоненты в последовательном массиве*. Так как игровой сущности все равно, где находятся ее компоненты, вы можете организовать их в хорошо упакованный массив для оптимизации из обхода.

- *Получив сущность, вы легко можете добраться до ее компонентов*. Для этого нужно просто перейти по указателю.
- *Перемещать компоненты в памяти сложно*. Когда компоненты включаются или выключаются, вы можете попробовать перемещать их внутри массива, так что активные будут находиться вначале, а неактивные в конце. Но если вы будете перемещать компонент, на который ссылается сущность — вы разрушите эту связь, если не будете действовать осторожно. Вам нужно будет обязательно обновить при этом и указатель на компонент внутри сущности.

- **Если игровые сущности — это классы с `ID` своих компонентов:**

  Сложность работы с указателями на компоненты заключается в том, что при этом их становится сложнее перемещать в памяти. Вы можете применить более абстрактную адресацию: `ID` или индекс, который можно использовать для *поиска* компонента.

  Семантику `ID` и реализацию поиска я оставляю на ваше усмотрение. Это может быть простой `ID`, хранимый внутри компонента и обход массива или более сложное решение с применением хеш таблиц, связывающих `ID` с индексом компонента в массиве.

  - *Это сложнее*. Конечно ваша система ID не сравнится по сложности с ракетостроением, но работы явно больше чем при использовании обычных указателей. Вам нужно будет ее реализовать и отладить. А еще у вас появятся накладные расходы на бухгалтерию.
  - *Это медленнее*. Сложно тягаться с сырыми указателями. Для того, чтобы получить компонент сущности, нам нужно будет выполнять поиск или какую-то операцию с хешированием.
  - *Вам потребуется доступ к "менеджеру" компонентов*. Основная идея заключается в том, что у вас есть абстрактный `ID`, идентифицирующий компонент. И вы можете использовать его, чтобы получить указатель на сам компонент. Но чтобы это сделать, вам нужен кто-то, кто может найти компонент по его `ID`. Это будет класс — обертка над сырым последовательным массивом компонентов наших объектов.

    При работе с сырыми указателями, если у вас есть сущность — значит вы можете получить и ее компоненты. В этом случае вам потребуется не только сама сущность, но и *предварительная регистрация компонентов*.

    > Вы можете подумать: "Так я применю синглтон! И проблема решена!" Ну как бы да. Только советую сначала заглянуть в эту главу.

- **Если игровые сущности представляют *из себя* всего лишь `ID`:**

  Это сравнительно новый способ, который применяется в некоторых игровых движках. Как только вы перенесете все поведение и состояние сущности из главного класса в компоненты, то что останется? Немного на самом деле. Единственное что делает сущность — это связывает вместе набор компонентов. Она нужна только для того, чтобы сообщать нам, что *этот* `AI` компонент, *этот* физический компонент и *этот* компонент рендеринга образуют игровую сущность в нашем мире.

  Важно здесь то, как компоненты взаимодействуют. Компоненту рендеринга нужно знать, где находится сущность. А эта информация может храниться в физическом компоненте. Компонент `AI` хочет перемещать сущность и для этого ему нужно приложить силу к физическому компоненту. Каждому компоненту нужно иметь возможность получить доступ к родственному компоненту, вместе с которыми они образуют сущность.

  Думаю некоторые уже догадались, что все что нам нужно — это `ID`. Вместо того, чтобы сущность знала о своих компонентах, компоненты могут знать о своей сущности. Каждый компонент может знать `ID` сущности, которой он принадлежит. Когда `AI` компоненту потребуется физический компонент своей сущности, он просто запросит физический компонент с таким же `ID` как и у него.

  Ваши *классы* сущностей полностью исчезают, заменяемые удобной оболочкой над числами.

  - *Сущности занимают мало места.* Если вам нужно передать ссылку на игровую сущность — вы просто передаете число.
  - *Сущности пустые.* Логично, что недостатком такого решения является то, что вам придется убрать из сущностей все. У вас больше нет места для хранения не компонентно-специфического состояния или поведения. Эти функции тоже возлагаются на шаблон Компонент.
  - *Вам не нужно управлять временем жизни.* Так как сущности — это значения простых типов, их не нужно явно создавать или удалять. Сущность "умирает", когда уничтожаются все ее компоненты.
  - *Поиск компонентов сущности может быть достаточно медленным.* Это та же самая проблема, как и в предыдущем решении, но немного с другой точки зрения. Чтобы найти компонент какой-либо сущности, вам нужно найти объект по `ID`. Этот процесс может быть достаточно затратным.

На этот раз поиск является критически важным для производительности. Компоненты часто взаимодействуют со своими родственниками во время обновления, так что искать их придется часто. В качестве решения можно сделать "ID" сущности индексом компонентов в массивах.

Если у каждой сущности будет одинаковый набор компонентов, то все ваши массивы с компонентами будут полностью параллельными. Компонент в слоте три в массиве `AI` компонентов будет принадлежать той же сущности, что и физический компонент в массиве три в *соответствующем* массиве.

Имейте ввиду, что вам придется *затрачивать усилия* для сохранения параллельности массивов. Это может быть сложным, если вам захочется сортировать или упаковывать их по разным критериям. У вас могут быть сущности с выключенной физикой и другие — невидимые. Поэтому не получится отсортировать физические и рендер-компоненты таким образом, чтобы массивы остались синхронизированными друг с другом.

Смотрите также

- Большая часть этой главы вращается вокруг шаблона Компонент (Component) и это определенно одна из наиболее часто используемых структур данных для оптимизации работы с кэшем. И действительно, применение компонентов этому способствует. Так как сущности обновляются по одной "области" (`AI` , физика и т.д.) за раз, разбиение их на компоненты позволяет поделить сущности на более удобные в плане работы с кэшем кусочки.

  Но это не значит, что вы можете использовать этот шаблон только с компонентами! Каждый раз, когда у вас встречается код критический в плане производительности, работающий с большим объемом данных — подумайте о локальности данных.

- "Ловушки объектно-ориентированного программирования" Pitfalls of Object-Oriented Programming[PDF] Тони Альбрехта — это, наверное, самое лучшее чтиво на тему организации структур данных в игре в максимально кэш-дружественном виде. Я встречал много людей (включая меня самого), кому эта книга помогла с улучшением производительности.

- Примерно в то же время Ноэль Ллопис написал хороший пост в своем блоге на ту же тему.

- Этот шаблон практически полностью основан на использовании последовательного массива гомогенных объектов. Время от времени вам наверняка придется добавлять или удалять из него объекты. Шаблон Пул объектов (Object Pool) как раз этому и посвящен.
- Игровой движок Artemis — один из первых и наверное самый известный из фреймворков, использующих `ID` в качестве игровой сущности.

Грязный флаг (Dirty Flag)

Задача

Избегать ненужной работы, откладывая ее до тех пор, пока не потребуется результат.

Мотивация

Во многих играх есть нечто, называемое *графом сцены* (*scene graph*). Это большая структура данных, хранящая все объекты в мире. Движок рендеринга использует его для определения кого и где на экране отрисовывать.

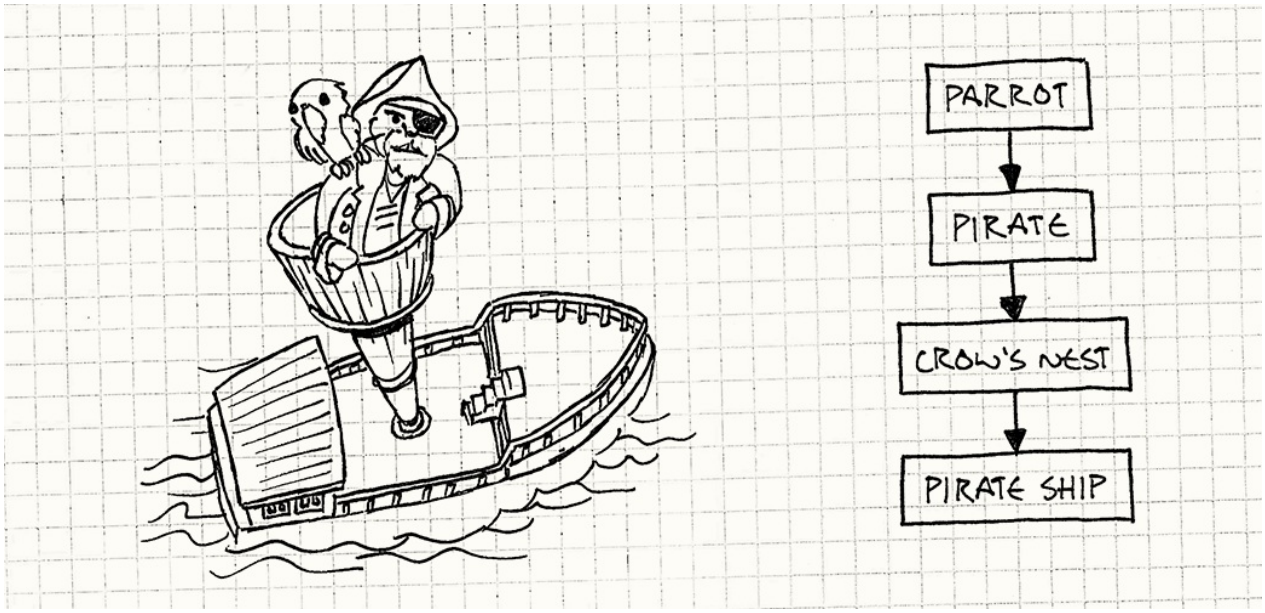
В самом простом виде граф сцены представляет из себя просто плоский список объектов. У каждого объекта есть модель или какой-то графический примитив и *трансформация*. Трансформация описывает позицию объекта, поворот и масштаб в мире. Чтобы переместить или повернуть объект, вы просто изменяете трансформацию.

> Механизм того, как хранится трансформация выходит за рамки нашего рассмотрения. Короче говоря, это матрица 4x4. Вы можете создать трансформацию, которая будет объединять две других — например перенос и потом поворот объекта с помощью перемножения двух матриц.
>
> Как и почему это работает я оставляю вам на самостоятельное изучение.

Когда рендер отрисовывает объект, он берет модель объекта, применяет трансформацию и затем рендерит его в мире. Если у нас будет просто *мешок* с объектами, а не *граф* сцены, то все так просто работать и будет.

Однако большая часть графов сцены представляет собой *иерархию*. Объект в графе может иметь родительский объект, к которому он привязан. В этом случае трансформация является относительной по отношению к позиции *родительского* объекта, а не абсолютной позицией в мире.

Представьте себе для примера игру про пиратский корабль в море. Наверху мачты находится воронье гнездо. В этом гнезде сидит пират. На плече у пирата — попугай. Локальная трансформация корабля определяет его положение в море. Позиция гнезда задается относительно корабля и т.д.



> Образчик программистского искусства!

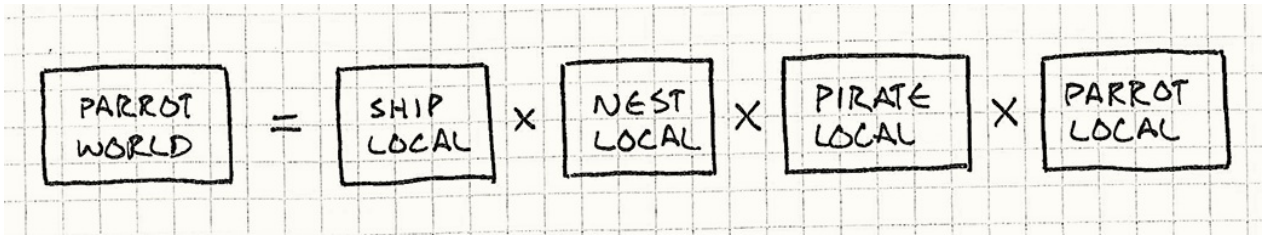
Таким образом, когда родительский объект перемещается, его дочерние объекты перемещаются вместе с ним автоматически. Если мы изменяем локальную позицию корабля, воронье гнездо, пират и попугай будут изменяться вместе с ней. Если бы мы каждый раз пересчитывали позицию вручную, чтобы ничего не сползало — это было бы настоящей проблемой.

> Хотя честно говоря, когда вы находитесь в море, вам нужно вручную корректировать свою позицию, чтобы никуда не сползти. Наверное, нужно было подобрать пример получше.

Но на самом деле для того чтобы отрисовать на экране попугая, нам нужно знать его абсолютную позицию в мире. Я бы назвал это трансформацией *локальной трансформации* объекта относительно его родителя. Чтобы отрендерить объект, нам нужна его *мировая трансформация*.

Локальные и мировые трансформации

Расчет мировой трансформации объекта довольно незамысловат: вам просто нужно обойти цепочку его родительских объектов, начиная с корня и заканчивая самим объектом, и объединить все их трансформации. Другими словами мировая трансформация попугая будет следующей:



> В простейшем случае, когда у объекта нет родителей, его локальная и мировая трансформации совпадают.

Нам понадобится мировая трансформация для каждого объекта в мире на каждом кадре, так что несмотря на то, что это всего лишь несколько умножений матриц для каждой модели, этот код является горячим и критичным в плане производительности. Хранить их в актуальном состоянии проблематично, потому что каждое смещение родительского объекта, рекурсивно влияет на трансформации всех его детей.

Проще всего вычислять трансформации на лету во время рендеринга. На каждом кадре мы рекурсивно обходим граф сцену начиная с вершины иерархии. Для каждого объекта мы вычисляем мировую трансформацию и отрисовываем его.

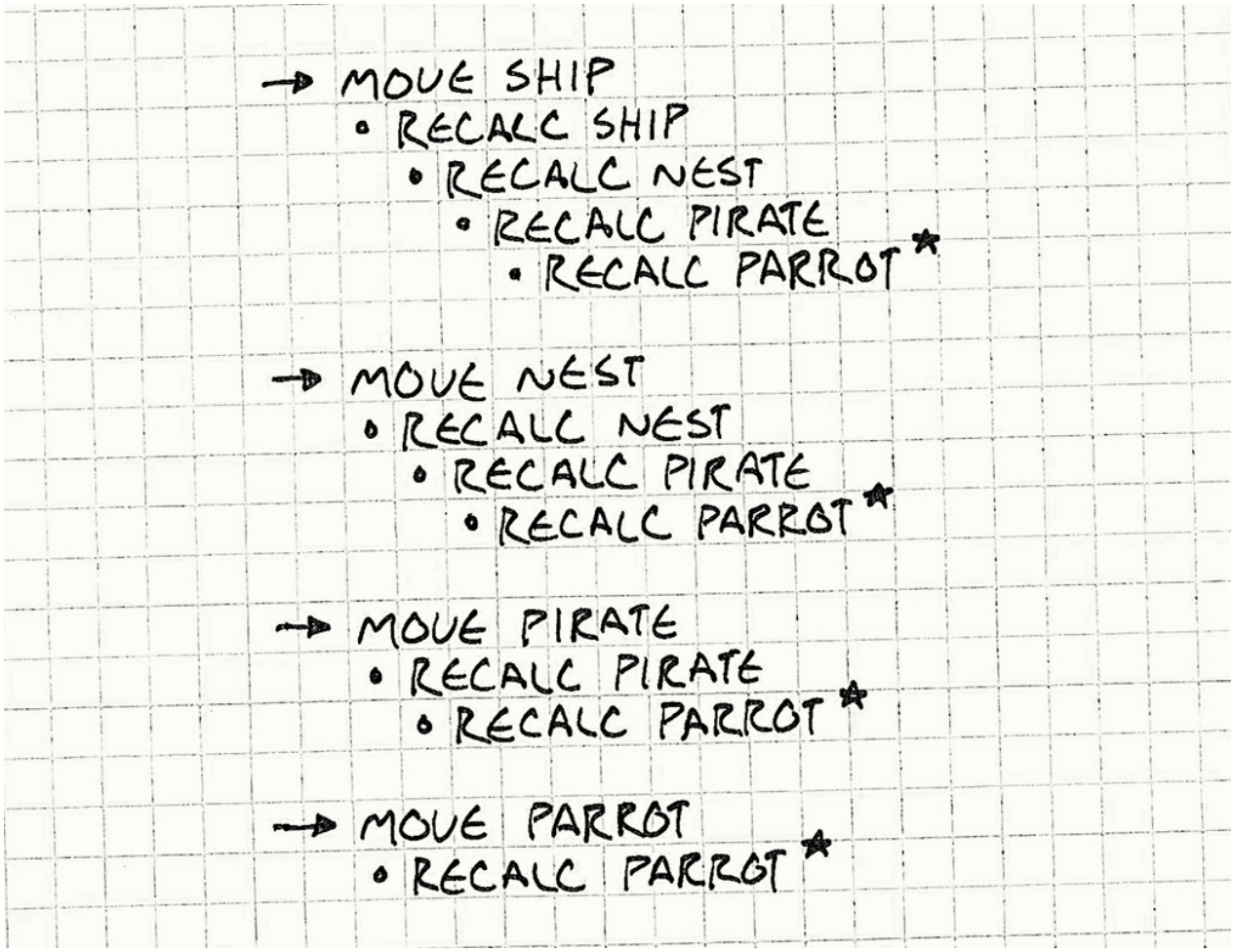
Но при этом тратится куча драгоценного процессорного времени! Ведь многие объекты в мире *не* двигаются на каждом кадре. Подумайте о статической геометрии, образующей уровень. Пересчитывать ее мировые трансформации на каждом кадре иначе как пустой тратой ресурсов и не назовешь.

Кэширование мировых трансформаций

Очевидный ответ — нужно *кэшировать* данные. В каждом объекте мы будем хранить локальную трансформацию и вычисленную мировую трансформацию. Во время рендеринга мы будем использовать готовую мировую трансформацию. Если объект никогда не движется, кэшированная трансформация всегда будет корректной и все будут счастливы.

А вот когда объект *движется*, проще всего будет сразу обновить его мировую трансформацию. Только не забудьте об иерархии. Когда объект движется, нам нужно будет пересчитывать *рекурсивно* трансформацию *всех его дочерних объектов*.

Представьте себе довольно плотный геймплей. На одном кадре, корабль попадает в океан, воронье гнездо раскачивается от ветра, пират из него свешивается, а попугай перескакивает ему на голову. У нас изменилось четыре локальных трансформации. Если мы будем пересчитывать мировую трансформацию каждый раз, когда локальная трансформация изменилась, то что получится?



> По строкам со звездочкой (★) вы можете заметить, что мы пересчитываем мировую трансформацию попугая четыре раза, хотя результат нам нужен всего один.

Мы изменили всего четыре объекта, но выполнили *десять* пересчетов мировых трансформаций. Результаты шести бесполезных пересчетов придется выкинуть, потому что наш рендер их использовать не будет. Мы пересчитали мировую трансформацию попугая *четыре* раза, а рендерим его всего один раз.

Проблема здесь в том что мировая трансфоромация зависит от нескольких локальных трансформаций. Так как мы делаем пересчет всех, как только изменится хотя бы *одна* из них, у нас получилось, что мы пересчитываем одну и ту же трансформацию столько раз за кадр, сколько локальных трансформаций от которых она зависит изменяется.

Отложенный пересчет

Мы можем решить эту проблему с помощью снижения связности между локальной трансформацией и обновлением мировой трансформации. Это позволит нам изменять кучу локальных трансформаций за раз и *затем* пересчитывать результирующую мировую трансформацию всего один раз, после того как произойдут все эти изменения и прямо перед рендерингом.

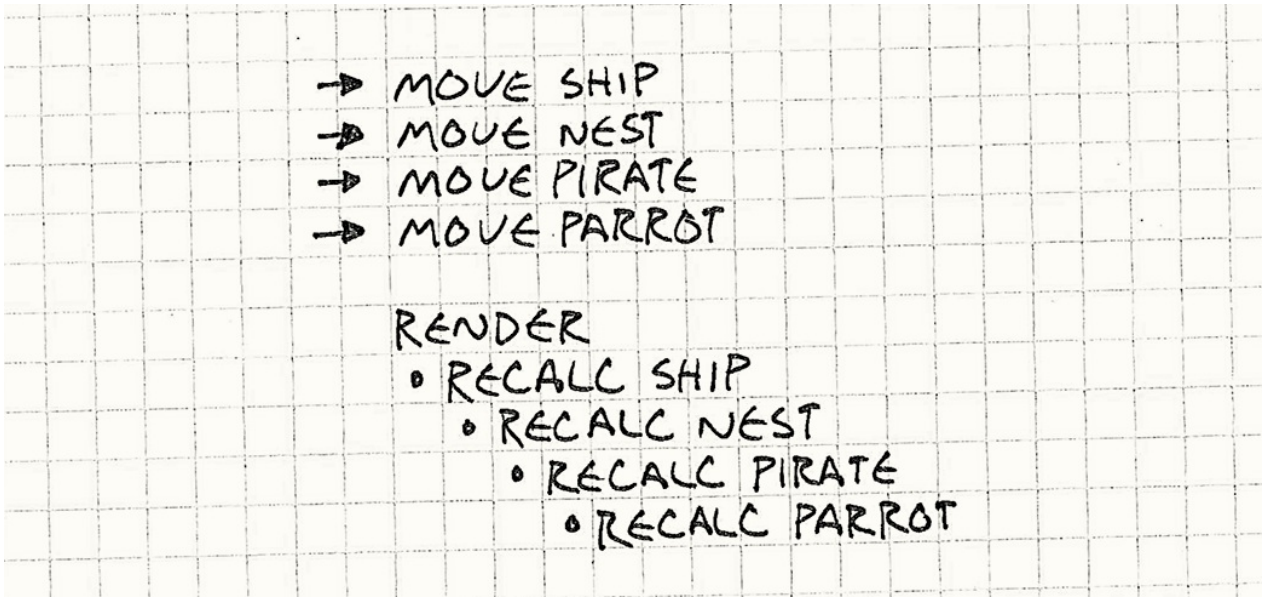
> Интересно, в какое количество архитектур ПО умышленно добавлено небольшое проскальзывание.

Чтобы этого добиться, мы добавим *флаг* в каждый объект графа. "Флаг" и "бит" в программировании почти синонимы: они оба значат наименьшую порцию данных, могущую быть всего в двух состояниях. Эти состояния можно называть "правда" и "ложь" или "установлен", "снят", "очищен". Я буду использовать их во взаимозаменяемом смысле.

Когда локальная трансформация будет изменяться, мы его установим. И когда нам понадобится мировая трансформация объекта, мы его будем проверять. Если флаг установлен, мы рассчитываем мировую трансформацию и очищаем флаг. Флаг отвечает на вопрос "Мировая трансформация устарела?" По неустановленной причине традиционно эта "устарелость" называется "грязью". Отсюда: грязный флаг. "Грязный бит" (Dirty bit) — это еще одно название данного шаблона, но я лично привык к менее похотливо звучащему названию.

> Редакторы википедии не разделяют мое мнение и используют название грязный бит.

Если мы применим этот шаблон и переместим объекты из предыдущего примера, у нас получится вот что:



На что мы можем надеяться в лучшем случае, так это на то, что мировая трансформация каждого затрагиваемого объекта вычисляется только один раз. С помощью всего одного бита, шаблон добился следующего:

- Он разбивает изменения на множество локальных трансформаций по цепочке наследования, по одному вычислению на объект.
- Мы избавились от вычислений для объектов, которые не двигались.

- И еще маленький бонус: если объект был удален перед рендерингом, мировая трансформация для него вычисляться не будет.

Шаблон

Набор **первичных данных** изменяется со временем. Набор **вторичных данных** вычисляется на основе первичных с помощью **ресурсоемкого процесса**. **"Грязный" флаг** отслеживает рассинхронизацию вторичных данных с первичными. Он устанавливается, когда **первичные данные изменяются**. Если флаг установлен, когда нам понадобились вторичные данные, **они вычисляются и флаг снимается**. В противном случае используются уже **вычисленные вторичные данные**.

Когда использовать

По сравнению с другим шаблонами в этой книге, данный шаблон решает весьма специфическую проблему. Как и в случае с другим оптимизациями, эту стоит применять только тогда, когда у вас есть реальная проблема с производительностью, ради решения которой имеет смысл заплатить увеличением сложности кода.

Грязный флаг применяется к двум типам работ: *вычислениям* и *синхронизации*. В обеих случаях процесс перехода от первичных данных к вторичным требует либо времени, либо дополнительных затрат.

В нашем примере с графом сцены, процесс замедляется из-за количества математических вычислений. С другой стороны когда этот шаблон используется для синхронизации, чаще всего вторичные данные находятся *где-то еще* — либо на диске, либо в сети — и даже простая передача данных из A в B будет затратной.

Есть и еще несколько требований:

- **Первичные данные должны изменяться чаще, чем используются вторичные данные.** Этот шаблон избегает обработки вторичных данных, если последующее изменение первичных данных может сделать их неактуальными до того, как они будут использованы. Если вы обнаружите, что вам требуется пересчет вторичных данных после каждого изменения первичных, этот шаблон вам не поможет.
- **Их сложно обновлять постепенно.** Скажем наш пиратский корабль может перевозить определенный вес. Поэтому нам нужен общий вес всего, что в него нагружено. Мы *можем* использовать этот шаблон и грязный флаг для общего

веса. Каждый раз, когда мы будем загружать или выгружать добычу, мы будем устанавливать флаг. А когда нам потребуется общий вес, мы будем его вычислять и снимать флаг.

Но, возможно, проще будет просто *поддерживать текущий общий вес*. Когда мы добавляем или удаляем вещь, мы добавляем или вычитаем ее вес из общего веса. Если мы можем "платить на ходу" таким образом и поддерживать вторичные данные в актуальном состоянии, такой вариант может работать лучше чем наш шаблон и пересчитывать вторичные данные целиком не потребуется.

Звучит так, как будто грязный флаг мало где нужен, но иногда он все таки полезен. Если поискать в любой кодовой базе слово "dirty", наверняка отыщется несколько мест, где используется этот шаблон.

> По моим наблюдениям, таким образом можно найти и множество комментариев с извинениями за "грязные" хаки.

Имейте в виду

Даже когда вы убедили себя, что этот шаблон вам подходит, есть несколько мелочей, способных доставить вам дискомфорт.

Бывает что стоимость вычислений слишком высока, чтобы ее откладывать

Этот шаблон откладывает выполнение медленной работы до тех пор, пока потребуется ее результат, но когда он нужен, он зачастую нужен *прямо сейчас*. Но ведь мы и применяли этот шаблон только потому, что наше вычисление слишком медленное!

Для нашего примера это не проблема, потому что мировые координаты все равно вычисляются достаточно быстро для вычисления в кадре, но вы легко можете представить себе случай, когда придется выполнить значительно больше вычислений и это отнимет слишком много времени. И если игра не *начнет* такую работу раньше, чем результат уже нужно демонстрировать игроку, у нас может образоваться весьма неприятная пауза в игре.

Еще одной проблемой отложенного выполнения является тот факт, что если что-то пойдет не так, вы можете вообще не суметь выполнить работу. Это может быть еще более проблематично, когда вы используете этот шаблон для сохранения некоего состояния в более устойчивой форме.

Например, текстовый редактор знает о том, что в вашем документе есть "несохраненные данные". Эта маленькая пулька или звездочка в заголовке программы и является наглядной визуализацией грязного флага. Первичные данные — это открытый в памяти документ, а вторичные данные — это файл на диске.

Многие программы не выполняют сохранение на диск до тех пор, пока документ не будет закрыт или произойдет выход из приложения. В большинстве случаев это нормально, но если вы случайно выдерните сетевой кабель из розетки, вы потеряете свой шедевр.

Редакторы, в которых реализовано фоновое авто-сохранение пытаются таким образом компенсировать этот недостаток. Частота авто сохранения — это точки в континууме, между которыми мы не будем терять слишком много своей работы, если произойдет сбой и не будем напрягать файловую систему постоянным сохранением.

> Это напоминает различные стратегии работы сборщика мусора, автоматически управляющего памятью. Подсчет ссылок (Reference counting) освобождает память сразу после того, как она становится больше не нужной, но при этом тратит процессорное время на обновление счетчика каждый раз, когда ссылки на объект изменяются.
>
> Простой сборщик мусора откладывает освобождение памяти до тех пор, пока она снова понадобится, но расплатой за это является страшная "пауза сборщика мусора", способная заморозить всю вашу игру до тех пор, пока сборщик закончит обработку кучи.
>
> Между ними двумя находятся более сложные системы, такие как система отложенного подсчета ссылок и инкрементный сборщик мусора, которые освобождают память реже, чем чистый подсчет ссылок, но чаще, чем останавливающие весь мир сборщики.

Вам нужно быть уверенным, что вы устанавливаете грязный флаг при *каждом* изменении состояния

Так как вторичные данные вычисляются на основе первичных, это по сути обычный кэш. Когда у вас есть кэшированные данные, самое сложное в работе с ними — это следить за *актуальностью кэша* (*cache invalidation*), т.е. за тем, чтобы кэшированные данные были синхронизированы с исходными данными. В данном шаблоне — это означает установку грязного флага, когда *любая* часть первичных данных изменяется.

> Фил Карлтон когда-то сказал: "Есть только две сложные вещи в компьютерных науках: актуальность кэша и придумывание имен".

Упустите что-то, и у вас в программе появятся некорректные вторичные данные. А это означает разочарованных игроков и крайне сложно отлавливаемые баги. Когда вы используете этот шаблон, вам нужно особенно заботиться о коде, который модифицирует первичное состояние и устанавливает грязный флаг.

Одним из способов этого добиться является инкапсуляция изменений первичных данных за каким-либо интерфейсом. Если все что может изменить состояние будет проходить через специальный `API`, вы можете устанавливать грязный флаг в этом интерфейсе и быть уверенным что вы ничего не упустите.

Вам нужно хранить предыдущие вторичные данные в памяти

Когда вам понадобятся вторичные данные и грязный флаг *не* установлен, используются предыдущие вычисленные данные. Это очевидно, но это не подразумевает то, что вам нужно хранить старые вторичные данные где-то в памяти, на случай если они вам позже понадобятся.

> Если вы используете шаблон для синхронизации первичного состояния с каким-либо другим местом — это не слишком большая проблема. В этом случае вторичные данные обычно вообще не находятся в памяти.

Если вы не используете этот шаблон, вам придется пересчитывать вторичные данные на лету, по мере необходимости и сразу от них избавляться. При этом вам не придется хранить их кэшированными в памяти за счет необходимости их вычисления каждый раз, когда они будут нужны.

Как и другие методы оптимизации, этот шаблон жертвует памятью ради скорости. В обмен на необходимость хранить ранее рассчитанные данные, вам не нужно будет пересчитывать их, если они не менялись. Такие расходы имеют смысл, когда вычисления медленные, а памятью можно пренебречь. Если у вас больше свободного времени, чем памяти, будет лучше рассчитывать их по мере необходимости.

> Алгоритмы компрессии занимаются прямо противоположными вещами: они оптимизируют занимаемое пространство за счет времени, необходимого для распаковки данных.

Пример кода

Предположим, что мы ознакомились с удивительно длинным списком требований и увидели, как шаблон выглядит в коде. Как я говорил раньше, настоящая математика матриц трансформации выходит за рамки рассмотрения этой книги, так что я инкапсулировал ее в класс, внутри которого как мы будем предполагать и находится вся реализация:

```
class Transform
{
public:
  static Transform origin();

  Transform combine(Transform& other);
};
```

Единственная нужная нам операция здесь — это `combine()`, с помощью которой вы можете получить мировую трансформацию, которая объединяет локальную трансформацию и всю цепочку предков. Также, здесь есть метод для получения "начальной" трансформации — обычно единичной матрицы, вообще не содержащей переноса, поворота и масштабирования.

Теперь напишем набросок класса для объекта графа сцены. Это самый минимум того, что нам понадобится *до* применения шаблона.

```
class GraphNode
{
public:
  GraphNode(Mesh* mesh) : mesh_(mesh), local_(Transform::origin())
  {}

private:
  Transform local_;
  Mesh* mesh_;

  GraphNode* children_[MAX_CHILDREN];
  int numChildren_;
};
```

Каждый узел имеет локальную трансформацию, описывающую его позицию относительно родителя. Он содержит сетку, представляющую графическую часть объекта (мы позволим `mesh_` принимать значение `NULL` для обработки невидимых узлов, используемых для группировки своих потомков.) И, наконец, каждый узел имеет коллекцию дочерних узлов, которая может быть и пустой.

Таким образом "граф сцены" — это всего лишь единственный корневой `GraphNode`, чьи дети (и внуки и т.д.) представляют все объекты в мире:

```
GraphNode* graph_ = new GraphNode(NULL);
// Добавление дочернего узла к корневому узлу графа...
```

Для того, чтобы отрендерить граф сцены, всё, что нам нужно, так это обойти всё дерево узлов, начиная с корневого, и вызвать следующую функцию для сетки каждого узла вместе с правильной мировой трансформацией:

```
void renderMesh(Mesh* mesh, Transform transform);
```

Мы не будем ее здесь реализовывать, но если бы решились, она выполняла бы всю магию, необходимую для отрисовки сетки в нужной координате в мире. Если мы сможем корректно и эффективно вызвать ее для каждого узла в графе сцены, мы будем счастливы.

Неоптимизированный обход

Чтобы запачкать руки, давайте напишем простейший обход графа сцены для рендеринга, подсчитывающий мировые координаты на лету. Он не будет оптимальным, но по крайней мере будет простым. Добавим в `GraphNode` новый метод:

```
void GraphNode::render(Transform parentWorld)
{
  Transform world = local_.combine(parentWorld);

  if (mesh_)
    renderMesh(mesh_, world);

  for (int i = 0; i < numChildren_; i++) {
    children_[i]->render(world);
  }
}
```

Мы передаем сюда с помощью `parentWorld` трансформацию родителя узла. Таким образом, все, что нам остается для вычисления правильной мировой трансформации данного узла — это объединить ее с локальной трансформацией. Нам не нужно выполнять обход *вверх* по цепочке родителей для подсчета мировых трансформаций, потому что они уже будут вычислены по мере того, как мы спускаемся *вниз* по цепочке.

Мы вычисляем мировую трансформацию узла и сохраняем ее в `world`, а затем рендерим сетку, если она есть. И, наконец, мы рекурсивно обходим дочерние узлы, передавая внутрь мировую трансформацию *данного* узла. В конце концов, это простой рекурсивный метод.

Для отрисовки всего графа сцены, мы запускаем процесс начиная с корневого узла:

```
graph_->render(Transform::origin());
```

Давайте запачкаемся

Итак, данный код работает правильно — рендерит все сетки в нужных местах — но не слишком эффективно. Он вызывает `local_.combine(parentWorld)` для каждого узла в графе на каждом кадре. Давайте посмотрим, как с этим сможет справиться шаблон. Для начала нам нужно добавить в `GraphNode` два поля:

```
class GraphNode
{
public:
  GraphNode(Mesh* mesh) : mesh_(mesh),
    local_(Transform::origin()),
    dirty_(true)
  {}

  // Другие методы...

private:
  Transform world_;
  bool dirty_;
  // Другие поля...
};
```

Поле `world_` кэширует ранее вычисленную мировую трансформацию, а `dirty_` — это естественно грязный флаг. Обратите внимание, что вначале он равен `true`. Когда мы создаем новый узел, у нас еще нет рассчитанной мировой трансформации и поэтому она еще не синхронизирована с локальной трансформацией.

Единственная причина, почему нам вообще нужен этот шаблон — это способность объектов *двигаться*, так что добавим соответствующую поддержку:

```
void GraphNode::setTransform(Transform local)
{
  local_ = local;
  dirty_ = true;
}
```

Главное здесь то, что здесь снова устанавливается грязный флаг. Мы ничего не забыли? Ах да! Дочерние узлы!

Когда родительский узел двигается, мировые координаты всех дочерних становятся недействительными. Но здесь мы не будем устанавливать их грязный флаг. Мы *могли бы* это сделать, но такая рекурсия будет слишком медленной. Вместо этого мы поступим умнее, когда доберемся до рендера. Итак:

```
void GraphNode::render(Transform parentWorld, bool dirty)
{
  dirty |= dirty_;
  if (dirty) {
    world_ = local_.combine(parentWorld);
    dirty_ = false;
  }

  if (mesh_)
    renderMesh(mesh_, world_);

  for (int i = 0; i < numChildren_; i++) {
    children_[i]->render(world_, dirty);
  }
}
```

> Здесь используется допущение, что если проверка `if` работает быстрее, чем умножение матриц. Интуитивно вам может таки показаться. Ведь логично, что проверка одного бита быстрее, чем куча операций с вещественными числами.
>
> Однако современные процессоры очень сложные штуки. Они нацелены на *конвейерную обработку* (*pipelining*) — очереди из серии последовательных инструкций. Ветвление типа нашего `if` может привести к *ошибке предсказаний перехода* (*branch misprediction*) и заставить процессор потерять циклы на повторное заполнение конвейера.
>
> Глава Локальность данных (Data Locality) рассказывает о том, как вы можете помочь современным процессорам работать быстрее и не попадать в такие неприятные ситуации.

Очень похоже на оригинальную наивную реализацию. Основное отличие заключается в том, что мы проверяем грязность узла перед расчетом мировой трансформации и сохраняем результат в поле, а не в локальной переменной. Когда узел становится чистым, мы полностью пропускаем `combine()` и используем старое, но корректное значение `world_`.

Хитрость заключается в параметре `dirty`. Он будет равен `true`, если любой узел выше этого узла в цепочке иерархии был грязным. Примерно также, как `parentWorld` инкрементно обновляет мировую трансформацию во время спуска по иерархии, `dirty` отслеживает грязность дочерней цепочки.

Это позволяет нам избежать рекурсивной установки флага `dirty_` для каждого ребенка в `setTransform()`. Вместо этого мы просто передаем родительский грязный флаг вниз его детям во время рендеринга и смотрим, нужно ли пересчитывать их мировую трансформацию.

В результате получается как раз то, что мы и хотели — изменение локальной трансформации узла требует всего нескольких назначений, а рендеринг мира требует минимального количества вычислений мировых трансформаций, изменившихся со времени последнего кадра.

> Обратите внимание, что этот трюк работает только благодаря тому что `render()` — это *единственная* вещь в `GraphNode`, которой нужно обновлять мировую трансформацию. Если бы эта информация была бы нужна кому-то еще, нам пришлось бы выдумывать что-то другое.

Архитектурные решения

Этот шаблон довольно специфичен, так что настраивать здесь можно всего несколько вещей.

Когда очищается грязный флаг?

- **Когда нам потребуется результат:**
  - *Если результат нам не понадобится, вычисления вообще не будут выполняться.* Для первичных данных, которые изменяются значительно чаще, чем выполняется доступ к вторичным, это даст заметный выигрыш.
  - *Если вычисления требуют много времени, это может привести к возникновению заметной паузы.* Откладывание работы на потом до тех пор, пока игрок должен будет увидеть результат, может испортить впечатление от

игры. Обычно, если работа не занимает слишком много времени — это незаметно, но в противном случае ее нужно выполнять все-таки раньше.

- **В жестко определенных контрольных точках:**

  Иногда, в работе игры возникают моменты, где процесс выполнения отложенных вычислений смотрится органично. Например, мы можем захотеть выполнять сохранение, когда наш пират заходит в порт. Или же точка синхронизации вообще может не быть частью игровой механики. Мы можем захотеть скрыть работу за загрузочным экраном или катсценой.

  - *Выполнение работы не влияет на впечатление игрока от игрового процесса*. В отличие от предыдущего варианта, у вас обычно есть чем отвлечь игрока, пока выполняется нужная вам обработка.
  - *Когда возникает необходимость выполнить работу вы теряете управление*. Это своего рода противоположность предыдущего пункта. Вы можете на микро-уровне управлять тем, когда выполнится работа и вы можете быть уверены что игра ее корректно выполнит.

    А вот в чем вы *не можете* быть уверены — так это в том, что игрок действительно доберется до контрольной точки или выполнит определенный вами критерий. Если он заблудится или игра поведет себя некорректно, работа отложится на более долгое время чем вы рассчитывали.

- **В фоне:**

  Обычно, при первом изменении, запускается таймер и затем обрабатываются все изменения, произошедшие между следующим тиком таймера и предыдущим состоянием.

  > В терминах человеко-компьютерного взаимодействия специальная задержка между получением программой пользовательских данных и их обработкой называется гистеризисом.

  - *Вы можете настраивать частоту выполняемой работы*. Настраивая интервал таймера, вы можете задавать нужную вам частоту или нерегулярность.
  - *Вы можете выполнять больше лишней работы*. Если за время работы таймера первичное состояние изменилось незначительно, вы можете столкнуться с тем, что будете обрабатывать большой объем почти неизменившихся данных.

- *Вам нужна поддержка для выполнения работы асинхронно*. Обработка данных "в фоне" подразумевает, что игрок в то же время может продолжать делать, что ему хочется. Это значит, что вам придется воспользоваться потоками или другим методом поддержки конкурентной работы, чтобы игра могла работать с вашими данными прямо во время своей работы.

  Так как игрок может взаимодействовать с тем же первичным состоянием, которое вы обрабатываете, вам нужно позаботиться о том, чтобы сделать его более безопасным в плане конкурентного изменения.

Насколько детализированным будет ваше применение грязных флагов?

Допустим, ваша онлайн игра про пиратов позволяет строить и модифицировать пиратский корабль. Корабли автоматически сохраняются на сервере, чтобы игроки могли потом продолжить с того места, на котором прервались. Мы используем грязный флаг для определения, какая палуба у нас настроена и нуждается в отправке на сервер. Каждая порция данных, отсылаемая на сервер содержит часть модифицированных данных корабля и частичку метаданных, описывающих где на корабле произошло это изменение.

- **Если оно сильно детализированное:**

  Предположим, что у вас есть грязный флаг для каждой отдельной доски на каждой палубе.

  - *Вы будете обрабатывать только изменившиеся данные*. Вы будете отсылать на сервер только ту часть состояния корабля, которая изменилась.

- **Менее детализированное:**

  В качестве альтернативного решения вы можете ассоциировать отдельный грязный флаг с каждой палубой. Изменение чего либо на палубе делает грязной всю палубу.

  > Я хотел вставить сюда какую-нибудь шутку про швабру и мойку, но воздержусь.

  - *Вы будете вынуждены обрабатывать неизменившиеся данные*. Добавьте на палубу всего одну бочку и придется отправлять на сервер данные о содержимом всей палубы.
  - *На хранение грязных флагов понадобится меньше памяти*. Добавьте на палубу десять бочек и вам понадобится всего один бит, чтобы отслеживать их всех.

- *На постоянные накладные расходы получится тратить меньше времени.*/ Когда мы обрабатываем некоторые измененные данные, для самой обработки этих данных чаще всего нужно выполнять какую-либо определенную работу. В нашем примере это метаданные, которые нужны для того, чтобы определить, где на корабле были изменены данные. Чем больше обрабатываемый кусок данных, тем меньше этих метаданных и тем меньше накладные расходы на их обработку.

Смотрите также

- Этот шаблон очень часто можно встретить не в играх, а в браузерных фреймворках, типа Angular. Они используют грязные флаги для отслеживания того, какие данные были изменены в браузере и должны быть отправлены на сервер.
- Физические движки следят как за движущимися объектами, так и за покоящимися. Так, как покоящееся тело не двигается до тех пор, пока к нему не будет приложен импульс, их не нужно обрабатывать, пока их кто-нибудь не тронет. Этот флаг "двигается ли" представляет из себя грязный флаг, обозначающий к каким объектам приложена сила, и то, что для них нужно применять физические вычисления.

Пул объектов (Object Pool)

Задача

Улучшение производительности и эффективности использования памяти за счет повторного использования объектов из фиксированного пула, вместо их индивидуального выделения и освобождения.

Мотивация

Мы работаем над визуальными эффектами в игре. Когда герой кастует заклинание, мы хотим, чтобы мерцающие блестки рассыпались по всему экрану. Это будут вызовы *системы частиц*: движок, порождающий маленькие блестящие картинки и анимирующий их до тех, пор пока они не исчезнут.

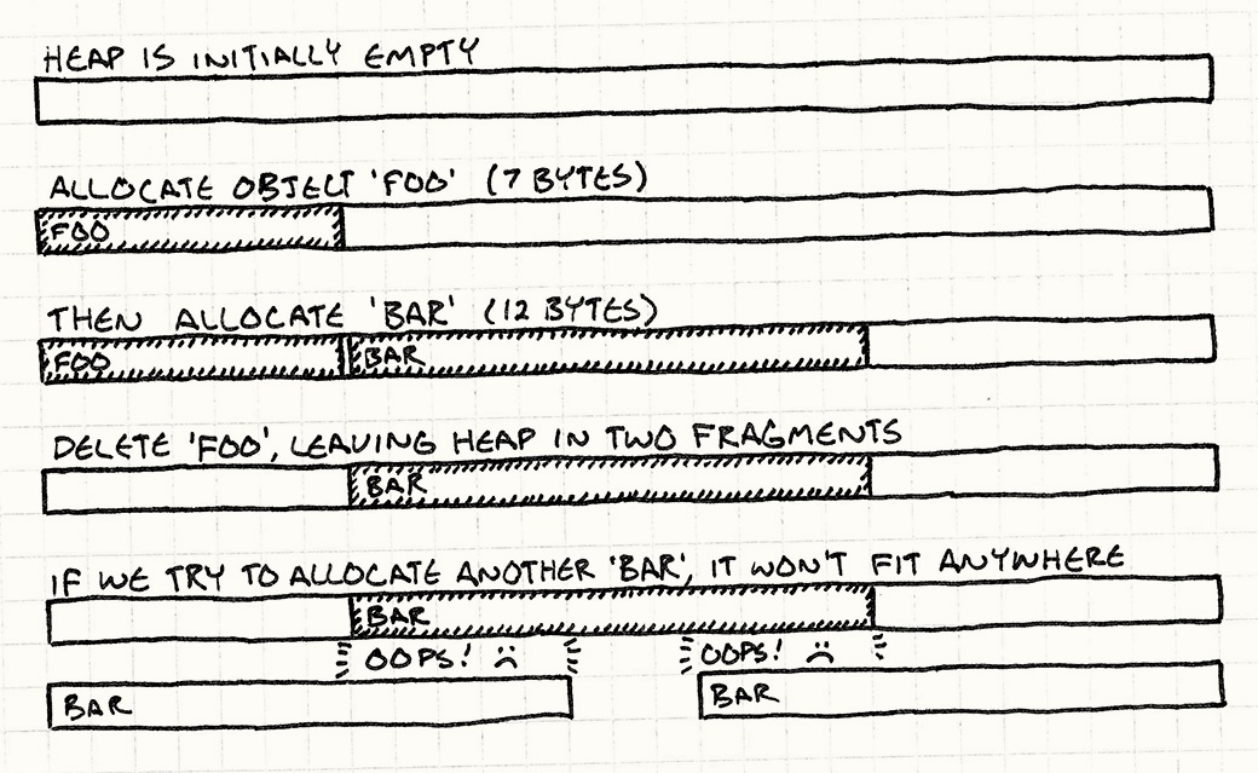
Так как по мановению волшебной палочки мы можем породить сотни частиц, нашей системе нужно иметь возможность создавать их очень быстро. Что более важно, нам нужно удостовериться, что создание и уничтожение частиц не приведет к *фрагментации памяти*.

Проклятье фрагментации

Программирование для игровых консолей типа `XBox 360` больше похоже на программирование для встроенных систем, чем на программирование для `PC`. Как и в программировании для встроенных систем, консольные игры должны работать очень долгое время без падений и утечек памяти, при том, что эффективные менеджеры памяти встречаются не так уж и часто. В такой рабочей среде фрагментация памяти смертельно опасна.

Фрагментация означает, что свободное место в нашей куче разбивается на мелкие кусочки памяти вместо больших открытых блоков. *Общее* количество доступной памяти может быть большим, но наибольший *последовательный* участок может быть ужасно маленьким. Предположим, что у нас есть четырнадцать свободных байт, но они фрагментированы на два отдельных куска, разделенные участком занятой памяти посередине. Если мы попробуем разместить здесь объект длиной в двенадцать байт, мы получим ошибку. И больше никаких блесток на экране.

> Очень похоже на параллельную парковку на заставленной улице, когда припаркованные авто распределены слишком далеко друг от друга. Если бы их можно было посдвигать чуть поближе, нашлось бы еще достаточно места. Но к сожалению это место *фрагментировано* на промежутки между дюжинами машин.



> Вот как будет фрагментирована наша куча и почему у нас будет ошибка выделения памяти, хотя теоретически памяти у нас достаточно.

Даже если фрагментация встречается нечасто, она может постепенно привести кучу в состояние бесполезных пузырей из дырок и щелей, полностью лишив игру возможности с ней работать.

> Большинство консольных платформодержателей требуют, чтобы игры проходили "тест на протечку" (soak test), когда игра оставляется работающей на несколько дней в демо-режиме. Если игра падает — ей не разрешают выйти. Тест на протечку иногда проваливается и из-за какого-нибудь редкого бага, но чаще всего игру обрушивает утечка памяти, вызванная фрагментацией.

Лучшее из двух миров

Из-за фрагментации, и потому что выделение памяти может работать медленно, игры всегда очень осторожны насчет того, где и как работать с памятью. Простейшее решение обычно и самое лучшее: захватите кусок памяти побольше при старте игры и

освободите его при выходе из игры. К сожалению, такую стратегию сложно использовать, когда нам нужно постоянно создавать и удалять объекты во время работы игры.

Пул объектов дает нам лучшее из двух миров: с точки зрения менеджера памяти мы просто выделяем большой кусок памяти и не освобождаем ее, пока игра работает. Для пользователей пула мы можем создавать и удалять объекты сколько нашей душе будет угодно.

Шаблон

Определим класс *пула*, содержащего коллекцию *многоразовых объектов*. Каждый объект поддерживает *запрос "используется"*, означающий, что он сейчас "жив". Когда пул инициализируется, он сразу создает всю коллекцию объектов (обычно выделяя один последовательный участок памяти) и инициализирует их всех состоянием "не используется".

Когда вам понадобится новый объект, вы запрашиваете его у пула. Он ищет доступный объект, инициализирует его значением "используется" и возвращает. Когда объект больше не нужен, он снова возвращается в состояние "не используется". Таким образом, объекты можно свободно создавать и удалять без необходимости выделять память или другие ресурсы.

Когда использовать

Этот шаблон широко используется в играх не только для очевидных вещей типа игровых сущностей и визуальных эффектов, но и для менее заметных структур данных типа проигрываемых звуков. Пул объектов используется когда:

- Вам нужно часто создавать и удалять объекты.
- Объекты одного размера.
- Выделение объектов из кучи работает медленно или может привести к фрагментации памяти.
- Каждый объект инкапсулирует ресурс типа базы данных или сетевого соединения, который сложно получать и можно использовать повторно.

Имейте в виду

Обычно, вы можете положиться на сборщик мусора или операторы `new` и `delete`, которые сделают всю работу за вас. Когда вы используете пул объектов, вы как будто говорите "Я лучше знаю, как обращаться с этими байтами". А еще это значит, что на вас ложится бремя ограничений шаблона.

Пул может тратить память на неиспользуемые объекты

Размер пула нужно настраивать соразмерно с нуждами игры. При настройке обычно проще всего понять, когда пул *недостаточного* размера (уверен, что падение игры наверняка привлечет ваше внимание). Но еще нужно следить и за тем, чтобы пул не был слишком *большим*. Если уменьшить пул, освободившуюся память можно использовать для чего-либо более полезного.

В каждый момент времени может быть активно только определенное количество объектов

Иногда это хорошо. Разделение памяти на отдельные пулы для различных типов объектов означает с одной стороны то, что последовательность взрывов не заставит вашу систему частиц отожрать *всю* доступную память, не позволив вам создать что-либо более полезное, типа нового противника.

Кроме того, вы должны быть готовы к ситуации, что выделить объект из пула не удастся, потому что все объекты будут заняты. Есть несколько стратегий обработки таких случаев:

- *Прямое вмешательство*. Это самое очевидное "исправление": будем настраивать размер пула таким образом, чтобы он никогда не переполнялся, независимо от действий пользователя. Для пулов с важными объектами, такими как противники или геймплейные предметы, это хороший выход. Не может быть "правильной" обработки недостатка свободных слотов для создания большого босса, когда игрок дошел до конца уровня. Так что лучше придумать что-то такое, что не позволит нам оказаться в подобной ситуации.

  Недостатком является то, что вам придется держать занятыми большие объемы памяти ради каких-то редких крайних случаев. Поэтому фиксированный размер пула не может считаться лучшим решением для всех состояний игры. Например, некоторые уровни могут больше налегать на визуальные эффекты, а другие на звуки. В таких случаях нам лучше иметь пулы объектов, настраиваемые отдельно для обеих сценариев.

- *Просто не создаем объект*. Звучит грубо, но в случаях, типа работы с системами частиц имеет смысл. Если все ваши частицы используются, экран и так кишит эффектами. Пользователь не обратит внимание, если следующий взрыв будет менее впечатляющим, чем уже отображаемые.
- *Принудительное убийство существующего объекта*. Представим себе пул проигрываемых сейчас звуков и предположим, что вы хотите запустить новый звук, но наш пул заполнен. У вас нет желания просто проигнорировать новый звук: пользователь заметит, что его магическая палочка обычно драматически посвистывает, а *иногда* не издает ни звука. Лучше вместо этого найти самый тихий звук из тех, что уже играются и заменить его новым звуком. Новый звук заглушит слышимый обрыв предыдущего звука.

  В целом, если *исчезновение* существующего объекта будет менее заметным, чем *непоявление* нового — это вполне хорошее решение.
- *Увеличение размера пула*. Если ваша игра позволяет вам распоряжаться памятью более гибко, вы можете увеличивать размер пула во время выполнения или создавать дополнительные пулы переполнения. Если воспользовавшись одним из этих вариантов вы отхватите больше памяти, подумайте о том, имеет ли смысл в будущем вернуться к прежнему размеру, когда дополнительная вместимость вам уже будет не нужна.

Размер памяти для каждого объекта фиксирован

Большинство реализаций пула хранят объекты в массиве объектов на месте (in-place). Если все ваши объекты одного типа — это нормально. Однако, если вы захотите хранить в пуле объекты нескольких типов или экземпляры подклассов с дополнительными полями, вам нужно быть уверенными, что каждый слот пула обладает достаточным размером, чтобы вместить *максимально* возможный объект. В противном случае неожиданно большой объект вылезет за свои границы на соседний и разрушит память.

В то же время, когда ваши объекты могут быть разного размера, вы впустую тратите память. Каждый слот должен быть достаточно большим, чтобы вместить максимально возможный объект. Если такие большие объекты встречаются редко, вы будете попусту тратить память всякий раз, когда будете помещать в слот маленький объект. Это все равно, что проходить таможню в аэропорту с огромным чемоданом, внутри которого лежат только ключи и бумажник.

Когда вы обнаружите, что тратите на это слишком много памяти, вспомните о разделении пула на отдельные пулы для разных размеров объектов — большие отделения для чемоданов и маленькие для карманной мелочи.

> Такой шаблон чаще всего используется в наиболее эффективных в плане скорости менеджерах памяти. У менеджера есть несколько пулов с блоками разного размера. Когда вы просите у него выделить вам блок, он ищет открытый слот в пуле подходящего размера и выделяет его из пула.

Повторно используемые объекты не очищаются автоматически

Большинство менеджеров памяти обладают отладочными функциями, которые очищают только что выделенную или освобожденную память магическими значениями типа `0xdeadbeef`. Это помогает обнаруживать болезненные баги, вызванные использованием неинициализированных значений или обращением к уже освобожденной памяти.

Так как наш пул объектов не заходит в деле управления памяти дальше повторного использования объектов, он не имеет подобной страховочной сетки. Еще хуже то, что память, используемая для "нового" объекта, хранила раньше объект того же самого типа. Это делает весьма возможной ситуацию, когда вы забудете инициализировать что-то внутри нового созданного объекта, а память, где он будет размещаться, уже будет содержать почти корректные данные, оставшиеся с прошлой жизни.

Поэтому, нужно с особой тщательностью следить за тем, чтобы код инициализации нового объекта в пуле выполнял инициализацию объекта *полностью*. Возможно, даже стоит потратить немного времени на написание отладочного функционала, очищающего память в слоте объекта при его повторном использовании.

> Я буду гордится, если вы выберете для очистки магическое число `0x1deadb0b`.

Неиспользуемые объекты остаются в памяти

Пулы объектов реже всего используются в системах со сборщиками мусора, потому что в таком случае менеджер памяти сам занимается проблемой фрагментации за вас. Но пулы все равно полезны тем, что помогают вам избегать выделения и освобождения памяти, особенно на мобильных устройствах с медленным процессором и простым сборщиком мусора.

Если все таки будете использовать пул объектов, опасайтесь потенциальных конфликтов. Так как пул на самом деле не освобождает объекты, когда они больше не используются, они остаются в памяти. Если они содержат ссылки на другие объекты, они тем самым не дадут сборщику утилизировать и эти объекты тоже. Чтобы этого избежать, нам нужно очищать все ссылки на другие объекты, когда объект из пула нам больше ненужен.

Пример кода

Настоящие системы частиц обычно оперируют гравитацией, ветром, трением и другими физическими эффектами. Наш максимально простой пример будет просто перемещать частицы по прямой линии на протяжении некоторого количество кадров и потом будет убивать частицы. Не совсем киношная картинка, но для иллюстрации работы с пулом вполне достаточно.

Начнем с самой наипростейшей реализации. Для начала наш класс частиц:

```
class Particle
{
public:
  Particle() : framesLeft_(0)
  {}

  void init(double x, double y,
      double xVel, double yVel, int lifetime) {
    x_ = x; y_ = y;
    xVel_ = xVel; yVel_ = yVel;
    framesLeft_ = lifetime;
  }
```

```
  void animate() {
    if (!inUse()) return;

    framesLeft_--;
    x_ += xVel_;
    y_ += yVel_;
  }

  bool inUse() const {
    return framesLeft_ > 0;
  }

private:
  int framesLeft_;
  double x_, y_;
  double xVel_, yVel_;
};
```

Конструктор по умолчанию инициализирует частицу как "не используемую". Следующий вызов `init()` инициализирует частицу уже в живом состоянии.

Частицы анимируются с помощью функции с именем `animate()`, которая должна вызываться на каждом кадре.

Пулу нужно знать о том, какая из частиц доступна для повторного использования. Он узнает это с помощью функции частицы `inUse()`. Учитывая, что жизнь частиц ограничена, она использует переменную `_framesLeft` для определения того, что частица используется без хранения отдельного флага.

Класс пула также предельно прост:

```
class ParticlePool
{
public:
  void create(double x, double y,
    double xVel, double yVel, int lifetime);

  void animate() {
    for (int i = 0; i < POOL_SIZE; i++) {
      particles_[i].animate();
    }
  }

private:
  static const int POOL_SIZE = 100;
  Particle particles_[POOL_SIZE];
};
```

Функция `create()` позволяет внешнему коду создавать новые частицы. Игра вызывает на каждом кадре `animate()`, анимируя тем самым все частицы в пуле.

> Этот метод `animate()` представляет собой пример шаблона Метод обновления (Update Method).

Сами частицы просто сохраняются в массив фиксированного размера в классе. В этой простой реализации размер пула жестко закодирован в определении класса, но может быть определен и извне с помощью динамического массива определенного размера или с помощью значения для параметра шаблона.

Создание новой частицы предельно простое:

```
void ParticlePool::create(double x, double y,
    double xVel, double yVel, int lifetime)
{
  // Find an available particle.
  for (int i = 0; i < POOL_SIZE; i++) {
    if (!particles_[i].inUse()) {
      particles_[i].init(x, y, xVel, yVel, lifetime);
      return;
    }
  }
}
```

Мы обходим пул и ищем первую доступную частицу. Когда мы ее находим, мы инициализируем ее и на этом все. Обратите внимание, что в этой реализации, если у нас нет доступных частиц, мы вообще не создаем новую.

Это и есть вся простая система частиц за исключением рендеринга частиц конечно. Теперь мы можем создать пул и несколько частиц с его помощью. Частицы будут деактивировать сами себя автоматически, когда закончится их время жизни.

Такая реализация вполне подходит для игры, но вы уже наверняка заметили, что создание новой частицы может потребовать (потенциально) обхода всей коллекции частиц до тех пор, пока не найдем пустой слот. Если пул достаточно большой и практически заполнен, это может быть довольно медленно. Посмотрим, как мы сможем с этим справиться.

> Для тех из нас, кто еще помнит теорию алгоритмов, создание частицы имеет сложность `O(n)`.

Свободный список

Если мы не хотим терять время на *поиск* свободных частиц, логичным решением будет следить за ними. Мы можем хранить отдельный список указателей на каждую неиспользуемую частицу. И когда нам нужно будет создать новую частицы, мы просто удалим первый указатель из списка и повторно используем частицу, на которую он указывает.

К сожалению, для этого придется поддерживать еще один отдельный массив с количеством указателей, равным количеству объектов в пуле. В конце концов, когда мы создаем пул, *все* объекты в нем являются неиспользуемыми, так что изначально нам понадобятся указатели на все объекты.

И все-таки, мне хотелось бы решить проблему с производительностью *без* дополнительных затрат памяти. К счастью, у нас уже есть свободная память, которую мы можем позаимствовать — это сами неиспользуемые частицы.

Когда частица не используется, большая часть ее состояния не имеет никакого значения. Ее позиция и скорость не используются. Единственное состояние которое для нас важно — это мертва ли частица. В нашем примере это член класса `_framesLeft`. Все остальные биты можно использовать. Вот пересмотренный вариант:

```
class Particle
{
public:
  // ...

  Particle* getNext() const { return state_.next; }
  void setNext(Particle* next) { state_.next = next; }

private:
  int framesLeft_;

  union {
    // Состояние когда частица используется.
    struct {
      double x, y;
      double xVel, yVel;
    } live;

    // Состояние когда частица доступна.
    Particle* next;
  } state_;
};
```

Мы взяли все члены переменные за исключением `framesLeft_` и переместили их в структуру `live` внутри объединения `state_`. Эта структура хранит состояние частицы, когда она анимируется. Когда частица не используется, в дело вступает другая часть объединения — член `next`. Он хранит указатель на следующую доступную частицу за данной.

> В наши дни объединения используются не слишком часто, так что даже их синтаксис может выглядеть для вас незнакомым. Если вы работаете в команде, у вас наверняка есть "гуру по памяти", который вам помогает в случаях, когда у вас появляются проблемы с бюджетом памяти. Спросите его об объединениях. Такие люди много о них знают, включая довольно забавные трюки с упаковкой битов.

Мы можем использовать эти указатели для создания связанного списка, связывающего воедино все неиспользуемые частицы в пуле. У нас есть нужный нам список доступных частиц и мы не использовали никакой дополнительной памяти. Вместо этого мы отобрали память для хранения списка у самих мертвых частиц.

Такая хитрая техника называется *свободный список* (free list). Чтобы она заработала, нам нужно удостовериться, что указатели инициализируются корректно и поддерживать их, когда частицы создаются и уничтожаются. И конечно, нам нужно следить за головой списка:

```
class ParticlePool
{
// ...
private:
  Particle* firstAvailable_;
};
```

Когда пул создается впервые, *все* частицы доступны, так что свободный список должен распространяться на весь пул. Конструктор пула делает это следующим образом:

```
ParticlePool::ParticlePool()
{
  // Доступен первый.
  firstAvailable_ = &particles_[0];

  // Каждая частица указывает на следующую.
  for (int i = 0; i < POOL_SIZE – 1; i++) {
    particles_[i].setNext(&particles_[i + 1]);
  }

  // Последняя завершает список.
  particles_[POOL_SIZE – 1].setNext(NULL);
}
```

Теперь для создания новой частицы нам нужно перейти к первой доступной:

> Сложность O(1), детка! Вот чего мы добились!

```
void ParticlePool::create(double x, double y,
    double xVel, double yVel, int lifetime)
{
  // Проверяем что пул не заполнен полностью.
  assert(firstAvailable_ != NULL);

  // Удаляем ее из списка доступных.
  Particle* newParticle = firstAvailable_;
  firstAvailable_ = newParticle->getNext();

  newParticle->init(x, y, xVel, yVel, lifetime);
}
```

Нам нужно знать, когда частица умирает для того, чтобы добавить ее в свободный список. Для этого мы сделаем так, чтобы `animate()` возвращала `true`, если предыдущая живая частица испустила дух в этом кадре:

```
bool Particle::animate()
{
  if (!inUse()) return false;

  framesLeft_--;
  x_ += xVel_;
  y_ += yVel_;

  return framesLeft_ == 0;
}
```

Когда это происходит, мы просто переносим ее обратно в список:

```
void ParticlePool::animate()
{
  for (int i = 0; i < POOL_SIZE; i++) {
    if (particles_[i].animate()) {
      // Добавляем эту частицу в начало списка.
      particles_[i].setNext(firstAvailable_);
      firstAvailable_ = &particles_[i];
    }
  }
}
```

Вот и готово. Маленький и удобный пул объектов с константным временем создания и удаления.

Архитектурные решения

Как вы могли увидеть, простейшая реализация пула объектов практически тривиальна: создается массив объектов и они переинициализируются по мере необходимости. Реальный код редко бывает настолько минималистичным. Существует несколько способов сделать пул более обобщенным, безопасным для использования и простым для поддержки. Прежде, чем вы решите реализовать пул в собственной игре, попробуйте ответить на несколько вопросов:

Привязаны ли объекты к пулу?

Первый вопрос, с которым мы сталкивается, когда хотим написать пул объектов — это должны ли объекты знать о том, что находятся в пуле. Обычно это так, но у вас не будет такой роскоши, если вы пишете класс обобщенного пула, в котором можно хранить произвольные объекты.

Если объекты связаны с пулом:

- *Реализация будет проще.* Вы можете просто добавить в объект пула флаг "используется" или функцию, и на этом остановиться.
- *Вы можете быть уверены, что объекты создаются только в пуле.* В `C++` это легко сделать, объявив класс пула дружественным классом класса объекта и сделать его конструктор приватным.

```
class Particle
{
  friend class ParticlePool;

private:
  Particle() : inUse_(false)
  {}

  bool inUse_;
};

class ParticlePool
{
  Particle pool_[100];
};
```

  Это отношение описывает предполагаемый способ использования класса и обеспечивает то, что пользователь не сможет создать объект, не отслеживаемый пулом.

- *Вы можете избежать необходимости использовать флаг "используется".* У многих объектов уже есть какое-то состояние, которое можно использовать для определения живой объект или нет. Например, частица может быть доступна для повторного использования, если ее координаты находятся за экраном. Если класс объекта знает, что его можно использовать в пуле, он может предоставлять метод `inUse()` для проверки этого состояния. Таким образом, мы оберегаем пул от необходимости тратить лишнюю память для хранения флагов "используется".

Если объекты не связаны с пулом:

- *Можно помещать в пул объекты любых типов.* Это большое преимущество. Снижая связность объекта с пулом, вы имеете возможность реализовать обобщенный класс пула.
- *Состояние "используется" можно отслеживать извне объекта.* Проще всего это сделать с помощью отдельного битового флага.

```
template
class GenericPool
{
private:
  static const int POOL_SIZE = 100;

  TObject pool_[POOL_SIZE];
  bool inUse_[POOL_SIZE];
};
```

Кто отвечает за инициализацию повторно используемых объектов?

Для того, чтобы повторно использовать существующие объекты, их нужно повторно инициализировать новым состоянием. Ключевым вопросом здесь является где объект повторно инициализируется — внутри класса пула или снаружи.

Если пул повторно инициализируется внутри:

- *Пул может полностью инкапсулировать свои объекты.* В зависимости от других возможностей, нужных вашим объектам, вы можете полностью хранить их внутри пула. В таком случае, вы можете быть уверенными, что никакой другой код не будет хранить ссылку на объект пула, и его можно будет свободно использовать повторно.
- *Пул отвечает за то, как объект будет инициализирован.* Объект пула может предлагать несколько функций для своей инициализации. Если инициализацией управляет пул, его интерфейсу нужно поддерживать их все и перенаправлять вызовы объекту.

```
class Particle
{
  // Несколько способов инициализации.
  void init(double x, double y);
  void init(double x, double y, double angle);
  void init(double x, double y, double xVel, double yVel);
};
```

```
class ParticlePool
{
public:
  void create(double x, double y) {
    // Переход к частице...
  }

  void create(double x, double y, double angle) {
    // Переход к частице...
  }

  void create(double x, double y, double xVel, double yVel) {
    // Переход к частице...
  }
};
```

Если объект инициализируется из внешнего кода:

- *Интерфейс пула может быть проще.* Вместо предоставления нескольких функций для сокрытия всех возможных способов инициализации объекта, пул может просто возвращать ссылку на новый объект.

```
class Particle
{
public:
  // Несколько способов инициализации.
  void init(double x, double y);
  void init(double x, double y, double angle);
  void init(double x, double y, double xVel, double yVel);
};

class ParticlePool
{
public:
  Particle* create() {
    // Возвращаем ссылку на доступную частицу...
  }
private:
  Particle pool_[100];
};
```

  После этого вызывающий код может инициализировать объект с помощью любого метода, демонстрируемого объектом.

```
ParticlePool pool;

pool.create()->init(1, 2);
pool.create()->init(1, 2, 0.3);
pool.create()->init(1, 2, 3.3, 4.4);
```

- *Внешнему коду придется обрабатывать ошибку при создании нового объекта.* Предыдущий пример предполагает, что `create()` всегда будет успешно заканчиваться возвращением указателя на объект. Если пул переполнен, он может возвращать `NULL`. Чтобы делать это безопасно, вам нужно добавить проверку перед попыткой инициализации объекта.

```
Particle* particle = pool.create();
if (particle != NULL) particle->init(1, 2);
```

Смотрите также

- Довольно похоже на шаблон Приспособленец (Flyweight) GoF. Оба поддерживают коллекцию повторно используемых объектов. Разница в том, что означает это "повторное использование". Объекты приспособленца используются повторно, разделяя один и тот же экземпляр между множеством обладателей *одновременно*. Таким образом, мы избегаем *дублирования* использования памяти, используя один и тот же объект в нескольких контекстах.

  Объекты в пуле тоже можно использовать повторно, но только через некоторое время. "Повторное использование" в контексте пула объектов означает повторное использование памяти объекта *после* того, как предыдущий владелец ее освободит. Когда мы используем пул объектов, мы не рассчитываем, что объект будет разделяться между несколькими владельцами во время его жизни.

- Упаковка множества объектов одного типа вместе в памяти позволяет вам держать кэш процессора заполненным, пока игра обходит все его объекты. Шаблон Локальность данных (Data Locality) как раз об этом.

Пространственное разбиение

Задача

Эффективный поиск находящихся рядом объектов с помощью сохранения их в структуре данных с организацией на основе их местоположения.

Мотивация

Игры дают нам возможность посещать другие миры, но зачастую эти миры не сильно отличаются от нашего. У них схожие с нашим миром физические законы и осязаемость. Именно поэтому мы можем так свободно себя в них чувствовать, несмотря на образующие их биты и пиксели.

Один из аспектов фальшивой реальности, на котором я хочу заострить внимание — это *позиция* (*location*). Игровой мир дает ощущение пространства и объекты находятся где-то в этом *пространстве*. Это декларируется множеством способов. Очевиднее всего физика — объекты движутся, сталкиваются и взаимодействуют. Но есть и другие примеры. Аудио движок тоже должен принимать во внимание расположение источников звука по отношению к игроку, потому что издалека слышимый звук должен быть тише. Сетевая часть может ограничиваться находящимися рядом игроками.

Это значит что вашей игре часто требуется отвечать на вопрос: "Что за объекты находятся рядом?". Если, чтобы ответить на это вопрос, придется тратить слишком много времени на каждом кадре, такой поиск станет узким местом производительности нашей игры.

Боевые единицы на поле боя

Пускай мы делаем стратегию реального времени. Противоборствующие армии из сотен юнитов сталкиваются друг с другом на поле боя. Воинам нужно понимать, кого из находящихся рядом противников рубить мечом. Самым примитивным способом это узнать, будет перебор всех пар юнитов и проверка расстояния между ними:

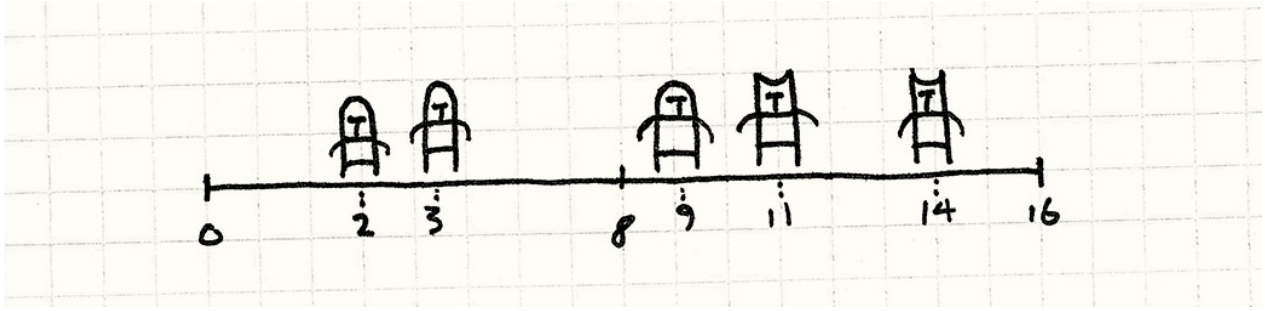
```
void handleMelee(Unit* units[], int numUnits)
{
  for (int a = 0; a < numUnits – 1; a++) {
    for (int b = a + 1; b < numUnits; b++) {
      if (units[a]->position() == units[b]->position()) {
        handleAttack(units[a], units[b]);
      }
    }
  }
}
```

У нас получился цикл с двойным вложением, обходящий на каждом из циклов все юниты на поле боя. Это значит, что нам придется выполнить количество проверок, равное *квадрату* количества юнитов. Каждый дополнительный юнит придется сравнивать со *всеми* уже существующими. Когда юнитов будет достаточно много, такое решение выйдет из под контроля.

> Внутренний цикл на самом деле не проходит все юниты. Он обходит только тех, кого еще не посетил внешний цикл. Таким образом, мы исключаем повторное сравнение юнитов и сравнение юнитов самих с собой. Если мы уже обработали коллизию между `A` и `B`, нам не нужно проверять ее между `B` и `A`.
>
> В нотации `Большой О` такой алгоритм все равно остается со сложностью $O(n^2)$.

Отрисовка боевых линий

Проблема, с которой мы столкнулись, заключается в том, что в основе расположения юнитов в массиве не заложено никакого порядка. Чтобы найти юнитов возле какого-то определенного места, нам нужно обойти весь массив. Давайте немного упростим себе задачу и упростим нашу игру. Пускай вместо `2D` *поля* боя у нас будет `1D` *линия* боя.



В таком случае, мы можем упростить себе задачу, *отсортировав* массив юнитов по их позиции на линии боя. Как только мы это сделаем, мы сможем использовать что-то наподобие бинарного поиска для обнаружения соседних юнитов без необходимости сканировать весь массив.

> Сложность бинарного поиска составляет *O(log n)*. А это значит, что сложность поиска среди сражающихся юнитов снизилась с *O(n²)* до *O(log n)*. Если применить *поиск со списком* (pigeonhole sort), ее можно снизить вообще до *O(n)*.

Думаю, урок очевиден: если мы будем хранить свои объекты в структуре данных, оптимизированной по их местоположению, мы сможем искать их гораздо быстрее. Шаблон занимается тем, что применяет эту идею к пространствам с большим чем единица количеством измерений.

Шаблон

Есть набор **объектов**, каждый из которых обладает **позицией в пространстве**. Сохраняем объекты в **пространственной структуре данных**, организованной на основе их позиций. Эта структура данных позволяет вам **эффективно запрашивать объекты, находящиеся возле указанной позиции**. Когда позиция объекта изменяется, **обновляем позиционную структуру данных** и поиск можно продолжать.

Когда использовать

Этот шаблон часто используется для хранения как живых, подвижных объектов, так и статичных декораций и геометрии игрового мира. Сложные игры обычно имеют сразу несколько структур пространственного разбиения для различных типов содержимого.

Базовым требованием для этого шаблона является наличие набора объектов, каждый из которых обладает подобием позиции и вам приходится выполнять много запросов для поиска этих объектов, от чего может пострадать производительность вашей игры.

Имейте в виду

Пространственное разбиение существует для того, чтобы заменить операции со сложностью *O(n)* или *O(n²)* на что-то более простое. Чем *больше* у вас объектов, тем полезнее такое решение. И наоборот, если `n` достаточно малое — связываться с такими вещами не стоит.

Так как этот шаблон предполагает организацию объектов по их местоположению, с объектами, которые *меняют* свою позицию, работать тяжелее. Вам нужно будет реорганизовывать структуру данных, чтобы отслеживать объекты на новых позициях,

что в свою очередь добавляет сложности *и* требует траты процессорного времени. Так что убедитесь в том, что затраты окупятся.

> Представьте себе хэш-таблицу, в которой ключи хэшированных объектов могут произвольно меняться и вы сможете себе представить, как это сложно.

Пространственное разбиение требует и дополнительной памяти для хранения структуры данных. Подобно другим оптимизациям, это компромисс между использованием памяти и скоростью выполнения.

Пример кода

Сущность шаблонов крайне *переменчива*: каждая реализация может добавлять что-то свое и пространственное разделение не является исключением. В отличие от других шаблонов, многие вариации этого шаблона подробно задокументированы. Академики обожают публиковать статьи, доказывающие их эффективность. Так как я хочу показать вам всего лишь общую концепцию шаблона, я покажу вам простейший пример пространственного разбиения: *фиксированную сетку*.

> Список наиболее часто используемых в играх разновидностей пространственного разбиения вы сможете найти в конце данной главы.

Листок бумаги в клетку

Вот базовая идея: представьте себе все поле боя целиком. А теперь наложите на него сетку с фиксированным размером ячеек, наподобие тетрадного листка в клетку. Вместо хранения юнитов в одном массиве, мы будем помещать их в ячейки сетки. Каждая клетка будет хранить список юнитов, позиции которых находятся внутри границы клетки.

Когда мы будем обрабатывать сражение, мы будем заботиться только о юнитах, находящихся в одной клетке. Вместо сравнения каждого юнита со всеми остальными юнитами в игре, мы *разбиваем* поле боя на множество более мелких полей боя, на каждом из которых гораздо меньше юнитов.

Сетка связанных юнитов

Хорошо, перейдем к коду. Для начало немного подготовительной работы. Вот наш базовый класс юнита:

```
class Unit
{
  friend class Grid;

public:
  Unit(Grid* grid, double x, double y)
    : grid_(grid), x_(x), y_(y)
  {}

  void move(double x, double y);

private:
  double x_, y_;
  Grid* grid_;
};
```

У каждого юнита есть позиция (в `2D`) и указатель на `Grid`, в которой он находится. Мы делаем `Grid` `friend` классом, потому что как вы скоро увидите, когда позиция юнита изменяется, ему придется выполнить сложный танец, чтобы сетка могла правильно обновиться.

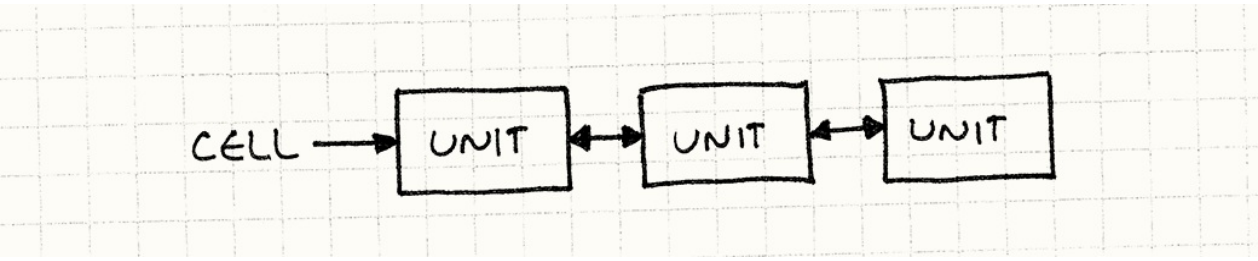
Вот набросок сетки:

```
class Grid
{
public:
  Grid() {
    // Очистка сетки.
    for (int x = 0; x < NUM_CELLS; x++) {
      for (int y = 0; y < NUM_CELLS; y++) {
        cells_[x][y] = NULL;
      }
    }
  }

  static const int NUM_CELLS = 10;
  static const int CELL_SIZE = 20;
private:
  Unit* cells_[NUM_CELLS][NUM_CELLS];
};
```

Обратите внимание, что каждая ячейка — это просто указатель на юнит. Теперь мы дополним `Unit` указателями `next` и `prev`:

```
class Unit
{
  // Предыдущий код...
private:
  Unit* prev_;
  Unit* next_;
};
```

Это даст нам возможность организовывать юниты в двусвязный список вместо массива.



Каждая ячейка в сетке указывает на первый юнит в списке юнитов данной ячейки и каждый из юнитов указывает на юнит, находящийся в списке перед ним и за ним. И скоро мы увидим для чего это нам нужно.

> На протяжении всей книги я избегал использования встроенных типов коллекций стандартной библиотеки `C++`. Я хотел минимизировать необходимые для понимания примеров в книге знания и как честный маг "ничего не прячущий в рукаве" сделать код *максимально* понятным. Дело ведь в деталях, особенно когда речь идет о паттернах, связанных с производительностью.
>
> Но такой выбор я сделал исключительно для *демонстрации* шаблонов. Когда вы будете *использовать* их в реальном коде, избавьте себя от лишней головной боли и используйте удобные коллекции, встроенные практически во все современны языки программирования. Жизнь слишком коротка, чтобы тратить ее на программирование с нуля связанных списков.

Выходим на поле боя

Первое, что нам нужно сделать — это удостовериться, что новые юниты попадают в нашу сетку при создании. Пусть это делает `Unit` в своем конструкторе:

```
Unit::Unit(Grid* grid, double x, double y)
  : grid_(grid), x_(x), y_(y)
  , prev_(NULL), next_(NULL)
{
  grid_->add(this);
}
```

Метод `add()` определяется таким образом:

```
void Grid::add(Unit* unit)
{
  // Определяем в какой ячейке сетки мы находимся.
  int cellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int cellY = (int)(unit->y_ / Grid::CELL_SIZE);

  // Добавляем в начало списка найденной ячейки.
  unit->prev_ = NULL;
  unit->next_ = cells_[cellX][cellY];
  cells_[cellX][cellY] = unit;

  if (unit->next_ != NULL) {
    unit->next_->prev_ = unit;
  }
}
```

> С помощью деления на размер ячейки мы преобразуем мировые координаты в координаты в пространстве ячеек. И, приводя результат к `int` и отбрасывая дробную часть, мы получаем индекс ячейки.

Деталей многовато, как и в коде со связанным списком выше, но базовая идея простая. Мы ищем ячейку, в которой сидит юнит и потом добавляем его в начало списка ячейки. Если там уже есть список юнитов, мы ставим его после нового юнита.

Звон мечей

Как только все юниты размещены в своих ячейках, мы можем позволить им атаковать друг друга. С использованием новой сетки, главный метод обработки боя будет выглядеть таким образом:

```
void Grid::handleMelee()
{
  for (int y = 0; y < NUM_CELLS; y++) {
    for (int x = 0; x < NUM_CELLS; x++) {
      handleCell(cells_[x][y]);
    }
  }
}
```

Он обходит каждую клетку и затем вызывает для нее `handleCell()`. Как вы видите, мы уже разбили пространство поля боя на маленькие изолированные схватки. Каждая ячейка обрабатывает бой таким образом:

```
void Grid::handleCell(Unit* unit)
{
  while (unit != NULL) {
    Unit* other = unit->next_;
    while (other != NULL) {
      if (unit->x_ == other->x_ &&
        unit->y_ == other->y_)
      {
        handleAttack(unit, other);
      }
      other = other->next_;
    }

    unit = unit->next_;
  }
}
```

Посмотрите, что мы проделываем с указателем на связанный список и обратите внимание, что это практически тоже самое, что и наш примитивный метод обработки боя: он сравнивает каждую пару юнитов и проверяет, находятся ли они в одной позиции.

Единственное различие заключается в том, что нам теперь не нужно сравнивать между собой *все* юниты, принимающие участие в битве, а только те из них, что находятся в одной клетке. Вот в чем заключается оптимизация.

> На первый взгляд может показаться, что мы только ухудшили производительность. Мы перешли от цикла обхода юнитов с двойным вложением к циклу с тройным вложением, который обходит сначала ячейки, а потом юниты. Хитрость здесь заключается в том, что два вложенных цикла теперь работают со сравнительно малым количеством юнитов, что полностью окупает появление дополнительного цикла обхода ячеек.
>
> Однако, это всё еще зависит и от дробности ячеек: если они будут слишком мелкими, влияние внешнего цикла станет более заметным.

Идем в атаку

Мы решили нашу проблему с производительностью, но создали себе новую проблему. Юниты застряли в своих ячейках. Если мы переместим юнит за границы ячейки, в которой они находятся, юниты в ячейке его больше не увидят, как и никто другой. Наше поле боя получилось *слишком* разбитым.

Чтобы это исправить, нам нужно делать небольшую работу каждый раз, когда юнит двигается. Если он пересекает линии границ ячеек, нам нужно удалить его из одной ячейки и перенести в другую. Для начала мы добавим в `Unit` метод для изменения позиции:

```
void Unit::move(double x, double y)
{
  grid_->move(this, x, y);
}
```

Предположительно, он будет вызываться кодом `AI` для управляемых компьютером юнитов и обработчиком пользовательского ввода для юнита игрока. Все, что он делает — это передает управление сетке, которая в свою очередь делает вот что:

```
void Grid::move(Unit* unit, double x, double y)
{
  // Смотрим в какой ячейке находимся.
  int oldCellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int oldCellY = (int)(unit->y_ / Grid::CELL_SIZE);

  // Смотрим в какую ячейку перемещаемся.
  int cellX = (int)(x / Grid::CELL_SIZE);
  int cellY = (int)(y / Grid::CELL_SIZE);

  unit->x_ = x;
  unit->y_ = y;
```

```
  // если ячейка не меняется =- мы закончили
  if (oldCellX == cellX && oldCellY == cellY) return;

  // Убираем юнит из списка старой ячейки.
  if (unit->prev_ != NULL) {
    unit->prev_->next_ = unit->next_;
  }
  if (unit->next_ != NULL) {
    unit->next_->prev_ = unit->prev_;
  }

  // Если это голова списка — удаляем ее.
  if (cells_[oldCellX][oldCellY] == unit) {
    cells_[oldCellX][oldCellY] = unit->next_;
  }

  // Добавление обратно в сетку в новую ячейку.
  add(unit);
}
```

Довольно много кода, но ничего сложного в нем нет. Первая часть проверяет, пересекали ли мы вообще границу ячейки. Если нет — мы можем ограничиться просто обновлением позиции юнита.

Если юнит *покинул* текущую ячейку, мы удаляем его из списка этой ячейки и добавляем обратно в сетку. Как будто мы добавляем новый юнит, мы вставляем юнит в связанный список новой ячейки.

Вот поэтому мы и будем использовать двухсвязный список: мы можем легко добавлять и удалят юниты из списка, переназначая всего несколько указателей. Когда у нас за кадр будет двигаться много юнитов — это важно.

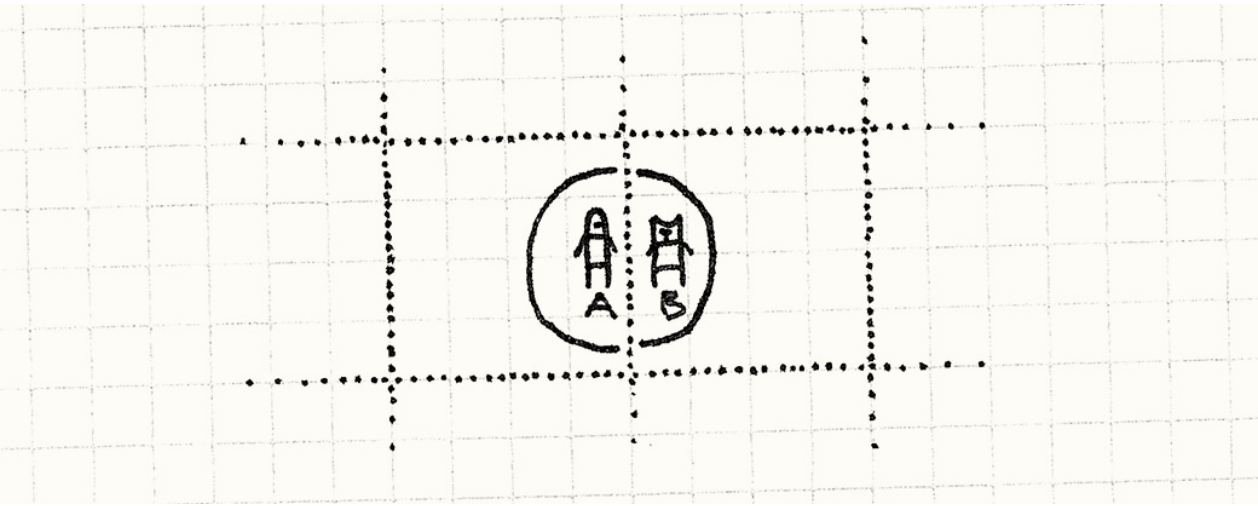
На расстоянии вытянутой руки

Выглядит просто, но все-таки я немного смухлевал. В примере, который я вам показал, юниты взаимодействуют только тогда, когда находятся *в одной и той же* позиции. Для шахмат и шашек это нормально, но не очень подходит для более реалистичных игр. Поэтому нам обычно придется учитывать *дистанцию* атаки.

В этом случае шаблон все равно работает. Вместо простой проверки равенства позиций, мы будем использовать нечто наподобие:

```
if (distance(unit, other) < ATTACK_DISTANCE) {
  handleAttack(unit, other);
}
```

Теперь, когда в дело вступает дистанция, нам нужно подумать о крайних случаях: юниты в разных ячейках могут быть все равно достаточно близки для того чтобы взаимодействовать.



Здесь `B` находится в радиусе атаки `A`, несмотря на то, что их центральные точки находятся в разных ячейках. Для обработки этого нам нужно сравнивать не только находящиеся в одной ячейке юниты, но и юниты в соседних ячейках. Для этого мы начнем с разделения внутреннего цикла `handleCell()`:

```
void Grid::handleUnit(Unit* unit, Unit* other)
{
  while (other != NULL) {
    if (distance(unit, other) < ATTACK_DISTANCE) {
      handleAttack(unit, other);
    }

    other = other->next_;
  }
}
```

Теперь у нас есть функция, которая берет один юнит и список других юнитов и проверяет их пересечение. Дальше мы изменяем handleCell() таким образом:

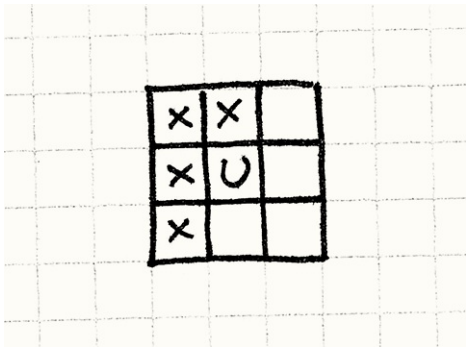
```
void Grid::handleCell(int x, int y)
{
  Unit* unit = cells_[x][y];
  while (unit != NULL) {
    // Обработка остальных юнитов в ячейке.
    handleUnit(unit, unit->next_);

    unit = unit->next_;
  }
}
```

Обратите внимание, что теперь мы передаем внутрь координаты ячейки, а не просто ее список юнитов. Пока что никаких особых отличий от предыдущего примера нет, но мы его немного расширим:

```
void Grid::handleCell(int x, int y)
{
  Unit* unit = cells_[x][y];
  while (unit != NULL) {
    // Обработка остальных юнитов в ячейке.
    handleUnit(unit, unit->next_);

    // Теперь пробуем соседние ячейки.
    if (x > 0 && y > 0) handleUnit(unit, cells_[x - 1][y - 1]);
    if (x > 0) handleUnit(unit, cells_[x - 1][y]);
    if (y > 0) handleUnit(unit, cells_[x][y - 1]);
    if (x > 0 && y < NUM_CELLS - 1) {
      handleUnit(unit, cells_[x - 1][y + 1]);
    }

    unit = unit->next_;
  }
}
```

Этот дополнительный вызов `handleUnit()` проверяет столкновения между текущим юнитом и юнитами из восьми соседних ячеек. Если любой юнит в этих соседних ячейках находится достаточно близко к границе, чтобы попасть в радиус атаки юнита, пересечение засчитывается.

> Ячейка с юнитом обозначена как `U`, а соседние ячейки как `X`.



Мы осматриваем только *половину* соседских ячеек, потому что внутренний цикл начинается *после* текущего юнита, чтобы избежать сравнение каждой пары дважды. Представьте, что произойдет, если мы начнем проверять все восемь соседних ячеек.

Представим, что у нас есть два юнита в соседних ячейках, находящиеся на расстоянии удара друг от друга как в предыдущем примере. Если для каждого юнита мы будем осматривать все восемь окружающих ячеек, произойдет вот что:

1. Когда мы нашли столкновения для `A`, мы рассмотрим соседей справа и найдем там `B`. Таким образом мы зарегистрируем бой между `A` и `B`.
2. Далее, когда мы находим столкновения для `B`, мы смотрим на соседей *слева* и находим там `A`. И регистрируем бой между `A` и `B` *второй* раз.

Исправить это можно, если просматривать только половину соседних клеток. *Какую* половину рассматривать — не имеет никакого значения.

Есть еще один граничный случай, о котором нам нужно позаботиться. При этом мы предполагаем, что радиус атаки меньше размера ячейки. Если у нас есть мелкие ячейки и большой радиус атаки, нам придется сканировать гораздо больше ячеек.

Архитектурные решения

Существует сравнительно небольшой список хорошо описанных структур данных разбиения пространства и можно было бы просто их перечислить. Но вместо этого я хочу попробовать организовать их по их основным характеристикам. Надеюсь после того, как вы узнаете про дерево квадрантов (quadtrees), *двоичное разбиение пространства* (binary space partitions, BSP) и прочие, это поможет вам понять, *как* и *почему* стоит выбирать именно их.

Иерархическое или плоское разбиение?

Наш пример разбивает пространства на плоский набор отдельных ячеек. И наоборот, иерархическое пространственное разбиение делит пространство всего на несколько областей. Дальше, если одна из этих областей по прежнему содержит много объектов, она тоже в свою очередь разбивается. Этот процесс продолжается рекурсивно, до тех пор, пока в каждой области не окажется меньше объектов, чем установленный нами максимум.

> Обычно разбиение выполняется на две, четыре, восемь частей —любимые в программировании круглые числа.

1. Если это плоское разбиение:

- *Это просто.* Плоские структуры данных проще для понимания и для реализации.

  > Об этом я говорил, и говорю практически в каждой главе: выбирайте по возможности самые простые решения. Одна из задач программирования — это борьба со сложностью.

- *Константное использование памяти.* Так как при добавлении новых объектов не нужно выполнять дополнительное разбиение, используемое пространственным разбиением количество памяти со временем не меняется.

- *Его проще обновлять, когда объект изменяет позицию.* Когда объект движется, структуру данных нужно обновить, чтобы найти объект в их новой позиции. Если вы будете использовать иерархическое разбиение пространства, для этого потребуется изменять несколько слоев иерархии.

2. Если оно иерархическое:

- *Пустое пространство обрабатывается эффективнее.* Вспомните наш первый пример и представьте, что половина поля боя у нас пустая. У нас будет множество пустых ячеек, для которых все равно выделяется память и которые все равно нужно будет проверять на каждом кадре.

  Так как иерархическое разбиение пространства не разбивает разряженное пространство, большие пустые пространства попадут в одну область. Вместо множества маленьких областей нам придется обходить несколько больших.

- *Плотно населенные пространства тоже обрабатываются эффективнее.* Это другая сторона медали: если у вас есть много объектов, скученных в одном месте, неиерархическое разбиение может оказаться неэффективным. Вы рискуете обнаружить, что в одних областях у вас находится куча объектов, а другие совсем пустые. Иерархическое разбиение адаптивно разобьет их на более мелкие области и вам снова придется иметь дело всего с несколькими объектами за раз.

Зависит ли разбиение от набора объектов?

В нашем примере кода сетка была создана предварительно и только после этого юниты были помещены в ее слоты. Другие схемы разбиения пространства работают адаптивно: они выбирают границы областей на основе реального набора объектов и их расположения в мире.

Смысл заключается в *сбалансированном* разбиении, когда в каждой области содержится примерно одинаковое количество объектов для получения лучшей производительности. Представьте, что случится, если в нашем первом примере все юниты скучкуются в одном углу поля боя. Они все окажутся в одной ячейке, и наш код поиска столкновений снова вернется к изначальной проблеме $O(n^2)$ сложности, которую мы и хотели решить.

1. Если разбиение не зависит от объектов:

- *Объекты можно добавлять инкрементно*. Добавление объекта означает поиск нужной области и помещение его туда. Поэтому объекты можно добавлять по одному без всяких последствий для производительности.
- *Объекты можно быстро перемещать*. Когда разбиение фиксировано, перемещение юнита означает удаление из одной области и добавление в другую. Если в зависимости от набора объектов изменяются сами границы областей, перемещение хотя бы одного объекта может привести к изменению границ, что в свою очередь может привести к необходимости переноса еще многих объектов.

  > Это прямая аналогия с деревьями бинарного поиска (binary search trees) типа красно-черных деревьев (red-black trees) или АВЛ деревьями (AVL trees): когда вам нужно добавить один элемент, вы можете закончить тем, что вам потребуется пересортировать все дерево и посмещать еще кучу узлов.

- *Разбиение может быть несбалансированным*. Очевидным недостатком такого разбиения является то, что оно эффективно, только когда ваши объекты распределены более-менее равномерно. Если объекты сосредоточены в одном месте, у вас начнутся проблемы с производительностью, и вы будете впустую тратить память на пустые пространства.

2. Если разбиения адаптируются к набору объектов:

Пространственное разбиение типа `BSP` и `k-мерных` деревьев (k-d trees) делит пространство рекурсивно на половинки таким образом, чтобы в каждую попадало примерно одинаковое количество объектов. Для этого вам придется подсчитывать сколько объектов попадет в каждую из областей, и выбирать подходящую плоскость в

качестве границы. Иерархия ограничивающих объемов (Bounding volume hierarchie) — это еще один тип пространственного разделения, оптимизированный для специфических наборов объектов в мире.

- *Вы можете быть уверенными, что разбиение сбалансировано*. Мы получаем не слишком хорошую, но зато *стабильную* производительность: если в каждой области примерно одинаковое количество объектов, вы можете рассчитывать на то, что все запросы в мире будут выполняться с примерно одинаковой скоростью. Когда вам нужно поддерживать стабильную частоту кадров, такое постоянство может быть поважнее чистой производительности.
- *Выгоднее выполнять разделение пространства для всего набора объектов целиком*. Когда набор объектов влияет на границы между объектами, лучше сначала получить полный набор объектов и только затем выполнять разделение. Вот почему такое разделение пространства лучше подходит для статичной геометрии, которая остается постоянно на протяжении всей игры.

3. Если разбиение не зависит от объектов, а *иерархия* зависит:

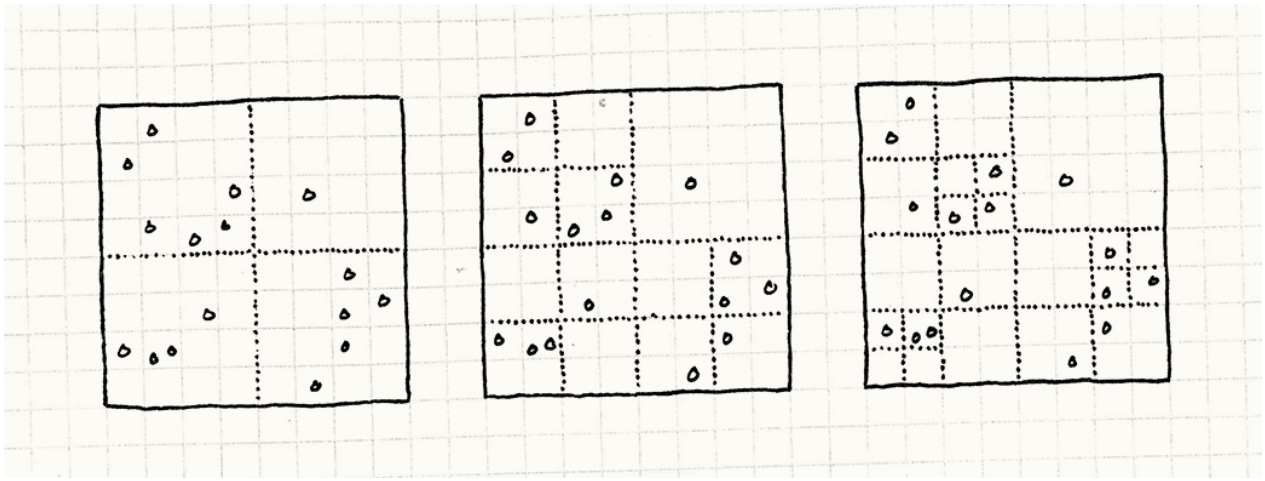
Этот тип разбиения пространства стоит особого упоминания потому, что он объединяет лучшие черты фиксированного разбиения и адаптивного: квадратичные деревья (quadtree).

> Квадратичное дерево разделяет `2D` пространство. В `3D` его аналогом является *октадерево* (octree): он берет *объем* и делит его на восемь *кубов*. За исключением одного дополнительного измерения, работает также, как и плоский родственник.

Квад-дерево начинает с одной области, занимающей все пространство. Если количество объектов в области превышает заданный порог, она делится на четыре меньших квадрата. *Границы* этих квадратов фиксированы: они всегда делят пространство на равные части.

Далее, мы повторяем тот же самый процесс для всех четырех получившихся квадратов, до тех пор, пока во всех квадратах не окажется сравнительно небольшое количество объектов. Так как мы рекурсивно делим только наиболее населенные квадраты, разбиение адаптируется под набор объектов, но границы разбиения не *смещаются*.

На рисунке ниже слева направо показан пример работы такого разбиения:



- *Объекты можно добавлять инкрементно*. Добавление новых объектов означает поиск нужного квадрата и добавление объекта в него. Если после этого количество объектов в квадрате превышает максимум, он делится еще на четыре. Другие объекты в квадрате переносятся в четыре квадрата меньшего размера. Для этого нужно приложить некоторые усилия, но их количество *фиксировано*: количество объектов, которые нужно будет переместить, всегда будет меньше, чем общее количество. Добавление одного объекта никогда не вызовет больше одного дополнительного разделения.

  Удалять объект также просто. Вы удаляете объект из квадрата и, если количество объектов в родительском квадрате теперь меньше заданного порога, мы избавляемся от дочернего разделения.

- *Объекты быстрее удаляются*. Это следует из предыдущего пункта. "Перемещение" объекта — просто удаление и добавление, что работает в квад-деревьях с вполне приемлемой скоростью.

- *Разделение сбалансировано*. Так как в каждом из квадратов разбиения объектов меньше, чем некий установленный предел, даже если объекты скучены в одном месте, вам не потребуется иметь области с большим количеством объектов внутри.

Хранятся ли объекты только в областях?

Вы можете считать разделение пространства тем *местом*, где живут объекты игры, или можете считать его всего лишь дополнительным кэшем, для ускорения поиска, тогда как сами объекты хранятся в другом списке объектов.

1. Если это единственное место, где хранятся объекты:

- *Таким образом мы избегаем дополнительного расхода памяти и сложностей поддержания двух коллекций*. Конечно, всегда проще хранить объекты в одном месте, чем в нескольких. Кроме этого, если у вас есть две коллекции, вам нужно

следить за их синхронизацией. Каждый раз, когда объект создается или уничтожается, он должен добавляться или удаляться из обеих коллекций.

2. Если для объектов есть еще одна коллекция:

- *Обход объектов выполняется быстрее.* Если наши объекты "живые" и требуют обработки, вы можете обнаружить, что вам нужно будет часто посещать объекты вне зависимости от их местонахождения. Представьте, что в нашем первом примере большинство ячеек пустое. Поэтому обход всех ячеек будет просто пустой тратой времени.

  Вторая коллекция просто хранит все объекты и позволяет вам выполнять их обход напрямую. У вас будет две структуры данных, каждая из которых оптимизирована под каждый случай.

Смотрите также

- Я старался не обсуждать специфические детали структур разбиения пространства, чтобы оставить главу достаточно высокоуровневой (и не слишком длинной!), но дальше вам придется изучать эти структуры самостоятельно. Несмотря на устрашающие названия, все они довольно прямолинейны. Вот наиболее распространенные:
  - Сетка (Grid))
  - Квад-дерево (Quadtree)
  - Двоичное разбиение пространства (BSP)
  - K-мерное дерево (k-d tree)
  - Иерархия ограниченных объемов (Bounding volume hierarchy)
- Каждая из этих структур данных для разбиения пространства чаще всего адаптирует идею хорошо известных структур для одномерного пространства на большее количество измерений. Знание их линейных родственников поможет вам легче понять как они могут решить вашу проблему:
  - Сетка — это разновидность блочной сортировки (bucket sort).
  - BSP, k-мерное дерево (k-d tree) и иерархия ограниченных объемов (Bounding volume hierarchy) — это разновидность бинарных деревьев поиска (binary search trees).
  - Квад-деревья и окта-деревья — это просто деревья (tries).