Hey, Game Developer!

- Do you struggle to make your code hang together into a cohesive whole?
- Find it harder to make changes as your codebase grows?
- Feel like your game is a giant hairball where everything is intertwined with everything else?
- Wonder if and how design patterns apply to games?
- Hear things like “cache coherency” and “object pools”, but don’t know how to use them to make your game faster?

I’m here to help! **Game Programming Patterns** is a collection of patterns I found in games that **make code cleaner, easier to understand, and faster**.

This is the book I wish I had when I started making games, and now I want you to have it. It’s available in four formats:

eBook

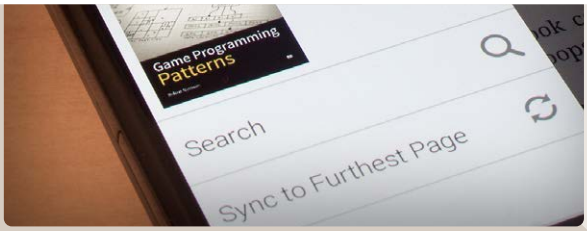
游戏编程模式中文版

Meticulously tuned CSS looks great on

as many readers as I could get my hands on.

Full-color syntax highlighting.

Works great offline!



Web

Read Now

Responsive design looks great on your desktop browser, tablet, or phone.

Free!

Absolutely zero cost!

Seriously, did I mention the price, or lack thereof?

```
<aside name="addition">
  <p>I'm sure you can f
  like.</p>
</aside>
<p>Pretty neat right? J
and evaluate arbitraril
create the right object
<aside name="ruby">
  <p>Ruby was implement
  1.9, they switched to
  time I'm saving you!<
```

Frequently Asked Questions

Do the different versions have different content?
Nope! Each format has every chapter in full, every illustration, and all of the asides you know and love. Even the free web version.

Which version pays you the most?
First of all, thank you for caring about this! Since I self-published, I set the prices so that the royalties are about the same for each format. (I also get the lion's share of the money since there's no publisher taking a cut.)

Buy the format you want and I'll get paid pretty much the same either way. If you want to give me money, but don't actually want a physical book, consider giving it to a friend or your local library. I get money, you feel good, and someone gets a free book!

If I buy the print edition, can I get the eBook cheaper?

Readers Say

"If you're a game dev programmer you need to add this site to your list of resources."

— *Ryan Leonski*

"I can't overstate how completely brilliantly written Game Programming Patterns is. And I'm only on chapter 2. Hats off."

— *Mark Richards*

"This is going to be the #1 book I recommend to new (and some old) game programmers."

— *Alistair Doulin*

Yes, mostly. I have MatchBook enabled on the *Kindle* edition. If you buy the print copy, you can get the Kindle version for just $3.00. I don't have a way to set up anything similar for the other eBook formats, unfortunately.

I am a poor student. How can I get your book cheaply?
I had you in mind when I decided to put the entire contents of the book on the web for free. I put more than five years of my life into this book, and I want as many people to have access to it as possible. The web version is also a great starting point to see if you like the book before you plunk down cash.

Do the digital editions use DRM?
Heck no! If you have been kind enough to pay for the book, I want to give you the most flexibility I can. You should be able to freely transfer it to all of your devices, archive it, etc.

I'm in Canada. How can I get the print edition?
CreateSpace does not directly ship to Canada which is why you don't see it on amazon.ca. Instead, a kind reader tells me that you can get it from Book Depository. If that doesn't work, you may be able to buy it from amazon.com or barnesandnoble.com and get it shipped from the US.

Who Am I?

I'm Bob Nystrom. I started writing this book while working at Electronic Arts. In my eight years there, I saw a lot of beautiful code, and a lot of not-so-beautiful code. My hope was that I could take what I learned from the good stuff, write it down here, and then teach it to the people writing the awful stuff.

If you want to get in touch with me, you can email `bob` at this site or just ask me (`@munificentbob`) on twitter. If you just can't get enough of my writing, I also have a blog. If you like the book, you'll probably like it too.

Keep in Touch

Part of the magic of writing a book online is that it's easy to change. If you find mistakes or have suggestions, please don't hesitate to file a bug or send me a pull request.

I'd love to be able to contact *you* too. If you put your email address in the little box, I'll let you know about updates to the book. I post less than once a month, so don't worry about me spamming you.

The mailing list!

Your email address

Sign me up!

Game Programming Patterns

Hey, Game Developer!

- Do you struggle to make your code hang together into a cohesive whole?
- Find it harder to make changes as your codebase grows?
- Feel like your game is a giant hairball where everything is intertwined with everything else?
- Wonder if and how design patterns apply to games?
- Hear things like “cache coherency” and “object pools”, but don’t know how to use them to make your game faster?

I’m here to help! I wrote this book to answer those questions. It’s a collection of patterns I found in games to make code cleaner, easier to understand, and faster.

It’s Free and Online!

This is the book I wish I had when I started making games, and I want you to have it now!

Want to read it on paper/eBook?

I’m working on those now! Join the mailing list and I’ll tell you when they’re ready:

email address

Sign me up!

(I post less than once a month. Don’t worry, I won’t spam you.)

Start Reading!

Table of Contents

Who Am I?

I'm Bob Nystrom. I started writing this book while working at Electronic Arts. In my eight years there, I saw a lot of beautiful code, and a lot of really horrendous code. My hope was that I could take what I learned from the beautiful stuff, write it down here, and then teach it to the people writing the awful stuff.

If you want to get in touch with me, you can email `bob` at this site or just ask me (`@munificentbob`) on twitter.

Have Feedback?

Part of the magic of writing a book online is that it's easy to change. If you find mistakes or have suggestions, please don't hesitate to file a bug or send me a pull request.

Table of Contents

Game Programming Patterns

Acknowledgements

Game Programming Patterns

I've heard only other authors know what's involved in writing a book, but there is another tribe who know the precise weight of that burden—those with the misfortune of being in a relationship with a writer. I wrote this in a space of time painstakingly carved from the dense rock of life for me by my wife Megan. Washing dishes and giving the kids baths may not be "writing", but without her doing those, this book wouldn't be here.

I started this project while a programmer at Electronic Arts. I don't think the company knew quite what to make of it, and I'm grateful to Michael Malone, Olivier Nallet, and Richard Wifall for supporting it and providing detailed, insightful feedback on the first few chapters.

About halfway through writing, I decided to forgo a traditional publisher. I knew that meant losing the guidance an editor brings, but I had email from dozens of readers telling me where they wanted the book to go. I'd lose proofreaders, but I had over 250 bug reports to help improve the prose. I'd give up the incentive of a writing schedule, but with readers patting my back when I finished each chapter, I had more than enough motivation.

> What I didn't lose was a copy editor. Lauren Briese showed up just when I needed her and did a wonderful job.

They call this "self publishing", but "crowd publishing" is closer to the mark. Writing can be lonely work, but I was never alone. Even when I put the book on a shelf for two years, the encouragement continued. Without the dozens of people who didn't let me forget that they were waiting for more chapters, I never would have picked it back up and finished.

> Special thanks go to Colm Sloan who pored over every single chapter in the book *twice* and gave me mountains of fantastic feedback, all out of the goodness of his own heart. I owe you a beer or twenty.

To everyone who emailed or commented, upvoted or favorited, tweeted or retweeted, anyone who reached out to me, or told a friend about the book, or sent me a bug report: my heart is filled with gratitude for you. Completing this book was one of my biggest goals

in life, and you made it happen.

Thank you!

← Previous Chapter ≡ The Book Next Chapter →

Introduction

Game Programming Patterns

In fifth grade, my friends and I were given access to a little unused classroom housing a couple of very beat-up TRS-80s. Hoping to inspire us, a teacher found a printout of some simple BASIC programs for us to tinker with.

The audio cassette drives on the computers were broken, so any time we wanted to run some code, we'd have to carefully type it in from scratch. This led us to prefer programs that were only a few lines long:

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

> Maybe if the computer prints it enough times, it will magically become true.

Even so, the process was fraught with peril. We didn't know *how* to program, so a tiny syntax error was impenetrable to us. If the program didn't work, which was often, we started over from the beginning.

At the back of the stack of pages was a real monster: a program that took up several dense pages of code. It took a while before we worked up the courage to even try it, but it was irresistible—the title above the listing was "Tunnels and Trolls". We had no idea what it did, but it sounded like a game, and what could be cooler than a computer game that you programmed yourself?

We never did get it running, and after a year, we moved out of that classroom. (Much later when I actually knew a bit of BASIC, I realized that it was just a character generator for the table-top game and not a game in itself.) But the die was cast—from there on out, I wanted to be a game programmer.

When I was in my teens, my family got a Macintosh with QuickBASIC and later THINK C. I spent almost all of my summer vacations hacking together games. Learning on my own was slow and painful. I'd get something up and running easily—maybe a map screen or a

little puzzle — but as the program grew, it got harder and harder.

> Many of my summers were also spent catching snakes and turtles in the swamps of southern Louisiana. If it wasn't so blisteringly hot outside, there's a good chance this would be a herpetology book instead of a programming one.

At first, the challenge was just getting something working. Then, it became figuring out how to write programs bigger than what would fit in my head. Instead of just reading about "How to Program in C++", I started trying to find books about how to *organize* programs.

Fast-forward several years, and a friend hands me a book: *Design Patterns: Elements of Reusable Object-Oriented Software*. Finally! The book I'd been looking for since I was a teenager. I read it cover to cover in one sitting. I still struggled with my own programs, but it was such a relief to see that other people struggled too and came up with solutions. I felt like I finally had a couple of *tools* to use instead of just my bare hands.

> This was the first time we'd met, and five minutes after being introduced, I sat down on his couch and spent the next few hours completely ignoring him and reading. I'd like to think my social skills have improved at least a little since then.

In 2001, I landed my dream job: software engineer at Electronic Arts. I couldn't wait to get a look at some real games and see how the pros put them together. What was the architecture like for an enormous game like Madden Football? How did the different systems interact? How did they get a single codebase to run on multiple platforms?

Cracking open the source code was a humbling and surprising experience. There was brilliant code in graphics, AI, animation, and visual effects. We had people who knew how to squeeze every last cycle out of a CPU and put it to good use. Stuff I didn't even know was *possible*, these people did before lunch.

But the *architecture* this brilliant code hung from was often an afterthought. They were so focused on *features* that organization went overlooked. Coupling was rife between modules. New features were often bolted onto the codebase wherever they could be made to fit. To my disillusioned eyes, it looked like many programmers, if they ever cracked open *Design Patterns* at all, never got past Singleton.

Of course, it wasn't really that bad. I'd imagined game programmers sitting in some ivory tower covered in whiteboards, calmly discussing architectural minutiae for weeks on end. The reality was that the code I was looking at was written by people working to meet intense deadlines. They did the best they could, and, as I gradually realized, their best was often very good. The more time I spent working on game code, the more bits of brilliance I found hiding under the surface.

Unfortunately, “hiding” was often a good description. There were gems buried in the code, but many people walked right over them. I watched coworkers struggle to reinvent good solutions when examples of exactly what they needed were nestled in the same codebase they were standing on.

That problem is what this book aims to solve. I dug up and polished the best patterns I’ve found in games, and presented them here so that we can spend our time inventing new things instead of *re*-inventing them.

What’s in Store

There are already dozens of game programming books out there. Why write another?

Most game programming books I’ve seen fall into one of two categories:

- **Domain-specific books.** These narrowly-focused books give you a deep dive on some specific aspect of game development. They’ll teach you about 3D graphics, real-time rendering, physics simulation, artificial intelligence, or audio. These are the areas that many game programmers specialize in as their careers progress.

- **Whole-engine books.** In contrast, these try to span all of the different parts of an entire game engine. They are oriented towards building a complete engine suited to some specific genre of game, usually a 3D first-person shooter.

I like both of these kinds of books, but I think they leave some gaps. Books specific to a domain rarely tell you how that chunk of code interacts with the rest of the game. You may be a wizard at physics and rendering, but do you know how to tie them together gracefully?

The second category covers that, but I often find whole-engine books to be too monolithic and too genre-specific. Especially with the rise of mobile and casual gaming, we’re in a period where lots of different genres of games are being created. We aren’t all just cloning Quake anymore. Books that walk you through a single engine aren’t helpful when *your* game doesn’t fit that mold.

Instead, what I’m trying to do here is more *à la carte*. Each of the chapters in this book is an independent idea that you can apply to your code. This way, you can mix and match them in a way that works best for the game *you* want to make.

> Another example of this *à la carte* style is the widely beloved *Game Programming Gems* series.

How it Relates to Design Patterns

Any programming book with "Patterns" in its name clearly bears a relationship to the classic *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (ominously called the "Gang of Four").

> *Design Patterns* itself was in turn inspired by a previous book. The idea of crafting a language of patterns to describe open-ended solutions to problems comes from *A Pattern Language*, by Christopher Alexander (along with Sarah Ishikawa and Murray Silverstein).
>
> Their book was about architecture (like *real* architecture with buildings and walls and stuff), but they hoped others would use the same structure to describe solutions in other fields. *Design Patterns* is the Gang of Four's attempt to do that for software.

By calling this book "Game Programming Patterns", I'm not trying to imply that the Gang of Four's book is inapplicable to games. On the contrary: the Design Patterns Revisited section of this book covers many of the patterns from *Design Patterns*, but with an emphasis on how they can be applied to game programming.

Conversely, I think this book is applicable to non-game software too. I could just as well have called this book *More Design Patterns*, but I think games make for more engaging examples. Do you really want to read yet another book about employee records and bank accounts?

That being said, while the patterns introduced here are useful in other software, I think they're particularly well-suited to engineering challenges commonly encountered in games:

- Time and sequencing are often a core part of a game's architecture. Things must happen in the right order and at the right time.
- Development cycles are highly compressed, and a number of programmers need to be able to rapidly build and iterate on a rich set of different behavior without stepping on each other's toes or leaving footprints all over the codebase.
- After all of this behavior is defined, it starts interacting. Monsters bite the hero, potions are mixed together, and bombs blast enemies and friends alike. Those interactions must happen without the codebase turning into an intertwined hairball.
- And, finally, performance is critical in games. Game developers are in a constant race to see who can squeeze the most out of their platform. Tricks for shaving off cycles can mean the difference between an A-rated game and millions of sales or dropped frames and angry reviewers.

How to Read the Book

Game Programming Patterns is divided into three broad sections. The first introduces and frames the book. It's the chapter you're reading now along with the next one.

The second section, Design Patterns Revisited, goes through a handful of patterns from the Gang of Four book. With each chapter, I give my spin on a pattern and how I think it relates to game programming.

The last section is the real meat of the book. It presents thirteen design patterns that I've found useful. They're grouped into four categories: Sequencing Patterns, Behavioral Patterns, Decoupling Patterns, and Optimization Patterns.

Each of these patterns is described using a consistent structure so that you can use this book as a reference and quickly find what you need:

- The **Intent** section provides a snapshot description of the pattern in terms of the problem it intends to solve. This is first so that you can hunt through the book quickly to find a pattern that will help you with your current struggle.
- The **Motivation** section describes an example problem that we will be applying the pattern to. Unlike concrete algorithms, a pattern is usually formless unless applied to some specific problem. Teaching a pattern without an example is like teaching baking without mentioning dough. This section provides the dough that the later sections will bake.
- The **Pattern** section distills the essence of the pattern out of the previous example. If you want a dry textbook description of the pattern, this is it. It's also a good refresher if you're familiar with a pattern already and want to make sure you don't forget an ingredient.
- So far, the pattern has only been explained in terms of a single example. But how do you know if the pattern will be good for *your* problem? The **When to Use It** section provides some guidelines on when the pattern is useful and when it's best avoided. The **Keep in Mind** section points out consequences and risks when using the pattern.
- If, like me, you need concrete examples to really *get* something, then **Sample Code** is your section. It walks step by step through a full implementation of the pattern so you can see exactly how it works.
- Patterns differ from single algorithms because they are open-ended. Each time you use a pattern, you'll likely implement it differently. The next section, **Design Decisions**, explores that space and shows you different options to consider when applying a pattern.
- To wrap it up, there's a short **See Also** section that shows how this pattern relates to others and points you to real-world open source code that uses it.

About the Sample Code

Code samples in this book are in C++, but that isn't to imply that these patterns are only useful in that language or that C++ is a better language for them than others. Almost any language will work fine, though some patterns do tend to presume your language has objects and classes.

I chose C++ for a couple of reasons. First, it's the most popular language for commercially shipped games. It is the *lingua franca* of the industry. Moreso, the C syntax that C++ is based on is also the basis for Java, C#, JavaScript, and many other languages. Even if you don't know C++, the odds are good you can understand the code samples here with a little bit of effort.

The goal of this book is *not* to teach you C++. The samples are kept as simple as possible and don't represent good C++ style or usage. Read the code samples for the idea being expressed, not the code expressing it.

In particular, the code is not written in "modern"—C++11 or newer—style. It does not use the standard library and rarely uses templates. This makes for "bad" C++ code, but I hope that by keeping it stripped down, it will be more approachable to people coming from C, Objective-C, Java, and other languages.

To avoid wasting space on code you've already seen or that isn't relevant to the pattern, code will sometimes be omitted in examples. When this occurs, an ellipsis will be placed in the sample to show where the missing code goes.

Consider a function that will do some work and then return a value. The pattern being explained is only concerned with the return value, and not the work being done. In that case, the sample code will look like:

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

Where to Go From Here

Patterns are a constantly changing and expanding part of software development. This book continues the process started by the Gang of Four of documenting and sharing the software patterns they saw, and that process will continue after the ink dries on these pages.

You are a core part of that process. As you develop your own patterns and refine (or refute!) the patterns in this book, you contribute to the software community. If you have

suggestions, corrections, or other feedback about what's in here, please get in touch!

← Previous Chapter ≡ The Book Next Chapter →

Architecture, Performance, and Games

Before we plunge headfirst into a pile of patterns, I thought it might help to give you some context about how I think about software architecture and how it applies to games. It may help you understand the rest of this book better. If nothing else, when you get dragged into an argument about how terrible (or awesome) design patterns and software architecture are, it will give you some ammo to use.

> Note that I didn't presume which side you're taking in that fight. Like any arms dealer, I have wares for sale to all combatants.

What is Software Architecture?

If you read this book cover to cover, you won't come away knowing the linear algebra behind 3D graphics or the calculus behind game physics. It won't show you how to alpha-beta prune your AI's search tree or simulate a room's reverberation in your audio playback.

> Wow, this paragraph would make a terrible ad for the book.

Instead, this book is about the code *between* all of that. It's less about writing code than it is about *organizing* it. Every program has *some* organization, even if it's just "jam the whole thing into `main()` and see what happens", so I think it's more interesting to talk about what makes for *good* organization. How do we tell a good architecture from a bad one?

I've been mulling over this question for about five years. Of course, like you, I have an

intuition about good design. We've all suffered through codebases so bad, the best you could hope to do for them is take them out back and put them out of their misery.

> Let's admit it, most of us are *responsible* for a few of those.

A lucky few have had the opposite experience, a chance to work with beautifully designed code. The kind of codebase that feels like a perfectly appointed luxury hotel festooned with concierges waiting eagerly on your every whim. What's the difference between the two?

What is *good* software architecture?

For me, good design means that when I make a change, it's as if the entire program was crafted in anticipation of it. I can solve a task with just a few choice function calls that slot in perfectly, leaving not the slightest ripple on the placid surface of the code.

That sounds pretty, but it's not exactly actionable. "Just write your code so that changes don't disturb its placid surface." Right.

Let me break that down a bit. The first key piece is that *architecture is about change*. Someone has to be modifying the codebase. If no one is touching the code—whether because it's perfect and complete or so wretched no one will sully their text editor with it—its design is irrelevant. The measure of a design is how easily it accommodates changes. With no changes, it's a runner who never leaves the starting line.

How do you make a change?

Before you can change the code to add a new feature, to fix a bug, or for whatever reason caused you to fire up your editor, you have to understand what the existing code is doing. You don't have to know the whole program, of course, but you need to load all of the relevant pieces of it into your primate brain.

> It's weird to think that this is literally an OCR process.

We tend to gloss over this step, but it's often the most time-consuming part of programming. If you think paging some data from disk into RAM is slow, try paging it into a simian cerebrum over a pair of optical nerves.

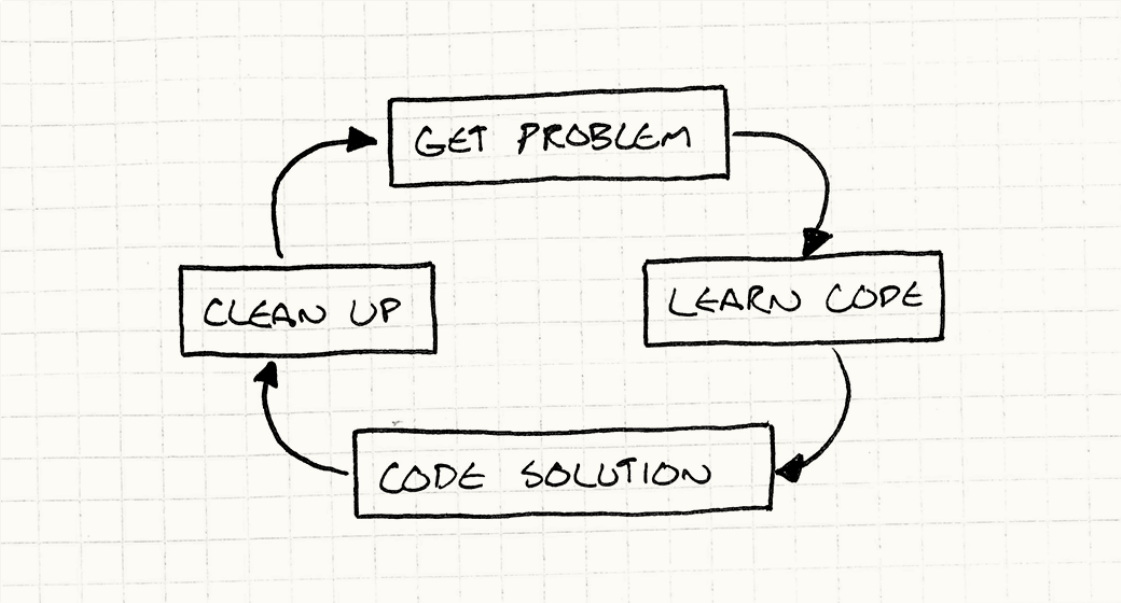
Once you've got all the right context into your wetware, you think for a bit and figure out your solution. There can be a lot of back and forth here, but often this is relatively straightforward. Once you understand the problem and the parts of the code it touches, the actual coding is sometimes trivial.

You beat your meaty fingers on the keyboard for a while until the right colored lights blink on screen and you're done, right? Not just yet! Before you write tests and send it off for code review, you often have some cleanup to do.

> Did I say "tests"? Oh, yes, I did. It's hard to write unit tests for some game code, but a large fraction of the codebase is perfectly testable.
>
> I won't get on a soapbox here, but I'll ask you to consider doing more automated testing if you aren't already. Don't you have better things to do than manually validate stuff over and over again?

You jammed a bit more code into your game, but you don't want the next person to come along to trip over the wrinkles you left throughout the source. Unless the change is minor, there's usually a bit of reorganization to do to make your new code integrate seamlessly with the rest of the program. If you do it right, the next person to come along won't be able to tell when any line of code was written.

In short, the flow chart for programming is something like:



> The fact that there is no escape from that loop is a little alarming now that I think about it.

How can decoupling help?

While it isn't obvious, I think much of software architecture is about that learning phase. Loading code into neurons is so painfully slow that it pays to find strategies to reduce the volume of it. This book has an entire section on *decoupling* patterns, and a large chunk of *Design Patterns* is about the same idea.

You can define "decoupling" a bunch of ways, but I think if two pieces of code are coupled, it means you can't understand one without understanding the other. If you *de*-couple them, you can reason about either side independently. That's great because if only one of those pieces is relevant to your problem, you just need to load *it* into your monkey brain and not the other half too.

To me, this is a key goal of software architecture: **minimize the amount of knowledge you need to have in-cranium before you can make progress.**

The later stages come into play too, of course. Another definition of decoupling is that a *change* to one piece of code doesn't necessitate a change to another. We obviously need to change *something*, but the less coupling we have, the less that change ripples throughout the rest of the game.

At What Cost?

This sounds great, right? Decouple everything and you'll be able to code like the wind. Each change will mean touching only one or two select methods, and you can dance across the surface of the codebase leaving nary a shadow.

This feeling is exactly why people get excited about abstraction, modularity, design patterns, and software architecture. A well-architected program really is a joyful experience to work in, and everyone loves being more productive. Good architecture makes a *huge* difference in productivity. It's hard to overstate how profound an effect it can have.

But, like all things in life, it doesn't come free. Good architecture takes real effort and discipline. Every time you make a change or implement a feature, you have to work hard to integrate it gracefully into the rest of the program. You have to take great care to both organize the code well and *keep* it organized throughout the thousands of little changes that make up a development cycle.

> The second half of this—maintaining your design—deserves special attention. I've seen many programs start out beautifully and then die a death of a thousand cuts as programmers add "just one tiny little hack" over and over again.
>
> Like gardening, it's not enough to put in new plants. You must also weed and prune.

You have to think about which parts of the program should be decoupled and introduce abstractions at those points. Likewise, you have to determine where extensibility should be engineered in so future changes are easier to make.

People get really excited about this. They envision future developers (or just their future self) stepping into the codebase and finding it open-ended, powerful, and just beckoning to be extended. They imagine The One Game Engine To Rule Them All.

But this is where it starts to get tricky. Whenever you add a layer of abstraction or a place where extensibility is supported, you're *speculating* that you will need that flexibility later. You're adding code and complexity to your game that takes time to develop, debug, and

maintain.

That effort pays off if you guess right and end up touching that code later. But predicting the future is *hard*, and when that modularity doesn't end up being helpful, it quickly becomes actively harmful. After all, it is more code you have to deal with.

> Some folks coined the term "YAGNI"—You aren't gonna need it—as a mantra to use to fight this urge to speculate about what your future self may want.

When people get overzealous about this, you get a codebase whose architecture has spiraled out of control. You've got interfaces and abstractions everywhere. Plug-in systems, abstract base classes, virtual methods galore, and all sorts of extension points.

It takes you forever to trace through all of that scaffolding to find some real code that does something. When you need to make a change, sure, there's probably an interface there to help, but good luck finding it. In theory, all of this decoupling means you have less code to understand before you can extend it, but the layers of abstraction themselves end up filling your mental scratch disk.

Codebases like this are what turn people *against* software architecture, and design patterns in particular. It's easy to get so wrapped up in the code itself that you lose sight of the fact that you're trying to ship a *game*. The siren song of extensibility sucks in countless developers who spend years working on an "engine" without ever figuring out what it's an engine *for*.

Performance and Speed

There's another critique of software architecture and abstraction that you hear sometimes, especially in game development: that it hurts your game's performance. Many patterns that make your code more flexible rely on virtual dispatch, interfaces, pointers, messages, and other mechanisms that all have at least some runtime cost.

> One interesting counter-example is templates in C++. Template metaprogramming can sometimes give you the abstraction of interfaces without any penalty at runtime.
>
> There's a spectrum of flexibility here. When you write code to call a concrete method in some class, you're fixing that class at *author* time—you've hard-coded which class you call into. When you go through a virtual method or interface, the class that gets called isn't known until *runtime*. That's much more flexible but implies some runtime overhead.
>
> Template metaprogramming is somewhere between the two. There, you make the decision of which class to call at *compile time* when the template is instantiated.

There's a reason for this. A lot of software architecture is about making your program more flexible. It's about making it take less effort to change it. That means encoding fewer assumptions in the program. You use interfaces so that your code works with *any* class that implements it instead of just the one that does today. You use observers [GoF] and messaging [] to let two parts of the game talk to each other so that tomorrow, it can easily be three or four.

But performance is all about assumptions. The practice of optimization thrives on concrete limitations. Can we safely assume we'll never have more than 256 enemies? Great, we can pack an ID into a single byte. Will we only call a method on one concrete type here? Good, we can statically dispatch or inline it. Are all of the entities going to be the same class? Great, we can make a nice contiguous array [] of them.

This doesn't mean flexibility is bad, though! It lets us change our game quickly, and *development* speed is absolutely vital for getting to a fun experience. No one, not even Will Wright, can come up with a balanced game design on paper. It demands iteration and experimentation.

The faster you can try out ideas and see how they feel, the more you can try and the more likely you are to find something great. Even after you've found the right mechanics, you need plenty of time for tuning. A tiny imbalance can wreck the fun of a game.

There's no easy answer here. Making your program more flexible so you can prototype faster will have some performance cost. Likewise, optimizing your code will make it less flexible.

My experience, though, is that it's easier to make a fun game fast than it is to make a fast game fun. One compromise is to keep the code flexible until the design settles down and then tear out some of the abstraction later to improve your performance.

The Good in Bad Code

That brings me to the next point which is that there's a time and place for different styles of coding. Much of this book is about making maintainable, clean code, so my allegiance is pretty clearly to doing things the "right" way, but there's value in slapdash code too.

Writing well-architected code takes careful thought, and that translates to time. Moreso, *maintaining* a good architecture over the life of a project takes a lot of effort. You have to treat your codebase like a good camper does their campsite: always try to leave it a little better than you found it.

This is good when you're going to be living in and working on that code for a long time. But, like I mentioned earlier, game design requires a lot of experimentation and exploration. Especially early on, it's common to write code that you *know* you'll throw

away.

If you just want to find out if some gameplay idea plays right at all, architecting it beautifully means burning more time before you actually get it on screen and get some feedback. If it ends up not working, that time spent making the code elegant goes to waste when you delete it.

Prototyping—slapping together code that's just barely functional enough to answer a design question—is a perfectly legitimate programming practice. There is a very large caveat, though. If you write throwaway code, you *must* ensure you're able to throw it away. I've seen bad managers play this game time and time again:

> *Boss: "Hey, we've got this idea that we want to try out. Just a prototype, so don't feel you need to do it right. How quickly can you slap something together?"*
>
> *Dev: "Well, if I cut lots of corners, don't test it, don't document it, and it has tons of bugs, I can give you some temp code in a few days."*
>
> *Boss: "Great!"*

A few days pass...

> *Boss: "Hey, that prototype is great. Can you just spend a few hours cleaning it up a bit now and we'll call it the real thing?"*

You need to make sure the people using the throwaway code understand that even though it kind of looks like it works, it *cannot* be maintained and *must* be rewritten. If there's a *chance* you'll end up having to keep it around, you may have to defensively write it well.

> One trick to ensuring your prototype code isn't obliged to become real code is to write it in a language different from the one your game uses. That way, you have to rewrite it before it can end up in your actual game.

Striking a Balance

We have a few forces in play:

1. We want nice architecture so the code is easier to understand over the lifetime of the project.
2. We want fast runtime performance.
3. We want to get today's features done quickly.

> I think it's interesting that these are all about some kind of speed: our long-term development speed, the game's execution speed, and our short-term development speed.

These goals are at least partially in opposition. Good architecture improves productivity over the long term, but maintaining it means every change requires a little more effort to keep things clean.

The implementation that's quickest to write is rarely the quickest to *run*. Instead, optimization takes significant engineering time. Once it's done, it tends to calcify the codebase: highly optimized code is inflexible and very difficult to change.

There's always pressure to get today's work done today and worry about everything else tomorrow. But if we cram in features as quickly as we can, our codebase will become a mess of hacks, bugs, and inconsistencies that saps our future productivity.

There's no simple answer here, just trade-offs. From the email I get, this disheartens a lot of people. Especially for novices who just want to make a game, it's intimidating to hear, "There is no right answer, just different flavors of wrong."

But, to me, this is exciting! Look at any field that people dedicate careers to mastering, and in the center you will always find a set of intertwined constraints. After all, if there was an easy answer, everyone would just do that. A field you can master in a week is ultimately boring. You don't hear of someone's distinguished career in ditch digging.

> Maybe you do; I didn't research that analogy. For all I know, there could be avid ditch digging hobbyists, ditch digging conventions, and a whole subculture around it. Who am I to judge?

To me, this has much in common with games themselves. A game like chess can never be mastered because all of the pieces are so perfectly balanced against one another. This means you can spend your life exploring the vast space of viable strategies. A poorly designed game collapses to the one winning tactic played over and over until you get bored and quit.

Simplicity

Lately, I feel like if there is any method that eases these constraints, it's *simplicity*. In my code today, I try very hard to write the cleanest, most direct solution to the problem. The kind of code where after you read it, you understand exactly what it does and can't imagine any other possible solution.

I aim to get the data structures and algorithms right (in about that order) and then go from there. I find if I can keep things simple, there's less code overall. That means less code to load into my head in order to change it.

It often runs fast because there's simply not as much overhead and not much code to execute. (This certainly isn't always the case though. You can pack a lot of looping and

recursion in a tiny amount of code.)

However, note that I'm not saying simple code takes less time to *write*. You'd think it would since you end up with less total code, but a good solution isn't an accretion of code, it's a *distillation* of it.

> Blaise Pascal famously ended a letter with, "I would have written a shorter letter, but I did not have the time."
>
> Another choice quote comes from Antoine de Saint-Exupery: "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."
>
> Closer to home, I'll note that every time I revise a chapter in this book, it gets shorter. Some chapters are tightened by 20% by the time they're done.

We're rarely presented with an elegant problem. Instead, it's a pile of use cases. You want the X to do Y when Z, but W when A, and so on. In other words, a long list of different example behaviors.

The solution that takes the least mental effort is to just code up those use cases one at a time. If you look at novice programmers, that's what they often do: they churn out reams of conditional logic for each case that popped into their head.

But there's nothing elegant in that, and code in that style tends to fall over when presented with input even slightly different than the examples the coder considered. When we think of elegant solutions, what we often have in mind is a *general* one: a small bit of logic that still correctly covers a large space of use cases.

Finding that is a bit like pattern matching or solving a puzzle. It takes effort to see through the scattering of example use cases to find the hidden order underlying them all. It's a great feeling when you pull it off.

Get On With It, Already

Almost everyone skips the introductory chapters, so I congratulate you on making it this far. I don't have much in return for your patience, but I'll offer up a few bits of advice that I hope may be useful to you:

- Abstraction and decoupling make evolving your program faster and easier, but don't waste time doing them unless you're confident the code in question needs that flexibility.
- Think about and design for performance throughout your development cycle, but put off the low-level, nitty-gritty optimizations that lock assumptions into your code until

as late as possible.

> Trust me, two months before shipping is *not* when you want to start worrying about that nagging little "game only runs at 1 FPS" problem.

- Move quickly to explore your game's design space, but don't go so fast that you leave a mess behind you. You'll have to live with it, after all.
- If you are going to ditch code, don't waste time making it pretty. Rock stars trash hotel rooms because they know they're going to check out the next day.
- But, most of all, **if you want to make something fun, have fun making it.**

← Previous Chapter ≡ The Book Next Chapter →

Design Patterns Revisited

Game Programming Patterns

Design Patterns: Elements of Reusable Object-Oriented Software is nearly twenty years old by my watch. Unless you're looking over my shoulder, there's a good chance *Design Patterns* will be old enough to drink by the time you read this. For an industry as quickly moving as software, that's practically ancient. The enduring popularity of the book says something about how timeless design is compared to many frameworks and methodologies.

While I think *Design Patterns* is still relevant, we've learned a lot in the past couple of decades. In this section, we'll walk through a handful of the original patterns the Gang of Four documented. For each pattern, I hope to have something useful or interesting to say.

I think some patterns are overused (Singleton), while others are underappreciated (Command). A couple are in here because I want to explore their relevance specifically to games (Flyweight and Observer). Finally, sometimes I just think it's fun to see how patterns are enmeshed in the larger field of programming (Prototype and State).

The Patterns

- Command
- Flyweight
- Observer
- Prototype
- Singleton
- State

Command

Command is one of my favorite patterns. Most large programs I write, games or otherwise, end up using it somewhere. When I've used it in the right place, it's neatly untangled some really gnarly code. For such a swell pattern, the Gang of Four has a predictably abstruse description:

> *Encapsulate a request as an object, thereby letting users parameterize clients with different requests, queue or log requests, and support undoable operations.*

I think we can all agree that that's a terrible sentence. First of all, it mangles whatever metaphor it's trying to establish. Outside of the weird world of software where words can mean anything, a "client" is a *person*—someone you do business with. Last I checked, human beings can't be "parameterized".

Then, the rest of that sentence is just a list of stuff you could maybe possibly use the pattern for. Not very illuminating unless your use case happens to be in that list. *My* pithy tagline for the Command pattern is:

A command is a *reified method call.*

> "Reify" comes from the Latin "res", for "thing", with the English suffix "-fy". So it basically means "thingify", which, honestly, would be a more fun word to use.

Of course, "pithy" often means "impenetrably terse", so this may not be much of an improvement. Let me unpack that a bit. "Reify", in case you've never heard it, means "make real". Another term for reifying is making something "first-class".

> *Reflection systems* in some languages let you work with the types in your program imperatively at runtime. You can get an object that represents the class of some other object, and you can play with that to see what the type can do. In other words, reflection is a *reified type system*.

Both terms mean taking some *concept* and turning it into a piece of *data*—an object—that you can stick in a variable, pass to a function, etc. So by saying the Command pattern is a "reified method call", what I mean is that it's a method call wrapped in an object.

That sounds a lot like a "callback", "first-class function", "function pointer", "closure", or "partially applied function" depending on which language you're coming from, and indeed those are all in the same ballpark. The Gang of Four later says:

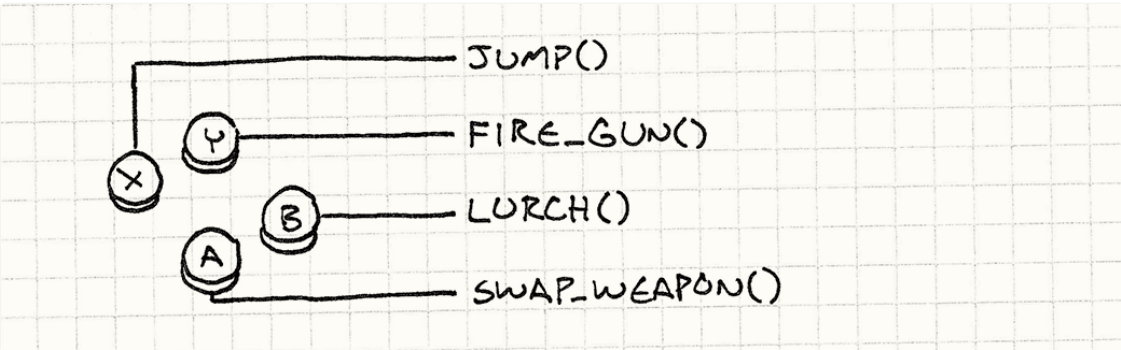
> *Commands are an object-oriented replacement for callbacks.*

That would be a better slugline for the pattern than the one they chose.

But all of this is abstract and nebulous. I like to start chapters with something concrete, and I blew that. To make up for it, from here on out it's all examples where commands are a brilliant fit.

Configuring Input

Somewhere in every game is a chunk of code that reads in raw user input — button presses, keyboard events, mouse clicks, whatever. It takes each input and translates it to a meaningful action in the game:



A dead simple implementation looks like:

```
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();
}
```

> Pro tip: Don't press B very often.

This function typically gets called once per frame by the Game Loop, and I'm sure you can figure out what it does. This code works if we're willing to hard-wire user inputs to

game actions, but many games let the user *configure* how their buttons are mapped.

To support that, we need to turn those direct calls to `jump()` and `fireGun()` into something that we can swap out. "Swapping out" sounds a lot like assigning a variable, so we need an *object* that we can use to represent a game action. Enter: the Command pattern.

We define a base class that represents a triggerable game command:

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

> When you have an interface with a single method that doesn't return anything, there's a good chance it's the Command pattern.

Then we create subclasses for each of the different game actions:

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

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

// You get the idea...
```

In our input handler, we store a pointer to a command for each button:

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

  // Methods to bind commands...

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

```
  Command* buttonB_;
};
```

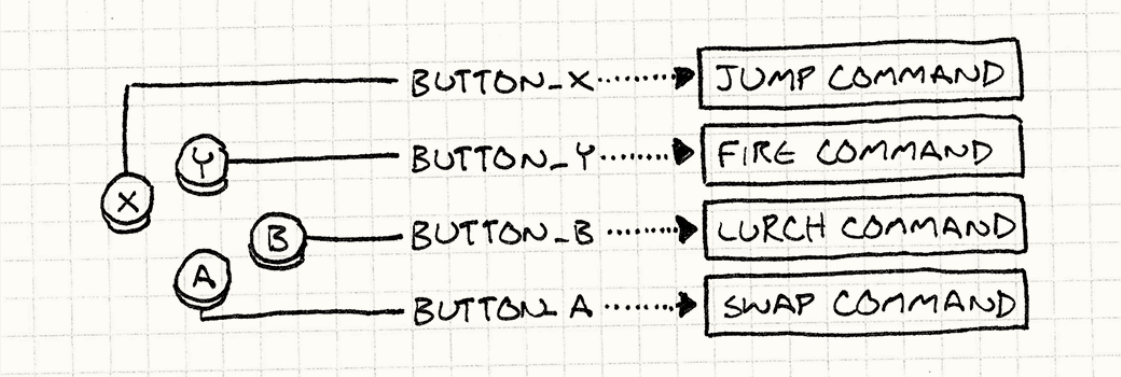
Now the input handling just delegates to those:

```
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();
}
```

Notice how we don't check for `NULL` here? This assumes each button will have *some* command wired up to it.

If we want to support buttons that do nothing without having to explicitly check for `NULL`, we can define a command class whose `execute()` method does nothing. Then, instead of setting a button handler to `NULL`, we point it to that object. This is a pattern called Null Object.

Where each input used to directly call a function, now there's a layer of indirection:



This is the Command pattern in a nutshell. If you can see the merit of it already, consider the rest of this chapter a bonus.

Directions for Actors

The command classes we just defined work for the previous example, but they're pretty limited. The problem is that they assume there are these top-level `jump()`, `fireGun()`, etc. functions that implicitly know how to find the player's avatar and make him dance like the puppet he is.

That assumed coupling limits the usefulness of those commands. The *only* thing the `JumpCommand` can make jump is the player. Let's loosen that restriction. Instead of calling functions that find the commanded object themselves, we'll *pass in* the object that we

want to order around:

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

Here, `GameActor` is our "game object" class that represents a character in the game world. We pass it in to `execute()` so that the derived command can invoke methods on an actor of our choice, like so:

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

Now, we can use this one class to make any character in the game hop around. We're just missing a piece between the input handler and the command that takes the command and invokes it on the right object. First, we change `handleInput()` so that it *returns* commands:

```
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_;

  // Nothing pressed, so do nothing.
  return NULL;
}
```

It can't execute the command immediately since it doesn't know what actor to pass in. Here's where we take advantage of the fact that the command is a reified call—we can *delay* when the call is executed.

Then, we need some code that takes that command and runs it on the actor representing the player. Something like:

```
Command* command = inputHandler.handleInput();
if (command)
```

```
{
  command->execute(actor);
}
```

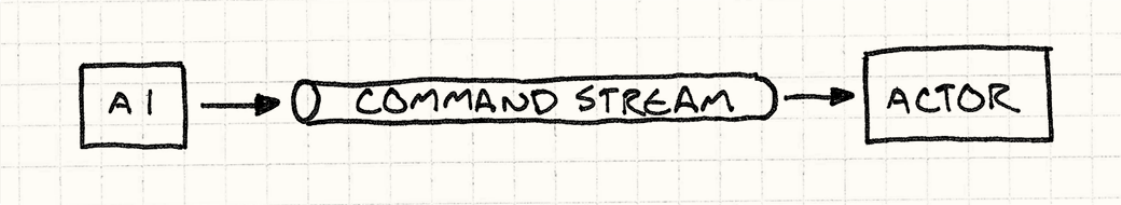
Assuming `actor` is a reference to the player's character, this correctly drives him based on the user's input, so we're back to the same behavior we had in the first example. But adding a layer of indirection between the command and the actor that performs it has given us a neat little ability: *we can let the player control any actor in the game now by changing the actor we execute the commands on.*

In practice, that's not a common feature, but there is a similar use case that *does* pop up frequently. So far, we've only considered the player-driven character, but what about all of the other actors in the world? Those are driven by the game's AI. We can use this same command pattern as the interface between the AI engine and the actors; the AI code simply emits `Command` objects.

The decoupling here between the AI that selects commands and the actor code that performs them gives us a lot of flexibility. We can use different AI modules for different actors. Or we can mix and match AI for different kinds of behavior. Want a more aggressive opponent? Just plug-in a more aggressive AI to generate commands for it. In fact, we can even bolt AI onto the *player's* character, which can be useful for things like demo mode where the game needs to run on auto-pilot.

By making the commands that control an actor first-class objects, we've removed the tight coupling of a direct method call. Instead, think of it as a queue or stream of commands:

> For lots more on what queueing can do for you, see Event Queue.



> Why did I feel the need to draw a picture of a "stream" for you? And why does it look like a tube?

Some code (the input handler or AI) produces commands and places them in the stream. Other code (the dispatcher or actor itself) consumes commands and invokes them. By sticking that queue in the middle, we've decoupled the producer on one end from the consumer on the other.

> If we take those commands and make them *serializable*, we can send the stream of them over the network. We can take the player's input, push it over the network to another machine, and then replay it. That's one

> important piece of making a networked multi-player game.

Undo and Redo

The final example is the most well-known use of this pattern. If a command object can *do* things, it's a small step for it to be able to *undo* them. Undo is used in some strategy games where you can roll back moves that you didn't like. It's *de rigueur* in tools that people use to *create* games. The surest way to make your game designers hate you is giving them a level editor that can't undo their fat-fingered mistakes.

> I may be speaking from experience here.

Without the Command pattern, implementing undo is surprisingly hard. With it, it's a piece of cake. Let's say we're making a single-player, turn-based game and we want to let users undo moves so they can focus more on strategy and less on guesswork.

We're conveniently already using commands to abstract input handling, so every move the player makes is already encapsulated in them. For example, moving a unit may look like:

```
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_;
};
```

Note this is a little different from our previous commands. In the last example, we wanted to *abstract* the command from the actor that it modified. In this case, we specifically want to *bind* it to the unit being moved. An instance of this command isn't a general "move something" operation that you could use in a bunch of contexts; it's a specific concrete move in the game's sequence of turns.

This highlights a variation in how the Command pattern gets implemented. In some cases, like our first couple of examples, a command is a reusable object that represents a *thing*

that can be done. Our earlier input handler held on to a single command object and called its `execute()` method anytime the right button was pressed.

Here, the commands are more specific. They represent a thing that can be done at a specific point in time. This means that the input handling code will be *creating* an instance of this every time the player chooses a move. Something like:

```
Command* handleInput()
{
  Unit* unit = getSelectedUnit();

  if (isPressed(BUTTON_UP)) {
    // Move the unit up one.
    int destY = unit->y() - 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  if (isPressed(BUTTON_DOWN)) {
    // Move the unit down one.
    int destY = unit->y() + 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  // Other moves...

  return NULL;
}
```

> Of course, in a non-garbage-collected language like C++, this means the code executing commands will also be responsible for freeing their memory.

The fact that commands are one-use-only will come to our advantage in a second. To make commands undoable, we define another operation each command class needs to implement:

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

An `undo()` method reverses the game state changed by the corresponding `execute()` method. Here's our previous move command with undo support:

```
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()
  {
    // Remember the unit's position before the move
    // so we can restore it.
    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_;
};
```

Note that we added some more state to the class. When a unit moves, it forgets where it used to be. If we want to be able to undo that move, we have to remember the unit's previous position ourselves, which is what `xBefore_` and `yBefore_` do.

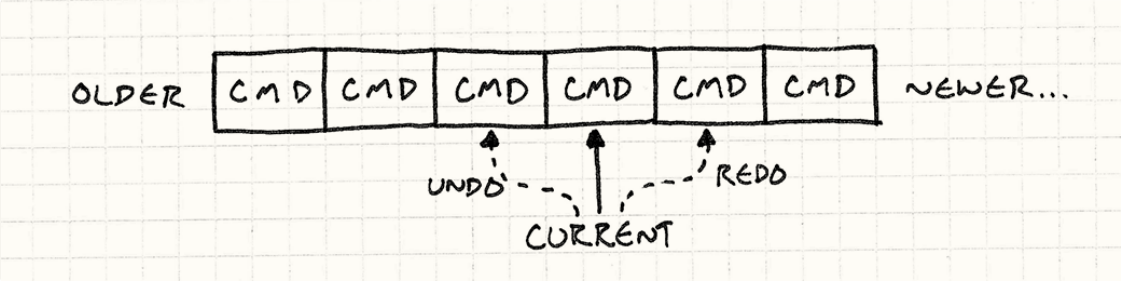
This seems like a place for the Memento [GoF] pattern, but I haven't found it to work well. Since commands tend to modify only a small part of an object's state, snapshotting the rest of its data is a waste of memory. It's cheaper to manually store only the bits you change.

Persistent data structures are another option. With these, every modification to an object returns a new one, leaving the original unchanged. Through clever implementation, these new objects share data with the previous ones, so it's much cheaper than cloning the entire object.

Using a persistent data structure, each command stores a reference to the object before the command was performed, and undo just means switching back to the old object.

To let the player undo a move, we keep around the last command they executed. When they bang on Control-Z, we call that command's `undo()` method. (If they've already undone, then it becomes "redo" and we execute the command again.)

Supporting multiple levels of undo isn't much harder. Instead of remembering the last command, we keep a list of commands and a reference to the "current" one. When the player executes a command, we append it to the list and point "current" at it.



When the player chooses "Undo", we undo the current command and move the current pointer back. When they choose "Redo", we advance the pointer and then execute that command. If they choose a new command after undoing some, everything in the list after the current command is discarded.

The first time I implemented this in a level editor, I felt like a genius. I was astonished at how straightforward it was and how well it worked. It takes discipline to make sure every data modification goes through a command, but once you do that, the rest is easy.

> Redo may not be common in games, but re-*play* is. A naïve implementation would record the entire game state at each frame so it can be replayed, but that would use too much memory.
>
> Instead, many games record the set of commands every entity performed each frame. To replay the game, the engine just runs the normal game simulation, executing the pre-recorded commands.

Classy and Dysfunctional?

Earlier, I said commands are similar to first-class functions or closures, but every example I showed here used class definitions. If you're familiar with functional programming, you're probably wondering where the functions are.

I wrote the examples this way because C++ has pretty limited support for first-class functions. Function pointers are stateless, functors are weird and still require defining a class, and the lambdas in C++11 are tricky to work with because of manual memory management.

That's *not* to say you shouldn't use functions for the Command pattern in other languages.

If you have the luxury of a language with real closures, by all means, use them! In some ways, the Command pattern is a way of emulating closures in languages that don't have them.

> I say *some* ways here because building actual classes or structures for commands is still useful even in languages that have closures. If your command has multiple operations (like undoable commands), mapping that to a single function is awkward.
>
> Defining an actual class with fields also helps readers easily tell what data the command contains. Closures are a wonderfully terse way of automatically wrapping up some state, but they can be so automatic that it's hard to see what state they're actually holding.

For example, if we were building a game in JavaScript, we could create a move unit command just like this:

```
function makeMoveUnitCommand(unit, x, y) {
  // This function here is the command object:
  return function() {
    unit.moveTo(x, y);
  }
}
```

We could add support for undo as well using a pair of closures:

```
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);
    }
  };
}
```

If you're comfortable with a functional style, this way of doing things is natural. If you aren't, I hope this chapter helped you along the way a bit. For me, the usefulness of the Command pattern really shows how effective the functional paradigm is for many problems.

See Also

- You may end up with a lot of different command classes. In order to make it easier to implement those, it's often helpful to define a concrete base class with a bunch of convenient high-level methods that the derived commands can compose to define their behavior. That turns the command's main `execute()` method into a Subclass Sandbox.

- In our examples, we explicitly chose which actor would handle a command. In some cases, especially where your object model is hierarchical, it may not be so cut-and-dried. An object may respond to a command, or it may decide to pawn it off on some subordinate object. If you do that, you've got yourself a Chain of Responsibility GoF.

- Some commands are stateless chunks of pure behavior like the `JumpCommand` in the first example. In cases like that, having more than one instance of that class wastes memory since all instances are equivalent. The Flyweight GoF pattern addresses that.

> You could make it a Singleton too, but friends don't let friends create singletons.

← Previous Chapter ≡ The Book Next Chapter →

Flyweight

The fog lifts, revealing a majestic old growth forest. Ancient hemlocks, countless in number, tower over you forming a cathedral of greenery. The stained glass canopy of leaves fragments the sunlight into golden shafts of mist. Between giant trunks, you can make out the massive forest receding into the distance.

This is the kind of otherworldly setting we dream of as game developers, and scenes like these are often enabled by a pattern whose name couldn't possibly be more modest: the humble Flyweight.

Forest for the Trees

I can describe a sprawling woodland with just a few sentences, but actually *implementing* it in a realtime game is another story. When you've got an entire forest of individual trees filling the screen, all that a graphics programmer sees is the millions of polygons they'll have to somehow shovel onto the GPU every sixtieth of a second.

We're talking thousands of trees, each with detailed geometry containing thousands of polygons. Even if you have enough *memory* to describe that forest, in order to render it, that data has to make its way over the bus from the CPU to the GPU.

Each tree has a bunch of bits associated with it:

- A mesh of polygons that define the shape of the trunk, branches, and greenery.
- Textures for the bark and leaves.
- Its location and orientation in the forest.
- Tuning parameters like size and tint so that each tree looks different.

If you were to sketch it out in code, you'd have something like this:

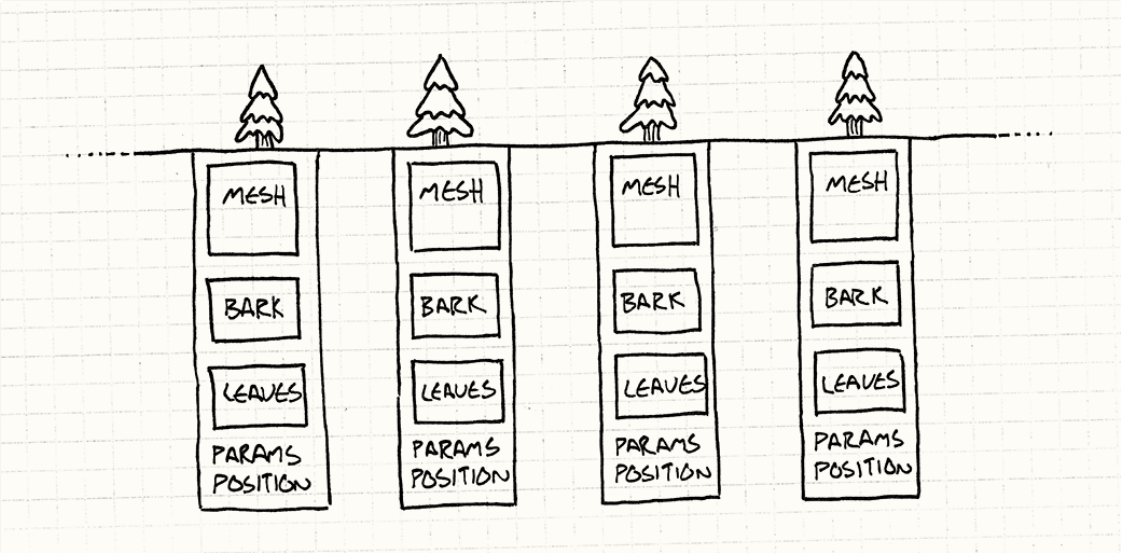
```
class Tree
{
private:
  Mesh mesh_;
```

```
  Texture bark_;
  Texture leaves_;
  Vector position_;
  double height_;
  double thickness_;
  Color barkTint_;
  Color leafTint_;
};
```

That's a lot of data, and the mesh and textures are particularly large. An entire forest of these objects is too much to throw at the GPU in one frame. Fortunately, there's a time-honored trick to handling this.

The key observation is that even though there may be thousands of trees in the forest, they mostly look similar. They will likely all use the same mesh and textures. That means most of the fields in these objects are the *same* between all of those instances.

> You'd have to be crazy or a billionaire to budget for the artists to individually model each tree in an entire forest.



> Note that the stuff in the small boxes is the same for each tree.

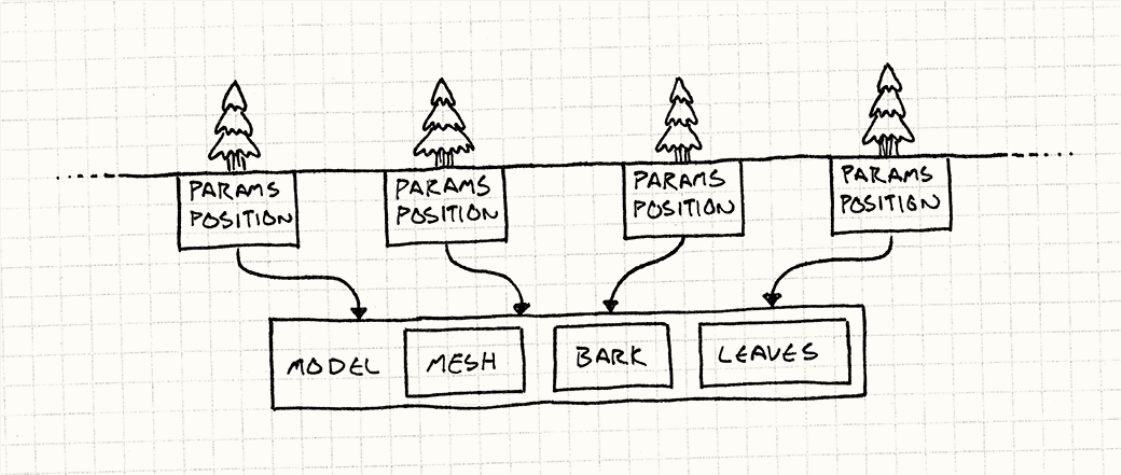
We can model that explicitly by splitting the object in half. First, we pull out the data that all trees have in common and move it into a separate class:

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

The game only needs a single one of these, since there's no reason to have the same meshes and textures in memory a thousand times. Then, each *instance* of a tree in the world has a *reference* to that shared `TreeModel`. What remains in `Tree` is the state that is instance-specific:

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

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

You can visualize it like this:



> This looks a lot like the Type Object pattern. Both involve delegating part of an object's state to some other object shared between a number of instances. However, the intent behind the patterns differs.
>
> With a type object, the goal is to minimize the number of classes you have to define by lifting "types" into your own object model. Any memory sharing you get from that is a bonus. The Flyweight pattern is purely about efficiency.

This is all well and good for storing stuff in main memory, but that doesn't help rendering. Before the forest gets on screen, it has to work its way over to the GPU. We need to express this resource sharing in a way that the graphics card understands.

A Thousand Instances

To minimize the amount of data we have to push to the GPU, we want to be able to send the shared data—the `TreeModel`—just *once*. Then, separately, we push over every tree instance's unique data—its position, color, and scale. Finally, we tell the GPU, "Use that one model to render each of these instances."

Fortunately, today's graphics APIs and cards support exactly that. The details are fiddly and out of the scope of this book, but both Direct3D and OpenGL can do something called *instanced rendering*.

In both APIs, you provide two streams of data. The first is the blob of common data that will be rendered multiple times—the mesh and textures in our arboreal example. The second is the list of instances and their parameters that will be used to vary that first chunk of data each time it's drawn. With a single draw call, an entire forest grows.

> The fact that this API is implemented directly by the graphics card means the Flyweight pattern may be the only Gang of Four design pattern to have actual hardware support.

The Flyweight Pattern

Now that we've got one concrete example under our belts, I can walk you through the general pattern. Flyweight, like its name implies, comes into play when you have objects that need to be more lightweight, generally because you have too many of them.

With instanced rendering, it's not so much that they take up too much memory as it is they take too much *time* to push each separate tree over the bus to the GPU, but the basic idea is the same.

The pattern solves that by separating out an object's data into two kinds. The first kind of data is the stuff that's not specific to a single *instance* of that object and can be shared across all of them. The Gang of Four calls this the *intrinsic* state, but I like to think of it as the "context-free" stuff. In the example here, this is the geometry and textures for the tree.

The rest of the data is the *extrinsic* state, the stuff that is unique to that instance. In this case, that is each tree's position, scale, and color. Just like in the chunk of sample code up there, this pattern saves memory by sharing one copy of the intrinsic state across every place where an object appears.

From what we've seen so far, this seems like basic resource sharing, hardly worth being called a pattern. That's partially because in this example here, we could come up with a clear separate *identity* for the shared state: the `TreeModel`.

I find this pattern to be less obvious (and thus more clever) when used in cases where there isn't a really well-defined identity for the shared object. In those cases, it feels more like an object is magically in multiple places at the same time. Let me show you another

example.

A Place To Put Down Roots

The ground these trees are growing on needs to be represented in our game too. There can be patches of grass, dirt, hills, lakes, rivers, and whatever other terrain you can dream up. We'll make the ground *tile-based*: the surface of the world is a huge grid of tiny tiles. Each tile is covered in one kind of terrain.

Each terrain type has a number of properties that affect gameplay:

- A movement cost that determines how quickly players can move through it.
- A flag for whether it's a watery terrain that can be crossed by boats.
- A texture used to render it.

Because we game programmers are paranoid about efficiency, there's no way we'd store all of that state in each tile in the world. Instead, a common approach is to use an enum for terrain types:

> After all, we already learned our lesson with those trees.

```
enum Terrain
{
  TERRAIN_GRASS,
  TERRAIN_HILL,
  TERRAIN_RIVER
  // Other terrains...
};
```

Then the world maintains a huge grid of those:

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

> Here I'm using a nested array to store the 2D grid. That's efficient in C/C++ because it will pack all of the elements together. In Java or other memory- managed languages, doing that will actually give you an array of rows where each element is a *reference* to the array of columns, which may not be as memory- friendly as you'd like.
>
> In either case, real code would be better served by hiding this implementation detail behind a nice 2D grid data structure. I'm doing this here just to keep it simple.

To actually get the useful data about a tile, we do something like:

```
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;
      // Other terrains...
  }
}

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;
      // Other terrains...
  }
}
```

You get the idea. This works, but I find it ugly. I think of movement cost and wetness as *data* about a terrain, but here that's embedded in code. Worse, the data for a single terrain type is smeared across a bunch of methods. It would be really nice to keep all of that encapsulated together. After all, that's what objects are designed for.

It would be great if we could have an actual terrain *class*, like:

```
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_;
};
```

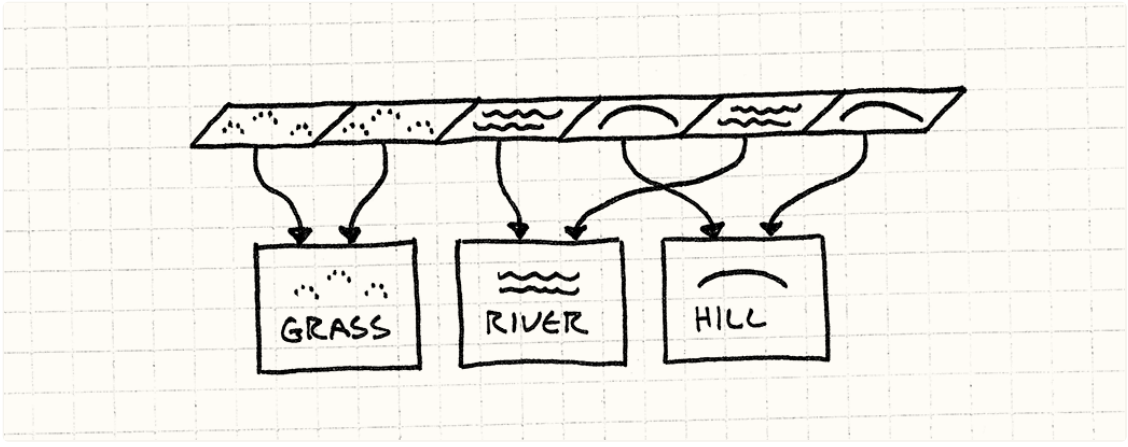
> You'll notice that all of the methods here are `const`. That's no coincidence. Since the same object is used in multiple contexts, if you were to modify it, the changes would appear in multiple places simultaneously.
>
> That's probably not what you want. Sharing objects to save memory should be an optimization that doesn't affect the visible behavior of the app. Because of this, Flyweight objects are almost always immutable.

But we don't want to pay the cost of having an instance of that for each tile in the world. If you look at that class, you'll notice that there's actually *nothing* in there that's specific to *where* that tile is. In flyweight terms, *all* of a terrain's state is "intrinsic" or "context-free".

Given that, there's no reason to have more than one of each terrain type. Every grass tile on the ground is identical to every other one. Instead of having the world be a grid of enums or Terrain objects, it will be a grid of *pointers* to `Terrain` objects:

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

  // Other stuff...
};
```

Each tile that uses the same terrain will point to the same terrain instance.



Since the terrain instances are used in multiple places, their lifetimes would be a little more complex to manage if you were to dynamically allocate them. Instead, we'll just store them directly in the world:

```
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_;

  // Other stuff...
};
```

Then we can use those to paint the ground like this:

```
void World::generateTerrain()
{
  // Fill the ground with grass.
  for (int x = 0; x < WIDTH; x++)
  {
    for (int y = 0; y < HEIGHT; y++)
    {
      // Sprinkle some hills.
      if (random(10) == 0)
      {
        tiles_[x][y] = &hillTerrain_;
      }
      else
      {
        tiles_[x][y] = &grassTerrain_;
      }
    }
  }

  // Lay a river.
  int x = random(WIDTH);
  for (int y = 0; y < HEIGHT; y++) {
    tiles_[x][y] = &riverTerrain_;
  }
}
```

I'll admit this isn't the world's greatest procedural terrain generation algorithm.

Now instead of methods on `World` for accessing the terrain properties, we can expose the `Terrain` object directly:

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

This way, `World` is no longer coupled to all sorts of details of terrains. If you want some property of the tile, you can get it right from that object:

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

We're back to the pleasant API of working with real objects, and we did this with almost no overhead—a pointer is often no larger than an enum.

What About Performance?

I say "almost" here because the performance bean counters will rightfully want to know how this compares to using an enum. Referencing the terrain by pointer implies an indirect lookup. To get to some terrain data like the movement cost, you first have to follow the pointer in the grid to find the terrain object and then find the movement cost there. Chasing a pointer like this can cause a cache miss, which can slow things down.

> For lots more on pointer chasing and cache misses, see the chapter on Data Locality.

As always, the golden rule of optimization is *profile first*. Modern computer hardware is too complex for performance to be a game of pure reason anymore. In my tests for this chapter, there was no penalty for using a flyweight over an enum. Flyweights were actually noticeably faster. But that's entirely dependent on how other stuff is laid out in memory.

What I *am* confident of is that using flyweight objects shouldn't be dismissed out of hand. They give you the advantages of an object-oriented style without the expense of tons of objects. If you find yourself creating an enum and doing lots of switches on it, consider this pattern instead. If you're worried about performance, at least profile first before changing your code to a less maintainable style.

See Also

- In the tile example, we just eagerly created an instance for each terrain type and stored it in `World`. That made it easy to find and reuse the shared instances. In many cases, though, you won't want to create *all* of the flyweights up front.

  If you can't predict which ones you actually need, it's better to create them on demand. To get the advantage of sharing, when you request one, you first see if you've already created an identical one. If so, you just return that instance.

This usually means that you have to encapsulate construction behind some interface that can first look for an existing object. Hiding a constructor like this is an example of the Factory Method pattern.

- In order to return a previously created flyweight, you'll have to keep track of the pool of ones that you've already instantiated. As the name implies, that means that an Object Pool might be a helpful place to store them.

- When you're using the State pattern, you often have "state" objects that don't have any fields specific to the machine that the state is being used in. The state's identity and methods are enough to be useful. In that case, you can apply this pattern and reuse that same state instance in multiple state machines at the same time without any problems.

← Previous Chapter ≡ The Book Next Chapter →

Observer

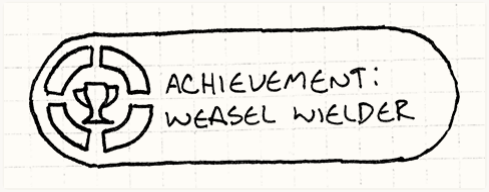
You can't throw a rock at a computer without hitting an application built using the Model-View-Controller architecture, and underlying that is the Observer pattern. Observer is so pervasive that Java put it in its core library (`java.util.Observer`) and C# baked it right into the *language* (the `event` keyword).

> Like so many things in software, MVC was invented by Smalltalkers in the seventies. Lispers probably claim they came up with it in the sixties but didn't bother writing it down.

Observer is one of the most widely used and widely known of the original Gang of Four patterns, but the game development world can be strangely cloistered at times, so maybe this is all news to you. In case you haven't left the abbey in a while, let me walk you through a motivating example.

Achievement Unlocked

Say we're adding an achievements system to our game. It will feature dozens of different badges players can earn for completing specific milestones like "Kill 100 Monkey Demons", "Fall off a Bridge", or "Complete a Level Wielding Only a Dead Weasel".



I swear I had no double meaning in mind when I drew this.

This is tricky to implement cleanly since we have such a wide range of achievements that are unlocked by all sorts of different behaviors. If we aren't careful, tendrils of our achievement system will twine their way through every dark corner of our codebase. Sure, "Fall off a Bridge" is somehow tied to the physics engine, but do we really want to see a call to `unlockFallOffBridge()` right in the middle of the linear algebra in our collision

resolution algorithm?

> This is a rhetorical question. No self-respecting physics programmer would ever let us sully their beautiful mathematics with something as pedestrian as *gameplay*.

What we'd like, as always, is to have all the code concerned with one facet of the game nicely lumped in one place. The challenge is that achievements are triggered by a bunch of different aspects of gameplay. How can that work without coupling the achievement code to all of them?

That's what the observer pattern is for. It lets one piece of code announce that something interesting happened *without actually caring who receives the notification*.

For example, we've got some physics code that handles gravity and tracks which bodies are relaxing on nice flat surfaces and which are plummeting toward sure demise. To implement the "Fall off a Bridge" badge, we could just jam the achievement code right in there, but that's a mess. Instead, we can just do:

```
void Physics::updateEntity(Entity& entity)
{
  bool wasOnSurface = entity.isOnSurface();
  entity.accelerate(GRAVITY);
  entity.update();
  if (wasOnSurface && !entity.isOnSurface())
  {
    notify(entity, EVENT_START_FALL);
  }
}
```

All it does is say, "Uh, I don't know if anyone cares, but this thing just fell. Do with that as you will."

> The physics engine does have to decide what notifications to send, so it isn't entirely decoupled. But in architecture, we're most often trying to make systems *better*, not *perfect*.

The achievement system registers itself so that whenever the physics code sends a notification, the achievement system receives it. It can then check to see if the falling body is our less-than-graceful hero, and if his perch prior to this new, unpleasant encounter with classical mechanics was a bridge. If so, it unlocks the proper achievement with associated fireworks and fanfare, and it does all of this with no involvement from the physics code.

In fact, we can change the set of achievements or tear out the entire achievement system without touching a line of the physics engine. It will still send out its notifications,

oblivious to the fact that nothing is receiving them anymore.

> Of course, if we *permanently* remove achievements and nothing else ever listens to the physics engine's notifications, we may as well remove the notification code too. But during the game's evolution, it's nice to have this flexibility.

How it Works

If you don't already know how to implement the pattern, you could probably guess from the previous description, but to keep things easy on you, I'll walk through it quickly.

The observer

We'll start with the nosy class that wants to know when another object does something interesting. These inquisitive objects are defined by this interface:

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

> The parameters to `onNotify()` are up to you. That's why this is the Observer *pattern* and not the Observer "ready-made code you can paste into your game". Typical parameters are the object that sent the notification and a generic "data" parameter you stuff other details into.
>
> If you're coding in a language with generics or templates, you'll probably use them here, but it's also fine to tailor them to your specific use case. Here, I'm just hardcoding it to take a game entity and an enum that describes what happened.

Any concrete class that implements this becomes an observer. In our example, that's the achievement system, so we'd have something like so:

```
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;

      // Handle other events, and update heroIsOnBridge_...
    }
  }

private:
  void unlock(Achievement achievement)
  {
    // Unlock if not already unlocked...
  }

  bool heroIsOnBridge_;
};
```

The subject

The notification method is invoked by the object being observed. In Gang of Four parlance, that object is called the "subject". It has two jobs. First, it holds the list of observers that are waiting oh-so-patiently for a missive from it:

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

> In real code, you would use a dynamically-sized collection instead of a dumb array. I'm sticking with the basics here for people coming from other languages who don't know C++'s standard library.

The important bit is that the subject exposes a *public* API for modifying that list:

```
class Subject
{
public:
  void addObserver(Observer* observer)
  {
    // Add to array...
  }

  void removeObserver(Observer* observer)
  {
    // Remove from array...
```

```
  }

  // Other stuff...
};
```

That allows outside code to control who receives notifications. The subject communicates with the observers, but it isn't *coupled* to them. In our example, no line of physics code will mention achievements. Yet, it can still talk to the achievements system. That's the clever part about this pattern.

It's also important that the subject has a *list* of observers instead of a single one. It makes sure that observers aren't implicitly coupled to *each other*. For example, say the audio engine also observes the fall event so that it can play an appropriate sound. If the subject only supported one observer, when the audio engine registered itself, that would *un*-register the achievements system.

That means those two systems would interfere with each other—and in a particularly nasty way, since the second would disable the first. Supporting a list of observers ensures that each observer is treated independently from the others. As far as they know, each is the only thing in the world with eyes on the subject.

The other job of the subject is sending notifications:

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

  // Other stuff...
};
```

> Note that this code assumes observers don't modify the list in their `onNotify()` methods. A more robust implementation would either prevent or gracefully handle concurrent modification like that.

Observable physics

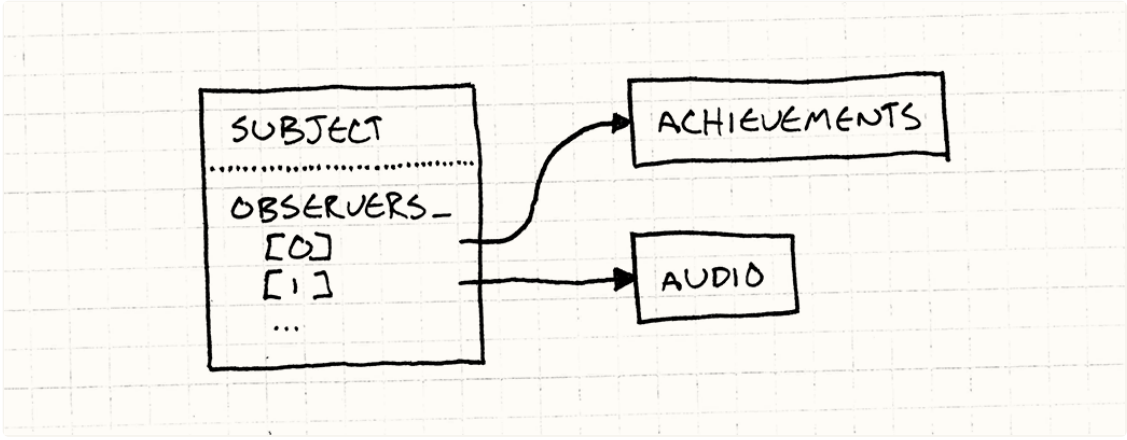
Now, we just need to hook all of this into the physics engine so that it can send notifications and the achievement system can wire itself up to receive them. We'll stay close to the original *Design Patterns* recipe and inherit `Subject`:

```
class Physics : public Subject
{
public:
  void updateEntity(Entity& entity);
};
```

This lets us make `notify()` in `Subject` protected. That way the derived physics engine class can call it to send notifications, but code outside of it cannot. Meanwhile, `addObserver()` and `removeObserver()` are public, so anything that can get to the physics system can observe it.

> In real code, I would avoid using inheritance here. Instead, I'd make `Physics` *have* an instance of `Subject`. Instead of observing the physics engine itself, the subject would be a separate "falling event" object. Observers could register themselves using something like:
>
> ```
> physics.entityFell()
>   .addObserver(this);
> ```
>
> To me, this is the difference between "observer" systems and "event" systems. With the former, you observe *the thing that did something interesting*. With the latter, you observe an object that represents *the interesting thing that happened*.

Now, when the physics engine does something noteworthy, it calls `notify()` like in the motivating example before. That walks the observer list and gives them all the heads up.



Pretty simple, right? Just one class that maintains a list of pointers to instances of some interface. It's hard to believe that something so straightforward is the communication backbone of countless programs and app frameworks.

But the Observer pattern isn't without its detractors. When I've asked other game programmers what they think about this pattern, they bring up a few complaints. Let's see what we can do to address them, if anything.

“It’s Too Slow”

I hear this a lot, often from programmers who don’t actually know the details of the pattern. They have a default assumption that anything that smells like a “design pattern” must involve piles of classes and indirection and other creative ways of squandering CPU cycles.

The Observer pattern gets a particularly bad rap here because it’s been known to hang around with some shady characters named “events”, “messages”, and even “data binding”. Some of those systems *can* be slow (often deliberately, and for good reason). They involve things like queuing or doing dynamic allocation for each notification.

> This is why I think documenting patterns is important. When we get fuzzy about terminology, we lose the ability to communicate clearly and succinctly. You say, “Observer”, and someone hears “Events” or “Messaging” because either no one bothered to write down the difference or they didn’t happen to read it.
>
> That’s what I’m trying to do with this book. To cover my bases, I’ve got a chapter on events and messages too: Event Queue.

But, now that you’ve seen how the pattern is actually implemented, you know that isn’t the case. Sending a notification is simply walking a list and calling some virtual methods. Granted, it’s a *bit* slower than a statically dispatched call, but that cost is negligible in all but the most performance-critical code.

I find this pattern fits best outside of hot code paths anyway, so you can usually afford the dynamic dispatch. Aside from that, there’s virtually no overhead. We aren’t allocating objects for messages. There’s no queueing. It’s just an indirection over a synchronous method call.

It’s too *fast?*

In fact, you have to be careful because the Observer pattern *is* synchronous. The subject invokes its observers directly, which means it doesn’t resume its own work until all of the observers have returned from their notification methods. A slow observer can block a subject.

This sounds scary, but in practice, it’s not the end of the world. It’s just something you have to be aware of. UI programmers—who’ve been doing event-based programming like this for ages—have a time-worn motto for this: “stay off the UI thread”.

If you’re responding to an event synchronously, you need to finish and return control as quickly as possible so that the UI doesn’t lock up. When you have slow work to do, push it onto another thread or a work queue.

You do have to be careful mixing observers with threading and explicit locks, though. If an observer tries to grab a lock that the subject has, you can deadlock the game. In a highly threaded engine, you may be better off with asynchronous communication using an Event Queue.

"It Does Too Much Dynamic Allocation"

Whole tribes of the programmer clan—including many game developers—have moved onto garbage collected languages, and dynamic allocation isn't the boogie man that it used to be. But for performance-critical software like games, memory allocation still matters, even in managed languages. Dynamic allocation takes time, as does reclaiming memory, even if it happens automatically.

> Many game developers are less worried about allocation and more worried about *fragmentation.* When your game needs to run continuously for days without crashing in order to get certified, an increasingly fragmented heap can prevent you from shipping.
>
> The Object Pool chapter goes into more detail about this and a common technique for avoiding it.

In the example code before, I used a fixed array because I'm trying to keep things dead simple. In real implementations, the observer list is almost always a dynamically allocated collection that grows and shrinks as observers are added and removed. That memory churn spooks some people.

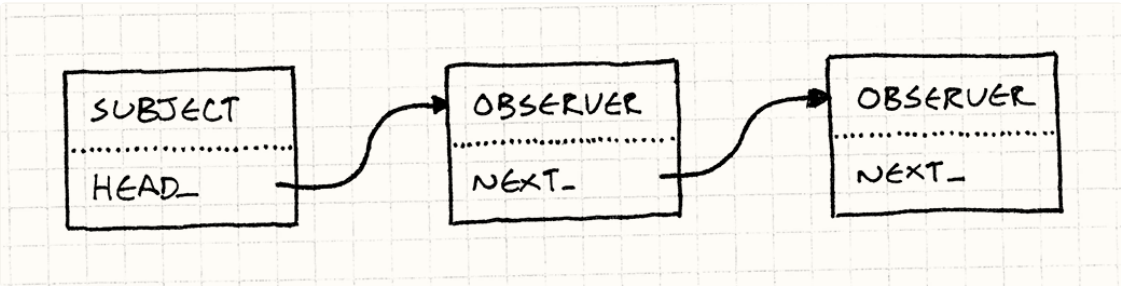
Of course, the first thing to notice is that it only allocates memory when observers are being wired up. *Sending* a notification requires no memory allocation whatsoever—it's just a method call. If you hook up your observers at the start of the game and don't mess with them much, the amount of allocation is minimal.

If it's still a problem, though, I'll walk through a way to implement adding and removing observers without any dynamic allocation at all.

Linked observers

In the code we've seen so far, `Subject` owns a list of pointers to each `Observer` watching it. The `Observer` class itself has no reference to this list. It's just a pure virtual interface. Interfaces are preferred over concrete, stateful classes, so that's generally a good thing.

But if we *are* willing to put a bit of state in `Observer`, we can solve our allocation problem by threading the subject's list *through the observers themselves*. Instead of the subject having a separate collection of pointers, the observer objects become nodes in a linked list:



To implement this, first we'll get rid of the array in `Subject` and replace it with a pointer to the head of the list of observers:

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

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

Then we'll extend `Observer` with a pointer to the next observer in the list:

```
class Observer
{
  friend class Subject;

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

  // Other stuff...
private:
  Observer* next_;
};
```

We're also making `Subject` a friend class here. The subject owns the API for adding and removing observers, but the list it will be managing is now inside the `Observer` class itself. The simplest way to give it the ability to poke at that list is by making it a friend.

Registering a new observer is just wiring it into the list. We'll take the easy option and insert it at the front:

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

```
  head_ = observer;
}
```

The other option is to add it to the end of the linked list. Doing that adds a bit more complexity. `Subject` has to either walk the list to find the end or keep a separate `tail_` pointer that always points to the last node.

Adding it to the front of the list is simpler, but does have one side effect. When we walk the list to send a notification to every observer, the most *recently* registered observer gets notified *first*. So if you register observers A, B, and C, in that order, they will receive notifications in C, B, A order.

In theory, this doesn't matter one way or the other. It's a tenet of good observer discipline that two observers observing the same subject should have no ordering dependencies relative to each other. If the ordering *does* matter, it means those two observers have some subtle coupling that could end up biting you.

Let's get removal working:

```
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_;
  }
}
```

Removing a node from a linked list usually requires a bit of ugly special case handling for removing the very first node, like you see here. There's a more elegant solution using a pointer to a pointer.

I didn't do that here because it confuses at least half the people I show it to. It's a worthwhile exercise for you to do, though: It helps you really

think in terms of pointers.

Because we have a singly linked list, we have to walk it to find the observer we're removing. We'd have to do the same thing if we were using a regular array for that matter. If we use a *doubly* linked list, where each observer has a pointer to both the observer after it and before it, we can remove an observer in constant time. If this were real code, I'd do that.

The only thing left to do is send a notification. That's as simple as walking the list:

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

Here, we walk the entire list and notify every single observer in it. This ensures that all of the observers get equal priority and are independent of each other.

We could tweak this such that when an observer is notified, it can return a flag indicating whether the subject should keep walking the list or stop. If you do that, you're pretty close to having the Chain of Responsibility GoF pattern.

Not too bad, right? A subject can have as many observers as it wants, without a single whiff of dynamic memory. Registering and unregistering is as fast as it was with a simple array. We have sacrificed one small feature, though.

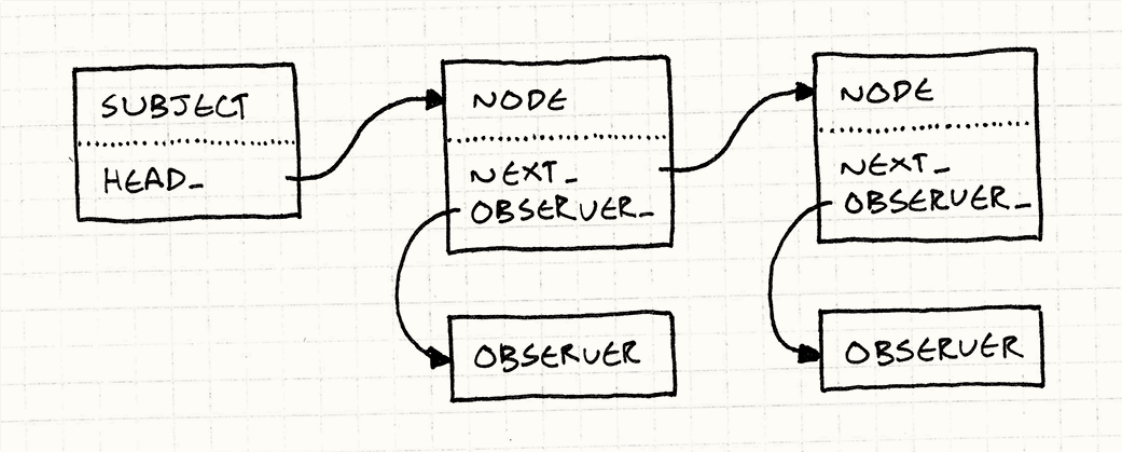
Since we are using the observer object itself as a list node, that implies it can only be part of one subject's observer list. In other words, an observer can only observe a single subject at a time. In a more traditional implementation where each subject has its own independent list, an observer can be in more than one of them simultaneously.

You may be able to live with that limitation. I find it more common for a *subject* to have multiple *observers* than vice versa. If it *is* a problem for you, there is another more complex solution you can use that still doesn't require dynamic allocation. It's too long to cram into this chapter, but I'll sketch it out and let you fill in the blanks...

A pool of list nodes

Like before, each subject will have a linked list of observers. However, those list nodes won't be the observer objects themselves. Instead, they'll be separate little "list node"

objects that contain a pointer to the observer and then a pointer to the next node in the list.



Since multiple nodes can all point to the same observer, that means an observer can be in more than one subject's list at the same time. We're back to being able to observe multiple subjects simultaneously.

> Linked lists come in two flavors. In the one you learned in school, you have a node object that contains the data. In our previous linked observer example, that was flipped around: the *data* (in this case the observer) contained the *node* (i.e. the `next_` pointer).
>
> The latter style is called an "intrusive" linked list because using an object in a list intrudes into the definition of that object itself. That makes intrusive lists less flexible but, as we've seen, also more efficient. They're popular in places like the Linux kernel where that trade-off makes sense.

The way you avoid dynamic allocation is simple: since all of those nodes are the same size and type, you pre-allocate an Object Pool of them. That gives you a fixed-size pile of list nodes to work with, and you can use and reuse them as you need without having to hit an actual memory allocator.

Remaining Problems

I think we've banished the three boogie men used to scare people off this pattern. As we've seen, it's simple, fast, and can be made to play nice with memory management. But does that mean you should use observers all the time?

Now, that's a different question. Like all design patterns, the Observer pattern isn't a cure-all. Even when implemented correctly and efficiently, it may not be the right solution. The reason design patterns get a bad rap is because people apply good patterns to the wrong problem and end up making things worse.

Two challenges remain, one technical and one at something more like the maintainability

level. We'll do the technical one first because those are always easiest.

Destroying subjects and observers

The sample code we walked through is solid, but it side-steps an important issue: what happens when you delete a subject or an observer? If you carelessly call `delete` on some observer, a subject may still have a pointer to it. That's now a dangling pointer into deallocated memory. When that subject tries to send a notification, well... let's just say you're not going to have a good time.

> Not to point fingers, but I'll note that *Design Patterns* doesn't mention this issue at all.

Destroying the subject is easier since in most implementations, the observer doesn't have any references to it. But even then, sending the subject's bits to the memory manager's recycle bin may cause some problems. Those observers may still be expecting to receive notifications in the future, and they don't know that that will never happen now. They aren't observers at all, really, they just think they are.

You can deal with this in a couple of different ways. The simplest is to do what I did and just punt on it. It's an observer's job to unregister itself from any subjects when it gets deleted. More often than not, the observer *does* know which subjects it's observing, so it's usually just a matter of adding a `removeObserver()` call to its destructor.

> As is often the case, the hard part isn't doing it, it's *remembering* to do it.

If you don't want to leave observers hanging when a subject gives up the ghost, that's easy to fix. Just have the subject send one final "dying breath" notification right before it gets destroyed. That way, any observer can receive that and take whatever action it thinks is appropriate.

> Mourn, send flowers, compose elegy, etc.

People—even those of us who've spent enough time in the company of machines to have some of their precise nature rub off on us—are reliably terrible at being reliable. That's why we invented computers: they don't make the mistakes we so often do.

A safer answer is to make observers automatically unregister themselves from every subject when they get destroyed. If you implement the logic for that once in your base observer class, everyone using it doesn't have to remember to do it themselves. This does add some complexity, though. It means each *observer* will need a list of the *subjects* it's observing. You end up with pointers going in both directions.

Don't worry, I've got a GC

All you cool kids with your hip modern languages with garbage collectors are feeling pretty smug right now. Think you don't have to worry about this because you never explicitly delete anything? Think again!

Imagine this: you've got some UI screen that shows a bunch of stats about the player's character like their health and stuff. When the player brings up the screen, you instantiate a new object for it. When they close it, you just forget about the object and let the GC clean it up.

Every time the character takes a punch to the face (or elsewhere, I suppose), it sends a notification. The UI screen observes that and updates the little health bar. Great. Now what happens when the player dismisses the screen, but you don't unregister the observer?

The UI isn't visible anymore, but it won't get garbage collected since the character's observer list still has a reference to it. Every time the screen is loaded, we add a new instance of it to that increasingly long list.

The entire time the player is playing the game, running around, and getting in fights, the character is sending notifications that get received by *all* of those screens. They aren't on screen, but they receive notifications and waste CPU cycles updating invisible UI elements. If they do other things like play sounds, you'll get noticeably wrong behavior.

This is such a common issue in notification systems that it has a name: the *lapsed listener problem*. Since subjects retain references to their listeners, you can end up with zombie UI objects lingering in memory. The lesson here is to be disciplined about unregistration.

> An even surer sign of its significance: it has a Wikipedia article.

What's going on?

The other, deeper issue with the Observer pattern is a direct consequence of its intended purpose. We use it because it helps us loosen the coupling between two pieces of code. It lets a subject indirectly communicate with some observer without being statically bound to it.

This is a real win when you're trying to reason about the subject's behavior, and any hangers-on would be an annoying distraction. If you're poking at the physics engine, you really don't want your editor—or your mind—cluttered up with a bunch of stuff about achievements.

On the other hand, if your program isn't working and the bug spans some chain of observers, reasoning about that communication flow is much more difficult. With an explicit coupling, it's as easy as looking up the method being called. This is child's play for your average IDE since the coupling is static.

But if that coupling happens through an observer list, the only way to tell who will get notified is by seeing which observers happen to be in that list *at runtime*. Instead of being able to *statically* reason about the communication structure of the program, you have to reason about its *imperative, dynamic* behavior.

My guideline for how to cope with this is pretty simple. If you often need to think about *both* sides of some communication in order to understand a part of the program, don't use the Observer pattern to express that linkage. Prefer something more explicit.

When you're hacking on some big program, you tend to have lumps of it that you work on all together. We have lots of terminology for this like "separation of concerns" and "coherence and cohesion" and "modularity", but it boils down to "this stuff goes together and doesn't go with this other stuff".

The observer pattern is a great way to let those mostly unrelated lumps talk to each other without them merging into one big lump. It's less useful *within* a single lump of code dedicated to one feature or aspect.

That's why it fits our example well: achievements and physics are almost entirely unrelated domains, likely implemented by different people. We want the bare minimum of communication between them so that working on either one doesn't require much knowledge of the other.

Observers Today

Design Patterns came out in 1994. Back then, object-oriented programming was *the* hot paradigm. Every programmer on Earth wanted to "Learn OOP in 30 Days," and middle managers paid them based on the number of classes they created. Engineers judged their mettle by the depth of their inheritance hierarchies.

> That same year, Ace of Base had not one but *three* hit singles, so that may tell you something about our taste and discernment back then.

The Observer pattern got popular during that zeitgeist, so it's no surprise that it's class-heavy. But mainstream coders now are more comfortable with functional programming. Having to implement an entire interface just to receive a notification doesn't fit today's aesthetic.

It feels heavyweight and rigid. It *is* heavyweight and rigid. For example, you can't have a single class that uses different notification methods for different subjects.

> This is why the subject usually passes itself to the observer. Since an observer only has a single `onNotify()` method, if it's observing multiple subjects, it needs to be able to tell which one called it.

A more modern approach is for an "observer" to be only a reference to a method or function. In languages with first-class functions, and especially ones with closures, this is a much more common way to do observers.

> These days, practically *every* language has closures. C++ overcame the challenge of closures in a language without garbage collection, and even Java finally got its act together and introduced them in JDK 8.

For example, C# has "events" baked into the language. With those, the observer you register is a "delegate", which is that language's term for a reference to a method. In JavaScript's event system, observers *can* be objects supporting a special `EventListener` protocol, but they can also just be functions. The latter is almost always what people use.

If I were designing an observer system today, I'd make it function-based instead of class-based. Even in C++, I would tend toward a system that let you register member function pointers as observers instead of instances of some `Observer` interface.

> Here's an interesting blog post on one way to implement this in C++.

Observers Tomorrow

Event systems and other observer-like patterns are incredibly common these days. They're a well-worn path. But if you write a few large apps using them, you start to notice something. A lot of the code in your observers ends up looking the same. It's usually something like:

1. Get notified that some state has changed.
2. Imperatively modify some chunk of UI to reflect the new state.

It's all, "Oh, the hero health is 7 now? Let me set the width of the health bar to 70 pixels." After a while, that gets pretty tedious. Computer science academics and software engineers have been trying to eliminate that tedium for a *long* time. Their attempts have gone under a number of different names: "dataflow programming", "functional reactive programming", etc.

While there have been some successes, usually in limited domains like audio processing or chip design, the Holy Grail still hasn't been found. In the meantime, a less ambitious approach has started gaining traction. Many recent application frameworks now use "data binding".

Unlike more radical models, data binding doesn't try to entirely eliminate imperative code and doesn't try to architect your entire application around a giant declarative dataflow graph. What it does do is automate the busywork where you're tweaking a UI element or

calculated property to reflect a change to some value.

Like other declarative systems, data binding is probably a bit too slow and complex to fit inside the core of a game engine. But I would be surprised if I didn't see it start making inroads into less critical areas of the game like UI.

In the meantime, the good old Observer pattern will still be here waiting for us. Sure, it's not as exciting as some hot technique that manages to cram both "functional" and "reactive" in its name, but it's dead simple and it works. To me, those are often the two most important criteria for a solution.

← Previous Chapter ≡ The Book Next Chapter →

Prototype

The first time I heard the word "prototype" was in *Design Patterns*. Today, it seems like everyone is saying it, but it turns out they aren't talking about the design pattern [GoF]. We'll cover that here, but I'll also show you other, more interesting places where the term "prototype" and the concepts behind it have popped up. But first, let's revisit the original pattern.

> I don't say "original" lightly here. *Design Patterns* cites Ivan Sutherland's legendary Sketchpad project in *1963* as one of the first examples of this pattern in the wild. While everyone else was listening to Dylan and the Beatles, Sutherland was busy just, you know, inventing the basic concepts of CAD, interactive graphics, and object-oriented programming.
>
> Watch the demo and prepare to be blown away.

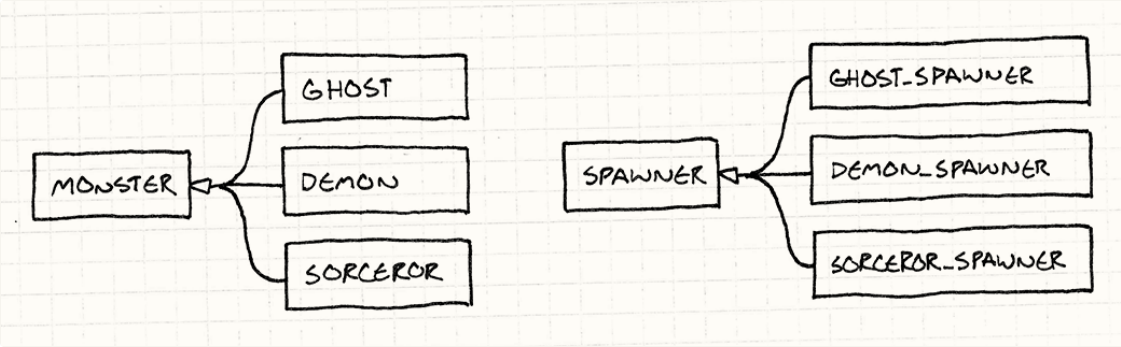
The Prototype Design Pattern

Pretend we're making a game in the style of Gauntlet. We've got creatures and fiends swarming around the hero, vying for their share of his flesh. These unsavory dinner companions enter the arena by way of "spawners", and there is a different spawner for each kind of enemy.

For the sake of this example, let's say we have different classes for each kind of monster in the game—`Ghost`, `Demon`, `Sorcerer`, etc., like:

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

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

A spawner constructs instances of one particular monster type. To support every monster in the game, we *could* brute-force it by having a spawner class for each monster class, leading to a parallel class hierarchy:



I had to dig up a dusty UML book to make this diagram. The ⇦ means "inherits from".

Implementing it would look like this:

```
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();
  }
};

// You get the idea...
```

Unless you get paid by the line of code, this is obviously not a fun way to hack this together. Lots of classes, lots of boilerplate, lots of redundancy, lots of duplication, lots of repeating myself...

The Prototype pattern offers a solution. The key idea is that *an object can spawn other objects similar to itself*. If you have one ghost, you can make more ghosts from it. If you have a demon, you can make other demons. Any monster can be treated as a *prototypal* monster used to generate other versions of itself.

To implement this, we give our base class, `Monster`, an abstract `clone()` method:

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

  // Other stuff...
};
```

Each monster subclass provides an implementation that returns a new object identical in class and state to itself. For example:

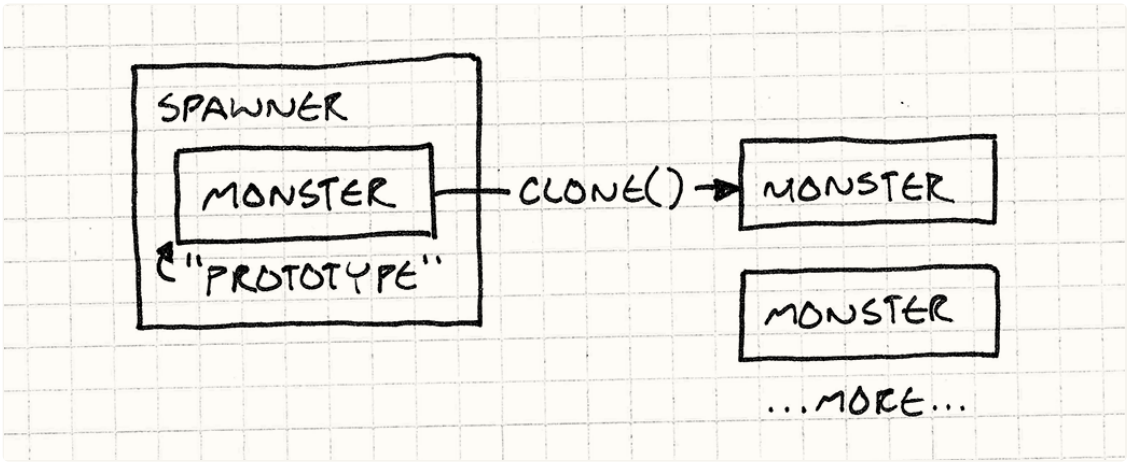
```
class Ghost : public Monster {
public:
  Ghost(int health, int speed)
  : health_(health),
    speed_(speed)
  {}

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

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

Once all our monsters support that, we no longer need a spawner class for each monster class. Instead, we define a single one:

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

  Monster* spawnMonster()
  {
```

```
    return prototype_->clone();
  }

private:
  Monster* prototype_;
};
```

It internally holds a monster, a hidden one whose sole purpose is to be used by the spawner as a template to stamp out more monsters like it, sort of like a queen bee who never leaves the hive.



To create a ghost spawner, we create a prototypal ghost instance and then create a spawner holding that prototype:

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

One neat part about this pattern is that it doesn't just clone the *class* of the prototype, it clones its *state* too. This means we could make a spawner for fast ghosts, weak ghosts, or slow ghosts just by creating an appropriate prototype ghost.

I find something both elegant and yet surprising about this pattern. I can't imagine coming up with it myself, but I can't imagine *not* knowing about it now that I do.

How well does it work?

Well, we don't have to create a separate spawner class for each monster, so that's good. But we *do* have to implement `clone()` in each monster class. That's just about as much code as the spawners.

There are also some nasty semantic ratholes when you sit down to try to write a correct `clone()`. Does it do a deep clone or shallow one? In other words, if a demon is holding a pitchfork, does cloning the demon clone the pitchfork too?

Also, not only does this not look like it's saving us much code in this contrived problem, there's the fact that it's a *contrived problem*. We had to take as a given that we have

separate classes for each monster. These days, that's definitely *not* the way most game engines roll.

Most of us learned the hard way that big class hierarchies like this are a pain to manage, which is why we instead use patterns like Component and Type Object to model different kinds of entities without enshrining each in its own class.

Spawn functions

Even if we do have different classes for each monster, there are other ways to decorticate this *Felis catus*. Instead of making separate spawner *classes* for each monster, we could make spawn *functions*, like so:

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

This is less boilerplate than rolling a whole class for constructing a monster of some type. Then the one spawner class can simply store a function pointer:

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

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

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

private:
  SpawnCallback spawn_;
};
```

To create a spawner for ghosts, you do:

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

Templates

By now, most C++ developers are familiar with templates. Our spawner class needs to construct instances of some type, but we don't want to hard code some specific monster

class. The natural solution then is to make it a *type parameter*, which templates let us do:

I'm not sure if C++ programmers learned to love them or if templates just scared some people completely away from C++. Either way, everyone I see using C++ today uses templates too.

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

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

Using it looks like:

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

The `Spawner` class here is so that code that doesn't care what kind of monster a spawner creates can just use it and work with pointers to `Monster`.

If we only had the `SpawnerFor<T>` class, there would be no single supertype the instantiations of that template all shared, so any code that worked with spawners of any monster type would itself need to take a template parameter.

First-class types

The previous two solutions address the need to have a class, `Spawner`, which is parameterized by a type. In C++, types aren't generally first-class, so that requires some gymnastics. If you're using a dynamically-typed language like JavaScript, Python, or Ruby where classes *are* regular objects you can pass around, you can solve this much more directly.

In some ways, the Type Object pattern is another workaround for the lack of first-class types. That pattern can still be useful even in languages with them, though, because it lets *you* define what a "type" is. You may want different semantics than what the language's built-in classes provide.

When you make a spawner, just pass in the class of monster that it should construct—the actual runtime object that represents the monster's class. Easy as pie.

With all of these options, I honestly can't say I've found a case where I felt the Prototype *design pattern* was the best answer. Maybe your experience will be different, but for now let's put that away and talk about something else: prototypes as a *language paradigm*.

The Prototype Language Paradigm

Many people think "object-oriented programming" is synonymous with "classes". Definitions of OOP tend to feel like credos of opposing religious denominations, but a fairly non-contentious take on it is that *OOP lets you define "objects" which bundle data and code together*. Compared to structured languages like C and functional languages like Scheme, the defining characteristic of OOP is that it tightly binds state and behavior together.

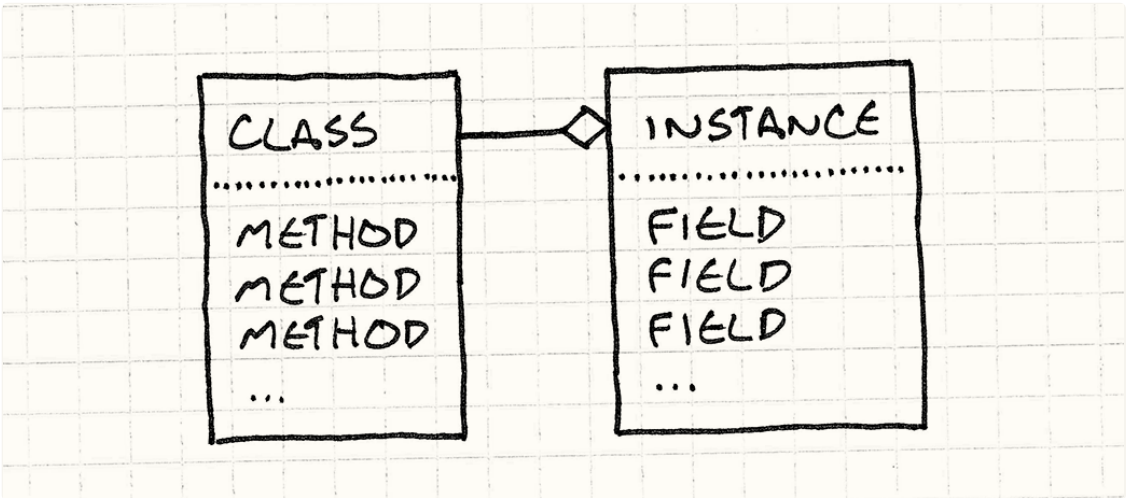
You may think classes are the one and only way to do that, but a handful of guys including Dave Ungar and Randall Smith beg to differ. They created a language in the 80s called Self. While as OOP as can be, it has no classes.

Self

In a pure sense, Self is *more* object-oriented than a class-based language. We think of OOP as marrying state and behavior, but languages with classes actually have a line of separation between them.

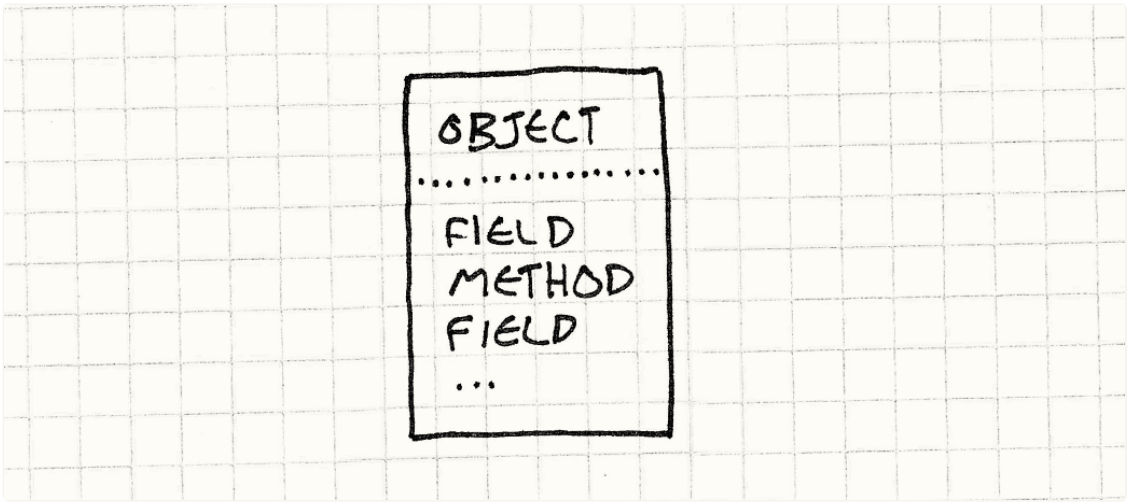
Consider the semantics of your favorite class-based language. To access some state on an object, you look in the memory of the instance itself. State is *contained* in the instance.

To invoke a method, though, you look up the instance's class, and then you look up the method *there*. Behavior is contained in the *class*. There's always that level of indirection to get to a method, which means fields and methods are different.



For example, to invoke a virtual method in C++, you look in the instance for the pointer to its vtable, then look up the method there.

Self eliminates that distinction. To look up *anything*, you just look on the object. An instance can contain both state and behavior. You can have a single object that has a method completely unique to it.

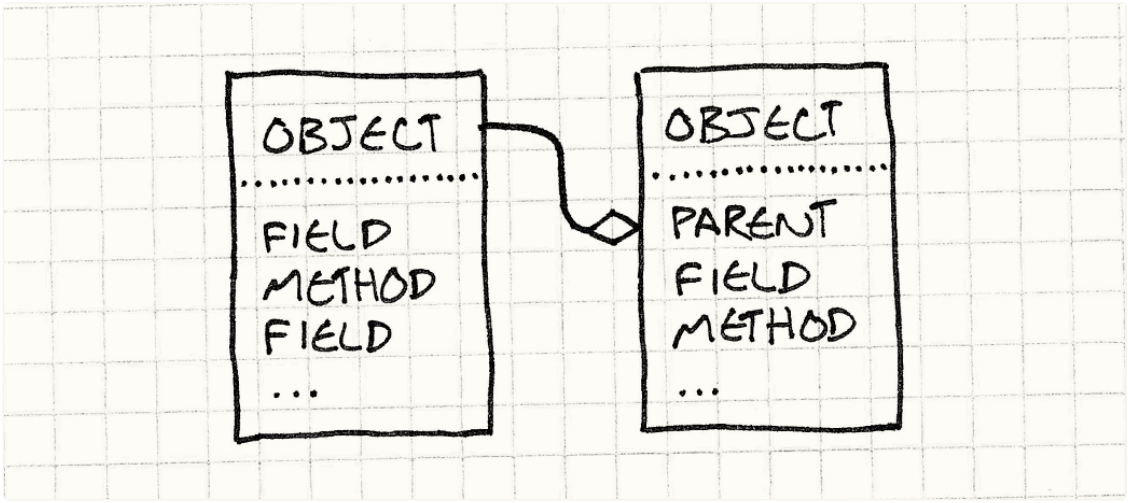


No man is an island, but this object is.

If that was all Self did, it would be hard to use. Inheritance in class-based languages, despite its faults, gives you a useful mechanism for reusing polymorphic code and avoiding duplication. To accomplish something similar without classes, Self has *delegation*.

To find a field or call a method on some object, we first look in the object itself. If it has it, we're done. If it doesn't, we look at the object's *parent*. This is just a reference to some other object. When we fail to find a property on the first object, we try its parent, and its parent, and so on. In other words, failed lookups are *delegated* to an object's parent.

I'm simplifying here. Self actually supports multiple parents. Parents are just specially marked fields, which means you can do things like inherit parents or change them at runtime, leading to what's called *dynamic inheritance*.



Parent objects let us reuse behavior (and state!) across multiple objects, so we've covered part of the utility of classes. The other key thing classes do is give us a way to create instances. When you need a new thingamabob, you can just do `new Thingamabob()`, or whatever your preferred language's syntax is. A class is a factory for instances of itself.

Without classes, how do we make new things? In particular, how do we make a bunch of new things that all have stuff in common? Just like the design pattern, the way you do this in Self is by *cloning*.

In Self, it's as if *every* object supports the Prototype design pattern automatically. Any object can be cloned. To make a bunch of similar objects, you:

1. Beat one object into the shape you want. You can just clone the base `Object` built into the system and then stuff fields and methods into it.
2. Clone it to make as many... uh... clones as you want.

This gives us the elegance of the Prototype design pattern without the tedium of having to implement `clone()` ourselves; it's built into the system.

This is such a beautiful, clever, minimal system that as soon as I learned about it, I started creating a prototype-based language to get more experience with it.

> I realize building a language from scratch is not the most efficient way to learn, but what can I say? I'm a bit peculiar. If you're curious, the language is called Finch.

How did it go?

I was super excited to play with a pure prototype-based language, but once I had mine up and running, I discovered an unpleasant fact: it just wasn't that fun to program in.

> I've since heard through the grapevine that many of the Self programmers came to the same conclusion. The project was far from a

> loss, though. Self was so dynamic that it needed all sorts of virtual machine innovations in order to run fast enough.
>
> The ideas they invented for just-in-time compilation, garbage collection, and optimizing method dispatch are the exact same techniques—often implemented by the same people!—that now make many of the world's dynamically-typed languages fast enough to use for massively popular applications.

Sure, the language was simple to implement, but that was because it punted the complexity onto the user. As soon as I started trying to use it, I found myself missing the structure that classes give. I ended up trying to recapitulate it at the library level since the language didn't have it.

Maybe this is because my prior experience is in class-based languages, so my mind has been tainted by that paradigm. But my hunch is that most people just like well-defined "kinds of things".

In addition to the runaway success of class-based languages, look at how many games have explicit character classes and a precise roster of different sorts of enemies, items, and skills, each neatly labeled. You don't see many games where each monster is a unique snowflake, like "sort of halfway between a troll and a goblin with a bit of snake mixed in".

While prototypes are a really cool paradigm and one that I wish more people knew about, I'm glad that most of us aren't actually programming using them every day. The code I've seen that fully embraces prototypes has a weird mushiness to it that I find hard to wrap my head around.

> It's also telling how *little* code there actually is written in a prototypal style. I've looked.

What about JavaScript?

OK, if prototype-based languages are so unfriendly, how do I explain JavaScript? Here's a language with prototypes used by millions of people every day. More computers run JavaScript than any other language on Earth.

Brendan Eich, the creator of JavaScript, took inspiration directly from Self, and many of JavaScript's semantics are prototype-based. Each object can have an arbitrary set of properties, both fields and "methods" (which are really just functions stored as fields). An object can also have another object, called its "prototype", that it delegates to if a field access fails.

> As a language designer, one appealing thing about prototypes is that they are simpler to implement than classes. Eich took full advantage of this: the first version of JavaScript was created in ten days.

But, despite that, I believe that JavaScript in practice has more in common with class-based languages than with prototypal ones. One hint that JavaScript has taken steps away from Self is that the core operation in a prototype-based language, *cloning*, is nowhere to be seen.

There is no method to clone an object in JavaScript. The closest it has is `Object.create()`, which lets you create a new object that delegates to an existing one. Even that wasn't added until ECMAScript 5, fourteen years after JavaScript came out. Instead of cloning, let me walk you through the typical way you define types and create objects in JavaScript. You start with a *constructor function*:

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

This creates a new object and initializes its fields. You invoke it like:

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

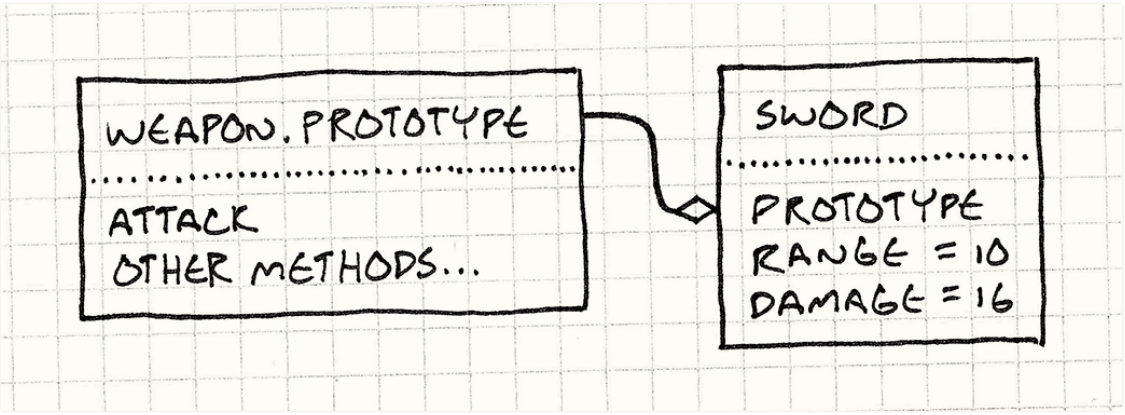
The `new` here invokes the body of the `Weapon()` function with `this` bound to a new empty object. The body adds a bunch of fields to it, then the now-filled-in object is automatically returned.

The `new` also does one other thing for you. When it creates that blank object, it wires it up to delegate to a prototype object. You can get to that object directly using `Weapon.prototype`.

While state is added in the constructor body, to define *behavior*, you usually add methods to the prototype object. Something like this:

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

This adds an `attack` property to the weapon prototype whose value is a function. Since every object returned by `new Weapon()` delegates to `Weapon.prototype`, you can now call `sword.attack()` and it will call that function. It looks a bit like this:



Let's review:

- The way you create objects is by a "new" operation that you invoke using an object that represents the type—the constructor function.
- State is stored on the instance itself.
- Behavior goes through a level of indirection—delegating to the prototype — and is stored on a separate object that represents the set of methods shared by all objects of a certain type.

Call me crazy, but that sounds a lot like my description of classes earlier. You *can* write prototype-style code in JavaScript (*sans* cloning), but the syntax and idioms of the language encourage a class-based approach.

Personally, I think that's a good thing. Like I said, I find doubling down on prototypes makes code harder to work with, so I like that JavaScript wraps the core semantics in something a little more classy.

Prototypes for Data Modeling

OK, I keep talking about things I *don't* like prototypes for, which is making this chapter a real downer. I think of this book as more comedy than tragedy, so let's close this out with an area where I *do* think prototypes, or more specifically *delegation*, can be useful.

If you were to count all the bytes in a game that are code compared to the ones that are data, you'd see the fraction of data has been increasing steadily since the dawn of programming. Early games procedurally generated almost everything so they could fit on floppies and old game cartridges. In many games today, the code is just an "engine" that drives the game, which is defined entirely in data.

That's great, but pushing piles of content into data files doesn't magically solve the organizational challenges of a large project. If anything, it makes it harder. The reason we use programming languages is because they have tools for managing complexity.

Instead of copying and pasting a chunk of code in ten places, we move it into a function

that we can call by name. Instead of copying a method in a bunch of classes, we can put it in a separate class that those classes inherit from or mix in.

When your game's data reaches a certain size, you really start wanting similar features. Data modeling is a deep subject that I can't hope to do justice here, but I do want to throw out one feature for you to consider in your own games: using prototypes and delegation for reusing data.

Let's say we're defining the data model for the shameless Gauntlet rip-off I mentioned earlier. The game designers need to specify the attributes for monsters and items in some kind of files.

> I mean completely original title in no way inspired by any previously existing top-down multi-player dungeon crawl arcade games. Please don't sue me.

One common approach is to use JSON. Data entities are basically *maps*, or *property bags*, or any of a dozen other terms because there's nothing programmers like more than inventing a new name for something that already has one.

> We've re-invented them so many times that Steve Yegge calls them "The Universal Design Pattern".

So a goblin in the game might be defined something like this:

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

This is pretty straightforward and even the most text-averse designer can handle that. So you throw in a couple of sibling branches on the Great Goblin Family Tree:

```
{
  "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"]
}
```

Now, if this was code, our aesthetic sense would be tingling. There's a lot of duplication between these entities, and well-trained programmers *hate* that. It wastes space and takes more time to author. You have to read carefully to tell if the data even *is* the same. It's a maintenance headache. If we decide to make all of the goblins in the game stronger, we need to remember to update the health of all three of them. Bad bad bad.

If this was code, we'd create an abstraction for a "goblin" and reuse that across the three goblin types. But dumb JSON doesn't know anything about that. So let's make it a bit smarter.

We'll declare that if an object has a `"prototype"` field, then that defines the name of another object that this one delegates to. Any properties that don't exist on the first object fall back to being looked up on the prototype.

> This makes the `"prototype"` a piece of *meta*data instead of data. Goblins have warty green skin and yellow teeth. They don't have prototypes. Prototypes are a property of the *data object representing the goblin*, and not the goblin itself.

With that, we can simplify the JSON for our goblin horde:

```
{
  "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"]
}
```

Since the archer and wizard have the grunt as their prototype, we don't have to repeat the health, resists, and weaknesses in each of them. The logic we've added to our data model is super simple—basic single delegation—but we've already gotten rid of a bunch of duplication.

One interesting thing to note here is that we didn't set up a fourth "base goblin" *abstract* prototype for the three concrete goblin types to delegate to. Instead, we just picked one of the goblins who was the simplest and delegated to it.

That feels natural in a prototype-based system where any object can be used as a clone to create new refined objects, and I think it's equally natural here too. It's a particularly good fit for data in games where you often have one-off special entities in the game world.

Think about bosses and unique items. These are often refinements of a more common object in the game, and prototypal delegation is a good fit for defining those. The magic Sword of Head-Detaching, which is really just a longsword with some bonuses, can be expressed as that directly:

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

A little extra power in your game engine's data modeling system can make it easier for designers to add lots of little variations to the armaments and beasties populating your game world, and that richness is exactly what delights players.

← Previous Chapter ≡ The Book Next Chapter →

Singleton

This chapter is an anomaly. Every other chapter in this book shows you how to use a design pattern. This chapter shows you how *not* to use one.

Despite noble intentions, the Singleton [GoF] pattern described by the Gang of Four usually does more harm than good. They stress that the pattern should be used sparingly, but that message was often lost in translation to the game industry.

Like any pattern, using Singleton where it doesn't belong is about as helpful as treating a bullet wound with a splint. Since it's so overused, most of this chapter will be about *avoiding* singletons, but first, let's go over the pattern itself.

> When much of the industry moved to object-oriented programming from C, one problem they ran into was "how do I get an instance?" They had some method they wanted to call but didn't have an instance of the object that provides that method in hand. Singletons (in other words, making it global) were an easy way out.

The Singleton Pattern

Design Patterns summarizes Singleton like this:

> *Ensure a class has one instance, and provide a global point of access to it.*

We'll split that at "and" and consider each half separately.

Restricting a class to one instance

There are times when a class cannot perform correctly if there is more than one instance of it. The common case is when the class interacts with an external system that maintains its own global state.

Consider a class that wraps an underlying file system API. Because file operations can take

a while to complete, our class performs operations asynchronously. This means multiple operations can be running concurrently, so they must be coordinated with each other. If we start one call to create a file and another one to delete that same file, our wrapper needs to be aware of both to make sure they don't interfere with each other.

To do this, a call into our wrapper needs to have access to every previous operation. If users could freely create instances of our class, one instance would have no way of knowing about operations that other instances started. Enter the singleton. It provides a way for a class to ensure at compile time that there is only a single instance of the class.

Providing a global point of access

Several different systems in the game will use our file system wrapper: logging, content loading, game state saving, etc. If those systems can't create their own instances of our file system wrapper, how can they get ahold of one?

Singleton provides a solution to this too. In addition to creating the single instance, it also provides a globally available method to get it. This way, anyone anywhere can get their paws on our blessed instance. All together, the classic implementation looks like this:

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

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

The static `instance_` member holds an instance of the class, and the private constructor ensures that it is the *only* one. The public static `instance()` method grants access to the instance from anywhere in the codebase. It is also responsible for instantiating the singleton instance lazily the first time someone asks for it.

A modern take looks like this:

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

```
    static FileSystem *instance = new FileSystem();
    return *instance;
  }

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

C++11 mandates that the initializer for a local static variable is only run once, even in the presence of concurrency. So, assuming you've got a modern C++ compiler, this code is thread-safe where the first example is not.

> Of course, the thread-safety of your singleton class itself is an entirely different question! This just ensures that its *initialization* is.

Why We Use It

It seems we have a winner. Our file system wrapper is available wherever we need it without the tedium of passing it around everywhere. The class itself cleverly ensures we won't make a mess of things by instantiating a couple of instances. It's got some other nice features too:

- **It doesn't create the instance if no one uses it.** Saving memory and CPU cycles is always good. Since the singleton is initialized only when it's first accessed, it won't be instantiated at all if the game never asks for it.

- **It's initialized at runtime.** A common alternative to Singleton is a class with static member variables. I like simple solutions, so I use static classes instead of singletons when possible, but there's one limitation static members have: automatic initialization. The compiler initializes statics before `main()` is called. This means they can't use information known only once the program is up and running (for example, configuration loaded from a file). It also means they can't reliably depend on each other —the compiler does not guarantee the order in which statics are initialized relative to each other.

  Lazy initialization solves both of those problems. The singleton will be initialized as late as possible, so by that time any information it needs should be available. As long as they don't have circular dependencies, one singleton can even refer to another when initializing itself.

- **You can subclass the singleton.** This is a powerful but often overlooked capability. Let's say we need our file system wrapper to be cross-platform. To make this work, we want it to be an abstract interface for a file system with subclasses that implement the interface for each platform. Here is the base class:

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

Then we define derived classes for a couple of platforms:

```
class PS3FileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path)
  {
    // Use Sony file IO API...
  }

  virtual void writeFile(char* path, char* contents)
  {
    // Use sony file IO API...
  }
};

class WiiFileSystem : public FileSystem
{
public:
  virtual char* readFile(char* path)
  {
    // Use Nintendo file IO API...
  }

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

Next, we turn `FileSystem` into a singleton:

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

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

protected:
```

```
  FileSystem() {}
};
```

The clever part is how the instance is created:

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

  return *instance;
}
```

With a simple compiler switch, we bind our file system wrapper to the appropriate concrete type. Our entire codebase can access the file system using `FileSystem::instance()` without being coupled to any platform-specific code. That coupling is instead encapsulated within the implementation file for the `FileSystem` class itself.

This takes us about as far as most of us go when it comes to solving a problem like this. We've got a file system wrapper. It works reliably. It's available globally so every place that needs it can get to it. It's time to check in the code and celebrate with a tasty beverage.

Why We Regret Using It

In the short term, the Singleton pattern is relatively benign. Like many design choices, we pay the cost in the long term. Once we've cast a few unnecessary singletons into cold hard code, here's the trouble we've bought ourselves:

It's a global variable

When games were still written by a couple of guys in a garage, pushing the hardware was more important than ivory-tower software engineering principles. Old-school C and assembly coders used globals and statics without any trouble and shipped good games. As games got bigger and more complex, architecture and maintainability started to become the bottleneck. We struggled to ship games not because of hardware limitations, but because of *productivity* limitations.

So we moved to languages like C++ and started applying some of the hard-earned wisdom of our software engineer forebears. One lesson we learned is that global variables are bad for a variety of reasons:

- **They make it harder to reason about code.** Say we're tracking down a bug in a

function someone else wrote. If that function doesn't touch any global state, we can wrap our heads around it just by understanding the body of the function and the arguments being passed to it.

> Computer scientists call functions that don't access or modify global state "pure" functions. Pure functions are easier to reason about, easier for the compiler to optimize, and let you do neat things like memoization where you cache and reuse the results from previous calls to the function.
>
> While there are challenges to using purity exclusively, the benefits are enticing enough that computer scientists have created languages like Haskell that *only* allow pure functions.

Now, imagine right in the middle of that function is a call to `SomeClass::getSomeGlobalData()`. To figure out what's going on, we have to hunt through the entire codebase to see what touches that global data. You don't really hate global state until you've had to `grep` a million lines of code at three in the morning trying to find the one errant call that's setting a static variable to the wrong value.

- **They encourage coupling.** The new coder on your team isn't familiar with your game's beautifully maintainable loosely coupled architecture, but he's just been given his first task: make boulders play sounds when they crash onto the ground. You and I know we don't want the physics code to be coupled to *audio* of all things, but he's just trying to get his task done. Unfortunately for us, the instance of our `AudioPlayer` is globally visible. So, one little `#include` later, and our new guy has compromised a carefully constructed architecture.

  Without a global instance of the audio player, even if he *did* `#include` the header, he still wouldn't be able to do anything with it. That difficulty sends a clear message to him that those two modules should not know about each other and that he needs to find another way to solve his problem. *By controlling access to instances, you control coupling.*

- **They aren't concurrency-friendly.** The days of games running on a simple single-core CPU are pretty much over. Code today must at the very least *work* in a multi-threaded way even if it doesn't take full advantage of concurrency. When we make something global, we've created a chunk of memory that every thread can see and poke at, whether or not they know what other threads are doing to it. That path leads to deadlocks, race conditions, and other hell-to-fix thread-synchronization bugs.

Issues like these are enough to scare us away from declaring a global variable, and thus the Singleton pattern too, but that still doesn't tell us how we *should* design the game. How do you architect a game without global state?

There are some extensive answers to that question (most of this book in many ways *is* an answer to just that), but they aren't apparent or easy to come by. In the meantime, we have to get games out the door. The Singleton pattern looks like a panacea. It's in a book on object-oriented design patterns, so it *must* be architecturally sound, right? And it lets us design software the way we have been doing for years.

Unfortunately, it's more placebo than cure. If you scan the list of problems that globals cause, you'll notice that the Singleton pattern doesn't solve any of them. That's because a singleton *is* global state—it's just encapsulated in a class.

It solves two problems even when you just have one

The word "and" in the Gang of Four's description of Singleton is a bit strange. Is this pattern a solution to one problem or two? What if we have only one of those? Ensuring a single instance is useful, but who says we want to let *everyone* poke at it? Likewise, global access is convenient, but that's true even for a class that allows multiple instances.

The latter of those two problems, convenient access, is almost always why we turn to the Singleton pattern. Consider a logging class. Most modules in the game can benefit from being able to log diagnostic information. However, passing an instance of our `Log` class to every single function clutters the method signature and distracts from the intent of the code.

The obvious fix is to make our `Log` class a singleton. Every function can then go straight to the class itself to get an instance. But when we do that, we inadvertently acquire a strange little restriction. All of a sudden, we can no longer create more than one logger.

At first, this isn't a problem. We're writing only a single log file, so we only need one instance anyway. Then, deep in the development cycle, we run into trouble. Everyone on the team has been using the logger for their own diagnostics, and the log file has become a massive dumping ground. Programmers have to wade through pages of text just to find the one entry they care about.

We'd like to fix this by partitioning the logging into multiple files. To do this, we'll have separate loggers for different game domains: online, UI, audio, gameplay. But we can't. Not only does our `Log` class no longer allow us to create multiple instances, that design limitation is entrenched in every single call site that uses it:

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

In order to make our `Log` class support multiple instantiation (like it originally did), we'll have to fix both the class itself and every line of code that mentions it. Our convenient access isn't so convenient anymore.

> It could be even worse than this. Imagine your `Log` class is in a library being shared across several *games*. Now, to change the design, you'll

have to coordinate the change across several groups of people, most of whom have neither the time nor the motivation to fix it.

Lazy initialization takes control away from you

In the desktop PC world of virtual memory and soft performance requirements, lazy initialization is a smart trick. Games are a different animal. Initializing a system can take time: allocating memory, loading resources, etc. If initializing the audio system takes a few hundred milliseconds, we need to control when that's going to happen. If we let it lazy-initialize itself the first time a sound plays, that could be in the middle of an action-packed part of the game, causing visibly dropped frames and stuttering gameplay.

Likewise, games generally need to closely control how memory is laid out in the heap to avoid fragmentation. If our audio system allocates a chunk of heap when it initializes, we want to know *when* that initialization is going to happen, so that we can control *where* in the heap that memory will live.

See Object Pool for a detailed explanation of memory fragmentation.

Because of these two problems, most games I've seen don't rely on lazy initialization. Instead, they implement the Singleton pattern like this:

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

private:
  FileSystem() {}

  static FileSystem instance_;
};
```

That solves the lazy initialization problem, but at the expense of discarding several singleton features that *do* make it better than a raw global variable. With a static instance, we can no longer use polymorphism, and the class must be constructible at static initialization time. Nor can we free the memory that the instance is using when not needed.

Instead of creating a singleton, what we really have here is a simple static class. That isn't necessarily a bad thing, but if a static class is all you need, why not get rid of the `instance()` method entirely and use static functions instead? Calling `Foo::bar()` is simpler than `Foo::instance().bar()`, and also makes it clear that you really are dealing with static memory.

The usual argument for choosing singletons over static classes is that if you decide to change the static class into a non-static one later, you'll need to fix every call site. In theory, you don't have to do that with singletons because you could be passing the instance around and calling it like a normal instance method.

In practice, I've never seen it work that way. Everyone just does `Foo::instance().bar()` in one line. If we changed Foo to not be a singleton, we'd still have to touch every call site. Given that, I'd rather have a simpler class and a simpler syntax to call into it.

What We Can Do Instead

If I've accomplished my goal so far, you'll think twice before you pull Singleton out of your toolbox the next time you have a problem. But you still have a problem that needs solving. What tool *should* you pull out? Depending on what you're trying to do, I have a few options for you to consider, but first...

See if you need the class at all

Many of the singleton classes I see in games are "managers"—those nebulous classes that exist just to babysit other objects. I've seen codebases where it seems like *every* class has a manager: Monster, MonsterManager, Particle, ParticleManager, Sound, SoundManager, ManagerManager. Sometimes, for variety, they'll throw a "System" or "Engine" in there, but it's still the same idea.

While caretaker classes are sometimes useful, often they just reflect unfamiliarity with OOP. Consider these two contrived classes:

```
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);
  }
};
```

Maybe this example is a bit dumb, but I've seen plenty of code that reveals a design just like this after you scrape away the crusty details. If you look at this code, it's natural to think that `BulletManager` should be a singleton. After all, anything that has a `Bullet` will need the manager too, and how many instances of `BulletManager` do you need?

The answer here is *zero*, actually. Here's how we solve the "singleton" problem for our manager class:

```
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_;
};
```

There we go. No manager, no problem. Poorly designed singletons are often "helpers" that add functionality to another class. If you can, just move all of that behavior into the class it helps. After all, OOP is about letting objects take care of themselves.

Outside of managers, though, there are other problems where we'd reach to Singleton for a solution. For each of those problems, there are some alternative solutions to consider.

To limit a class to a single instance

This is one half of what the Singleton pattern gives you. As in our file system example, it can be critical to ensure there's only a single instance of a class. However, that doesn't necessarily mean we also want to provide *public*, *global* access to that instance. We may want to restrict access to certain areas of the code or even make it private to a single class. In those cases, providing a public global point of access weakens the architecture.

> For example, we may be wrapping our file system wrapper inside *another* layer of abstraction.

We want a way to ensure single instantiation *without* providing global access. There are a couple of ways to accomplish this. Here's one:

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

  ~FileSystem() { instantiated_ = false; }

private:
  static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

This class allows anyone to construct it, but it will assert and fail if you try to construct more than one instance. As long as the right code creates the instance first, then we've ensured no other code can either get at that instance or create their own. The class ensures the single instantiation requirement it cares about, but it doesn't dictate how the class should be used.

> An *assertion* function is a way of embedding a contract into your code. When `assert()` is called, it evaluates the expression passed to it. If it evaluates to `true`, then it does nothing and lets the game continue. If it evaluates to `false`, it immediately halts the game at that point. In a debug build, it will usually bring up the debugger or at least print out the file and line number where the assertion failed.

> An `assert()` means, "I assert that this should always be true. If it's not, that's a bug and I want to stop *now* so you can fix it." This lets you define contracts between regions of code. If a function asserts that one of its arguments is not `NULL`, that says, "The contract between me and the caller is that I will not be passed `NULL`."
>
> Assertions help us track down bugs as soon as the game does something unexpected, not later when that error finally manifests as something visibly wrong to the user. They are fences in your codebase, corralling bugs so that they can't escape from the code that created them.

The downside with this implementation is that the check to prevent multiple instantiation is only done at *runtime*. The Singleton pattern, in contrast, guarantees a single instance at compile time by the very nature of the class's structure.

To provide convenient access to an instance

Convenient access is the main reason we reach for singletons. They make it easy to get our hands on an object we need to use in a lot of different places. That ease comes at a cost, though—it becomes equally easy to get our hands on the object in places where we *don't* want it being used.

The general rule is that we want variables to be as narrowly scoped as possible while still getting the job done. The smaller the scope an object has, the fewer places we need to keep in our head while we're working with it. Before we take the shotgun approach of a singleton object with *global* scope, let's consider other ways our codebase can get access to an object:

- **Pass it in.** The simplest solution, and often the best, is to simply pass the object you need as an argument to the functions that need it. It's worth considering before we discard it as too cumbersome.

  > Some use the term "dependency injection" to refer to this. Instead of code reaching *out* and finding its dependencies by calling into something global, the dependencies are pushed *in* to the code that needs it through parameters. Others reserve "dependency injection" for more complex ways of providing dependencies to code.

  Consider a function for rendering objects. In order to render, it needs access to an object that represents the graphics device and maintains the render state. It's very common to simply pass that in to all of the rendering functions, usually as a parameter named something like `context`.

  On the other hand, some objects don't belong in the signature of a method. For example, a function that handles AI may need to also write to a log file, but logging isn't its core concern. It would be strange to see `Log` show up in its argument list, so

for cases like that we'll want to consider other options.

> The term for things like logging that appear scattered throughout a codebase is "cross-cutting concern". Handling cross-cutting concerns gracefully is a continuing architectural challenge, especially in statically typed languages.
>
> Aspect-oriented programming was designed to address these concerns.

- **Get it from the base class.** Many game architectures have shallow but wide inheritance hierarchies, often only one level deep. For example, you may have a base `GameObject` class with derived classes for each enemy or object in the game. With architectures like this, a large portion of the game code will live in these "leaf" derived classes. This means that all these classes already have access to the same thing: their `GameObject` base class. We can use that to our advantage:

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

private:
  static Log& log_;
};

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

This ensures nothing outside of `GameObject` has access to its `Log` object, but every derived entity does using `getLog()`. This pattern of letting derived objects implement themselves in terms of protected methods provided to them is covered in the Subclass Sandbox chapter.

> This raises the question, "how does `GameObject` get the `Log` instance?" A simple solution is to have the base class simply create and own a static instance.
>
> If you don't want the base class to take such an active role, you can provide an initialization function to pass it in or use the Service Locator pattern to find it.

- **Get it from something already global.** The goal of removing *all* global state is admirable, but rarely practical. Most codebases will still have a couple of globally available objects, such as a single `Game` or `World` object representing the entire game state.

  We can reduce the number of global classes by piggybacking on existing ones like that. Instead of making singletons out of `Log`, `FileSystem`, and `AudioPlayer`, do this:

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

  // Functions to set log_, et. al. ...

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

private:
  static Game instance_;

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

  With this, only `Game` is globally available. Functions can get to the other systems through it:

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

  > Purists will claim this violates the Law of Demeter. I claim that's still better than a giant pile of singletons.

  If, later, the architecture is changed to support multiple `Game` instances (perhaps for streaming or testing purposes), `Log`, `FileSystem`, and `AudioPlayer` are all unaffected—they won't even know the difference. The downside with this, of course, is that more code ends up coupled to `Game` itself. If a class just needs to play sound, our example still requires it to know about the world in order to get to the audio player.

  We solve this with a hybrid solution. Code that already knows about `Game` can simply access `AudioPlayer` directly from it. For code that doesn't, we provide access to `AudioPlayer` using one of the other options described here.

- **Get it from a Service Locator.** So far, we're assuming the global class is some regular concrete class like `Game`. Another option is to define a class whose sole reason

for being is to give global access to objects. This common pattern is called a Service Locator and gets its own chapter.

What's Left for Singleton

The question remains, where *should* we use the real Singleton pattern? Honestly, I've never used the full Gang of Four implementation in a game. To ensure single instantiation, I usually simply use a static class. If that doesn't work, I'll use a static flag to check at runtime that only one instance of the class is constructed.

There are a couple of other chapters in this book that can also help here. The Subclass Sandbox pattern gives instances of a class access to some shared state without making it globally available. The Service Locator pattern *does* make an object globally available, but it gives you more flexibility with how that object is configured.

← Previous Chapter ≡ The Book Next Chapter →

State

Confession time: I went a little overboard and packed way too much into this chapter. It's ostensibly about the State [GoF] design pattern, but I can't talk about that and games without going into the more fundamental concept of *finite state machines* (or "FSMs"). But then once I went there, I figured I might as well introduce *hierarchical state machines* and *pushdown automata*.

That's a lot to cover, so to keep things as short as possible, the code samples here leave out a few details that you'll have to fill in on your own. I hope they're still clear enough for you to get the big picture.

Don't feel sad if you've never heard of a state machine. While well known to AI and compiler hackers, they aren't that familiar to other programming circles. I think they should be more widely known, so I'm going to throw them at a different kind of problem here.

> This pairing echoes the early days of artificial intelligence. In the '50s and '60s, much of AI research was focused on language processing. Many of the techniques compilers now use for parsing programming languages were invented for parsing human languages.

We've All Been There

We're working on a little side-scrolling platformer. Our job is to implement the heroine that is the player's avatar in the game world. That means making her respond to user input. Push the B button and she should jump. Simple enough:

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

```
  }
}
```

Spot the bug?

There's nothing to prevent "air jumping"—keep hammering B while she's in the air, and she will float forever. The simple fix is to add an `isJumping_` Boolean field to `Heroine` that tracks when she's jumping, and then do:

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

> There should also be code that sets `isJumping_` back to `false` when the heroine touches the ground. I've omitted that here for brevity's sake.

Next, we want the heroine to duck if the player presses down while she's on the ground and stand back up when the button is released:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    // Jump if not jumping...
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    setGraphics(IMAGE_STAND);
  }
}
```

Spot the bug this time?

With this code, the player could:

1. Press down to duck.
2. Press B to jump from a ducking position.
3. Release down while still in the air.

The heroine will switch to her standing graphic in the middle of the jump. Time for another flag...

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

Next, it would be cool if the heroine did a dive attack if the player presses down in the middle of a jump:

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // Jump...
    }
  }
  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_)
    {
      // Stand...
    }
  }
}
```

Bug hunting time again. Find it?

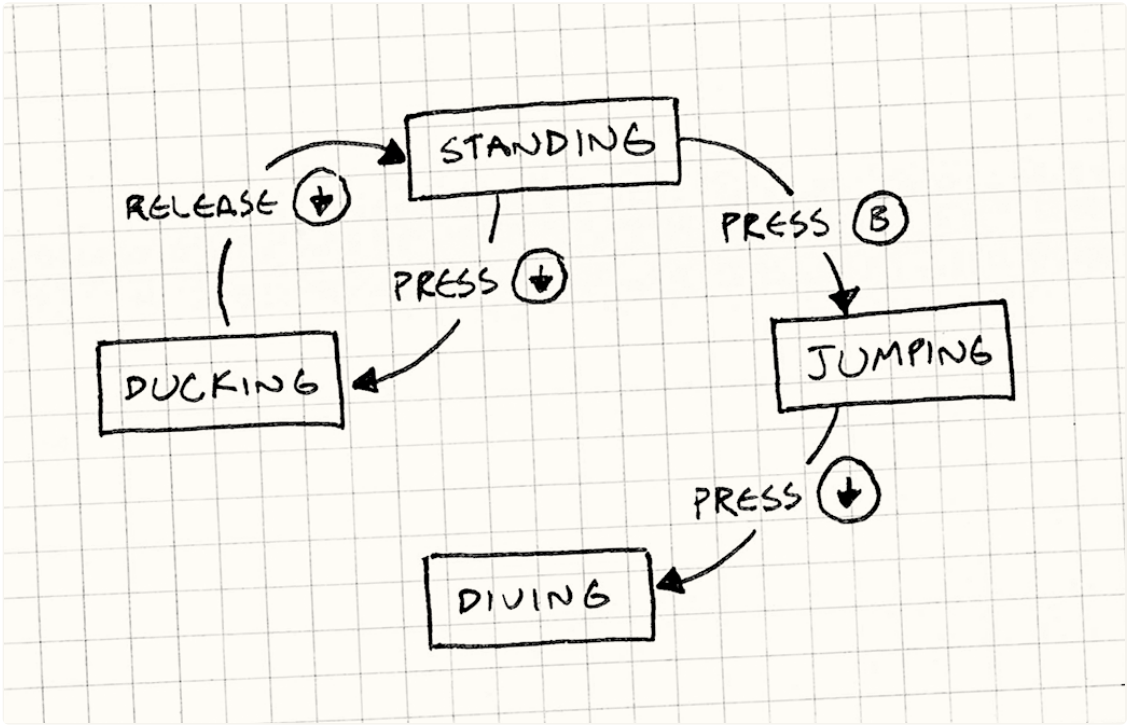
We check that you can't air jump while jumping, but not while diving. Yet another field...

Something is clearly wrong with our approach. Every time we touch this handful of code, we break something. We need to add a bunch more moves—we haven't even added *walking* yet—but at this rate, it will collapse into a heap of bugs before we're done with it.

> Those coders you idolize who always seem to create flawless code aren't simply superhuman programmers. Instead, they have an intuition about which *kinds* of code are error-prone, and they steer away from them.
>
> Complex branching and mutable state—fields that change over time—are two of those error-prone kinds of code, and the examples above have both.

Finite State Machines to the Rescue

In a fit of frustration, you sweep everything off your desk except a pen and paper and start drawing a flowchart. You draw a box for each thing the heroine can be doing: standing, jumping, ducking, and diving. When she can respond to a button press in one of those states, you draw an arrow from that box, label it with that button, and connect it to the state she changes to.



Congratulations, you've just created a *finite state machine*. These came out of a branch of computer science called *automata theory* whose family of data structures also includes the famous Turing machine. FSMs are the simplest member of that family.

The gist is:

- **You have a fixed *set of states* that the machine can be in.** For our example, that's standing, jumping, ducking, and diving.
- **The machine can only be in *one* state at a time.** Our heroine can't be jumping and standing simultaneously. In fact, preventing that is one reason we're going to use an FSM.
- **A sequence of *inputs* or *events* is sent to the machine.** In our example, that's the raw button presses and releases.
- **Each state has a *set of transitions*, each associated with an input and pointing to a state.** When an input comes in, if it matches a transition for the current state, the machine changes to the state that transition points to.

  For example, pressing down while standing transitions to the ducking state. Pressing down while jumping transitions to diving. If no transition is defined for an input on the current state, the input is ignored.

In their pure form, that's the whole banana: states, inputs, and transitions. You can draw it out like a little flowchart. Unfortunately, the compiler doesn't recognize our scribbles, so how do we go about *implementing* one? The Gang of Four's State pattern is one method—which we'll get to—but let's start simpler.

My favorite analogy for FSMs is the old text adventure games like Zork.

You have a world of rooms that are connected to each other by exits. You explore them by entering commands like “go north”.

This maps directly to a state machine: Each room is a state. The room you’re in is the current state. Each room’s exits are its transitions. The navigation commands are the inputs.

Enums and Switches

One problem our `Heroine` class has is some combinations of those Boolean fields aren’t valid: `isJumping_` and `isDucking_` should never both be true, for example. When you have a handful of flags where only one is `true` at a time, that’s a hint that what you really want is an `enum`.

In this case, that `enum` is exactly the set of states for our FSM, so let’s define that:

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

Instead of a bunch of flags, `Heroine` will just have one `state_` field. We also flip the order of our branching. In the previous code, we switched on input, *then* on state. This kept the code for handling one button press together, but it smeared around the code for one state. We want to keep that together, so we switch on state first. That gives us:

```
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;
  }
}
```

This seems trivial, but it's a real improvement over the previous code. We still have some conditional branching, but we simplified the mutable state to a single field. All of the code for handling a single state is now nicely lumped together. This is the simplest way to implement a state machine and is fine for some uses.

> In particular, the heroine can no longer be in an *invalid* state. With the Boolean flags, some sets of values were possible but meaningless. With the `enum`, each value is valid.

Your problem may outgrow this solution, though. Say we want to add a move where our heroine can duck for a while to charge up and unleash a special attack. While she's ducking, we need to track the charge time.

We add a `chargeTime_` field to `Heroine` to store how long the attack has charged. Assume we already have an `update()` that gets called each frame. In there, we add:

```
void Heroine::update()
{
  if (state_ == STATE_DUCKING)
  {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
      superBomb();
    }
  }
}
```

> If you guessed that this is the Update Method pattern, you win a prize!

We need to reset the timer when she starts ducking, so we modify `handleInput()`:

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

    // Other states...
  }
}
```

All in all, to add this charge attack, we had to modify two methods and add a `chargeTime_` field onto `Heroine` even though it's only meaningful while in the ducking state. What we'd prefer is to have all of that code and data nicely wrapped up in one place. The Gang of Four has us covered.

The State Pattern

For people deeply into the object-oriented mindset, every conditional branch is an opportunity to use dynamic dispatch (in other words a virtual method call in C++). I think you can go too far down that rabbit hole. Sometimes an `if` is all you need.

> There's a historical basis for this. Many of the original object-oriented apostles like *Design Patterns*' Gang of Four, and *Refactoring*'s Martin Fowler came from Smalltalk. There, `ifThen:` is just a method you invoke on the condition, which is implemented differently by the `true` and `false` objects.

But in our example, we've reached a tipping point where something object-oriented is a better fit. That gets us to the State pattern. In the words of the Gang of Four:

> *Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.*

That doesn't tell us much. Heck, our `switch` does that. The concrete pattern they describe looks like this when applied to our heroine:

A state interface

First, we define an interface for the state. Every bit of behavior that is state-dependent—every place we had a `switch` before—becomes a virtual method in that interface. For us, that's `handleInput()` and `update()`:

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

Classes for each state

For each state, we define a class that implements the interface. Its methods define the heroine's behavior when in that state. In other words, take each `case` from the earlier `switch` statements and move them into their state's class. For example:

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

  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN)
    {
      // Change to standing state...
      heroine.setGraphics(IMAGE_STAND);
    }
  }

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

private:
  int chargeTime_;
};
```

Note that we also moved `chargeTime_` out of `Heroine` and into the `DuckingState` class. This is great—that piece of data is only meaningful while in that state, and now our

object model reflects that explicitly.

Delegate to the state

Next, we give the `Heroine` a pointer to her current state, lose each big `switch`, and delegate to the state instead:

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

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

  // Other methods...
private:
  HeroineState* state_;
};
```

In order to "change state", we just need to assign `state_` to point to a different `HeroineState` object. That's the State pattern in its entirety.

This looks like the Strategy [GoF] and Type Object [] patterns. In all three, you have a main object that delegates to another subordinate one. The difference is *intent*.

- With Strategy, the goal is to *decouple* the main class from some portion of its behavior.
- With Type Object, the goal is to make a *number* of objects behave similarly by *sharing* a reference to the same type object.
- With State, the goal is for the main object to *change* its behavior by *changing* the object it delegates to.

Where Are the State Objects?

I did gloss over one bit here. To change states, we need to assign `state_` to point to the new one, but where does that object come from? With our `enum` implementation, that was a no-brainer—`enum` values are primitives like numbers. But now our states are classes,

which means we need an actual instance to point to. There are two common answers to this:

Static states

If the state object doesn't have any other fields, then the only data it stores is a pointer to the internal virtual method table so that its methods can be called. In that case, there's no reason to ever have more than one instance of it. Every instance would be identical anyway.

> If your state has no fields and only *one* virtual method in it, you can simplify this pattern even more. Replace each state *class* with a state *function*—just a plain vanilla top-level function. Then, the `state_` field in your main class becomes a simple function pointer.

In that case, you can make a single *static* instance. Even if you have a bunch of FSMs all going at the same time in that same state, they can all point to the same instance since it has nothing machine-specific about it.

> This is the Flyweight [GoF] pattern.

Where you put that static instance is up to you. Find a place that makes sense. For no particular reason, let's put ours inside the base state class:

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

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

Each of those static fields is the one instance of that state that the game uses. To make the heroine jump, the standing state would do something like:

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

Instantiated states

Sometimes, though, this doesn't fly. A static state won't work for the ducking state. It has a `chargeTime_` field, and that's specific to the heroine that happens to be ducking. This may coincidentally work in our game if there's only one heroine, but if we try to add two-player co-op and have two heroines on screen at the same time, we'll have problems.

In that case, we have to create a state object when we transition to it. This lets each FSM have its own instance of the state. Of course, if we're allocating a *new* state, that means we need to free the *current* one. We have to be careful here, since the code that's triggering the change is in a method in the current state. We don't want to delete `this` out from under ourselves.

Instead, we'll allow `handleInput()` in `HeroineState` to optionally return a new state. When it does, `Heroine` will delete the old one and swap in the new one, like so:

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

That way, we don't delete the previous state until we've returned from its method. Now, the standing state can transition to ducking by creating a new instance:

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

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

When I can, I prefer to use static states since they don't burn memory and CPU cycles allocating objects each state change. For states that are more, uh, *stateful*, though, this is the way to go.

> When you dynamically allocate states, you may have to worry about fragmentation. The Object Pool pattern can help.

Enter and Exit Actions

The goal of the State pattern is to encapsulate all of the behavior and data for one state in a single class. We're partway there, but we still have some loose ends.

When the heroine changes state, we also switch her sprite. Right now, that code is owned by the state she's switching *from*. When she goes from ducking to standing, the ducking state sets her image:

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

  // Other code...
}
```

What we really want is each state to control its own graphics. We can handle that by giving the state an *entry action*:

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

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

Back in `Heroine`, we modify the code for handling state changes to call that on the new state:

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

    // Call the enter action on the new state.
    state_->enter(*this);
```

```
  }
}
```

This lets us simplify the ducking code to:

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

  // Other code...
}
```

All it does is switch to standing and the standing state takes care of the graphics. Now our states really are encapsulated. One particularly nice thing about entry actions is that they run when you enter the state regardless of which state you're coming *from*.

Most real-world state graphs have multiple transitions into the same state. For example, our heroine will also end up standing after she lands a jump or dive. That means we would end up duplicating some code everywhere that transition occurs. Entry actions give us a place to consolidate that.

We can, of course, also extend this to support an *exit action*. This is just a method we call on the state we're *leaving* right before we switch to the new state.

What's the Catch?

I've spent all this time selling you on FSMs, and now I'm going to pull the rug out from under you. Everything I've said so far is true, and FSMs are a good fit for some problems. But their greatest virtue is also their greatest flaw.

State machines help you untangle hairy code by enforcing a very constrained structure on it. All you've got is a fixed set of states, a single current state, and some hardcoded transitions.

> A finite state machine isn't even *Turing complete*. Automata theory describes computation using a series of abstract models, each more complex than the previous. A *Turing machine* is one of the most expressive models.
>
> "Turing complete" means a system (usually a programming language) is powerful enough to implement a Turing machine in it, which means all Turing complete languages are, in some ways, equally expressive. FSMs are not flexible enough to be in that club.

If you try using a state machine for something more complex like game AI, you will slam face-first into the limitations of that model. Thankfully, our forebears have found ways to dodge some of those barriers. I'll close this chapter out by walking you through a couple of them.

Concurrent State Machines

We've decided to give our heroine the ability to carry a gun. When she's packing heat, she can still do everything she could before: run, jump, duck, etc. But she also needs to be able to fire her weapon while doing it.

If we want to stick to the confines of an FSM, we have to *double* the number of states we have. For each existing state, we'll need another one for doing the same thing while she's armed: standing, standing with gun, jumping, jumping with gun, you get the idea.

Add a couple of more weapons and the number of states explodes combinatorially. Not only is it a huge number of states, it's a huge amount of redundancy: the unarmed and armed states are almost identical except for the little bit of code to handle firing.

The problem is that we've jammed two pieces of state—what she's *doing* and what she's *carrying*—into a single machine. To model all possible combinations, we would need a state for each *pair*. The fix is obvious: have two separate state machines.

> If we want to cram n states for what she's doing and m states for what she's carrying into a single machine, we need $n \times m$ states. With two machines, it's just $n + m$.

We keep our original state machine for what she's doing and leave it alone. Then we define a separate state machine for what she's carrying. `Heroine` will have *two* "state" references, one for each, like:

```
class Heroine
{
  // Other code...

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

> For illustrative purposes, we're using the full State pattern for her equipment. In practice, since it only has two states, a Boolean flag would work too.

When the heroine delegates inputs to the states, she hands it to both of them:

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

> A more full-featured system would probably have a way for one state machine to *consume* an input so that the other doesn't receive it. That would prevent both machines from erroneously trying to respond to the same input.

Each state machine can then respond to inputs, spawn behavior, and change its state independently of the other machine. When the two sets of states are mostly unrelated, this works well.

In practice, you'll find a few cases where the states do interact. For example, maybe she can't fire while jumping, or maybe she can't do a dive attack if she's armed. To handle that, in the code for one state, you'll probably just do some crude `if` tests on the *other* machine's state to coordinate them. It's not the most elegant solution, but it gets the job done.

Hierarchical State Machines

After fleshing out our heroine's behavior some more, she'll likely have a bunch of similar states. For example, she may have standing, walking, running, and sliding states. In any of those, pressing B jumps and pressing down ducks.

With a simple state machine implementation, we have to duplicate that code in each of those states. It would be better if we could implement that once and reuse it across all of the states.

If this was just object-oriented code instead of a state machine, one way to share code across those states would be using inheritance. We could define a class for an "on ground" state that handles jumping and ducking. Standing, walking, running, and sliding would then inherit from that and add their own additional behavior.

> This has both good and bad implications. Inheritance is a powerful means of code reuse, but it's also a very strong coupling between two chunks of code. It's a big hammer, so swing it carefully.

It turns out, this is a common structure called a *hierarchical state machine*. A state can have a *superstate* (making itself a *substate*). When an event comes in, if the substate

doesn't handle it, it rolls up the chain of superstates. In other words, it works just like overriding inherited methods.

In fact, if we're using the State pattern to implement our FSM, we can use class inheritance to implement the hierarchy. Define a base class for the superstate:

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

And then each substate inherits it:

```
class DuckingState : public OnGroundState
{
public:
  virtual void handleInput(Heroine& heroine, Input input)
  {
    if (input == RELEASE_DOWN)
    {
      // Stand up...
    }
    else
    {
      // Didn't handle input, so walk up hierarchy.
      OnGroundState::handleInput(heroine, input);
    }
  }
};
```

This isn't the only way to implement the hierarchy, of course. If you aren't using the Gang of Four's State pattern, this won't work. Instead, you can model the current state's chain of superstates explicitly using a *stack* of states instead of a single state in the main class.

The current state is the one on the top of the stack, under that is its immediate superstate, and then *that* state's superstate and so on. When you dish out some state-specific behavior, you start at the top of the stack and walk down until one of the states handles it. (If none do, you ignore it.)

Pushdown Automata

There's another common extension to finite state machines that also uses a stack of states. Confusingly, the stack represents something entirely different, and is used to solve a different problem.

The problem is that finite state machines have no concept of *history*. You know what state you *are* in, but have no memory of what state you *were* in. There's no easy way to go back to a previous state.

Here's an example: Earlier, we let our fearless heroine arm herself to the teeth. When she fires her gun, we need a new state that plays the firing animation and spawns the bullet and any visual effects. So we slap together a `FiringState` and make all of the states that she can fire from transition into that when the fire button is pressed.

> Since this behavior is duplicated across several states, it may also be a good place to use a hierarchical state machine to reuse that code.

The tricky part is what state she transitions to *after* firing. She can pop off a round while standing, running, jumping, and ducking. When the firing sequence is complete, she should transition back to what she was doing before.

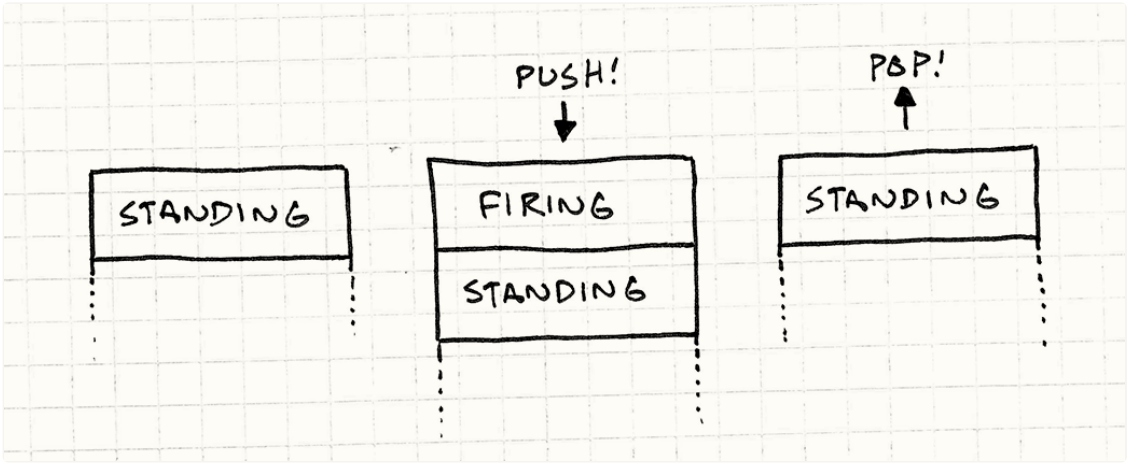
If we're sticking with a vanilla FSM, we've already forgotten what state she was in. To keep track of it, we'd have to define a slew of nearly identical states — firing while standing, firing while running, firing while jumping, and so on — just so that each one can have a hardcoded transition that goes back to the right state when it's done.

What we'd really like is a way to *store* the state she was in before firing and then *recall* it later. Again, automata theory is here to help. The relevant data structure is called a *pushdown automaton*.

Where a finite state machine has a *single* pointer to a state, a pushdown automaton has a *stack* of them. In an FSM, transitioning to a new state *replaces* the previous one. A pushdown automaton lets you do that, but it also gives you two additional operations:

1. You can *push* a new state onto the stack. The "current" state is always the one on top of the stack, so this transitions to the new state. But it leaves the previous state directly under it on the stack instead of discarding it.

2. You can *pop* the topmost state off the stack. That state is discarded, and the state under it becomes the new current state.



This is just what we need for firing. We create a *single* firing state. When the fire button is pressed while in any other state, we *push* the firing state onto the stack. When the firing animation is done, we *pop* that state off, and the pushdown automaton automatically transitions us right back to the state we were in before.

So How Useful Are They?

Even with those common extensions to state machines, they are still pretty limited. The trend these days in game AI is more toward exciting things like *behavior trees* and *planning systems*. If complex AI is what you're interested in, all this chapter has done is whet your appetite. You'll want to read other books to satisfy it.

This doesn't mean finite state machines, pushdown automata, and other simple systems aren't useful. They're a good modeling tool for certain kinds of problems. Finite state machines are useful when:

- You have an entity whose behavior changes based on some internal state.
- That state can be rigidly divided into one of a relatively small number of distinct options.
- The entity responds to a series of inputs or events over time.

In games, they are most known for being used in AI, but they are also common in implementations of user input handling, navigating menu screens, parsing text, network protocols, and other asynchronous behavior.

← Previous Chapter ≡ The Book Next Chapter →

Sequencing Patterns

Game Programming Patterns

Videogames are exciting in large part because they take us somewhere else. For a few minutes (or, let's be honest with ourselves, much longer), we become inhabitants of a virtual world. Creating these worlds is one of the supreme delights of being a game programmer.

One aspect that most of these game worlds feature is *time*—the artificial world lives and breathes at its own cadence. As world builders, we must invent time and craft the gears that drive our game's great clock.

The patterns in this section are tools for doing just that. A Game Loop is the central axle that the clock spins on. Objects hear its ticking through Update Methods. We can hide the computer's sequential nature behind a facade of snapshots of moments in time using Double Buffering so that the world appears to update simultaneously.

The Patterns

- Double Buffer
- Game Loop
- Update Method

Double Buffer

Intent

Cause a series of sequential operations to appear instantaneous or simultaneous.

Motivation

In their hearts, computers are sequential beasts. Their power comes from being able to break down the largest tasks into tiny steps that can be performed one after another. Often, though, our users need to see things occur in a single instantaneous step or see multiple tasks performed simultaneously.

> With threading and multi-core architectures this is becoming less true, but even with several cores, only a few operations are running concurrently.

A typical example, and one that every game engine must address, is rendering. When the game draws the world the users see, it does so one piece at a time—the mountains in the distance, the rolling hills, the trees, each in its turn. If the user *watched* the view draw incrementally like that, the illusion of a coherent world would be shattered. The scene must update smoothly and quickly, displaying a series of complete frames, each appearing instantly.

Double buffering solves this problem, but to understand how, we first need to review how a computer displays graphics.

How computer graphics work (briefly)

A video display like a computer monitor draws one pixel at a time. It sweeps across each row of pixels from left to right and then moves down to the next row. When it reaches the bottom right corner, it scans back up to the top left and starts all over again. It does this so fast — around sixty times a second—that our eyes can't see the scanning. To us, it's a

single static field of colored pixels—an image.

> This explanation is, err, "simplified". If you're a low-level hardware person and you're cringing right now, feel free to skip to the next section. You already know enough to understand the rest of the chapter. If you *aren't* that person, my goal here is to give you just enough context to understand the pattern we'll discuss later.

You can think of this process like a tiny hose that pipes pixels to the display. Individual colors go into the back of the hose, and it sprays them out across the display, one bit of color to each pixel in its turn. So how does the hose know what colors go where?

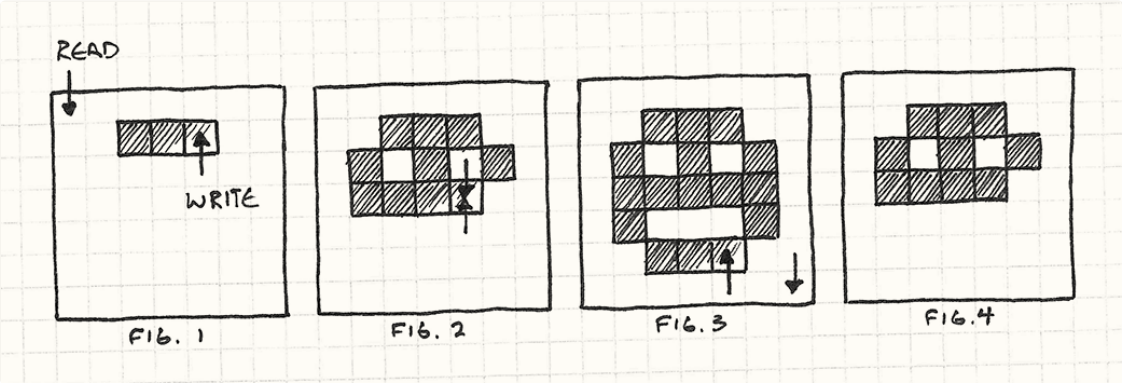
In most computers, the answer is that it pulls them from a *framebuffer*. A framebuffer is an array of pixels in memory, a chunk of RAM where each couple of bytes represents the color of a single pixel. As the hose sprays across the display, it reads in the color values from this array, one byte at a time.

> The specific mapping between byte values and colors is described by the *pixel format* and the *color depth* of the system. In most gaming consoles today, each pixel gets 32 bits: eight each for the red, green, and blue channels, and another eight left over for various other purposes.

Ultimately, in order to get our game to appear on screen, all we do is write to that array. All of the crazy advanced graphics algorithms we have boil down to just that: setting byte values in the framebuffer. But there's a little problem.

Earlier, I said computers are sequential. If the machine is executing a chunk of our rendering code, we don't expect it to be doing anything else at the same time. That's mostly accurate, but a couple of things *do* happen in the middle of our program running. One of those is that the video display will be reading from the framebuffer *constantly* while our game runs. This can cause a problem for us.

Let's say we want a happy face to appear on screen. Our program starts looping through the framebuffer, coloring pixels. What we don't realize is that the video driver is pulling from the framebuffer right as we're writing to it. As it scans across the pixels we've written, our face starts to appear, but then it outpaces us and moves into pixels we haven't written yet. The result is *tearing*, a hideous visual bug where you see half of something drawn on screen.



We start drawing pixels just as the video driver starts reading from the framebuffer (Fig. 1). The video driver eventually catches up to the renderer and then races past it to pixels we haven't written yet (Fig. 2). We finish drawing (Fig. 3), but the driver doesn't catch those new pixels.

The result (Fig. 4) is that the user sees half of the drawing. The name "tearing" comes from the fact that it looks like the bottom half was torn off.

This is why we need this pattern. Our program renders the pixels one at a time, but we need the display driver to see them all at once—in one frame the face isn't there, and in the next one it is. Double buffering solves this. I'll explain how by analogy.

Act 1, Scene 1

Imagine our users are watching a play produced by ourselves. As scene one ends and scene two starts, we need to change the stage setting. If we have the stagehands run on after the scene and start dragging props around, the illusion of a coherent place will be broken. We could dim the lights while we do that (which, of course, is what real theaters do), but the audience still knows *something* is going on. We want there to be no gap in time between scenes.

With a bit of real estate, we come up with this clever solution: we build *two* stages set up so the audience can see both. Each has its own set of lights. We'll call them stage A and stage B. Scene one is shown on stage A. Meanwhile, stage B is dark as the stagehands are setting up scene two. As soon as scene one ends, we cut the lights on stage A and bring them up on stage B. The audience looks to the new stage and scene two begins immediately.

At the same time, our stagehands are over on the now darkened stage *A*, striking scene one and setting up scene *three*. As soon as scene two ends, we switch the lights back to stage A again. We continue this process for the entire play, using the darkened stage as a work area where we can set up the next scene. Every scene transition, we just toggle the lights between the two stages. Our audience gets a continuous performance with no delay between scenes. They never see a stagehand.

> Using a half-silvered mirror and some very smart layout, you could actually build this so that the two stages would appear to the audience in the same *place*. As soon as the lights switch, they would be looking at a different stage, but they would never have to change where they look. Building this is left as an exercise for the reader.

Back to the graphics

That is exactly how double buffering works, and this process underlies the rendering system of just about every game you've ever seen. Instead of a single framebuffer, we have *two*. One of them represents the current frame, stage A in our analogy. It's the one the video hardware is reading from. The GPU can scan through it as much as it wants whenever it wants.

> Not *all* games and consoles do this, though. Older and simpler consoles where memory is limited carefully sync their drawing to the video refresh instead. It's tricky.

Meanwhile, our rendering code is writing to the *other* framebuffer. This is our darkened stage B. When our rendering code is done drawing the scene, it switches the lights by *swapping* the buffers. This tells the video hardware to start reading from the second buffer now instead of the first one. As long as it times that switch at the end of a refresh, we won't get any tearing, and the entire scene will appear all at once.

Meanwhile, the old framebuffer is now available for use. We start rendering the next frame onto it. Voilà!

The Pattern

A **buffered class** encapsulates a **buffer**: a piece of state that can be modified. This buffer is edited incrementally, but we want all outside code to see the edit as a single atomic change. To do this, the class keeps *two* instances of the buffer: a **next buffer** and a **current buffer**.

When information is read *from* a buffer, it is always from the *current* buffer. When information is written *to* a buffer, it occurs on the *next* buffer. When the changes are complete, a **swap** operation swaps the next and current buffers instantly so that the new buffer is now publicly visible. The old current buffer is now available to be reused as the new next buffer.

When to Use It

This pattern is one of those ones where you'll know when you need it. If you have a system that lacks double buffering, it will probably look visibly wrong (tearing, etc.) or will behave

incorrectly. But saying, "you'll know when you need it" doesn't give you much to go on. More specifically, this pattern is appropriate when all of these are true:

- We have some state that is being modified incrementally.
- That same state may be accessed in the middle of modification.
- We want to prevent the code that's accessing the state from seeing the work in progress.
- We want to be able to read the state and we don't want to have to wait while it's being written.

Keep in Mind

Unlike larger architectural patterns, double buffering exists at a lower implementation level. Because of this, it has fewer consequences for the rest of the codebase—most of the game won't even be aware of the difference. There are a couple of caveats, though.

The swap itself takes time

Double-buffering requires a *swap* step once the state is done being modified. That operation must be atomic—no code can access *either* state while they are being swapped. Often, this is as quick as assigning a pointer, but if it takes longer to swap than it does to modify the state to begin with, then we haven't helped ourselves at all.

We have to have two buffers

The other consequence of this pattern is increased memory usage. As its name implies, the pattern requires you to keep *two* copies of your state in memory at all times. On memory-constrained devices, this can be a heavy price to pay. If you can't afford two buffers, you may have to look into other ways to ensure your state isn't being accessed during modification.

Sample Code

Now that we've got the theory, let's see how it works in practice. We'll write a very bare-bones graphics system that lets us draw pixels on a framebuffer. In most consoles and PCs, the video driver provides this low-level part of the graphics system, but implementing it by hand here will let us see what's going on. First up is the buffer itself:

```
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];
};
```

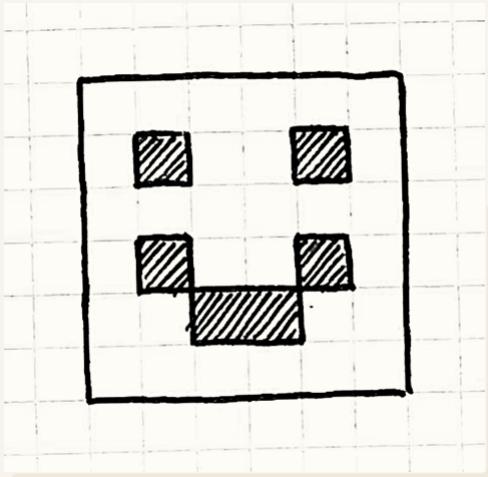
It has basic operations for clearing the entire buffer to a default color and setting the color of an individual pixel. It also has a function, `getPixels()`, to expose the raw array of memory holding the pixel data. We won't see this in the example, but the video driver will call that function frequently to stream memory from the buffer onto the screen.

We wrap this raw buffer in a `Scene` class. It's job here is to render something by making a bunch of `draw()` calls on its buffer:

```
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_;
};
```

Specifically, it draws this artistic masterpiece:



Every frame, the game tells the scene to draw. The scene clears the buffer and then draws a bunch of pixels, one at a time. It also provides access to the internal buffer through `getBuffer()` so that the video driver can get to it.

This seems pretty straightforward, but if we leave it like this, we'll run into problems. The trouble is that the video driver can call `getPixels()` on the buffer at *any* point in time, even here:

```
buffer_.draw(1, 1);
buffer_.draw(4, 1);
// <- Video driver reads pixels here!
buffer_.draw(1, 3);
buffer_.draw(2, 4);
buffer_.draw(3, 4);
buffer_.draw(4, 3);
```

When that happens, the user will see the eyes of the face, but the mouth will disappear for a single frame. In the next frame, it could get interrupted at some other point. The end result is horribly flickering graphics. We'll fix this with double buffering:

```
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_;
};
```

Now `Scene` has two buffers, stored in the `buffers_` array. We don't directly reference them from the array. Instead, there are two members, `next_` and `current_`, that point into the array. When we draw, we draw onto the next buffer, referenced by `next_`. When the video driver needs to get the pixels, it always accesses the *other* buffer through `current_`.

This way, the video driver never sees the buffer that we're working on. The only remaining piece of the puzzle is the call to `swap()` when the scene is done drawing the frame. That swaps the two buffers by simply switching the `next_` and `current_` references. The next time the video driver calls `getBuffer()`, it will get the new buffer we just finished drawing and put our recently drawn buffer on screen. No more tearing or unsightly glitches.

Not just for graphics

The core problem that double buffering solves is state being accessed while it's being modified. There are two common causes of this. We've covered the first one with our graphics example—the state is directly accessed from code on another thread or interrupt.

There is another equally common cause, though: when the code *doing the modification* is accessing the same state that it's modifying. This can manifest in a variety of places, especially physics and AI where you have entities interacting with each other. Double-

buffering is often helpful here too.

Artificial unintelligence

Let's say we're building the behavioral system for, of all things, a game based on slapstick comedy. The game has a stage containing a bunch of actors that run around and get up to various hijinks and shenanigans. Here's our base actor:

```
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_;
};
```

Every frame, the game is responsible for calling `update()` on the actor so that it has a chance to do some processing. Critically, from the user's perspective, *all actors should appear to update simultaneously*.

> This is an example of the Update Method pattern.

Actors can also interact with each other, if by "interacting", we mean "they can slap each other around". When updating, the actor can call `slap()` on another actor to slap it and call `wasSlapped()` to determine if it has been slapped.

The actors need a stage where they can interact, so let's build that:

```
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];
};
```

`Stage` lets us add actors, and provides a single `update()` call that updates each actor. To the user, actors appear to move simultaneously, but internally, they are updated one at a time.

The only other point to note is that each actor's "slapped" state is cleared immediately after updating. This is so that an actor only responds to a given slap once.

To get things going, let's define a concrete actor subclass. Our comedian here is pretty simple. He faces a single actor. Whenever he gets slapped—by anyone—he responds by slapping the actor he faces.

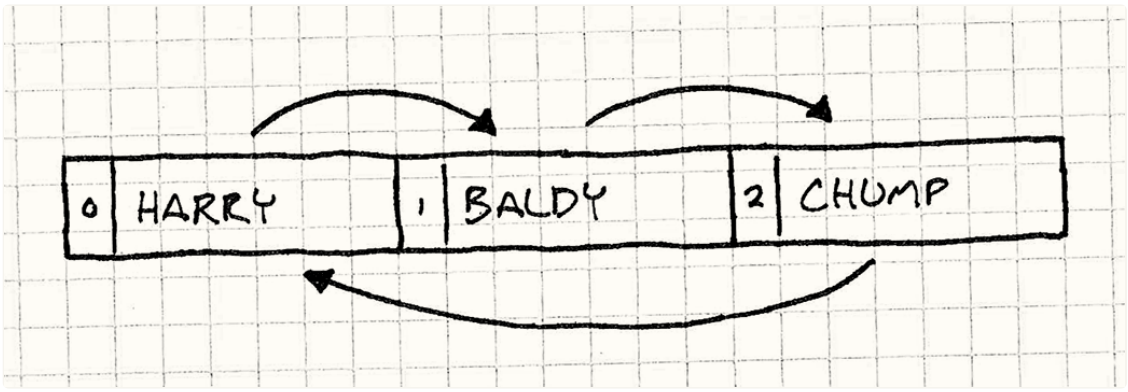
```
class Comedian : public Actor
{
public:
  void face(Actor* actor) { facing_ = actor; }

  virtual void update()
  {
    if (wasSlapped()) facing_->slap();
  }

private:
  Actor* facing_;
};
```

Now, let's throw some comedians on a stage and see what happens. We'll set up three comedians, each facing the next. The last one will face the first, in a big circle:

```
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);
```

The resulting stage is set up as shown in the following image. The arrows show who the actors are facing, and the numbers show their index in the stage's array.



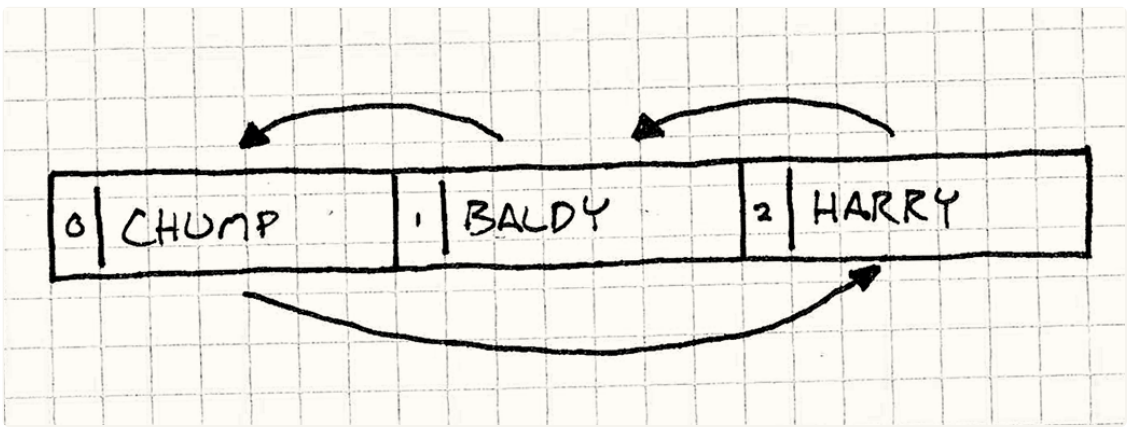
We'll slap Harry to get things going and see what happens when we start processing:

```
harry->slap();

stage.update();
```

Remember that the `update()` function in `Stage` updates each actor in turn, so if we step through the code, we'll find that the following occurs:

```
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
```

In a single frame, our initial slap on Harry has propagated through all of the comedians. Now, to mix things up a bit, let's say we reorder the comedians within the stage's array but leave them facing each other the same way.



We'll leave the rest of the stage setup alone, but we'll replace the chunk of code where we add the actors to the stage with this:

```
stage.add(harry, 2);
stage.add(baldy, 1);
stage.add(chump, 0);
```

Let's see what happens when we run our experiment again:

```
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
```

Uh, oh. Totally different. The problem is straightforward. When we update the actors, we modify their "slapped" states, the exact same state we also *read* during the update. Because of this, changes to that state early in the update affect later parts of that *same* update step.

> If you continue to update the stage, you'll see the slaps gradually cascade through the actors, one per frame. In the first frame, Harry slaps Baldy. In the next frame, Baldy slaps Chump, and so on.

The ultimate result is that an actor may respond to being slapped in either the *same* frame as the slap or in the *next* frame based entirely on how the two actors happen to be ordered on the stage. This violates our requirement that actors need to appear to run in parallel—the order that they update within a single frame shouldn't matter.

Buffered slaps

Fortunately, our Double Buffer pattern can help. This time, instead of having two copies of a monolithic "buffer" object, we'll be buffering at a much finer granularity: each actor's "slapped" state:

```
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_;
};
```

Instead of a single `slapped_` state, each actor now has two. Just like the previous graphics example, the current state is used for reading, and the next state is used for writing.

The `reset()` function has been replaced with `swap()`. Now, right before clearing the swap state, it copies the next state into the current one, making it the new current state. This also requires a small change in `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();
  }
}
```

The `update()` function now updates all of the actors and *then* swaps all of their states. The end result of this is that an actor will only see a slap in the frame *after* it was actually slapped. This way, the actors will behave the same no matter their order in the stage's array. As far as the user or any outside code can tell, all of the actors update simultaneously within a frame.

Design Decisions

Double Buffer is pretty straightforward, and the examples we've seen so far cover most of the variations you're likely to encounter. There are two main decisions that come up when implementing this pattern.

How are the buffers swapped?

The swap operation is the most critical step of the process since we must lock out all reading and modification of both buffers while it's occurring. To get the best performance, we want this to happen as quickly as possible.

- **Swap pointers or references to the buffer:**

  This is how our graphics example works, and it's the most common solution for double-buffering graphics.

  - *It's fast.* Regardless of how big the buffer is, the swap is simply a couple of pointer assignments. It's hard to beat that for speed and simplicity.

  - *Outside code cannot store persistent pointers to the buffer.* This is the main limitation. Since we don't actually move the *data*, what we're essentially doing is periodically telling the rest of the codebase to look somewhere else for the buffer, like in our original stage analogy. This means that the rest of the codebase can't store pointers directly to data within the buffer—they may be pointing at the wrong one a moment later.

    This can be particularly troublesome on a system where the video driver expects the framebuffer to always be at a fixed location in memory. In that case, we won't be able to use this option.

  - *Existing data on the buffer will be from two frames ago, not the last frame.* Successive frames are drawn on alternating buffers with no data copied between them, like so:

    ```
    Frame 1 drawn on buffer A
    Frame 2 drawn on buffer B
    Frame 3 drawn on buffer A
    ...
    ```

    You'll note that when we go to draw the third frame, the data already on the buffer is from frame *one*, not the more recent second frame. In most cases, this isn't an issue—we usually clear the whole buffer right before drawing. But if we intend to reuse some of the existing data on the buffer, it's important to take into account that that data will be a frame older than we might expect.

    > One classic use of old framebuffer data is simulating motion blur. The current frame is blended with a bit of the previously rendered frame to make a resulting image that looks more like what a real camera captures.

- **Copy the data between the buffers:**

If we can't repoint users to the other buffer, the only other option is to actually copy the data from the next frame to the current frame. This is how our slapstick comedians work. In that case, we chose this method because the state—a single Boolean flag—doesn't take any longer to copy than a pointer to the buffer would.

- *Data on the next buffer is only a single frame old.* This is the nice thing about copying the data as opposed to ping-ponging back and forth between the two buffers. If we need access to previous buffer data, this will give us more up-to-date data to work with.

- *Swapping can take more time.* This, of course, is the big negative point. Our swap operation now means copying the entire buffer in memory. If the buffer is large, like an entire framebuffer, it can take a significant chunk of time to do this. Since nothing can read or write to *either* buffer while this is happening, that's a big limitation.

What is the granularity of the buffer?

The other question is how the buffer itself is organized—is it a single monolithic chunk of data or distributed among a collection of objects? Our graphics example uses the former, and the actors use the latter.

Most of the time, the nature of what you're buffering will lead to the answer, but there's some flexibility. For example, our actors all could have stored their messages in a single message block that they all reference into by their index.

- **If the buffer is monolithic:**

  - *Swapping is simpler.* Since there is only one pair of buffers, a single swap does it. If you can swap by changing pointers, then you can swap the entire buffer, regardless of size, with just a couple of assignments.

- **If many objects have a piece of data:**

  - *Swapping is slower.* In order to swap, we need to iterate through the entire collection of objects and tell each one to swap.

    In our comedian example, that was OK since we needed to clear the next slap state anyway—every piece of buffered state needed to be touched each frame. If we don't need to otherwise touch the old buffer, there's a simple optimization we can do to get the same performance of a monolithic buffer while distributing the buffer across multiple objects.

    The idea is to get the "current" and "next" pointer concept and apply it to each of our objects by turning them into object-relative *offsets*. Like so:

    ```
    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];
};
```

Actors access their current slap state by using `current_` to index into the state array. The next state is always the other index in the array, so we can calculate that with `next()`. Swapping the state simply alternates the `current_` index. The clever bit is that `swap()` is now a *static* function—it only needs to be called once, and *every* actor's state will be swapped.

See Also

- You can find the Double Buffer pattern in use in almost every graphics API out there. For example, OpenGL has `swapBuffers()`, Direct3D has "swap chains", and Microsoft's XNA framework swaps the framebuffers within its `endDraw()` method.

Game Loop

Intent

Decouple the progression of game time from user input and processor speed.

Motivation

If there is one pattern this book couldn't live without, this is it. Game loops are the quintessential example of a "game programming pattern". Almost every game has one, no two are exactly alike, and relatively few programs outside of games use them.

To see how they're useful, let's take a quick trip down memory lane. In the olden days of computer programming when everyone had beards, programs worked like your dishwasher. You dumped a load of code in, pushed a button, waited, and got results out. Done. These were *batch mode* programs—once the work was done, the program stopped.

> Ada Lovelace and Rear Admiral Grace Hopper had honorary beards.

You still see these today, though thankfully we don't have to write them on punch cards anymore. Shell scripts, command line programs, and even the little Python script that turns a pile of Markdown into this book are all batch mode programs.

Interview with a CPU

Eventually, programmers realized having to drop off a batch of code at the computing office and come back a few hours later for the results was a terribly slow way to get the bugs out of a program. They wanted immediate feedback. *Interactive* programs were born. Some of the first interactive programs were games:

```
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.
```

This is Colossal Cave Adventure, the first adventure game.

You could have a live conversation with the program. It waited for your input, then it would respond to you. You would reply back, taking turns just like you learned to do in kindergarten. When it was your turn, it sat there doing nothing. Something like:

```
while (true)
{
  char* command = readCommand();
  handleCommand(command);
}
```

This loops forever, so there's no way to quit the game. A real game would do something like `while (!done)` and set `done` to exit. I've omitted that to keep things simple.

Event loops

Modern graphic UI applications are surprisingly similar to old adventure games once you shuck their skin off. Your word processor usually just sits there doing nothing until you press a key or click something:

```
while (true)
{
  Event* event = waitForEvent();
  dispatchEvent(event);
}
```

The main difference is that instead of *text commands*, the program is waiting for *user input events*—mouse clicks and key presses. It still works basically like the old text adventures where the program *blocks* waiting for user input, which is a problem.

Unlike most other software, games keep moving even when the user isn't providing input. If you sit staring at the screen, the game doesn't freeze. Animations keep animating. Visual effects dance and sparkle. If you're unlucky, that monster keeps chomping on your hero.

Most event loops do have "idle" events so you can intermittently do stuff without user input. That's good enough for a blinking cursor or a progress bar, but too rudimentary for games.

This is the first key part of a real game loop: *it processes user input, but doesn't wait for it.* The loop always keeps spinning:

```
while (true)
{
  processInput();
  update();
  render();
}
```

We'll refine this later, but the basic pieces are here. `processInput()` handles any user input that has happened since the last call. Then, `update()` advances the game simulation one step. It runs AI and physics (usually in that order). Finally, `render()` draws the game so the player can see what happened.

> As you might guess from the name, `update()` is a good place to use the Update Method pattern.

A world out of time

If this loop isn't blocking on input, that leads to the obvious question: how *fast* does it spin? Each turn through the game loop advances the state of the game by some amount. From the perspective of an inhabitant of the game world, the hand of their clock has ticked forward.

> The common terms for one crank of the game loop are "tick" and "frame".

Meanwhile, the *player's* actual clock is ticking. If we measure how quickly the game loop cycles in terms of real time, we get the game's "frames per second". If the game loop cycles quickly, the FPS is high and the game moves smoothly and quickly. If it's slow, the game jerks along like a stop motion movie.

With the crude loop we have now where it just cycles as quickly as it can, two factors determine the frame rate. The first is *how much work it has to do each frame*. Complex physics, a bunch of game objects, and lots of graphic detail all will keep your CPU and GPU busy, and it will take longer to complete a frame.

The second is *the speed of the underlying platform*. Faster chips churn through more code in the same amount of time. Multiple cores, GPUs, dedicated audio hardware, and the OS's scheduler all affect how much you get done in one tick.

Seconds per second

In early video games, that second factor was fixed. If you wrote a game for the NES or Apple IIe, you knew *exactly* what CPU your game was running on and you could (and did) code specifically for that. All you had to worry about was how much work you did each tick.

Older games were carefully coded to do just enough work each frame so that the game ran at the speed the developers wanted. But if you tried to play that same game on a faster or slower machine, then the game itself would speed up or slow down.

> This is why old PCs used to have "turbo" buttons. New PCs were faster and couldn't play old games because the games would run too fast. Turning the turbo button *off* would slow the machine down and make old games playable.

These days, though, few developers have the luxury of knowing exactly what hardware their game will run on. Instead, our games must intelligently adapt to a variety of devices.

This is the other key job of a game loop: *it runs the game at a consistent speed despite differences in the underlying hardware.*

The Pattern

A **game loop** runs continuously during gameplay. Each turn of the loop, it **processes user input** without blocking, **updates the game state**, and **renders the game**. It tracks the passage of time to **control the rate of gameplay**.

When to Use It

Using the wrong pattern can be worse than using no pattern at all, so this section is normally here to caution against over-enthusiasm. The goal of design patterns isn't to cram as many into your codebase as you can.

But this pattern is a bit different. I can say with pretty good confidence that you *will* use this pattern. If you're using a game engine, you won't write it yourself, but it's still there.

> For me, this is the difference between an "engine" and a "library". With libraries, you own the main game loop and call into the library. An engine owns the loop and calls into *your* code.

You might think you won't need this if you're making a turn-based game. But even there, though the *game state* won't advance until the user takes their turn, the *visual* and *audible* states of the game usually do. Animation and music keep running even when the game is "waiting" for you to take your turn.

Keep in Mind

The loop we're talking about here is some of the most important code in your game. They say a program spends 90% of its time in 10% of the code. Your game loop will be firmly in

that 10%. Take care with this code, and be mindful of its efficiency.

> Made up statistics like this are why "real" engineers like mechanical and electrical engineers don't take us seriously.

You may need to coordinate with the platform's event loop

If you're building your game on top of an OS or platform that has a graphic UI and an event loop built in, then you have *two* application loops in play. They'll need to play nice together.

Sometimes, you can take control and make your loop the only one. For example, if you're writing a game against the venerable Windows API, your `main()` can just have a game loop. Inside, you can call `PeekMessage()` to handle and dispatch events from the OS. Unlike `GetMessage()`, `PeekMessage()` doesn't block waiting for user input, so your game loop will keep cranking.

Other platforms don't let you opt out of the event loop so easily. If you're targeting a web browser, the event loop is deeply built into browser's execution model. There, the event loop will run the show, and you'll use it as your game loop too. You'll call something like `requestAnimationFrame()` and it will call back into your code to keep the game running.

Sample Code

For such a long introduction, the code for a game loop is actually pretty straightforward. We'll walk through a couple of variations and go over their good and bad points.

The game loop drives AI, rendering, and other game systems, but those aren't the point of the pattern itself, so we'll just call into fictitious methods here. Actually implementing `render()`, `update()` and others is left as a (challenging!) exercise for the reader.

Run, run as fast as you can

We've already seen the simplest possible game loop:

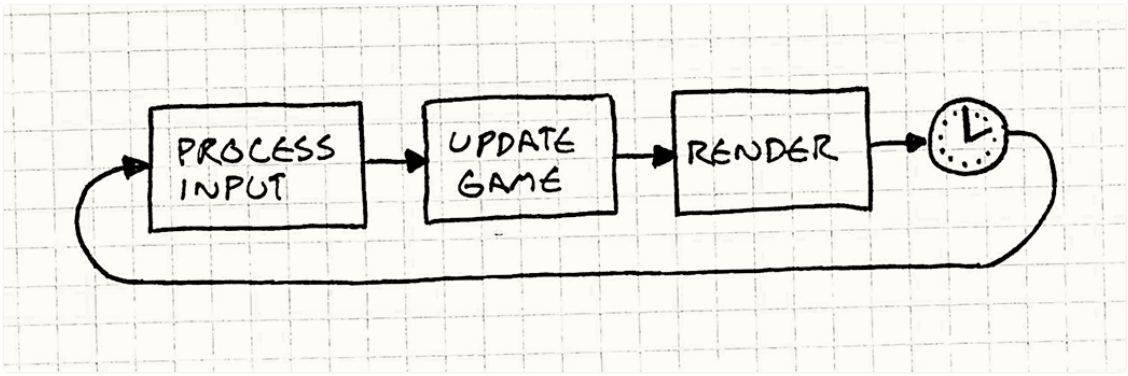
```
while (true)
{
  processInput();
  update();
  render();
}
```

The problem with it is you have no control over how fast the game runs. On a fast machine, that loop will spin so fast users won't be able to see what's going on. On a slow machine, the game will crawl. If you have a part of the game that's content-heavy or does

more AI or physics, the game will actually play slower there.

Take a little nap

The first variation we'll look at adds a simple fix. Say you want your game to run at 60 FPS. That gives you about 16 milliseconds per frame. As long as you can reliably do all of your game processing and rendering in less than that time, you can run at a steady frame rate. All you do is process the frame and then *wait* until it's time for the next one, like so:



The code looks a bit like this:

```
1000 ms / FPS = ms per frame.
```

```
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();

  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

The `sleep()` here makes sure the game doesn't run too *fast* if it processes a frame quickly. It *doesn't* help if your game runs too *slowly*. If it takes longer than 16ms to update and render the frame, your sleep time goes *negative*. If we had computers that could travel back in time, lots of things would be easier, but we don't.

Instead, the game slows down. You can work around this by doing less work each frame—cut down on the graphics and razzle dazzle or dumb down the AI. But that impacts the quality of gameplay for all users, even ones on fast machines.

One small step, one giant step

Let's try something a bit more sophisticated. The problem we have basically boils down to:

1. Each update advances game time by a certain amount.

2. It takes a certain amount of *real* time to process that.

If step two takes longer than step one, the game slows down. If it takes more than 16 ms of processing to advance game time by 16ms, it can't possibly keep up. But if we can advance the game by *more* than 16ms of game time in a single step, then we can update the game less frequently and still keep up.

The idea then is to choose a time step to advance based on how much *real* time passed since the last frame. The longer the frame takes, the bigger steps the game takes. It always keeps up with real time because it will take bigger and bigger steps to get there. They call this a *variable* or *fluid* time step. It looks like:

```
double lastTime = getCurrentTime();
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - lastTime;
  processInput();
  update(elapsed);
  render();
  lastTime = current;
}
```

Each frame, we determine how much *real* time passed since the last game update (`elapsed`). When we update the game state, we pass that in. The engine is then responsible for advancing the game world forward by that amount of time.

Say you've got a bullet shooting across the screen. With a fixed time step, in each frame, you'll move it according to its velocity. With a variable time step, you *scale that velocity by the elapsed time*. As the time step gets bigger, the bullet moves farther in each frame. That bullet will get across the screen in the *same* amount of *real* time whether it's twenty small fast steps or four big slow ones. This looks like a winner:

- The game plays at a consistent rate on different hardware.
- Players with faster machines are rewarded with smoother gameplay.

But, alas, there's a serious problem lurking ahead: we've made the game non-deterministic and unstable. Here's one example of the trap we've set for ourselves:

> "Deterministic" means that every time you run the program, if you give it the same inputs, you get the exact same outputs back. As you can imagine, it's much easier to track down bugs in deterministic programs—find the inputs that caused the bug the first time, and you can cause it every time.
>
> Computers are naturally deterministic; they follow programs mechanically. Non-determinism appears when the messy real world

> creeps in. For example, networking, the system clock, and thread scheduling all rely on bits of the external world outside of the program's control.

Say we've got a two-player networked game and Fred has some beast of a gaming machine while George is using his grandmother's antique PC. That aforementioned bullet is flying across both of their screens. On Fred's machine, the game is running super fast, so each time step is tiny. We cram, like, 50 frames in the second it takes the bullet to cross the screen. Poor George's machine can only fit in about five frames.

This means that on Fred's machine, the physics engine updates the bullet's position 50 times, but George's only does it five times. Most games use floating point numbers, and those are subject to *rounding error*. Each time you add two floating point numbers, the answer you get back can be a bit off. Fred's machine is doing ten times as many operations, so he'll accumulate a bigger error than George. The *same* bullet will end up in *different places* on their machines.

This is just one nasty problem a variable time step can cause, but there are more. In order to run in real time, game physics engines are approximations of the real laws of mechanics. To keep those approximations from blowing up, damping is applied. That damping is carefully tuned to a certain time step. Vary that, and the physics gets unstable.

> "Blowing up" is literal here. When a physics engine flakes out, objects can get completely wrong velocities and launch themselves into the air.

This instability is bad enough that this example is only here as a cautionary tale and to lead us to something better...

Play catch up

One part of the engine that usually *isn't* affected by a variable time step is rendering. Since the rendering engine captures an instant in time, it doesn't care how much time advanced since the last one. It renders things wherever they happen to be right then.

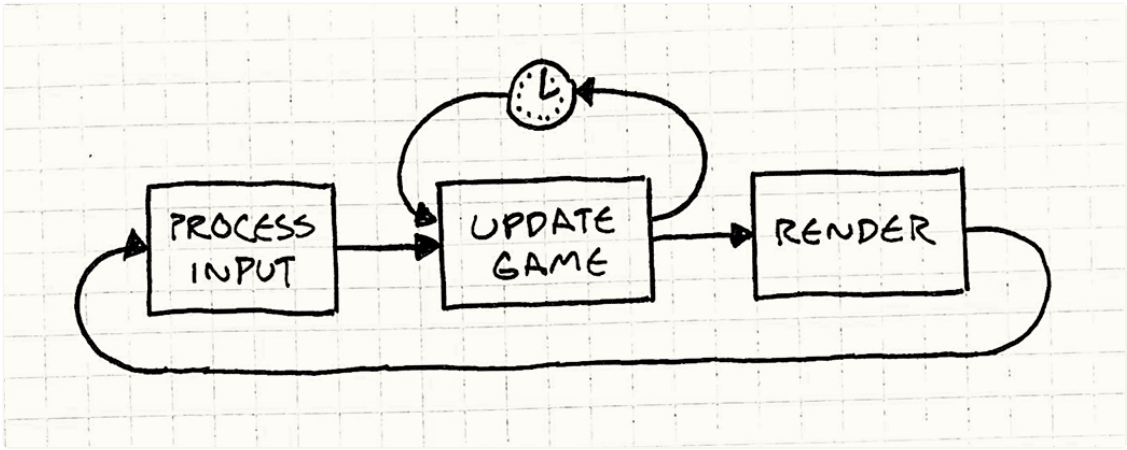
> This is more or less true. Things like motion blur can be affected by time step, but if they're a bit off, the player doesn't usually notice.

We can use this fact to our advantage. We'll *update* the game using a fixed time step because that makes everything simpler and more stable for physics and AI. But we'll allow flexibility in when we *render* in order to free up some processor time.

It goes like this: A certain amount of real time has elapsed since the last turn of the game loop. This is how much game time we need to simulate for the game's "now" to catch up with the player's. We do that using a *series* of *fixed* time steps. The code looks a bit like:

```
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();
}
```

There's a few pieces here. At the beginning of each frame, we update `lag` based on how much real time passed. This measures how far the game's clock is behind compared to the real world. We then have an inner loop to update the game, one fixed step at a time, until it's caught up. Once we're caught up, we render and start over again. You can visualize it sort of like this:



Note that the time step here isn't the *visible* frame rate anymore. `MS_PER_UPDATE` is just the *granularity* we use to update the game. The shorter this step is, the more processing time it takes to catch up to real time. The longer it is, the choppier the gameplay is. Ideally, you want it pretty short, often faster than 60 FPS, so that the game simulates with high fidelity on fast machines.

But be careful not to make it *too* short. You need to make sure the time step is greater than the time it takes to process an `update()`, even on the slowest hardware. Otherwise, your game simply can't catch up.

I left it out here, but you can safeguard this by having the inner update

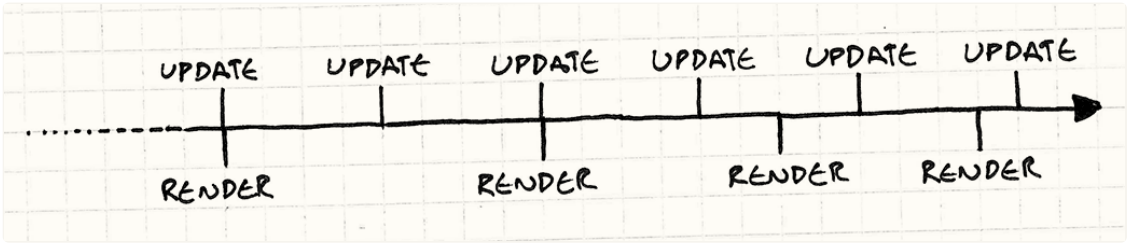
loop bail after a maximum number of iterations. The game will slow down then, but that's better than locking up completely.

Fortunately, we've bought ourselves some breathing room here. The trick is that we've *yanked rendering out of the update loop*. That frees up a bunch of CPU time. The end result is the game *simulates* at a constant rate using safe fixed time steps across a range of hardware. It's just that the player's *visible window* into the game gets choppier on a slower machine.

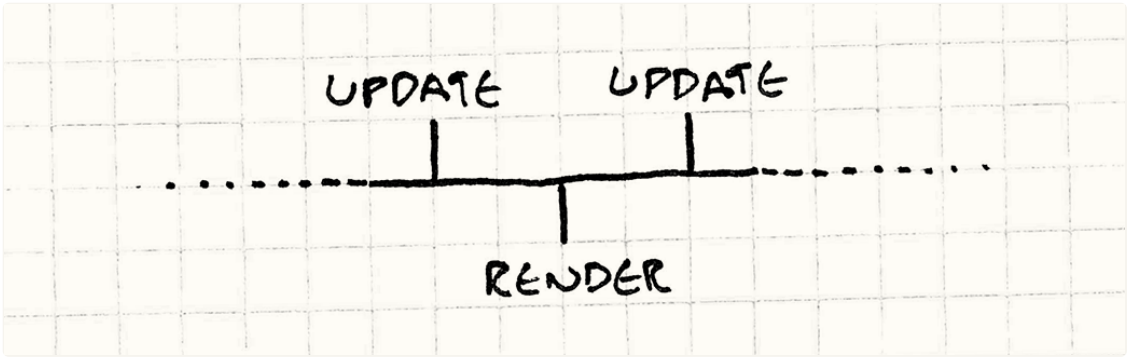
Stuck in the middle

There's one issue we're left with, and that's residual lag. We update the game at a fixed time step, but we render at arbitrary points in time. This means that from the user's perspective, the game will often display at a point in time between two updates.

Here's a timeline:



As you can see, we update at a nice tight, fixed interval. Meanwhile, we render whenever we can. It's less frequent than updating, and it isn't steady either. Both of those are OK. The lame part is that we don't always render right at the point of updating. Look at the third render time. It's right between two updates:



Imagine a bullet is flying across the screen. On the first update, it's on the left side. The second update moves it to the right side. The game is rendered at a point in time between those two updates, so the user expects to see that bullet in the center of the screen. With our current implementation, it will still be on the left side. This means motion looks jagged or stuttery.

Conveniently, we actually know *exactly* how far between update frames we are when we render: it's stored in `lag`. We bail out of the update loop when it's less than the update time step, not when it's *zero*. That leftover amount? That's how far into the next frame we

are.

When we go to render, we'll pass that in:

```
render(lag / MS_PER_UPDATE);
```

> We divide by `MS_PER_UPDATE` here to *normalize* the value. The value passed to `render()` will vary from 0 (right at the previous frame) to just under 1.0 (right at the next frame), regardless of the update time step. This way, the renderer doesn't have to worry about the frame rate. It just deals in values from 0 to 1.

The renderer knows each game object *and its current velocity*. Say that bullet is 20 pixels from the left side of the screen and is moving right 400 pixels per frame. If we are halfway between frames, then we'll end up passing 0.5 to `render()`. So it draws the bullet half a frame ahead, at 220 pixels. Ta-da, smooth motion.

Of course, it may turn out that that extrapolation is wrong. When we calculate the next frame, we may discover the bullet hit an obstacle or slowed down or something. We rendered its position interpolated between where it was on the last frame and where we *think* it will be on the next frame. But we don't know that until we've actually done the full update with physics and AI.

So the extrapolation is a bit of a guess and sometimes ends up wrong. Fortunately, though, those kinds of corrections usually aren't noticeable. At least, they're less noticeable than the stuttering you get if you don't extrapolate at all.

Design Decisions

Despite the length of this chapter, I've left out more than I've included. Once you throw in things like synchronizing with the display's refresh rate, multithreading, and GPUs, a real game loop can get pretty hairy. At a high level, though, here are a few questions you'll likely answer:

Do you own the game loop, or does the platform?

This is less a choice you make and more one that's made for you. If you're making a game that runs in a web browser, you pretty much *can't* write your own classic game loop. The browser's event-based nature precludes it. Likewise, if you're using an existing game engine, you will probably rely on its game loop instead of rolling your own.

- **Use the platform's event loop:**
    - *It's simple.* You don't have to worry about writing and optimizing the core loop of the game.

- *It plays nice with the platform.* You don't have to worry about explicitly giving the host time to process its own events, caching events, or otherwise managing the impedance mismatch between the platform's input model and yours.

- *You lose control over timing.* The platform will call your code as it sees fit. If that's not as frequently or as smoothly as you'd like, too bad. Worse, most application event loops weren't designed with games in mind and usually *are* slow and choppy.

- **Use a game engine's loop:**

  - *You don't have to write it.* Writing a game loop can get pretty tricky. Since that core code gets executed every frame, minor bugs or performance problems can have a large impact on your game. A tight game loop is one reason to consider using an existing engine.

  - *You don't get to write it.* Of course, the flip side to that coin is the loss of control if you *do* have needs that aren't a perfect fit for the engine.

- **Write it yourself:**

  - *Total control.* You can do whatever you want with it. You can design it specifically for the needs of your game.

  - *You have to interface with the platform.* Application frameworks and operating systems usually expect to have a slice of time to process events and do other work. If you own your app's core loop, it won't get any. You'll have to explicitly hand off control periodically to make sure the framework doesn't hang or get confused.

How do you manage power consumption?

This wasn't an issue five years ago. Games ran on things plugged into walls or on dedicated handheld devices. But with the advent of smartphones, laptops, and mobile gaming, the odds are good that you do care about this now. A game that runs beautifully but turns players' phones into space heaters before running out of juice thirty minutes later is not a game that makes people happy.

Now, you may need to think not only about making your game look great, but also use as little CPU as possible. There will likely be an *upper* bound to performance where you let the CPU sleep if you've done all the work you need to do in a frame.

- **Run as fast as it can:**

  This is what you're likely to do for PC games (though even those are increasingly being played on laptops). Your game loop will never explicitly tell the OS to sleep. Instead, any spare cycles will be spent cranking up the FPS or graphic fidelity.

  This gives you the best possible gameplay experience but, it will use as much power as

it can. If the player is on a laptop, they'll have a nice lap warmer.

- **Clamp the frame rate:**

  Mobile games are often more focused on the quality of gameplay than they are on maximizing the detail of the graphics. Many of these games will set an upper limit on the frame rate (usually 30 or 60 FPS). If the game loop is done processing before that slice of time is spent, it will just sleep for the rest.

  This gives the player a "good enough" experience and then goes easy on their battery beyond that.

How do you control gameplay speed?

A game loop has two key pieces: non-blocking user input and adapting to the passage of time. Input is straightforward. The magic is in how you deal with time. There are a near-infinite number of platforms that games can run on, and any single game may run on quite a few. How it accommodates that variation is key.

> Game-making seems to be part of human nature, because every time we've built a machine that can do computing, one of the first things we've done is made games on it. The PDP-1 was a 2 kHz machine with only 4,096 words of memory, yet Steve Russell and friends managed to create Spacewar! on it.

- **Fixed time step with no synchronization:**

  This was our first sample code. You just run the game loop as fast as you can.

  - *It's simple*. This is its main (well, only) virtue.
  - *Game speed is directly affected by hardware and game complexity*. And its main vice is that if there's any variation, it will directly affect the game speed. It's the fixie of game loops.

- **Fixed time step with synchronization:**

  The next step up on the complexity ladder is running the game at a fixed time step but adding a delay or synchronization point at the end of the loop to keep the game from running too fast.

  - *Still quite simple.* It's only one line of code more than the probably-too-simple-to-actually-work example. In most game loops, you will likely do synchronization *anyway*. You will probably double buffer your graphics and synchronize the buffer flip to the refresh rate of the display.
  - *It's power-friendly*. This is a surprisingly important consideration for mobile

games. You don't want to kill the user's battery unnecessarily. By simply sleeping for a few milliseconds instead of trying to cram ever more processing into each tick, you save power.

    - *The game doesn't play too fast.* This fixes half of the speed concerns of a fixed loop.

    - *The game can play too slowly.* If it takes too long to update and render a game frame, playback will slow down. Because this style doesn't separate updating from rendering, it's likely to hit this sooner than more advanced options. Instead of just dropping *rendering* frames to catch up, gameplay will slow down.

- **Variable time step:**

  I'll put this in here as an option in the solution space with the caveat that most game developers I know recommend against it. It's good to remember *why* it's a bad idea, though.

    - *It adapts to playing both too slowly and too fast.* If the game can't keep up with real time, it will just take larger and larger time steps until it does.

    - *It makes gameplay non-deterministic and unstable.* And this is the real problem, of course. Physics and networking in particular become much harder with a variable time step.

- **Fixed update time step, variable rendering:**

  The last option we covered in the sample code is the most complex, but also the most adaptable. It updates with a fixed time step, but it can drop *rendering* frames if it needs to to catch up to the player's clock.

    - *It adapts to playing both too slowly and too fast.* As long as the game can *update* in real time, the game won't fall behind. If the player's machine is top-of-the-line, it will respond with a smoother gameplay experience.

    - *It's more complex.* The main downside is there is a bit more going on in the implementation. You have to tune the update time step to be both as small as possible for the high-end, while not being too slow on the low end.

See Also

- The classic article on game loops is Glenn Fiedler's "Fix Your Timestep". This chapter wouldn't be the same without it.

- Witters' article on game loops is a close runner-up.

- The Unity framework has a complex game loop detailed in a wonderful illustration

here.

← Previous Chapter ≡ The Book Next Chapter →

Update Method

Intent

Simulate a collection of independent objects by telling each to process one frame of behavior at a time.

Motivation

The player's mighty valkyrie is on a quest to steal glorious jewels from where they rest on the bones of the long-dead sorcerer-king. She tentatively approaches the entrance of his magnificent crypt and is attacked by... *nothing*. No cursed statues shooting lightning at her. No undead warriors patrolling the entrance. She just walks right in and grabs the loot. Game over. You win.

Well, that won't do.

This crypt needs some guards—enemies our brave heroine can grapple with. First up, we want a re-animated skeleton warrior to patrol back and forth in front of the door. If you ignore everything you probably already know about game programming, the simplest possible code to make that skeleton lurch back and forth is something like:

> If the sorcerer-king wanted more intelligent behavior, he should have re-animated something that still had brain tissue.

```
while (true)
{
  // Patrol right.
  for (double x = 0; x < 100; x++)
  {
    skeleton.setX(x);
  }

  // Patrol left.
```

```
  for (double x = 100; x > 0; x--)
  {
    skeleton.setX(x);
  }
}
```

The problem here, of course, is that the skeleton moves back and forth, but the player never sees it. The program is locked in an infinite loop, which is not exactly a fun gameplay experience. What we actually want is for the skeleton to move one step *each frame*.

We'll have to remove those loops and rely on the outer game loop for iteration. That ensures the game keeps responding to user input and rendering while the guard is making his rounds. Like:

> Naturally, Game Loop is another pattern in this book.

```
Entity skeleton;
bool patrollingLeft = false;
double x = 0;

// Main game loop:
while (true)
{
  if (patrollingLeft)
  {
    x--;
    if (x == 0) patrollingLeft = false;
  }
  else
  {
    x++;
    if (x == 100) patrollingLeft = true;
  }

  skeleton.setX(x);

  // Handle user input and render game...
}
```

I did the before/after here to show you how the code gets more complex. Patrolling left and right used to be two simple `for` loops. It kept track of which direction the skeleton was moving implicitly by which loop was executing. Now that we have to yield to the outer game loop each frame and then resume where we left off, we have to track the direction explicitly using that `patrollingLeft` variable.

But this more or less works, so we keep going. A brainless bag of bones doesn't give yon Norse maiden too much of a challenge, so the next thing we add is a couple of enchanted statues. These will fire bolts of lightning at her every so often to keep her on her toes.

Continuing our, "what's the simplest way to code this" style, we end up with:

```
// Skeleton variables...
Entity leftStatue;
Entity rightStatue;
int leftStatueFrames = 0;
int rightStatueFrames = 0;

// Main game loop:
while (true)
{
  // Skeleton code...

  if (++leftStatueFrames == 90)
  {
    leftStatueFrames = 0;
    leftStatue.shootLightning();
  }

  if (++rightStatueFrames == 80)
  {
    rightStatueFrames = 0;
    rightStatue.shootLightning();
  }

  // Handle user input and render game...
}
```

You can tell this isn't trending towards code we'd enjoy maintaining. We've got an increasingly large pile of variables and imperative code all stuffed in the game loop, each handling one specific entity in the game. To get them all up and running at the same time, we've mushed their code together.

> Anytime "mushed" accurately describes your architecture, you likely have a problem.

The pattern we'll use to fix this is so simple you probably have it in mind already: *each entity in the game should encapsulate its own behavior*. This will keep the game loop uncluttered and make it easy to add and remove entities.

To do this, we need an *abstraction layer*, and we create that by defining an abstract `update()` method. The game loop maintains a collection of objects, but it doesn't know their concrete types. All it knows is that they can be updated. This separates each object's behavior both from the game loop and from the other objects.

Once per frame, the game loop walks the collection and calls `update()` on each object. This gives each one a chance to perform one frame's worth of behavior. By calling it on all

objects every frame, they all behave simultaneously.

> Since some stickler will call me on this, yes, they don't behave *truly concurrently*. While one object is updating, none of the others are. We'll get into this more in a bit.

The game loop has a dynamic collection of objects, so adding and removing them from the level is easy—just add and remove them from the collection. Nothing is hardcoded anymore, and we can even populate the level using some kind of data file, which is exactly what our level designers want.

The Pattern

The **game world** maintains a **collection of objects**. Each object implements an **update method** that **simulates one frame** of the object's behavior. Each frame, the game updates every object in the collection.

When to Use It

If the Game Loop pattern is the best thing since sliced bread, then the Update Method pattern is its butter. A wide swath of games featuring live entities that the player interacts with use this pattern in some form or other. If the game has space marines, dragons, Martians, ghosts, or athletes, there's a good chance it uses this pattern.

However, if the game is more abstract and the moving pieces are less like living actors and more like pieces on a chessboard, this pattern is often a poor fit. In a game like chess, you don't need to simulate all of the pieces concurrently, and you probably don't need to tell the pawns to update themselves every frame.

> You may not need to update their *behavior* each frame, but even in a board game, you may still want to update their *animation* every frame. This pattern can help with that too.

Update methods work well when:

- Your game has a number of objects or systems that need to run simultaneously.
- Each object's behavior is mostly independent of the others.
- The objects need to be simulated over time.

Keep in Mind

This pattern is pretty simple, so there aren't a lot of hidden surprises in its dark corners.

Still, every line of code has its ramifications.

Splitting code into single frame slices makes it more complex

When you compare the first two chunks of code, the second is a good bit more complex. Both simply make the skeleton guard walk back and forth, but the second one does this while yielding control to the game loop each frame.

That change is almost always necessary to handle user input, rendering, and the other stuff that the game loop takes care of, so the first example wasn't very practical. But it's worth keeping in mind that there's a big up front complexity cost when you julienne your behavioral code like this.

I say "almost" here because sometimes you can have your cake and eat it too. You can have straight-line code that never returns for your object behavior, while simultaneously having a number of objects running concurrently and coordinating with the game loop.

What you need is a system that lets you have multiple "threads" of execution going on at the same time. If the code for an object can pause and resume in the middle of what it's doing, instead of having to *return* completely, you can write it in a more imperative form.

Actual threads are usually too heavyweight for this to work well, but if your language supports lightweight concurrency constructs like generators, coroutines, or fibers, you may be able to use those.

The Bytecode pattern is another option that creates threads of execution at the application level.

You have to store state to resume where you left off each frame

In the first code sample, we didn't have any variables to indicate whether the guard was moving left or right. That was implicit based on which code was currently executing.

When we changed this to a one-frame-at-a-time form, we had to create a `patrollingLeft` variable to track that. When we return out of the code, the execution position is lost, so we need to explicitly store enough information to restore it on the next frame.

The State pattern can often help here. Part of the reason state machines are common in games is because (like their name implies) they store the kind of state that you need to pick up where you left off.

Objects all simulate each frame but are not truly concurrent

In this pattern, the game loops over a collection of objects and updates each one. Inside the `update()` call, most objects are able to reach out and touch the rest of the game

world, including other objects that are being updated. This means the *order* in which the objects are updated is significant.

If A comes before B in the list of objects, then when A updates, it will see B's previous state. But when B updates, it will see A's *new* state, since A has already been updated this frame. Even though from the player's perspective, everything is moving at the same time, the core of the game is still turn-based. It's just that a complete "turn" is only one frame long.

> If, for some reason, you decide you *don't* want your game to be sequential like this, you would need to use something like the Double Buffer pattern. That makes the order in which A and B update not matter because *both* of them will see the previous frame's state.

This is mostly a good thing as far as the game logic is concerned. Updating objects in parallel leads you to some unpleasant semantic corners. Imagine a game of chess where black and white moved at the same time. They both try to make a move that places a piece in the same currently empty square. How should this be resolved?

Updating sequentially solves this—each update incrementally changes the world from one valid state to the next with no period of time where things are ambiguous and need to be reconciled.

> It also helps online play since you have a serialized set of moves that can be sent over the network.

Be careful modifying the object list while updating

When you're using this pattern, a lot of the game's behavior ends up nestled in these update methods. That often includes code that adds or removes updatable objects from the game.

For example, say a skeleton guard drops an item when slain. With a new object, you can usually add it to the end of the list without too much trouble. You'll keep iterating over that list and eventually get to the new one at the end and update it too.

But that does mean that the new object gets a chance to act during the frame that it was spawned, before the player has even had a chance to see it. If you don't want that to happen, one simple fix is to cache the number of objects in the list at the beginning of the update loop and only update that many before stopping:

```
int numObjectsThisTurn = numObjects_;
for (int i = 0; i < numObjectsThisTurn; i++)
{
  objects_[i]->update();
```

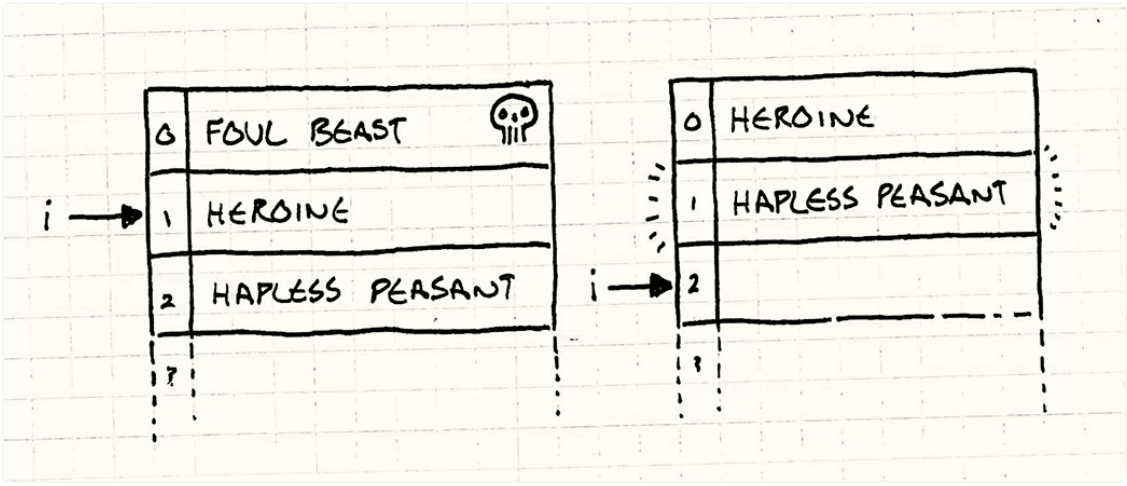
```
}
```

Here, `objects_` is an array of the updatable objects in the game, and `numObjects_` is its length. When new objects are added, it gets incremented. We cache the length in `numObjectsThisTurn` at the beginning of the loop so that the iteration stops before we get to any new objects added during the current frame.

A hairier problem is when objects are *removed* while iterating. You vanquish some foul beast and now it needs to get yanked out of the object list. If it happens to be before the current object you're updating in the list, you can accidentally skip an object:

```
for (int i = 0; i < numObjects_; i++)
{
  objects_[i]->update();
}
```

This simple loop increments the index of the object being updated each iteration. The left side of the illustration below shows what the array looks like while we're updating the heroine:



Since we're updating her, `i` is 1. She slays the foul beast so it gets removed from the array. The heroine shifts up to 0, and the hapless peasant shifts up to 1. After updating the heroine, `i` is incremented to 2. As you can see on the right, the hapless peasant is skipped over and never gets updated.

> A cheap solution is to walk the list *backwards* when you update. That way removing an object only shifts items that were already updated.

One fix is to just be careful when you remove objects and update any iteration variables to take the removal into account. Another is to defer removals until you're done walking the list. Mark the object as "dead", but leave it in place. During updating, make sure to skip any dead objects. Then, when that's done, walk the list again to remove the corpses.

If you have multiple threads processing the items in the update loop, then you are even more likely to defer any modification to it to avoid costly thread synchronization during updates.

Sample Code

This pattern is so straightforward that the sample code almost belabors the point. That doesn't mean the pattern isn't *useful*. It's useful in part *because* it's simple: it's a clean solution to a problem without a lot of ornamentation.

But to keep things concrete, let's walk through a basic implementation. We'll start with an `Entity` class that will represent the skeletons and statues:

```
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_;
};
```

I stuck a few things in there, but just the bare minimum we'll need later. Presumably in real code, there'd be lots of other stuff like graphics and physics. The important bit for this pattern is that it has an abstract `update()` method.

The game maintains a collection of these entities. In our sample, we'll put that in a class representing the game world:

```
class World
{
public:
  World()
  : numEntities_(0)
  {}
```

```
  void gameLoop();

private:
  Entity* entities_[MAX_ENTITIES];
  int numEntities_;
};
```

In a real-world program, you'd probably use an actual collection class, but I'm just using a vanilla array here to keep things simple.

Now that everything is set up, the game implements the pattern by updating each entity every frame:

```
void World::gameLoop()
{
  while (true)
  {
    // Handle user input...

    // Update each entity.
    for (int i = 0; i < numEntities_; i++)
    {
      entities_[i]->update();
    }

    // Physics and rendering...
  }
}
```

As the name of the method implies, this is an example of the Game Loop pattern.

Subclassing entities?!

There are some readers whose skin is crawling right now because I'm using inheritance on the main `Entity` class to define different behaviors. If you don't happen to see the problem, I'll provide some context.

When the game industry emerged from the primordial seas of 6502 assembly code and VBLANKs onto the shores of object-oriented languages, developers went into a software architecture fad frenzy. One of the biggest was using inheritance. Towering, Byzantine class hierarchies were built, big enough to blot out the sun.

It turns out that was a terrible idea and no one can maintain a giant class hierarchy without it crumbling around them. Even the Gang of Four knew this in 1994 when they wrote:

> *Favor 'object composition' over 'class inheritance'.*
>
> Between you and me, I think the pendulum has swung a bit too far *away* from subclassing. I generally avoid it, but being dogmatic about *not* using inheritance is as bad as being dogmatic about using it. You can use it in moderation without having to be a teetotaler.

When this realization percolated through the game industry, the solution that emerged was the Component pattern. Using that, `update()` would be on the entity's *components* and not on `Entity` itself. That lets you avoid creating complicated class hierarchies of entities to define and reuse behavior. Instead, you just mix and match components.

If I were making a real game, I'd probably do that too. But this chapter isn't about components. It's about `update()` methods, and the simplest way I can show them, with as few moving parts as possible, is by putting that method right on `Entity` and making a few subclasses.

> This one is.

Defining entities

OK, back to the task at hand. Our original motivation was to be able to define a patrolling skeleton guard and some lightning-bolt-unleashing magical statues. Let's start with our bony friend. To define his patrolling behavior, we make a new entity that implements `update()` appropriately:

```
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_;
};
```

As you can see, we pretty much just cut that chunk of code from the game loop earlier in the chapter and pasted it into `Skeleton`'s `update()` method. The one minor difference is that `patrollingLeft_` has been made into a field instead of a local variable. That way, its value sticks around between calls to `update()`.

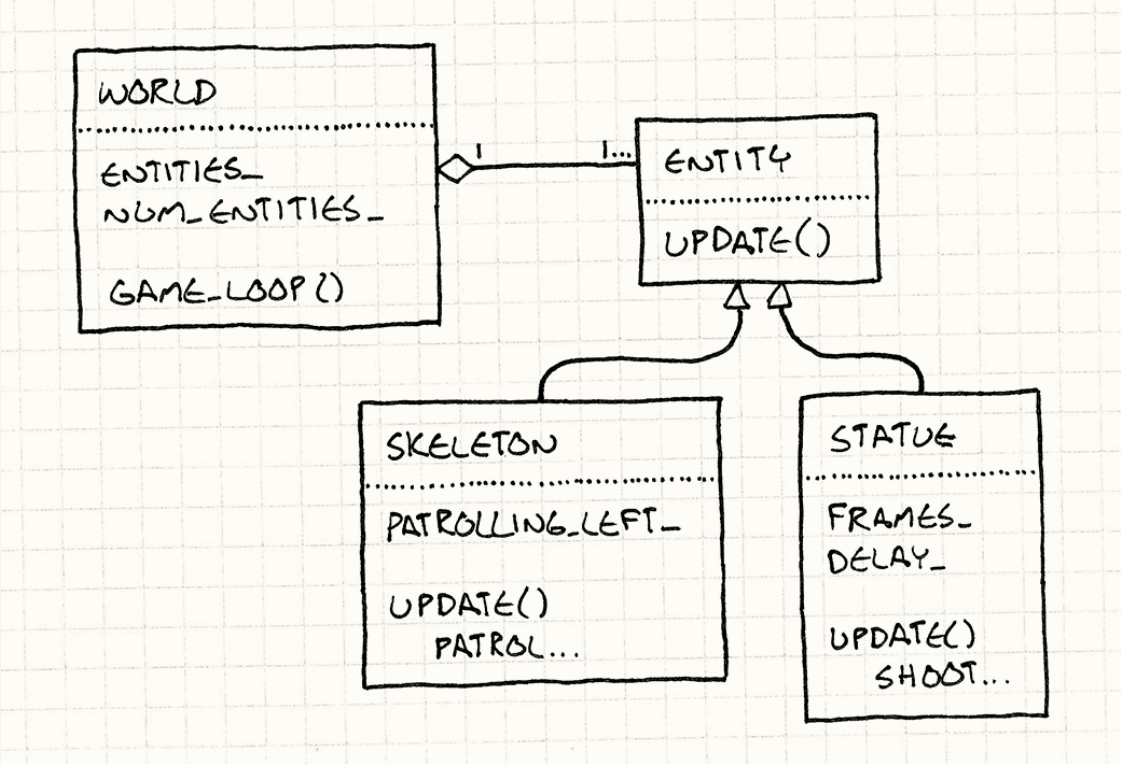
Let's do this again with the statue:

```
class Statue : public Entity
{
public:
  Statue(int delay)
  : frames_(0),
    delay_(delay)
  {}

  virtual void update()
  {
    if (++frames_ == delay_)
    {
      shootLightning();

      // Reset the timer.
      frames_ = 0;
    }
  }

private:
  int frames_;
  int delay_;

  void shootLightning()
  {
    // Shoot the lightning...
  }
};
```

Again, most of the change is moving code from the game loop into the class and renaming some stuff. In this case, though, we've actually made the codebase simpler. In the original nasty imperative code, there were separate local variables for each statue's frame counter and rate of fire.

Now that those have been moved into the `Statue` class itself, you can create as many as you want and each instance will have its own little timer. That's really the motivation behind this pattern—it's now much easier to add new entities to the game world because each one brings along everything it needs to take care of itself.

This pattern lets us separate *populating* the game world from *implementing* it. This in turn gives us the flexibility to populate the world using something like a separate data file or level editor.



Do people still care about UML? If so, here's what we just created.

Passing time

That's the key pattern, but I'll just touch on a common refinement. So far, we've assumed every call to `update()` advances the state of the game world by the same fixed unit of time.

I happen to prefer that, but many games use a *variable time step*. In those, each turn of the game loop may simulate a larger or smaller slice of time depending on how long it took to process and render the previous frame.

The Game Loop chapter has a lot more on the advantages and disadvantages of fixed and variable time steps.

That means that each `update()` call needs to know how far the hand of the virtual clock has swung, so you'll often see the elapsed time passed in. For example, we can make our patrolling skeleton handle a variable time step like so:

```
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);
    }
  }
}
```

Now, the distance the skeleton moves increases as the elapsed time grows. You can also see the additional complexity of dealing with a variable time step. The skeleton may overshoot the bounds of its patrol with a large time slice, and we have to handle that carefully.

Design Decisions

With a simple pattern like this, there isn't too much variation, but there are still a couple of knobs you can turn.

What class does the update method live on?

The most obvious and most important decision you'll make is what class to put `update()` on.

- **The entity class:**

  This is the simplest option if you already have an entity class since it doesn't bring any additional classes into play. This may work if you don't have too many kinds of entities, but the industry is generally moving away from this.

  Having to subclass `Entity` every time you want a new behavior is brittle and painful when you have a large number of different kinds. You'll eventually find yourself wanting to reuse pieces of code in a way that doesn't gracefully map to a single inheritance hierarchy, and then you're stuck.

- **The component class:**

  If you're already using the Component pattern, this is a no-brainer. It lets each

component update itself independently. In the same way that the Update Method pattern in general lets you decouple game entities from each other in the game world, this lets you decouple *parts of a single entity* from each other. Rendering, physics, and AI can all take care of themselves.

- **A delegate class:**

  There are other patterns that involve delegating part of a class's behavior to another object. The State pattern does this so that you can change an object's behavior by changing what it delegates to. The Type Object pattern does this so that you can share behavior across a bunch of entities of the same "kind".

  If you're using one of those patterns, it's natural to put `update()` on that delegated class. In that case, you may still have the `update()` method on the main class, but it will be non-virtual and will simply forward to the delegated object. Something like:

  ```
  void Entity::update()
  {
    // Forward to state object.
    state_->update();
  }
  ```

  Doing this lets you define new behavior by changing out the delegated object. Like using components, it gives you the flexibility to change behavior without having to define an entirely new subclass.

How are dormant objects handled?

You often have a number of objects in the world that, for whatever reason, temporarily don't need to be updated. They could be disabled, or off-screen, or not unlocked yet. If a large number of objects are in this state, it can be a waste of CPU cycles to walk over them each frame only to do nothing.

One alternative is to maintain a separate collection of just the "live" objects that do need updating. When an object is disabled, it's removed from the collection. When it gets re-enabled, it's added back. This way, you only iterate over items that actually have real work do to.

- **If you use a single collection containing inactive objects:**

  - *You waste time.* For inactive objects, you'll end up either checking some "am I enabled" flag or calling a method that does nothing.

    > In addition to wasted CPU cycles checking if the object is enabled and skipping past it, pointlessly iterating over objects can blow your data cache. CPUs optimize reads by loading memory from RAM into much faster on-chip caches. They do this speculatively

by assuming you're likely to read memory right after a location you just read.

When you skip over an object, you can skip past the end of the cache, forcing it to go and slowly pull in another chunk of main memory.

- **If you use a separate collection of only active objects:**
  - *You use extra memory to maintain the second collection.* There's still usually another master collection of all entities for cases where you need them all. In that case, this collection is technically redundant. When speed is tighter than memory (which it often is), this can still be a worthwhile trade-off.

    Another option to mitigate this is to have two collections, but have the other collection only contain the *inactive* entities instead of all of them.
  - *You have to keep the collections in sync.* When objects are created or completely destroyed (and not just made temporarily inactive), you have to remember to modify both the master collection and active object one.

The metric that should guide your approach here is how many inactive objects you tend to have. The more you have, the more useful it is to have a separate collection that avoids them during your core game loop.

See Also

- This pattern, along with Game Loop and Component, is part of a trinity that often forms the nucleus of a game engine.
- When you start caring about the cache performance of updating a bunch of entities or components in a loop each frame, the Data Locality pattern can help make that faster.
- The Unity framework uses this pattern in several classes, including `MonoBehaviour`.
- Microsoft's XNA platform uses this pattern both in the `Game` and `GameComponent` classes.
- The Quintus JavaScript game engine uses this pattern on its main `Sprite` class.

Behavioral Patterns

Game Programming Patterns

Once you've built your game's set and festooned it with actors and props, all that remains is to start the scene. For this, you need behavior—the screenplay that tells each entity in your game what to do.

Of course all code is "behavior", and all software is defining behavior, but what's different about games is often the *breadth* of it that you have to implement. While your word processor may have a long list of features, it pales in comparison with the number of inhabitants, items, and quests in your average role-playing game.

The patterns in this chapter help to quickly define and refine a large quantity of maintainable behavior. Type Objects create categories of behavior without the rigidity of defining an actual class. A Subclass Sandbox gives you a safe set of primitives you can use to define a variety of behaviors. The most advanced option is Bytecode, which moves behavior out of code entirely and into data.

The Patterns

- Bytecode
- Subclass Sandbox
- Type Object

Bytecode

Intent

Give behavior the flexibility of data by encoding it as instructions for a virtual machine.

Motivation

Making games may be fun, but it certainly ain't easy. Modern games require enormous, complex codebases. Console manufacturers and app marketplace gatekeepers have stringent quality requirements, and a single crash bug can prevent your game from shipping.

> I worked on a game that had six million lines of C++ code. For comparison, the software controlling the Mars Curiosity rover is less than half that.

At the same time, we're expected to squeeze every drop of performance out of the platform. Games push hardware like nothing else, and we have to optimize relentlessly just to keep pace with the competition.

To handle these high stability and performance requirements, we reach for heavyweight languages like C++ that have both low-level expressiveness to make the most of the hardware and rich type systems to prevent or at least corral bugs.

We pride ourselves on our skill at this, but it has its cost. Being a proficient programmer takes years of dedicated training, after which you must contend with the sheer scale of your codebase. Build times for large games can vary somewhere between "go get a coffee" and "go roast your own beans, hand-grind them, pull an espresso, foam some milk, and practice your latte art in the froth".

On top of these challenges, games have one more nasty constraint: *fun*. Players demand a play experience that's both novel and yet carefully balanced. That requires constant

iteration, but if every tweak requires bugging an engineer to muck around in piles of low-level code and then waiting for a glacial recompile, you've killed your creative flow.

Spell fight!

Let's say we're working on a magic-based fighting game. A pair of wizards square off and fling enchantments at each other until a victor is pronounced. We could define these spells in code, but that means an engineer has to be involved every time one is modified. When a designer wants to tweak a few numbers and get a feel for them, they have to recompile the entire game, reboot it, and get back into a fight.

Like most games these days, we also need to be able to update the game after it ships, both to fix bugs and to add new content. If all of these spells are hard-coded, then updating them means patching the actual game executable.

Let's take things a bit further and say that we also want to support *modding*. We want *users* to be able to create their own spells. If those are in code, that means every modder needs a full compiler toolchain to build the game, and we have to release the sources. Worse, if they have a bug in their spell, it can crash the game on some other player's machine.

Data > code

It's pretty clear that our engine's implementation language isn't the right fit. We need spells to be safely sandboxed from the core game. We want them to be easy to modify, easy to reload, and physically separate from the rest of the executable.

I don't know about you, but to me that sounds a lot like *data*. If we can define our behavior in separate data files that the game engine loads and "executes" in some way, we can achieve all of our goals.

We just need to figure out what "execute" means for data. How do you make some bytes in a file express behavior? There are a few ways to do this. I think it will help you get a picture of *this* pattern's strengths and weaknesses if we compare it to another one: the Interpreter [GoF] pattern.

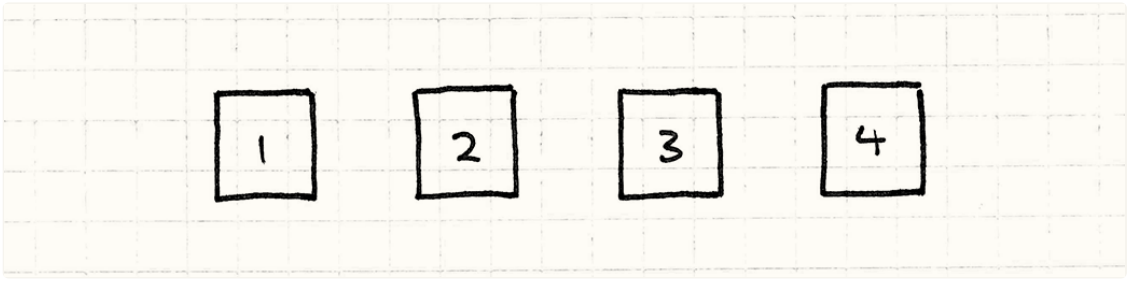
The Interpreter pattern

I could write a whole chapter on this pattern, but four other guys already covered that for me. Instead, I'll cram the briefest of introductions in here. It starts with a language—think *programming* language—that you want to execute. Say, for example, it supports arithmetic expressions like this:

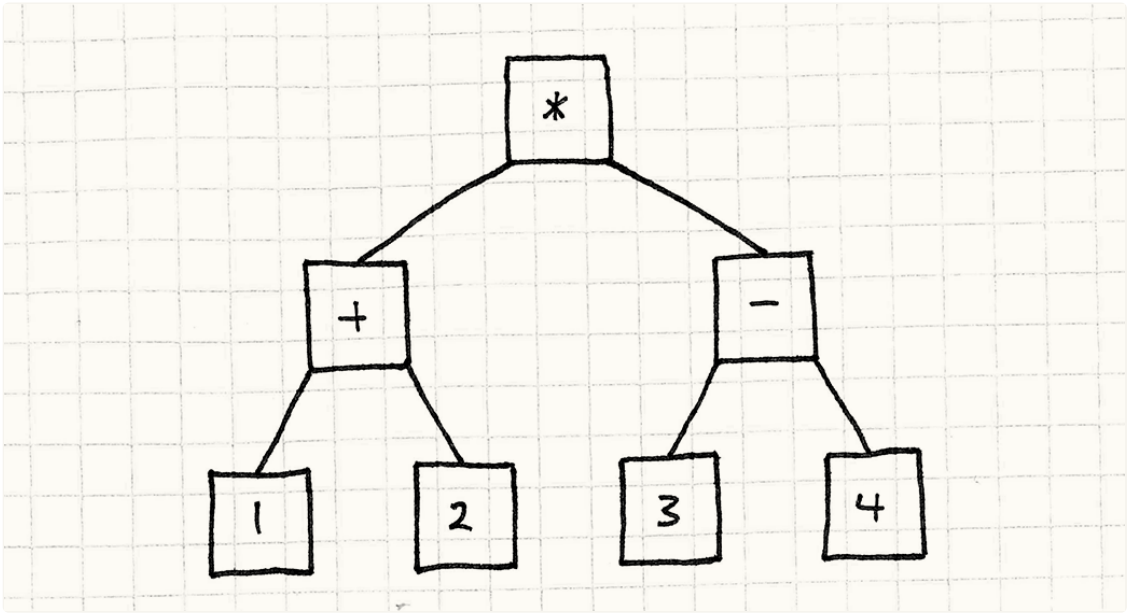
```
(1 + 2) * (3 - 4)
```

Then, you take each piece of that expression, each rule in the language's grammar, and

turn it into an *object*. The number literals will be objects:



Basically, they're little wrappers around the raw value. The operators will be objects too, and they'll have references to their operands. If you take into account the parentheses and precedence, that expression magically turns into a little tree of objects like so:



> What "magic" is this? It's simple—*parsing*. A parser takes a string of characters and turns it into an *abstract syntax tree*, a collection of objects representing the grammatical structure of the text.
>
> Whip up one of these and you've got yourself half of a compiler.

The Interpreter pattern isn't about *creating* that tree; it's about *executing* it. The way it works is pretty clever. Each object in the tree is an expression or a subexpression. In true object-oriented fashion, we'll let expressions evaluate themselves.

First, we define a base interface that all expressions implement:

```
class Expression
{
public:
  virtual ~Expression() {}
  virtual double evaluate() = 0;
};
```

Then, we define a class that implements this interface for each kind of expression in our language's grammar. The simplest one is numbers:

```
class NumberExpression : public Expression
{
public:
  NumberExpression(double value)
  : value_(value)
  {}

  virtual double evaluate()
  {
    return value_;
  }

private:
  double value_;
};
```

A literal number expression simply evaluates to its value. Addition and multiplication are a bit more complex because they contain subexpressions. Before they can evaluate themselves, they need to recursively evaluate their subexpressions. Like so:

```
class AdditionExpression : public Expression
{
public:
  AdditionExpression(Expression* left, Expression* right)
  : left_(left),
    right_(right)
  {}

  virtual double evaluate()
  {
    // Evaluate the operands.
    double left = left_->evaluate();
    double right = right_->evaluate();

    // Add them.
    return left + right;
  }

private:
  Expression* left_;
  Expression* right_;
};
```

I'm sure you can figure out what the implementation of multiply looks like.

Pretty neat right? Just a couple of simple classes and now we can represent and evaluate arbitrarily complex arithmetic expressions. We just need to create the right objects and wire them up correctly.

> Ruby was implemented like this for something like 15 years. At version 1.9, they switched to bytecode like this chapter describes. Look how much time I'm saving you!

It's a beautiful, simple pattern, but it has some problems. Look up at the illustration. What do you see? Lots of little boxes, and lots of arrows between them. Code is represented as a sprawling fractal tree of tiny objects. That has some unpleasant consequences:

- Loading it from disk requires instantiating and wiring up tons of these small objects.
- Those objects and the pointers between them use a lot of memory. On a 32-bit machine, that little arithmetic expression up there takes up at least 68 bytes, not including padding.

  > If you're playing along at home, don't forget to take into account the vtable pointers.

- Traversing the pointers into subexpressions is murder on your data cache. Meanwhile, all of those virtual method calls wreak carnage on your instruction cache.

  > See the chapter on Data Locality for more on what the cache is and how it affects your performance.

Put those together, and what do they spell? S-L-O-W. There's a reason most programming languages in wide use aren't based on the Interpreter pattern. It's just too slow, and it uses up too much memory.

Machine code, virtually

Consider our game. When we run it, the player's computer doesn't traverse a bunch of C++ grammar tree structures at runtime. Instead, we compile it ahead of time to machine code, and the CPU runs that. What's machine code got going for it?

- *It's dense.* It's a solid, contiguous blob of binary data, and no bit goes to waste.
- *It's linear.* Instructions are packed together and executed one right after another. No jumping around in memory (unless you're doing actual control flow, of course).
- *It's low-level.* Each instruction does one relatively minimal thing, and interesting behavior comes from *composing* them.
- *It's fast.* As a consequence of all of these (well, and the fact that it's implemented

directly in hardware), machine code runs like the wind.

This sounds swell, but we don't want actual machine code for our spells. Letting users provide machine code which our game executes is just begging for security problems. What we need is a compromise between the performance of machine code and the safety of the Interpreter pattern.

What if instead of loading actual machine code and executing it directly, we defined our own *virtual* machine code? We'd then write a little emulator for it in our game. It would be similar to machine code—dense, linear, relatively low-level—but would also be handled entirely by our game so we could safely sandbox it.

> This is why many game consoles and iOS don't allow programs to execute machine code loaded or generated at runtime. That's a drag because the fastest programming language implementations do exactly that. They contain a "just-in-time" compiler, or *JIT*, that translates the language to optimized machine code on the fly.

We'd call our little emulator a *virtual machine* (or "VM" for short), and the synthetic binary machine code it runs *bytecode*. It's got the flexibility and ease of use of defining things in data, but it has better performance than higher-level representations like the Interpreter pattern.

> In programming language circles, "virtual machine" and "interpreter" are synonymous, and I use them interchangeably here. When I refer to the Gang of Four's Interpreter pattern, I'll use "pattern" to make it clear.

This sounds daunting, though. My goal for the rest of this chapter is to show you that if you keep your feature list pared down, it's actually pretty approachable. Even if you end up not using this pattern yourself, you'll at least have a better understanding of Lua and many other languages which are implemented using it.

The Pattern

An **instruction set** defines the low-level operations that can be performed. A series of instructions is encoded as a **sequence of bytes**. A **virtual machine** executes these instructions one at a time, using a **stack for intermediate values**. By combining instructions, complex high-level behavior can be defined.

When to Use It

This is the most complex pattern in this book, and it's not something to throw into your game lightly. Use it when you have a lot of behavior you need to define and your game's implementation language isn't a good fit because:

- It's too low-level, making it tedious or error-prone to program in.
- Iterating on it takes too long due to slow compile times or other tooling issues.
- It has too much trust. If you want to ensure the behavior being defined can't break the game, you need to sandbox it from the rest of the codebase.

Of course, that list describes a bunch of your game. Who doesn't want a faster iteration loop or more safety? However, that doesn't come for free. Bytecode is slower than native code, so it isn't a good fit for performance-critical parts of your engine.

Keep in Mind

There's something seductive about creating your own language or system-within-a-system. I'll be doing a minimal example here, but in the real world, these things tend to grow like vines.

> For me, game development is seductive in the same way. In both cases, I'm striving to create a virtual space for others to play and be creative in.

Every time I see someone define a little language or a scripting system, they say, "Don't worry, it will be tiny." Then, inevitably, they add more and more little features until it's a full-fledged language. Except, unlike some other languages, it grew in an ad-hoc, organic fashion and has all of the architectural elegance of a shanty town.

> For example, see every templating language ever.

Of course, there's nothing *wrong* with making a full-fledged language. Just make sure you do so deliberately. Otherwise, be very careful to control the scope of what your bytecode can express. Put a short leash on it before it runs away from you.

You'll need a front-end

Low-level bytecode instructions are great for performance, but a binary bytecode format is *not* what your users are going to author. One reason we're moving behavior out of code is so that we can express it at a *higher* level. If C++ is too low-level, making your users effectively write in assembly language—even one of your own design—isn't an improvement!

> Challenging that assertion is the venerable game RoboWar. In that game, *players* write little programs to control a robot in a language very similar to assembly and the kind of instruction sets we'll be discussing here.
>
> It was my first introduction to assembly-like languages.

Much like the Gang of Four's Interpreter pattern, it's assumed that you also have some way to *generate* the bytecode. Usually, users author their behavior in some higher-level format, and a tool translates that to the bytecode that our virtual machine understands. In other words, a compiler.

I know, that sounds scary. That's why I'm mentioning it here. If you don't have the resources to build an authoring tool, then bytecode isn't for you. But as we'll see later, it may not be as bad as you think.

You'll miss your debugger

Programming is hard. We know what we want the machine to do, but we don't always communicate that correctly—we write bugs. To help find and fix those, we've amassed a pile of tools to understand what our code is doing wrong, and how to right it. We have debuggers, static analyzers, decompilers, etc. All of those tools are designed to work with some existing language: either machine code or something higher level.

When you define your own bytecode VM, you leave those tools behind. Sure, you can step through the VM in your debugger, but that tells you what the VM *itself* is doing, and not what the bytecode it's interpreting is up to. It certainly doesn't help you map that bytecode back to the high-level form it was compiled from.

If the behavior you're defining is simple, you can scrape by without too much tooling to help you debug it. But as the scale of your content grows, plan to invest real time into features that help users see what their bytecode is doing. Those features might not ship in your game, but they'll be critical to ensure that you actually *can* ship your game.

> Of course, if you want your game to be moddable, then you *will* ship those features, and they'll be even more important.

Sample Code

After the previous couple of sections, you might be surprised how straightforward the implementation is. First, we need to craft an instruction set for our VM. Before we start thinking about bytecode and stuff, let's just think about it like an API.

A magical API

If we were defining spells in straight C++ code, what kind of API would we need for that code to call into? What are the basic operations in the game engine that spells are defined in terms of?

Most spells ultimately change one of the stats of a wizard, so we'll start with a couple for that:

```
void setHealth(int wizard, int amount);
void setWisdom(int wizard, int amount);
void setAgility(int wizard, int amount);
```

The first parameter identifies which wizard is affected, say `0` for the player's and `1` for their opponent. This way, healing spells can affect the player's own wizard, while damaging attacks harm their nemesis. These three little methods cover a surprisingly wide variety of magical effects.

If the spells just silently tweaked stats, the game logic would be fine, but playing it would bore players to tears. Let's fix that:

```
void playSound(int soundId);
void spawnParticles(int particleType);
```

These don't affect gameplay, but they crank up the intensity of the gameplay *experience*. We could add more for camera shake, animation, etc., but this is enough to get us started.

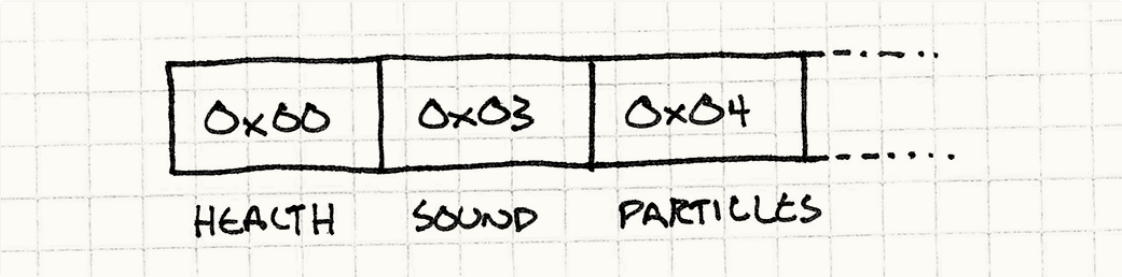
A magical instruction set

Now let's see how we'd turn this *programmatic* API into something that can be controlled from data. Let's start small and then we'll work our way up to the whole shebang. For now, we'll ditch all of the parameters to these methods. We'll say the `set___()` methods always affect the player's own wizard and always max out the stat. Likewise, the FX operations always play a single hard-coded sound and particle effect.

Given that, a spell is just a series of instructions. Each one identifies which operation you want to perform. We can enumerate them:

```
enum Instruction
{
  INST_SET_HEALTH      = 0x00,
  INST_SET_WISDOM      = 0x01,
  INST_SET_AGILITY     = 0x02,
  INST_PLAY_SOUND      = 0x03,
  INST_SPAWN_PARTICLES = 0x04
};
```

To encode a spell in data, we store an array of `enum` values. We've only got a few different primitives, so the range of `enum` values easily fits into a byte. This means the code for a spell is just a list of bytes—ergo "bytecode".



Some bytecode VMs use more than a single byte for each instruction and have more complicated rules for how they are decoded. Actual machine code on common chips like x86 is a good bit more complex.

But a single byte is good enough for the Java Virtual Machine and Microsoft's Common Language Runtime, which forms the backbone of the .NET platform, and it's good enough for us.

To execute a single instruction, we see which primitive it is and dispatch to the right API method:

```
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;
}
```

In this way, our interpreter forms the bridge between code world and data world. We can wrap this in a little VM that executes an entire spell like so:

```
class VM
{
public:
  void interpret(char bytecode[], int size)
```

```
  {
    for (int i = 0; i < size; i++)
    {
      char instruction = bytecode[i];
      switch (instruction)
      {
        // Cases for each instruction...
      }
    }
  }
};
```

Type that in and you'll have written your first virtual machine. Unfortunately, it's not very flexible. We can't define a spell that touches the player's opponent or lowers a stat. We can only play one sound!

To get something that starts to have the expressive feel of an actual language, we need to get parameters in here.

A stack machine

To execute a complex nested expression, you start with the innermost subexpressions. You calculate those, and the results flow outward as arguments to the expressions that contain them until eventually, the whole expression has been evaluated.

The Interpreter pattern models this explicitly as a tree of nested objects, but we want the speed of a flat list of instructions. We still need to ensure results from subexpressions flow to the right surrounding expressions. But, since our data is flattened, we'll have to use the *order* of the instructions to control that. We'll do it the same way your CPU does—with a stack.

> This architecture is unimaginatively called a *stack machine*. Programming languages like Forth, PostScript, and Factor expose this model directly to the user.

```
class VM
{
public:
  VM()
  : stackSize_(0)
  {}

  // Other stuff...

private:
  static const int MAX_STACK = 128;
  int stackSize_;
  int stack_[MAX_STACK];
```

```
};
```

The VM maintains an internal stack of values. In our example, the only kinds of values our instructions work with are numbers, so we can use a simple array of `int`s. Whenever a bit of data needs to work its way from one instruction to another, it gets there through the stack.

Like the name implies, values can be pushed onto or popped off of the stack, so let's add a couple of methods for that:

```
class VM
{
private:
  void push(int value)
  {
    // Check for stack overflow.
    assert(stackSize_ < MAX_STACK);
    stack_[stackSize_++] = value;
  }

  int pop()
  {
    // Make sure the stack isn't empty.
    assert(stackSize_ > 0);
    return stack_[--stackSize_];
  }

  // Other stuff...
};
```

When an instruction needs to receive parameters, it pops them off the stack like so:

```
switch (instruction)
{
  case INST_SET_HEALTH:
  {
    int amount = pop();
    int wizard = pop();
    setHealth(wizard, amount);
    break;
  }

  case INST_SET_WISDOM:
  case INST_SET_AGILITY:
    // Same as above...

  case INST_PLAY_SOUND:
    playSound(pop());
    break;
```

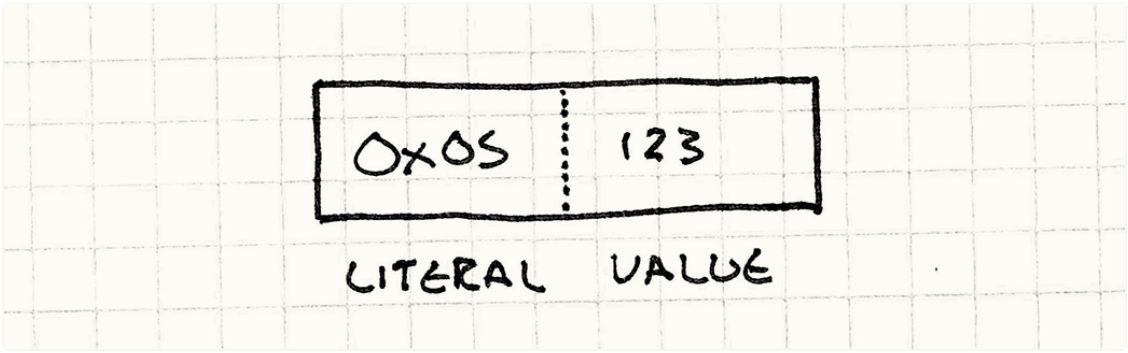
```
    case INST_SPAWN_PARTICLES:
      spawnParticles(pop());
      break;
  }
```

To get some values *onto* that stack, we need one more instruction: a literal. It represents a raw integer value. But where does *it* get its value from? How do we avoid some turtles-all-the-way-down infinite regress here?

The trick is to take advantage of the fact that our instruction stream is a sequence of bytes—we can stuff the number directly in the byte array. We define another instruction type for a number literal like so:

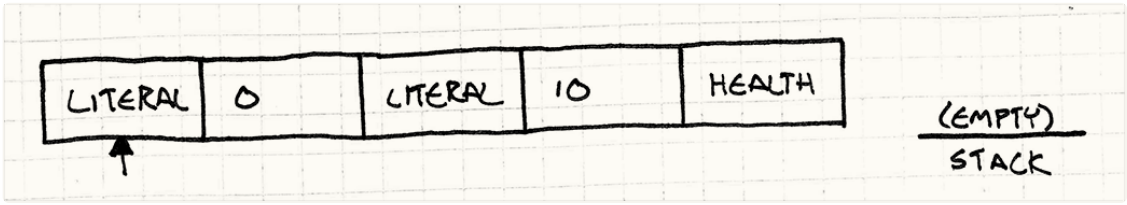
```
case INST_LITERAL:
{
  // Read the next byte from the bytecode.
  int value = bytecode[++i];
  push(value);
  break;
}
```

Here, I'm reading a single byte for the value to avoid the fiddly code required to decode a multiple-byte integer, but in a real implementation, you'll want to support literals that cover your full numeric range.



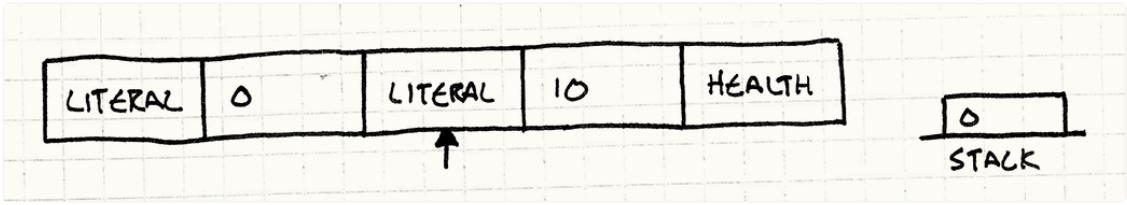
It reads the next byte in the bytecode stream *as a number* and pushes it onto the stack.

Let's string a few of these instructions together and watch the interpreter execute them to get a feel for how the stack works. We start with an empty stack and the interpreter pointing to the first instruction:

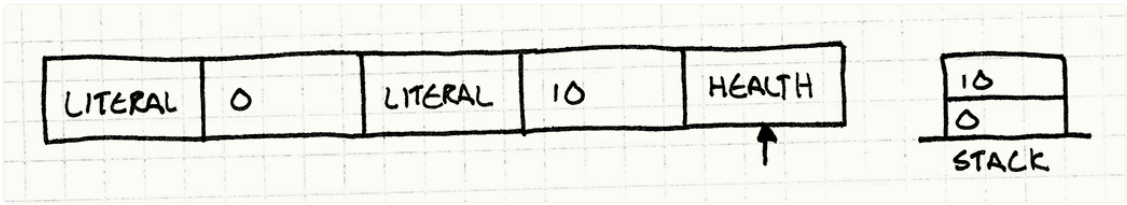


First, it executes the first `INST_LITERAL`. That reads the next byte from the bytecode (`0`)

and pushes it onto the stack:



Then, it executes the second `INST_LITERAL`. That reads the `10` and pushes it:



Finally, it executes `INST_SET_HEALTH`. That pops `10` and stores it in `amount`, then pops `0` and stores it in `wizard`. Then, it calls `setHealth()` with those parameters.

Ta-da! We've got a spell that sets the player's wizard's health to ten points. Now, we've got enough flexibility to define spells that set either wizard's stats to whatever amounts we want. We can also play different sounds and spawn particles.

But... this still feels like a *data* format. We can't, for example, raise a wizard's health by half of their wisdom. Our designers want to be able to express *rules* for spells, not just *values*.

Behavior = composition

If we think of our little VM like a programming language, all it supports now is a couple of built-in functions and constant parameters for them. To get bytecode to feel like *behavior*, what we're missing is *composition*.

Our designers need to be able to create expressions that combine different values in interesting ways. For a simple example, they want spells that modify a stat *by* a certain amount instead of *to* a certain amount.

That requires taking into account a stat's current value. We have instructions for *writing* a stat, but we need to add a couple to *read* stats:

```
case INST_GET_HEALTH:
{
  int wizard = pop();
  push(getHealth(wizard));
  break;
}

case INST_GET_WISDOM:
case INST_GET_AGILITY:
```

```
  // You get the idea...
```

As you can see, these work with the stack in both directions. They pop a parameter to determine which wizard to get the stat for, and then they look up the stat's value and push that back onto the stack.

This lets us write spells that copy stats around. We could create a spell that set a wizard's agility to their wisdom or a strange incantation that set one wizard's health to mirror his opponent's.

Better, but still quite limited. Next, we need arithmetic. It's time our baby VM learned how to add 1 + 1. We'll add a few more instructions. By now, you've probably got the hang of it and can guess how they look. I'll just show addition:

```
case INST_ADD:
{
  int b = pop();
  int a = pop();
  push(a + b);
  break;
}
```

Like our other instructions, it pops a couple of values, does a bit of work, and then pushes the result back. Up until now, every new instruction gave us an incremental improvement in expressiveness, but we just made a big leap. It isn't obvious, but we can now handle all sorts of complicated, deeply nested arithmetic expressions.

Let's walk through a slightly more complex example. Say we want a spell that increases the player's wizard's health by the average of their agility and wisdom. In code, that's:

```
setHealth(0, getHealth(0) +
    (getAgility(0) + getWisdom(0)) / 2);
```

You might think we'd need instructions to handle the explicit grouping that parentheses give you in the expression here, but the stack supports that implicitly. Here's how you could evaluate this by hand:

1. Get the wizard's current health and remember it.
2. Get the wizard's agility and remember it.
3. Do the same for their wisdom.
4. Get those last two, add them, and remember the result.
5. Divide that by two and remember the result.
6. Recall the wizard's health and add it to that result.
7. Take that result and set the wizard's health to that value.

Do you see all of those "remembers" and "recalls"? Each "remember" corresponds to a

push, and the “recalls” are pops. That means we can translate this to bytecode pretty easily. For example, the first line to get the wizard’s current health is:

```
LITERAL 0
GET_HEALTH
```

This bit of bytecode pushes the wizard’s health onto the stack. If we mechanically translate each line like that, we end up with a chunk of bytecode that evaluates our original expression. To give you a feel for how the instructions compose, I’ve done that below.

To show how the stack changes over time, we’ll walk through a sample execution where the wizard’s current stats are 45 health, 7 agility, and 11 wisdom. Next to each instruction is what the stack looks like after executing it and then a little comment explaining the instruction’s purpose:

```
LITERAL 0    [0]            # Wizard index
LITERAL 0    [0, 0]         # Wizard index
GET_HEALTH   [0, 45]        # getHealth()
LITERAL 0    [0, 45, 0]     # Wizard index
GET_AGILITY  [0, 45, 7]     # getAgility()
LITERAL 0    [0, 45, 7, 0]  # Wizard index
GET_WISDOM   [0, 45, 7, 11] # getWisdom()
ADD          [0, 45, 18]    # Add agility and wisdom
LITERAL 2    [0, 45, 18, 2] # Divisor
DIVIDE       [0, 45, 9]     # Average agility and wisdom
ADD          [0, 54]        # Add average to current health
SET_HEALTH   []             # Set health to result
```

If you watch the stack at each step, you can see how data flows through it almost like magic. We push `0` for the wizard index at the beginning, and it just hangs around at the bottom of the stack until we finally need it for the last `SET_HEALTH` at the end.

> Maybe my threshold for “magic” is a little too low here.

A virtual machine

I could keep going, adding more and more instructions, but this is a good place to stop. As it is, we’ve got a nice little VM that lets us define fairly open-ended behavior using a simple, compact data format. While “bytecode” and “virtual machines” sound intimidating, you can see they’re often as simple as a stack, a loop, and a switch statement.

Remember our original goal to have behavior be nicely sandboxed? Now that you’ve seen exactly how the VM is implemented, it’s obvious that we’ve accomplished that. The bytecode can’t do anything malicious or reach out into weird parts of the game engine because we’ve only defined a few instructions that touch the rest of the game.

We control how much memory it uses by how big of a stack we create, and we're careful to make sure it can't overflow that. We can even control how much *time* it uses. In our instruction loop, we can track how many we've executed and bail out if it goes over some limit.

> Controlling execution time isn't necessary in our sample because we don't have any instructions for looping. We could limit execution time by limiting the total size of the bytecode. This also means our bytecode isn't Turing-complete.

There's just one problem left: actually creating the bytecode. So far, we've taken bits of pseudocode and compiled them to bytecode by hand. Unless you've got a *lot* of free time, that's not going to work in practice.

Spellcasting tools

One of our initial goals was to have a *higher*-level way to author behavior, but we've gone and created something *lower*-level than C++. It has the runtime performance and safety we want, but absolutely none of the designer-friendly usability.

To fill that gap, we need some tooling. We need a program that lets users define the high-level behavior of a spell and then takes that and generates the appropriate low-level stack machine bytecode.

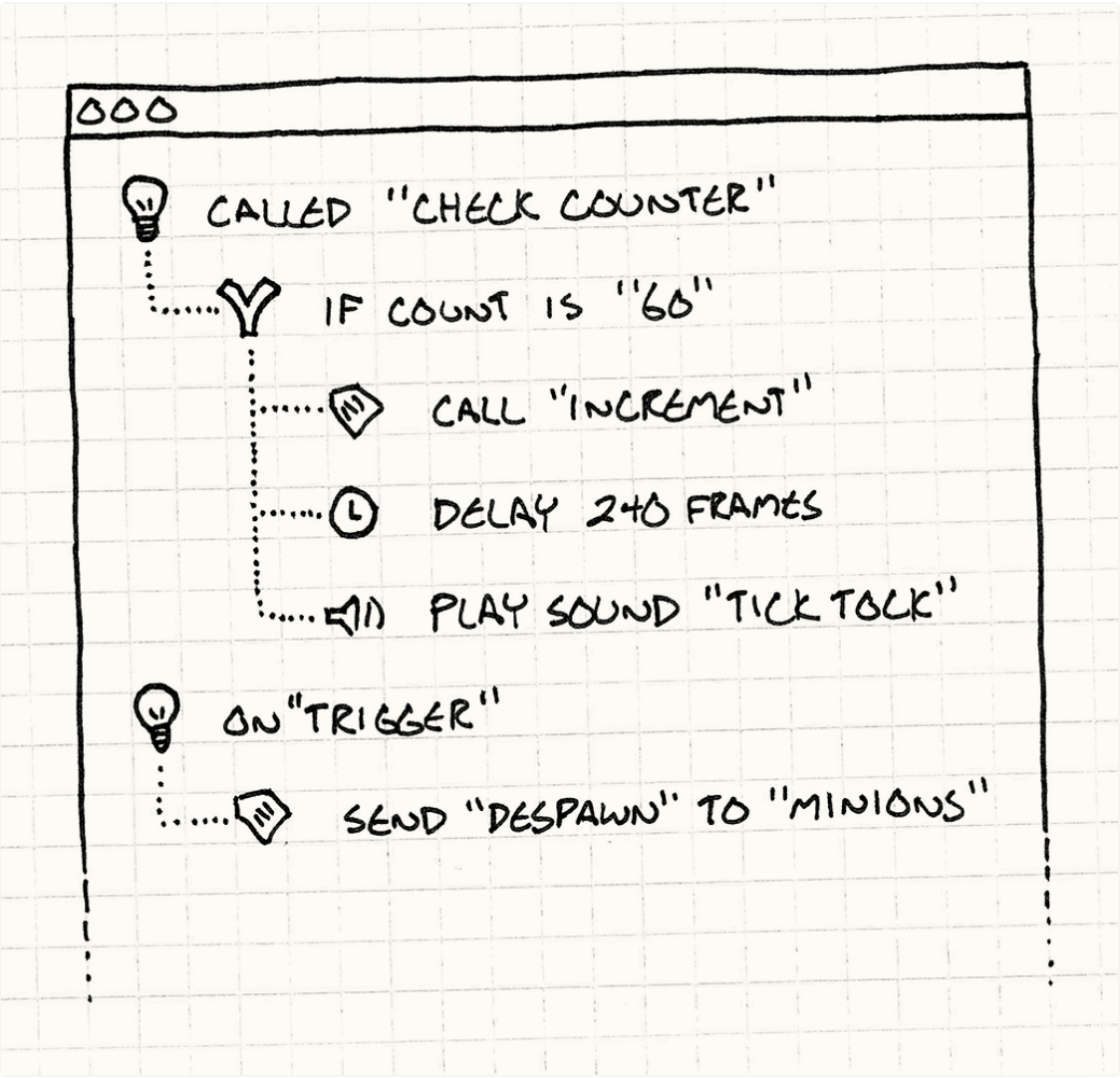
That probably sounds way harder than making the VM. Many programmers were dragged through a compilers class in college and took away from it nothing but PTSD triggered by the sight of a book with a dragon on the cover or the words "lex" and "yacc".

> I'm referring, of course, to the classic text *Compilers: Principles, Techniques, and Tools*.

In truth, compiling a text-based language isn't that bad, though it's a *bit* too broad of a topic to cram in here. However, you don't have to do that. What I said we need is a *tool*—it doesn't have to be a *compiler* whose input format is a *text file*.

On the contrary, I encourage you to consider building a graphical interface to let users define their behavior, especially if the people using it won't be highly technical. Writing text that's free of syntax errors is difficult for people who haven't spent years getting used to a compiler yelling at them.

Instead, you can build an app that lets users "script" by clicking and dragging little boxes, pulling down menu items, or whatever else makes sense for the kind of behavior you want them to create.



The scripting system I wrote for Henry Hatsworth in the Puzzling Adventure worked like this.

The nice thing about this is that your UI can make it impossible for users to create "invalid" programs. Instead of vomiting error messages on them, you can proactively disable buttons or provide default values to ensure that the thing they've created is valid at all points in time.

I want to stress how important error-handling is. As programmers, we tend to view human error as a shameful personality flaw that we strive to eliminate in ourselves.

To make a system that users enjoy, you have to embrace their humanity, *including their fallibility*. Making mistakes is what people do, and is a fundamental part of the creative process. Handling them gracefully with features like undo helps your users be more creative and create better work.

This spares you from designing a grammar and writing a parser for a little language. But, I know, some of you find UI programming equally unpleasant. Well, in that case, I don't

have any good news for you.

Ultimately, this pattern is about expressing behavior in a user-friendly, high-level way. You have to craft the user experience. To execute the behavior efficiently, you then need to translate that into a lower-level form. It is real work, but if you're up to the challenge, it can pay off.

Design Decisions

I tried to keep this chapter as simple as I could, but what we're really doing is creating a language. That's a pretty open-ended design space. Exploring it can be tons of fun, so make sure you don't forget to finish your game.

> Since this is the longest chapter in the book, it seems I failed that task.

How do instructions access the stack?

Bytecode VMs come in two main flavors: stack-based and register-based. In a stack-based VM, instructions always work from the top of the stack, like in our sample code. For example, `INST_ADD` pops two values, adds them, and pushes the result.

Register-based VMs still have a stack. The only difference is that instructions can read their inputs from deeper in the stack. Instead of `INST_ADD` always *popping* its operands, it has two indexes stored in the bytecode that identify where in the stack to read the operands from.

- **With a stack-based VM:**
    - *Instructions are small.* Since each instruction implicitly finds its arguments on top of the stack, you don't need to encode any data for that. This means each instruction can be pretty small, usually a single byte.
    - *Code generation is simpler.* When you get around to writing the compiler or tool that outputs bytecode, you'll find it simpler to generate stack-based bytecode. Since each instruction implicitly works from the top of the stack, you just need to output instructions in the right order to pass parameters between them.
    - *You have more instructions.* Each instruction only sees the very top of the stack. This means that to generate code for something like `a = b + c`, you need separate instructions to move `b` and `c` to the top of the stack, perform the operation, then move the result into `a`.
- **With a register-based VM:**
    - *Instructions are larger.* Since instructions need arguments for stack offsets, a single instruction needs more bits. For example, an instruction in Lua—probably

the most well-known register-based VM—is a full 32-bits. It uses 6 bits for the instruction type, and the rest are arguments.

> The Lua folks don't specify Lua's bytecode format, and it changes from version to version. What I'm describing here is true as of Lua 5.1. For an absolutely amazing deep dive into Lua's internals, read this.

- *You have fewer instructions.* Since each instruction can do more work, you don't need as many of them. Some say you get a performance improvement since you don't have to shuffle values around in the stack as much.

So which should you do? My recommendation is to stick with a stack-based VM. They're simpler to implement and much simpler to generate code for. Register-based VMs got a reputation for being a bit faster after Lua converted to that style, but it depends *deeply* on your actual instructions and on lots of other details of your VM.

What instructions do you have?

Your instruction set defines the boundaries of what can and cannot be expressed in bytecode, and it also has a big impact on the performance of your VM. Here's a laundry list of the different kinds of instructions you may want:

- **External primitives.** These are the ones that reach out of the VM into the rest of the game engine and do stuff that the user can see. They control what kinds of real behavior can be expressed in bytecode. Without these, your VM can't do anything more than burn CPU cycles.

- **Internal primitives.** These manipulate values inside the VM—things like literals, arithmetic, comparison operators, and instructions that juggle the stack around.

- **Control flow.** Our example didn't cover these, but when you want behavior that's imperative and conditionally executes instructions or loops and executes instructions more than once, you need control flow. In the low-level language of bytecode, they're surprisingly simple: jumps.

  In our instruction loop, we had an index to track where we were in the bytecode. All a jump instruction does is modify that variable and change where we're currently executing. In other words, it's a `goto`. You can build all kinds of higher-level control flow using that.

- **Abstraction.** If your users start defining a *lot* of stuff in data, eventually they'll want to start reusing bits of bytecode instead of having to copy and paste it. You may want something like callable procedures.

  In their simplest form, procedures aren't much more complex than a jump. The only

difference is that the VM maintains a second *return* stack. When it executes a "call" instruction, it pushes the current instruction index onto the return stack and then jumps to the called bytecode. When it hits a "return", the VM pops the index from the return stack and jumps back to it.

How are values represented?

Our sample VM only works with one kind of value, integers. That makes answering this easy — the stack is just a stack of `int`s. A more full-featured VM will support different data types: strings, objects, lists, etc. You'll have to decide how those are stored internally.

- **A single datatype:**
    - *It's simple.* You don't have to worry about tagging, conversions, or type-checking.
    - *You can't work with different data types.* This is the obvious downside. Cramming different types into a single representation—think storing numbers as strings—is asking for pain.
- **A tagged variant:**

    This is the common representation for dynamically typed languages. Every value has two pieces. The first is a type tag—an `enum`—that identifies what data type is being stored. The rest of the bits are then interpreted appropriately according to that type, like:

    ```
    enum ValueType
    {
      TYPE_INT,
      TYPE_DOUBLE,
      TYPE_STRING
    };

    struct Value
    {
      ValueType type;
      union
      {
        int    intValue;
        double doubleValue;
        char*  stringValue;
      };
    };
    ```

    - *Values know their type.* The nice thing about this representation is that you can check the type of a value at runtime. That's important for dynamic dispatch and for ensuring that you don't try to perform operations on types that don't support it.
    - *It takes more memory.* Every value has to carry around a few extra bits with it to

identify its type. In something as low-level as a VM, a few bits here and there add up quickly.

- **An untagged union:**

  This uses a union like the previous form, but it does *not* have a type tag that goes along with it. You have a little blob of bits that could represent more than one type, and it's up to you to ensure you don't misinterpret them.

  This is how statically typed languages represent things in memory. Since the type system ensures at compile time that you aren't misinterpreting values, you don't need to validate it at runtime.

  > This is also how *untyped* languages like assembly and Forth store values. Those languages leave it to the *user* to make sure they don't write code that misinterprets a value's type. Not for the faint of heart!

  - *It's compact.* You can't get any more efficient than storing just the bits you need for the value itself.
  - *It's fast.* Not having type tags implies you're not spending cycles checking them at runtime either. This is one of the reasons statically typed languages tend to be faster than dynamic ones.
  - *It's unsafe.* This is the real cost, of course. A bad chunk of bytecode that causes you to misinterpret a value and treat a number like a pointer or vice versa can violate the security of your game or make it crash.

    > If your bytecode was compiled from a statically typed language, you might think you're safe here because the compiler won't generate unsafe bytecode. That may be true, but remember that malicious users may hand-craft evil bytecode without going through your compiler.
    >
    > That's why, for example, the Java Virtual Machine has to do *bytecode verification* when it loads a program.

- **An interface:**

  The object-oriented solution for a value that maybe be one of several different types is through polymorphism. An interface provides virtual methods for the various type tests and conversions, along the lines of:

```
class Value
{
public:
  virtual ~Value() {}
```

```
  virtual ValueType type() = 0;

  virtual int asInt() {
    // Can only call this on ints.
    assert(false);
    return 0;
  }

  // Other conversion methods...
};
```

Then you have concrete classes for each specific data type, like:

```
class IntValue : public Value
{
public:
  IntValue(int value)
  : value_(value)
  {}

  virtual ValueType type() { return TYPE_INT; }
  virtual int asInt() { return value_; }

private:
  int value_;
};
```

- *It's open-ended.* You can define new value types outside of the core VM as long as they implement the base interface.
- *It's object-oriented.* If you adhere to OOP principles, this does things the "right" way and uses polymorphic dispatch for type-specific behavior instead of something like switching on a type tag.
- *It's verbose.* You have to define a separate class with all of the associated ceremonial verbiage for each data type. Note that in the previous examples, we showed the entire definition of *all* of the value types. Here, we only cover one!
- *It's inefficient.* To get polymorphism, you have to go through a pointer, which means even tiny values like Booleans and numbers get wrapped in objects that are allocated on the heap. Every time you touch a value, you have to do a virtual method call.

  In something like the core of a virtual machine, small performance hits like this quickly add up. In fact, this suffers from many of the problems that caused us to avoid the Interpreter pattern, except now the problem is in our *values* instead of our *code*.

My recommendation is that if you can stick with a single data type, do that. Otherwise, do a tagged union. That's what almost every language interpreter in the world does.

How is the bytecode generated?

I saved the most important question for last. I've walked you through the code to *consume* and *interpret* bytecode, but it's up to you to build something to *produce* it. The typical solution here is to write a compiler, but it's not the only option.

- **If you define a text-based language:**
  - *You have to define a syntax.* Both amateur and professional language designers categorically underestimate how difficult this is to do. Defining a grammar that makes parsers happy is easy. Defining one that makes *users* happy is *hard*.

    Syntax design is user interface design, and that process doesn't get easier when you constrain the user interface to a string of characters.

  - *You have to implement a parser.* Despite their reputation, this part is pretty easy. Either use a parser generator like ANTLR or Bison, or — like I do—hand-roll a little recursive descent one, and you're good to go.

  - *You have to handle syntax errors.* This is one of the most important and most difficult parts of the process. When users make syntax and semantic errors—which they will, constantly—it's your job to guide them back onto the right path. Giving helpful feedback isn't easy when all you know is that your parser is sitting on some unexpected punctuation.

  - *It will likely turn off non-technical users.* We programmers like text files. Combined with powerful command-line tools, we think of them as the LEGO blocks of computing—simple, but easily composable in a million ways.

    Most non-programmers don't think of plaintext like that. To them, text files feel like filling in tax forms for an angry robotic auditor that yells at them if they forget a single semicolon.

- **If you define a graphical authoring tool:**
  - *You have to implement a user interface.* Buttons, clicks, drags, stuff like that. Some cringe at the idea of this, but I personally love it. If you go down this route, it's important to treat designing the user interface as a core part of doing your job well—not just an unpleasant task to be muddled through.

    Every little bit of extra work you do here will make your tool easier and more pleasant to use, and that directly leads to better content in your game. If you look behind many of the games you love, you'll often find the secret was fun authoring tools.

- *You have fewer error cases.* Because the user is building behavior interactively one step at a time, your application can guide them away from mistakes as soon as they happen.

  With a text-based language, the tool doesn't see *any* of the user's content until they throw an entire file at it. That makes it harder to prevent and handle errors.

- *Portability is harder.* The nice thing about text compilers is that text files are universal. A simple compiler just reads in one file and writes one out. Porting that across operating systems is trivial.

  > Except for line endings. And encodings.

  When you're building a UI, you have to choose which framework to use, and many of those are specific to one OS. There are cross-platform UI toolkits too, but those often get ubiquity at the expense of familiarity—they feel equally foreign on all of platforms.

See Also

- This pattern's close sister is the Gang of Four's Interpreter [GoF] pattern. Both give you a way to express composable behavior in terms of data.

  In fact, you'll often end up using *both* patterns. The tool you use to generate bytecode will have an internal tree of objects that represents the code. This is exactly what the Interpreter pattern expects.

  In order to compile that to bytecode, you'll recursively walk the tree, just like you do to interpret it with the Interpreter pattern. The *only* difference is that instead of executing a primitive piece of behavior immediately, you output the bytecode instruction to perform that later.

- The Lua programming language is the most widely used scripting language in games. It's implemented internally as a very compact register-based bytecode VM.

- Kismet is a graphical scripting tool built into UnrealEd, the editor for the Unreal engine.

- My own little scripting language, Wren, is a simple stack-based bytecode interpreter.

Subclass Sandbox

Intent

Define behavior in a subclass using a set of operations provided by its base class.

Motivation

Every kid has dreamed of being a superhero, but unfortunately, cosmic rays are in short supply here on Earth. Games that let you pretend to be a superhero are the closest approximation. Because our game designers have never learned to say, "no", *our* superhero game aims to feature dozens, if not hundreds, of different superpowers that heroes may choose from.

Our plan is that we'll have a `Superpower` base class. Then, we'll have a derived class that implements each superpower. We'll divvy up the design doc among our team of programmers and get coding. When we're done, we'll have a hundred superpower classes.

> When you find yourself with a *lot* of subclasses, like in this example, that often means a data-driven approach is better. Instead of lots of *code* for defining different powers, try finding a way to define that behavior in *data* instead.
>
> Patterns like Type Object, Bytecode, and Interpreter [GoF] can all help.

We want to immerse our players in a world teeming with variety. Whatever power they dreamed up when they were a kid, we want in our game. That means these superpower subclasses will be able to do just about everything: play sounds, spawn visual effects, interact with AI, create and destroy other game entities, and mess with physics. There's no corner of the codebase that they won't touch.

Let's say we unleash our team and get them writing superpower classes. What's going to happen?

- *There will be lots of redundant code.* While the different powers will be wildly varied, we can still expect plenty of overlap. Many of them will spawn visual effects and play sounds in the same way. A freeze ray, heat ray, and Dijon mustard ray are all pretty similar when you get down to it. If the people implementing those don't coordinate, there's going to be a lot of duplicate code and effort.

- *Every part of the game engine will get coupled to these classes.* Without knowing better, people will write code that calls into subsystems that were never meant to be tied directly to the superpower classes. If our renderer is organized into several nice neat layers, only one of which is intended to be used by code outside of the graphics engine, we can bet that we'll end up with superpower code that pokes into every one of them.

- *When these outside systems need to change, odds are good some random superpower code will get broken.* Once we have different superpower classes coupling themselves to various and sundry parts of the game engine, it's inevitable that changes to those systems will impact the power classes. That's no fun because your graphics, audio, and UI programmers probably don't want to also have to be gameplay programmers *too*.

- *It's hard to define invariants that all superpowers obey.* Let's say we want to make sure that all audio played by our powers gets properly queued and prioritized. There's no easy way to do that if our hundred classes are all directly calling into the sound engine on their own.

What we want is to give each of the gameplay programmers who is implementing a superpower a set of primitives they can play with. You want your power to play a sound? Here's your `playSound()` function. You want particles? Here's `spawnParticles()`. We'll make sure these operations cover everything you need to do so that you don't need to `#include` random headers and nose your way into the rest of the codebase.

We do this by making these operations *protected methods of the* `Superpower` *base class*. Putting them in the base class gives every power subclass direct, easy access to the methods. Making them protected (and likely non-virtual) communicates that they exist specifically to be *called* by subclasses.

Once we have these toys to play with, we need a place to use them. For that, we'll define a *sandbox method*, an abstract protected method that subclasses must implement. Given those, to implement a new kind of power, you:

1. Create a new class that inherits from `Superpower`.

2. Override `activate()`, the sandbox method.

3. Implement the body of that by calling the protected methods that `Superpower` provides.

We can fix our redundant code problem now by making those provided operations as

high-level as possible. When we see code that's duplicated between lots of the subclasses, we can always roll it up into `Superpower` as a new operation that they can all use.

We've addressed our coupling problem by constraining the coupling to one place. `Superpower` itself will end up coupled to the different game systems, but our hundred derived classes will not. Instead, they are *only* coupled to their base class. When one of those game systems changes, modification to `Superpower` may be necessary, but dozens of subclasses shouldn't have to be touched.

This pattern leads to an architecture where you have a shallow but wide class hierarchy. Your inheritance chains aren't *deep*, but there are a *lot* of classes that hang off `Superpower`. By having a single class with a lot of direct subclasses, we have a point of leverage in our codebase. Time and love that we put into `Superpower` can benefit a wide set of classes in the game.

> Lately, you find a lot of people criticizing inheritance in object-oriented languages. Inheritance *is* problematic—there's really no deeper coupling in a codebase than the one between a base class and its subclass—but I find *wide* inheritance trees to be easier to work with than *deep* ones.

The Pattern

A **base class** defines an abstract **sandbox method** and several **provided operations**. Marking them protected makes it clear that they are for use by derived classes. Each derived **sandboxed subclass** implements the sandbox method using the provided operations.

When to Use It

The Subclass Sandbox pattern is a very simple, common pattern lurking in lots of codebases, even outside of games. If you have a non-virtual protected method laying around, you're probably already using something like this. Subclass Sandbox is a good fit when:

- You have a base class with a number of derived classes.
- The base class is able to provide all of the operations that a derived class may need to perform.
- There is behavioral overlap in the subclasses and you want to make it easier to share code between them.
- You want to minimize coupling between those derived classes and the rest of the program.

Keep in Mind

"Inheritance" is a bad word in many programming circles these days, and one reason is that base classes tend to accrete more and more code. This pattern is particularly susceptible to that.

Since subclasses go through their base class to reach the rest of the game, the base class ends up coupled to every system *any* derived class needs to talk to. Of course, the subclasses are also intimately tied to their base class. That spiderweb of coupling makes it very hard to change the base class without breaking something—you've got the brittle base class problem.

The flip side of the coin is that since most of your coupling has been pushed up to the base class, the derived classes are now much more cleanly separated from the rest of the world. Ideally, most of your behavior will be in those subclasses. That means much of your codebase is isolated and easier to maintain.

Still, if you find this pattern is turning your base class into a giant bowl of code stew, consider pulling some of the provided operations out into separate classes that the base class can dole out responsibility to. The Component pattern can help here.

Sample Code

Because this is such a simple pattern, there isn't much to the sample code. That doesn't mean it isn't useful—the pattern is about the *intent*, not the complexity of its implementation.

We'll start with our `Superpower` base class:

```
class Superpower
{
public:
  virtual ~Superpower() {}

protected:
  virtual void activate() = 0;

  void move(double x, double y, double z)
  {
    // Code here...
  }

  void playSound(SoundId sound, double volume)
  {
    // Code here...
  }
```

```
  void spawnParticles(ParticleType type, int count)
  {
    // Code here...
  }
};
```

The `activate()` method is the sandbox method. Since it is virtual and abstract, subclasses *must* override it. This makes it clear to someone creating a power subclass where their work has to go.

The other protected methods, `move()`, `playSound()`, and `spawnParticles()`, are the provided operations. These are what the subclasses will call in their implementation of `activate()`.

We didn't implement the provided operations in this example, but an actual game would have real code there. Those methods are where `Superpower` gets coupled to other systems in the game—`move()` may call into physics code, `playSound()` will talk to the audio engine, etc. Since this is all in the *implementation* of the base class, it keeps that coupling encapsulated within `Superpower` itself.

OK, now let's get our radioactive spiders out and create a power. Here's one:

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    // Spring into the air.
    playSound(SOUND_SPROING, 1.0f);
    spawnParticles(PARTICLE_DUST, 10);
    move(0, 0, 20);
  }
};
```

> OK, maybe being able to *jump* isn't all that *super*, but I'm trying to keep things basic here.

This power springs the superhero into the air, playing an appropriate sound and kicking up a little cloud of dust. If all of the superpowers were this simple — just a combination of sound, particle effect, and motion—then we wouldn't need this pattern at all. Instead, `Superpower` could have a baked-in implementation of `activate()` that accesses fields for the sound ID, particle type, and movement. But that only works when every power essentially works the same way with only some differences in data. Let's elaborate on it a bit:

```
class Superpower
```

```
{
protected:
  double getHeroX()
  {
    // Code here...
  }

  double getHeroY()
  {
    // Code here...
  }

  double getHeroZ()
  {
    // Code here...
  }

  // Existing stuff...
};
```

Here, we've added a couple of methods to get the hero's position. Our `SkyLaunch` subclass can now use those:

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    if (getHeroZ() == 0)
    {
      // On the ground, so spring into the air.
      playSound(SOUND_SPROING, 1.0f);
      spawnParticles(PARTICLE_DUST, 10);
      move(0, 0, 20);
    }
    else if (getHeroZ() < 10.0f)
    {
      // Near the ground, so do a double jump.
      playSound(SOUND_SWOOP, 1.0f);
      move(0, 0, getHeroZ() + 20);
    }
    else
    {
      // Way up in the air, so do a dive attack.
      playSound(SOUND_DIVE, 0.7f);
      spawnParticles(PARTICLE_SPARKLES, 1);
      move(0, 0, -getHeroZ());
    }
  }
};
```

Since we have access to some state, now our sandbox method can do actual, interesting control flow. Here, it's still just a couple of simple `if` statements, but you can do anything you want. By having the sandbox method be an actual full-fledged method that contains arbitrary code, the sky's the limit.

> Earlier, I suggested a data-driven approach for powers. This is one reason why you may decide *not* to do that. If your behavior is complex and imperative, it is more difficult to define in data.

Design Decisions

As you can see, Subclass Sandbox is a fairly "soft" pattern. It describes a basic idea, but it doesn't have a lot of detailed mechanics. That means you'll be making some interesting choices each time you apply it. Here are some questions to consider.

What operations should be provided?

This is the biggest question. It deeply affects how this pattern feels and how well it works. At the minimal end of the spectrum, the base class doesn't provide *any* operations. It just has a sandbox method. To implement it, you'll have to call into systems outside of the base class. If you take that angle, it's probably not even fair to say you're using this pattern.

On the other end of the spectrum, the base class provides *every* operation that a subclass may need. Subclasses are *only* coupled to the base class and don't call into any outside systems whatsoever.

> Concretely, this means each source file for a subclass would only need a single `#include`—the one for its base class.

Between these two points, there's a wide middle ground where some operations are provided by the base class and others are accessed directly from the outside system that defines it. The more operations you provide, the less coupled subclasses are to outside systems, but the *more* coupled the base class is. It removes coupling from the derived classes, but it does so by pushing that up to the base class itself.

That's a win if you have a bunch of derived classes that were all coupled to some outside system. By moving the coupling up into a provided operation, you've centralized it into one place: the base class. But the more you do this, the bigger and harder to maintain that one class becomes.

So where should you draw the line? Here are a few rules of thumb:

- If a provided operation is only used by one or a few subclasses, you don't get a lot of bang for your buck. You're adding complexity to the base class, which affects everyone,

but only a couple of classes benefit.

This may be worth it to make the operation consistent with other provided operations, or it may be simpler and cleaner to let those special case subclasses call out to the external systems directly.

- When you call a method in some other corner of the game, it's less intrusive if that method doesn't modify any state. It still creates a coupling, but it's a "safe" coupling because it can't break anything in the game.

  > "Safe" is in quotes here because technically, even just accessing data can cause problems. If your game is multi-threaded, you could read something at the same time that it's being modified. If you aren't careful, you could end up with bogus data.
  >
  > Another nasty case is if your game state is strictly deterministic (which many online games are in order to keep players in sync). If you access something outside of the set of synchronized game state, you can cause incredibly painful non-determinism bugs.

  Calls that do modify state, on the other hand, more deeply tie you to those parts of the codebase, and you need to be much more cognizant of that. That makes them good candidates for being rolled up into provided operations in the more visible base class.

- If the implementation of a provided operation only forwards a call to some outside system, then it isn't adding much value. In that case, it may be simpler to call the outside method directly.

  However, even simple forwarding can still be useful—those methods often access state that the base class doesn't want to directly expose to subclasses. For example, let's say `Superpower` provided this:

  ```
  void playSound(SoundId sound, double volume)
  {
    soundEngine_.play(sound, volume);
  }
  ```

  It's just forwarding the call to some `soundEngine_` field in `Superpower`. The advantage, though, is that it keeps that field encapsulated in `Superpower` so subclasses can't poke at it.

Should methods be provided directly, or through objects that contain them?

The challenge with this pattern is that you can end up with a painfully large number of methods crammed into your base class. You can mitigate that by moving some of those methods over to other classes. The provided operations in the base class then just return

one of those objects.

For example, to let a power play sounds, we could add these directly to `Superpower`:

```
class Superpower
{
protected:
  void playSound(SoundId sound, double volume)
  {
    // Code here...
  }

  void stopSound(SoundId sound)
  {
    // Code here...
  }

  void setVolume(SoundId sound)
  {
    // Code here...
  }

  // Sandbox method and other operations...
};
```

But if `Superpower` is already getting large and unwieldy, we might want to avoid that. Instead, we create a `SoundPlayer` class that exposes that functionality:

```
class SoundPlayer
{
  void playSound(SoundId sound, double volume)
  {
    // Code here...
  }

  void stopSound(SoundId sound)
  {
    // Code here...
  }

  void setVolume(SoundId sound)
  {
    // Code here...
  }
};
```

Then `Superpower` provides access to it:

```
class Superpower
```

```
{
protected:
  SoundPlayer& getSoundPlayer()
  {
    return soundPlayer_;
  }

  // Sandbox method and other operations...

private:
  SoundPlayer soundPlayer_;
};
```

Shunting provided operations into auxiliary classes like this can do a few things for you:

- *It reduces the number of methods in the base class.* In the example here, we went from three methods to just a single getter.
- *Code in the helper class is usually easier to maintain.* Core base classes like `Superpower`, despite our best intentions, tend to be tricky to change since so much depends on them. By moving functionality over to a less coupled secondary class, we make that code easier to poke at without breaking things.
- *It lowers the coupling between the base class and other systems.* When `playSound()` was a method directly on `Superpower`, our base class was directly tied to `SoundId` and whatever audio code the implementation called into. Moving that over to `SoundPlayer` reduces `Superpower`'s coupling to the single `SoundPlayer` class, which then encapsulates all of its other dependencies.

How does the base class get the state that it needs?

Your base class will often need some data that it wants to encapsulate and keep hidden from its subclasses. In our first example, the `Superpower` class provided a `spawnParticles()` method. If the implementation of that needs some particle system object, how would it get one?

- **Pass it to the base class constructor:**

  The simplest solution is to have the base class take it as a constructor argument:

  ```
  class Superpower
  {
  public:
    Superpower(ParticleSystem* particles)
    : particles_(particles)
    {}

    // Sandbox method and other operations...
  ```

```
private:
  ParticleSystem* particles_;
};
```

This safely ensures that every superpower does have a particle system by the time it's constructed. But let's look at a derived class:

```
class SkyLaunch : public Superpower
{
public:
  SkyLaunch(ParticleSystem* particles)
  : Superpower(particles)
  {}
};
```

Here we see the problem. Every derived class will need to have a constructor that calls the base class one and passes along that argument. That exposes every derived class to a piece of state that we don't want them to know about.

This is also a maintenance headache. If we later add another piece of state to the base class, every constructor in each of our derived classes will have to be modified to pass it along.

- **Do two-stage initialization:**

  To avoid passing everything through the constructor, we can split initialization into two steps. The constructor will take no parameters and just create the object. Then, we call a separate method defined directly on the base class to pass in the rest of the data that it needs:

  ```
  Superpower* power = new SkyLaunch();
  power->init(particles);
  ```

  Note here that since we aren't passing anything into the constructor for `SkyLaunch`, it isn't coupled to anything we want to keep private in `Superpower`. The trouble with this approach, though, is that you have to make sure you always remember to call `init()`. If you ever forget, you'll have a power that's in some twilight half-created state and won't work.

  You can fix that by encapsulating the entire process into a single function, like so:

  ```
  Superpower* createSkyLaunch(ParticleSystem* particles)
  {
    Superpower* power = new SkyLaunch();
    power->init(particles);
    return power;
  }
  ```

> With a little trickery like private constructors and friend classes, you can ensure this `createSkylaunch()` function is the *only* function that can actually create powers. That way, you can't forget any of the initialization stages.

- **Make the state static:**

  In the previous example, we were initializing each `Superpower` *instance* with a particle system. That makes sense when every power needs its own unique state. But let's say that the particle system is a Singleton, and every power will be sharing the same state.

  In that case, we can make the state private to the base class and also make it *static*. The game will still have to make sure that it initializes the state, but it only has to initialize the `Superpower` *class* once for the entire game, and not each instance.

  > Keep in mind that this still has many of the problems of a singleton. You've got some state shared between lots and lots of objects (all of the `Superpower` instances). The particle system is encapsulated, so it isn't globally *visible*, which is good, but it can still make reasoning about powers harder because they can all poke at the same object.

  ```
  class Superpower
  {
  public:
    static void init(ParticleSystem* particles)
    {
      particles_ = particles;
    }

    // Sandbox method and other operations...

  private:
    static ParticleSystem* particles_;
  };
  ```

  Note here that `init()` and `particles_` are both static. As long as the game calls `Superpower::init()` once early on, every power can access the particle system. At the same time, `Superpower` instances can be created freely by calling the right derived class's constructor.

  Even better, now that `particles_` is a *static* variable, we don't have to store it for each instance of `Superpower`, so we've made the class use less memory.

- **Use a service locator:**

The previous option requires that outside code specifically remembers to push in the state that the base class needs before it needs it. That places the burden of initialization on the surrounding code. Another option is to let the base class handle it by pulling in the state it needs. One way to do that is by using the Service Locator pattern:

```
class Superpower
{
protected:
  void spawnParticles(ParticleType type, int count)
  {
    ParticleSystem& particles = Locator::getParticles();
    particles.spawn(type, count);
  }

  // Sandbox method and other operations...
};
```

Here, `spawnParticles()` needs a particle system. Instead of being *given* one by outside code, it fetches one itself from the service locator.

See Also

- When you apply the Update Method pattern, your update method will often also be a sandbox method.
- This pattern is a role reversal of the Template Method GoF pattern. In both patterns, you implement a method using a set of primitive operations. With Subclass Sandbox, the method is in the derived class and the primitive operations are in the base class. With Template Method, the *base* class has the method and the primitive operations are implemented by the *derived* class.
- You can also consider this a variation on the Facade GoF pattern. That pattern hides a number of different systems behind a single simplified API. With Subclass Sandbox, the base class acts as a facade thats hides the entire game engine from the subclasses.

Type Object

Intent

Allow the flexible creation of new "classes" by creating a single class, each instance of which represents a different type of object.

Motivation

Imagine we're working on a fantasy role-playing game. Our task is to write the code for the hordes of vicious monsters that seek to slay our brave hero. Monsters have a bunch of different attributes: health, attacks, graphics, sounds, etc., but for example purposes we'll just worry about the first two.

Each monster in the game has a value for its current health. It starts out full, and each time the monster is wounded, it diminishes. Monsters also have an attack string. When the monster attacks our hero, that text will be shown to the user somehow. (We don't care how here.)

The designers tell us that monsters come in a variety of different *breeds*, like "dragon" or "troll". Each breed describes a *kind* of monster that exists in the game, and there can be multiple monsters of the same breed running around in the dungeon at the same time.

The breed determines a monster's starting health—dragons start off with more than trolls, making them harder to kill. It also determines the attack string — all monsters of the same breed attack the same way.

The typical OOP answer

With that game design in mind, we fire up our text editor and start coding. According to the design, a dragon is a kind of monster, a troll is another kind, and so on with the other breeds. Thinking object-oriented, that leads us to a `Monster` base class:

This is the so-called "is-a" relationship. In conventional OOP thinking,

since a dragon "is-a" monster, we model that by making `Dragon` a subclass of `Monster`. As we'll see, subclassing is only one way of enshrining a conceptual relation like that into code.

```
class Monster
{
public:
  virtual ~Monster() {}
  virtual const char* getAttack() = 0;

protected:
  Monster(int startingHealth)
  : health_(startingHealth)
  {}

private:
  int health_; // Current health.
};
```

The public `getAttack()` function lets the combat code get the string that should be displayed when the monster attacks the hero. Each derived breed class will override this to provide a different message.

The constructor is protected and takes the starting health for the monster. We'll have derived classes for each breed that provide their own public constructors that call this one, passing in the starting health that is appropriate for that breed.

Now let's see a couple of breed subclasses:

```
class Dragon : public Monster
{
public:
  Dragon() : Monster(230) {}

  virtual const char* getAttack()
  {
    return "The dragon breathes fire!";
  }
};

class Troll : public Monster
{
public:
  Troll() : Monster(48) {}

  virtual const char* getAttack()
  {
    return "The troll clubs you!";
  }
```

```
};
```

```
Exclamation points make everything more exciting!
```

Each class derived from `Monster` passes in the starting health and overrides `getAttack()` to return the attack string for that breed. Everything works as expected, and before long, we've got our hero running around slaying a variety of beasties. We keep slinging code, and before we know it, we've got dozens of monster subclasses, from acidic slimes to zombie goats.

Then, strangely, things start to bog down. Our designers ultimately want to have *hundreds* of breeds, and we find ourselves spending all of our time writing these little seven-line subclasses and recompiling. It gets worse—the designers want to start tuning the breeds we've already coded. Our formerly productive workday degenerates to:

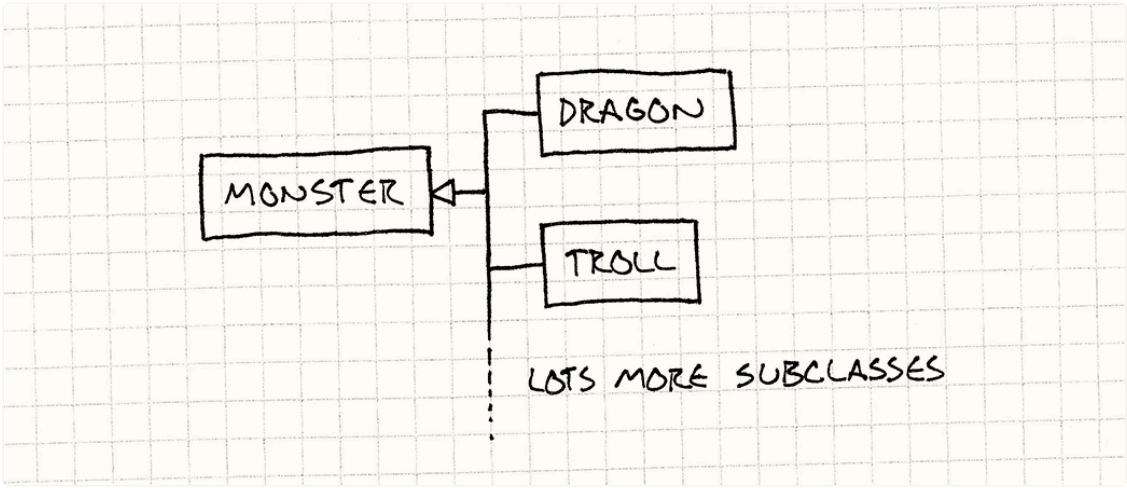
1. Get email from designer asking to change health of troll from 48 to 52.
2. Check out and change `Troll.h`.
3. Recompile game.
4. Check in change.
5. Reply to email.
6. Repeat.

We spend the day frustrated because we've turned into data monkeys. Our designers are frustrated because it takes them forever to get a simple number tuned. What we need is the ability to change breed stats without having to recompile the whole game every time. Even better, we'd like designers to be able to create and tune breeds without *any* programmer intervention at all.

A class for a class

At a very high level, the problem we're trying to solve is pretty simple. We have a bunch of different monsters in the game, and we want to share certain attributes between them. A horde of monsters are beating on the hero, and we want some of them to use the same text for their attack. We define that by saying that all of those monsters are the same "breed", and that the breed determines the attack string.

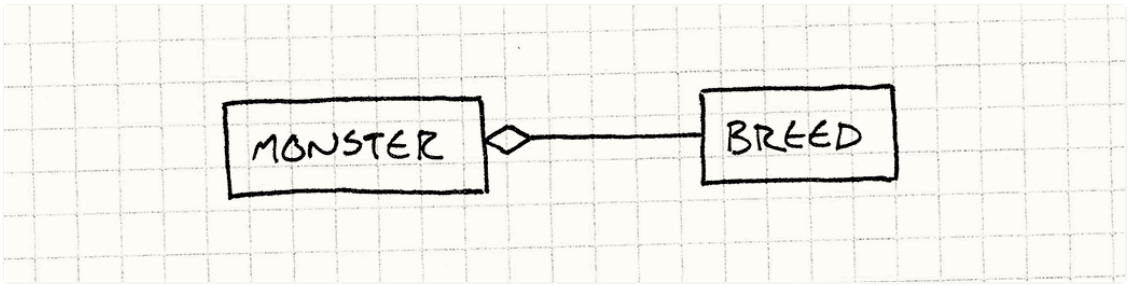
We decided to implement this concept using inheritance since it lines up with our intuition of classes. A dragon is a monster, and each dragon in the game is an instance of this dragon "class". Defining each breed as a subclass of an abstract base `Monster` class, and having each monster in the game be an instance of that derived breed class mirrors that. We end up with a class hierarchy like this:



Here, the ◁— means "inherits from".

Each instance of a monster in the game will be of one of the derived monster types. The more breeds we have, the bigger the class hierarchy. That's the problem of course: adding new breeds means adding new code, and each breed has to be compiled in as its own type.

This works, but it isn't the only option. We could also architect our code so that each monster *has* a breed. Instead of subclassing `Monster` for each breed, we have a single `Monster` class and a single `Breed` class:



Here, the ◇— means "is referenced by".

That's it. Two classes. Notice that there's no inheritance at all. With this system, each monster in the game is simply an instance of class `Monster`. The `Breed` class contains the information that's shared between all monsters of the same breed: starting health and the attack string.

To associate monsters with breeds, we give each `Monster` instance a reference to a `Breed` object containing the information for that breed. To get the attack string, a monster just calls a method on its breed. The `Breed` class essentially defines a monster's "type". Each breed instance is an *object* that represents a different conceptual *type*, hence the name of the pattern: Type Object.

What's especially powerful about this pattern is that now we can define new *types* of things without complicating the codebase at all. We've essentially lifted a portion of the type system out of the hard-coded class hierarchy into data we can define at runtime.

We can create hundreds of different breeds by instantiating more instances of `Breed` with different values. If we create breeds by initializing them from data read from some configuration file, we have the ability to define new types of monsters completely in data. So easy, a designer could do it!

The Pattern

Define a **type object** class and a **typed object** class. Each type object instance represents a different logical type. Each typed object stores a **reference to the type object that describes its type**.

Instance-specific data is stored in the typed object instance, and data or behavior that should be shared across all instances of the same conceptual type is stored in the type object. Objects referencing the same type object will function as if they were the same type. This lets us share data and behavior across a set of similar objects, much like subclassing lets us do, but without having a fixed set of hard-coded subclasses.

When to Use It

This pattern is useful anytime you need to define a variety of different "kinds" of things, but baking the kinds into your language's type system is too rigid. In particular, it's useful when either of these is true:

- You don't know what types you will need up front. (For example, what if our game needed to support downloading content that contained new breeds of monsters?)
- You want to be able to modify or add new types without having to recompile or change code.

Keep in Mind

This pattern is about moving the definition of a "type" from the imperative but rigid language of code into the more flexible but less behavioral world of objects in memory. The flexibility is good, but you lose some things by hoisting your types into data.

The type objects have to be tracked manually

One advantage of using something like C++'s type system is that the compiler handles all of the bookkeeping for the classes automatically. The data that defines each class is automatically compiled into the static memory segment of the executable and just works.

With the Type Object pattern, we are now responsible for managing not only our monsters in memory, but also their *types*—we have to make sure all of the breed objects are instantiated and kept in memory as long as our monsters need them. Whenever we create

a new monster, it's up to us to ensure that it's correctly initialized with a reference to a valid breed.

We've freed ourselves from some of the limitations of the compiler, but the cost is that we have to re-implement some of what it used to be doing for us.

> Under the hood, C++ virtual methods are implemented using something called a "virtual function table", or just "vtable". A vtable is a simple `struct` containing a set of function pointers, one for each virtual method in a class. There is one vtable in memory for each class. Each instance of a class has a pointer to the vtable for its class.
>
> When you call a virtual function, the code first looks up the vtable for the object, then it calls the function stored in the appropriate function pointer in the table.
>
> Sound familiar? The vtable is our breed object, and the pointer to the vtable is the reference the monster holds to its breed. C++ classes are the Type Object pattern applied to C, handled automatically by the compiler.

It's harder to define *behavior* for each type

With subclassing, you can override a method and do whatever you want to — calculate values procedurally, call other code, etc. The sky is the limit. We could define a monster subclass whose attack string changed based on the phase of the moon if we wanted to. (Handy for werewolves, I suppose.)

When we use the Type Object pattern instead, we replace an overridden method with a member variable. Instead of having monster subclasses that override a method to *calculate* an attack string using different *code*, we have a breed object that *stores* an attack string in a different *variable*.

This makes it very easy to use type objects to define type-specific *data*, but hard to define type-specific *behavior*. If, for example, different breeds of monster needed to use different AI algorithms, using this pattern becomes more challenging.

There are a couple of ways we can get around this limitation. A simple solution is to have a fixed set of pre-defined behaviors and then use data in the type object to simply *select* one of them. For example, let's say our monster AI will always be either "stand still", "chase hero", or "whimper and cower in fear" (hey, they can't all be mighty dragons). We can define functions to implement each of those behaviors. Then, we can associate an AI algorithm with a breed by having it store a pointer to the appropriate function.

> Sound familiar again? Now we're back to really implementing vtables in *our* type objects.

Another more powerful solution is to actually support defining behavior completely in data. The Interpreter [GoF] and Bytecode patterns both let us build objects that represent behavior. If we read in a data file and use that to create a data structure for one of these patterns, we've moved the behavior's definition completely out of code and into content.

> Over time, games are getting more data-driven. Hardware gets more powerful, and we find ourselves limited more by how much content we can author than how hard we can push the hardware. With a 64K cartridge, the challenge was *cramming* the gameplay into it. With a double-sided DVD, the challenge is *filling* it with gameplay.
>
> Scripting languages and other higher-level ways of defining game behavior can give us a much needed productivity boost, at the expense of less optimal runtime performance. Since hardware keeps getting better but our brainpower doesn't, that trade-off starts to make more and more sense.

Sample Code

For our first pass at an implementation, let's start simple and build the basic system described in the motivation section. We'll start with the `Breed` class:

```
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_; // Starting health.
  const char* attack_;
};
```

Very simple. It's basically just a container for two data fields: the starting health and the attack string. Let's see how monsters use it:

```
class Monster
{
public:
  Monster(Breed& breed)
  : health_(breed.getHealth()),
```

```
    breed_(breed)
  {}

  const char* getAttack()
  {
    return breed_.getAttack();
  }

private:
  int    health_; // Current health.
  Breed& breed_;
};
```

When we construct a monster, we give it a reference to a breed object. This defines the monster's breed instead of the subclasses we were previously using. In the constructor, `Monster` uses the breed to determine its starting health. To get the attack string, the monster simply forwards the call to its breed.

This very simple chunk of code is the core idea of the pattern. Everything from here on out is bonus.

Making type objects more like types: constructors

With what we have now, we construct a monster directly and are responsible for passing in its breed. This is a bit backwards from how regular objects are instantiated in most OOP languages—we don't usually allocate a blank chunk of memory and then *give* it its class. Instead, we call a constructor function on the class itself, and it's responsible for giving us a new instance.

We can apply this same pattern to our type objects:

```
class Breed
{
public:
  Monster* newMonster() { return new Monster(*this); }

  // Previous Breed code...
};
```

> "Pattern" is the right word here. What we're talking about is one of the classic patterns from Design Patterns: Factory Method [GoF].
>
> In some languages, this pattern is applied for constructing *all* objects. In Ruby, Smalltalk, Objective-C, and other languages where classes are objects, you construct new instances by calling a method on the class object itself.

And the class that uses them:

```
class Monster
{
  friend class Breed;

public:
  const char* getAttack() { return breed_.getAttack(); }

private:
  Monster(Breed& breed)
  : health_(breed.getHealth()),
    breed_(breed)
  {}

  int health_; // Current health.
  Breed& breed_;
};
```

The key difference is the `newMonster()` function in `Breed`. That's our "constructor" factory method. With our original implementation, creating a monster looked like:

> There's another minor difference here. Because the sample code is in C++, we can use a handy little feature: *friend classes.*
>
> We've made `Monster`'s constructor private, which prevents anyone from calling it directly. Friend classes sidestep that restriction so `Breed` can still access it. This means the *only* way to create monsters is by going through `newMonster()`.

```
Monster* monster = new Monster(someBreed);
```

After our changes, it's like this:

```
Monster* monster = someBreed.newMonster();
```

So, why do this? There are two steps to creating an object: allocation and initialization. `Monster`'s constructor lets us do all of the initialization we need. In our example, that's only storing the breed, but a full game would be loading graphics, initializing the monster's AI, and doing other set-up work.

However, that all happens *after* allocation. We've already got a chunk of memory to put our monster into before its constructor is called. In games, we often want to control that aspect of object creation too: we'll typically use things like custom allocators or the Object Pool pattern to control where in memory our objects end up.

Defining a "constructor" function in `Breed` gives us a place to put that logic. Instead of simply calling `new`, the `newMonster()` function can pull the memory from a pool or

custom heap before passing control off to `Monster` for initialization. By putting this logic inside `Breed`, in the *only* function that has the ability to create monsters, we ensure that all monsters go through the memory management scheme we want.

Sharing data through inheritance

What we have so far is a perfectly serviceable type object system, but it's pretty basic. Our game will eventually have *hundreds* of different breeds, each with dozens of attributes. If a designer wants to tune all of the thirty different breeds of troll to make them a little stronger, she's got a lot of tedious data entry ahead of her.

What would help is the ability to share attributes across multiple *breeds* in the same way that breeds let us share attributes across multiple *monsters*. Just like we did with our original OOP solution, we can solve this using inheritance. Only, this time, instead of using our language's inheritance mechanism, we'll implement it ourselves within our type objects.

To keep things simple, we'll only support single inheritance. In the same way that a class can have a parent base class, we'll allow a breed to have a parent breed:

```
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_; // Starting health.
  const char* attack_;
};
```

When we construct a breed, we give it a parent that it inherits from. We can pass in `NULL` for a base breed that has no ancestors.

To make this useful, a child breed needs to control which attributes are inherited from its parent and which attributes it overrides and specifies itself. For our example system, we'll say that a breed overrides the monster's health by having a non-zero value and overrides the attack by having a non-`NULL` string. Otherwise, the attribute will be inherited from its parent.

There are two ways we can implement this. One is to handle the delegation dynamically

every time the attribute is requested, like this:

```
int Breed::getHealth()
{
  // Override.
  if (health_ != 0 || parent_ == NULL) return health_;

  // Inherit.
  return parent_->getHealth();
}

const char* Breed::getAttack()
{
  // Override.
  if (attack_ != NULL || parent_ == NULL) return attack_;

  // Inherit.
  return parent_->getAttack();
}
```

This has the advantage of doing the right thing if a breed is modified at runtime to no longer override, or no longer inherit some attribute. On the other hand, it takes a bit more memory (it has to retain a pointer to its parent), and it's slower. It has to walk the inheritance chain each time you look up an attribute.

If we can rely on a breed's attributes not changing, a faster solution is to apply the inheritance at *construction time*. This is called "copy-down" delegation because we *copy* inherited attributes *down* into the derived type when it's created. It looks like this:

```
Breed(Breed* parent, int health, const char* attack)
: health_(health),
  attack_(attack)
{
  // Inherit non-overridden attributes.
  if (parent != NULL)
  {
    if (health == 0) health_ = parent->getHealth();
    if (attack == NULL) attack_ = parent->getAttack();
  }
}
```

Note that we no longer need a field for the parent breed. Once the constructor is done, we can forget the parent since we've already copied all of its attributes in. To access a breed's attribute, now we just return the field:

```
int         getHealth() { return health_; }
const char* getAttack() { return attack_; }
```

Nice and fast!

Let's say our game engine is set up to create the breeds by loading a JSON file that defines them. It could look like:

```
{
  "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!"
  }
}
```

We'd have a chunk of code that reads each breed entry and instantiates a new breed instance with its data. As you can see from the `"parent": "Troll"` fields, the `Troll Archer` and `Troll Wizard` breeds inherit from the base `Troll` breed.

Since both of them have zero for their health, they'll inherit it from the base `Troll` breed instead. This means now our designer can tune the health in `Troll` and all three breeds will be updated. As the number of breeds and the number of different attributes each breed has increase, this can be a big time-saver. Now, with a pretty small chunk of code, we have an open-ended system that puts control in our designers' hands and makes the best use of their time. Meanwhile, we can get back to coding other features.

Design Decisions

The Type Object pattern lets us build a type system as if we were designing our own programming language. The design space is wide open, and we can do all sorts of interesting stuff.

In practice, a few things curtail our fancy. Time and maintainability will discourage us from anything particularly complicated. More importantly, whatever type object system we design, our users (often non-programmers) will need to be able to easily understand it. The simpler we can make it, the more usable it will be. So what we'll cover here is the well-trodden design space, and we'll leave the far reaches for the academics and explorers.

Is the type object encapsulated or exposed?

In our sample implementation, `Monster` has a reference to a breed, but it doesn't publicly expose it. Outside code can't get directly at the monster's breed. From the codebase's perspective, monsters are essentially typeless, and the fact that they have breeds is an implementation detail.

We can easily change this and allow `Monster` to return its `Breed`:

```
class Monster
{
public:
  Breed& getBreed() { return breed_; }

  // Existing code...
};
```

> As in other examples in this book, we're following a convention where we return objects by reference instead of pointer to indicate to users that `NULL` will never be returned.

Doing this changes the design of `Monster`. The fact that all monsters have breeds is now a publicly visible part of its API. There are benefits with either choice.

- **If the type object is encapsulated:**
  - *The complexity of the Type Object pattern is hidden from the rest of the codebase.* It becomes an implementation detail that only the typed object has to worry about.
  - *The typed object can selectively override behavior from the type object.* Let's say we wanted to change the monster's attack string when it's near death. Since the attack string is always accessed through `Monster`, we have a convenient place to put that code:

    ```
    const char* Monster::getAttack()
    {
      if (health_ < LOW_HEALTH)
      {
        return "The monster flails weakly.";
      }

      return breed_.getAttack();
    }
    ```

    If outside code was calling `getAttack()` directly on the breed, we wouldn't have the opportunity to insert that logic.
  - *We have to write forwarding methods for everything the type object exposes.* This is the tedious part of this design. If our type object class has a large number of

methods, the object class will have to have its own methods for each of the ones that we want to be publicly visible.

- **If the type object is exposed:**
  - *Outside code can interact with type objects without having an instance of the typed class.* If the type object is encapsulated, there's no way to use it without also having a typed object that wraps it. This prevents us, for example, from using our constructor pattern where new monsters are created by calling a method on the breed. If users can't get to breeds directly, they wouldn't be able to call it.
  - *The type object is now part of the object's public API.* In general, narrow interfaces are easier to maintain than wide ones—the less you expose to the rest of the codebase, the less complexity and maintenance you have to deal with. By exposing the type object, we widen the object's API to include everything the type object provides.

How are typed objects created?

With this pattern, each "object" is now a pair of objects: the main object and the type object it uses. So how do we create and bind the two together?

- **Construct the object and pass in its type object:**
  - *Outside code can control allocation.* Since the calling code is constructing both objects itself, it can control where in memory that occurs. If we want our objects to be usable in a variety of different memory scenarios (different allocators, on the stack, etc.) this gives us the flexibility to do that.
- **Call a "constructor" function on the type object:**
  - *The type object controls memory allocation.* This is the other side of the coin. If we *don't* want users to choose where in memory our objects are created, requiring them to go through a factory method on the type object gives us control over that. This can be useful if we want to ensure all of our objects come from a certain Object Pool or other memory allocator.

Can the type change?

So far, we've presumed that once an object is created and bound to its type object that that binding will never change. The type an object is created with is the type it dies with. This isn't strictly necessary. We *could* allow an object to change its type over time.

Let's look back at our example. When a monster dies, the designers tell us sometimes they want its corpse to become a reanimated zombie. We could implement this by spawning a new monster with a zombie breed when a monster dies, but another option is to simply get the existing monster and change its breed to a zombie one.

- **If the type doesn't change:**
    - *It's simpler both to code and to understand.* At a conceptual level, "type" is something most people probably will not expect to change. This codifies that assumption.
    - *It's easier to debug.* If we're trying to track down a bug where a monster gets into some weird state, it simplifies our job if we can take for granted that the breed we're looking at *now* is the breed the monster has always had.
- **If the type can change:**
    - *There's less object creation.* In our example, if the type can't change, we'll be forced to burn CPU cycles creating a new zombie monster, copying over any attributes from the original monster that need to be preserved, and then deleting it. If we can change the type, all that work gets replaced by a simple assignment.
    - *We need to be careful that assumptions are met.* There's a fairly tight coupling between an object and its type. For example, a breed might assume that a monster's *current* health is never above the starting health that comes from the breed.

      If we allow the breed to change, we need to make sure that the new type's requirements are met by the existing object. When we change the type, we will probably need to execute some validation code to make sure the object is now in a state that makes sense for the new type.

What kind of inheritance is supported?

- **No inheritance:**
    - *It's simple.* Simplest is often best. If you don't have a ton of data that needs sharing between your type objects, why make things hard on yourself?
    - *It can lead to duplicated effort.* I've yet to see an authoring system where designers *didn't* want some kind of inheritance. When you've got fifty different kinds of elves, having to tune their health by changing the same number in fifty different places *sucks*.
- **Single inheritance:**
    - *It's still relatively simple.* It's easy to implement, but, more importantly, it's also pretty easy to understand. If non-technical users are going to be working with the system, the fewer moving parts, the better. There's a reason a lot of programming languages only support single inheritance. It seems to be a sweet spot between power and simplicity.
    - *Looking up attributes is slower.* To get a given piece of data from a type object, we

might need to walk up the inheritance chain to find the type that ultimately decides the value. If we're in performance-critical code, we may not want to spend time on this.

- **Multiple inheritance:**
  - *Almost all data duplication can be avoided.* With a good multiple inheritance system, users can build a hierarchy for their type objects that has almost no redundancy. When it comes time to tune numbers, we can avoid a lot of copy and paste.
  - *It's complex.* Unfortunately, the benefits for this seem to be more theoretical than practical. Multiple inheritance is hard to understand and reason about.

    If our Zombie Dragon type inherits both from Zombie and Dragon, which attributes come from Zombie and which come from Dragon? In order to use the system, users will need to understand how the inheritance graph is traversed and have the foresight to design an intelligent hierarchy.

    Most C++ coding standards I see today tend to ban multiple inheritance, and Java and C# lack it completely. That's an acknowledgement of a sad fact: it's so hard to get it right that it's often best to not use it at all. While it's worth thinking about, it's rare that you'll want to use multiple inheritance for the type objects in your games. As always, simpler is better.

See Also

- The high-level problem this pattern addresses is sharing data and behavior between several objects. Another pattern that addresses the same problem in a different way is Prototype [GoF].
- Type Object is a close cousin to Flyweight [GoF]. Both let you share data across instances. With Flyweight, the intent is on saving memory, and the shared data may not represent any conceptual "type" of object. With the Type Object pattern, the focus is on organization and flexibility.
- There's a lot of similarity between this pattern and the State [GoF] pattern. Both patterns let an object delegate part of what defines itself to another object. With a type object, we're usually delegating what the object *is*: invariant data that broadly describes the object. With State, we delegate what an object *is right now*: temporal data that describes an object's current configuration.

  When we discussed having an object change its type, you can look at that as having our Type Object serve double duty as a State too.

Decoupling Patterns

Game Programming Patterns

Once you get the hang of a programming language, writing code to do what you want is actually pretty easy. What's hard is writing code that's easy to adapt when your requirements *change*. Rarely do we have the luxury of a perfect feature set before we've fired up our editor.

A powerful tool we have for making change easier is *decoupling*. When we say two pieces of code are "decoupled", we mean a change in one usually doesn't require a change in the other. When you change some feature in your game, the fewer places in code you have to touch, the easier it is.

Components decouple different domains in your game from each other within a single entity that has aspects of all of them. Event Queues decouple two objects communicating with each other, both statically and *in time*. Service Locators let code access a facility without being bound to the code that provides it.

The Patterns

- Component
- Event Queue
- Service Locator

Component

Intent

Allow a single entity to span multiple domains without coupling the domains to each other.

Motivation

Let's say we're building a platformer. The Italian plumber demographic is covered, so ours will star a Danish baker, Bjørn. It stands to reason that we'll have a class representing our friendly pastry chef, and it will contain everything he does in the game.

> Brilliant game ideas like this are why I'm a programmer and not a designer.

Since the player controls him, that means reading controller input and translating that input into motion. And, of course, he needs to interact with the level, so some physics and collision go in there. Once that's done, he's got to show up on screen, so toss in animation and rendering. He'll probably play some sounds too.

Hold on a minute; this is getting out of control. Software Architecture 101 tells us that different domains in a program should be kept isolated from each other. If we're making a word processor, the code that handles printing shouldn't be affected by the code that loads and saves documents. A game doesn't have the same domains as a business app, but the rule still applies.

As much as possible, we don't want AI, physics, rendering, sound and other domains to know about each other, but now we've got all of that crammed into one class. We've seen where this road leads to: a 5,000-line dumping ground source file so big that only the bravest ninja coders on your team even dare to go in there.

This is great job security for the few who can tame it, but it's hell for the rest of us. A class

that big means even the most seemingly trivial changes can have far-reaching implications. Soon, the class collects *bugs* faster than it collects *features*.

The Gordian knot

Even worse than the simple scale problem is the coupling one. All of the different systems in our game have been tied into a giant knotted ball of code like:

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
  playSound(HIT_FLOOR);
}
```

Any programmer trying to make a change in code like that will need to know something about physics, graphics, and sound just to make sure they don't break anything.

> While coupling like this sucks in *any* game, it's even worse on modern games that use concurrency. On multi-core hardware, it's vital that code is running on multiple threads simultaneously. One common way to split a game across threads is along domain boundaries—run AI on one core, sound on another, rendering on a third, etc.
>
> Once you do that, it's critical that those domains stay decoupled in order to avoid deadlocks or other fiendish concurrency bugs. Having a single class with an `UpdateSounds()` method that must be called from one thread and a `RenderGraphics()` method that must be called from another is begging for those kinds of bugs to happen.

These two problems compound each other; the class touches so many domains that every programmer will have to work on it, but it's so huge that doing so is a nightmare. If it gets bad enough, coders will start putting hacks in other parts of the codebase just to stay out of the hairball that this `Bjorn` class has become.

Cutting the knot

We can solve this like Alexander the Great—with a sword. We'll take our monolithic `Bjorn` class and slice it into separate parts along domain boundaries. For example, we'll take all of the code for handling user input and move it into a separate `InputComponent` class. `Bjorn` will then own an instance of this component. We'll repeat this process for each of the domains that `Bjorn` touches.

When we're done, we'll have moved almost everything out of `Bjorn`. All that remains is a thin shell that binds the components together. We've solved our huge class problem by simply dividing it up into multiple smaller classes, but we've accomplished more than just that.

Loose ends

Our component classes are now decoupled. Even though `Bjorn` has a `PhysicsComponent` and a `GraphicsComponent`, the two don't know about each other. This means the person working on physics can modify their component without needing to know anything about graphics and vice versa.

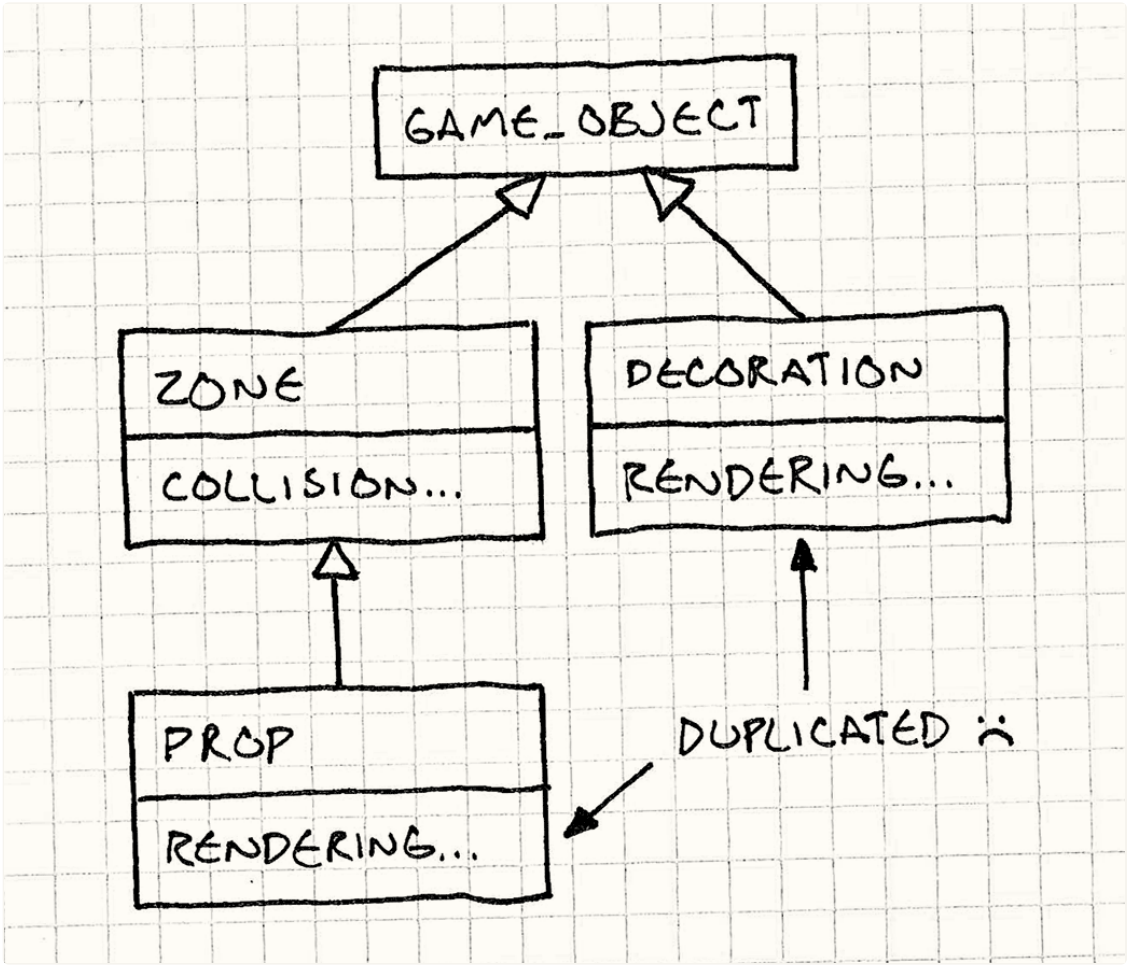
In practice, the components will need to have *some* interaction between themselves. For example, the AI component may need to tell the physics component where Bjørn is trying to go. However, we can restrict this to the components that *do* need to talk instead of just tossing them all in the same playpen together.

Tying back together

Another feature of this design is that the components are now reusable packages. So far, we've focused on our baker, but let's consider a couple of other kinds of objects in our game world. *Decorations* are things in the world the player sees but doesn't interact with: bushes, debris and other visual detail. *Props* are like decorations but can be touched: boxes, boulders, and trees. *Zones* are the opposite of decorations—invisible but interactive. They're useful for things like triggering a cutscene when Bjørn enters an area.

> When object-oriented programming first hit the scene, inheritance was the shiniest tool in its toolbox. It was considered the ultimate code-reuse hammer, and coders swung it often. Since then, we've learned the hard way that it's a heavy hammer indeed. Inheritance has its uses, but it's often too cumbersome for simple code reuse.
>
> Instead, the growing trend in software design is to use composition instead of inheritance when possible. Instead of sharing code between two classes by having them *inherit* from the same class, we do so by having them both *own an instance* of the same class.

Now, consider how we'd set up an inheritance hierarchy for those classes if we weren't using components. A first pass might look like:



We have a base `GameObject` class that has common stuff like position and orientation. `Zone` inherits from that and adds collision detection. Likewise, `Decoration` inherits from `GameObject` and adds rendering. `Prop` inherits from `Zone`, so it can reuse the collision code. However, `Prop` can't *also* inherit from `Decoration` to reuse the *rendering* code without running into the Deadly Diamond.

> The "Deadly Diamond" occurs in class hierarchies with multiple inheritance where there are two different paths to the same base class. The pain that causes is a bit out of the scope of this book, but understand that they named it "deadly" for a reason.

We could flip things around so that `Prop` inherits from `Decoration`, but then we end up having to duplicate the *collision* code. Either way, there's no clean way to reuse the collision and rendering code between the classes that need it without resorting to multiple inheritance. The only other option is to push everything up into `GameObject`, but then `Zone` is wasting memory on rendering data it doesn't need and `Decoration` is doing the same with physics.

Now, let's try it with components. Our subclasses disappear completely. Instead, we have a single `GameObject` class and two component classes: `PhysicsComponent` and `GraphicsComponent`. A decoration is simply a `GameObject` with a `GraphicsComponent` but no `PhysicsComponent`. A zone is the opposite, and a prop has both components. No code duplication, no multiple inheritance, and only three classes

instead of four.

> A restaurant menu is a good analogy. If each entity is a monolithic class, it's like you can only order combos. We need to have a separate class for each possible *combination* of features. To satisfy every customer, we would need dozens of combos.
>
> Components are à la carte dining—each customer can select just the dishes they want, and the menu is a list of the dishes they can choose from.

Components are basically plug-and-play for objects. They let us build complex entities with rich behavior by plugging different reusable component objects into sockets on the entity. Think software Voltron.

The Pattern

A **single entity spans multiple domains**. To keep the domains isolated, the code for each is placed in its own **component class**. The entity is reduced to a simple **container of components**.

> "Component", like "Object", is one of those words that means everything and nothing in programming. Because of that, it's been used to describe a few concepts. In business software, there's a "Component" design pattern that describes decoupled services that communicate over the web.
>
> I tried to find a different name for this unrelated pattern found in games, but "Component" seems to be the most common term for it. Since design patterns are about documenting existing practices, I don't have the luxury of coining a new term. So, following in the footsteps of XNA, Delta3D, and others, "Component" it is.

When to Use It

Components are most commonly found within the core class that defines the entities in a game, but they may be useful in other places as well. This pattern can be put to good use when any of these are true:

- You have a class that touches multiple domains which you want to keep decoupled from each other.
- A class is getting massive and hard to work with.
- You want to be able to define a variety of objects that share different capabilities, but

using inheritance doesn't let you pick the parts you want to reuse precisely enough.

Keep in Mind

The Component pattern adds a good bit of complexity over simply making a class and putting code in it. Each conceptual "object" becomes a cluster of objects that must be instantiated, initialized, and correctly wired together. Communication between the different components becomes more challenging, and controlling how they occupy memory is more complex.

For a large codebase, this complexity may be worth it for the decoupling and code reuse it enables, but take care to ensure you aren't over-engineering a "solution" to a non-existent problem before applying this pattern.

Another consequence of using components is that you often have to hop through a level of indirection to get anything done. Given the container object, first you have to get the component you want, *then* you can do what you need. In performance-critical inner loops, this pointer following may lead to poor performance.

> There's a flip side to this coin. The Component pattern can often *improve* performance and cache coherence. Components make it easier to use the Data Locality pattern to organize your data in the order that the CPU wants it.

Sample Code

One of the biggest challenges for me in writing this book is figuring out how to isolate each pattern. Many design patterns exist to contain code that itself isn't part of the pattern. In order to distill the pattern down to its essence, I try to cut as much of that out as possible, but at some point it becomes a bit like explaining how to organize a closet without showing any clothes.

The Component pattern is a particularly hard one. You can't get a real feel for it without seeing some code for each of the domains that it decouples, so I'll have to sketch in a bit more of Bjørn's code than I'd like. The pattern is really only the component *classes* themselves, but the code in them should help clarify what the classes are for. It's fake code—it calls into other classes that aren't presented here—but it should give you an idea of what we're going for.

A monolithic class

To get a clearer picture of how this pattern is applied, we'll start by showing a monolithic `Bjorn` class that does everything we need but *doesn't* use this pattern:

> I should point out that using the actual name of the character in the codebase is usually a bad idea. The marketing department has an annoying habit of demanding name changes days before you ship. “Focus tests show males between 11 and 15 respond negatively to ‘Bjørn’. Use ‘Sven’ instead.”
>
> This is why many software projects use internal-only codenames. Well, that and because it’s more fun to tell people you’re working on “Big Electric Cat” than just “the next version of 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_;
};
```

`Bjorn` has an `update()` method that gets called once per frame by the game:

```
void Bjorn::update(World& world, Graphics& graphics)
{
  // Apply user input to hero's velocity.
  switch (Controller::getJoystickDirection())
  {
    case DIR_LEFT:
      velocity_ -= WALK_ACCELERATION;
      break;

    case DIR_RIGHT:
      velocity_ += WALK_ACCELERATION;
      break;
  }

  // Modify position by velocity.
```

```
  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_);
}
```

It reads the joystick to determine how to accelerate the baker. Then it resolves its new position with the physics engine. Finally, it draws Bjørn onto the screen.

The sample implementation here is trivially simple. There's no gravity, animation, or any of the dozens of other details that make a character fun to play. Even so, we can see that we've got a single function that several different coders on our team will probably have to spend time in, and it's starting to get a bit messy. Imagine this scaled up to a thousand lines and you can get an idea of how painful it can become.

Splitting out a domain

Starting with one domain, let's pull a piece out of `Bjorn` and push it into a separate component class. We'll start with the first domain that gets processed: input. The first thing `Bjorn` does is read in user input and adjust his velocity based on it. Let's move that logic out into a separate class:

```
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;
};
```

Pretty simple. We've taken the first section of `Bjorn`'s `update()` method and put it into this class. The changes to `Bjorn` are also straightforward:

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);

    // Modify position by velocity.
    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_;
};
```

`Bjorn` now owns an `InputComponent` object. Where before he was handling user input directly in the `update()` method, now he delegates to the component:

```
input_.update(*this);
```

We've only started, but already we've gotten rid of some coupling—the main `Bjorn` class no longer has any reference to `Controller`. This will come in handy later.

Splitting out the rest

Now, let's go ahead and do the same cut-and-paste job on the physics and graphics code. Here's our new `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_;
};
```

In addition to moving the physics *behavior* out of the main `Bjorn` class, you can see we've also moved out the *data* too: The `Volume` object is now owned by the component.

Last but not least, here's where the rendering code lives now:

```
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_;
```

```
};
```

We've yanked almost everything out, so what's left of our humble pastry chef? Not much:

```
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_;
};
```

The `Bjorn` class now basically does two things: it holds the set of components that actually define it, and it holds the state that is shared across multiple domains. Position and velocity are still in the core `Bjorn` class for two reasons. First, they are "pan-domain" state—almost every component will make use of them, so it isn't clear which component *should* own them if we did want to push them down.

Secondly, and more importantly, it gives us an easy way for the components to communicate without being coupled to each other. Let's see if we can put that to use.

Robo-Bjørn

So far, we've pushed our behavior out to separate component classes, but we haven't *abstracted* the behavior out. `Bjorn` still knows the exact concrete classes where his behavior is defined. Let's change that.

We'll take our component for handling user input and hide it behind an interface. We'll turn `InputComponent` into an abstract base class:

```
class InputComponent
{
public:
  virtual ~InputComponent() {}
  virtual void update(Bjorn& bjorn) = 0;
};
```

Then, we'll take our existing user input handling code and push it down into a class that implements that interface:

```
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;
};
```

We'll change `Bjorn` to hold a pointer to the input component instead of having an inline instance:

```
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_;
};
```

Now, when we instantiate `Bjorn`, we can pass in an input component for it to use, like so:

```
Bjorn* bjorn = new Bjorn(new PlayerInputComponent());
```

This instance can be any concrete type that implements our abstract `InputComponent` interface. We pay a price for this—`update()` is now a virtual method call, which is a little slower. What do we get in return for this cost?

Most consoles require a game to support "demo mode." If the player sits at the main menu without doing anything, the game will start playing automatically, with the computer standing in for the player. This keeps the game from burning the main menu into your TV and also makes the game look nicer when it's running on a kiosk in a store.

Hiding the input component class behind an interface lets us get that working. We already have our concrete `PlayerInputComponent` that's normally used when playing the game. Now, let's make another one:

```
class DemoInputComponent : public InputComponent
{
public:
  virtual void update(Bjorn& bjorn)
  {
    // AI to automatically control Bjorn...
  }
};
```

When the game goes into demo mode, instead of constructing Bjørn like we did earlier, we'll wire him up with our new component:

```
Bjorn* bjorn = new Bjorn(new DemoInputComponent());
```

And now, just by swapping out a component, we've got a fully functioning computer-controlled player for demo mode. We're able to reuse all of the other code for Bjørn—physics and graphics don't even know there's a difference. Maybe I'm a bit strange, but it's stuff like this that gets me up in the morning.

> That, and coffee. Sweet, steaming hot coffee.

No Bjørn at all?

If you look at our `Bjorn` class now, you'll notice there's nothing really "Bjørn" about it—it's just a component bag. In fact, it looks like a pretty good candidate for a base "game object" class that we can use for *every* object in the game. All we need to do is pass in *all* the components, and we can build any kind of object by picking and choosing parts like Dr. Frankenstein.

Let's take our two remaining concrete components—physics and graphics—and hide them behind interfaces like we did with input:

```
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;
};
```

Then we re-christen `Bjorn` into a generic `GameObject` class that uses those interfaces:

```
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_;
};
```

Some component systems take this even further. Instead of a `GameObject` that contains its components, the game entity is just an ID, a number. Then, you maintain separate collections of components where each one

knows the ID of the entity its attached to.

These entity component systems take decoupling components to the extreme and let you add new components to an entity without the entity even knowing. The Data Locality chapter has more details.

Our existing concrete classes will get renamed and implement those interfaces:

```
class BjornPhysicsComponent : public PhysicsComponent
{
public:
  virtual void update(GameObject& obj, World& world)
  {
    // Physics code...
  }
};

class BjornGraphicsComponent : public GraphicsComponent
{
public:
  virtual void update(GameObject& obj, Graphics& graphics)
  {
    // Graphics code...
  }
};
```

And now we can build an object that has all of Bjørn's original behavior without having to actually create a class for him, just like this:

```
GameObject* createBjorn()
{
  return new GameObject(new PlayerInputComponent(),
                        new BjornPhysicsComponent(),
                        new BjornGraphicsComponent());
}
```

This `createBjorn()` function is, of course, an example of the classic Gang of Four Factory Method GoF pattern.

By defining other functions that instantiate `GameObjects` with different components, we can create all of the different kinds of objects our game needs.

Design Decisions

The most important design question you'll need to answer with this pattern is, "What set of components do I need?" The answer there is going to depend on the needs and genre of your game. The bigger and more complex your engine is, the more finely you'll likely want

to slice your components.

Beyond that, there are a couple of more specific options to consider:

How does the object get its components?

Once we've split up our monolithic object into a few separate component parts, we have to decide who puts the parts back together.

- **If the object creates its own components:**
    - *It ensures that the object always has the components it needs.* You never have to worry about someone forgetting to wire up the right components to the object and breaking the game. The container object itself takes care of it for you.
    - *It's harder to reconfigure the object.* One of the powerful features of this pattern is that it lets you build new kinds of objects simply by recombining components. If our object always wires itself with the same set of hard-coded components, we aren't taking advantage of that flexibility.
- **If outside code provides the components:**
    - *The object becomes more flexible.* We can completely change the behavior of the object by giving it different components to work with. Taken to its fullest extent, our object becomes a generic component container that we can reuse over and over again for different purposes.
    - *The object can be decoupled from the concrete component types.* If we're allowing outside code to pass in components, odds are good that we're also letting it pass in *derived* component types. At that point, the object only knows about the component *interfaces* and not the concrete types themselves. This can make for a nicely encapsulated architecture.

How do components communicate with each other?

Perfectly decoupled components that function in isolation is a nice ideal, but it doesn't really work in practice. The fact that these components are part of the *same* object implies that they are part of a larger whole and need to coordinate. That means communication.

So how can the components talk to each other? There are a couple of options, but unlike most design "alternatives" in this book, these aren't exclusive—you will likely support more than one at the same time in your designs.

- **By modifying the container object's state:**
    - *It keeps the components decoupled.* When our `InputComponent` set Bjørn's velocity and the `PhysicsComponent` later used it, the two components had no idea that the other even existed. For all they knew, Bjørn's velocity could have

changed through black magic.

- *It requires any information that components need to share to get pushed up into the container object.* Often, there's state that's really only needed by a subset of the components. For example, an animation and a rendering component may need to share information that's graphics-specific. Pushing that information up into the container object where *every* component can get to it muddies the object class.

  Worse, if we use the same container object class with different component configurations, we can end up wasting memory on state that isn't needed by *any* of the object's components. If we push some rendering-specific data into the container object, any invisible object will be burning memory on it with no benefit.

- *It makes communication implicit and dependent on the order that components are processed.* In our sample code, the original monolithic `update()` method had a very carefully laid out order of operations. The user input modified the velocity, which was then used by the physics code to modify the position, which in turn was used by the rendering code to draw Bjørn at the right spot. When we split that code out into components, we were careful to preserve that order of operations.

  If we hadn't, we would have introduced subtle, hard-to-track bugs. For example, if we'd updated the graphics component *first*, we would wrongly render Bjørn at his position on the *last* frame, not this one. If you imagine several more components and lots more code, then you can get an idea of how hard it can be to avoid bugs like this.

  > Shared mutable state like this where lots of code is reading and writing the same data is notoriously hard to get right. That's a big part of why academics are spending time researching pure functional languages like Haskell where there is no mutable state at all.

- **By referring directly to each other:**

  The idea here is that components that need to talk will have direct references to each other without having to go through the container object at all.

  Let's say we want to let Bjørn jump. The graphics code needs to know if he should be drawn using a jump sprite or not. It can determine this by asking the physics engine if he's currently on the ground. An easy way to do this is by letting the graphics component know about the physics component directly:

```
class BjornGraphicsComponent
{
public:
  BjornGraphicsComponent(BjornPhysicsComponent* physics)
```

```
  : physics_(physics)
  {}

  void Update(GameObject& obj, Graphics& graphics)
  {
    Sprite* sprite;
    if (!physics_->isOnGround())
    {
      sprite = &spriteJump_;
    }
    else
    {
      // Existing graphics code...
    }

    graphics.draw(*sprite, obj.x, obj.y);
  }

private:
  BjornPhysicsComponent* physics_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
  Sprite spriteJump_;
};
```

When we construct Bjørn's `GraphicsComponent`, we'll give it a reference to his corresponding `PhysicsComponent`.

- *It's simple and fast.* Communication is a direct method call from one object to another. The component can call any method that is supported by the component it has a reference to. It's a free-for-all.
- *The two components are tightly coupled.* The downside of the free-for-all. We've basically taken a step back towards our monolithic class. It's not quite as bad as the original single class though, since we're at least restricting the coupling to only the component pairs that need to interact.

- **By sending messages:**
  - This is the most complex alternative. We can actually build a little messaging system into our container object and let the components broadcast information to each other.

    Here's one possible implementation. We'll start by defining a base `Component` interface that all of our components will implement:

    ```
    class Component
    ```

```
{
public:
  virtual ~Component() {}
  virtual void receive(int message) = 0;
};
```

It has a single `receive()` method that component classes implement in order to listen to an incoming message. Here, we're just using an `int` to identify the message, but a fuller implementation could attach additional data to the message.

Then, we'll add a method to our container object for sending messages:

```
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];
};
```

Now, if a component has access to its container, it can send messages to the container, which will rebroadcast the message to all of the contained components. (That includes the original component that sent the message; be careful that you don't get stuck in a feedback loop!) This has a couple of consequences:

> If you really want to get fancy, you can even make this message system *queue* messages to be delivered later. For more on this, see Event Queue.

- *Sibling components are decoupled.* By going through the parent container object, like our shared state alternative, we ensure that the components are still decoupled from each other. With this system, the only coupling they have is the message values themselves.

> The Gang of Four call this the Mediator GoF pattern — two or more objects communicate with each other indirectly by routing the

message through an intermediate object. In this case, the container object itself is the mediator.

- *The container object is simple.* Unlike using shared state where the container object itself owns and knows about data used by the components, here, all it does is blindly pass the messages along. That can be useful for letting two components pass very domain-specific information between themselves without having that bleed into the container object.

Unsurprisingly, there's no one best answer here. What you'll likely end up doing is using a bit of all of them. Shared state is useful for the really basic stuff that you can take for granted that every object has—things like position and size.

Some domains are distinct but still closely related. Think animation and rendering, user input and AI, or physics and collision. If you have separate components for each half of those pairs, you may find it easiest to just let them know directly about their other half.

Messaging is useful for "less important" communication. Its fire-and-forget nature is a good fit for things like having an audio component play a sound when a physics component sends a message that the object has collided with something.

As always, I recommend you start simple and then add in additional communication paths if you need them.

See Also

- The Unity framework's core `GameObject` class is designed entirely around components.
- The open source Delta3D engine has a base `GameActor` class that implements this pattern with the appropriately named `ActorComponent` base class.
- Microsoft's XNA game framework comes with a core `Game` class. It owns a collection of `GameComponent` objects. Where our example uses components at the individual game entity level, XNA implements the pattern at the level of the main game object itself, but the purpose is the same.
- This pattern bears resemblance to the Gang of Four's Strategy [GoF] pattern. Both patterns are about taking part of an object's behavior and delegating it to a separate subordinate object. The difference is that with the Strategy pattern, the separate "strategy" object is usually stateless—it encapsulates an algorithm, but no data. It defines *how* an object behaves, but not *what* it is.

  Components are a bit more self-important. They often hold state that describes the object and helps define its actual identity. However, the line may blur. You may have some components that don't need any local state. In that case, you're free to use the

same component *instance* across multiple container objects. At that point, it really is behaving more akin to a strategy.

← Previous Chapter ≡ The Book Next Chapter →

Event Queue

Intent

Decouple when a message or event is sent from when it is processed.

Motivation

Unless you live under one of the few rocks that still lack Internet access, you've probably already heard of an "event queue". If not, maybe "message queue", or "event loop", or "message pump" rings a bell. To refresh your memory, let's walk through a couple of common manifestations of the pattern.

> For most of the chapter, I use "event" and "message" interchangeably. Where the distinction matters, I'll make it obvious.

GUI event loops

If you've ever done any user interface programming, then you're well acquainted with *events*. Every time the user interacts with your program—clicks a button, pulls down a menu, or presses a key—the operating system generates an event. It throws this object at your app, and your job is to grab it and hook it up to some interesting behavior.

> This application style is so common, it's considered a paradigm: *event-driven programming*.

In order to receive these missives, somewhere deep in the bowels of your code is an *event loop*. It looks roughly like this:

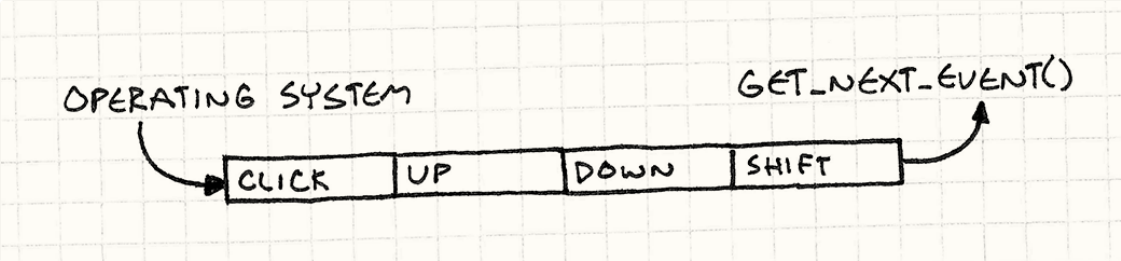
```
while (running)
{
  Event event = getNextEvent();
  // Handle event...
```

```
}
```

The call to `getNextEvent()` pulls a bit of unprocessed user input into your app. You route it to an event handler and, like magic, your application comes to life. The interesting part is that the application *pulls* in the event when *it* wants it. The operating system doesn't just immediately jump to some code in your app when the user pokes a peripheral.

> In contrast, *interrupts* from the operating system *do* work like that. When an interrupt happens, the OS stops whatever your app was doing and forces it to jump to an interrupt handler. This abruptness is why interrupts are so hard to work with.

That means when user input comes in, it needs to go somewhere so that the operating system doesn't lose it between when the device driver reported the input and when your app gets around to calling `getNextEvent()`. That "somewhere" is a *queue*.



When user input comes in, the OS adds it to a queue of unprocessed events. When you call `getNextEvent()`, that pulls the oldest event off the queue and hands it to your application.

Central event bus

Most games aren't event-driven like this, but it is common for a game to have its own event queue as the backbone of its nervous system. You'll often hear "central", "global", or "main" used to describe it. It's used for high level communication between game systems that want to stay decoupled.

> If you want to know *why* they aren't event-driven, crack open the Game Loop chapter.

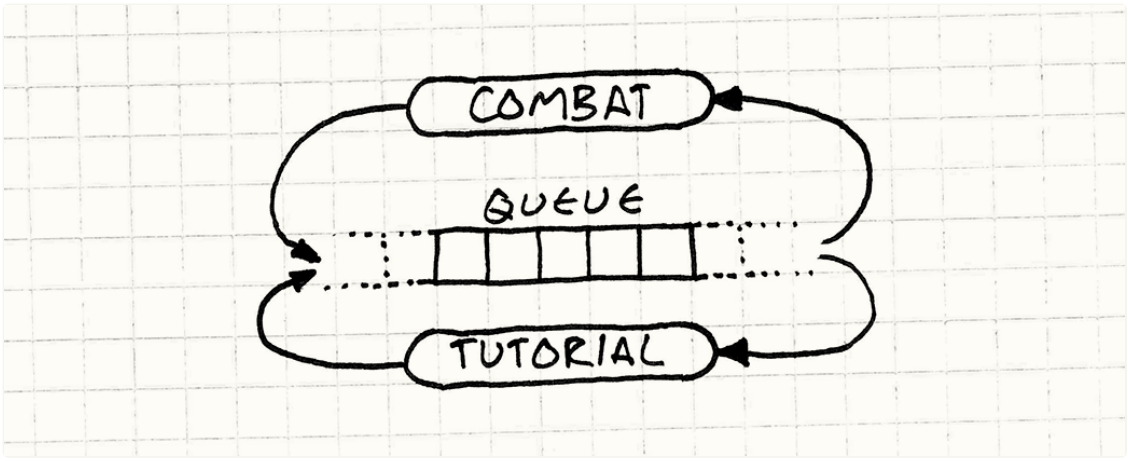
Say your game has a tutorial system to display help boxes after specific in-game events. For example, the first time the player vanquishes a foul beastie, you want to show a little balloon that says, "Press X to grab the loot!"

> Tutorial systems are a pain to implement gracefully, and most players will spend only a fraction of their time using in-game help, so it feels like they aren't worth the effort. But that fraction where they *are* using the tutorial can be invaluable for easing the player into your game.

Your gameplay and combat code are likely complex enough as it is. The last thing you want to do is stuff a bunch of checks for triggering tutorials in there. Instead, you could have a central event queue. Any game system can send to it, so the combat code can add an "enemy died" event every time you slay a foe.

Likewise, any game system can *receive* events from the queue. The tutorial engine registers itself with the queue and indicates it wants to receive "enemy died" events. This way, knowledge of an enemy dying makes its way from the combat system over to the tutorial engine without the two being directly aware of each other.

> This model where you have a shared space that entities can post information to and get notified by is similar to blackboard systems in the AI field.



I thought about using this as the example for the rest of the chapter, but I'm not generally a fan of big global systems. Event queues don't have to be for communicating across the entire game engine. They can be just as useful within a single class or domain.

Say what?

So, instead, let's add sound to our game. Humans are mainly visual animals, but hearing is deeply connected to our emotions and our sense of physical space. The right simulated echo can make a black screen feel like an enormous cavern, and a well-timed violin adagio can make your heartstrings hum in sympathetic resonance.

To get our game wound for sound, we'll start with the simplest possible approach and see how it goes. We'll add a little "audio engine" that has an API for playing a sound given an identifier and a volume:

> While I almost always shy away from the Singleton [GoF] pattern, this is one of the places where it may fit since the machine likely only has one set of speakers. I'm taking a simpler approach and just making the method static.

```
class Audio
{
public:
  static void playSound(SoundId id, int volume);
};
```

It's responsible for loading the appropriate sound resource, finding an available channel to play it on, and starting it up. This chapter isn't about some platform's real audio API, so I'll conjure one up that we can presume is implemented elsewhere. Using it, we write our method like so:

```
void Audio::playSound(SoundId id, int volume)
{
  ResourceId resource = loadSound(id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, volume);
}
```

We check that in, create a few sound files, and start sprinkling `playSound()` calls through our codebase like some magical audio fairy. For example, in our UI code, we play a little bloop when the selected menu item changes:

```
class Menu
{
public:
  void onSelect(int index)
  {
    Audio::playSound(SOUND_BLOOP, VOL_MAX);
    // Other stuff...
  }
};
```

After doing this, we notice that sometimes when you switch menu items, the whole screen freezes for a few frames. We've hit our first issue:

- **Problem 1: The API blocks the caller until the audio engine has completely processed the request.**

Our `playSound()` method is *synchronous*—it doesn't return back to the caller until bloops are coming out of the speakers. If a sound file has to be loaded from disc first, that may take a while. In the meantime, the rest of the game is frozen.

Ignoring that for now, we move on. In the AI code, we add a call to let out a wail of anguish when an enemy takes damage from the player. Nothing warms a gamer's heart like inflicting simulated pain on a virtual living being.

It works, but sometimes when the hero does a mighty attack, it hits two enemies in the exact same frame. That causes the game to play the wail sound twice simultaneously. If you know anything about audio, you know mixing multiple sounds together sums their waveforms. When those are the *same* waveform, it's the same as *one* sound played *twice as loud*. It's jarringly loud.

> I ran into this exact issue working on Henry Hatsworth in the Puzzling Adventure. My solution there is similar to what we'll cover here.

We have a related problem in boss fights when piles of minions are running around causing mayhem. The hardware can only play so many sounds at one time. When we go over that limit, sounds get ignored or cut off.

To handle these issues, we need to look at the entire *set* of sound calls to aggregate and prioritize them. Unfortunately, our audio API handles each `playSound()` call independently. It sees requests through a pinhole, one at a time.

- **Problem 2: Requests cannot be processed in aggregate.**

These problems seem like mere annoyances compared to the next issue that falls in our lap. By now, we've strewn `playSound()` calls throughout the codebase in lots of different game systems. But our game engine is running on modern multi-core hardware. To take advantage of those cores, we distribute those systems on different threads—rendering on one, AI on another, etc.

Since our API is synchronous, it runs on the *caller's* thread. When we call it from different game systems, we're hitting our API concurrently from multiple threads. Look at that sample code. See any thread synchronization? Me neither.

This is particularly egregious because we intended to have a *separate* thread for audio. It's just sitting there totally idle while these other threads are busy stepping all over each other and breaking things.

- **Problem 3: Requests are processed on the wrong thread.**

The common theme to these problems is that the audio engine interprets a call to `playSound()` to mean, "Drop everything and play the sound right now!" *Immediacy* is the problem. Other game systems call `playSound()` at *their* convenience, but not necessarily when it's convenient for the audio engine to handle that request. To fix that, we'll decouple *receiving* a request from *processing* it.

The Pattern

A **queue** stores a series of **notifications or requests** in first-in, first-out order. Sending a notification **enqueues the request and returns**. The request processor then

processes items from the queue at a later time. Requests can be **handled directly** or **routed to interested parties**. This **decouples the sender from the receiver** both **statically** and **in time**.

When to Use It

If you only want to decouple *who* receives a message from its sender, patterns like Observer and Command will take care of this with less complexity. You only need a queue when you want to decouple something *in time*.

> I mention this in nearly every chapter, but it's worth emphasizing. Complexity slows you down, so treat simplicity as a precious resource.

I think of it in terms of pushing and pulling. You have some code A that wants another chunk B to do some work. The natural way for A to initiate that is by *pushing* the request to B.

Meanwhile, the natural way for B to process that request is by *pulling* it in at a convenient time in *its* run cycle. When you have a push model on one end and a pull model on the other, you need a buffer between them. That's what a queue provides that simpler decoupling patterns don't.

Queues give control to the code that pulls from it—the receiver can delay processing, aggregate requests, or discard them entirely. But queues do this by taking control *away* from the sender. All the sender can do is throw a request on the queue and hope for the best. This makes queues a poor fit when the sender needs a response.

Keep in Mind

Unlike some more modest patterns in this book, event queues are complex and tend to have a wide-reaching effect on the architecture of our games. That means you'll want to think hard about how—or if—you use one.

A central event queue is a global variable

One common use of this pattern is for a sort of Grand Central Station that all parts of the game can route messages through. It's a powerful piece of infrastructure, but *powerful* doesn't always mean *good*.

It took a while, but most of us learned the hard way that global variables are bad. When you have a piece of state that any part of the program can poke at, all sorts of subtle interdependencies creep in. This pattern wraps that state in a nice little protocol, but it's still a global, with all of the danger that entails.

The state of the world can change under you

Say some AI code posts an "entity died" event to a queue when a virtual minion shuffles off its mortal coil. That event hangs out in the queue for who knows how many frames until it eventually works its way to the front and gets processed.

Meanwhile, the experience system wants to track the heroine's body count and reward her for her grisly efficiency. It receives each "entity died" event and determines the kind of entity slain and the difficulty of the kill so it can dish out an appropriate reward.

That requires various pieces of state in the world. We need the entity that died so we can see how tough it was. We may want to inspect its surroundings to see what other obstacles or minions were nearby. But if the event isn't received until later, that stuff may be gone. The entity may have been deallocated, and other nearby foes may have wandered off.

When you receive an event, you have to be careful not to assume the *current* state of the world reflects how the world was *when the event was raised*. This means queued events tend to be more data heavy than events in synchronous systems. With the latter, the notification can say "something happened" and the receiver can look around for the details. With a queue, those ephemeral details must be captured when the event is sent so they can be used later.

You can get stuck in feedback loops

All event and message systems have to worry about cycles:

1. A sends an event.
2. B receives it and responds by sending an event.
3. That event happens to be one that A cares about, so it receives it. In response, it sends an event...
4. Go to 2.

When your messaging system is *synchronous*, you find cycles quickly—they overflow the stack and crash your game. With a queue, the asynchrony unwinds the stack, so the game may keep running even though spurious events are sloshing back and forth in there. A common rule to avoid this is to avoid *sending* events from within code that's *handling* one.

> A little debug logging in your event system is probably a good idea too.

Sample Code

We've already seen some code. It's not perfect, but it has the right basic functionality—the public API we want and the right low-level audio calls. All that's left for us to do now is fix its problems.

The first is that our API *blocks*. When a piece of code plays a sound, it can't do anything else until `playSound()` finishes loading the resource and actually starts making the speaker wiggle.

We want to defer that work until later so that `playSound()` can return quickly. To do that, we need to *reify* the request to play a sound. We need a little structure that stores the details of a pending request so we can keep it around until later:

```
struct PlayMessage
{
  SoundId id;
  int volume;
};
```

Next, we need to give `Audio` some storage space to keep track of these pending play messages. Now, your algorithms professor might tell you to use some exciting data structure here like a Fibonacci heap or a skip list, or, hell, at least a *linked* list. But in practice, the best way to store a bunch of homogenous things is almost always a plain old array:

> Algorithm researchers get paid to publish analyses of novel data structures. They aren't exactly incentivized to stick to the basics.

- No dynamic allocation.
- No memory overhead for bookkeeping information or pointers.
- Cache-friendly contiguous memory usage.

> For lots more on what being "cache friendly" means, see the chapter on Data Locality.

So let's do that:

```
class Audio
{
public:
  static void init()
  {
    numPending_ = 0;
  }

  // Other stuff...
private:
  static const int MAX_PENDING = 16;

  static PlayMessage pending_[MAX_PENDING];
```

```
  static int numPending_;
};
```

We can tune the array size to cover our worst case. To play a sound, we simply slot a new message in there at the end:

```
void Audio::playSound(SoundId id, int volume)
{
  assert(numPending_ < MAX_PENDING);

  pending_[numPending_].id = id;
  pending_[numPending_].volume = volume;
  numPending_++;
}
```

This lets `playSound()` return almost instantly, but we do still have to play the sound, of course. That code needs to go somewhere, and that somewhere is an `update()` method:

```
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;
  }

  // Other stuff...
};
```

As the name implies, this is the Update Method pattern.

Now, we need to call that from somewhere convenient. What "convenient" means depends on your game. It may mean calling it from the main game loop or from a dedicated audio thread.

This works fine, but it does presume we can process *every* sound request in a single call to `update()`. If you're doing something like processing a request asynchronously after its sound resource is loaded, that won't work. For `update()` to work on one request at a time, it needs to be able to pull requests out of the buffer while leaving the rest. In other

words, we need an actual queue.

A ring buffer

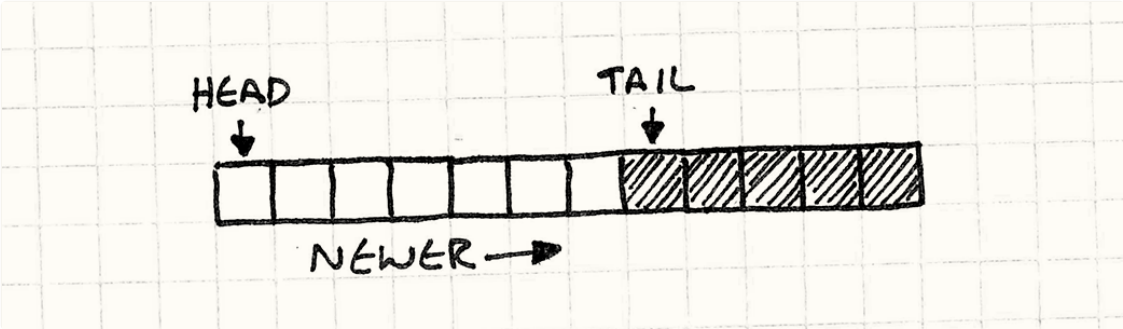
There are a bunch of ways to implement queues, but my favorite is called a *ring buffer*. It preserves everything that's great about arrays while letting us incrementally remove items from the front of the queue.

Now, I know what you're thinking. If we remove items from the beginning of the array, don't we have to shift all of the remaining items over? Isn't that slow?

This is why they made us learn linked lists—you can remove nodes from them without having to shift things around. Well, it turns out you can implement a queue without any shifting in an array too. I'll walk you through it, but first let's get precise on some terms:

- The **head** of the queue is where requests are *read* from. The head is the oldest pending request.
- The **tail** is the other end. It's the slot in the array where the next enqueued request will be *written*. Note that it's just *past* the end of the queue. You can think of it as a half-open range, if that helps.

Since `playSound()` appends new requests at the end of the array, the head starts at element zero and the tail grows to the right.



Let's code that up. First, we'll tweak our fields a bit to make these two markers explicit in the class:

```
class Audio
{
public:
  static void init()
  {
    head_ = 0;
    tail_ = 0;
  }

  // Methods...
private:
  static int head_;
```

```
  static int tail_;

  // Array...
};
```

In the implementation of `playSound()`, `numPending_` has been replaced with `tail_`, but otherwise it's the same:

```
void Audio::playSound(SoundId id, int volume)
{
  assert(tail_ < MAX_PENDING);

  // Add to the end of the list.
  pending_[tail_].id = id;
  pending_[tail_].volume = volume;
  tail_++;
}
```

The more interesting change is in `update()`:

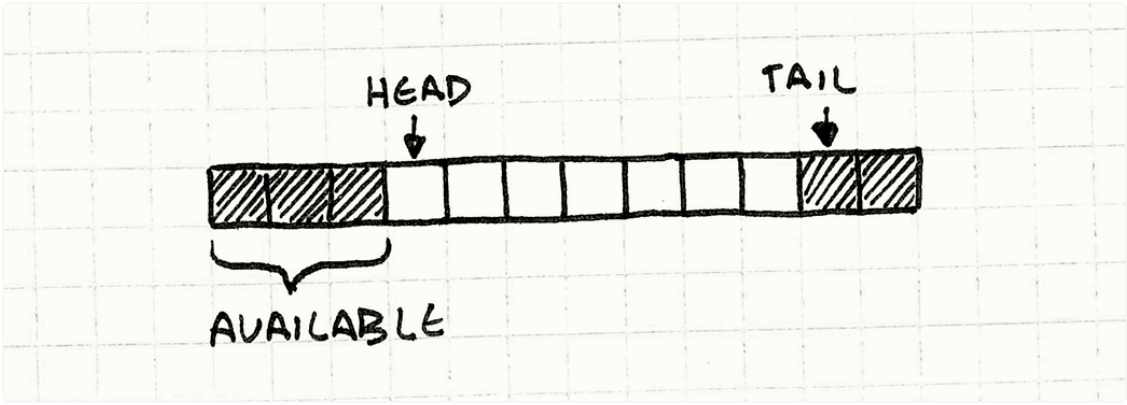
```
void Audio::update()
{
  // If there are no pending requests, do nothing.
  if (head_ == tail_) return;

  ResourceId resource = loadSound(pending_[head_].id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, pending_[head_].volume);

  head_++;
}
```

We process the request at the head and then discard it by advancing the head pointer to the right. We detect an empty queue by seeing if there's any distance between the head and tail.

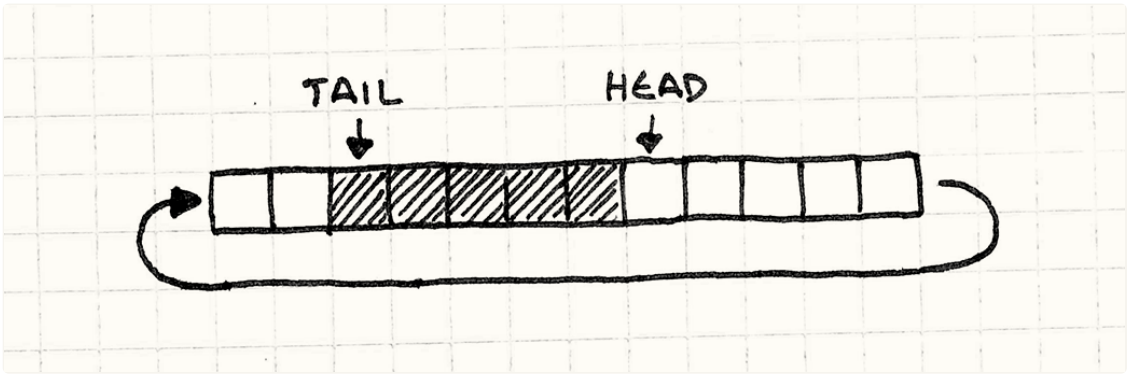
> This is why we made the tail one *past* the last item. It means that the queue will be empty if the head and tail are the same index.

Now we've got a queue—we can add to the end and remove from the front. There's an obvious problem, though. As we run requests through the queue, the head and tail keep crawling to the right. Eventually, `tail_` hits the end of the array, and party time is over. This is where it gets clever.

> Do you want party time to be over? No. You do not.



Notice that while the tail is creeping forward, the *head* is too. That means we've got array elements at the *beginning* of the array that aren't being used anymore. So what we do is wrap the tail back around to the beginning of the array when it runs off the end. That's why it's called a *ring* buffer—it acts like a circular array of cells.



Implementing that is remarkably easy. When we enqueue an item, we just need to make sure the tail wraps around to the beginning of the array when it reaches the end:

```
void Audio::playSound(SoundId id, int volume)
{
  assert((tail_ + 1) % MAX_PENDING != head_);

  // Add to the end of the list.
  pending_[tail_].id = id;
  pending_[tail_].volume = volume;
  tail_ = (tail_ + 1) % MAX_PENDING;
}
```

Replacing `tail_++` with an increment modulo the array size wraps the tail back around. The other change is the assertion. We need to ensure the queue doesn't overflow. As long as there are fewer than `MAX_PENDING` requests in the queue, there will be a little gap of unused cells between the head and the tail. If the queue fills up, those will be gone and, like some weird backwards Ouroboros, the tail will collide with the head and start overwriting it. The assertion ensures that this doesn't happen.

In `update()`, we wrap the head around too:

```
void Audio::update()
{
  // If there are no pending requests, do nothing.
  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;
}
```

There you go—a queue with no dynamic allocation, no copying elements around, and the cache-friendliness of a simple array.

> If the maximum capacity bugs you, you can use a growable array. When the queue gets full, allocate a new array twice the size of the current array (or some other multiple), then copy the items over.
>
> Even though you copy when they array grows, enqueuing an item still has constant *amortized* complexity.

Aggregating requests

Now that we've got a queue in place, we can move onto the other problems. The first is that multiple requests to play the same sound end up too loud. Since we know which requests are waiting to be processed now, all we need to do is merge a request if it matches an already pending one:

```
void Audio::playSound(SoundId id, int volume)
{
  // Walk the pending requests.
  for (int i = head_; i != tail_;
       i = (i + 1) % MAX_PENDING)
  {
    if (pending_[i].id == id)
    {
      // Use the larger of the two volumes.
      pending_[i].volume = max(volume, pending_[i].volume);

      // Don't need to enqueue.
      return;
    }
  }

  // Previous code...
}
```

When we get two requests to play the same sound, we collapse them to a single request for whichever is loudest. This "aggregation" is pretty rudimentary, but we could use the same idea to do more interesting batching.

Note that we're merging when the request is *enqueued*, not when it's *processed*. That's easier on our queue since we don't waste slots on redundant requests that will end up being collapsed later. It's also simpler to implement.

It does, however, put the processing burden on the caller. A call to `playSound()` will walk the entire queue before it returns, which could be slow if the queue is large. It may make more sense to aggregate in `update()` instead.

> Another way to avoid the *O(n)* cost of scanning the queue is to use a different data structure. If we use a hash table keyed on the `SoundId`, then we can check for duplicates in constant time.

There's something important to keep in mind here. The window of "simultaneous" requests that we can aggregate is only as big as the queue. If we process requests more quickly and the queue size stays small, then we'll have fewer opportunities to batch things together. Likewise, if processing lags behind and the queue gets full, we'll find more things to collapse.

This pattern insulates the requester from knowing when the request gets processed, but when you treat the entire queue as a live data structure to be played with, then lag between making a request and processing it can visibly affect behavior. Make sure you're OK with that before doing this.

Spanning threads

Finally, the most pernicious problem. With our synchronous audio API, whatever thread called `playSound()` was the thread that processed the request. That's often not what we want.

On today's multi-core hardware, you need more than one thread if you want to get the most out of your chip. There are infinite ways to distribute code across threads, but a common strategy is to move each domain of the game onto its own thread—audio, rendering, AI, etc.

> Straight-line code only runs on a single core at a time. If you don't use threads, even if you do the asynchronous-style programming that's in vogue, the best you'll do is keep one core busy, which is a fraction of your CPU's abilities.
>
> Server programmers compensate for that by splitting their application into multiple independent *processes*. That lets the OS run them concurrently on different cores. Games are almost always a single

process, so a bit of threading really helps.

We're in good shape to do that now that we have three critical pieces:

1. The code for requesting a sound is decoupled from the code that plays it.
2. We have a queue for marshalling between the two.
3. The queue is encapsulated from the rest of the program.

All that's left is to make the methods that modify the queue—`playSound()` and `update()`—thread-safe. Normally, I'd whip up some concrete code to do that, but since this is a book about architecture, I don't want to get mired in the details of any specific API or locking mechanism.

At a high level, all we need to do is ensure that the queue isn't modified concurrently. Since `playSound()` does a very small amount of work—basically just assigning a few fields—it can lock without blocking processing for long. In `update()`, we wait on something like a condition variable so that we don't burn CPU cycles until there's a request to process.

Design Decisions

Many games use event queues as a key part of their communication structure, and you can spend a ton of time designing all sorts of complex routing and filtering for messages. But before you go off and build something like the Los Angeles telephone switchboard, I encourage you to start simple. Here's a few starter questions to consider:

What goes in the queue?

I've used "event" and "message" interchangeably so far because it mostly doesn't matter. You get the same decoupling and aggregation abilities regardless of what you're stuffing in the queue, but there are some conceptual differences.

- **If you queue events:**

  An "event" or "notification" describes something that *already* happened, like "monster died". You queue it so that other objects can *respond* to the event, sort of like an asynchronous Observer [GoF] pattern.

  - *You are likely to allow multiple listeners.* Since the queue contains things that already happened, the sender probably doesn't care who receives it. From its perspective, the event is in the past and is already forgotten.
  - *The scope of the queue tends to be broader.* Event queues are often used to *broadcast* events to any and all interested parties. To allow maximum flexibility for which parties can be interested, these queues tend to be more globally visible.

- **If you queue messages:**

  A "message" or "request" describes an action that we *want* to happen *in the future*, like "play sound". You can think of this as an asynchronous API to a service.

  > Another word for "request" is "command", as in the Command GoF pattern, and queues can be used there too.

  - *You are more likely to have a single listener.* In the example, the queued messages are requests specifically for *the audio API* to play a sound. If other random parts of the game engine started stealing messages off the queue, it wouldn't do much good.

    > I say "more likely" here, because you can enqueue messages without caring which code processes it, as long as it gets processed *how* you expect. In that case, you're doing something akin to a Service Locator.

Who can read from the queue?

In our example, the queue is encapsulated and only the `Audio` class can read from it. In a user interface's event system, you can register listeners to your heart's content. You sometimes hear the terms "single-cast" and "broadcast" to distinguish these, and both styles are useful.

- **A single-cast queue:**

  This is the natural fit when a queue is part of a class's API. Like in our audio example, from the caller's perspective, they just see a `playSound()` method they can call.

  - *The queue becomes an implementation detail of the reader.* All the sender knows is that it sent a message.
  - *The queue is more encapsulated.* All other things being equal, more encapsulation is usually better.
  - *You don't have to worry about contention between listeners.* With multiple listeners, you have to decide if they *all* get every item (broadcast) or if *each* item in the queue is parceled out to *one* listener (something more like a work queue).

    In either case, the listeners may end up doing redundant work or interfering with each other, and you have to think carefully about the behavior you want. With a single listener, that complexity disappears.

- **A broadcast queue:**

This is how most "event" systems work. If you have ten listeners when an event comes in, all ten of them see the event.

- *Events can get dropped on the floor.* A corollary to the previous point is that if you have *zero* listeners, all zero of them see the event. In most broadcast systems, if there are no listeners at the point in time that an event is processed, the event gets discarded.

- *You may need to filter events.* Broadcast queues are often widely visible to much of the program, and you can end up with a bunch of listeners. Multiply lots of events times lots of listeners, and you end up with a ton of event handlers to invoke.

  To cut that down to size, most broadcast event systems let a listener winnow down the set of events they receive. For example, they may say they only want to receive mouse events or events within a certain region of the UI.

- **A work queue:**

  Like a broadcast queue, here you have multiple listeners too. The difference is that each item in the queue only goes to *one* of them. This is a common pattern for parceling out jobs to a pool of concurrently running threads.

  - *You have to schedule.* Since an item only goes to one listener, the queue needs logic to figure out the best one to choose. This may be as simple as round robin or random choice, or it could be some more complex prioritizing system.

Who can write to the queue?

This is the flip side of the previous design choice. This pattern works with all of the possible read/write configurations: one-to-one, one-to-many, many-to-one, or many-to-many.

> You sometimes hear "fan-in" used to describe many-to-one communication systems and "fan-out" for one-to-many.

- **With one writer:**

  This style is most similar to the synchronous Observer [GoF] pattern. You have one privileged object that generates events that others can then receive.

  - *You implicitly know where the event is coming from.* Since there's only one object that can add to the queue, any listener can safely assume that's the sender.

  - *You usually allow multiple readers.* You can have a one-sender-one-receiver queue, but that starts to feel less like the communication system this pattern is about and more like a vanilla queue data structure.

- **With multiple writers:**

  This is how our audio engine example works. Since `playSound()` is a public method, any part of the codebase can add a request to the queue. "Global" or "central" event buses work like this too.

  - *You have to be more careful of cycles.* Since anything can potentially put something onto the queue, it's easier to accidentally enqueue something in the middle of handling an event. If you aren't careful, that may trigger a feedback loop.

  - *You'll likely want some reference to the sender in the event itself.* When a listener gets an event, it doesn't know who sent it, since it could be anyone. If that's something they need to know, you'll want to pack that into the event object so that the listener can use it.

What is the lifetime of the objects in the queue?

With a synchronous notification, execution doesn't return to the sender until all of the receivers have finished processing the message. That means the message itself can safely live in a local variable on the stack. With a queue, the message outlives the call that enqueues it.

If you're using a garbage collected language, you don't need to worry about this too much. Stuff the message in the queue, and it will stick around in memory as long as it's needed. In C or C++, it's up to you to ensure the object lives long enough.

- **Pass ownership:**

  This is the traditional way to do things when managing memory manually. When a message gets queued, the queue claims it and the sender no longer owns it. When it gets processed, the receiver takes ownership and is responsible for deallocating it.

  > In C++, `unique_ptr<T>` gives you these exact semantics out of the box.

- **Share ownership:**

  These days, now that even C++ programmers are more comfortable with garbage collection, shared ownership is more acceptable. With this, the message sticks around as long as anything has a reference to it and is automatically freed when forgotten.

  > Likewise, the C++ type for this is `shared_ptr<T>`.

- **The queue owns it:**

  Another option is to have messages *always* live on the queue. Instead of allocating the message itself, the sender requests a "fresh" one from the queue. The queue returns a

reference to a message already in memory inside the queue, and the sender fills it in. When the message gets processed, the receiver refers to the same message in the queue.

> In other words, the backing store for the queue is an Object Pool.

See Also

- I've mentioned this a few times already, but in many ways, this pattern is the asynchronous cousin to the well-known Observer [GoF] pattern.

- Like many patterns, event queues go by a number of aliases. One established term is "message queue". It's usually referring to a higher-level manifestation. Where our event queues are *within* an application, message queues are usually used for communicating *between* them.

  Another term is "publish/subscribe", sometimes abbreviated to "pubsub". Like "message queue", it usually refers to larger distributed systems unlike the humble coding pattern we're focused on.

- A finite state machine, similar to the Gang of Four's State [GoF] pattern, requires a stream of inputs. If you want it to respond to those asynchronously, it makes sense to queue them.

  When you have a bunch of state machines sending messages to each other, each with a little queue of pending inputs (called a *mailbox*), then you've re-invented the actor model of computation.

- The Go programming language's built-in "channel" type is essentially an event or message queue.

Service Locator

Intent

Provide a global point of access to a service without coupling users to the concrete class that implements it.

Motivation

Some objects or systems in a game tend to get around, visiting almost every corner of the codebase. It's hard to find a part of the game that *won't* need a memory allocator, logging, or random numbers at some point. Systems like those can be thought of as *services* that need to be available to the entire game.

For our example, we'll consider audio. It doesn't have quite the reach of something lower-level like a memory allocator, but it still touches a bunch of game systems. A falling rock hits the ground with a crash (physics). A sniper NPC fires his rifle and a shot rings out (AI). The user selects a menu item with a beep of confirmation (user interface).

Each of these places will need to be able to call into the audio system with something like one of these:

```
// Use a static class?
AudioSystem::playSound(VERY_LOUD_BANG);

// Or maybe a singleton?
AudioSystem::instance()->playSound(VERY_LOUD_BANG);
```

Either gets us where we're trying to go, but we stumbled into some sticky coupling along the way. Every place in the game calling into our audio system directly references the concrete `AudioSystem` class and the mechanism for accessing it—either as a static class or a singleton [GoF].

These call sites, of course, have to be coupled to *something* in order to make a sound play,

but letting them poke at the concrete audio implementation directly is like giving a hundred strangers directions to your house just so they can drop a letter on your doorstep. Not only is it a little bit *too* personal, it's a real pain when you move and you have to tell each person the new directions.

There's a better solution: a phone book. People that need to get in touch with us can look us up by name and get our current address. When we move, we tell the phone company. They update the book, and everyone gets the new address. In fact, we don't even need to give out our real address at all. We can list a P.O. box or some other "representation" of ourselves instead. By having callers go through the book to find us, we have *a convenient single place where we control how we're found.*

This is the Service Locator pattern in a nutshell—it decouples code that needs a service from both *who* it is (the concrete implementation type) and *where* it is (how we get to the instance of it).

The Pattern

A **service** class defines an abstract interface to a set of operations. A concrete **service provider** implements this interface. A separate **service locator** provides access to the service by finding an appropriate provider while hiding both the provider's concrete type and the process used to locate it.

When to Use It

Anytime you make something accessible to every part of your program, you're asking for trouble. That's the main problem with the Singleton [GoF] pattern, and this pattern is no different. My simplest advice for when to use a service locator is: *sparingly*.

Instead of using a global mechanism to give some code access to an object it needs, first consider *passing the object to it instead*. That's dead simple, and it makes the coupling completely obvious. That will cover most of your needs.

But... there are some times when manually passing around an object is gratuitous or actively makes code harder to read. Some systems, like logging or memory management, shouldn't be part of a module's public API. The parameters to your rendering code should have to do with *rendering*, not stuff like logging.

Likewise, other systems represent facilities that are fundamentally singular in nature. Your game probably only has one audio device or display system that it can talk to. It is an ambient property of the environment, so plumbing it through ten layers of methods just so one deeply nested call can get to it is adding needless complexity to your code.

In those kinds of cases, this pattern can help. As we'll see, it functions as a more flexible, more configurable cousin of the Singleton pattern. When used well, it can make your

codebase more flexible with little runtime cost.

> Conversely, when used poorly, it carries with it all of the baggage of the Singleton pattern with worse runtime performance.

Keep in Mind

The core difficulty with a service locator is that it takes a dependency—a bit of coupling between two pieces of code—and defers wiring it up until runtime. This gives you flexibility, but the price you pay is that it's harder to understand what your dependencies are by reading the code.

The service actually has to be located

With a singleton or a static class, there's no chance for the instance we need to *not* be available. Calling code can take for granted that it's there. But since this pattern has to *locate* the service, we may need to handle cases where that fails. Fortunately, we'll cover a strategy later to address this and guarantee that we'll always get *some* service when you need it.

The service doesn't know who is locating it

Since the locator is globally accessible, any code in the game could be requesting a service and then poking at it. This means that the service must be able to work correctly in any circumstance. For example, a class that expects to be used only during the simulation portion of the game loop and not during rendering may not work as a service—it wouldn't be able to ensure that it's being used at the right time. So, if a class expects to be used only in a certain context, it's safest to avoid exposing it to the entire world with this pattern.

Sample Code

Getting back to our audio system problem, let's address it by exposing the system to the rest of the codebase through a service locator.

The service

We'll start off with the audio API. This is the interface that our service will be exposing:

```
class Audio
{
public:
  virtual ~Audio() {}
  virtual void playSound(int soundID) = 0;
  virtual void stopSound(int soundID) = 0;
  virtual void stopAllSounds() = 0;
```

```
};
```

A real audio engine would be much more complex than this, of course, but this shows the basic idea. What's important is that it's an abstract interface class with no implementation bound to it.

The service provider

By itself, our audio interface isn't very useful. We need a concrete implementation. This book isn't about how to write audio code for a game console, so you'll have to imagine there's some actual code in the bodies of these functions, but you get the idea:

```
class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID)
  {
    // Play sound using console audio api...
  }

  virtual void stopSound(int soundID)
  {
    // Stop sound using console audio api...
  }

  virtual void stopAllSounds()
  {
    // Stop all sounds using console audio api...
  }
};
```

Now we have an interface and an implementation. The remaining piece is the service locator—the class that ties the two together.

A simple locator

The implementation here is about the simplest kind of service locator you can define:

```
class Locator
{
public:
  static Audio* getAudio() { return service_; }

  static void provide(Audio* service)
  {
    service_ = service;
  }

private:
```

```
  static Audio* service_;
};
```

The technique this uses is called *dependency injection*, an awkward bit of jargon for a very simple idea. Say you have one class that depends on another. In our case, our `Locator` class needs an instance of the `Audio` service. Normally, the locator would be responsible for constructing that instance itself. Dependency injection instead says that outside code is responsible for *injecting* that dependency into the object that needs it.

The static `getAudio()` function does the locating. We can call it from anywhere in the codebase, and it will give us back an instance of our `Audio` service to use:

```
Audio *audio = Locator::getAudio();
audio->playSound(VERY_LOUD_BANG);
```

The way it "locates" is very simple—it relies on some outside code to register a service provider before anything tries to use the service. When the game is starting up, it calls some code like this:

```
ConsoleAudio *audio = new ConsoleAudio();
Locator::provide(audio);
```

The key part to notice here is that the code that calls `playSound()` isn't aware of the concrete `ConsoleAudio` class; it only knows the abstract `Audio` interface. Equally important, not even the *locator* class is coupled to the concrete service provider. The *only* place in code that knows about the actual concrete class is the initialization code that provides the service.

There's one more level of decoupling here: the `Audio` interface isn't aware of the fact that it's being accessed in most places through a service locator. As far as it knows, it's just a regular abstract base class. This is useful because it means we can apply this pattern to *existing* classes that weren't necessarily designed around it. This is in contrast with Singleton [GoF], which affects the design of the "service" class itself.

A null service

Our implementation so far is certainly simple, and it's pretty flexible too. But it has one big shortcoming: if we try to use the service before a provider has been registered, it returns `NULL`. If the calling code doesn't check that, we're going to crash the game.

I sometimes hear this called "temporal coupling"—two separate pieces of code that must be called in the right order for the program to work correctly. All stateful software has some degree of this, but as with other kinds of coupling, reducing temporal coupling makes the codebase

easier to manage.

Fortunately, there's another design pattern called "Null Object" that we can use to address this. The basic idea is that in places where we would return `NULL` when we fail to find or create an object, we instead return a special object that implements the same interface as the desired object. Its implementation basically does nothing, but it allows code that receives the object to safely continue on as if it had received a "real" one.

To use this, we'll define another "null" service provider:

```
class NullAudio: public Audio
{
public:
  virtual void playSound(int soundID) { /* Do nothing. */ }
  virtual void stopSound(int soundID) { /* Do nothing. */ }
  virtual void stopAllSounds()        { /* Do nothing. */ }
};
```

As you can see, it implements the service interface, but doesn't actually do anything. Now, we change our locator to this:

```
class Locator
{
public:
  static void initialize() { service_ = &nullService_; }

  static Audio& getAudio() { return *service_; }

  static void provide(Audio* service)
  {
    if (service == NULL)
    {
      // Revert to null service.
      service_ = &nullService_;
    }
    else
    {
      service_ = service;
    }
  }

private:
  static Audio* service_;
  static NullAudio nullService_;
};
```

You may notice we're returning the service by reference instead of by pointer now. Since references in C++ are (in theory!) never `NULL`,

returning a reference is a hint to users of the code that they can expect to always get a valid object back.

The other thing to notice is that we're checking for `NULL` in the `provide()` function instead of checking for the accessor. That requires us to call `initialize()` early on to make sure that the locator initially correctly defaults to the null provider. In return, it moves the branch out of `getAudio()`, which will save us a couple of cycles every time the service is accessed.

Calling code will never know that a "real" service wasn't found, nor does it have to worry about handling `NULL`. It's guaranteed to always get back a valid object.

This is also useful for *intentionally* failing to find services. If we want to disable a system temporarily, we now have an easy way to do so: simply don't register a provider for the service, and the locator will default to a null provider.

Turning off audio is handy during development. It frees up some memory and CPU cycles. More importantly, when you break into a debugger just as a loud sound starts playing, it saves you from having your eardrums shredded. There's nothing like twenty milliseconds of a scream sound effect looping at full volume to get your blood flowing in the morning.

Logging decorator

Now that our system is pretty robust, let's discuss another refinement this pattern lets us do—decorated services. I'll explain with an example.

During development, a little logging when interesting events occur can help you figure out what's going on under the hood of your game engine. If you're working on AI, you'd like to know when an entity changes AI states. If you're the sound programmer, you may want a record of every sound as it plays so you can check that they trigger in the right order.

The typical solution is to litter the code with calls to some `log()` function. Unfortunately, that replaces one problem with another—now we have *too much* logging. The AI coder doesn't care when sounds are playing, and the sound person doesn't care about AI state transitions, but now they both have to wade through each other's messages.

Ideally, we would be able to selectively enable logging for just the stuff we care about, and in the final game build, there'd be no logging at all. If the different systems we want to conditionally log are exposed as services, then we can solve this using the Decorator [GoF] pattern. Let's define another audio service provider implementation like this:

```
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)
  {
    // Code to log message...
  }

  Audio &wrapped_;
};
```

As you can see, it wraps another audio provider and exposes the same interface. It forwards the actual audio behavior to the inner provider, but it also logs each sound call. If a programmer wants to enable audio logging, they call this:

```
void enableAudioLogging()
{
  // Decorate the existing service.
  Audio *service = new LoggedAudio(Locator::getAudio());

  // Swap it in.
  Locator::provide(service);
}
```

Now, any calls to the audio service will be logged before continuing as before. And, of course, this plays nicely with our null service, so you can both *disable* audio and yet still log the sounds that it *would* play if sound were enabled.

Design Decisions

We've covered a typical implementation, but there are a couple of ways that it can vary based on differing answers to a few core questions:

How is the service located?

- **Outside code registers it:**

  This is the mechanism our sample code uses to locate the service, and it's the most common design I see in games:

  - *It's fast and simple.* The `getAudio()` function simply returns a pointer. It will often get inlined by the compiler, so we get a nice abstraction layer at almost no performance cost.

  - *We control how the provider is constructed.* Consider a service for accessing the game's controllers. We have two concrete providers: one for regular games and one for playing online. The online provider passes controller input over the network so that, to the rest of the game, remote players appear to be using local controllers.

    To make this work, the online concrete provider needs to know the IP address of the other remote player. If the locator itself was constructing the object, how would it know what to pass in? The `Locator` class doesn't know anything about online at all, much less some other user's IP address.

    Externally registered providers dodge the problem. Instead of the locator constructing the class, the game's networking code instantiates the online-specific service provider, passing in the IP address it needs. Then it gives that to the locator, who knows only about the service's abstract interface.

  - *We can change the service while the game is running.* We may not use this in the final game, but it's a neat trick during development. While testing, we can swap out, for example, the audio service with the null service we talked about earlier to temporarily disable sound while the game is still running.

  - *The locator depends on outside code.* This is the downside. Any code accessing the service presumes that some code somewhere has already registered it. If that initialization doesn't happen, we'll either crash or have a service mysteriously not working.

- **Bind to it at compile time:**

  The idea here is that the "location" process actually occurs at compile time using preprocessor macros. Like so:

```
class Locator
{
public:
  static Audio& getAudio() { return service_; }

private:
  #if DEBUG
    static DebugAudio service_;
  #else
    static ReleaseAudio service_;
  #endif
};
```

Locating the service like this implies a few things:

- *It's fast.* Since all of the real work is done at compile time, there's nothing left to do at runtime. The compiler will likely inline the `getAudio()` call, giving us a solution that's as fast as we could hope for.
- *You can guarantee the service is available.* Since the locator owns the service now and selects it at compile time, we can be assured that if the game compiles, we won't have to worry about the service being unavailable.
- *You can't change the service easily*. This is the major downside. Since the binding happens at build time, anytime you want to change the service, you've got to recompile and restart the game.

- **Configure it at runtime:**

  Over in the khaki-clad land of enterprise business software, if you say "service locator", this is what they'll have in mind. When the service is requested, the locator does some magic at runtime to hunt down the actual implementation requested.

  > *Reflection* is a capability of some programming languages to interact with the type system at runtime. For example, we could find a class with a given name, find its constructor, and then invoke it to create an instance.
  >
  > Dynamically typed languages like Lisp, Smalltalk, and Python get this by their very nature, but newer static languages like C# and Java also support it.

  Typically, this means loading a configuration file that identifies the provider and then using reflection to instantiate that class at runtime. This does a few things for us:

  - *We can swap out the service without recompiling.* This is a little more flexible than a compile-time-bound service, but not quite as flexible as a registered one where you can actually change the service while the game is running.

- *Non-programmers can change the service.* This is nice for when the designers want to be able to turn certain game features on and off but aren't comfortable mucking through source code. (Or, more likely, the *coders* aren't comfortable with them mucking through it.)

- *The same codebase can support multiple configurations simultaneously.* Since the location process has been moved out of the codebase entirely, we can use the same code to support multiple service configurations simultaneously.

  This is one of the reasons this model is appealing over in enterprise web-land: you can deploy a single app that works on different server setups just by changing some configs. Historically, this was less useful in games since console hardware is pretty well-standardized, but as more games target a heaping hodgepodge of mobile devices, this is becoming more relevant.

- *It's complex.* Unlike the previous solutions, this one is pretty heavyweight. You have to create some configuration system, possibly write code to load and parse a file, and generally *do some stuff* to locate the service. Time spent writing this code is time not spent on other game features.

- *Locating the service takes time.* And now the smiles really turn to frowns. Going with runtime configuration means you're burning some CPU cycles locating the service. Caching can minimize this, but that still implies that the first time you use the service, the game's got to go off and spend some time hunting it down. Game developers *hate* burning CPU cycles on something that doesn't improve the player's game experience.

What happens if the service can't be located?

- **Let the user handle it:**

  The simplest solution is to pass the buck. If the locator can't find the service, it just returns `NULL`. This implies:

  - *It lets users determine how to handle failure.* Some users may consider failing to find a service a critical error that should halt the game. Others may be able to safely ignore it and continue. If the locator can't define a blanket policy that's correct for all cases, then passing the failure down the line lets each call site decide for itself what the right response is.

  - *Users of the service must handle the failure.* Of course, the corollary to this is that each call site *must* check for failure to find the service. If almost all of them handle failure the same way, that's a lot duplicate code spread throughout the codebase. If just one of the potentially hundreds of places that use the service fails to make that check, our game is going to crash.

- **Halt the game:**

  I said that we can't *prove* that the service will always be available at compile-time, but that doesn't mean we can't *declare* that availability is part of the runtime contract of the locator. The simplest way to do this is with an assertion:

```
class Locator
{
public:
  static Audio& getAudio()
  {
    Audio* service = NULL;

    // Code here to locate service...

    assert(service != NULL);
    return *service;
  }
};
```

  If the service isn't located, the game stops before any subsequent code tries to use it. The `assert()` call there doesn't solve the problem of failing to locate the service, but it does make it clear whose problem it is. By asserting here, we say, "Failing to locate a service is a bug in the locator."

> The Singleton chapter explains the `assert()` function if you've never seen it before.

  So what does this do for us?

  - *Users don't need to handle a missing service.* Since a single service may be used in hundreds of places, this can be a significant code saving. By declaring it the locator's job to always provide a service, we spare the users of the service from having to pick up that slack.

  - *The game is going to halt if the service can't be found.* On the off chance that a service really can't be found, the game is going to halt. This is good in that it forces us to address the bug that's preventing the service from being located (likely some initialization code isn't being called when it should), but it's a real drag for everyone else who's blocked until it's fixed. With a large dev team, you can incur some painful programmer downtime when something like this breaks.

- **Return a null service:**

  We showed this refinement in our sample implementation. Using this means:

  - *Users don't need to handle a missing service.* Just like the previous option, we

ensure that a valid service object will always be returned, simplifying code that uses the service.

- *The game will continue if the service isn't available.* This is both a boon and a curse. It's helpful in that it lets us keep running the game even when a service isn't there. This can be really helpful on a large team when a feature we're working on may be dependent on some other system that isn't in place yet.

  The downside is that it may be harder to debug an *unintentionally* missing service. Say the game uses a service to access some data and then make a decision based on it. If we've failed to register the real service and that code gets a null service instead, the game may not behave how we want. It will take some work to trace that issue back to the fact that a service wasn't there when we thought it would be.

  > We can alleviate this by having the null service print some debug output whenever it's used.

Among these options, the one I see used most frequently is simply asserting that the service will be found. By the time a game gets out the door, it's been very heavily tested, and it will likely be run on a reliable piece of hardware. The chances of a service failing to be found by then are pretty slim.

On a larger team, I encourage you to throw a null service in. It doesn't take much effort to implement, and can spare you from some downtime during development when a service isn't available. It also gives you an easy way to turn off a service if it's buggy or is just distracting you from what you're working on.

What is the scope of the service?

Up to this point, we've assumed that the locator will provide access to the service to *anyone* who wants it. While this is the typical way the pattern is used, another option is to limit access to a single class and its descendants, like so:

```
class Base
{
  // Code to locate service and set service_...

protected:
  // Derived classes can use service
  static Audio& getAudio() { return *service_; }

private:
  static Audio* service_;
};
```

With this, access to the service is restricted to classes that inherit `Base`. There are

advantages either way:

- **If access is global:**
  - *It encourages the entire codebase to all use the same service.* Most services are intended to be singular. By allowing the entire codebase to have access to the same service, we can avoid random places in code instantiating their own providers because they can't get to the "real" one.
  - *We lose control over where and when the service is used.* This is the obvious cost of making something global—anything can get to it. The Singleton [GoF] chapter has a full cast of characters for the horror show that global scope can spawn.
- **If access is restricted to a class:**
  - *We control coupling.* This is the main advantage. By limiting a service to a branch of the inheritance tree, we can make sure systems that should be decoupled stay decoupled.
  - *It can lead to duplicate effort.* The potential downside is that if a couple of unrelated classes *do* need access to the service, they'll each need to have their own reference to it. Whatever process is used to locate or register the service will have to be duplicated between those classes.

    (The other option is to change the class hierarchy around to give those classes a common base class, but that's probably more trouble than it's worth.)

My general guideline is that if the service is restricted to a single domain in the game, then limit its scope to a class. For example, a service for getting access to the network can probably be limited to online classes. Services that get used more widely like logging should be global.

See Also

- The Service Locator pattern is a sibling to Singleton [GoF] in many ways, so it's worth looking at both to see which is most appropriate for your needs.
- The Unity framework uses this pattern in concert with the Component pattern in its `GetComponent()` method.
- Microsoft's XNA framework for game development has this pattern built into its core `Game` class. Each instance has a `GameServices` object that can be used to register and locate services of any type.

Optimization Patterns

Game Programming Patterns

While the rising tide of faster and faster hardware has lifted most software above worrying about performance, games are one of the few remaining exceptions. Players always want richer, more realistic and exciting experiences. Screens are crowded with games vying for a player's attention—and cash!—and the game that pushes the hardware the furthest often wins.

Optimizing for performance is a deep art that touches all aspects of software. Low-level coders master the myriad idiosyncrasies of hardware architectures. Meanwhile, algorithms researchers compete to prove mathematically whose procedure is the most efficient.

Here, I touch on a few mid-level patterns that are often used to speed up a game. Data Locality introduces you to the modern computer's memory hierarchy and how you can use it to your advantage. The Dirty Flag pattern helps you avoid unnecessary computation while Object Pools help you avoid unnecessary allocation. Spatial Partitioning speeds up the virtual world and its inhabitants' arrangement in space.

The Patterns

- Data Locality
- Dirty Flag
- Object Pool
- Spatial Partition

Data Locality

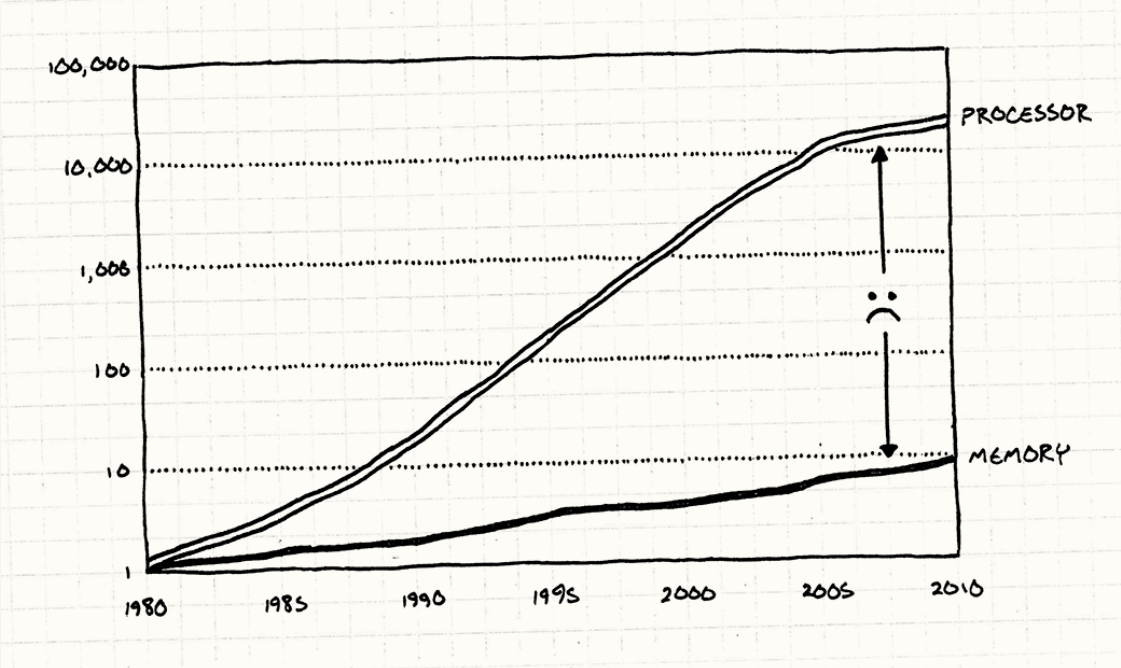
Intent

Accelerate memory access by arranging data to take advantage of CPU caching.

Motivation

We've been lied to. They keep showing us charts where CPU speed goes up and up every year as if Moore's Law isn't just a historical observation but some kind of divine right. Without lifting a finger, we software folks watch our programs magically accelerate just by virtue of new hardware.

Chips *have* been getting faster (though even that's plateauing now), but the hardware heads failed to mention something. Sure, we can *process* data faster than ever, but we can't *get* that data faster.



Processor and RAM speed relative to their respective speeds in 1980. As you can see, CPUs have grown in leaps and bounds, but RAM access is lagging far behind.

Data for this is from *Computer Architecture: A Quantitative Approach* by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau by way of Tony Albrecht's "Pitfalls of Object-Oriented Programming".

For your super-fast CPU to blow through a ream of calculations, it actually has to get the data out of main memory and into registers. As you can see, RAM hasn't been keeping up with increasing CPU speeds. Not even close.

With today's hardware, it can take *hundreds* of cycles to fetch a byte of data from RAM. If most instructions need data, and it takes hundreds of cycles to get it, how is it that our CPUs aren't sitting idle 99% of the time waiting for data?

Actually, they *are* stuck waiting on memory an astonishingly large fraction of time these days, but it's not as bad as it could be. To explain how, let's take a trip to the Land of Overly Long Analogies...

It's called "random access memory" because, unlike disc drives, you can theoretically access any piece of it as quick as any other. You don't have to worry about reading things consecutively like you do a disc.

Or, at least, you *didn't*. As we'll see, RAM isn't so random access anymore.

A data warehouse

Imagine you're an accountant in a tiny little office. Your job is to request a box of papers and then do some accountant-y stuff with them—add up a bunch of numbers or something. You must do this for specific labeled boxes according to some arcane logic that only makes sense to other accountants.

I probably shouldn't have used a job I know absolutely nothing about in this analogy.

Thanks to a mixture of hard work, natural aptitude, and stimulants, you can finish an entire box in, say, a minute. There's a little problem, though. All of those boxes are stored in a warehouse in a separate building. To get a box, you have to ask the warehouse guy to bring it to you. He goes and gets a forklift and drives around the aisles until he finds the box you want.

It takes him, seriously, an entire day to do this. Unlike you, he's not getting employee of the month any time soon. This means that no matter how fast you are, you only get one box a day. The rest of the time, you just sit there and question the life decisions that led to

this soul-sucking job.

One day, a group of industrial designers shows up. Their job is to improve the efficiency of operations—things like making assembly lines go faster. After watching you work for a few days, they notice a few things:

- Pretty often, when you're done with one box, the next box you request is right next to it on the same shelf in the warehouse.
- Using a forklift to carry a single box of papers is pretty dumb.
- There's actually a little bit of spare room in the corner of your office.

> The technical term for using something near the thing you just used is *locality of reference*.

They come up with a clever fix. Whenever you request a box from the warehouse guy, he'll grab an entire pallet of them. He gets the box you want and then some more boxes that are next to it. He doesn't know if you want those (and, given his work ethic, clearly doesn't care); he simply takes as many as he can fit on the pallet.

He loads the whole pallet and brings it to you. Disregarding concerns for workplace safety, he drives the forklift right in and drops the pallet in the corner of your office.

When you need a new box, now, the first thing you do is see if it's already on the pallet in your office. If it is, great! It only takes you a second to grab it and get back to crunching numbers. If a pallet holds fifty boxes and you got lucky and *all* of the boxes you need happen to be on it, you can churn through fifty times more work than you could before.

But if you need a box that's *not* on the pallet, you're back to square one. Since you can only fit one pallet in your office, your warehouse friend will have to take that one back and then bring you an entirely new one.

A pallet for your CPU

Strangely enough, this is similar to how CPUs in modern computers work. In case it isn't obvious, you play the role of the CPU. Your desk is the CPU's registers, and the box of papers is the data you can fit in them. The warehouse is your machine's RAM, and that annoying warehouse guy is the bus that pulls data from main memory into registers.

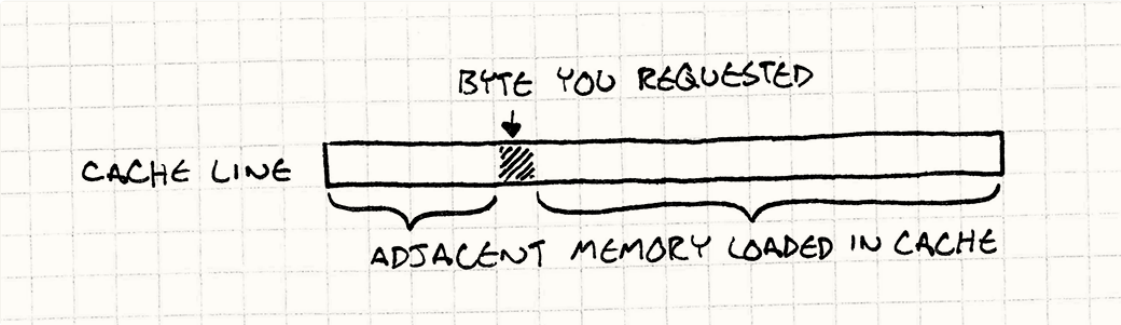
If I were writing this chapter thirty years ago, the analogy would stop there. But as chips got faster and RAM, well, *didn't*, hardware engineers started looking for solutions. What they came up with was *CPU caching*.

Modern computers have a little chunk of memory right inside the chip. The CPU can pull data from this much faster than it can from main memory. It's small because it has to fit in the chip and because the faster type of memory it uses (static RAM or "SRAM") is way

more expensive.

> Modern hardware has multiple levels of caching, which is what they mean when you hear "L1", "L2", "L3", etc. Each level is larger but slower than the previous. For this chapter, we won't worry about the fact that memory is actually a hierarchy, but it's important to know.

This little chunk of memory is called a *cache* (in particular, the chunk on the chip is your *L1 cache*), and in my belabored analogy, its part was played by the pallet of boxes. Whenever your chip needs a byte of data from RAM, it automatically grabs a whole chunk of contiguous memory—usually around 64 to 128 bytes—and puts it in the cache. This dollop of memory is called a *cache line*.



If the next byte of data you need happens to be in that chunk, the CPU reads it straight from the cache, which is *much* faster than hitting RAM. Successfully finding a piece of data in the cache is called a *cache hit*. If it can't find it in there and has to go to main memory, that's a *cache miss*.

> I glossed over (at least) one detail in the analogy. In your office, there was only room for one pallet, or one cache line. A real cache contains a number of cache lines. The details about how those work is out of scope here, but search for "cache associativity" to feed your brain.

When a cache miss occurs, the CPU *stalls*—it can't process the next instruction because it needs data. It sits there, bored out of its mind for a few hundred cycles until the fetch completes. Our mission is to avoid that. Imagine you're trying to optimize some performance-critical piece of game code and it looks like this:

```
for (int i = 0; i < NUM_THINGS; i++)
{
  sleepFor500Cycles();
  things[i].doStuff();
}
```

What's the first change you're going to make to that code? Right. Take out that pointless, expensive function call. That call is equivalent to the performance cost of a cache miss. Every time you bounce to main memory, it's like you put a delay in your code.

Wait, data is performance?

When I started working on this chapter, I spent some time putting together little game-like programs that would trigger best case and worst case cache usage. I wanted benchmarks that would thrash the cache so I could see first-hand how much bloodshed it causes.

When I got some stuff working, I was surprised. I knew it was a big deal, but there's nothing quite like seeing it with your own eyes. I wrote two programs that did the *exact same* computation. The only difference was how many cache misses they caused. The slow one was *fifty times* slower than the other.

> There are a lot of caveats here. In particular, different computers have different cache setups, so my machine may be different from yours, and dedicated game consoles are very different from PCs, which are quite different from mobile devices.
>
> Your mileage will vary.

This was a real eye-opener to me. I'm used to thinking of performance being an aspect of *code*, not *data*. A byte isn't slow or fast, it's just some static thing sitting there. But because of caching, *the way you organize data directly impacts performance*.

The challenge now is to wrap that up into something that fits into a chapter here. Optimization for cache usage is a huge topic. I haven't even touched on *instruction caching*. Remember, code is in memory too and has to be loaded onto the CPU before it can be executed. Someone more versed on the subject could write an entire book on it.

> In fact, someone *did* write a book on it: *Data-Oriented Design*, by Richard Fabian.

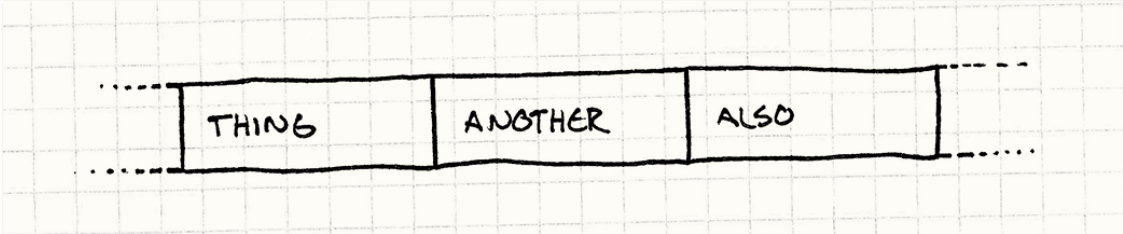
Since you're already reading *this* book right now, though, I have a few basic techniques that will get you started along the path of thinking about how data structures impact your performance.

It all boils down to something pretty simple: whenever the chip reads some memory, it gets a whole cache line. The more you can use stuff in that cache line, the faster you go. So the goal then is to *organize your data structures so that the things you're processing are next to each other in memory*.

> There's a key assumption here, though: one thread. If you are modifying nearby data on multiple threads, it's faster to have it on *different* cache lines. If two threads try to tweak data on the same cache line, both cores have to do some costly synchronization of their caches.

In other words, if your code is crunching on `Thing`, then `Another`, then `Also`, you want

them laid out in memory like this:



Note, these aren't *pointers* to `Thing`, `Another`, and `Also`. This is the actual data for them, in place, lined up one after the other. As soon as the CPU reads in `Thing`, it will start to get `Another` and `Also` too (depending on how big they are and how big a cache line is). When you start working on them next, they'll already be cached. Your chip is happy, and you're happy.

The Pattern

Modern CPUs have **caches to speed up memory access**. These can access memory **adjacent to recently accessed memory much quicker**. Take advantage of that to improve performance by **increasing data locality**—keeping data in **contiguous memory in the order that you process it**.

When to Use It

Like most optimizations, the first guideline for using the Data Locality pattern is *when you have a performance problem*. Don't waste time applying this to some infrequently executed corner of your codebase. Optimizing code that doesn't need it just makes your life harder since the result is almost always more complex and less flexible.

With this pattern specifically, you'll also want to be sure your performance problems *are caused by cache misses*. If your code is slow for other reasons, this won't help.

The cheap way to profile is to manually add a bit of instrumentation that checks how much time has elapsed between two points in the code, hopefully using a precise timer. To catch poor cache usage, you'll want something a little more sophisticated. You really want to see how many cache misses are occurring and where.

Fortunately, there are profilers out there that report this. It's worth spending the time to get one of these working and make sure you understand the (surprisingly complex) numbers it throws at you before you do major surgery on your data structures.

> Unfortunately, most of those tools aren't cheap. If you're on a console dev team, you probably already have licenses for them.
>
> If not, an excellent free option is Cachegrind. It runs your program on top of a simulated CPU and cache hierarchy and then reports all of the

cache interactions.

That being said, cache misses *will* affect the performance of your game. While you shouldn't spend a ton of time pre-emptively optimizing for cache usage, do think about how cache-friendly your data structures are throughout the design process.

Keep in Mind

One of the hallmarks of software architecture is *abstraction*. A large chunk of this book is about patterns to decouple pieces of code from each other so that they can be changed more easily. In object-oriented languages, this almost always means interfaces.

In C++, using interfaces implies accessing objects through pointers or references. But going through a pointer means hopping across memory, which leads to the cache misses this pattern works to avoid.

> The other half of interfaces is *virtual method calls*. Those require the CPU to look up an object's vtable and then find the pointer to the actual method to call there. So, again, you're chasing pointers, which can cause cache misses.

In order to please this pattern, you will have to sacrifice some of your precious abstractions. The more you design your program around data locality, the more you will have to give up inheritance, interfaces, and the benefits those tools can provide. There's no silver bullet here, only challenging trade-offs. That's what makes it fun!

Sample Code

If you really go down the rathole of optimizing for data locality, you'll discover countless ways to slice and dice your data structures into pieces your CPU can most easily digest. To get you started, I'll show an example for each of a few of the most common ways to organize your data. We'll cover them in the context of some specific part of a game engine, but (as with other patterns), keep in mind that the general technique can be applied anywhere it fits.

Contiguous arrays

Let's start with a game loop that processes a bunch of game entities. Those entities are decomposed into different domains—AI, physics, and rendering—using the Component pattern. Here's the `GameEntity` class:

```
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_;
};
```

Each component has a relatively small amount of state, maybe little more than a few vectors or a matrix, and then a method to update it. The details aren't important here, but imagine something roughly along the lines of:

As the name implies, these are examples of the Update Method pattern. Even `render()` is this pattern, just by another name.

```
class AIComponent
{
public:
  void update() { /* Work with and modify state... */ }

private:
  // Goals, mood, etc. ...
};

class PhysicsComponent
{
public:
  void update() { /* Work with and modify state... */ }

private:
  // Rigid body, velocity, mass, etc. ...
};

class RenderComponent
{
public:
  void render() { /* Work with and modify state... */ }

private:
  // Mesh, textures, shaders, etc. ...
};
```

The game maintains a big array of pointers to all of the entities in the world. Each spin of the game loop, we need to run the following:

1. Update the AI components for all of the entities.
2. Update the physics components for them.
3. Render them using their render components.

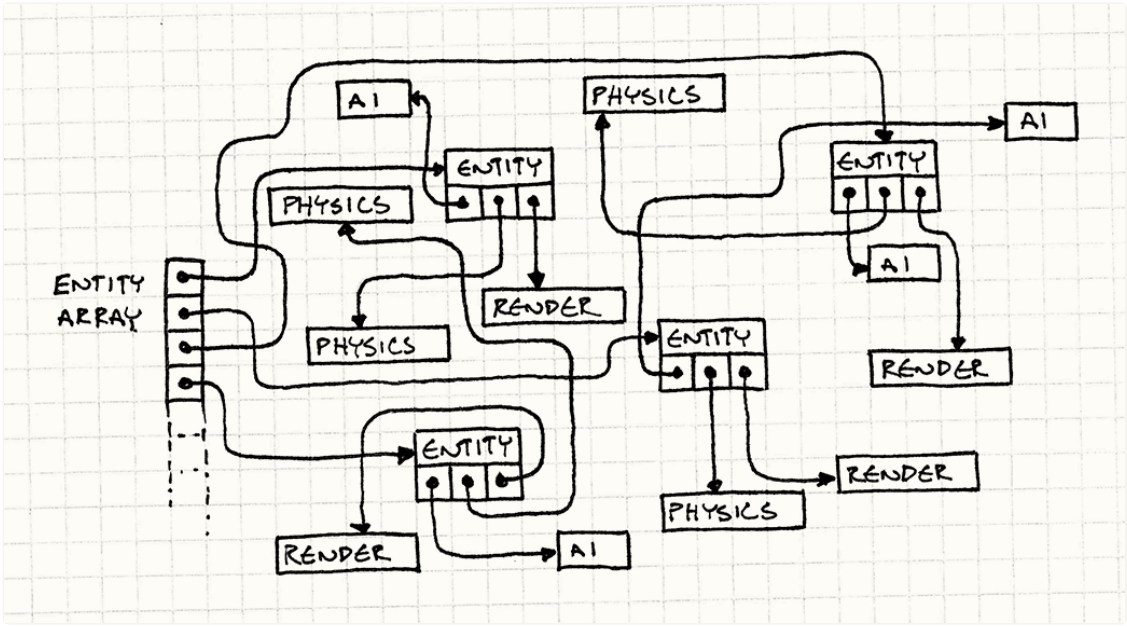
Lots of game engines implement that like so:

```
while (!gameOver)
{
  // Process AI.
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->ai()->update();
  }

  // Update physics.
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->physics()->update();
  }

  // Draw to screen.
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->render()->render();
  }

  // Other game loop machinery for timing...
}
```

Before you ever heard of a CPU cache, this looked totally innocuous. But by now, you've got an inkling that something isn't right here. This code isn't just thrashing the cache, it's taking it around back and beating it to a pulp. Watch what it's doing:

1. The array of game entities is storing *pointers* to them, so for each element in the array, we have to traverse that pointer. That's a cache miss.
2. Then the game entity has a pointer to the component. Another cache miss.
3. Then we update the component.
4. Now we go back to step one for *every component of every entity in the game*.

The scary part is that we have no idea how these objects are laid out in memory. We're completely at the mercy of the memory manager. As entities get allocated and freed over time, the heap is likely to become increasingly randomly organized.



> Every frame, the game loop has to follow all of those arrows to get to the data it cares about.

If our goal was to take a whirlwind tour around the game's address space like some "256MB of RAM in Four Nights!" cheap vacation package, this would be a fantastic deal. But our goal is to run the game quickly, and traipsing all over main memory is *not* the way to do that. Remember that `sleepFor500Cycles()` function? Well this code is effectively calling that *all the time*.

> The term for wasting a bunch of time traversing pointers is "pointer chasing", which it turns out is nowhere near as fun as it sounds.

Let's do something better. Our first observation is that the only reason we follow a pointer to get to the game entity is so we can immediately follow *another* pointer to get to a component. `GameEntity` itself has no interesting state and no useful methods. The *components* are what the game loop cares about.

Instead of a giant constellation of game entities and components scattered across the inky darkness of address space, we're going to get back down to Earth. We'll have a big array for each type of component: a flat array of AI components, another for physics, and another for rendering.

Like this:

```
AIComponent* aiComponents =
    new AIComponent[MAX_ENTITIES];
PhysicsComponent* physicsComponents =
    new PhysicsComponent[MAX_ENTITIES];
RenderComponent* renderComponents =
    new RenderComponent[MAX_ENTITIES];
```

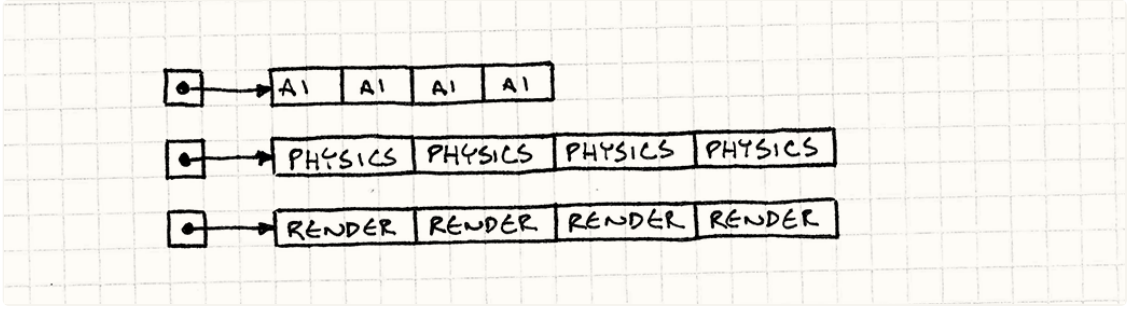
My least favorite part about using components is how long the word "component" is.

Let me stress that these are arrays of *components* and not *pointers to components*. The data is all there, one byte after the other. The game loop can then walk these directly:

```
while (!gameOver)
{
  // Process AI.
  for (int i = 0; i < numEntities; i++)
  {
    aiComponents[i].update();
  }

  // Update physics.
  for (int i = 0; i < numEntities; i++)
  {
    physicsComponents[i].update();
  }

  // Draw to screen.
  for (int i = 0; i < numEntities; i++)
  {
    renderComponents[i].render();
  }

  // Other game loop machinery for timing...
}
```

One hint that we're doing better here is how few -> operators there are in the new code. If you want to improve data locality, look for indirection operators you can get rid of.

We've ditched all of that pointer chasing. Instead of skipping around in memory, we're doing a straight crawl through three contiguous arrays.



This pumps a solid stream of bytes right into the hungry maw of the CPU. In my testing, this change made the update loop *fifty times* faster than the previous version.

Interestingly, we haven't lost much encapsulation here. Sure, the game loop is updating the components directly instead of going through the game entities, but it was doing that before to ensure they were processed in the right order. Even so, each component itself is still nicely encapsulated. It owns its own data and methods. We simply changed the way it's used.

This doesn't mean we need to get rid of `GameEntity` either. We can leave it as it is with pointers to its components. They'll just point into those arrays. This is still useful for other parts of the game where you want to pass around a conceptual "game entity" and everything that goes with it. The important part is that the performance-critical game loop sidesteps that and goes straight to the data.

Packed data

Say we're doing a particle system. Following the advice of the previous section, we've got all of our particles in a nice big contiguous array. Let's wrap it in a little manager class too:

> The `ParticleSystem` class is an example of an Object Pool custom built for a single type of object.

```
class Particle
{
public:
  void update() { /* Gravity, etc. ... */ }
  // Position, velocity, etc. ...
};

class ParticleSystem
{
public:
  ParticleSystem()
  : numParticles_(0)
  {}

  void update();
private:
  static const int MAX_PARTICLES = 100000;

  int numParticles_;
  Particle particles_[MAX_PARTICLES];
};
```

A rudimentary update method for the system just looks like this:

```
void ParticleSystem::update()
{
  for (int i = 0; i < numParticles_; i++)
```

```
  {
    particles_[i].update();
  }
}
```

But it turns out that we don't actually need to process *all* of the particles all the time. The particle system has a fixed-size pool of objects, but they aren't usually all actively twinkling across the screen. The easy answer is something like this:

```
for (int i = 0; i < numParticles_; i++)
{
  if (particles_[i].isActive())
  {
    particles_[i].update();
  }
}
```

We give `Particle` a flag to track whether its in use or not. In the update loop, we check that for each particle. That loads the flag into the cache along with all of that particle's other data. If the particle *isn't* active, then we skip over it to the next one. The rest of the particle's data that we loaded into the cache is a waste.

The fewer active particles there are, the more we're skipping across memory. The more we do that, the more cache misses there are between actually doing useful work updating active particles. If the array is large and has *lots* of inactive particles in it, we're back to thrashing the cache again.

Having objects in a contiguous array doesn't solve much if the objects we're actually processing aren't contiguous in it. If it's littered with inactive objects we have to dance around, we're right back to the original problem.

Savvy low-level coders can see another problem here. Doing an `if` check for every particle can cause a *branch misprediction* and a *pipeline stall*. In modern CPUs, a single "instruction" actually takes several clock cycles. To keep the CPU busy, instructions are *pipelined* so that the subsequent instructions start processing before the first one finishes.

To do that, the CPU has to guess which instructions it will be executing next. In straight line code, that's easy, but with control flow, it's harder. While it's executing the instructions for that `if`, does it guess that the particle is active and start executing the code for the `update()` call, or does it guess that it isn't?

To answer that, the chip does *branch prediction*—it sees which branches your code previously took and guesses that it will do that again. But when the loop is constantly toggling between particles that are and aren't active, that prediction fails.

When it does, the CPU has to ditch the instructions it had started speculatively processing (a *pipeline flush*) and start over. The performance impact of this varies widely by machine, but this is why you sometimes see developers avoid flow control in hot code.

Given the title of this section, you can probably guess the answer. Instead of *checking* the active flag, we'll *sort* by it. We'll keep all of the active particles in the front of the list. If we know all of those particles are active, we don't have to check the flag at all.

We can also easily keep track of how many active particles there are. With this, our update loop turns into this thing of beauty:

```
for (int i = 0; i < numActive_; i++)
{
  particles[i].update();
}
```

Now we aren't skipping over *any* data. Every byte that gets sucked into the cache is a piece of an active particle that we actually need to process.

Of course, I'm not saying you should quicksort the entire collection of particles every frame. That would more than eliminate the gains here. What we want to do is *keep* the array sorted.

Assuming the array is already sorted—and it is at first when all particles are inactive—the only time it can become *un*sorted is when a particle has been activated or deactivated. We can handle those two cases pretty easily. When a particle gets activated, we move it up to the end of the active particles by swapping it with the first *in*active one:

```
void ParticleSystem::activateParticle(int index)
{
  // Shouldn't already be active!
  assert(index >= numActive_);

  // Swap it with the first inactive particle
  // right after the active ones.
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;

  // Now there's one more.
  numActive_++;
}
```

To deactivate a particle, we just do the opposite:

```
void ParticleSystem::deactivateParticle(int index)
```

```
  {
    // Shouldn't already be inactive!
    assert(index < numActive_);

    // There's one fewer.
    numActive_--;

    // Swap it with the last active particle
    // right before the inactive ones.
    Particle temp = particles_[numActive_];
    particles_[numActive_] = particles_[index];
    particles_[index] = temp;
  }
```

Lots of programmers (myself included) have developed allergies to moving things around in memory. Schlepping a bunch of bytes around *feels* heavyweight compared to assigning a pointer. But when you add in the cost of *traversing* that pointer, it turns out that our intuition is sometimes wrong. In some cases, it's cheaper to push things around in memory if it helps you keep the cache full.

> This is your friendly reminder to *profile* when making these kinds of decisions.

There's a neat consequence of keeping the particles *sorted* by their active state—we don't need to store an active flag in each particle at all. It can be inferred by its position in the array and the `numActive_` counter. This makes our particle objects smaller, which means we can pack more in our cache lines, and that makes them even faster.

It's not all rosy, though. As you can see from the API, we've lost a bit of object orientation here. The `Particle` class no longer controls its own active state. You can't call some `activate()` method on it since it doesn't know its index. Instead, any code that wants to activate particles needs access to the particle *system*.

In this case, I'm OK with `ParticleSystem` and `Particle` being tightly tied like this. I think of them as a single *concept* spread across two physical *classes*. It just means accepting the idea that particles are *only* meaningful in the context of some particle system. Also, in this case it's likely to be the particle system that will be spawning and killing particles anyway.

Hot/cold splitting

OK, this is the last example of a simple technique for making your cache happier. Say we've got an AI component for some game entity. It has some state in it—the animation it's currently playing, a goal position it's heading towards, energy level, etc.—stuff it checks and tweaks every single frame. Something like:

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;
};
```

But it also has some state for rarer eventualities. It stores some data describing what loot it drops when it has an unfortunate encounter with the noisy end of a shotgun. That drop data is only used once in the entity's lifetime, right at its bitter end:

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  // Previous fields...
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

Assuming we followed the earlier patterns, when we update these AI components, we walk through a nice packed, contiguous array of data. But that data includes all of the loot drop information. That makes each component bigger, which reduces the number of components we can fit in a cache line. We get more cache misses because the total memory we walk over is larger. The loot data gets pulled into the cache for every component in every frame, even though we aren't even touching it.

The solution for this is called "hot/cold splitting". The idea is to break our data structure into two separate pieces. The first holds the "hot" data, the state we need to touch every frame. The other piece is the "cold" data, everything else that gets used less frequently.

The hot piece is the *main* AI component. It's the one we need to use the most, so we don't want to chase a pointer to find it. The cold component can be off to the side, but we still need to get to it, so we give the hot component a pointer to it, like so:

```
class AIComponent
{
public:
  // Methods...
private:
```

```
  Animation* animation_;
  double energy_;
  Vector goalPos_;

  LootDrop* loot_;
};

class LootDrop
{
  friend class AIComponent;
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

Now when we're walking the AI components every frame, the only data that gets loaded into the cache is stuff we are actually processing (with the exception of that one little pointer to the cold data).

> We could conceivably ditch the pointer too by having parallel arrays for the hot and cold components. Then we can find the cold AI data for a component since both pieces will be at the same index in their respective arrays.

You can see how this starts to get fuzzy, though. In my example here, it's pretty obvious which data should be hot and cold, but in a real game it's rarely so clear-cut. What if you have fields that are used when an entity is in a certain mode but not in others? What if entities use a certain chunk of data only when they're in certain parts of the level?

Doing this kind of optimization is somewhere between a black art and a rathole. It's easy to get sucked in and spend endless time pushing data around to see what speed difference it makes. It will take practice to get a handle on where to spend your effort.

Design Decisions

This pattern is really about a mindset—it's getting you to think about your data's arrangement in memory as a key part of your game's performance story. The actual concrete design space is wide open. You can let data locality affect your whole architecture, or maybe it's just a localized pattern you apply to a few core data structures.

The biggest questions you'll need to answer are when and where you apply this pattern, but here are a couple of others that may come up.

> Noel Llopis' famous article that got a lot more people thinking about designing games around cache usage calls this "data-oriented design".

How do you handle polymorphism?

Up to this point, we've avoided subclassing and virtual methods. We have assumed we have nice packed arrays of *homogenous* objects. That way, we know they're all the exact same size. But polymorphism and dynamic dispatch are useful tools too. How do we reconcile this?

- **Don't:**

  The simplest answer is to avoid subclassing, or at least avoid it in places where you're optimizing for cache usage. Software engineer culture is drifting away from heavy use of inheritance anyway.

  > One way to keep much of the flexibility of polymorphism without using subclassing is through the Type Object pattern.

  - *It's safe and easy.* You know exactly what class you're dealing with, and all objects are obviously the same size.
  - *It's faster.* Dynamic dispatch means looking up the method in the vtable and then traversing that pointer to get to the actual code. While the cost of this varies widely across different hardware, there is *some* cost to dynamic dispatch.

    > As usual, the only absolute is that there are no absolutes. In most cases, a C++ compiler will require an indirection for a virtual method call. But in *some* cases, the compiler may be able to do *devirtualization* and statically call the right method if it knows what concrete type the receiver is. Devirtualization is more common in just-in-time compilers for languages like Java and JavaScript.

  - *It's inflexible.* Of course, the reason we use dynamic dispatch is because it gives us a powerful way to vary behavior between objects. If you want different entities in your game to have their own rendering styles or their own special moves and attacks, virtual methods are a proven way to model that. Having to instead stuff all of that code into a single non-virtual method that does something like a big `switch` gets messy quickly.

- **Use separate arrays for each type:**

  We use polymorphism so that we can invoke behavior on an object whose type we don't know. In other words, we have a mixed bag of stuff, and we want each object in there to do its own thing when we tell it to go.

  But that raises the question of why mix the bag to begin with? Instead, why not maintain separate, homogenous collections for each type?

- *It keeps objects tightly packed*. Since each array only contains objects of one class, there's no padding or other weirdness.
- *You can statically dispatch*. Once you've got objects partitioned by type, you don't need polymorphism at all any more. You can use regular, non-virtual method calls.
- *You have to keep track of a bunch of collections*. If you have a lot of different object types, the overhead and complexity of maintaining separate arrays for each can be a chore.
- *You have to be aware of every type*. Since you have to maintain separate collections for each type, you can't be decoupled from the *set* of classes. Part of the magic of polymorphism is that it's *open-ended*—code that works with an interface can be completely decoupled from the potentially large set of types that implement that interface.

- **Use a collection of pointers:**

  If you weren't worried about caching, this is the natural solution. Just have an array of pointers to some base class or interface type. You get all the polymorphism you could want, and objects can be whatever size they want.

  - *It's flexible*. The code that consumes the collection can work with objects of any type as long as it supports the interface you care about. It's completely open-ended.
  - *It's less cache-friendly*. Of course, the whole reason we're discussing other options here is because this means cache-unfriendly pointer indirection. But, remember, if this code isn't performance-critical, that's probably OK.

How are game entities defined?

If you use this pattern in tandem with the Component pattern, you'll have nice contiguous arrays for all of the components that make up your game entities. The game loop will be iterating over those directly, so the object for the game entity itself is less important, but it's still useful in other parts of the codebase where you want to work with a single conceptual "entity".

The question then is how should it be represented? How does it keep track of its components?

- **If game entities are classes with pointers to their components:**

  This is what our first example looked like. It's sort of the vanilla OOP solution. You've got a class for `GameEntity`, and it has pointers to the components it owns. Since they're just pointers, it's agnostic about where and how those components are organized in memory.

- *You can store components in contiguous arrays.* Since the game entity doesn't care where its components are, you can organize them in a nice packed array to optimize iterating over them.

- *Given an entity, you can easily get to its components.* They're just a pointer indirection away.

- *Moving components in memory is hard.* When components get enabled or disabled, you may want to move them around in the array to keep the active ones up front and contiguous. If you move a component while the entity has a raw pointer to it, though, that pointer gets broken if you aren't careful. You'll have to make sure to update the entity's pointer at the same time.

- **If game entities are classes with IDs for their components:**

  The challenge with raw pointers to components is that it makes it harder to move them around in memory. You can address that by using something more abstract: an ID or index that can be used to *look up* a component.

  The actual semantics of the ID and lookup process are up to you. It could be as simple as storing a unique ID in each component and walking the array, or more complex like a hash table that maps IDs to their current index in the component array.

  - *It's more complex.* Your ID system doesn't have to be rocket science, but it's still more work than a basic pointer. You'll have to implement and debug it, and there will be memory overhead for bookkeeping.

  - *It's slower.* It's hard to beat traversing a raw pointer. There may be some searching or hashing involved to get from an entity to one of its components.

  - *You'll need access to the component "manager".* The basic idea is that you have some abstract ID that identifies a component. You can use it to get a reference to the actual component object. But to do that, you need to hand that ID to something that can actually find the component. That will be the class that wraps your raw contiguous array of component objects.

    With raw pointers, if you have a game entity, you can find its components. With this, you need the game entity *and the component registry too.*

    > You may be thinking, "I'll just make it a singleton! Problem solved!" Well, sort of. You might want to check out the chapter on those first.

- **If the game entity is *itself* just an ID:**

  This is a newer style that some game engines use. Once you've moved all of your entity's behavior and state out of the main class and into components, what's left? It

turns out, not much. The only thing an entity does is bind a set of components together. It exists just to say *this* AI component and *this* physics component and *this* render component define one living entity in the world.

That's important because components interact. The render component needs to know where the entity is, which may be a property of the physics component. The AI component wants to move the entity, so it needs to apply a force to the physics component. Each component needs a way to get the other sibling components of the entity it's a part of.

Some smart people realized all you need for that is an ID. Instead of the entity knowing its components, the components know their entity. Each component knows the ID of the entity that owns it. When the AI component needs the physics component for its entity, it simply asks for the physics component with the same entity ID that it holds.

Your entity *classes* disappear entirely, replaced by a glorified wrapper around a number.

- *Entities are tiny.* When you want to pass around a reference to a game entity, it's just a single value.
- *Entities are empty.* Of course, the downside of moving everything out of entities is that you *have* to move everything out of entities. You no longer have a place to put non-component-specific state or behavior. This style doubles down on the Component pattern.
- *You don't have to manage their lifetime.* Since entities are just dumb value types, they don't need to be explicitly allocated and freed. An entity implicitly "dies" when all of its components are destroyed.
- *Looking up a component for an entity may be slow.* This is the same problem as the previous answer, but in the opposite direction. To find a component for some entity, you have to map an ID to an object. That process may be costly.

  This time, though, it *is* performance-critical. Components often interact with their siblings during update, so you will need to find components frequently. One solution is to make the "ID" of an entity the index of the component in its array.

  If every entity has the same set of components, then your component arrays are completely parallel. The component in slot three of the AI component array will be for the same entity that the physics component in slot three of *its* array is associated with.

  Keep in mind, though, that this *forces* you to keep those arrays in parallel. That's hard if you want to start sorting or packing them by different criteria. You may have some entities with disabled physics and others that are invisible. There's no

way to sort the physics and render component arrays optimally for both cases if they have to stay in sync with each other.

See Also

- Much of this chapter revolves around the Component pattern, and that pattern is definitely one of the most common data structures that gets optimized for cache usage. In fact, using the Component pattern makes this optimization easier. Since entities are updated one "domain" (AI, physics, etc.) at a time, splitting them out into components lets you slice a bunch of entities into the right pieces to be cache-friendly.

  But that doesn't mean you can *only* use this pattern with components! Any time you have performance-critical code that touches a lot of data, it's important to think about locality.

- Tony Albrecht's "Pitfalls of Object-Oriented Programming" PDF is probably the most widely-read introduction to designing your game's data structures for cache-friendliness. It made a lot more people (including me!) aware of how big of a deal this is for performance.

- Around the same time, Noel Llopis wrote a very influential blog post on the same topic.

- This pattern almost invariably takes advantage of a contiguous array of homogenous objects. Over time, you'll very likely be adding and removing objects from that array. The Object Pool pattern is about exactly that.

- The Artemis game engine is one of the first and better-known frameworks that uses simple IDs for game entities.

Dirty Flag

Intent

Avoid unnecessary work by deferring it until the result is needed.

Motivation

Many games have something called a *scene graph*. This is a big data structure that contains all of the objects in the world. The rendering engine uses this to determine where to draw stuff on the screen.

At its simplest, a scene graph is just a flat list of objects. Each object has a model, or some other graphic primitive, and a *transform*. The transform describes the object's position, rotation, and scale in the world. To move or turn an object, we simply change its transform.

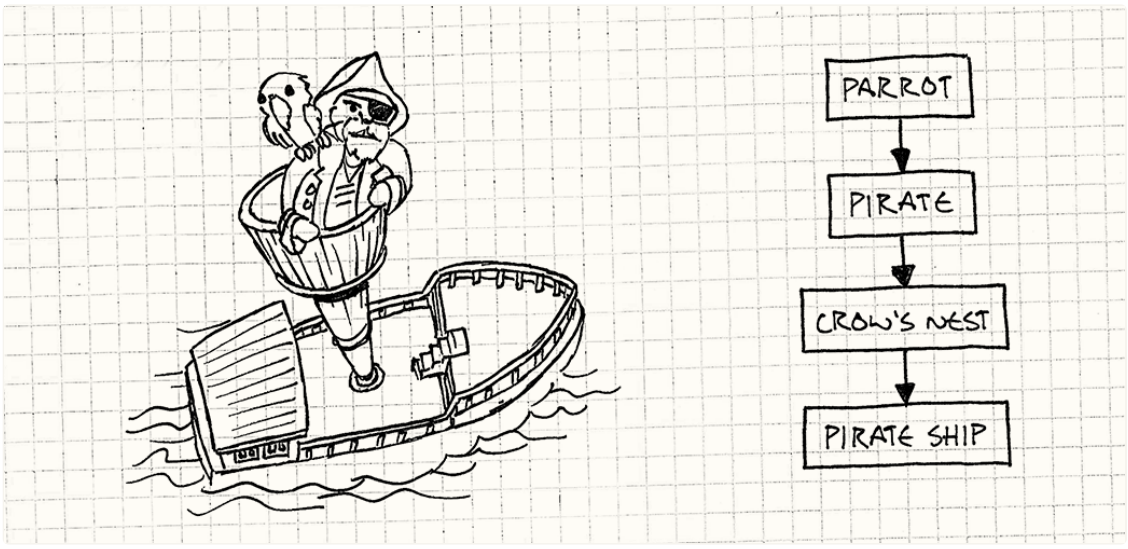
> The mechanics of *how* this transform is stored and manipulated are unfortunately out of scope here. The comically abbreviated summary is that it's a 4x4 matrix. You can make a single transform that combines two transforms—for example, translating and then rotating an object—by multiplying the two matrices.
>
> How and why that works is left as an exercise for the reader.

When the renderer draws an object, it takes the object's model, applies the transform to it, and then renders it there in the world. If we had a scene *bag* and not a scene *graph*, that would be it, and life would be simple.

However, most scene graphs are *hierarchical*. An object in the graph may have a parent object that it is anchored to. In that case, its transform is relative to the *parent's* position and isn't an absolute position in the world.

For example, imagine our game world has a pirate ship at sea. Atop the ship's mast is a

crow's nest. Hunched in that crow's nest is a pirate. Clutching the pirate's shoulder is a parrot. The ship's local transform positions the ship in the sea. The crow's nest's transform positions the nest on the ship, and so on.



Programmer art!

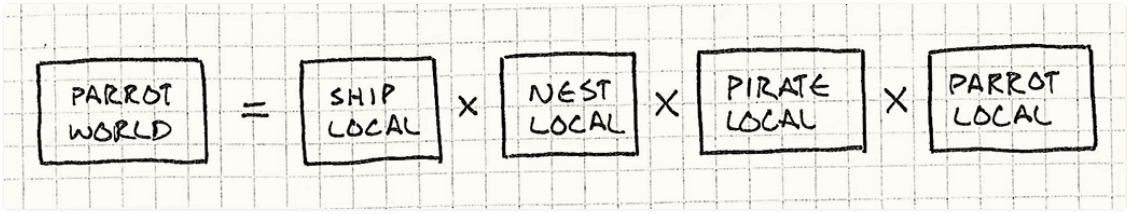
This way, when a parent object moves, its children move with it automatically. If we change the local transform of the ship, the crow's nest, pirate, and parrot go along for the ride. It would be a total headache if, when the ship moved, we had to manually adjust the transforms of all the objects on it to keep them from sliding off.

To be honest, when you are at sea you *do* have to keep manually adjusting your position to keep from sliding off. Maybe I should have chosen a drier example.

But to actually draw the parrot on screen, we need to know its absolute position in the world. I'll call the parent-relative transform the object's *local transform*. To render an object, we need to know its *world transform*.

Local and world transforms

Calculating an object's world transform is pretty straightforward—you just walk its parent chain starting at the root all the way down to the object, combining transforms as you go. In other words, the parrot's world transform is:



In the degenerate case where the object has no parent, its local and

world transforms are equivalent.

We need the world transform for every object in the world every frame, so even though there are only a handful of matrix multiplications per model, it's on the hot code path where performance is critical. Keeping them up to date is tricky because when a parent object moves, that affects the world transform of itself and all of its children, recursively.

The simplest approach is to calculate transforms on the fly while rendering. Each frame, we recursively traverse the scene graph starting at the top of the hierarchy. For each object, we calculate its world transform right then and draw it.

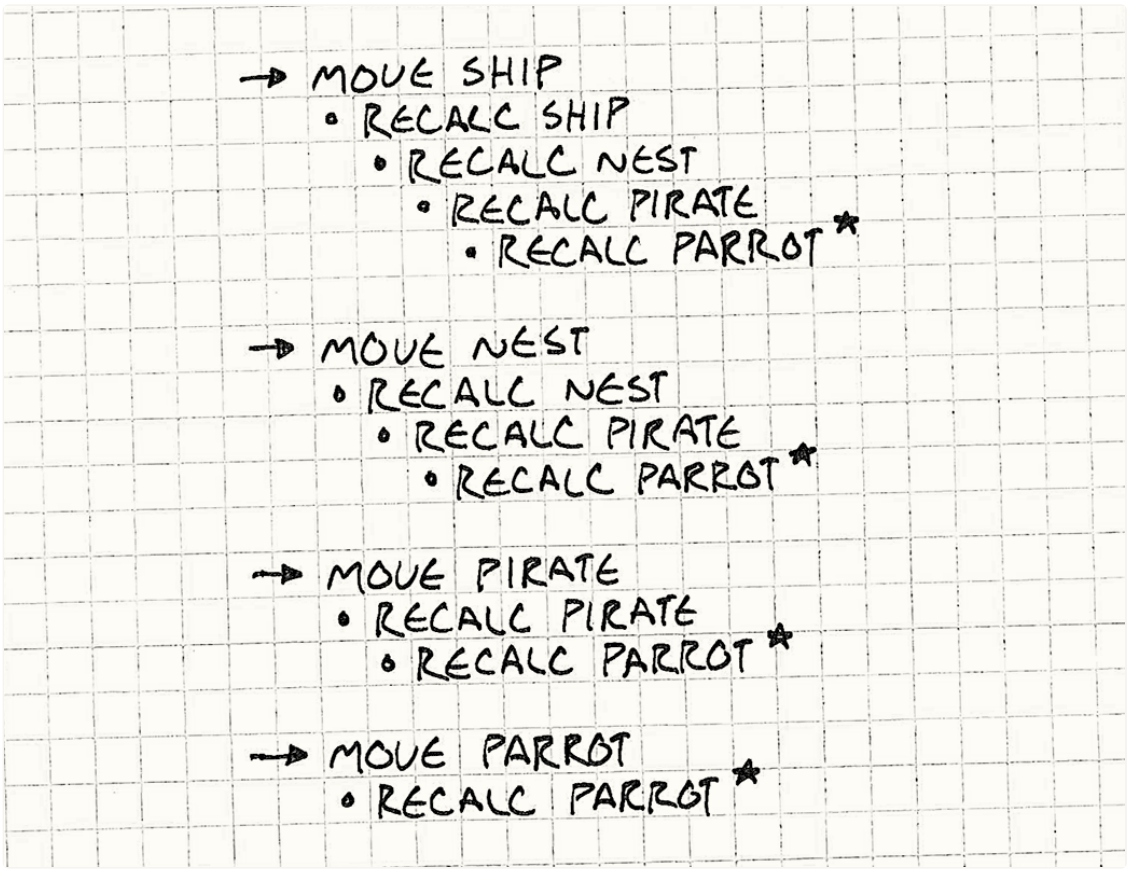
But this is terribly wasteful of our precious CPU juice! Many objects in the world are *not* moving every frame. Think of all of the static geometry that makes up the level. Recalculating their world transforms each frame even though they haven't changed is a waste.

Cached world transforms

The obvious answer is to *cache* it. In each object, we store its local transform and its derived world transform. When we render, we only use the precalculated world transform. If the object never moves, the cached transform is always up to date and everything's happy.

When an object *does* move, the simple approach is to refresh its world transform right then. But don't forget the hierarchy! When a parent moves, we have to recalculate its world transform *and all of its children's, recursively*.

Imagine some busy gameplay. In a single frame, the ship gets tossed on the ocean, the crow's nest rocks in the wind, the pirate leans to the edge, and the parrot hops onto his head. We changed four local transforms. If we recalculate world transforms eagerly whenever a local transform changes, what ends up happening?



> You can see on the lines marked ★ that we're recalculating the parrot's world transform *four* times when we only need the result of the final one.

We only moved four objects, but we did *ten* world transform calculations. That's six pointless calculations that get thrown out before they are ever used by the renderer. We calculated the parrot's world transform *four* times, but it is only rendered once.

The problem is that a world transform may depend on several local transforms. Since we recalculate immediately each time *one* of the transforms changes, we end up recalculating the same transform multiple times when more than one of the local transforms it depends on changes in the same frame.

Deferred recalculation

We'll solve this by decoupling changing local transforms from updating the world transforms. This lets us change a bunch of local transforms in a single batch and *then* recalculate the affected world transform just once after all of those modifications are done, right before we need it to render.

> It's interesting how much of software architecture is intentionally engineering a little slippage.

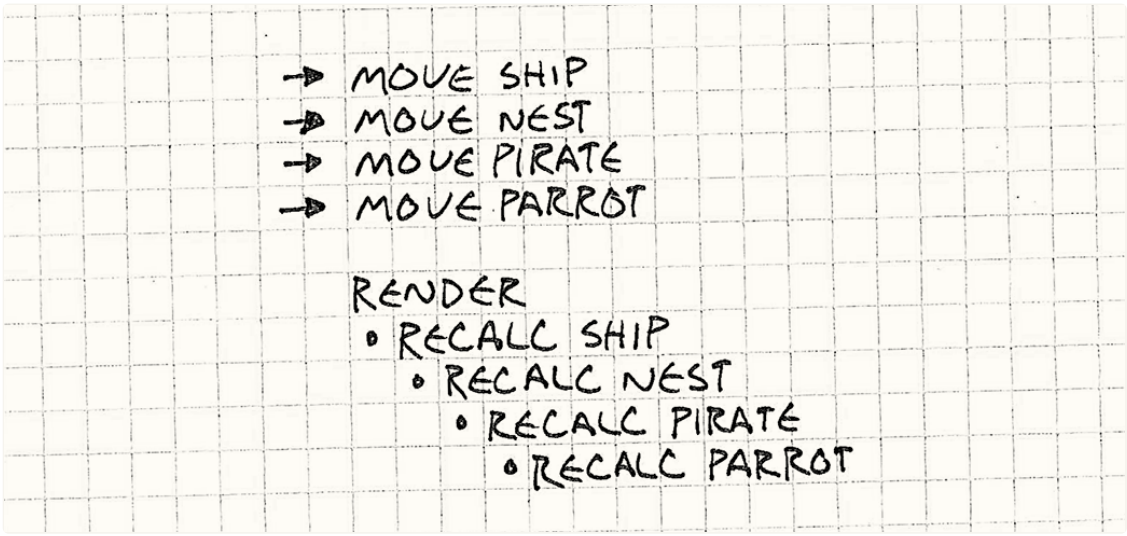
To do this, we add a *flag* to each object in the graph. "Flag" and "bit" are synonymous in programming—they both mean a single micron of data that can be in one of two states. We call those "true" and "false", or sometimes "set" and "cleared". I'll use all of these

interchangeably.

When the local transform changes, we set it. When we need the object's world transform, we check the flag. If it's set, we calculate the world transform and then clear the flag. The flag represents, "Is the world transform out of date?" For reasons that aren't entirely clear, the traditional name for this "out-of-date-ness" is "dirty". Hence: *a dirty flag*. "Dirty bit" is an equally common name for this pattern, but I figured I'd stick with the name that didn't seem as prurient.

> Wikipedia's editors don't have my level of self-control and went with dirty bit.

If we apply this pattern and then move all of the objects in our previous example, the game ends up doing:



That's the best you could hope to do—the world transform for each affected object is calculated exactly once. With only a single bit of data, this pattern does a few things for us:

- It collapses modifications to multiple local transforms along an object's parent chain into a single recalculation on the object.
- It avoids recalculation on objects that didn't move.
- And a minor bonus: if an object gets removed before it's rendered, it doesn't calculate its world transform at all.

The Pattern

A set of **primary data** changes over time. A set of **derived data** is determined from this using some **expensive process**. A **"dirty" flag** tracks when the derived data is out of sync with the primary data. It is **set when the primary data changes**. If the flag is set when the derived data is needed, then **it is reprocessed and the flag is cleared.** Otherwise, the previous **cached derived data** is used.

When to Use It

Compared to some other patterns in this book, this one solves a pretty specific problem. Also, like most optimizations, you should only reach for it when you have a performance problem big enough to justify the added code complexity.

Dirty flags are applied to two kinds of work: *calculation* and *synchronization*. In both cases, the process of going from the primary data to the derived data is time-consuming or otherwise costly.

In our scene graph example, the process is slow because of the amount of math to perform. When using this pattern for synchronization, on the other hand, it's more often that the derived data is *somewhere else*—either on disk or over the network on another machine—and simply getting it from point A to point B is what's expensive.

There are a couple of other requirements too:

- **The primary data has to change more often than the derived data is used.** This pattern works by avoiding processing derived data when a subsequent primary data change would invalidate it before it gets used. If you find yourself always needing that derived data after every single modification to the primary data, this pattern can't help.

- **It should be hard to update incrementally.** Let's say the pirate ship in our game can only carry so much booty. We need to know the total weight of everything in the hold. We *could* use this pattern and have a dirty flag for the total weight. Every time we add or remove some loot, we set the flag. When we need the total, we add up all of the booty and clear the flag.

  But a simpler solution is to *keep a running total*. When we add or remove an item, just add or remove its weight from the current total. If we can "pay as we go" like this and keep the derived data updated, then that's often a better choice than using this pattern and calculating the derived data from scratch when needed.

This makes it sound like dirty flags are rarely appropriate, but you'll find a place here or there where they help. Searching your average game codebase for the word "dirty" will often turn up uses of this pattern.

> From my research, it also turns up a lot of comments apologizing for "dirty" hacks.

Keep in Mind

Even after you've convinced yourself this pattern is a good fit, there are a few wrinkles that

can cause you some discomfort.

There is a cost to deferring for too long

This pattern defers some slow work until the result is actually needed, but when it is, it's often needed *right now*. But the reason we're using this pattern to begin with is because calculating that result is slow!

This isn't a problem in our example because we can still calculate world coordinates fast enough to fit within a frame, but you can imagine other cases where the work you're doing is a big chunk that takes noticeable time to chew through. If the game doesn't *start* chewing until right when the player expects to see the result, that can cause an unpleasant visible pause.

Another problem with deferring is that if something goes wrong, you may fail to do the work at all. This can be particularly problematic when you're using this pattern to save some state to a more persistent form.

For example, text editors know if your document has "unsaved changes". That little bullet or star in your file's title bar is literally the dirty flag visualized. The primary data is the open document in memory, and the derived data is the file on disk.

Many programs don't save to disk until either the document is closed or the application is exited. That's fine most of the time, but if you accidentally kick the power cable out, there goes your masterpiece.

Editors that auto-save a backup in the background are compensating specifically for this shortcoming. The auto-save frequency is a point on the continuum between not losing too much work when a crash occurs and not thrashing the file system too much by saving all the time.

> This mirrors the different garbage collection strategies in systems that automatically manage memory. Reference counting frees memory the second it's no longer needed, but it burns CPU time updating ref counts eagerly every time references are changed.
>
> Simple garbage collectors defer reclaiming memory until it's really needed, but the cost is the dreaded "GC pause" that can freeze your entire game until the collector is done scouring the heap.

> In between the two are more complex systems like deferred ref-counting and incremental GC that reclaim memory less eagerly than pure ref-counting but more eagerly than stop-the-world collectors.

You have to make sure to set the flag *every* time the state changes

Since the derived data is calculated from the primary data, it's essentially a cache. Whenever you have cached data, the trickiest aspect of it is *cache invalidation*—correctly noting when the cache is out of sync with its source data. In this pattern, that means setting the dirty flag when *any* primary data changes.

> Phil Karlton famously said, "There are only two hard things in Computer Science: cache invalidation and naming things."

Miss it in one place, and your program will incorrectly use stale derived data. This leads to confused players and bugs that are very hard to track down. When you use this pattern, you'll have to take care that any code that modifies the primary state also sets the dirty flag.

One way to mitigate this is by encapsulating modifications to the primary data behind some interface. If anything that can change the state goes through a single narrow API, you can set the dirty flag there and rest assured that it won't be missed.

You have to keep the previous derived data in memory

When the derived data is needed and the dirty flag *isn't* set, it uses the previously calculated data. This is obvious, but that does imply that you have to keep that derived data around in memory in case you end up needing it later.

> This isn't much of an issue when you're using this pattern to synchronize the primary state to some other place. In that case, the derived data isn't usually in memory at all.

If you weren't using this pattern, you could calculate the derived data on the fly whenever you needed it, then discard it when you were done. That avoids the expense of keeping it cached in memory at the cost of having to do that calculation every time you need the result.

Like many optimizations, then, this pattern trades memory for speed. In return for keeping the previously calculated data in memory, you avoid having to recalculate it when it hasn't changed. This trade-off makes sense when the calculation is slow and memory is cheap. When you've got more time than memory on your hands, it's better to calculate it as needed.

> Conversely, compression algorithms make the opposite trade-off: they optimize *space* at the expense of the processing time needed to decompress.

Sample Code

Let's assume we've met the surprisingly long list of requirements and see how the pattern looks in code. As I mentioned before, the actual math behind transform matrices is beyond the humble aims of this book, so I'll just encapsulate that in a class whose implementation you can presume exists somewhere out in the æther:

```
class Transform
{
public:
  static Transform origin();

  Transform combine(Transform& other);
};
```

The only operation we need here is `combine()` so that we can get an object's world transform by combining all of the local transforms along its parent chain. It also has a method to get an "origin" transform—basically an identity matrix that means no translation, rotation, or scaling at all.

Next, we'll sketch out the class for an object in the scene graph. This is the bare minimum we need *before* applying this pattern:

```
class GraphNode
{
public:
  GraphNode(Mesh* mesh)
  : mesh_(mesh),
    local_(Transform::origin())
  {}

private:
  Transform local_;
  Mesh* mesh_;

  GraphNode* children_[MAX_CHILDREN];
  int numChildren_;
};
```

Each node has a local transform which describes where it is relative to its parent. It has a mesh which is the actual graphic for the object. (We'll allow `mesh_` to be `NULL` too to handle non-visual nodes that are used just to group their children.) Finally, each node has

a possibly empty collection of child nodes.

With this, a "scene graph" is really only a single root `GraphNode` whose children (and grandchildren, etc.) are all of the objects in the world:

```
GraphNode* graph_ = new GraphNode(NULL);
// Add children to root graph node...
```

In order to render a scene graph, all we need to do is traverse that tree of nodes, starting at the root, and call the following function for each node's mesh with the right world transform:

```
void renderMesh(Mesh* mesh, Transform transform);
```

We won't implement this here, but if we did, it would do whatever magic the renderer needs to draw that mesh at the given location in the world. If we can call that correctly and efficiently on every node in the scene graph, we're happy.

An unoptimized traversal

To get our hands dirty, let's throw together a basic traversal for rendering the scene graph that calculates the world positions on the fly. It won't be optimal, but it will be simple. We'll add a new method to `GraphNode`:

```
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);
  }
}
```

We pass the world transform of the node's parent into this using `parentWorld`. With that, all that's left to get the correct world transform of *this* node is to combine that with its own local transform. We don't have to walk *up* the parent chain to calculate world transforms because we calculate as we go while walking *down* the chain.

We calculate the node's world transform and store it in `world`, then we render the mesh, if we have one. Finally, we recurse into the child nodes, passing in *this* node's world transform. All in all, it's a tight, simple recursive method.

To draw an entire scene graph, we kick off the process at the root node:

```
graph_->render(Transform::origin());
```

Let's get dirty

So this code does the right thing—it renders all the meshes in the right place — but it doesn't do it efficiently. It's calling `local_.combine(parentWorld)` on every node in the graph, every frame. Let's see how this pattern fixes that. First, we need to add two fields to `GraphNode`:

```
class GraphNode
{
public:
  GraphNode(Mesh* mesh)
  : mesh_(mesh),
    local_(Transform::origin()),
    dirty_(true)
  {}

  // Other methods...

private:
  Transform world_;
  bool dirty_;
  // Other fields...
};
```

The `world_` field caches the previously calculated world transform, and `dirty_`, of course, is the dirty flag. Note that the flag starts out `true`. When we create a new node, we haven't calculated its world transform yet. At birth, it's already out of sync with the local transform.

The only reason we need this pattern is because objects can *move*, so let's add support for that:

```
void GraphNode::setTransform(Transform local)
{
  local_ = local;
  dirty_ = true;
}
```

The important part here is that it sets the dirty flag too. Are we forgetting anything? Right—the child nodes!

When a parent node moves, all of its children's world coordinates are invalidated too. But here, we aren't setting their dirty flags. We *could* do that, but that's recursive and slow. Instead, we'll do something clever when we go to render. Let's see:

```
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);
  }
}
```

There's a subtle assumption here that the `if` check is faster than a matrix multiply. Intuitively, you would think it is; surely testing a single bit is faster than a bunch of floating point arithmetic.

However, modern CPUs are fantastically complex. They rely heavily on *pipelining*—queueing up a series of sequential instructions. A branch like our `if` here can cause a *branch misprediction* and force the CPU to lose cycles refilling the pipeline.

The Data Locality chapter has more about how modern CPUs try to go faster and how you can avoid tripping them up like this.

This is similar to the original naïve implementation. The key changes are that we check to see if the node is dirty before calculating the world transform and we store the result in a field instead of a local variable. When the node is clean, we skip `combine()` completely and use the old-but-still-correct `world_` value.

The clever bit is that `dirty` parameter. That will be `true` if any node above this node in the parent chain was dirty. In much the same way that `parentWorld` updates the world transform incrementally as we traverse down the hierarchy, `dirty` tracks the dirtiness of the parent chain.

This lets us avoid having to recursively mark each child's `dirty_` flag in `setTransform()`. Instead, we pass the parent's dirty flag down to its children when we render and look at that too to see if we need to recalculate the world transform.

The end result here is exactly what we want: changing a node's local transform is just a couple of assignments, and rendering the world calculates the exact minimum number of world transforms that have changed since the last frame.

Note that this clever trick only works because `render()` is the *only* thing in

`GraphNode` that needs an up-to-date world transform. If other things accessed it, we'd have to do something different.

Design Decisions

This pattern is fairly specific, so there are only a couple of knobs to twiddle:

When is the dirty flag cleaned?

- **When the result is needed:**
    - *It avoids doing calculation entirely if the result is never used.* For primary data that changes much more frequently than the derived data is accessed, this can be a big win.
    - *If the calculation is time-consuming, it can cause a noticeable pause.* Postponing the work until the player is expecting to see the result can affect their gameplay experience. It's often fast enough that this isn't a problem, but if it is, you'll have to do the work earlier.
- **At well-defined checkpoints:**

  Sometimes, there is a point in time or in the progression of the game where it's natural to do the deferred processing. For example, we may want to save the game only when the pirate sails into port. Or the sync point may not be part of the game mechanics. We may just want to hide the work behind a loading screen or a cut scene.
    - *Doing the work doesn't impact the user experience.* Unlike the previous option, you can often give something to distract the player while the game is busy processing.
    - *You lose control over when the work happens.* This is sort of the opposite of the earlier point. You have micro-scale control over when you process, and can make sure the game handles it gracefully.

      What you *can't* do is ensure the player actually makes it to the checkpoint or meets whatever criteria you've defined. If they get lost or the game gets in a weird state, you can end up deferring longer than you expect.
- **In the background:**

  Usually, you start a fixed timer on the first modification and then process all of the changes that happen between then and when the timer fires.

  The term in human-computer interaction for an intentional delay between when a program receives user input and when it responds is *hysteresis*.

- *You can tune how often the work is performed.* By adjusting the timer interval, you can ensure it happens as frequently (or infrequently) as you want.

- *You can do more redundant work.* If the primary state only changes a tiny amount during the timer's run, you can end up processing a large chunk of mostly unchanged data.

- *You need support for doing work asynchronously.* Processing the data "in the background" implies that the player can keep doing whatever it is that they're doing at the same time. That means you'll likely need threading or some other kind of concurrency support so that the game can work on the data while it's still being played.

  Since the player is likely interacting with the same primary state that you're processing, you'll need to think about making that safe for concurrent modification too.

How fine-grained is your dirty tracking?

Imagine our pirate game lets players build and customize their pirate ship. Ships are automatically saved online so the player can resume where they left off. We're using dirty flags to determine which decks of the ship have been fitted and need to be sent to the server. Each chunk of data we send to the server contains some modified ship data and a bit of metadata describing where on the ship this modification occurred.

- **If it's more fine-grained:**

  Say you slap a dirty flag on each tiny plank of each deck.

  - *You only process data that actually changed.* You'll send exactly the facets of the ship that were modified to the server.

- **If it's more coarse-grained:**

  Alternatively, we could associate a dirty bit with each deck. Changing anything on it marks the entire deck dirty.

  > I could make some terrible joke about it needing to be swabbed here, but I'll refrain.

  - *You end up processing unchanged data.* Add a single barrel to a deck and you'll have to send the whole thing to the server.

  - *Less memory is used for storing dirty flags.* Add ten barrels to a deck and you only need a single bit to track them all.

- *Less time is spent on fixed overhead.* When processing some changed data, there's often a bit of fixed work you have to do on top of handling the data itself. In the example here, that's the metadata required to identify where on the ship the changed data is. The bigger your processing chunks, the fewer of them there are, which means the less overhead you have.

See Also

- This pattern is common outside of games in browser-side web frameworks like Angular. They use dirty flags to track which data has been changed in the browser and needs to be pushed up to the server.

- Physics engines track which objects are in motion and which are resting. Since a resting body won't move until an impulse is applied to it, they don't need processing until they get touched. This "is moving" bit is a dirty flag to note which objects have had forces applied and need to have their physics resolved.

Object Pool

Intent

Improve performance and memory use by reusing objects from a fixed pool instead of allocating and freeing them individually.

Motivation

We're working on the visual effects for our game. When the hero casts a spell, we want a shimmer of sparkles to burst across the screen. This calls for a *particle system*, an engine that spawns little sparkly graphics and animates them until they wink out of existence.

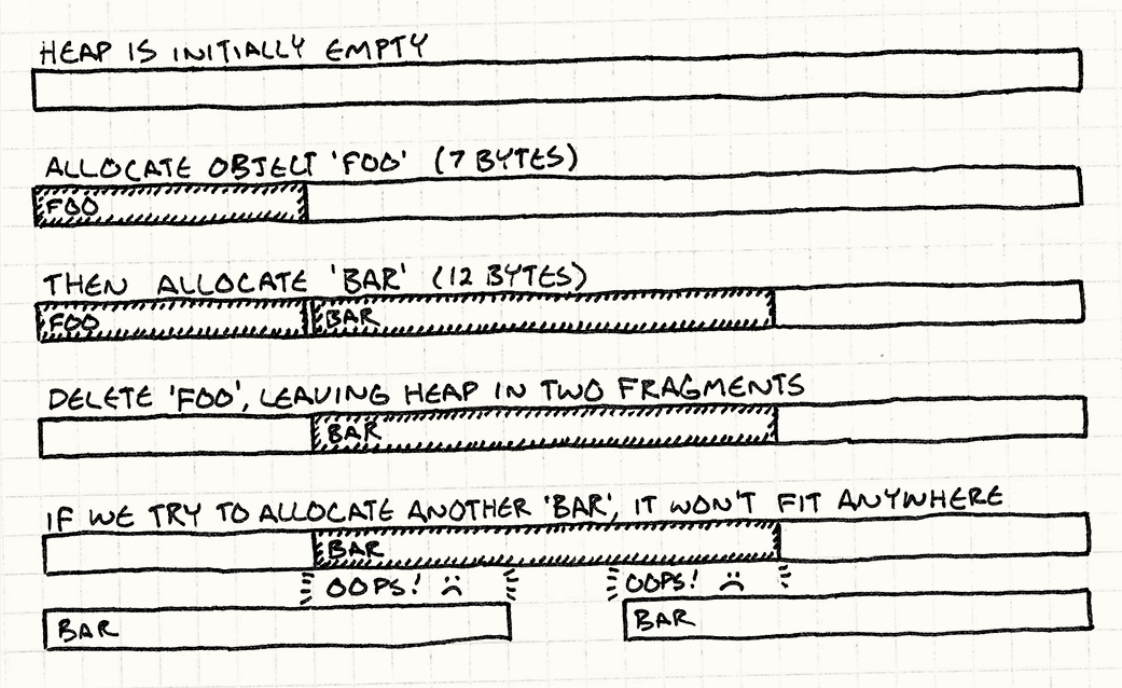
Since a single wave of the wand could cause hundreds of particles to be spawned, our system needs to be able to create them very quickly. More importantly, we need to make sure that creating and destroying these particles doesn't cause *memory fragmentation*.

The curse of fragmentation

Programming for a game console or mobile device is closer to embedded programming than conventional PC programming in many ways. Memory is scarce, users expect games to be rock solid, and efficient compacting memory managers are rarely available. In this environment, memory fragmentation is deadly.

Fragmentation means the free space in our heap is broken into smaller pieces of memory instead of one large open block. The *total* memory available may be large, but the largest *contiguous* region might be painfully small. Say we've got fourteen bytes free, but it's fragmented into two seven-byte pieces with a chunk of in-use memory between them. If we try to allocate a twelve-byte object, we'll fail. No more sparklies on screen.

> It's like trying to parallel park on a busy street where the already parked cars are spread out a bit too far. If they'd bunch up, there would be room, but the free space is *fragmented* into bits of open curb between half a dozen cars.



Here's how a heap becomes fragmented and how it can cause an allocation to fail even where there's theoretically enough memory available.

Even if fragmentation is infrequent, it can still gradually reduce the heap to an unusable foam of open holes and filled-in crevices, ultimately hosing the game completely.

Most console makers require games to pass "soak tests" where they leave the game running in demo mode for several days. If the game crashes, they don't allow it to ship. While soak tests sometimes fail because of a rarely occurring bug, it's usually creeping fragmentation or memory leakage that brings the game down.

The best of both worlds

Because of fragmentation and because allocation may be slow, games are very careful about when and how they manage memory. A simple solution is often best — grab a big chunk of memory when the game starts, and don't free it until the game ends. But this is a pain for systems where we need to create and destroy things while the game is running.

An object pool gives us the best of both worlds. To the memory manager, we're just allocating one big hunk of memory up front and not freeing it while the game is playing. To the users of the pool, we can freely allocate and deallocate objects to our heart's content.

The Pattern

Define a **pool** class that maintains a collection of **reusable objects**. Each object supports an **"in use" query** to tell if it is currently "alive". When the pool is initialized, it creates the entire collection of objects up front (usually in a single contiguous allocation) and initializes them all to the "not in use" state.

When you want a new object, ask the pool for one. It finds an available object, initializes it to "in use", and returns it. When the object is no longer needed, it is set back to the "not in use" state. This way, objects can be freely created and destroyed without needing to allocate memory or other resources.

When to Use It

This pattern is used widely in games for obvious things like game entities and visual effects, but it is also used for less visible data structures such as currently playing sounds. Use Object Pool when:

- You need to frequently create and destroy objects.
- Objects are similar in size.
- Allocating objects on the heap is slow or could lead to memory fragmentation.
- Each object encapsulates a resource such as a database or network connection that is expensive to acquire and could be reused.

Keep in Mind

You normally rely on a garbage collector or `new` and `delete` to handle memory management for you. By using an object pool, you're saying, "I know better how these bytes should be handled." That means the onus is on you to deal with this pattern's limitations.

The pool may waste memory on unneeded objects

The size of an object pool needs to be tuned for the game's needs. When tuning, it's usually obvious when the pool is too *small* (there's nothing like a crash to get your attention). But also take care that the pool isn't too *big*. A smaller pool frees up memory that could be used for other fun stuff.

Only a fixed number of objects can be active at any one time

In some ways, this is a good thing. Partitioning memory into separate pools for different types of objects ensures that, for example, a huge sequence of explosions won't cause your particle system to eat *all* of the available memory, preventing something more critical like a new enemy from being created.

Nonetheless, this also means being prepared for the possibility that your attempt to reuse an object from the pool will fail because they are all in use. There are a few common strategies to handle this:

- *Prevent it outright.* This is the most common "fix": tune the pool sizes so that they never overflow regardless of what the user does. For pools of important objects like enemies or gameplay items, this is often the right answer. There may be no "right" way to handle the lack of a free slot to create the big boss when the player reaches the end of the level, so the smart thing to do is make sure that never happens.

  The downside is that this can force you to sit on a lot of memory for object slots that are needed only for a couple of rare edge cases. Because of this, a single fixed pool size may not be the best fit for all game states. For instance, some levels may feature effects prominently while others focus on sound. In such cases, consider having pool sizes tuned differently for each scenario.

- *Just don't create the object.* This sounds harsh, but it makes sense for cases like our particle system. If all particles are in use, the screen is probably full of flashing graphics. The user won't notice if the next explosion isn't quite as impressive as the ones currently going off.

- *Forcibly kill an existing object.* Consider a pool for currently playing sounds, and assume you want to start a new sound but the pool is full. You do *not* want to simply ignore the new sound—the user will notice if their magical wand swishes dramatically *sometimes* and stays stubbornly silent other times. A better solution is to find the quietest sound already playing and replace that with our new sound. The new sound will mask the audible cutoff of the previous sound.

  In general, if the *disappearance* of an existing object would be less noticeable than the *absence* of a new one, this may be the right choice.

- *Increase the size of the pool.* If your game lets you be a bit more flexible with memory, you may be able to increase the size of the pool at runtime or create a second overflow pool. If you do grab more memory in either of these ways, consider whether or not the pool should contract to its previous size when the additional capacity is no longer needed.

Memory size for each object is fixed

Most pool implementations store the objects in an array of in-place objects. If all of your objects are of the same type, this is fine. However, if you want to store objects of different types in the pool, or instances of subclasses that may add fields, you need to ensure that each slot in the pool has enough memory for the *largest* possible object. Otherwise, an unexpectedly large object will stomp over the next one and trash memory.

At the same time, when your objects vary in size, you waste memory. Each slot needs to be

big enough to accommodate the largest object. If objects are rarely that big, you're throwing away memory every time you put a smaller one in that slot. It's like going through airport security and using a huge carry-on-sized luggage tray just for your keys and wallet.

When you find yourself burning a lot of memory this way, consider splitting the pool into separate pools for different sizes of object—big trays for luggage, little trays for pocket stuff.

> This is a common pattern for implementing speed-efficient memory managers. The manager has a number of pools of different block sizes. When you ask it to allocate a block, it finds in an open slot in the pool of the appropriate size and allocates from that pool.

Reused objects aren't automatically cleared

Most memory managers have a debug feature that will clear freshly allocated or freed memory to some obvious magic value like `0xdeadbeef`. This helps you find painful bugs caused by uninitialized variables or using memory after it's freed.

Since our object pool isn't going through the memory manager any more when it reuses an object, we lose that safety net. Worse, the memory used for a "new" object previously held an object of the exact same type. This makes it nearly impossible to tell if you forgot to initialize something when you created the new object: the memory where the object is stored may already contain *almost* correct data from its past life.

Because of this, pay special care that the code that initializes new objects in the pool *fully* initializes the object. It may even be worth spending a bit of time adding a debug feature that clears the memory for an object slot when the object is reclaimed.

> I'd be honored if you clear it to `0x1deadb0b`.

Unused objects will remain in memory

Object pools are less common in systems that support garbage collection because the memory manager will usually deal with fragmentation for you. But pools are still useful there to avoid the cost of allocation and deallocation, especially on mobile devices with slower CPUs and simpler garbage collectors.

If you do use an object pool in concert with a garbage collector, beware of a potential conflict. Since the pool doesn't actually deallocate objects when they're no longer in use, they remain in memory. If they contain references to *other* objects, it will prevent the collector from reclaiming those too. To avoid this, when a pooled object is no longer in use, clear any references it has to other objects.

Sample Code

Real-world particle systems will often apply gravity, wind, friction, and other physical effects. Our much simpler sample will only move particles in a straight line for a certain number of frames and then kill the particle. Not exactly film caliber, but it should illustrate how to use an object pool.

We'll start with the simplest possible implementation. First up is the little particle class:

```
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_;
};
```

The default constructor initializes the particle to "not in use". A later call to `init()` initializes the particle to a live state. Particles are animated over time using the unsurprisingly named `animate()` function, which should be called once per frame.

The pool needs to know which particles are available for reuse. It gets this from the particle's `inUse()` function. This function takes advantage of the fact that particles have a limited lifetime and uses the `framesLeft_` variable to discover which particles are in use

without having to store a separate flag.

The pool class is also simple:

```
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];
};
```

The `create()` function lets external code create new particles. The game calls `animate()` once per frame, which in turn animates each particle in the pool.

This `animate()` method is an example of the Update Method pattern.

The particles themselves are simply stored in a fixed-size array in the class. In this sample implementation, the pool size is hardcoded in the class declaration, but this could be defined externally by using a dynamic array of a given size or by using a value template parameter.

Creating a new particle is straightforward:

```
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;
    }
  }
}
```

We iterate through the pool looking for the first available particle. When we find it, we initialize it and we're done. Note that in this implementation, if there aren't any available particles, we simply don't create a new one.

That's all there is to a simple particle system, aside from rendering the particles, of course. We can now create a pool and create some particles using it. The particles will automatically deactivate themselves when their lifetime has expired.

This is good enough to ship a game, but keen eyes may have noticed that creating a new particle requires iterating through (potentially) the entire collection until we find an open slot. If the pool is very large and mostly full, that can get slow. Let's see how we can improve that.

> Creating a particle has *O(n)* complexity, for those of us who remember our algorithms class.

A free list

If we don't want to waste time *finding* free particles, the obvious answer is to not lose track of them. We could store a separate list of pointers to each unused particle. Then, when we need to create a particle, we remove the first pointer from the list and reuse the particle it points to.

Unfortunately, this would require us to maintain an entire separate array with as many pointers as there are objects in the pool. After all, when we first create the pool, *all* particles are unused, so the list would initially have a pointer to every object in the pool.

It would be nice to fix our performance problems *without* sacrificing any memory. Conveniently, there is some memory already lying around that we can borrow — the data for the unused particles themselves.

When a particle isn't in use, most of its state is irrelevant. Its position and velocity aren't being used. The only state it needs is the stuff required to tell if it's dead. In our example, that's the `framesLeft_` member. All those other bits can be reused. Here's a revised particle:

```
class Particle
{
public:
  // ...

  Particle* getNext() const { return state_.next; }
  void setNext(Particle* next) { state_.next = next; }

private:
  int framesLeft_;
```

```
  union
  {
    // State when it's in use.
    struct
    {
      double x, y;
      double xVel, yVel;
    } live;

    // State when it's available.
    Particle* next;
  } state_;
};
```

We've moved all of the member variables except for `framesLeft_` into a `live` struct inside a `state_` union. This struct holds the particle's state when it's being animated. When the particle is unused, the other case of the union, the `next` member, is used. It holds a pointer to the next available particle after this one.

> Unions don't seem to be used that often these days, so the syntax may be unfamiliar to you. If you're on a game team, you've probably got a "memory guru", that beleaguered compatriot whose job it is to come up with a solution when the game has inevitably blown its memory budget. Ask them about unions. They'll know all about them and other fun bit-packing tricks.

We can use these pointers to build a linked list that chains together every unused particle in the pool. We have the list of available particles we need, but we didn't need to use any additional memory. Instead, we cannibalize the memory of the dead particles themselves to store the list.

This clever technique is called a *free list*. For it to work, we need to make sure the pointers are initialized correctly and are maintained when particles are created and destroyed. And, of course, we need to keep track of the list's head:

```
class ParticlePool
{
  // ...
private:
  Particle* firstAvailable_;
};
```

When a pool is first created, *all* of the particles are available, so our free list should thread through the entire pool. The pool constructor sets that up:

```
ParticlePool::ParticlePool()
```

```
{
  // The first one is available.
  firstAvailable_ = &particles_[0];

  // Each particle points to the next.
  for (int i = 0; i < POOL_SIZE - 1; i++)
  {
    particles_[i].setNext(&particles_[i + 1]);
  }

  // The last one terminates the list.
  particles_[POOL_SIZE - 1].setNext(NULL);
}
```

Now to create a new particle, we jump directly to the first available one:

O(1) complexity, baby! Now we're cooking!

```
void ParticlePool::create(double x, double y,
                          double xVel, double yVel,
                          int lifetime)
{
  // Make sure the pool isn't full.
  assert(firstAvailable_ != NULL);

  // Remove it from the available list.
  Particle* newParticle = firstAvailable_;
  firstAvailable_ = newParticle->getNext();

  newParticle->init(x, y, xVel, yVel, lifetime);
}
```

We need to know when a particle dies so we can add it back to the free list, so we'll change `animate()` to return `true` if the previously live particle gave up the ghost in that frame:

```
bool Particle::animate()
{
  if (!inUse()) return false;

  framesLeft_--;
  x_ += xVel_;
  y_ += yVel_;

  return framesLeft_ == 0;
}
```

When that happens, we simply thread it back onto the list:

```
void ParticlePool::animate()
{
  for (int i = 0; i < POOL_SIZE; i++)
  {
    if (particles_[i].animate())
    {
      // Add this particle to the front of the list.
      particles_[i].setNext(firstAvailable_);
      firstAvailable_ = &particles_[i];
    }
  }
}
```

There you go, a nice little object pool with constant-time creation and deletion.

Design Decisions

As you've seen, the simplest object pool implementation is almost trivial: create an array of objects and reinitialize them as needed. Production code is rarely that minimal. There are several ways to expand on that to make the pool more generic, safer to use, or easier to maintain. As you implement pools in your games, you'll need to answer these questions:

Are objects coupled to the pool?

The first question you'll run into when writing an object pool is whether the objects themselves know they are in a pool. Most of the time they will, but you won't have that luxury when writing a generic pool class that can hold arbitrary objects.

- **If objects are coupled to the pool:**
  - *The implementation is simpler.* You can simply put an "in use" flag or function in your pooled object and be done with it.
  - *You can ensure that the objects can only be created by the pool.* In C++, a simple way to do this is to make the pool class a friend of the object class and then make the object's constructor private.

```
class Particle
{
  friend class ParticlePool;

private:
  Particle()
  : inUse_(false)
  {}

  bool inUse_;
};
```

```
class ParticlePool
{
  Particle pool_[100];
};
```

This relationship documents the intended way to use the class and ensures your users don't create objects that aren't tracked by the pool.

- *You may be able to avoid storing an explicit "in use" flag.* Many objects already retain some state that could be used to tell whether it is alive or not. For example, a particle may be available for reuse if its current position is offscreen. If the object class knows it may be used in a pool, it can provide an `inUse()` method to query that state. This saves the pool from having to burn some extra memory storing a bunch of "in use" flags.

- **If objects are not coupled to the pool:**

  - *Objects of any type can be pooled.* This is the big advantage. By decoupling objects from the pool, you may be able to implement a generic reusable pool class.

  - *The "in use" state must be tracked outside the objects.* The simplest way to do this is by creating a separate bit field:

```
template <class TObject>
class GenericPool
{
private:
  static const int POOL_SIZE = 100;

  TObject pool_[POOL_SIZE];
  bool    inUse_[POOL_SIZE];
};
```

What is responsible for initializing the reused objects?

In order to reuse an existing object, it must be reinitialized with new state. A key question here is whether to reinitialize the object inside the pool class or outside.

- **If the pool reinitializes internally:**

  - *The pool can completely encapsulate its objects.* Depending on the other capabilities your objects need, you may be able to keep them completely internal to the pool. This makes sure that other code doesn't maintain references to objects that could be unexpectedly reused.

  - *The pool is tied to how objects are initialized.* A pooled object may offer multiple functions that initialize it. If the pool manages initialization, its interface needs to

support all of those and forward them to the object.

```
class Particle
{
  // Multiple ways to initialize.
  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)
  {
    // Forward to Particle...
  }

  void create(double x, double y, double angle)
  {
    // Forward to Particle...
  }

  void create(double x, double y, double xVel, double yVel)
  {
    // Forward to Particle...
  }
};
```

- **If outside code initializes the object:**
  - *The pool's interface can be simpler*. Instead of offering multiple functions to cover each way an object can be initialized, the pool can simply return a reference to the new object:

```
class Particle
{
public:
  // Multiple ways to initialize.
  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()
  {
    // Return reference to available particle...
```

```
  }
private:
  Particle pool_[100];
};
```

The caller can then initialize the object by calling any method the object exposes:

```
ParticlePool pool;

pool.create()->init(1, 2);
pool.create()->init(1, 2, 0.3);
pool.create()->init(1, 2, 3.3, 4.4);
```

- *Outside code may need to handle the failure to create a new object.* The previous example assumes that `create()` will always successfully return a pointer to an object. If the pool is full, though, it may return `NULL` instead. To be safe, you'll need to check for that before you try to initialize the object:

```
Particle* particle = pool.create();
if (particle != NULL) particle->init(1, 2);
```

See Also

- This looks a lot like the Flyweight [GoF] pattern. Both maintain a collection of reusable objects. The difference is what "reuse" means. Flyweight objects are reused by sharing the same instance between multiple owners *simultaneously*. The Flyweight pattern avoids *duplicate* memory usage by using the same object in multiple contexts.

  The objects in a pool get reused too, but only over time. "Reuse" in the context of an object pool means reclaiming the memory for an object *after* the original owner is done with it. With an object pool, there isn't any expectation that an object will be shared within its lifetime.

- Packing a bunch of objects of the same type together in memory helps keep your CPU cache full as the game iterates over those objects. The Data Locality pattern is all about that.

Spatial Partition

Intent

Efficiently locate objects by storing them in a data structure organized by their positions.

Motivation

Games let us visit other worlds, but those worlds typically aren't so different from our own. They often share the same basic physics and tangibility of our universe. This is why they can feel real despite being crafted of mere bits and pixels.

One bit of fake reality that we'll focus on here is *location*. Game worlds have a sense of *space*, and objects are somewhere in that space. This manifests itself in a bunch of ways. The obvious one is physics—objects move, collide, and interact—but there are other examples. The audio engine may take into account where sound sources are relative to the player so that distant sounds are quieter. Online chat may be restricted to nearby players.

This means your game engine often needs to answer to the question, "What objects are near this location?" If it has to answer this enough times each frame, it can start to be a performance bottleneck.

Units on the field of battle

Say we're making a real-time strategy game. Opposing armies with hundreds of units will clash together on the field of battle. Warriors need to know which nearby enemy to swing their blades at. The naïve way to handle this is by looking at every pair of units and seeing how close they are to each other:

```
void handleMelee(Unit* units[], int numUnits)
{
  for (int a = 0; a < numUnits - 1; a++)
  {
    for (int b = a + 1; b < numUnits; b++)
```

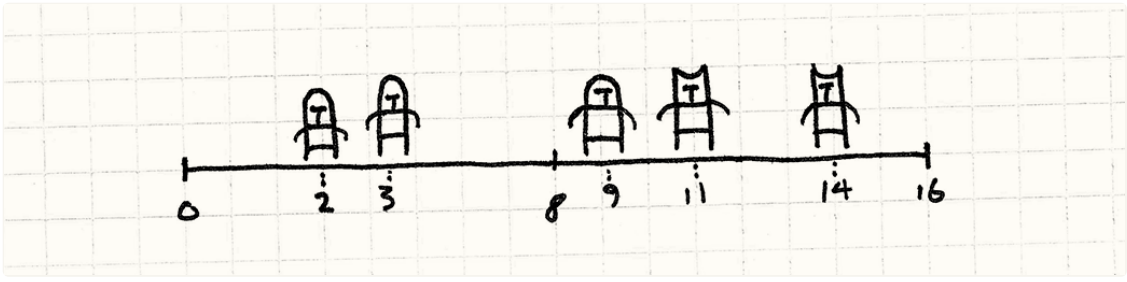
```
    {
      if (units[a]->position() == units[b]->position())
      {
        handleAttack(units[a], units[b]);
      }
    }
  }
}
```

Here we have a doubly nested loop where each loop is walking all of the units on the battlefield. That means the number of pairwise tests we have to perform each frame increases with the *square* of the number of units. Each additional unit we add has to be compared to *all* of the previous ones. With a large number of units, that can spiral out of control.

> The inner loop doesn't actually walk all of the units. It only walks the ones the outer loop hasn't already visited. This avoids comparing each pair of units *twice*, once in each order. If we've already handled a collision between A and B, we don't need to check it again for B and A.
>
> In Big-O terms, though, this is still $O(n^2)$.

Drawing battle lines

The problem we're running into is that there's no underlying order to the array of units. To find a unit near some location, we have to walk the entire array. Now, imagine we simplify our game a bit. Instead of a 2D battle*field*, imagine it's a 1D battle*line*.



In that case, we could make things easier on ourselves by *sorting* the array of units by their positions on the battleline. Once we do that, we can use something like a binary search to find nearby units without having to scan the entire array.

> A binary search has $O(\log n)$ complexity, which means find all battling units goes from $O(n^2)$ to $O(n \log n)$. Something like a pigeonhole sort could get that down to $O(n)$.

The lesson is pretty obvious: if we store our objects in a data structure organized by their locations, we can find them much more quickly. This pattern is about applying that idea to spaces that have more than one dimension.

The Pattern

For a set of **objects**, each has a **position in space**. Store them in a **spatial data structure** that organizes the objects by their positions. This data structure lets you **efficiently query for objects at or near a location**. When an object's position changes, **update the spatial data structure** so that it can continue to find the object.

When to Use It

This is a common pattern for storing both live, moving game objects and also the static art and geometry of the game world. Sophisticated games often have multiple spatial partitions for different kinds of content.

The basic requirements for this pattern are that you have a set of objects that each have some kind of position and that you are doing enough queries to find objects by location that your performance is suffering.

Keep in Mind

Spatial partitions exist to knock an $O(n)$ or $O(n^2)$ operation down to something more manageable. The *more* objects you have, the more valuable that becomes. Conversely, if your n is small enough, it may not be worth the bother.

Since this pattern involves organizing objects by their positions, objects that *change* position are harder to deal with. You'll have to reorganize the data structure to keep track of an object at a new location, and that adds code complexity *and* spends CPU cycles. Make sure the trade-off is worth it.

> Imagine a hash table where the keys of the hashed objects can change spontaneously, and you'll have a good feel for why it's tricky.

A spatial partition also uses additional memory for its bookkeeping data structures. Like many optimizations, it trades memory for speed. If you're shorter on memory than you are on clock cycles, that may be a losing proposition.

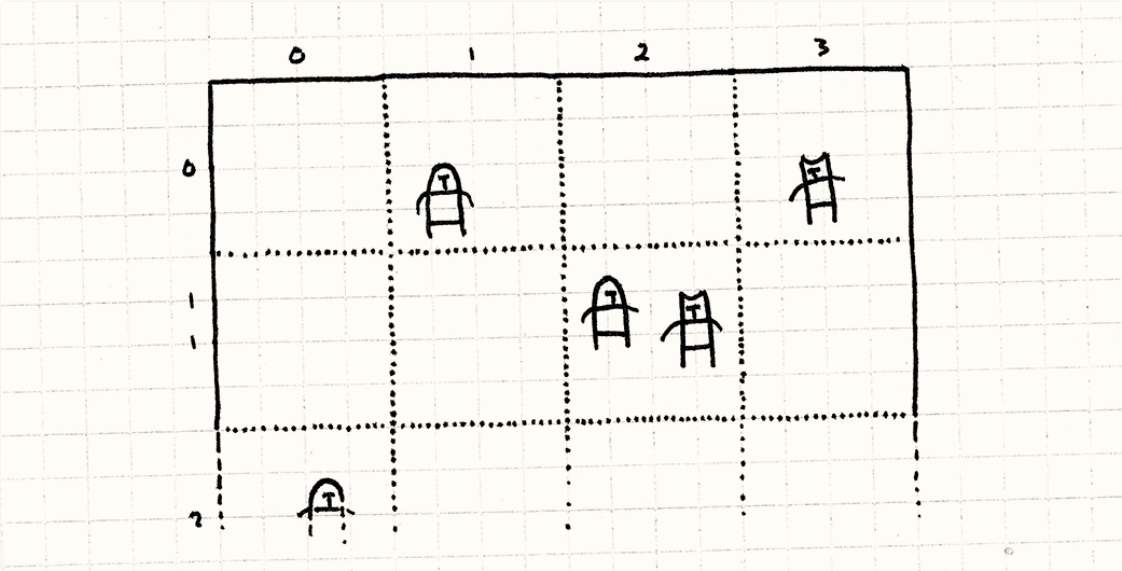
Sample Code

The nature of patterns is that they *vary*—each implementation will be a bit different, and spatial partitions are no exception. Unlike other patterns, though, many of these variations are well-documented. Academia likes publishing papers that prove performance gains. Since I only care about the concept behind the pattern, I'm going to show you the simplest spatial partition: a *fixed grid*.

See the last section of this chapter for a list of some of the most common spatial partitions used in games.

A sheet of graph paper

Imagine the entire field of battle. Now, superimpose a grid of fixed-size squares onto it like a sheet of graph paper. Instead of storing our units in a single array, we put them in the cells of this grid. Each cell stores the list of units whose positions are within that cell's boundary.



When we handle combat, we only consider units within the same cell. Instead of comparing each unit in the game with every other unit, we've *partitioned* the battlefield into a bunch of smaller mini-battlefields, each with many fewer units.

A grid of linked units

OK, let's get coding. First, some prep work. Here's our basic `Unit` class:

```
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_;
};
```

Each unit has a position (in 2D) and a pointer to the `Grid` that it lives on. We make `Grid` a `friend` class because, as we'll see, when a unit's position changes, it has to do an intricate dance with the grid to make sure everything is updated correctly.

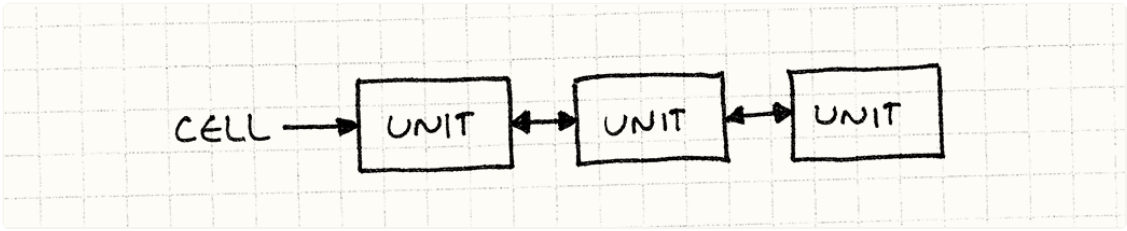
Here's a sketch of the grid:

```
class Grid
{
public:
  Grid()
  {
    // Clear the grid.
    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];
};
```

Note that each cell is just a pointer to a unit. Next, we'll extend `Unit` with `next` and `prev` pointers:

```
class Unit
{
  // Previous code...
private:
  Unit* prev_;
  Unit* next_;
};
```

This lets us organize units into a doubly linked list instead of an array.



Each cell in the grid points to the first unit in the list of units within that cell, and each unit has pointers to the units before it and after it in the list. We'll see why soon.

> Throughout this book, I've avoided using any of the built-in collection types in the C++ standard library. I want to require as little external knowledge as possible to understand the example, and, like a magician's "nothing up my sleeve", I want to make it clear *exactly* what's going on in the code. Details are important, especially with performance-related patterns.
>
> But this is my choice for *explaining* patterns. If you're *using* them in real code, spare yourself the headache and use the fine collections built into pretty much every programming language today. Life's too short to code linked lists from scratch.

Entering the field of battle

The first thing we need to do is make sure new units are actually placed into the grid when they are created. We'll make `Unit` handle this in its constructor:

```
Unit::Unit(Grid* grid, double x, double y)
: grid_(grid),
  x_(x),
  y_(y),
  prev_(NULL),
  next_(NULL)
{
  grid_->add(this);
}
```

This `add()` method is defined like so:

```
void Grid::add(Unit* unit)
{
  // Determine which grid cell it's in.
  int cellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int cellY = (int)(unit->y_ / Grid::CELL_SIZE);

  // Add to the front of list for the cell it's in.
  unit->prev_ = NULL;
  unit->next_ = cells_[cellX][cellY];
  cells_[cellX][cellY] = unit;

  if (unit->next_ != NULL)
  {
    unit->next_->prev_ = unit;
  }
}
```

Dividing by the cell size converts world coordinates to cell space. Then, casting to an `int` truncates the fractional part so we get the cell index.

It's a little finicky like linked list code always is, but the basic idea is pretty simple. We find the cell that the unit is sitting in and then add it to the front of that list. If there is already a list of units there, we link it in after the new unit.

A clash of swords

Once all of the units are nestled in their cells, we can let them start hacking at each other. With this new grid, the main method for handling combat looks like this:

```
void Grid::handleMelee()
{
  for (int x = 0; x < NUM_CELLS; x++)
  {
    for (int y = 0; y < NUM_CELLS; y++)
    {
      handleCell(cells_[x][y]);
    }
  }
}
```

It walks each cell and then calls `handleCell()` on it. As you can see, we really have partitioned the battlefield into little isolated skirmishes. Each cell then handles its combat like so:

```
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_;
  }
}
```

Aside from the pointer shenanigans to deal with walking a linked list, note that this is

exactly like our original naïve method for handling combat. It compares each pair of units to see if they're in the same position.

The only difference is that we no longer have to compare *all* of the units in the battle to each other—just the ones close enough to be in the same cell. That's the heart of the optimization.

> From a simple analysis, it looks like we've actually made the performance *worse*. We've gone from a doubly nested loop over the units to a *triply* nested loop over the cells and then the units. The trick here is that the two inner loops are now over a smaller number of units, which is enough to cancel out the cost of the outer loop over the cells.
>
> However, that does depend a bit on the granularity of our cells. Make them too small and that outer loop can start to matter.

Charging forward

We've solved our performance problem, but we've created a new problem in its stead. Units are now stuck in their cells. If we move a unit past the boundary of the cell that contains it, units in the cell won't see it anymore, but neither will anyone else. Our battlefield is a little *too* partitioned.

To fix that, we'll need to do a little work each time a unit moves. If it crosses a cell's boundary lines, we need to remove it from that cell and add it to the new one. First, we'll give `Unit` a method for changing its position:

```
void Unit::move(double x, double y)
{
  grid_->move(this, x, y);
}
```

Presumably, this gets called by the AI code for computer-controlled units and by the user input code for the player's units. All it does is hand off control to the grid, which then does:

```
void Grid::move(Unit* unit, double x, double y)
{
  // See which cell it was in.
  int oldCellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int oldCellY = (int)(unit->y_ / Grid::CELL_SIZE);

  // See which cell it's moving to.
  int cellX = (int)(x / Grid::CELL_SIZE);
  int cellY = (int)(y / Grid::CELL_SIZE);

  unit->x_ = x;
  unit->y_ = y;
```

```
  // If it didn't change cells, we're done.
  if (oldCellX == cellX && oldCellY == cellY) return;

  // Unlink it from the list of its old cell.
  if (unit->prev_ != NULL)
  {
    unit->prev_->next_ = unit->next_;
  }

  if (unit->next_ != NULL)
  {
    unit->next_->prev_ = unit->prev_;
  }

  // If it's the head of a list, remove it.
  if (cells_[oldCellX][oldCellY] == unit)
  {
    cells_[oldCellX][oldCellY] = unit->next_;
  }

  // Add it back to the grid at its new cell.
  add(unit);
}
```

That's a mouthful of code, but it's pretty straightforward. The first bit checks to see if we've crossed a cell boundary at all. If not, all we need to do is update the unit's position and we're done.

If the unit *has* left its current cell, we remove it from that cell's linked list and then add it back to the grid. Like with adding a new unit, that will insert the unit in the linked list for its new cell.

This is why we're using a doubly linked list—we can very quickly add and remove units from lists by setting a few pointers. With lots of units moving around each frame, that can be important.

At arm's length

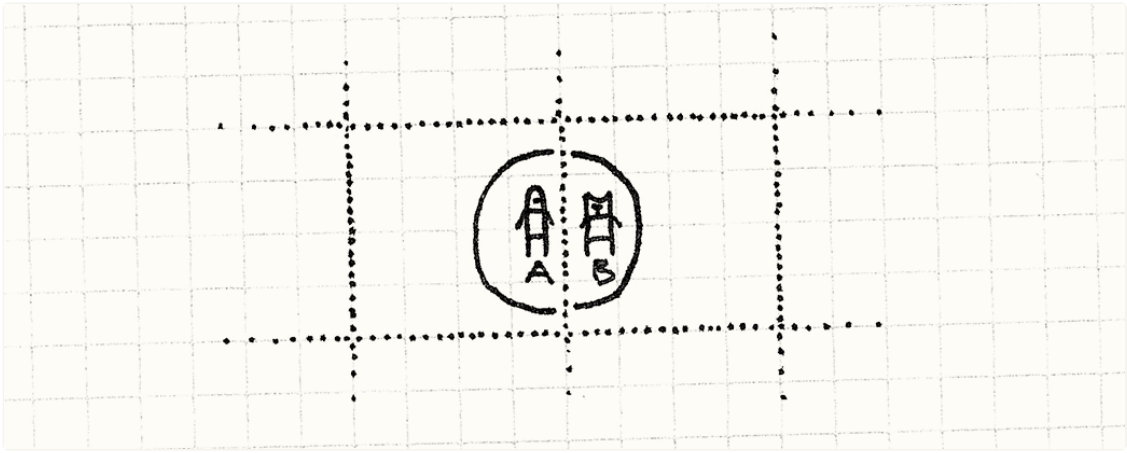
This seems pretty simple, but I have cheated in one way. In the example I've been showing, units only interact when they have the *exact same* position. That's true for checkers and chess, but less true for more realistic games. Those usually have attack *distances* to take into account.

This pattern still works fine. Instead of just checking for an exact location match, we'll do something more like:

```
if (distance(unit, other) < ATTACK_DISTANCE)
{
  handleAttack(unit, other);
```

```
}
```

When range gets involved, there's a corner case we need to consider: units in different cells may still be close enough to interact.



Here, B is within A's attack radius even through their centerpoints are in different cells. To handle this, we will need to compare units not only in the same cell, but in neighboring cells too. To do this, first we'll split the inner loop out of `handleCell()`:

```
void Grid::handleUnit(Unit* unit, Unit* other)
{
  while (other != NULL)
  {
    if (distance(unit, other) < ATTACK_DISTANCE)
    {
      handleAttack(unit, other);
    }

    other = other->next_;
  }
}
```

Now we have a function that will take a single unit and a list of other units and see if there are any hits. Then we'll make `handleCell()` use that:

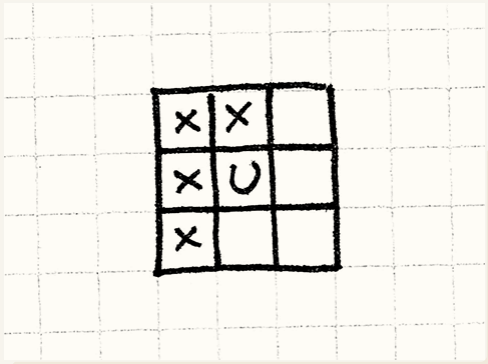
```
void Grid::handleCell(int x, int y)
{
  Unit* unit = cells_[x][y];
  while (unit != NULL)
  {
    // Handle other units in this cell.
    handleUnit(unit, unit->next_);

    unit = unit->next_;
  }
}
```

Note that we now also pass in the coordinates of the cell, not just its unit list. Right now, this doesn't do anything differently from the previous example, but we'll expand it slightly:

```
void Grid::handleCell(int x, int y)
{
  Unit* unit = cells_[x][y];
  while (unit != NULL)
  {
    // Handle other units in this cell.
    handleUnit(unit, unit->next_);

    // Also try the neighboring cells.
    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_;
  }
}
```

Those additional `handleUnit()` calls look for hits between the current unit and units in four of the eight neighboring cells. If any unit in those neighboring cells is close enough to the edge to be within the unit's attack radius, it will find the hit.

The cell with the unit is `U`, and the neighboring cells it looks at are `X`.



We only look at *half* of the neighbors for the same reason that the inner loop starts *after* the current unit— to avoid comparing each pair of units twice. Consider what would happen if we did check all eight neighboring cells.

Let's say we have two units in adjacent cells close enough to hit each other, like the previous example. Here's what would happen if we looked at all eight cells surrounding each unit:

1. When finding hits for A, we would look at its neighbor on the right and find B. So we'd register an attack between A and B.

2. Then, when finding hits for B, we would look at its neighbor on the *left* and find A. So we'd register a *second* attack between A and B.

Only looking at half of the neighboring cells fixes that. *Which* half we look at doesn't matter at all.

There's another corner case we may need to consider too. Here, we're assuming the maximum attack distance is smaller than a cell. If we have small cells and large attack distances, we may need to scan a bunch of neighboring cells several rows out.

Design Decisions

There's a relatively short list of well-defined spatial partitioning data structures, and one option would be to go through them one at a time here. Instead, I tried to organize this by their essential characteristics. My hope is that once you do learn about quadtrees and binary space partitions (BSPs) and the like, this will help you understand *how* and *why* they work and why you might choose one over the other.

Is the partition hierarchical or flat?

Our grid example partitioned space into a single flat set of cells. In contrast, hierarchical spatial partitions divide the space into just a couple of regions. Then, if one of these regions still contains many objects, it's subdivided. This process continues recursively until every region has fewer than some maximum number of objects in it.

> They usually split it in two, four, or eight—nice round numbers to a programmer.

- **If it's a flat partition:**

    - *It's simpler.* Flat data structures are easier to reason about and simpler to implement.

        > This is a design point I mention in almost every chapter, and for good reason. Whenever you can, take the simpler option. Much of software engineering is fighting against complexity.

    - *Memory usage is constant.* Since adding new objects doesn't require creating new partitions, the memory used by the spatial partition can often be fixed ahead of time.

    - *It can be faster to update when objects change their positions.* When an object

moves, the data structure needs to be updated to find the object in its new location. With a hierarchical spatial partition, this can mean adjusting several layers of the hierarchy.

- **If it's hierarchical:**
  - *It handles empty space more efficiently*. Imagine in our earlier example if one whole side of the battlefield was empty. We'd have a large number of empty cells that we'd still have to allocate memory for and walk each frame.

    Since hierarchical space partitions don't subdivide sparse regions, a large empty space will remain a single partition. Instead of lots of little partitions to walk, there is a single big one.

  - *It handles densely populated areas more efficiently*. This is the other side of the coin: if you have a bunch of objects all clumped together, a non-hierarchical partition can be ineffective. You'll end up with one partition that has so many objects in it that you may as well not be partitioning at all. A hierarchical partition will adaptively subdivide that into smaller partitions and get you back to having only a few objects to consider at a time.

Does the partitioning depend on the set of objects?

In our sample code, the grid spacing was fixed beforehand, and we slotted units into cells. Other partitioning schemes are adaptable—they pick partition boundaries based on the actual set of objects and where they are in the world.

The goal is have a *balanced* partitioning where each region has roughly the same number of objects in order to get the best performance. Consider in our grid example if all of the units were clustered in one corner of the battlefield. They'd all be in the same cell, and our code for finding attacks would regress right back to the original $O(n^2)$ problem that we're trying to solve.

- **If the partitioning is object-independent:**
  - *Objects can be added incrementally*. Adding an object means finding the right partition and dropping it in, so you can do this one at a time without any performance issues.

  - *Objects can be moved quickly*. With fixed partitions, moving a unit means removing it from one and adding it to another. If the partition boundaries themselves change based on the set of objects, then moving one can cause a boundary to move, which can in turn cause lots of other objects to need to be moved to different partitions.

    > This is directly analogous to sorted binary search trees like red-black trees or AVL trees: when you add a single item, you may end

up needing to re-sort the tree and shuffle a bunch of nodes around.

- *The partitions can be imbalanced.* Of course, the downside of this rigidity is that you have less control over your partitions being evenly distributed. If objects clump together, you get worse performance there while wasting memory in the empty areas.

- **If the partitioning adapts to the set of objects:**

  Spatial partitions like BSPs and k-d trees split the world recursively so that each half contains about the same number of objects. To do this, you have to count how many objects are on each side when selecting the planes you partition along. Bounding volume hierarchies are another type of spatial partition that optimizes for the specific set of objects in the world.

  - *You can ensure the partitions are balanced.* This gives not just good performance, but *consistent* performance: if each partition has the same number of objects, you ensure that all queries in the world will take about the same amount of time. When you need to maintain a stable frame rate, this consistency may be more important than raw performance.

  - *It's more efficient to partition an entire set of objects at once.* When the *set* of objects affects where boundaries are, it's best to have all of the objects up front before you partition them. This is why these kinds of partitions are more frequently used for art and static geometry that stays fixed during the game.

- **If the partitioning is object-independent, but *hierarchy* is object-dependent:**

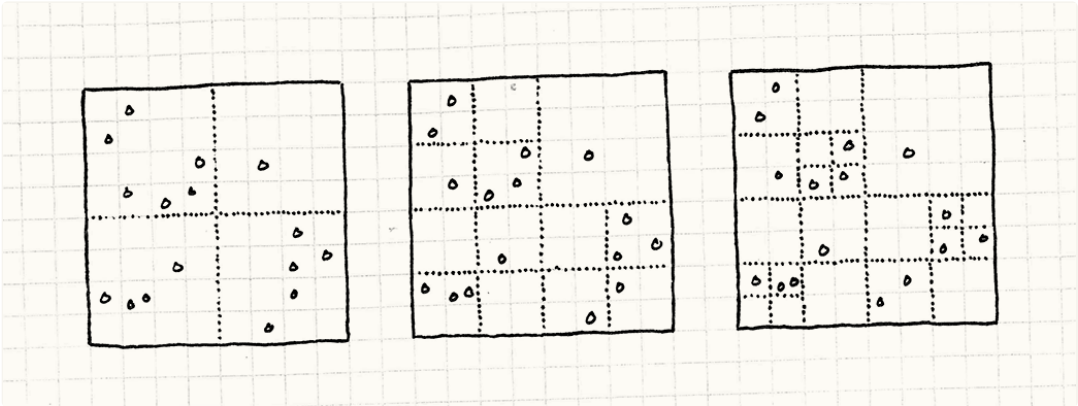
  One spatial partition deserves special mention because it has some of the best characteristics of both fixed partitions and adaptable ones: quadtrees.

  A quadtree partitions 2D space. Its 3D analogue is the *octree*, which takes a *volume* and partitions it into eight *cubes*. Aside from the extra dimension, it works the same as its flatter sibling.

  A quadtree starts with the entire space as a single partition. If the number of objects in the space exceeds some threshold, it is sliced into four smaller squares. The *boundaries* of these squares are fixed: they always slice space right in half.

  Then, for each of the four squares, we do the same process again, recursively, until every square has a small number of objects in it. Since we only recursively subdivide squares that have a high population, this partitioning adapts to the set of objects, but the partitions don't *move*.

You can see the partitioning in action reading from left to right here:



- *Objects can be added incrementally*. Adding a new object means finding the right square and adding it. If that bumps that square above the maximum count, it gets subdivided. The other objects in that square get pushed down into the new smaller squares. This requires a little work, but it's a *fixed* amount of effort: the number of objects you have to move will always be less than the maximum object count. Adding a single object can never trigger more than one subdivision.

  Removing objects is equally simple. You remove the object from its square and if the parent square's total count is now below the threshold, you can collapse those subdivisions.

- *Objects can be moved quickly*. This, of course, follows from the above. "Moving" an object is just an add and a remove, and both of those are pretty quick with quadtrees.

- *The partitions are balanced*. Since any given square will have less than some fixed maximum number of objects, even when objects are clustered together, you don't have single partitions with a huge pile of objects in them.

Are objects only stored in the partition?

You can treat your spatial partition as *the* place where the objects in your game live, or you can consider it just a secondary cache to make look-up faster while also having another collection that directly holds the list of objects.

- **If it is the only place objects are stored:**

  - *It avoids the memory overhead and complexity of two collections*. Of course, it's always cheaper to store something once instead of twice. Also, if you have two collections, you have to make sure to keep them in sync. Every time an object is created or destroyed, it has to be added or removed from both.

- **If there is another collection for the objects:**

  - *Traversing all objects is faster*. If the objects in question are "live" and have some

processing they need to do, you may find yourself frequently needing to visit every object regardless of its location. Imagine if, in our earlier example, most of the cells were empty. Having to walk the full grid of cells to find the non-empty ones can be a waste of time.

A second collection that just stores the objects gives you a way to walk all them directly. You have two data structures, one optimized for each use case.

See Also

- I've tried not to discuss specific spatial partitioning structures in detail here to keep the chapter high-level (and not too long!), but your next step from here should be to learn a few of the common structures. Despite their scary names, they are all surprisingly straightforward. The common ones are:
  - Grid
  - Quadtree
  - BSP
  - k-d tree
  - Bounding volume hierarchy
- Each of these spatial data structures basically extends an existing well-known data structure from 1D into more dimensions. Knowing their linear cousins will help you tell if they are a good fit for your problem:
  - A grid is a persistent bucket sort.
  - BSPs, k-d trees, and bounding volume hierarchies are binary search trees.
  - Quadtrees and octrees are tries.

游戏编程模式

嘿，游戏开发者们！

- 为代码整体规划而苦苦挣扎？
- 发现随着代码库增长不容易做出改动？
- 感觉你的游戏就是一个大毛线球？任何东西都和其他所有东西绕在一起？
- 考虑有什么设计模式可以应用到游戏中？
- 听说过“缓存一致性”和“对象池”，但却不知道如何使用来加速你的游戏？

我来帮忙！我写了这本书来回答这些问题。这是我在游戏中使用模式的总结，它们能让代码更整洁，更易懂，运行起来更快！

免费在线阅读！

这是我在开始做游戏时想要的书！现在，我也想让你拥有一本！

开始阅读!

作者是谁？

Bob Nystrom。他在EA工作时，就开始写这本书了。在那工作的8年里，他见过很多优美的代码，也见过很多可怕的代码。他希望能将从优美的代码中学到的东西写下来，并教大家怎样写出这样好的代码。

如果你想与他交流，你可以在Twitter上(`@munificentbob`).

有反馈？

如果对中文版有任何问题意见，不要犹豫，提交issues或者pull request.

目录

游戏编程模式

致谢

游戏设计模式

据说只有作家知道写作中会遇到什么，但还有另外一群人知道内情——那些不幸与作家有亲密关系的人。我的妻子Megan从岩石般致密的生活中，为我开凿出写作时间。洗盘子，给孩子洗澡也许不是“写作”，但没有她的这些付出，这本书永远没法写出来。

我在EA做程序员时开始写作这本书。我认为公司不知道这回事，我要感谢Michael Malone, Olivier Nallet, 以及Richard Wifall。他们为书籍的前几章提供了详尽有益的建议。

写到一半时，我决定放弃传统的出版方式。我知道，这意味着没有编辑的指导，但有成打的读者通过邮件告诉我书该怎么写；这意味着没有校对，但有超过250个bug报告来帮我改进；这意味着没有写作期限的鼓舞，但当我完成一章，读者会拍着我的背鼓励，我会有更强的动力。

> 我没有失去文字编辑。Lauren Briese在我需要的时候出现并杰出地完成了工作。

人们称之为“自出版”，但是“众出版”更加接近事实。写作是件孤独的事，但我从没感到孤单。哪怕是我停止写作两年后，仍有人来鼓励我。没有那些提醒我他们期待更多章节的人，我不会重拾工作并完成此书。

> 特别感谢Colm Sloan，完全出于内心的善意，他认真阅读了书中每个章节两遍，给了我众多超赞的反馈。我欠他一份人情，也许是两份。

那些写过邮件或者发过评论的人，那些点过赞或者收藏的人，那些发过推特的人，那些与我交流的人，那些向朋友宣传这本书的人，那些向我发送错误报告的人，我要对你们说：我心中充满了对你的感激。完成这本书是我人生中最大的目标之一，是你让我梦想成真。

谢谢！

序

游戏设计模式

在五年级时，我和我的朋友被准许使用一间存放有几台非常破旧的TRS-80s的房间。 为了鼓舞我们，一位老师给我们找了一些简单的BASIC程序打印文档。

电脑的磁带驱动器已经坏掉了，所以每当我们想要运行代码，就得小心地从头开始输入代码。 因此，我们更喜欢那些只有几行长的程序：

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

> 如果电脑打印的次数足够多，也许这句话就会魔法成真。

哪怕这样，过程也充满了困难。我们不知道如何编程，所以小小的语法错误对我们也是天险。 如果程序没有工作——这经常发生——我们就得从头再来一遍。

文档的最后几页是个真正的怪物：一个占据了几页篇幅的程序。 我们得花些时间才能鼓起勇气去试一试，但它实在太诱人——它的标题是“地道与巨魔”。 我们不知道它能做什么，但听起来像是个游戏，还有什么比自己编个电脑游戏更酷的吗？

我们从来没能让它运行起来，一年以后，我们离开了那间教室。 （很久以后，当我真的学会了点BASIC，我意识到那只是个桌面游戏角色生成器，而不是游戏。） 但是命运的车轮已经开始转动——自那时起，我就想要成为一名游戏程序员。

青少年时，我家有了一台能运行QuickBASIC的Macintosh，之后THINK C也能在其上运行。 几乎整个暑假我都在用它编游戏。 自学缓慢而痛苦。 我能轻松地编写并运行某些部分——地图或者小谜题——但随着程序代码量的增长，这越来越难。

> 暑假中的不少时间我花在路易斯安那州南部的沼泽里逮蛇和乌龟上了。 如果外面不是那么酷热，很有可能这是一本讲爬虫而不是编程的书了。

起初，挑战之处仅仅在于让程序成功运行。然后，是搞明白怎样写出内容超出我大脑容量的代码。 我不再只阅读关于“如何用C++编程”的书籍，而开始尝试找那些讲如何组织程序的书。

几年过后，一位朋友给我一本书：《设计模式：可复用面向对象软件的基础》。 终于！正是我从青年时期就在寻找的书。 我一口气从头读到尾。虽然我仍然与自己的程序挣扎，但看到别人也在挣扎并提出了解决方案是一种解脱。 我意识到手无寸铁的我终于有件像样的工具了。

> 那是我首次见到这位朋友，相互介绍五分钟后，我坐在他的沙发上，接下来几个小时，无视他并全神贯注地阅读。 我想自那以后我的社交技能还是有所提高的。

在2001年，我获得了梦想中的工作：EA的软件工程师。 我等不及要看看真正的游戏，还有专业人士是如何将组织一切的。 像实况足球这样的大型游戏使用了什么样的架构？不同的系统是如何交互的？一套代码库是如何在多个平台上运行的？

分析理解源代码是种震颤的体验。图形，AI，动画，视觉效果皆有杰出代码。 有专家知道如何榨干CPU的最后一个循环并好好使用。 那些我都不知道是否可行的事情，这些人在午饭前就能完成。

但是这些杰出代码依赖的架构通常是事后设计。 他们太注重功能而忽视了架构。耦合充斥在模块间。 新功能被塞到任何能塞进去的地方。 在梦想幻灭的我看来，这和其他程序员没什么不同， 如果他们阅读过《设计模式》，最多也就用用 单例 。

当然，没那么糟。我曾幻想游戏程序员坐在白板包围的象牙塔里，为架构冷静地讨论上几周。 而实际情况，我看到的代码是努力应对紧张截止期限的人赶工完成的。 他们已经竭尽全力，而且，就像我慢慢意识到的那样，他们的全力以赴的结果通常很好。 我花在游戏代码上的时间越多，我越能发现藏在表面下的天才之处。

不幸的是，“藏”是普遍现象。 宝石埋在代码中，但人们从未意识到它们的存在。 我看到同事重复寻找解决方案，而需要的示例代码就埋在他们所用的代码库中。

这个问题正是这本书要解决的。 我挖出了游戏代码库中能找到的设计模式，打磨然后在这里展示它们，这样可以节约时间用于在发明新事物上，而非重新发明它们。

书店里已有的书籍

书店里已经有很多游戏编程书籍了。为什么要再写一本呢？

我看到的很多编程书籍可以归为这两类：

- 特定领域的书籍。 这些关于细分领域的书籍带你深入理解游戏开发某一特定层面。 它们会教授你3D图形，实时渲染，物理模拟，人工智能，或者音频播放。 那些很多程序员穷其一生研究的细分领域。
- 完整引擎的书籍。 另一个方向，还有书籍试图包含游戏引擎的各个部分。 它们倾向于构建特定种类游戏的完整引擎，通常是3D FPS游戏。

这两种书我都喜欢，但我认为它们并未覆盖全部空间。 特定领域的书籍很少告诉你这些代码如何与游戏的其他部分打交道。 你擅长物理或者渲染，但是你知道怎么将两者优雅地组合吗？

第二类书包含这些，但是发现完整引擎的书籍通常过于整体，过于专注某类游戏了。 特别是，随着手游和休闲游戏的兴起，我们正处于众多游戏类型欣欣向荣的时刻。 我们不再只是复制Quake了。如果你的游戏与该类游戏不同，那些介绍单一引擎的书就不那么有用了。

相反，我在这里做的更**à la carte** 。 每一章都是独立的，可应用到代码上的思路。 这样，你可以用你认为最好的方式组合这些思路，用到你的游戏上去。

> 另一个广泛使用这种à la carte风格的例子是Game Programming Gems系列。

和设计模式的关联

任何名字中有“模式”的编程书 都与Erich Gamma，Richard Helm，Ralph Johnson，和John Vlissides（通常被称为GoF）合著的经典书籍： 《设计模式：可复用面向对象软件要素》相关。

> 《设计模式》也受到之前的书籍的启发。 创建一种模式语言来描述问题的开放式解法， 这思路来自 A Pattern Language， 作者是Christopher Alexander (还有Sarah Ishikawa和Murray Silverstein).
>
> 他们的书是关于架构的（建筑和墙那样的真正的框架结构）， 但他们希望其他人能使用相同的方法描述其他领域的解决方案。 《设计模式》正是是GoF用这一方法在软件业做出的努力。

称这本书为“游戏编程模式”，我不是暗示GoF的模式不适用于游戏编程。 相反：本书重返设计模式一节包含了《设计模式》中的很多模式， 但强调了这些模式在游戏编程的特定使用。

同样的，我认为本书也适用于非游戏软件。 我可以依样画瓢称本书为《更多设计模式》，但是我认为举游戏编程为例子更为契合。 你真的想要另一本介绍员工记录和银行账户的书吗？

也就是说，虽然这里介绍的模式在其他软件上也很有用，但它们更合适于处理游戏中常见的工程挑战：

- 时间和顺序通常是游戏架构的核心部分。事物必须在正确的时间按正确的顺序发生。
- 高度压缩的开发周期，大量程序员需要能快速构建和迭代一系列不同的行为，同时保证不烦扰他人，也不污染代码库。
- 在定义所有的行为后，游戏开始互动。怪物攻击英雄，药物相互混合，炸弹炸飞敌人或者友军。 实现这些互动不能把代码库搞成一团乱麻。
- 最后，游戏中性能很重要。 游戏开发者处于一场榨干平台性能的竞赛中。 节约CPU循环的技巧区分了A级百万销量游戏和掉帧差评游戏。

如何阅读这本书

《游戏设计模式》分为三大块。 第一部分介绍并划分本书的框架。包含你现在阅读的这章和下一章。

第二部分，重放设计模式，复习了GoF书籍里的很多模式。 在每一章中，我给出我对这个模式的看法，以及我认为它和游戏编程有什么关系。

最后一部分是这本书最肥美的部分。 它展示了十三种我发现有用的模式。它们被分为四类：序列模式, 行为模式, 解耦模式,和优化模式。

每种模式都使用固定的格式表述，这样你可以将这本书当成引用，快速找到你需要的：

- 意图 部分提供这个模式想要解决什么问题的简短介绍。 将它放在首位，这样你可以快速翻阅，找到你现在需要的模式。
- 动机 部分描述了模式处理的问题示例。 不同于具体的算法，模式通常不针对某个特定

问题。 不用示例教授模式，就像不用面团教授烘烤。动机部分提供了面团，而下部分会教你烘烤。

- 模式 部分将模式从示例中剥离出来。 如果你想要一段对模式的教科书式简短介绍，那就是这部分了。 如果你已经熟悉了这种模式，想要确保你没有拉下什么，这部分也是很好的提示。
- 到目前为止，模式只是用一两个示例解释。但是如何知道模式对你的问题有没有用呢？ 何时使用 部分提供了这个模式在何时使用何时不用的指导。 记住 部分指出了使用模式的结果和风险。
- 如果你像我一样需要具体的例子真正来理解某物，那么示例代码部分能让你称心如意。它描述模式的一步步具体实现，来展现模式是如何工作的。
- 模式与算法不同的是它们是开放的。 每次你使用模式，可以不同的方式实现。 下一部分设计决策，讨论这些方式，告诉你应用模式时可供考虑的不同选项。
- 作为结尾，这里有参见部分展示了这一模式与其他模式的关联，以及那些使用它的真实代码。

关于示例代码

这本书的示例代码使用C++写就，但这并不意味着这些模式只在C++中有用，或C++比其他语言更适合使用这些模式。 这些模式适用于几乎每种编程语言，虽然有的模式假设编程语言有对象和类。

我选择C++有几个原因。首先，这是在游戏制作中最流行的语言，是业界的通用语。 通常，C++基于的C语法也是Java，C#，JavaScript和其他很多语言的基础。 哪怕你不懂C++，你也只需一点点努力就能理解这里的示例代码。

这本书的目标不是教会你C++。 示例代码尽可能的简单，不一定符合好的C++风格或规范。 示例代码展示的是意图，而不是代码。

特别的，代码没用“现代的”——C++11或者更新的——标准。 没有使用标准库，很少使用模板。 它们是“糟糕”C++代码，但我希望保持这样，这样那些使用C，Objective-C，Java和其他语言的人更容易理解它们。

为了避免花费时间在你已经看过或者是与模式无关的代码上，示例中省略了部分代码。 如果是那样，示例代码中的省略号表明这里隐藏了一些代码。

假设有个函数，做了些工作然后返回值。 而用它作示例的模式只关心返回的值，而不是完成了什么工作。那样的话，示例代码长的像这样：

```
bool update()
{
  // 做点工作……
  return isDone();
}
```

接下来呢

设计模式在软件开发过程中不断地改变和扩展。 这本书继续了GoF记录分享设计模式的旅程，而这旅程也不会终于本书。

你是这段旅程的关键部分。改良（或者否决）了这本书中的模式，你就是为软件开发社区做贡献。 如果你有任何建议，更正，或者任何反馈，保持联络！

← 上一章 ≡ 首页 下一章 →

架构，性能和游戏

游戏设计模式 / Introduction

在一头扎入一堆设计模式之前，我想先讲一些我对软件架构和其应用到游戏之中的理解，这也许能帮你更好地理解这本书的其余部分。 至少，在你被卷入一场关于设计模式和软件架构有多么糟糕（或多么优秀）的辩论时，这可以给你一些火力支援。

> 注意我没有建议你在战斗中选哪一边。就像任何军火贩子一样，我愿意向作战双方出售武器。

什么是软件架构？

如果把本书从头到尾读一遍， 你不会学会3D图形背后的线性代数或者游戏物理背后的微积分。 这书不会告诉你如何用α-β修剪你的AI树，也不会告诉你如何在音频播放中模拟房间中的混响。

> Wow，这段给这本书打了个糟糕的广告啊。

相反，这本书告诉你在这些之间的事情。 与其说这本书是关于如何写代码，不如说是关于如何架构代码的。 每个程序都有一定架构，哪怕这架构是“将所有东西都塞到`main()`中看看如何”， 所以我认为讲讲什么造成了好架构是很有意思的。我们如何区分好架构和坏架构呢？

我思考这个问题五年了。当然，像你一样，我有对好设计有一种直觉。 我们都被糟糕的代码折磨的不轻，你唯一能做的好事就是删掉它们，结束它们的痛苦。

> 承认吧，我们中大多数人都该对一些糟糕代码负责。

少数幸运儿有相反的经验，有机会在好好设计的代码库上工作。 那种代码库看上去是间豪华酒店，里面的门房随时准备满足你心血来潮的需求。 这两者之间的区别是什么呢？

什么是好的软件架构？

对我而言，好的设计意味着当我作出改动，整个程序就好像正等着这种改动。 我可以加使用几个函数调用完成任务，而代码库本身无需改动。

这听起来很棒，只是完全不可行。“把代码写到改动不会影响其平静表面。”够了。

让我们通俗些。第一个关键点是架构是有关于改动的。 总有人改动代码。如果没人碰代码——无论是因为代码至善至美， 还是因为代码糟糕透顶以至于没人会为了修改它而玷污自己

的文本编辑器——那么它的架构设计就无关紧要。 评价架构设计的好坏就是评价它应对改动有多么轻松。 没有了改动，架构好似永远不会离开起跑线的运动员。

你如何处理改动？

在你改动代码去添加新特性，去修复漏洞，或者随便什么需要使用文本编辑器的时候， 你需要理解代码正在做什么。当然，你不需要理解整个程序， 但你需要将所有相关的东西装进你的大脑。

> 有点诡异，这字面上是一个OCR过程。

我们通常无视了这步，但这往往是编程中最耗时的部分。 如果你认为将数据从磁盘上分页到RAM上很慢， 那么试试通过一对神经纤维将数据分页到大脑中。

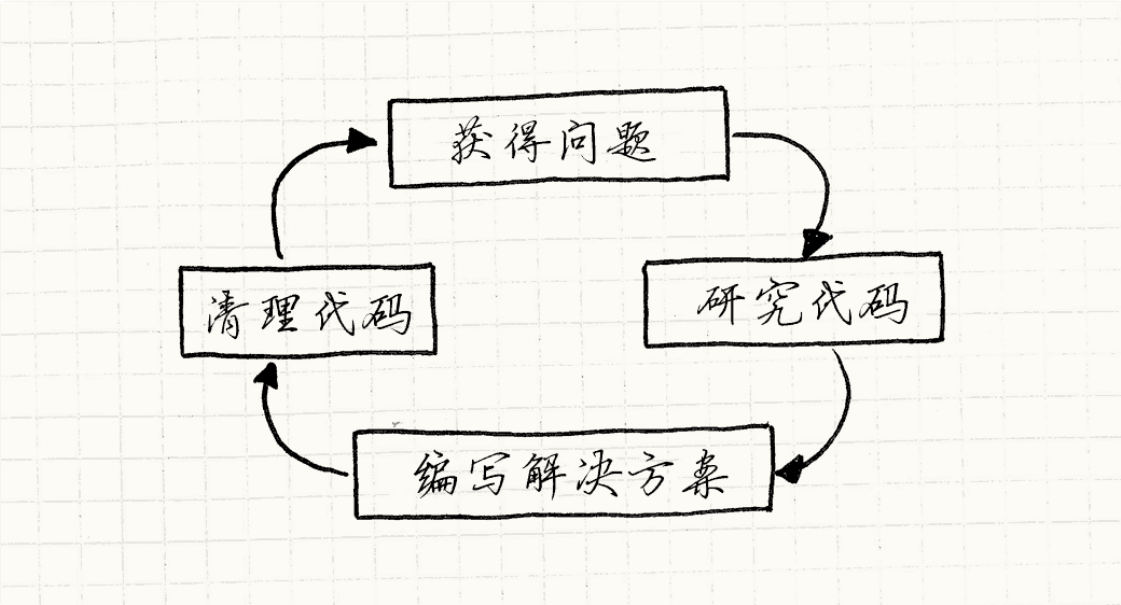
一旦把所有正确的上下文都记到了你的大脑里， 想一会，你就能找到解决方案。 可能需要反复斟酌的时刻，但通常比较简单。 一旦理解了问题和需要改动的代码，实际的编码工作有时是微不足道的。

你用手指在键盘上敲打一阵，直到屏幕上闪着正确的光芒， 搞定了，对吧？还没呢！ 在你为之写测试并发送到代码评审之前，你通常有些清理工作要做。

> 我是不是说了“测试”？噢，是的，我说了。为有些游戏代码写单元测试很难，但代码库的大部分是完全可以测试的。
>
> 我不会在这里发表演说，但是我建议你，如果还没有做自动测试，请考虑一下。除了手动验证以外你就没别的事要做了吗？

你将一些代码加入了游戏，但不想下一个人被留下来的小问题绊倒。 除非改动很小，否则就还需要一些工作去微调新代码，使之无缝对接到程序的其他部分。 如果你做对了，那么下个编写代码的人无法察觉到哪些代码是新加入的。

简而言之，编程的流程图看起来是这样的：



> 令人震惊的死循环，我看到了。

解耦帮了什么忙？

虽然并不明显，但我认为很多软件架构都是关于研究代码的阶段。 将代码载入到神经元太

过缓慢，找些策略减少载入的总量是件很值得做的事。 这本书有整整一章是关于解耦模式，还有很多设计模式是关于同样的主题。

可以用多种方式定义“解耦”，但我认为如果有两块代码是耦合的， 那就意味着无法只理解其中一个。 如果解耦了它俩，就可以独自地理解某一块。 这当然很好，因为只有一块与问题相关， 只需将这一块加载到你的大脑脑中而不需要加载另外一块。

对我来说，这是软件架构的关键目标： 最小化在编写代码前需要了解的信息。

当然，也可以从后期阶段来看。 另一种解耦的定义是：当一块代码有改动时，没必要修改另一块的代码。 肯定需要修改一些东西，但耦合程度越小，改动会波及的范围就越小。

代价呢？

听起来很棒，对吧？解耦任何东西，然后像风一样编码。 每个改动都只需修改一两个特定方法，在代码库上行云流水地编写代码。

这就是人们对抽象，模块化，设计模式和软件架构兴奋的原因。 在架构优良的程序上工作是极佳的体验，每个人都希望能更有效率地工作。 好架构能造成生产力上巨大的不同。它影响大得无以复加。

但是，天下没有免费的午餐。好的设计需要汗水和纪律。 每次做出改动或是实现特性，你都需要将它优雅的集成到程序的其他部分。 需要花费大量的努力去管理代码， 在开发过程中面对数千次变化仍然保持它的结构。

> 第二部分——管理代码——需要特别关注。 我看到无数程序有优雅的开始，然后死于程序员一遍又一遍添加的“微小黑魔法”。
>
> 就像园艺，仅仅种植是不够的，还需要除草和修剪。

你得考虑程序的哪部分需要解耦，然后再引入抽象。 同样，你需要决定哪部分能支持扩展来应对未来的改动。

人们对这点变得狂热。 他们设想，未来的开发者（或者他们自己）进入代码库， 发现它极为开放，功能强大，只需扩展。 他们想要有“至尊代码应众求”。（译著：这里是“至尊魔戒御众戒”的梗，很遗憾翻译不出来）

但是，事情从这里开始变得棘手。 每当你添加了抽象或者扩展支持，你是赌以后这里需要灵活性。 你向游戏中添加了代码和复杂性，这需要时间来开发，调试和维护。

如果你赌对了，后来使用了这些代码，那么功夫不负有心人。 但预测未来很难，模块化如果最终无益，那就有害。 毕竟，你得处理更多的代码。

> 有些人喜欢使用术语“YAGNI”——You aren’t gonna need it（你不需要那个）——来对抗这种预测将来需求的强烈冲动。

当你过分关注这点时，代码库就失控了。 接口和抽象无处不在。插件系统，抽象基类，虚方法，还有各种各样的扩展点，它们遍地都是。

你要消耗无尽的时间回溯所有的脚手架，去找真正做事的代码。 当需要作出改动，当然，有可能某个接口能帮上忙，但能不能找到就只能听天由命了。 理论上，解耦意味着在修改代码之前需要了解更少的代码， 但抽象层本身也会填满大脑。

像这样的代码库让人反对软件架构，特别是设计模式。 人们很容易沉浸在代码中，忽略目

标是要发布游戏。 无数的开发者听着加强可扩展性的警报，花费多年时间制作“引擎”， 却没有搞清楚做引擎是为了什么。

性能和速度

软件架构和抽象有时因损伤性能而被批评，而游戏开发尤甚。 让代码更灵活的许多模式依靠虚拟调度、 接口、 指针、 消息和其他机制， 它们都会加大运行时开销。

> 一个有趣的反面例子是C++中的模板。模板编程有时可以给你抽象接口而无需运行时开销。
>
> 这是灵活性的两极。 当写代码调用类中的具体方法时，你在写的时候指定类——硬编码了调用的是哪个类。 通过虚方法或接口，直到运行时才知道调用的类。这更加灵活但增加了运行时开销。
>
> 模板编程是在两极之间。在编译时初始化模板，决定调用哪些类。

还有一个原因。很多软件架构的目的是使程序更加灵活，作出改动需要更少的付出，编码时对程序有更少的假设。 你可以使用接口，让代码可与任何实现了接口的类交互，而不仅仅是现在写的类。 今天，你可以使用观察者 GoF和消息 让游戏的两部分交流， 以后可以很容易地扩展为三个或四个部分相互交流。

但性能与假设相关。实现优化需要基于确定的限制。 敌人永远不会超过256个？好，可以将敌人ID编码为一个字节。 只在这种类型上调用方法吗？好，可以做静态调度或内联。 所有实体都是同一类？太好了，可以使用 连续数组 存储它们。

但这并不意味着灵活性不好！它可以让我们快速改进游戏， 开发 速度是获取有趣开发经验的绝对重要因素。 没有人，哪怕是Will Wright，能在纸面上构建一个平衡的游戏。这需要迭代和实验。

越快尝试想法看看效果，就能尝试越多东西，越可能找到有价值的东西。 就算找到正确的机制，你也需要足够的时间调整。 一个微小的不平衡就有可能破坏整个游戏的乐趣。

这里没有普适的答案。 让你的程序更加灵活，在损失一点点性能的前提下更快地做出原型。 或者，优化代码，损失一些灵活性。

就我个人经验而言，让有趣的游戏变得高效比让高效的游戏变有趣简单得多。 一种折中的办法是保持代码灵活直到确定设计，再去除抽象层来提高性能。

糟糕代码的优势

下一观点：不同的代码风格各有千秋。 这本书的大部分是关于保持干净可控的代码，所以我坚持应该用正确方式写代码，但糟糕的代码也有一定的优势。

编写良好架构的代码需要仔细地思考，这会消耗时间。 在项目的整个周期中保持良好的架构需要花费大量的努力。 你需要像露营者处理营地一样小心处理代码库：总是保持其优于你刚刚接触它的时候。

当你要在项目上花费很久时间的话，这很好。 但，就像早先提到的，游戏设计需要很多实验和探索。 特别是在早期，写一些你知道要扔掉的代码是很普遍的事情。

如果只想试试游戏的某些点子是否可行的， 良好的架构意味着在屏幕上看到和获取反馈之

前要消耗很长时间。 如果最后证明这点子不对，那么删除代码时，那些让代码更优雅的时间就付之东流了。

原型——一坨勉强拼凑在一起，只能完成某个点子的简单代码——是个完全合理的编程实践。 虽然当你写一次性代码时，必须 保证将来可以扔掉它。 我见过很多次糟糕的经理人在玩这种把戏：

> 老板："嗨，我有些想试试的点子。只要原型，不需要做的很好。你能多快搞定？"
>
> 开发者："额，如果删掉这些部分，不测试，不写文档，允许很多的漏洞，那么几天能给你临时的代码文件。"
>
> 老板："太好了。"

几天后

> 老板："嘿，原型很棒，你能花上几个小时清理一下然后变为成品吗？"

你得让人们清楚，可抛弃的代码即使看上去能工作，也不能被维护，必须 重写。 如果有可能要维护这段代码，就得防御性好好编写它。

> 一个小技巧能保证原型代码不会变成真正用的代码：使用和游戏实现不同的编程语言。 这样，在将其实际应用于游戏中之前必须重写。

保持平衡

有些因素在相互角力：

1. 为了在项目的整个生命周期保持其可读性，需要好架构。 2. 需要更好的运行时性能。 3. 需要让现在的特性更快的实现。

> 有趣的是，这些都是速度：长期开发的速度，游戏运行的速度，和短期开发的速度。

这些目标至少是部分对立的。 好架构长期来看提高了生产力， 也意味着每个改动都需要消耗更多努力保持代码整洁。

草就的代码很少是运行时最快的。 相反，提升性能需要很多的编程时间。 一旦完成，它就会污染代码库：高度优化的代码不灵活，很难改动。

总有今日事今日毕的压力。但是如果尽可能快地实现特性， 代码库就会充满黑魔法，漏洞和混乱，阻碍未来的产出。

没有简单的答案，只有权衡。 从我收到的邮件看，这伤了很多人的心，特别是那些只是想做个游戏的人。 这似乎是在恐吓，"没有正确的答案，只有不同的错误。"

但，对我而言，这让人兴奋！看看任何人们从事的领域， 你总能发现某些相互抵触的限制。无论如何，如果有简单的答案，每个人都会那么做。 一周就能掌握的领域是很无聊的。你从来没有听说过有人讨论挖坑。

> 也许你会讨论挖坑；我没有深究这个类比。 可能有挖坑热爱者，挖坑规范，以及一整套亚文化。 我算什么人，能在此大放厥词？

对我来说，这和游戏有很多相似之处。 国际象棋之类的游戏永远不能被掌握，因为每个棋子都很完美的与其他棋子相平衡。 这意味你可以花费一生探索广阔的可选策略。糟糕的游戏就像井字棋，玩上几遍就会厌倦就退出。

简单

最近，我感觉如果有什么能简化这些限制，那就是简单。 在我现在的代码中，我努力去写最简单，最直接的解决方案。 你读过这种代码后，完全理解了它在做什么，想不到其他完成的方法。

我的目标是正确获得数据结构和算法（大致是这样的先后），然后在从那里开始。 我发现如果能让事物变得简单，就有更少的代码， 就意味着改动时有更少的代码载入脑海。

它通常跑的很快，因为没什么开销，也没什么代码需要执行。 （虽然大部分时候事实并非如此。你可以在一小段代码里加入大量的循环和递归。）

但是，注意我并没有说简单的代码需要更少的时间编写。 你会这么觉得是因为最终得到了更少的代码，但是好的解决方案不是往代码中注水，而是蒸干代码。

> Blaise Pascal有句著名的信件结尾，“我没时间写的更短。”
>
> 另一句名言来自Antoine de Saint-Exupery：“臻于完美之时，不是加无可加，而是减无可减。”
>
> 言归正传，我发现每次重写本书，它就更短。有些章节比刚完成时短了20%。

我们很少遇到优雅表达的问题，一般反而是一堆用况。 你想要X在Z情况下做Y，在A情况下做W，诸如此类。换言之，一长列不同行为。

最节约心血的方法是为每段用况编写一段代码。 看看新手程序员，他们经常这么干：为每个手头的问题编写条件逻辑。

但这一点也不优雅，那种风格的代码遇到一点点没想到的输入就会崩溃。 当我们想象优雅的代码时，想的是通用的那一个： 只需要很少的逻辑就可以覆盖整个用况。

找到这样的方法有点像模式识别或者解决谜题。 需要努力去识别散乱的用例下隐藏的规律。 完成时你会感觉好得不能再好。

就快完了

几乎每个人都会跳过介绍章节，所以祝贺你看到这里。 我没有太多东西回报你的耐心，但还有些建议给你，希望对你有用：

- 抽象和解耦让扩展代码更快更容易，但除非确信需要灵活性，否则不要在这上面浪费时间。
- 在整个开发周期中考虑并为性能设计，但是尽可能推迟那些底层的，基于假设的优化，那会锁死代码。

> 相信我，发布前两个月不是开始思考“游戏运行只有1FPS”问题的时候。

- 快速地探索游戏的设计空间，但不要跑的太快，在身后留下烂摊子。毕竟，你总得回来打扫。

- 如果打算抛弃这段代码，就不要尝试将其写完美。摇滚明星将旅店房间弄得一团糟，因为他们知道明天他们就走人了。
- 但最重要的是，如果你想要做出让人享受的东西，那就享受做它的过程。

← 上一章 ≡ 首页 下一章 →

重访设计模式

游戏设计模式

《设计模式：可复用面向对象软件的基础》出版已经二十年了。 除非你比我从业还久，否则《设计模式》已经酝酿成一坛足以饮用的老酒了。 对于像软件行业这样快速发展的行业，它已经是老古董了。 这本书的持久流行证明了设计方法比框架和方法论更经久不衰。

虽然我认为设计模式仍然有意义，但在过去几十年我们学到了很多。 在这一部分，我们会遇到GoF记载的一些模式。 对于每个模式，我希望能讲些有用有趣的东西。

我认为有些模式被过度使用了（单例模式）， 而另一些被冷落了（命令模式）。 有些模式在这里是因为我想探索其在游戏上的特殊应用（享元模式和观察者模式）。 最后，我认为看看有些模式在更广的编程领域是如何运用的是很有趣的（原型模式和状态模式）。

模式

- 命令模式
- 享元模式
- 观察者模式
- 原型模式
- 单例模式
- 状态模式

命令模式

游戏设计模式 / Design Patterns Revisited

命令模式是我最喜欢的模式之一。 大多数我写的大型程序，游戏或者别的什么，都会在某处用到它。 当在正确的地方使用时，它可以将复杂的代码清理干净。 对于这样一个了不起的模式，不出所料，GoF有个深奥的定义：

> 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化； 对请求排队或记录请求日志，以及支持可撤销的操作。

我想你也会觉得这个句子晦涩难懂。 第一，它的比喻难以理解。 在软件之外的狂野世界，词语可以指代任何事物，“客户”是一个人——那些和你做生意的人。 据我查证，人类不能被“参数化”。

然后，句子余下的部分介绍了可能会使用这个模式的场景。 如果你的场景不在这个列表中，那么这对你就没什么用处。 我的命令模式精简定义为：

命令是具现化的方法调用。

> “Reify（具现化）”来自于拉丁语“res”，意为“thing”（事物），加上英语后缀“-fy”。 所以它意为“thingify”，没准用“thingify”更合适。

当然，“精简”往往意味着着“缺少必要信息”，所以这可能没有太大的改善。 让我扩展一下。“具现化”，如果你没有听说过的话，它的意思是“实例化，对象化”。 另外一种具现化的解释方式是将某事物作为“第一公民”对待。

> 在某些语言中的反射允许你在程序运行时命令式地和类型交互。 你可以获得类的类型对象，可以与其交互看看这个类型能做什么。换言之，反射是具现化类型的系统。

两种术语都意味着将概念变成数据——一个对象——可以存储在变量中，传给函数。 所以称命令模式为“具现化方法调用”，意思是方法调用被存储在对象中。

这听起来有些像“回调”，“第一公民函数”，“函数指针”，“闭包”，“偏函数”， 取决于你在学哪种语言，事实上大致上是同一个东西。GoF随后说：

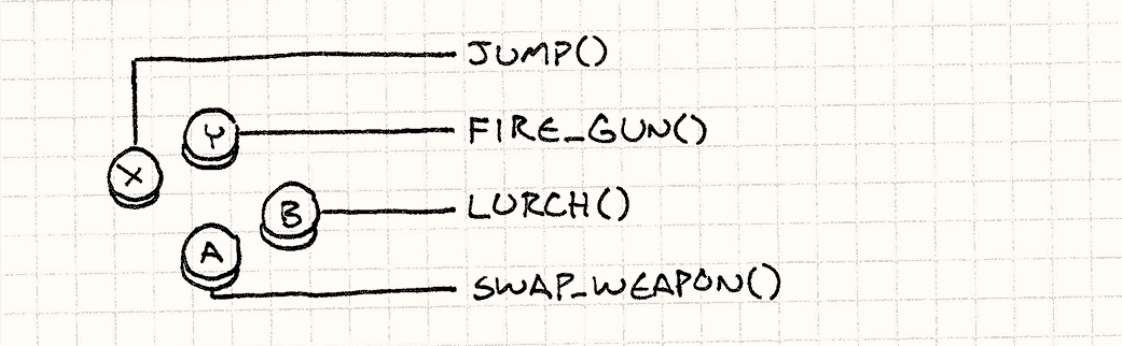
> 命令模式是一种回调的面向对象实现。

这是一种对命令模式更好的解释。

但这些都既抽象又模糊。我喜欢用实际的东西作为章节的开始，不好意思，搞砸了。 作为弥补，从这里开始都是命令模式能出色应用的例子。

配置输入

在每个游戏中都有一块代码读取用户的输入——按钮按下，键盘敲击，鼠标点击，诸如此类。 这块代码会获取用户的输入，然后将其变为游戏中有意义的行为：



下面是一种简单的实现：

```
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();
}
```

> 专家建议：不要太经常地按B。

这个函数通常在游戏循环中每帧调用一次，我确信你可以理解它做了什么。 在我们想将用户的输入和程序行为硬编码在一起时，这段代码可以正常工作，但是许多游戏允许玩家配置按键的功能。

为了支持这点，需要将这些对`jump()`和`fireGun()`的直接调用转化为可以变换的东西。 “变换”听起来有点像变量干的事，因此我们需要表示游戏行为的对象。进入：命令模式。

我们定义了一个基类代表可触发的游戏行为：

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

> 当你有接口只包含一个没有返回值的方法时，很可能你可以使用命令模式。

然后我们为不同的游戏行为定义相应的子类：

```
class JumpCommand : public Command
{
public:
```

```
  virtual void execute() { jump(); }
};

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

// 你知道思路了吧
```

在代码的输入处理部分，为每个按键存储一个指向命令的指针。

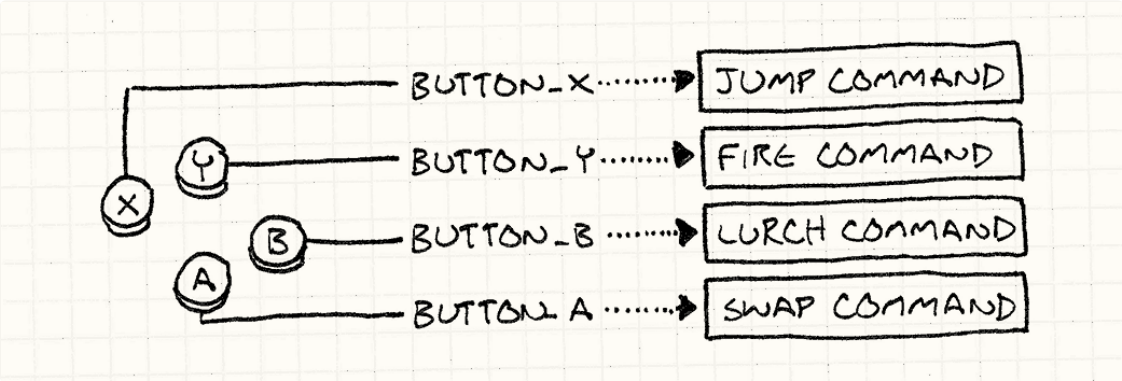
```
class InputHandler
{
public:
  void handleInput();

  // 绑定命令的方法……

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

现在输入处理部分这样处理：

```
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();
}
```

> 注意在这里没有检测NULL了吗？这假设每个按键都与某些命令相连。
>
> 如果想支持不做任何事情的按键又不想显式检测NULL，我们可以定义一个命令类，它的execute()什么也不做。 这样，某些按键处理器不必设为NULL，只需指向这个类。这种模式被称为空对象。

以前每个输入直接调用函数，现在会有一层间接寻址：



这是命令模式的简短介绍。如果你能够看出它的好处，就把这章剩下的部分作为奖励吧。

角色说明

我们刚才定义的类可以在之前的例子上正常工作，但有很大的局限。 问题在于假设顶层的`jump()`,`fireGun()`之类的函数可以找到玩家的角色，然后像操纵木偶一样操纵它。

这些假定的耦合限制了这些命令的用处。`JumpCommand`只能 让玩家的角色跳跃。让我们放松这个限制。 不让函数去找它们控制的角色，我们将函数控制的角色对象传进去：

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

这里`GameActor`是代表游戏世界中角色的“游戏对象”类。 我们将其传给`execute()`，这样可以在命令的子类中添加函数，来与我们选择的角色关联，就像这样：

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

现在，我们可以使用这个类让游戏中的任何角色跳来跳去了。 在输入控制部分和在对象上调用命令部分之间，我们还缺了一块代码。 第一，我们修改`handleInput()`，让它可以返回命令：

```
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_;
```

```
  // 没有按下任何按键，就什么也不做
  return NULL;
}
```

这里不能立即执行，因为还不知道哪个角色会传进来。 这里我们享受了命令是具体调用的好处——延迟到调用执行时再知道。

然后，需要一些接受命令的代码，作用在玩家角色上。像这样：

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

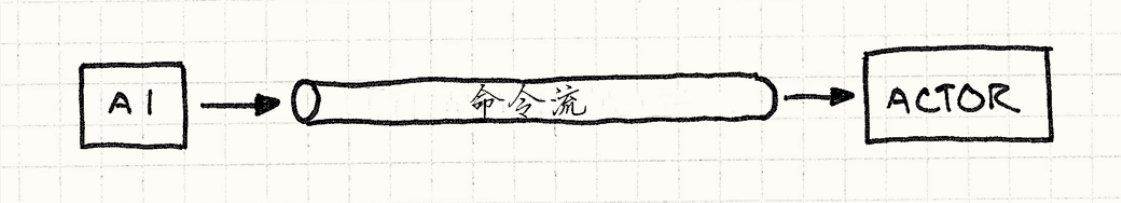
将actor视为玩家角色的引用，它会正确的按着玩家的输入移动， 所以我们赋予了角色和前面例子中相同的行为。 通过在命令和角色间增加了一层重定向， 我们获得了一个灵巧的功能：我们可以让玩家控制游戏中的任何角色，只需向命令传入不同的角色。

在实践中，这个特性并不经常使用，但是经常会有类似的用例跳出来。 到目前为止，我们只考虑了玩家控制的角色，但是游戏中的其他角色呢？ 它们被游戏AI控制。我们可以在AI和角色之间使用相同的命令模式；AI代码只需生成Command对象。

在选择命令的AI和展现命令的游戏角色间解耦给了我们很大的灵活度。 我们可以对不同的角色使用不同的AI，或者为了不同的行为而混合AI。 想要一个更加有攻击性的同伴？插入一个更加有攻击性的AI为其生成命令。 事实上，我们甚至可以为玩家角色加上AI， 这在原型阶段，游戏需要自动演示时是很有用的。

把控制角色的命令变为第一公民对象，去除直接方法调用中严厉的束缚。 将其视为命令队列，或者是命令流：

> 队列能为你做的更多事情，请看事件队列。



> 为什么我觉得需要为你画一幅“流”的图像？又是为什么它看上去像是管道？

一些代码（输入控制器或者AI）产生一系列命令放入流中。 另一些代码（调度器或者角色自身）调用并消耗命令。 通过在中间加入队列，我们解耦了消费者和生产者。

> 如果将这些指令序列化，我们可以通过网络流传输它们。 我们可以接受玩家的输入，将其通过网络发送到另外一台机器上，然后重现之。这是网络多人游戏的基础。

撤销和重做

最后的这个例子是这种模式最广为人知的使用情况。 如果一个命令对象可以做一件事，那么它亦可以撤销这件事。 在一些策略游戏中使用撤销，这样你就可以回滚那些你不喜欢的操作。 在人们创造游戏时，这是必不可少的工具。 一个不能撤销误操作导致的错误的编辑器，肯定会让游戏设计师恨你。

> 这是经验之谈。

没有了命令模式，实现撤销非常困难，有了它，就是小菜一碟。 假设我们在制作单人回合制游戏，想让玩家能撤销移动，这样他们就可以集中注意力在策略上而不是猜测上。

我们已经使用了命令来抽象输入控制，所以每个玩家的举动都已经被封装其中。 举个例子，移动一个单位的代码可能如下：

```
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_;
};
```

注意这和前面的命令有些许不同。 在前面的例子中，我们需要从修改的角色那里抽象命令。 在这个例子中，我们将命令绑定到要移动的单位上。 这条命令的实例不是通用的“移动某物”命令；而是游戏回合中特殊的一次移动。

这展现了命令模式应用时的一种情形。 在某些情形中，就像之前例子，指令是可重用的对象，代表了可执行的事件。 我们早期的输入控制器将其实现为一个命令对象，然后在按键按下时调用其`execute()`方法。

这里的命令更加特殊。它们代表了特定时间点能做的特定事件。 这意味着输入控制代码可以在玩家下决定时创造一个实例。就像这样：

```
Command* handleInput()
{
  Unit* unit = getSelectedUnit();

  if (isPressed(BUTTON_UP)) {
```

```
    // 向上移动单位
    int destY = unit->y() - 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  if (isPressed(BUTTON_DOWN)) {
    // 向下移动单位
    int destY = unit->y() + 1;
    return new MoveUnitCommand(unit, unit->x(), destY);
  }

  // 其他的移动……

  return NULL;
}
```

当然，在像C++这样没有垃圾回收的语言，这意味着执行命令的代码也要负责释放内存。

命令是一次性为我们很快地赢得了一个优点。 为了让指令可被取消，我们为每个类定义另一个需要实现的方法：

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

`undo()`方法回滚了`execute()`方法造成的游戏状态改变。 这里是添加了撤销功能后的移动命令：

```
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()
  {
    // 保存移动之前的位置
    // 这样之后可以复原。

    xBefore_ = unit_->x();
```

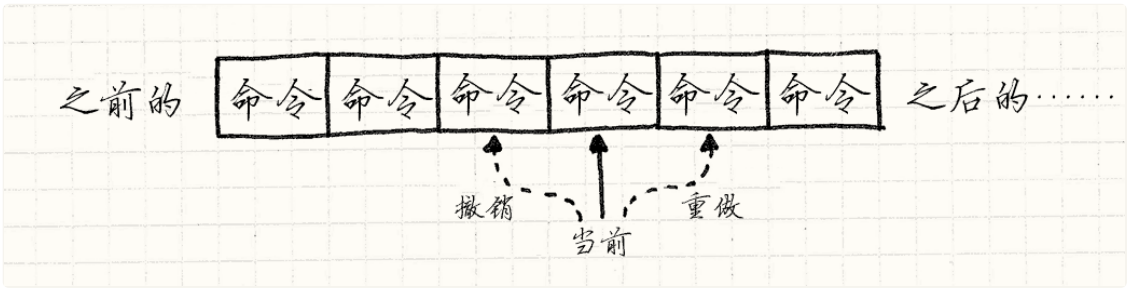
```
    yBefore_ = unit_->y();

    unit_->moveTo(x_, y_);
  }

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

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

注意我们为类添加了更多的状态。 当单位移动时，它忘记了它之前是什么样的。 如果我们想要撤销这个移动，我们需要记得单位之前的状态，也就是`xBefore_`和`yBefore_`的作用。

> 这看上去是备忘录 GoF模式使用的地方，它从来没有有效的工作过。 由于命令趋向于修改对象状态的一小部分，对数据其他部分的快照就是浪费内存。手动存储改动的部分消耗更小。
>
> *持久化数据结构*是另一个选项。 使用它，每次修改对象都返回一个新对象，保持原来的对象不变。巧妙的实现下，这些新对象与之前的对象共享数据，所以比克隆整个对象开销更小。
>
> 使用持久化数据结构，每条命令都存储了命令执行之前对象的引用，而撤销只是切换回之前的对象。

为了让玩家撤销移动，我们记录了执行的最后命令。当他们按下`control+z`时，我们调用命令的`undo()`方法。 （如果他们已经撤销了，那么就变成了“重做”，我们会再一次执行命令。）

支持多重的撤销也不太难。 我们不单单记录最后一条指令，还要记录指令列表，然后用一个引用指向“当前”的那个。 当玩家执行一条命令，我们将其添加到列表，然后将代表“当前”的指针指向它。



当玩家选择“撤销”，我们撤销现在的命令，将代表当前的指针往后退。 当他们选择“重做”，我们将代表当前的指针往前进，执行该指令。 如果在撤销后选择了新命令，那么清除命令列表中当前的指针所指命令之后的全部命令。

第一次在关卡编辑器中实现这点时，我觉得自己简直就是个天才。 我惊讶于它如此的简明有效。 你需要约束自己，保证每个数据修改都通过命令完成，一旦你做到了，余下的都很

简单。

> 重做在游戏中并不常见，但重放常见。 一种简单的重放实现是记录游戏每帧的状态，这样它可以回放，但那会消耗太多的内存。
>
> 相反，很多游戏记录每个实体每帧运行的命令。 为了重放游戏，引擎只需要正常运行游戏，执行之前存储的命令。

用类还是用函数？

早些时候，我说过命令与第一公民函数或者闭包类似， 但是在这里展现的每个例子都是通过类完成的。 如果你更熟悉函数式编程，你也许会疑惑函数都在哪里。

我用这种方式写例子是因为C++对第一公民函数支持非常有限。 函数指针没有状态，函子很奇怪而且仍然需要定义类， 在C++11中的lambda演算需要大量的人工记忆辅助才能使用。

这并不是说你在其他语言中不可以用函数来完成命令模式。 如果你使用的语言支持闭包，不管怎样，快去用它！ 在某种程度上，命令模式是为一些没有闭包的语言模拟闭包。

> (我说某种程度上是因为，即使是那些支持闭包的语言， 为命令建立真正的类或者结构也是很有用的。 如果你的命令拥有多重操作（比如可撤销的命令）， 将其全部映射到同一函数中并不优雅。)
>
> 定义一个有字段的真实类能帮助读者理解命令包含了什么数据。 闭包是自动包装状态的完美解决方案，但它们过于自动化而很难看清包装的真正状态有哪些。

举个例子，如果我们使用javascript来写游戏，那么我们可以用这种方式来写让单位移动的命令：

```
function makeMoveUnitCommand(unit, x, y) {
  // 这个函数就是命令对象:
  return function() {
    unit.moveTo(x, y);
  }
}
```

我们可以通过一对闭包来为撤销提供支持：

```
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);
    }
```

```
  };
}
```

如果你习惯了函数式编程风格，这种做事的方法是很自然的。 如果你没有，我希望这章可以帮你了解一些。 对于我而言，命令模式展现了函数式范式在很多问题上的高效性。

参见

- 你最终可能会得到很多不同的命令类。 为了更容易实现这些类，定义一个具体的基类，包含一些能定义行为的高层方法，往往会有帮助。 这将命令的主体`execute()`转到子类沙箱中。
- 在上面的例子中，我们明确地指定哪个角色会处理命令。 在某些情况下，特别是当对象模型分层时，也可以不这么简单粗暴。 对象可以响应命令，或者将命令交给它的从属对象。 如果你这样做，你就完成了一个职责链模式 GoF。

* 有些命令是无状态的纯粹行为，比如第一个例子中的`JumpCommand`。 在这种情况下，有多个实例是在浪费内存，因为所有的实例是等价的。 可以用享元模式 GoF解决。

> 你可以将其实现为单例 GoF，但真朋友不会让你用单例。

享元模式

游戏设计模式 / Design Patterns Revisited

迷雾散尽，露出了古朴庄严的森林。古老的铁杉，在头顶编成绿色穹顶。　阳光在树叶间破碎成金色顶棚。从树干间远眺，远处的森林渐渐隐去。

这是我们游戏开发者梦想的超凡场景，这样的场景通常由一个模式支撑着，它的名字低调至极：享元模式。

森林

用几句话就能描述一片巨大的森林，但是在实时游戏中做这件事就完全是另外一件事了。当屏幕上需要显示一整个森林时，图形程序员看到的是每秒需要送到GPU六十次的百万多边形。

我们讨论的是成千上万的树，每棵都由上千的多边形组成。　就算有足够的内存描述森林，渲染的过程中，CPU到GPU的部分也太过繁忙了。

每棵树都有一系列与之相关的位：

- 定义树干，树枝和树叶形状的多边形网格。
- 树皮和树叶的纹理。
- 在森林中树的位置和朝向。
- 大小和色彩之类的调节参数，让每棵树都看起来与众不同。

如果用代码表示，那么会得到这样的东西：

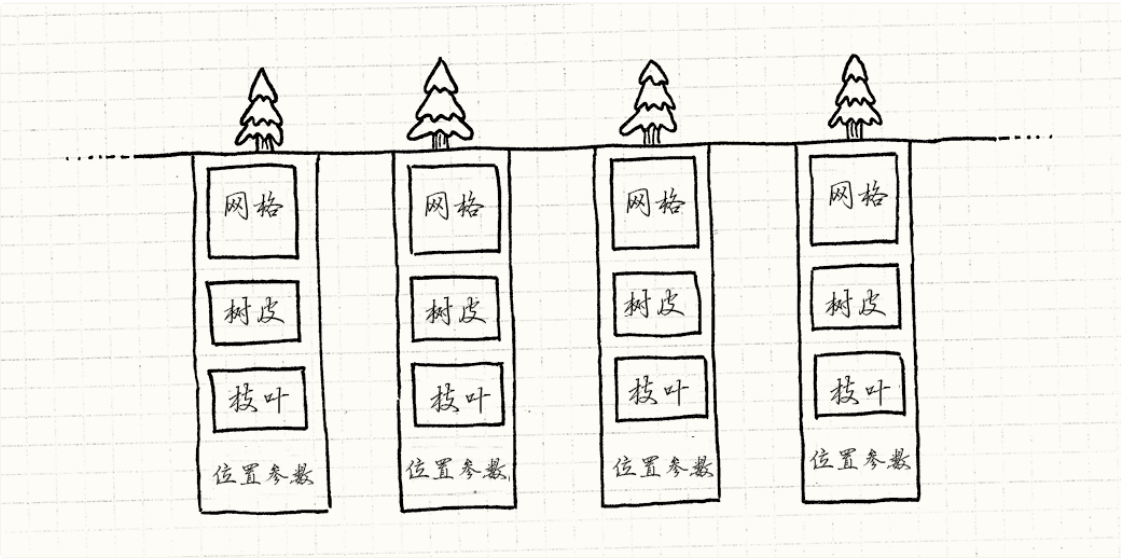
```
class Tree
{
private:
  Mesh mesh_;
  Texture bark_;
  Texture leaves_;
  Vector position_;
  double height_;
  double thickness_;
  Color barkTint_;
  Color leafTint_;
};
```

这是一大堆数据，多边形网格和纹理体积非常大。　描述整个森林的对象在一帧的时间就交

给GPU是太过了。 幸运的是，有一种老办法来处理它。

关键点在于，哪怕森林里有千千万万的树，它们大多数长得一模一样。 它们使用了相同的网格和纹理。 这意味着这些树的实例的大部分字段是一样的。

> 你要么是疯了，要么是亿万富翁，才能让美术给整个森林的每棵树建一个独立模型。



> 注意每一棵树的小盒子中的东西都是一样的。

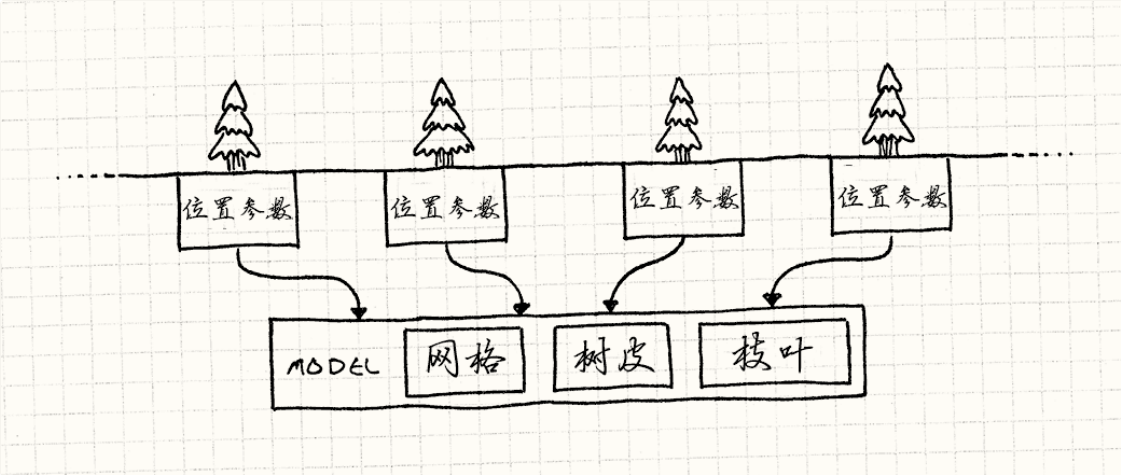
我们可以通过显式将对象切为两部分来更加明确地模拟。 第一，将树共有的数据拿出来分离到另一个类中：

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

游戏只需要一个这种类， 因为没有必要在内存中把相同的网格和纹理重复一千遍。 每个游戏世界中树的实例只需有一个对这个共享`TreeModel`的引用。 留在`Tree`中的是那些实例相关的数据：

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

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

你可以将其想象成这样：



> 这有点像类型对象模式。 两者都涉及将一个类中的状态委托给另外的类，来达到在不同实例间分享状态的目的。 但是，这两种模式背后的意图不同。
>
> 使用类型对象，目标是通过将类型引入对象模型，减少需要定义的类。 伴随而来的内容分享是额外的好处。享元模式则是纯粹的为了效率。

把所有的东西都存在主存里没什么问题，但是这对渲染也毫无帮助。 在森林到屏幕上之前，它得先到GPU。我们需要用显卡可以识别的方式共享数据。

一千个实例

为了减少需要推送到GPU的数据量，我们想把共享的数据——`TreeModel`——只发送一次。然后，我们分别发送每个树独特的数据——位置，颜色，大小。 最后，我们告诉GPU，“使用同一模型渲染每个实例”。

幸运的是，今日的图形接口和显卡正好支持这一点。 这些细节繁琐且超出了这部书的范围，但是Direct3D和OpenGL都可以做实例渲染。

在这些API中，你需要提供两部分数据流。 第一部分是一块需要渲染多次的共同数据——在例子中是树的网格和纹理。 第二部分是实例的列表以及绘制第一部分时需要使用的参数。然后调用一次渲染，绘制整个森林。

> 这个API是由显卡直接实现，意味着享元模式也许是唯一的有硬件支持的GoF设计模式。

享元模式

好了，我们已经看了一个具体的例子，下面我介绍模式的通用部分。 享元，就像它名字暗示的那样， 当你需要共享类时使用，通常是因为你有太多这种类了。

实例渲染时，每棵树通过总线送到GPU消耗的更多的是时间而非内存，但是基本要点是一样的。

这个模式通过将对象的数据分为两种来解决这个问题。 第一种数据没有特定指明是哪个对

象的实例，因此可以在它们间分享。 Gof称之为固有状态，但是我更喜欢将其视为“上下文无关”部分。 在这里的例子中，是树的网格和纹理。

数据的剩余部分是变化状态，那些每个实例独一无二的东西。 在这个例子中，是每棵树的位置，拉伸和颜色。 就像这里的示例代码块一样，这种模式通过在每个对象出现时共享一份固有状态，来节约内存。

就目前而言，这看上去像是基础的资源共享，很难被称为一种模式。 部分原因是在这个例子中，我们可以为共享状态划出一个清晰的身份：`TreeModel`。

我发现，当共享对象没有有效定义的实体时，使用这种模式就不那么明显（使用它也就越发显得精明）。 在那些情况下，这看上去是一个对象同时被魔术般的分配到了多个地方。 让我展示给你另外一个例子。

扎根之所

这些树长出来的地方也需要在游戏中表示。 这里可能有草，泥土，丘陵，湖泊，河流，以及其它任何你可以想到的地形。 我们基于区块建立地表：世界的表面被划分为由微小区块组成的巨大网格。 每个区块都由一种地形覆盖。

每种地形类型都有一系列特性会影响游戏玩法：

- 决定了玩家能够多快的穿过它的移动开销。
- 表明能否用船穿过的水域标识。
- 用来渲染它的纹理。

因为我们游戏程序员偏执于效率，我们不会在每个区块中保存这些状态。 相反，一个通用的方式是为每种地形使用一个枚举。

> 再怎么样，我们已经从树的例子吸取教训了。

```
enum Terrain
{
  TERRAIN_GRASS,
  TERRAIN_HILL,
  TERRAIN_RIVER
  // 其他地形
};
```

然后，世界管理巨大的网格：

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

> 这里我使用嵌套数组存储2D网格。 在C/C++中这样很有效率的，因为它会将所有元素打包在一起。 在Java或者其他内存管理语言中，那样做会实际给你一个数组，其中每个元素都是对数组的列的引用，那就不像你想要的那样内存友好了。

> 反正，隐藏2D网格数据结构背后的实现细节，能使代码能更好的工作。 我这里这样做只是为了让其保持简单。

为了获得区块的实际有用的数据，我们做了一些这样的事情：

```
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;
      // 其他地形……
  }
}

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;
      // 其他地形……
  }
}
```

你知道我的意思了。这可行，但是我觉得很丑。 移动开销和水域标识是区块的数据，但这里它们散布在代码中。 更糟的是，简单地形的数据被众多方法拆开了。 如果能够将这些包裹起来就好了。毕竟，那是我们设计对象的目的。

如果我们有实际的地形类就好了，像这样：

```
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_;
};
```

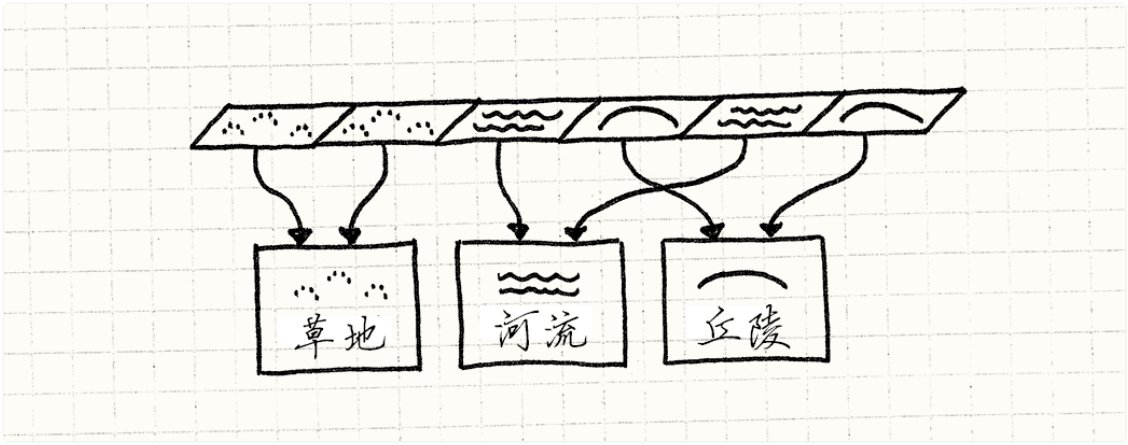
> 你会注意这里所有的方法都是const。这不是巧合。 由于同一对象在多处引用，如果你修改了它， 改变会同时在多个地方出现。
>
> 这也许不是你想要的。 通过分享对象来节约内存的这种优化，不应该影响到应用的显性行为。 因此，享元对象几乎总是不可变的。

但是我们不想为每个区块都保存一个实例。 如果你看看这个类里面，你会发现里面实际上什么也没有， 唯一特别的是区块在哪里。 用享元的术语讲，区块的所有状态都是“固有的”或者说“上下文无关的”。

鉴于此，我们没有必要保存多个同种地形类型。 地面上的草区块两两无异。 我们不用地形区块对象枚举构成世界网格，而是用Terrain对象指针组成网格：

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

  // 其他代码……
};
```

每个相同地形的区块会指向相同的地形实例。



由于地形实例在很多地方使用，如果你想要动态分配，它们的生命周期会有点复杂。 因此，我们直接在游戏世界中存储它们。

```
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_;

  // 其他代码……
};
```

然后我们可以像这样来描绘地面：

```
void World::generateTerrain()
{
  // 将地面填满草皮.
  for (int x = 0; x < WIDTH; x++)
  {
    for (int y = 0; y < HEIGHT; y++)
    {
      // 加入一些丘陵
      if (random(10) == 0)
      {
        tiles_[x][y] = &hillTerrain_;
      }
      else
      {
        tiles_[x][y] = &grassTerrain_;
      }
    }
  }

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

> 我承认这不是世界上最好的地形生成算法。

现在不需要World中的方法来接触地形属性，我们可以直接暴露出Terrain对象。

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

用这种方式，World不再与各种地形的细节耦合。 如果你想要某一区块的属性，可直接从那个对象获得：

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

我们回到了操作实体对象的API，几乎没有额外开销——指针通常不比枚举大。

性能如何？

我在这里说几乎，是因为性能偏执狂肯定会想要知道它和枚举比起来如何。 通过解引用指针获取地形需要一次间接跳转。 为了获得移动开销这样的地形数据，你首先需要跟着网格中的指针找到地形对象， 然后再找到移动开销。跟踪这样的指针会导致缓存不命中，降低运行速度。

> 对于更多指针追逐和缓存不命中，看看数据局部性这章。

就像往常一样，优化的金科玉律是需求优先。 现代计算机硬件过于复杂，性能只是游戏的一个考虑方面。 在我这章做的测试中，享元较枚举没有什么性能的优势。 享元实际上明显更快。但是这完全取决于内存中的事物是如何排列的。

我可以自信使用享元对象而不会搞到不可收拾。 它给了你面向对象的优势，而且没有产生一堆对象。 如果你创建了一个枚举，又在它上面做了很多分支跳转，考虑一下这个模式吧。 如果你担心性能，在把代码编程为难以维护的风格之前，至少先做些性能分析。

参见

- 在区块的例子中，我们只是为每种地形创建一个实例然后存储在`World`中。 这也许能更好找到和重用这些实例。 但是在多数情况下，你不会在一开始就创建所有享元。

如果你不能预料哪些是实际上需要的，最好在需要时才创建。 为了保持共享的优势，当你需要一个时，首先看看是否已经创建了一个相同的实例。 如果确实如此，那么只需返回那个实例。

这通常意味需要将构造函数封装在查询对象是否存在的接口之后。 像这样隐藏构造指令是工厂方法 GoF的一个例子。

- 为了返回一个早先创建的享元，需要追踪那些已经实例化的对象池。 正如其名，这意味着对象池是存储它们的好地方。
- 当使用状态模式时， 经常会出现一些没有任何特定字段的“状态对象”。 这个状态的标识和方法都很有用。 在这种情况下，你可以使用这个模式，然后在不同的状态机上使用相同的对象实例。

观察者模式

游戏设计模式 / Design Patterns Revisited

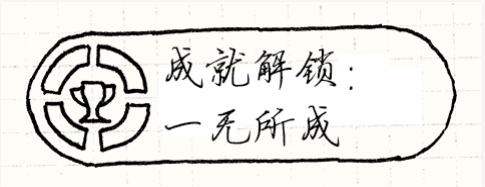
随便打开电脑中的一个应用，很有可能它就使用了MVC架构， 而究其根本，是因为观察者模式。 观察者模式应用广泛，Java甚至将其放到了核心库之中（`java.util.Observer`），而C#直接将其嵌入了语法（`event`关键字）。

> 就像软件中的很多东西，MVC是Smalltalkers在七十年代创造的。 Lisp程序员也许会说其实是他们在六十年代发明的，但是他们懒得记下来。

观察者模式是应用最广泛和最广为人知的GoF模式，但是游戏开发世界与世隔绝， 所以对你来说，它也许是全新的。 假设你与世隔绝，让我给你举个形象的例子。

成就解锁

假设我们向游戏中添加了成就系统。 它存储了玩家可以完成的各种各样的成就，比如“杀死1000只猴子恶魔”，“从桥上掉下去”，或者“一命通关”。



我发誓画的这个没有第二个意思，笑。

要实现这样一个包含各种行为来解锁成就的系统是很有技巧的。 如果我们不够小心，成就系统会缠绕在代码库的每个黑暗角落。 当然，“从桥上掉落”和物理引擎相关， 但我们并不想看到的在处理撞击代码的线性代数时， 有个对`unlockFallOffBridge()`的调用是不？

> 这只是随口一说。 有自尊的物理程序员绝不会允许像游戏玩法这样的平凡之物玷污他们优美的算式。

我们喜欢的是，照旧，让关注游戏一部分的所有代码集成到一块。 挑战在于，成就在游戏的不同层面被触发。怎么解耦成就系统和其他部分呢？

这就是观察者模式出现的原因。 这让代码宣称有趣的事情发生了，而不必关心到底是谁接受了通知。

举个例子，有物理代码处理重力，追踪哪些物体待在地表，哪些坠入深渊。 为了实现“桥上掉落”的徽章，我们可以直接把成就代码放在那里，但那就会一团糟。 相反，可以这样做：

```
void Physics::updateEntity(Entity& entity)
{
  bool wasOnSurface = entity.isOnSurface();
  entity.accelerate(GRAVITY);
  entity.update();
  if (wasOnSurface && !entity.isOnSurface())
  {
    notify(entity, EVENT_START_FALL);
  }
}
```

它做的就是声称，“额，我不知道有谁感兴趣，但是这个东西刚刚掉下去了。做你想做的事吧。”

> 物理引擎确实决定了要发送什么通知，所以这并没有完全解耦。但在架构这个领域，通常只能让系统变得更好，而不是完美。

成就系统注册它自己为观察者，这样无论何时物理代码发送通知，成就系统都能收到。 它可以检查掉落的物体是不是我们的失足英雄， 他之前有没有做过这种不愉快的与桥的经典力学遭遇。 如果满足条件，就伴着礼花和炫光解锁合适的成就，而这些都无需牵扯到物理代码。

事实上，我们可以改变成就的集合或者删除整个成就系统，而不必修改物理引擎。 它仍然会发送它的通知，哪怕实际没有东西接收。

> 当然，如果我们永久移除成就，没有任何东西需要物理引擎的通知， 我们也同样可以移除通知代码。但是在游戏的演进中，最好保持这里的灵活性。

它如何运作

如果你还不知道如何实现这个模式，你可能可以从之前的描述中猜到，但是为了减轻你的负担，我还是过一遍代码吧。

观察者

我们从那个需要知道别的对象做了什么事的类开始。 这些好打听的对象用如下接口定义：

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

> `onNotify()`的参数取决于你。这就是为什么是观察者模式， 而不是“可以粘贴到游戏中的真实代码”。 典型的参数是发送通知的对象和一个装入其他细节的“数据”参数。
>
> 如果你用泛型或者模板编程，你可能会在这里使用它们，但是根据你的特殊用况

> 裁剪它们也很好。 这里，我将其硬编码为接受一个游戏实体和一个描述发生了什么的枚举。

任何实现了这个的具体类就成为了观察者。 在我们的例子中，是成就系统，所以我们可以像这样实现：

```
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;

      // 处理其他事件，更新heroIsOnBridge_变量……
    }
  }

private:
  void unlock(Achievement achievement)
  {
    // 如果还没有解锁，那就解锁成就……
  }

  bool heroIsOnBridge_;
};
```

被观察者

被观察的对象拥有通知的方法函数，用GoF的说法，那些对象被称为“主题”。 它有两个任务。首先，它有一个列表，保存默默等它通知的观察者：

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

> 在真实代码中，你会使用动态大小的集合而不是一个定长数组。 在这里，我使用这种最基础的形式是为了那些不了解C++标准库的人们。

重点是被观察者暴露了公开的API来修改这个列表：

```
class Subject
{
public:
  void addObserver(Observer* observer)
  {
    // 添加到数组中……
  }

  void removeObserver(Observer* observer)
  {
    // 从数组中移除……
  }

  // 其他代码……
};
```

这就允许了外界代码控制谁接收通知。 被观察者与观察者交流，但是不与它们耦合。 在我们的例子中，没有一行物理代码会提及成就。 但它仍然可以与成就系统交流。这就是这个模式的聪慧之处。

被观察者有一列表观察者而不是单个观察者也是很重要的。 这保证了观察者不会相互干扰。 举个例子，假设音频引擎也需要观察坠落事件来播放合适的音乐。 如果客体只支持单个观察者，当音频引擎注册时，就会取消成就系统的注册。

这意味着这两个系统需要相互交互——而且是用一种极其糟糕的方式，第二个注册时会使第一个的注册失效。 支持一列表的观察者保证了每个观察者都是被独立处理的。 就它们各自的视角来看，自己是这世界上唯一看着被观察者的。

被观察者的剩余任务就是发送通知：

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

  // 其他代码……
};
```

注意，代码假设了观察者不会在它们的`onNotify()`方法中修改观察者列表。 更加可靠的实现方法会阻止或优雅地处理这样的并发修改。

可被观察的物理系统

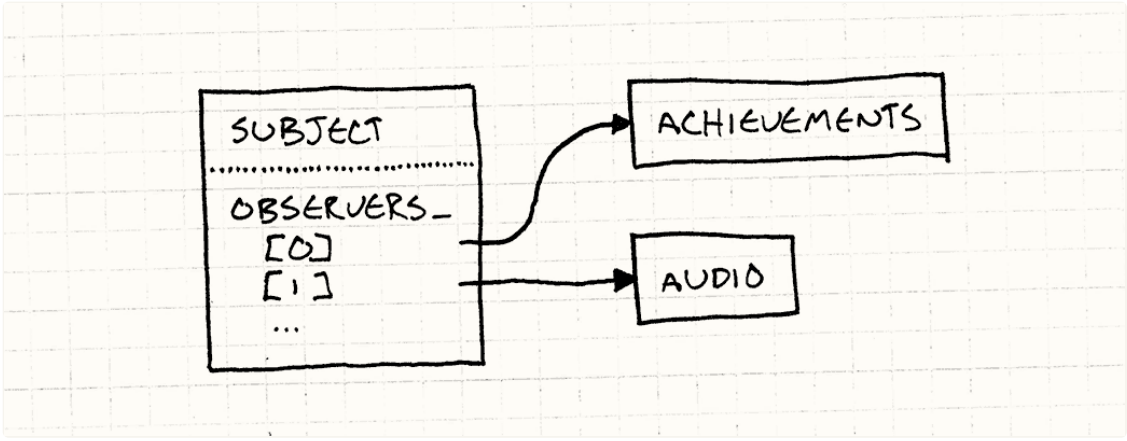
现在，我们只需要给物理引擎和这些挂钩，这样它可以发送消息， 成就系统可以和引擎连线来接受消息。 我们按照传统的设计模式方法实现，继承`Subject`：

```
class Physics : public Subject
{
public:
  void updateEntity(Entity& entity);
};
```

这让我们将`notify()`实现为了`Subject`内的保护方法。 这样派生的物理引擎类可以调用并发送通知，但是外部的代码不行。 同时，`addObserver()`和`removeObserver()`是公开的， 所以任何可以接触物理引擎的东西都可以观察它。

> 在真实代码中，我会避免使用这里的继承。 相反，我会让`Physics` 有 一个`Subject`的实例。 不再是观察物理引擎本身，被观察的会是独立的“下落事件”对象。 观察者可以用像这样注册它们自己：
>
> ```
> physics.entityFell()
>   .addObserver(this);
> ```
>
> 对我而言，这是“观察者”系统与“事件”系统的不同之处。 使用前者，你观察做了有趣事情的事物。 使用后者，你观察的对象代表了发生的有趣事情。

现在，当物理引擎做了些值得关注的东西，它调用`notify()`，就像之前的例子。 它遍历了观察者列表，通知所有观察者。



很简单，对吧？只要一个类管理一列表指向接口实例的指针。 难以置信的是，如此直观的东西是无数程序和应用框架交流的主心骨。

观察者模式不是完美无缺的。当我问其他程序员怎么看，他们提出了一些抱怨。 让我们看看可以做些什么来处理这些抱怨。

太慢了

我经常听到这点，通常是从那些不知道模式具体细节的程序员那里。 他们有一种假设，任何东西只要沾到了“设计模式”，那么一定包含了一堆类，跳转和浪费CPU循环其他行为。

观察者模式的名声特别坏，一些坏名声的事物与它如影随形， 比如“事件”，“消息”，甚至“数据绑定”。 其中的一些系统确实会慢。（通常是故意的，出于好的意图）。 他们使用队列，或者为每个通知动态分配内存。

> 这就是为什么我认为设计模式文档化很重要。 当我们没有统一的术语，我们就失去了简洁明确表达的能力。 你说“观察者”，我以为是“事件”，他以为是“消息”，因为没人花时间记下差异，也没人阅读。
>
> 而那就是在这本书中我要做的。 本书中也有一章关于事件和消息：事件队列.

现在你看到了模式是如何真正被实现的， 你知道事实并不如他们所想的这样。 发送通知只需简单地遍历列表，调用一些虚方法。 是的，这比静态调用慢一点，除非是性能攸关的代码，否则这点消耗都是微不足道的。

我发现这个模式在代码性能瓶颈以外的地方能有很好的应用， 那些你可以承担动态分配消耗的地方。 除那以外，使用它几乎毫无限制。 我们不必为消息分配对象，也无需使用队列。这里只多了一个用在同步方法调用上的额外跳转。

太快？

事实上，你得小心，观察者模式是同步的。 被观察者直接调用了观察者，这意味着直到所有观察者的通知方法返回后， 被观察者才会继续自己的工作。观察者会阻塞被观察者的运行。

这听起来很疯狂，但在实践中，这可不是世界末日。 这只是值得注意的事情。 UI程序员——那些使用基于事件的编程的程序员已经这么干了很多年了——有句经典名言：“远离UI线程”。

如果要对事件同步响应，你需要完成响应，尽可能快的返回，这样UI就不会锁死。 当你有耗时的操作要执行时，将这些操作推到另一个线程或工作队列中去。

你需要小心地在观察者中混合线程和锁。 如果观察者试图获得被观察者拥有的锁，游戏就进入死锁了。 在多线程引擎中，你最好使用事件队列来做异步通信。

“它做了太多动态分配”

整个程序员社区——包括很多游戏开发者——转向了拥有垃圾回收机制的语言， 动态分配今昔非比。 但在像游戏这样性能攸关的软件中，哪怕是在有垃圾回收机制的语言，内存分配也依然重要。 动态分配需要时间，回收内存也需要时间，哪怕是自动运行的。

> 很多游戏开发者不怎么担心分配,但很担心分页。 当游戏需要不崩溃的连续运行多日来获得发售资格，不断增加的分页堆会影响游戏的发售。
>
> 对象池模式一章介绍了避免这点的常用技术，以及更多其他细节。

在上面的示例代码中，我使用的是定长数组，因为我想尽可能保证简单。 在真实的项目中中，观察者列表随着观察者的添加和删除而动态地增长和缩短。 这种内存的分配吓坏了一些人。

当然，第一件需要注意的事情是只在观察者加入时分配内存。 发送 通知无需内存分配——只需一个方法调用。 如果你在游戏一开始就加入观察者而不乱动它们，分配的总量是很小的。

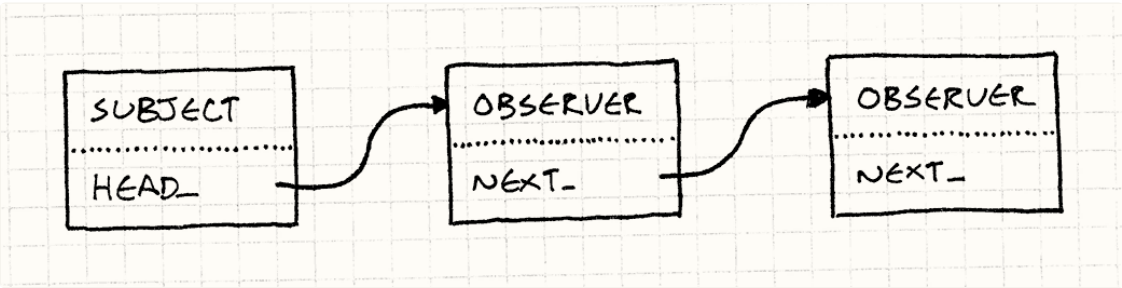
如果这仍然困扰你，我会介绍一种无需任何动态分配的方式来增加和删除观察者。

链式观察者

我们现在看到的所有代码中，`Subject`拥有一列指针指向观察它的`Observer`。 `Observer`

类本身没有对这个列表的引用。 它是纯粹的虚接口。优先使用接口，而不是有状态的具体类，这大体上是一件好事。

但是如果我们确实愿意在Observer中放一些状态， 我们可以将观察者的列表分布到观察者自己中来解决动态分配问题。 不是被观察者保留一列表分散的指针，观察者对象本身成为了链表中的一部分：



为了实现这一点，我们首先要摆脱Subject中的数组，然后用链表头部的指针取而代之：

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

  // 方法……
private:
  Observer* head_;
};
```

然后，我们在Observer中添加指向链表中下一观察者的指针。

```
class Observer
{
  friend class Subject;

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

  // 其他代码……
private:
  Observer* next_;
};
```

这里我们也让Subject成为了友类。 被观察者拥有增删观察者的API，但是现在链表在Observer内部管理。 最简单的实现办法就是让被观察者类成为友类。

注册一个新观察者就是将其连到链表中。我们用更简单的实现方法，将其插到开头：

```
void Subject::addObserver(Observer* observer)
{
```

```
  observer->next_ = head_;
  head_ = observer;
}
```

另一个选项是将其添加到链表的末尾。这么做增加了一定的复杂性。 `Subject`要么遍历整个链表来找到尾部，要么保留一个单独`tail_`指针指向最后一个节点。

加在在列表的头很简单，但也有另一副作用。 当我们遍历列表给每个观察者发送一个通知， 最新注册的观察者最先接到通知。 所以如果以A，B，C的顺序来注册观察者，它们会以C，B，A的顺序接到通知。

理论上，这种还是那种方式没什么差别。 在好的观察者设计中，观察同一被观察者的两个观察者互相之间不该有任何顺序相关。 如果顺序确实有影响，这意味着这两个观察者有一些微妙的耦合，最终会害了你。

让我们完成删除操作：

```
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_;
  }
}
```

> 如你所见，从链表移除一个节点通常需要处理一些丑陋的特殊情况，应对头节点。 还可以使用指针的指针，实现一个更优雅的方案。
>
> 我在这里没有那么做，是因为半数看到这个方案的人都迷糊了。 但这是一个很值得做的练习：它能帮助你深入思考指针。

因为使用的是链表，所以我们得遍历它才能找到要删除的观察者。 如果我们使用普通的数组，也得做相同的事。 如果我们使用双向链表，每个观察者都有指向前面和后面的指针，就可以用常量时间移除观察者。在实际项目中，我会这样做。

剩下的事情只有发送通知了，这和遍历列表同样简单；

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

这里，我们遍历了整个链表，通知了其中每一个观察者。 这保证了所有的观察者相互独立并有同样的优先级。

我们可以这样实现，当观察者接到通知，它返回了一个标识，表明被观察者是否应该继续遍历列表。 如果这样做，你就接近了职责链 GoF 模式。

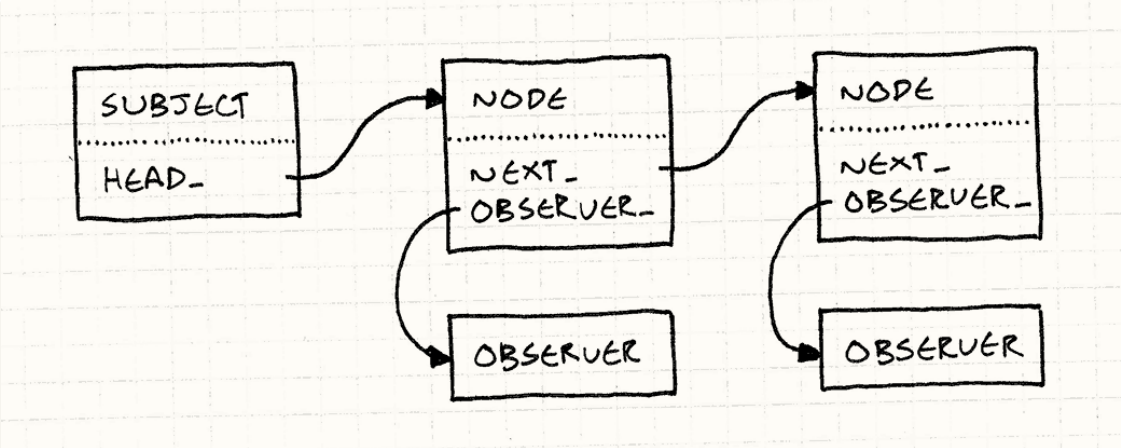
不差嘛，对吧？被观察者现在想有多少观察者就有多少观察者，无需动态内存。 注册和取消注册就像使用简单数组一样快。 但是，我们牺牲了一些小小的功能特性。

由于我们使用观察者对象作为链表节点，这暗示它只能存在于一个观察者链表中。 换言之，一个观察者一次只能观察一个被观察者。 在传统的实现中，每个被观察者有独立的列表，一个观察者同时可以存在于多个列表中。

你也许可以接受这一限制。 通常是一个被观察者有多个观察者，反过来就很少见了。 如果这真是一个问题，这里还有一种不必使用动态分配的解决方案。 详细介绍的话，这章就太长了，但我会大致描述一下，其余的你可以自行填补……

链表节点池

就像之前，每个被观察者有一链表的观察者。 但是，这些链表节点不是观察者本身。 相反，它们是分散的小“链表节点”对象， 包含了指向观察者的指针和指向链表下一节点的指针。



由于多个节点可以指向同一观察者，这就意味着观察者可以同时在超过多个被观察者的列表中。 我们可以同时观察多个对象了。

链表有两种风格。学校教授的那种，节点对象包含数据。 在我们之前的观察者链表的例子中，是另一种： 数据（这个例子中是观察者）包含了节点（next_指针）。

后者的风格被称为“侵入式”链表，因为在对象内部使用链表侵入了对象本身的定义

。 侵入式链表灵活性更小，但如我们所见，也更有效率。 在Linux核心这样的地方这种风格很流行。

避免动态分配的方法很简单：由于这些节点都是同样大小和类型，可以预先在对象池中分配它们。 这样你只需处理固定大小的列表节点，可以随你所需使用和重用，而无需牵扯到真正的内存分配器。

剩余的问题

我认为三个该模式将人们吓阻的主要问题已经被搞定了。 它简单，快速，对内存管理友好。 但是这意味着你总该使用观察者吗？

现在，这是另一个的问题。 就像所有的设计模式，观察者模式不是万能药。 哪怕可以正确高效的实现，它也不一定是好的解决方案。 设计模式有坏名声的原因之一就是人们将好模式用在错问题上，得到了糟糕的结果。

还有两个挑战，一个是关于技术，另一个更偏向是可维护性。 我们先处理关于技术的挑战，因为关于技术的问题总是更容易处理。

销毁被观察者和观察者

我们看到的样例代码健壮可用，但有一个严重的副作用： 当删除一个被观察者或观察者时会发生什么？ 如果你不小心的在某些观察者上面调用了`delete`，被观察者也许仍然持有指向它的指针。 那是一个指向一片已释放区域的悬挂指针。 当被观察者试图发送一个通知，额……就说发生的事情会出乎你的意料之外吧。

> 不是谴责，但我注意到设计模式完全没提这个问题。

删除被观察者更容易些，因为在大多数实现中，观察者没有对它的引用。 但是即使这样，将被观察者所占的字节直接回收也许会造成一些问题。 这些观察者也许仍然期待在以后收到通知，而这是不可能的了。 它们没法继续观察了，真的，它们只是认为它们可以。

你可以用好几种方式处理这点。 最简单的就是像我做的那样，以后一脚踩空。 在被删除时取消注册是观察者的职责。 多数情况下，观察者确实知道它在观察哪个被观察者，所以通常需要做的只是给它的析构器添加一个`removeObserver()`。

> 通常在这种情况下，难点不在如何做，而在记得做。

如果在删除被观察者时，你不想让观察者处理问题，这也很好解决。 只需要让被观察者在它被删除前发送一个最终的“死亡通知”。 这样，任何观察者都可以接收到，然后做些合适的行为。

> 默哀，献花，挽歌……

人——哪怕是那些花费在大量时间在机器前，拥有让我们黯然失色才能的人——也是绝对地不可靠。 这就是为什么我们发明了电脑：它们不像我们那样经常犯错误。

更安全的方案是在每个被观察者销毁时，让观察者自动取消注册。 如果你在观察者基类中实现了这个逻辑，每个人不必记住就可以使用它。 这确实增加了一定的复杂度。 这意味着每个观察者都需要有它在观察的被观察者的列表。 最终维护一个双向指针。

别担心，我有垃圾回收器

你们那些装备有垃圾回收系统的孩子现在一定很洋洋自得。 觉得你不必担心这个，因为你从来不必显式删除任何东西？再仔细想想！

想象一下：你有UI显示玩家角色情况的状态，比如健康和道具。 当玩家在屏幕上时，你为其初始化了一个对象。 当UI退出时，你直接忘掉这个对象，交给GC清理。

每当角色脸上（或者其他什么地方）挨了一拳，就发送一个通知。 UI观察到了，然后更新健康槽。很好。 当玩家离开场景，但你没有取消观察者的注册，会发生什么？

UI界面不再可见，但也不会进入垃圾回收系统，因为角色的观察者列表还保存着对它的引用。 每一次场景加载后，我们给那个不断增长的观察者列表添加一个新实例。

玩家玩游戏时，来回跑动，打架，角色的通知发送给所有的界面。 它们不在屏幕上，但它们接受通知，这样就浪费CPU循环在不可见的UI元素上了。 如果它们会播放声音之类的，这样的错误就会被人察觉。

这在通知系统中非常常见，甚至专门有个名字：失效监听者问题。 由于被观察者保留了对观察者的引用，最终有UI界面对象僵死在内存中。 这里的教训是要及时删除观察者。

> 它甚至有专门的维基条目。

然后呢？

观察者的另一个深层次问题是它意图直接导致的。 我们使用它是因为它帮助我们放松了两块代码之间的耦合。 它让观察者与没有静态绑定的观察者间接交流。

当你要理解被观察者的行为时，这很有价值，任何不相关的事情都是在分散注意力。 如果你在处理物理引擎，你根本不想要编辑器——或者你的大脑——被一堆成就系统的东西而搞糊涂。

另一方面，如果你的程序没能运行，漏洞散布在多个观察者之间，理清信息流变得更加困难。 显式耦合中更易于查看哪一个方法被调用了。 这是因为耦合是静态的，IDE分析它轻而易举。

但是如果耦合发生在观察者列表中，想要知道哪个观察者被通知到了，唯一的办法是看看哪个观察者在列表中，而且处于运行中。 不再是理清程序的静态交流结构，你得理清它的命令式，动态行为。

如何处理这个的指导原则很简单。 如果为了理解程序的一部分，两个交流的模块都需要考虑， 那就不要使用观察者模式，使用其他更加显式的东西。

当你在某些大型程序上用黑魔法时，你会感觉这样处理很笨拙。 我们有很多术语用来描述，比如“关注点分离”，“一致性和内聚性”和“模块化”， 总归就是“这些东西待在一起，而不是与那些东西待在一起。”

观察者模式是一个让这些不相关的代码块互相交流，而不必打包成更大块的好方法。 这在专注于一个特性或层面的单一代码块内不会太有用。

这就是为什么它能很好地适应我们的例子： 成就和物理是几乎完全不相干的领域，通常被不同人实现。 我们想要它们之间的交流最小化， 这样无论在哪一个上工作都不需要另一个的太多信息。

今日观察者

设计模式源于1994。 那时候，面向对象语言正是 热门的编程范式。 每个程序员都想要“30天学会面向对象编程”， 中层管理员根据程序员创建类的数量为他们支付工资。 工程师通过继承层次的深度评价代码质量。

> 同一年，Ace of Base的畅销单曲发行了三首而不是一首，这也许能让你了解一些我们那时的品味和洞察力。

观察者模式在那个时代中很流行，所以构建它需要很多类就不奇怪了。 但是现代的主流程序员更加适应函数式语言。 实现一整套接口只是为了接受一个通知不再符合今日的美学了。

它看上去是又沉重又死板。它确实又沉重又死板。 举个例子，在观察者类中，你不能为不同的被观察者调用不同通知方法。

> 这就是为什么被观察者经常将自身传给观察者。 观察者只有单一的`onNotify()`方法， 如果它观察多个被观察者，它需要知道哪个被观察者在调用它的方法。

现代的解决办法是让“观察者”只是对方法或者函数的引用。 在函数作为第一公民的语言中，特别是那些有闭包的， 这种实现观察者的方式更为普遍。

> 今日，几乎每种语言都有闭包。C++克服了在没有垃圾回收的语言中构建闭包的挑战， 甚至是Java都在JDK8中引入了闭包。

举个例子，C#有“事件”嵌在语言中。 通过这样，观察者是一个“委托”， （“委托”是方法的引用在C#中的术语）。 在JavaScript事件系统中，观察者可以是支持了特定`EventListener`协议的类， 但是它们也可以是函数。 后者是人们常用的方式。

如果设计今日的观察者模式，我会让它基于函数而不是基于类。 哪怕是在C++中，我倾向于让你注册一个成员函数指针作为观察者，而不是`Observer`接口的实例。

> 这里的一篇有趣博文以某种方式在C++上实现了这一点。

明日观察者

事件系统和其他类似观察者的模式如今遍地都是。 它们都是成熟的方案。 但是如果你用它们写一个稍微大一些的应用，你会发现一件事情。 在观察者中很多代码最后都长得一样。通常是这样：

1. 获知有状态改变了。
2. 下命令改变一些UI来反映新的状态。

就是这样，“哦，英雄的健康现在是7了？让我们把血条的宽度设为70像素。 过上一段时间，这会变得很沉闷。 计算机科学学术界和软件工程师已经用了很长时间尝试结束这种状况了。 这些方式被赋予了不同的名字：“数据流编程”，“函数反射编程”等等。

即使有所突破，一般也局限在特定的领域中，比如音频处理或芯片设计，我们还没有找到万能钥匙。 与此同时，一个更脚踏实地的方式开始获得成效。那就是现在的很多应用框架使用的“数据绑定”。

不像激进的方式，数据绑定不再指望完全终结命令式代码， 也不尝试基于巨大的声明式数据图表架构整个应用。 它做的只是自动改变UI元素或计算某些数值来反映一些值的变化。

就像其他声明式系统，数据绑定也许太慢，嵌入游戏引擎的核心也太复杂。 但是如果说它不会侵入游戏不那么性能攸关的部分，比如UI，那我会很惊讶。

与此同时，经典观察者模式仍然在那里等着我们。 是的，它不像其他的新热门技术一样在名字中填满了“函数”“反射”， 但是它超简单而且能正常工作。对我而言，这通常是解决方案最重要的条件。

← 上一章 ≡ 首页 下一章 →

原型模式

游戏设计模式 / Design Patterns Revisited

我第一次听到“原型”这个词是在设计模式中。 如今，似乎每个人都在用这个词，但他们讨论的实际不是设计模式 GoF。 我们会讨论他们所说的原型，也会讨论术语“原型”的有趣之处，和其背后的理念。 但首先，让我们重访传统的设计模式。

> “传统的”一词可不是随便用的。 设计模式引自1963年 Ivan Sutherland的Sketchpad传奇项目，那是这个模式首次出现。 当其他人在听迪伦和甲壳虫乐队时，Sutherland正忙于，你知道的，发明CAD，交互图形和面向对象编程的基本概念。
>
> 看看这个demo，跪服吧。

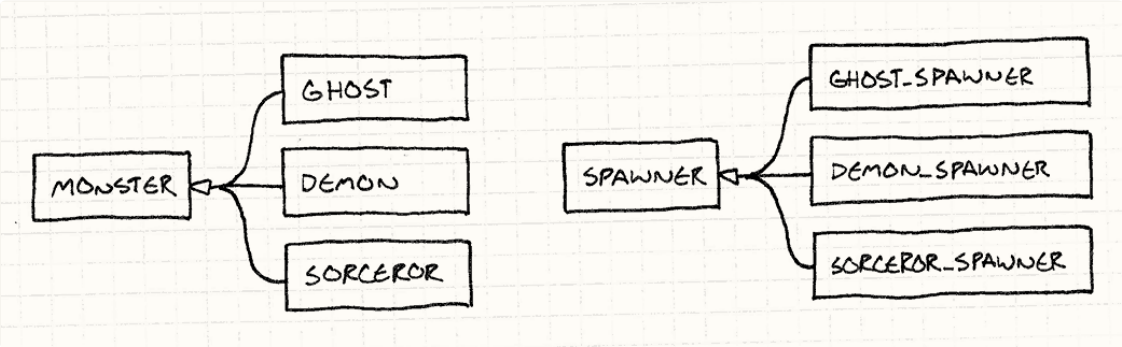
原型设计模式

假设我们要用《圣铠传说》的风格做款游戏。 野兽和恶魔围绕着英雄，争着要吃他的血肉。 这些可怖的同行者通过“生产者”进入这片区域，每种敌人有不同的生产者。

在这个例子中，假设我们游戏中每种怪物都有不同的类——`Ghost`，`Demon`，`Sorcerer`等等，像这样：

```
class Monster
{
  // 代码……
};

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

生产者构造特定种类怪物的实例。 为了在游戏中支持每种怪物，我们可以用一种暴力的实现方法， 让每个怪物类都有生产者类，得到平行的类结构：



我得翻出落满灰尘的UML书来画这个图表。 <- 代表“继承”。

实现后看起来像是这样：

```
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();
  }
};

// 你知道思路了……
```

除非你会根据代码数量来获得工资， 否则将这些焊在一起很明显不是好方法。 众多类，众多引用，众多冗余，众多副本，众多重复自我……

原型模式提供了一个解决方案。 关键思路是一个对象可以产出与它自己相近的对象。 如果你有一个恶灵，你可以制造更多恶灵。 如果你有一个恶魔，你可以制造其他恶魔。 任何怪物都可以被视为原型怪物，产出其他版本的自己。

为了实现这个功能，我们给基类`Monster`一个抽象方法`clone()`：

```
class Monster
{
```

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

  // 其他代码……
};
```

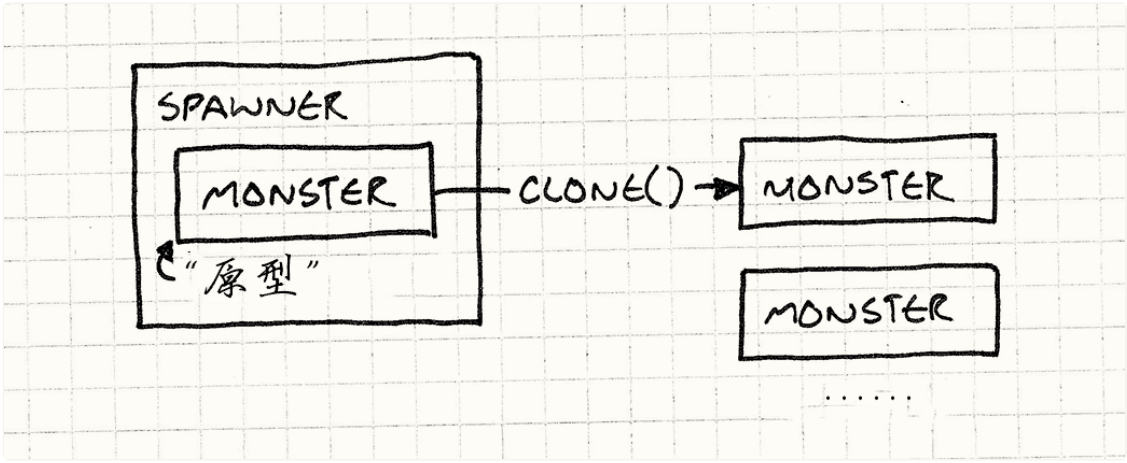
每个怪兽子类提供一个特定实现，返回与它自己的类和状态都完全一样的新对象。举个例子：

```
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_;
};
```

一旦我们所有的怪物都支持这个， 我们不再需要为每个怪物类创建生产者类。我们只需定义一个类：

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

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

private:
  Monster* prototype_;
};
```

它内部存有一个怪物，一个隐藏的怪物， 它唯一的任务就是被生产者当做模板，去产生更多一样的怪物， 有点像一个从来不离开巢穴的蜂后。



为了得到恶灵生产者，我们创建一个恶灵的原型实例，然后创建拥有这个实例的生产者：

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

这个模式的灵巧之处在于它不但拷贝原型的类，也拷贝它的状态。 这就意味着我们可以创建一个生产者，生产快速鬼魂，虚弱鬼魂，慢速鬼魂，而只需创建一个合适的原型鬼魂。

我在这个模式中找到了一些既优雅又令人惊叹的东西。 我无法想象自己是如何创造出它们的，但我更无法想象不知道这些东西的自己该如何是好。

效果如何？

好吧，我们不需要为每个怪物创建单独的生产者类，那很好。 但我们确实需要在每个怪物类中实现`clone()`。 这和使用生产者方法比起来也没节约多少代码量。

当你坐下来试着写一个正确的`clone()`，会遇见令人不快的语义漏洞。 做深层拷贝还是浅层拷贝呢？换言之，如果恶魔拿着叉子，克隆恶魔也要克隆叉子吗？

同时，这看上去没减少已存问题上的代码， 事实上还增添了些人为的问题。 我们需要将每个怪物有独立的类作为前提条件。 这绝对不是当今大多数游戏引擎运作的方法。

我们中大部分痛苦地学到，这样庞杂的类层次管理起来很痛苦， 那就是我们为什么用组件模式 和类型对象 为不同的实体建模，这样无需一一建构自己的类。

生产函数

哪怕我们确实需要为每个怪物构建不同的类，这里还有其他的实现方法。 不是使用为每个怪物建立分离的生产者类，我们可以创建生产函数，就像这样：

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

这比构建怪兽生产者类更简洁。生产者类只需简单地存储一个函数指针：

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

class Spawner
```

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

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

private:
  SpawnCallback spawn_;
};
```

为了给恶灵构建生产者，你需要做：

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

模板

如今，大多数C++开发者已然熟悉模板了。 生产者类需要为某类怪物构建实例，但是我们不想硬编码是哪类怪物。 自然的解决方案是将它作为模板中的类型参数：

> 我不太确定程序员是学着喜欢C++模板还是完全畏惧并远离了C++。 不管怎样，今日我见到的程序员中，使用C++的也都会使用模板。
>
> 这里的Spawner类不必考虑将生产什么样的怪物， 它总与指向Monster的指针打交道。
>
> 如果我们只有SpawnerFor<T>类，模板类型没有办法共享父模板， 这样的话，如果一段代码需要与产生多种怪物类型的生产者打交道，就都得接受模板参数。

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

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

像这样使用它：

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

第一公民类型

前面的两个解决方案使用类完成了需求，Spawner使用类型进行参数化。 在C++中，类型不是第一公民，所以需要一些改动。 如果你使用JavaScript，Python，或者Ruby这样的动态类型语言， 它们的类是可以传递的对象，你可以用更直接的办法解决这个问题。

> 某种程度上， 类型对象 也是为了弥补第一公民类型的缺失。 但那个模式在拥有第一公民类型的语言中也有用，因为它让你决定什么是“类型”。 你也许想要与语言内建的类不同的语义。

当你完成一个生产者，直接向它传递要构建的怪物类——那个代表了怪物类的运行时对象。超容易的，对吧。

综上所述，老实说，我不能说找到了一种情景，而在这个情景下，原型设计模式是最好的方案。 也许你的体验有所不同，但现在把它搁到一边，我们讨论点别的：将原型作为一种语言范式。

原型语言范式

很多人认为“面向对象编程”和“类”是同义词。 OOP的定义却让人感觉正好相反， 毫无疑问，**OOP**让你定义**“**对象**”**，将数据和代码绑定在一起。 与C这样的结构化语言相比，与Scheme这样的函数语言相比， OOP的特性是它将状态和行为紧紧地绑在一起。

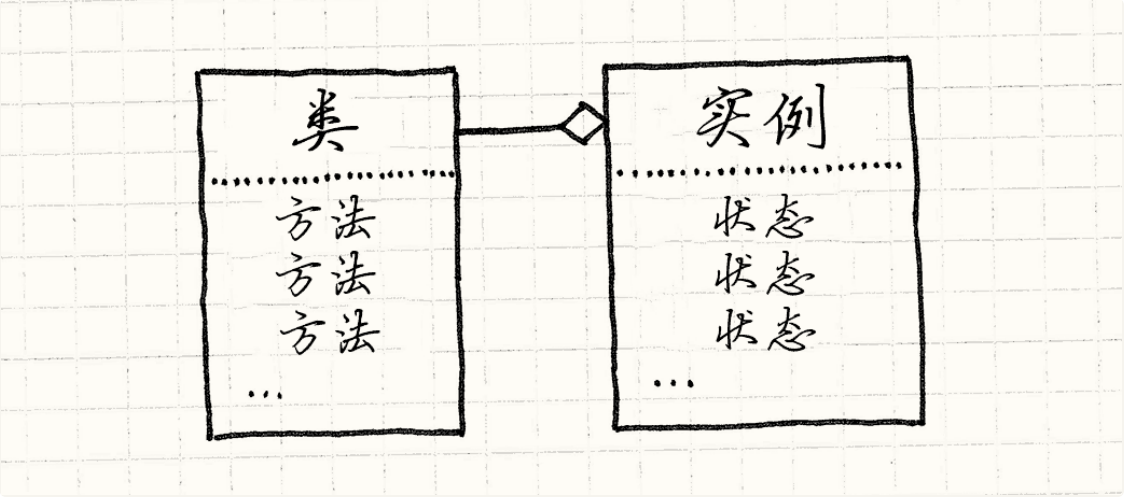
你也许认为类是完成这个的唯一方式方法， 但是包括Dave Ungar和Randall Smith的一大堆家伙一直在拼命区分OOP和类。 他们在80年代创建了一种叫做Self的语言。它不用类实现了OOP。

Self语言

就单纯意义而言，Self比基于类的语言更加面向对象。 我们认为OOP将状态和行为绑在一起，但是基于类的语言实际将状态和行为割裂开来。

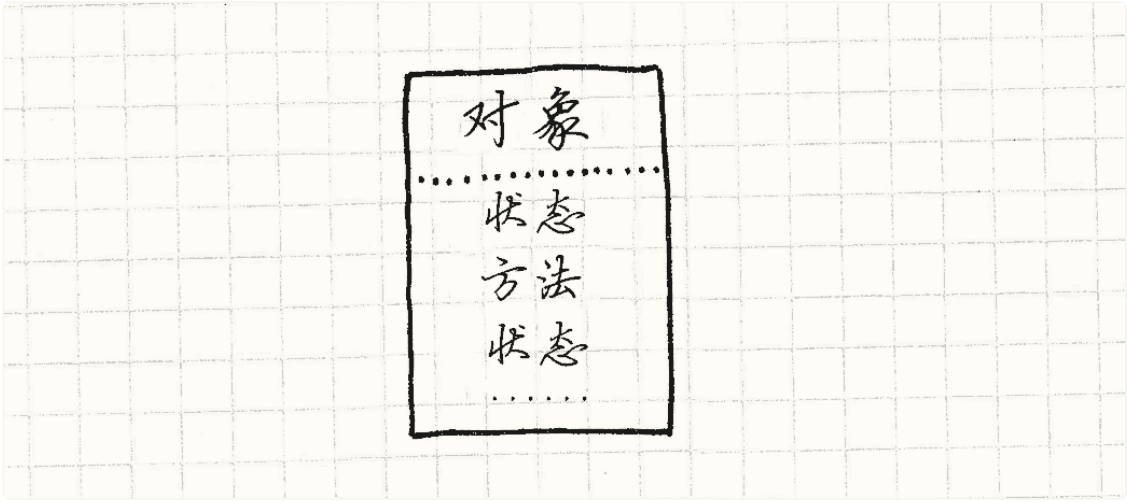
拿你最喜欢的基于类的语言的语法来说。 为了接触对象中的一些状态，你需要在实例的内存中查询。状态包含在实例中。

但是，为了调用方法，你需要找到实例的类， 然后在那里调用方法。行为包含在类中。 获得方法总需要通过中间层，这意味着字段和方法是不同的。



> 举个例子，为了调用C++中的虚方法，你需要在实例中找指向虚方法表的指针，然后再在那里找方法。

Self结束了这种分歧。无论你要找啥，都只需在对象中找。 实例同时包含状态和行为。你可以构建拥有完全独特方法的对象。

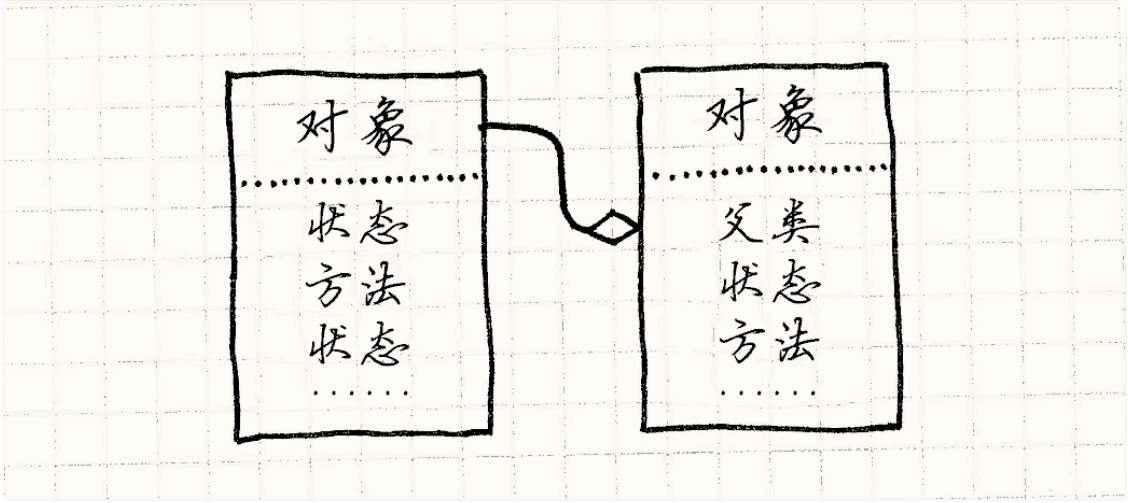


> 没有人能与世隔绝，但这个对象是。

如果这就是Self语言的全部，那它将很难使用。 基于类的语言中的继承，不管有多少缺陷，给了有用的机制来重用代码，避免重复。 为了不使用类而实现一些类似的功能，Self语言加入了委托。

如果要在对象中寻找字段或者调用方法，首先在对象内部查找。 如果能找到，那就成了。如果找不到，在对象的父对象中寻找。 这里的父类仅仅是一个对其他对象的引用。 当我们没能在第一个对象中找到属性，我们尝试它的父对象，然后父类的父对象，继续下去直到找到或者没有父对象为止。 换言之，失败的查找被委托给对象的父对象。

> 我在这里简化了。Self实际上支持多个父对象。 父对象只是特别标明的字段，意味着你可以继承它们或者在运行时改变他们， 你最终得到了“动态继承”。



父对象让我们在不同对象间重用行为（还有状态！），这样就完成了类的公用功能。 类做的另一个关键事情就是给出了创建实例的方法。 当你需要新的某物，你可以直接`new Thingamabob()`，或者随便什么你喜欢的表达法。 类是实例的生产工厂。

不用类，我们怎样创建新的实例？ 特别的，我们如何创建一堆有共同点的新东西？ 就像这个设计模式，在Self中，达到这点的方式是使用克隆。

在Self语言中，就好像每个对象都自动支持原型设计模式。 任何对象都能被克隆。为了获得一堆相似的对象，你：

1. 将对象塑造成你想要的状态。你可以直接克隆系统内建的基本`Object`，然后向其中添加字段和方法。
2. 克隆它来产出……额……随你想要多少就克隆多少个对象。

无需烦扰自己实现`clone()`；我们就实现了优雅的原型模式，原型被内建在系统中。

这个系统美妙，灵巧，而且小巧， 一听说它，我就开始创建一个基于原型的语言来进一步学习。

> 我知道从头开始构建一种编程语言语言不是学习它最有效率的办法，但我能说什么呢？我可算是个怪人。 如果你很好奇，我构建的语言叫Finch.

它的实际效果如何？

能使用纯粹基于原型的语言让我很兴奋，但是当我真正上手时， 我发现了一个令人不快的事实：用它编程没那么有趣。

> 从小道消息中，我听说很多Self程序员得出了相同的结论。 但这项目并不是一无是处。 Self非常的灵活，为此创造了很多虚拟机的机制来保持高速运行。
>
> 他们发明了JIT编译，垃圾回收，以及优化方法分配——这都是由同一批人实现的——这些新玩意让动态类型语言能快速运行，构建了很多大受欢迎的应用。

是的，语言本身很容易实现，那是因为它把复杂度甩给了用户。 一旦开始试着使用这语言，我发现我想念基于类语言中的层次结构。 最终，在构建语言缺失的库概念时，我放弃了。

鉴于我之前的经验都来自基于类的语言，因此我的头脑可能已经固定在它的范式上了。 但是直觉上，我认为大部分人还是喜欢有清晰定义的“事物”。

除去基于类语言自身的成功以外，看看有多少游戏用类建模描述玩家角色，以及不同的敌人、物品、技能。 不是游戏中的每个怪物都与众不同，你不会看到“洞穴人和哥布林还有雪混合在一起”这样的怪物。

原型是非常酷的范式，我希望有更多人了解它， 但我很庆幸不必天天用它编程。 完全皈依原型的代码是一团浆糊，难以阅读和使用。

> 这同时证明，很少 有人使用原型风格的代码。我查过了。

JavaScript又怎么样呢？

好吧，如果基于原型的语言不那么友好，怎么解释JavaScript呢？ 这是一个有原型的语言，每天被数百万人使用。运行JavaScript的机器数量超过了地球上其他所有的语言。

Brendan Eich，JavaScript的缔造者， 从Self语言中直接汲取灵感，很多JavaScript的语义都是基于原型的。 每个对象都有属性的集合，包含字段和“方法”（事实上只是存储为字段的函数）。 A对象可以拥有B对象，B对象被称为A对象的“原型”， 如果A对象的字段获取失败就会委托给B对象。

> 作为语言设计者，原型的诱人之处是它们比类更易于实现。 Eich充分利用了这一

点，他在十天内创建了JavaScript的第一个版本。

但除那以外，我相信在实践中，JavaScript更像是基于类的而不是基于原型的语言。 一个要点是JavaScript移除了一些基

在JavaScript中没有方法来克隆一个对象。 最接近的方法是`Object.create()`，允许你创建新对象作为现有对象的委托。 这个方法在ECMAScript5中才添加，而那已是JavaScript出现后的十四年了。 相对于克隆，让我带你参观一下JavaScript中定义类和创建对象的经典方法。 我们从构造器函数开始：

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

这创建了一个新对象，初始化了它的字段。你像这样引入它：

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

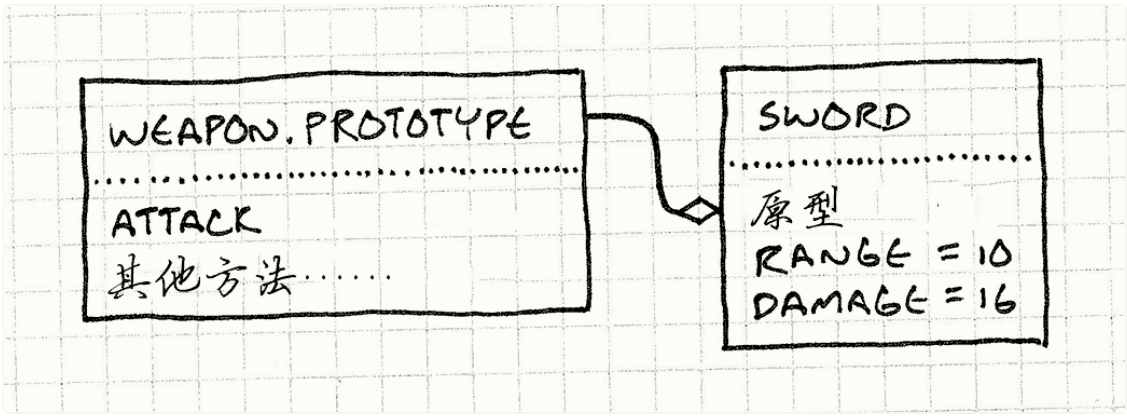
这里的`new`调用`Weapon()`函数，而`this`绑定在新的空对象上。 函数为新对象添加了一系列字段，然后返回填满的对象。

`new`也为你做了另外一件事。 当它创建那个新的空对象时，它将空对象的委托和一个原型对象连接起来。 你可以用`Weapon.prototype`来获得原型对象。

在构造器中添加属性，而通常向原型对象添加方法来定义行为。就像这样：

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

这给武器原型添加了`attack`属性，其值是一个函数。 由于`new Weapon()`返回的每一个对象都有给`Weapon.prototype`的委托， 你现在可以调用`sword.attack()`，这样调用那个函数。 看上去像是这样：



让我们复习一下：

- 通过“new”操作创建对象，该操作引入代表类型的对象——构造器函数。
- 状态存储在实例中。
- 行为通过间接层——原型的委托——被存储在独立的对象中，代表了一系列特定类型对象的共享方法。

说我疯了吧，但这听起来很像是我之前描述的类。 你可以在JavaScript中写原型风格的代码（不用 克隆），但是就语法和惯用法更推荐基于类的实现。

个人而言，我认为这是好事。 就像我说的，我发现一切都使用原型，那么很难编写代码，所以我喜欢JavaScript，它将整个核心语义包上了一层糖衣。

为数据模型构建原型

好吧，我之前不断的讨论我不喜欢原型的原因，这让这一章读起来令人沮丧。 我认为这本书该更欢乐些，所以在最后，让我们讨论讨论原型确实有用，或者更加精确，委托 有用的地方。

随着编程的进行，如果你比较程序与数据的字节数， 那么你会发现数据的占比稳定地增长。 早期的游戏在程序中生成几乎所有东西，这样程序可以塞进磁盘和老式游戏卡带。 在今日的游戏中，代码只是驱动游戏的“引擎”，游戏是完全由数据定义的。

这很好，但是将内容推到数据文件中并不能魔术般解决组织大项目的挑战。 它只能把这挑战变得更难。 我们使用编程语言就因为它们有办法管理复杂性。

不再是将一堆代码拷来拷去，我们将其移入函数中，通过名字调用。 不再是在一堆类之间复制方法，我们将其放入单独的类中，让其他类可以继承或者组合。

当游戏数据达到一定规模时，你真的需要考虑一些相似的方案。 我不指望在这里能说清数据模式这个问题， 但我确实希望提出个思路，让你在游戏中考虑考虑：使用原型和委托来重用数据。

假设我们为早先提到的山寨版《圣铠传说》定义数据模型。 游戏设计者需要在很多文件中设定怪物和物品的属性。

> 这标题是我原创的，没有受到任何已存的多人地下城游戏的影响。 请不要起诉我。

一个常用的方法是使用JSON。 数据实体一般是字典，或者属性集合，或者其他什么术语，因为程序员就喜欢为旧事物发明新名字。

> 我们重新发明了太多次，Steve Yegge称之为“通用设计模式”.

所以游戏中的哥布林也许被定义为像这样的东西：

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

这看上去很易懂，哪怕是最讨厌文本的设计者也能使用它。 所以，你可以给哥布林大家族添加几个兄弟分支：

```
{
  "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"]
}
```

现在，如果这是代码，我们会闻到了臭味。 在实体间有很多的重复，训练优良的程序员讨厌重复。 它浪费了空间，消耗了作者更多时间。 你需要仔细阅读代码才知道这些数据是不是相同的。 这难以维护。 如果我们决定让所有哥布林变强，需要记得将三个哥布林都更新一遍。糟糕糟糕糟糕。

如果这是代码，我们会为“哥布林”构建抽象，并在三个哥布林类型中重用。 但是无能的JSON没法这么做。所以让我们把它做得更加巧妙些。

我们可以为对象添加`"prototype"`字段，记录委托对象的名字。 如果在此对象内没找到一个字段，那去委托对象中查找。

> 这让`"prototype"`不再是数据，成为了元数据。 哥布林有绿色疣皮和黄色牙齿。它们没有原型。 原型是表示哥布林的数据模型的属性，而不是哥布林本身的属性。

这样，我们可以简化我们的哥布林JSON内容：

```
{
  "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"]
}
```

由于弓箭手和术士都将grunt作为原型，我们就不需要在它们中重复血量，防御和弱点。 我们为数据模型增加的逻辑超级简单——基本的单一委托——但已经成功摆脱了一堆冗余。

有趣的事情是，我们没有更进一步，把哥布林委托的抽象原型设置成“基本哥布林”。 相反，我们选择了最简单的哥布林，然后委托给它。

在基于原型的系统中，对象可以克隆产生新对象是很自然的， 我认为在这里也一样自然。这特别适合记录那些只有一处不同的实体的数据。

想想Boss和其他独特的事物，它们通常是更加常见事物的重新定义， 原型委托是定义它们的好方法。 断头魔剑，就是一把拥有加成的长剑，可以像下面这样表示：

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

只需在游戏引擎上多花点时间，你就能让设计者更加方便地添加不同的武器和怪物，而增加的这些丰富度能够取悦玩家。

← 上一章 ≡ 首页 下一章 →

单例模式

游戏设计模式 / Design Patterns Revisited

这个章节不同寻常。 其他章节展示如何使用某个个设计模式。 这个章节展示如何避免使用某个设计模式。

尽管它的意图是好的，GoF描述的单例模式 [GoF]通常弊大于利。 他们强调应该谨慎使用这个模式，但在游戏业界的口口相传中，这一提示经常被无视了。

就像其他模式一样，在不合适的地方使用单例模式就好像用夹板处理子弹伤口。 由于它被滥用得太严重了，这章的大部分都在讲如何回避单例模式， 但首先，让我们看看模式本身。

> 当业界从C语言迁移到面向对象的语言，他们遇到的首个问题是“如何访问实例？”他们知道有要调用的方法，但是找不到实例提供这个方法。 单例（换言之，全局化）是一条简单的解决方案。

单例模式

设计模式 像这样描述单例模式：

> 保证一个类只有一个实例，并且提供了访问该实例的全局访问点。

我们从“并且”那里将句子分为两部分，分别进行考虑。

保证一个类只有一个实例

有时候，如果类存在多个实例就不能正确的运行。 通常发生在类与保存全局状态的外部系统互动。

考虑封装文件系统的API类。 因为文件操作需要一段时间完成，所以类使用异步操作。 这就意味着可以同时运行多个操作，必须让它们相互协调。 如果一个操作创建文件，另一个操作删除同一文件，封装器类需要同时考虑，保证它们没有相互妨碍。

为了实现这点，对我们封装器类的调用必须接触之前的每个操作。 如果用户可以自由的创建类的实例，这个实例就无法知道另一实例之前的操作。 而单例模式提供的构建类的方式，在编译时保证类只有单一实例。

提供了访问该实例的全局访问点

游戏中的不同系统都会使用文件系统封装类：日志，内容加载，游戏状态保存，等等。 如

果这些系统不能创建文件系统封装类的实例，它们如何访问该实例呢？

单例为这点也提供了解决方案。 除了创建单一实例以外，它也提供了一种获得它的全局方法。 使用这种范式，无论何处何人都可以访问实例。 综合起来，经典的实现方案如下：

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    // 惰性初始化
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

静态的`instance_`成员保存了一个类的实例， 私有的构造器保证了它是唯一的。 公开的静态方法`instance()`让任何地方的代码都能访问实例。 在首次被请求时，它同样负责惰性实例化该单例。

现代的实现方案看起来是这样的：

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

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

哪怕是在多线程情况下，C++11标准也保证了本地静态变量只会初始化一次， 因此，假设你有一个现代C++编译器，这段代码是线程安全的，而前面的那个例子不是。

> 当然，你单例类本身的线程安全是个不同的问题！这里只保证了它的初始化没问题。

为什么我们使用它

看起来已有成效。 文件系统封装类在任何需要的地方都可用，而无需笨重地到处传递。 类

本身巧妙地保证了我们不会实例化多个实例而搞砸。它还具有很多其他的优良性质：

- 如果没人用，就不必创建实例。 节约内存和CPU循环总是好的。 由于单例只在第一次被请求时实例化，如果游戏永远不请求，那么它不会被实例化。
- 它在运行时实例化。 通常的替代方案是使用含有静态成员变量的类。 我喜欢简单的解决方案，因此我尽可能使用静态类而不是单例，但是静态成员有个限制：自动初始化。编译器在`main()`运行前初始化静态变量。 这就意味着不能使用在程序加载时才获取的信息（举个例子，从文件加载的配置）。 这也意味着它们的相互依赖是不可靠的——编译器可不保证以什么样的顺序初始化静态变量。

  惰性初始化解决了以上两个问题。 单例会尽可能晚的初始化，所以那时它需要的所有信息都应该可用了。 只要没有环状依赖，一个单例在初始化它自己的时甚至可以引用另一个单例。
- 可继承单例。 这是个很有用但通常被忽视的能力。 假设我们需要跨平台的文件系统封装类。 为了达到这一点，我们需要它变成文件系统抽象出来的接口，而子类为每个平台实现接口。 这是基类：

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

  然后为一堆平台定义子类：

  ```
  class PS3FileSystem : public FileSystem
  {
  public:
    virtual char* readFile(char* path)
    {
      // 使用索尼的文件读写API……
    }

    virtual void writeFile(char* path, char* contents)
    {
      // 使用索尼的文件读写API……
    }
  };

  class WiiFileSystem : public FileSystem
  {
  public:
    virtual char* readFile(char* path)
    {
      // 使用任天堂的文件读写API……
    }
  ```

```
  virtual void writeFile(char* path, char* contents)
  {
    // 使用任天堂的文件读写API……
  }
};
```

下一步，我们把FileSystem变成单例：

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

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

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

灵巧之处在于如何创建实例：

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

  return *instance;
}
```

通过一个简单的编译器转换，我们把文件系统包装类绑定到合适的具体类型上。 整个代码库都可以使用FileSystem::instance()接触到文件系统，而无需和任何平台相关的代码耦合。耦合发生在为特定平台写的FileSystem类实现文件中。

大多数人解决问题到这个程度就已经够了。 我们得到了一个文件系统封装类。 它工作可靠，它全局有效，只要请求就能获取。 是时候提交代码，开怀畅饮了。

为什么我们后悔使用它

短期来看，单例模式是相对良性的。 就像其他设计决策一样，我们需要从长期考虑。 这里是一旦我们将一些不必要的单例写进代码，会给自己带来的麻烦：

它是一个全局变量

当游戏还是由几个家伙在车库中完成时，榨干硬件性能比象牙塔里的软件工程原则更重要。C语言和汇编程序员前辈能毫无问题的使用全局变量和静态变量，发布好游戏。 但随着游戏

变得越来越大，越来越复杂，架构和管理开始变成瓶颈， 阻碍我们发布游戏的，除了硬件限制，还有生产力限制。

所以我们迁移到了像C++这样的语言， 开始将一些从软件工程师前辈那里学到的智慧应用于实际。 其中一课是全局变量有害的诸多原因：

- 理解代码更加困难。 假设我们在查找其他人所写函数中的漏洞。 如果函数没有碰到任何全局状态，脑子只需围着函数转， 只需搞懂函数和传给函数的变量。

  > 计算机科学家称不接触不修改全局状态的函数为“纯”函数。 纯函数易于理解，易于编译器优化， 易于完成优雅的任务，比如记住缓存的情况并继续上次调用。
  >
  > 完全使用纯函数是有难度的，但其好处足以引诱科学家创造像Haskell这样只使用纯函数的语言。

  现在考虑函数中间是个对`SomeClass::getSomeGlobalData()`的调用。为了查明发生了什么，得追踪整个代码库来看看什么修改了全局变量。你真的不需要讨厌全局变量，直到你在凌晨三点使用`grep`搜索数百万行代码，搞清楚哪一个错误的调用将一个静态变量设为了错误的值。

- 促进了耦合的发生。 新加入团队的程序员也许不熟悉你们完美，可维护，松散耦合的游戏架构， 但还是刚刚获得了第一个任务：在岩石撞击地面时播放声音。 你我都知道这不需要将物理和音频代码耦合，但是他只想着把任务完成。 不幸的是，我们的`AudioPlayer`是全局可见的。 所以之后一个小小的`#include`，新队员就打乱了整个精心设计的架构。

  如果不用全局实例实现音频播放器，那么哪怕他确实用`#include`包含了头文件，他还是啥也做不了。 这种阻碍给他发送了一个明确的信号，这两个模块不该接触，他需要另辟蹊径。通过控制对实例的访问，你控制了耦合。

- 对并行不友好。 那些在单核CPU上运行游戏的日子已经远去。 哪怕完全不需要并行的优势，现代的代码至少应考虑在多线程环境下工作。 当我们将某些东西转为全局变量时，我们创建了一块每个线程都能看到并访问的内存， 却不知道其他线程是否正在使用那块内存。 这种方式带来了死锁，竞争状态，以及其他很难解决的线程同步问题。

像这样的问题足够吓阻我们声明全局变量了， 同理单例模式也是一样，但是那还没有告诉我们应该如何设计游戏。 怎样不使用全局变量构建游戏？

有几个对这个问题的答案（这本书的大部分都是由答案构成）， 但是它们并非显而易见。与此同时，我们得发布游戏。 单例模式看起来是万能药。 它被写进了一本关于面向对象设计模式的书中，因此它肯定是个好的设计模式，对吧？ 况且我们已经借助它做了很多年软件设计了。

不幸的是，它不是解药，它是安慰剂。 如果浏览全局变量造成的问题列表，你会注意到单例模式解决不了其中任何一个。 因为单例确实是全局状态——它只是被封装在一个类中。

它能在你只有一个问题的时候解决两个

在GoF对单例模式的描述中，“并且”这个词有点奇怪。 这个模式解决了一个问题还是两个问题呢？如果我们只有其中一个问题呢？ 保证实例是唯一存在的很有用的，但是谁告诉我们要让每个人都能接触到它？ 同样，全局接触很方便，但是必须禁止存在多个实例吗？

这两问题中的后者，便利的访问，是使用单例模式几乎全部的原因。 想想日志类。大部分

模块都能从记录诊断日志中获益。 但是，如果将Log类的实例传给每个需要这个方法的函数，那就混杂了产生的数据，模糊了代码的意图。

明显的解决方案是让Log类成为单例。 每个函数都能从类那里获得一个实例。 但当我们这样做时，我们无意地制造了一个奇怪的小约束。 突然之间，我们不再能创建多个日志记录者了。

起初，这不是一个问题。 我们记录单独的日志文件，所以只需要一个实例。 然后，随着开发周期的逐次循环，我们遇到了麻烦。 每个团队的成员都使用日志记录各自的诊断信息，大量的日志倾泻在文件里。 程序员需要翻过很多页代码来找到他关心的记录。

我们想将日志分散到多个文件中来解决这点。 为了达到这点，我们得为游戏不同的领域创造单独的日志记录者： 网络，UI，声音，游戏，玩法。 但是我们做不到。 Log类不再允许我们创建多个实例，而且调用的方式也保证了这一点：

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

为了让Log类支持多个实例（就像它原来的那样）， 我们需要修改类和提及它的每一行代码。 之前便利的访问就不再那么便利了。

> 这可能更糟。想象一下你的Log类是在多个游戏间共享的库中。 现在，为了改变设计，需要在多组人之间协调改变， 他们中的大多数既没有时间，也没有动机修复它。

惰性初始化从你那里剥夺了控制权

拥有虚拟内存和软性性能需求的PC里，惰性初始化是一个小技巧。 游戏则是另一种状况。初始化系统需要消耗时间：分配内存，加载资源，等等。 如果初始化音频系统消耗了几百个毫秒，我们需要控制它何时发生。 如果在第一次声音播放时惰性初始化它自己，这初始化有可能发生游戏的高潮，导致可见的掉帧和断续的游戏体验。

同样，游戏通常需要严格管理在堆上分配的内存来避免碎片。 如果音频系统在初始化时分配到了堆上，我们需要知道何时初始化发生了， 这样我们可以控制内存待在堆的哪里。

> 对象池模式一节中有内存碎片的其他细节。

因为这两个原因，我见到的大多数游戏都不使用惰性初始化。 相反，它们像这样实现单例模式：

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

private:
  FileSystem() {}

  static FileSystem instance_;
};
```

这解决了惰性初始化问题，但是损失了几个单例确实比原生的全局变量优良的特性。 静态

实例中，我们不能使用多态，在静态初始化时，类也必须是可构建的。 我们也不能在不需要这个实例的时候，释放实例所占的内存。

与创建一个单例不同，这里实际上是一个简单的静态类。 这并非坏事，但是如果你需要的是静态类，为什么不完全摆脱`instance()`方法， 直接使用静态函数呢？调用`Foo::bar()`比`Foo::instance().bar()`更简单， 也更明确地表明你在处理静态内存。

> 通常使用单例而不是静态类的理由是， 如果你后来决定将静态类改为非静态的，你需要修改每一个调用点。 理论上，用单例就不必那么做，因为你可以将实例传来传去，像普通的实例方法一样使用。
>
> 实践中，我从未见过这种情况。 每个人都在使用`Foo::instance().bar()`。 如果我们将Foo改成非单例，我们还是得修改每一个调用点。 鉴于此，我更喜欢简单的类和简单的调用语法。

那该如何是好

如果我现在达到了目标，你在下次遇到问题使用单例模式之前就会三思而后行。 但是你还是有问题需要解决。你应该使用什么工具呢？ 这取决于你试图做什么，我有一些你可以考虑的选项，但是首先……

看看你是不是真正地需要类

我在游戏中看到的很多单例类都是“管理器”——那些类存在的意义就是照顾其他对象。 我曾看到一些代码库中，几乎所有类都有管理器： 怪物，怪物管理器，粒子，粒子管理器，声音，声音管理器，管理管理器的管理器。 有时候，它们被叫做“系统”或“引擎”，但是思路还是一样的。

管理器类有时是有用的，但通常它们只是反映出作者对OOP的不熟悉。思考这两个特制的类：

```
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);
  }
};
```

也许这个例子有些蠢，但是我见过很多代码，在剥离了外部的细节后是一样的设计。 如果你看看这个代码，`BulletManager`很自然应是一个单例。 无论如何，任何有`Bullet`的对象都需要管理，而你又需要多少个`BulletManager`实例呢？

事实上，这里的答案是零。 这里是我们如何为管理类解决“单例”问题：

```
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_;
};
```

好了。没有管理器，也没有问题。 糟糕设计的单例通常会“帮助”另一个类增加代码。 如果可以，把所有的行为都移到单例帮助的类中。 毕竟，OOP就是让对象管理好自己。

但是在管理器之外，还有其他问题我们需要寻求单例模式帮助。 对于每种问题，都有一些后续方案可供参考。

将类限制为单一的实例

这是单例模式帮你解决的一个问题。 就像在文件系统的例子中那样，保证类只有一个实例

是很重要的。 但是，这不意味着我们需要提供对实例的公众全局访问。 我们想要减少某部分代码的公众部分，甚至让它在类中是私有的。 在这些情况下，提供一个全局接触点消弱了整体架构。

> 举个例子，我们也许想把文件系统包在另一层抽象中。

我们希望有种方式能保证同事只有一个实例而无需提供全局接触点。 有好几种方法能做到。这是其中之一：

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

  ~FileSystem() { instantiated_ = false; }

private:
  static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

这个类允许任何人构建它，如果你试图构建超过一个实例，它会断言并失败。 只要正确的代码首先创建了实例，那么就保证了没有其他代码可以接触实例或者创建自己的实例。 这个类保证满足了它关注的单一实例，但是它没有指定类该如何被使用。

> 断言 函数是一种向你的代码中添加限制的方法。 当`assert()`被调用时，它计算传入的表达式。 如果结果为`true`，那么什么都不做，游戏继续。 如果结果为`false`，它立刻停止游戏。 在debug build时，这通常会启动调试器，或至少打印失败断言所在的文件和行号。
>
> `assert()`表示，“我断言这个总该是真的。如果不是，那就是漏洞，我想立刻停止并处理它。” 这样你在代码区域之间定义约束。 如果函数断言它的某个参数不能为`NULL`，那就是说，“我和调用者定下了协议：传入的参数不会`NULL`。”
>
> 断言帮助我们在游戏发生预期以外的事时立刻追踪漏洞， 而不是等到错误最终显现在用户可见的某些事物上。 它们是代码中的栅栏，围住漏洞，这样漏洞就不能从制造它的代码边逃开。

这个实现的缺点是只在运行时检查并阻止多重实例化。 单例模式，正相反，通过类的自然结构，在编译时就能确定实例是单一的。

为了给实例提供方便的访问方法

便利的访问是我们使用单例的一个主要原因。 这让我们在不同地方获取需要的对象更加容易。 这种便利是需要付出代价的——在我们不想要对象的地方，也能轻易地使用。

通用原则是在能完成工作的同时，将变量写得尽可能局部。 对象影响的范围越小，在处理

它时，我们需要放在脑子里的东西就越少。 在我们拿起有全局范围影响的单例对象前，先考虑考虑代码中其他获取对象的方式：

- 传进来。 最简单的解决办法，通常也是最好的，把你需要的对象简单地作为参数传给需要它的函数。 在用其他更加繁杂的方法前，考虑一下这个解决方案。

  > 有些人使用术语“依赖注入”来指代它。不是代码出来调用某些全局量来确认依赖， 而是依赖通过参数被传进到需要它的代码中去。 其他人将“依赖注入”保留为对代码提供更复杂依赖的方法。

  考虑渲染对象的函数。为了渲染，它需要接触一个代表图形设备的对象，管理渲染状态。 将其传给所有渲染函数是很自然的，通常是用一个名字像`context`之类的参数。

  另一方面，有些对象不该在方法的参数列表中出现。 举个例子，处理AI的函数可能也需要写日志文件，但是日志不是它的核心关注点。 看到`Log`出现在它的参数列表中是很奇怪的事情，像这样的情况，我们需要考虑其他的选项。

  > 像日志这样撒布在代码库各处的是“横切关注点”(cross-cutting concern)。小心地处理横切关注点是架构中的持久挑战，特别是在静态类型语言中。
  >
  > 面向切面编程被设计出来应对它们。

- 从基类中获得。 很多游戏架构有浅层但是宽泛的继承层次，通常只有一层深。 举个例子，你也许有`GameObject`基类，每个游戏中的敌人或者对象都继承它。 使用这样的架构，很大一部分游戏代码会存在这些“子”推导类中。 这就意味着这些类已经有了对同样事物的相同获取方法：它们的`GameObject`基类。 我们可以利用这点：

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

  private:
    static Log& log_;
  };

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

  这保证任何`GameObject`之外的代码都不能接触`Log`对象，但是每个派生的实体确实能使用`getLog()`。 这种使用protected函数，让派生对象使用的模式， 被涵盖在子类沙箱「这章中。

  > 这也引出了一个新问题，“`GameObject`是怎样获得`Log`实例的？”一个简单的方案是，让基类创建并拥有静态实例。
  >
  > 如果你不想要基类承担这些，你可以提供一个初始化函数传入`Log`实例， 或使

用服务定位器模式找到它。

- 从已经是全局的东西中获取。 移除所有全局状态的目标令人钦佩，但并不实际。 大多数代码库仍有一些全局可用对象，比如一个代表了整个游戏状态的Game或World对象。

  我们可以让现有的全局对象捎带需要的东西，来减少全局变量类的数目。 不让Log，FileSystem和AudioPlayer都变成单例，而是这样做：

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

    // 设置log_, et. al. ……

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

  private:
    static Game instance_;

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

  这样，只有Game是全局可见的。 函数可以通过它访问其他系统。

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

  > 纯粹主义者会声称这违反了Demeter法则。我则声称这比一大坨单例要好。

  如果，稍后，架构被改为支持多个Game实例（可能是为了流处理或者测试）， Log，FileSystem，和AudioPlayer都不会被影响到——它们甚至不知道有什么区别。 缺陷是，当然，更多的代码耦合到了Game中。 如果一个类简单地需要播放声音，为了访问音频播放器，上例中仍然需要它知道游戏世界。

  我们通过混合方案解决这点。 知道Game的代码可以直接从它那里访问AudioPlayer。而不知道的代码，我们用上面描述的其他选项来提供AudioPlayer。

- 从服务定位器中获得。 目前为止，我们假设全局类是具体的类，比如Game。 另一种选项是定义一个类，存在的唯一目标就是为对象提供全局访问。 这种常见的模式被称为服务定位器模式，有单独讲它的章节。

单例中还剩下什么

剩下的问题，何处我们应该使用真实的单例模式？ 说实话，我从来没有在游戏中使用全部的GoF模式。 为了保证实例是单一的，我通常简单的使用静态类。 如果这无效，我使用静

态标识位，在运行时检测是不是只有一个实例被创建了。

书中还有一些其他章节也许能有所帮助。 子类沙箱 模式通过分享状态， 给实例以类的访问权限而无需让其全局可用。 服务定位器 模式确实让一个对象全局可用， 但它给了你如何设置对象的灵活性。

状态模式

游戏设计模式 / Design Patterns Revisited

忏悔时间：我有些越界，将太多的东西打包到了这章中。 它表面上关于状态模式 GoF ， 但我无法只讨论它和游戏，而不涉及更加基础的有限状态机（ FSMs ）。 但是一旦讲了那个，我发现也想要介绍层次状态机和下推自动机。

有很多要讲，我会尽可能简短，这里的示例代码留下了一些你需要自己填补的细节。 我希望它们仍然足够清晰，能让你获取一份全景图。

如果你从来没有听说过状态机，不要难过。 虽然在AI和编译器程序很出名，但它在其他编程圈就没那么知名了。 我认为应该有更多人知道它，所以在这里我将其运用在不同的问题上。

> 这些状态机术语来自人工智能的早期时代。 在五十年代到六十年代，很多AI研究关注于语言处理。 很多现在用于分析程序语言的技术在当时是发明出来分析人类语言的。

感同身受

假设我们在完成一个卷轴平台游戏。 现在的工作是实现玩家在游戏世界中操作的女英雄。这就意味着她需要对玩家的输入做出响应。按B键她应该跳跃。简单实现如下：

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

看到漏洞了吗？

没有东西阻止“空中跳跃”——当角色在空中时狂按B，她就会浮空。 简单的修复方法是给Heroine增加isJumping_布尔字段，追踪它跳跃的状态。然后这样做：

```
void Heroine::handleInput(Input input)
{
```

```
  if (input == PRESS_B)
  {
    if (!isJumping_)
    {
      isJumping_ = true;
      // 跳跃……
    }
  }
}
```

这里也应该有在英雄接触到地面时将`isJumping_`设回`false`的代码。 我在这里为了简明没有写。

接下来，当玩家按下下方向键时，如果角色在地上，我们想要她卧倒，而松开按键时站起来：

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    // 如果没在跳跃，就跳起来……
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    setGraphics(IMAGE_STAND);
  }
}
```

这次看到漏洞了吗？

通过这个代码，玩家可以：

1. 按下键卧倒。
2. 按B从卧倒状态跳起。
3. 在空中放开下键。

英雄跳一半贴图变成了站立时的贴图。是时候增加另一个标识了……

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
```

```
      // 跳跃……
    }
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    if (isDucking_)
    {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

下面，如果玩家在跳跃途中按下下方向键，英雄能够做跳斩攻击就太酷了：

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // 跳跃……
    }
  }
  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_)
    {
      // 站立……
    }
```

```
    }
}
```

又是检查漏洞的时间了。找到了吗？

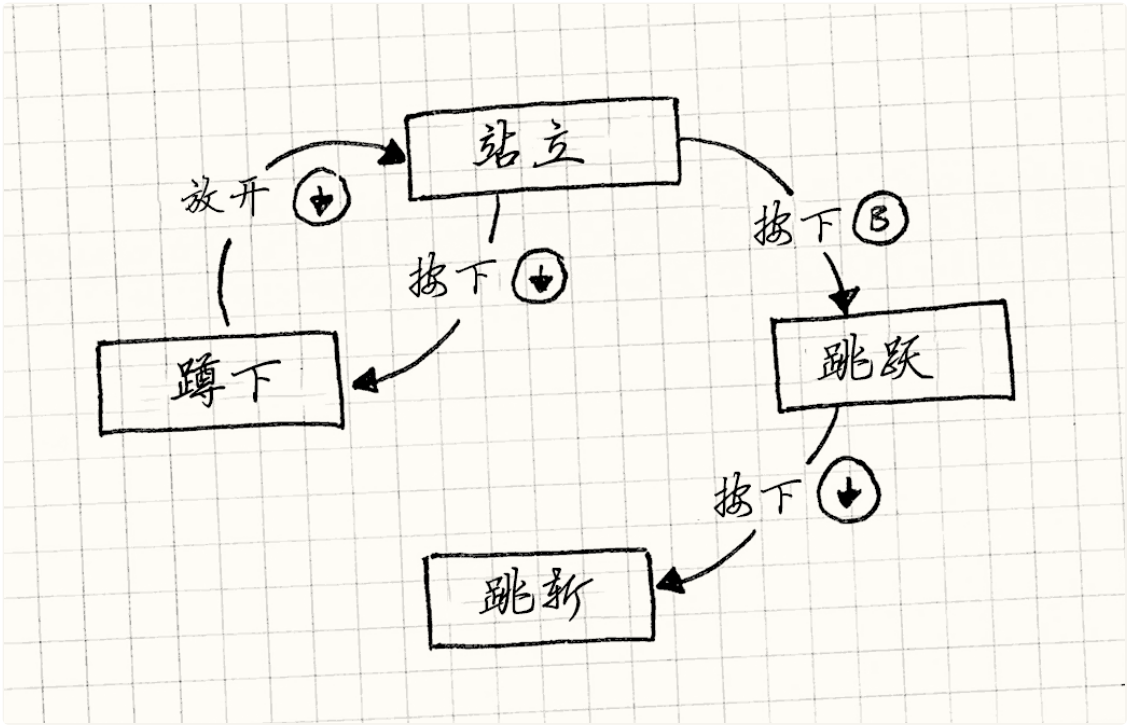
跳跃时我们检查了字段，防止了空气跳，但是速降时没有。又是另一个字段……

我们的实现方法很明显有错。 每次我们改动代码时，就破坏些东西。 我们需要增加更多动作——行走 都还没有加入呢——但以这种做法，完成之前就会造成一堆漏洞。

> 那些你崇拜的、看上去永远能写出完美代码的程序员并不是超人。 相反，他们有哪种代码易于出错的直觉，然后避开。
>
> 复杂分支和可变状态——随时间改变的字段——是两种易错代码，上面的例子覆盖了两者。

有限状态机前来救援

在经历了上面的挫败之后，把桌子扫空，只留下纸笔，我们开始画流程图。 你给英雄每件能做的事情都画了一个盒子：站立，跳跃，俯卧，跳斩。 当角色在能响应按键的状态时，你从那个盒子画出一个箭头，标记上按键，然后连接到她变到的状态。



祝贺，你刚刚建好了一个有限状态机。 它来自计算机科学的分支自动理论，那里有很多著名的数据结构，包括著名的图灵机。 FSMs是其中最简单的成员。

要点是：

- 你拥有状态机所有可能状态的集合。 在我们的例子中，是站立，跳跃，俯卧和速降。
- 状态机同时只能在一个状态。 英雄不可能同时处于跳跃和站立。事实上，防止这点是使用FSM的理由之一。
- 一连串的输入或事件被发送给状态机。 在我们的例子中，就是按键按下和松开。

每个状态都有一系列的转移，每个转移与输入和另一状态相关。 当输入进来，如果它与当前状态的某个转移相匹配，机器转换为所指的状态。

举个例子，在站立状态时，按下下方向键转换为俯卧状态。 在跳跃时按下下方向键转换为速降。 如果输入在当前状态没有定义转移，输入就被忽视。

这就是核心部分的全部了：状态，输入，和转移。 你可以用一张流程图把它画出来。不幸的是，编译器不认识流程图， 所以我们如何实现一个？ GoF的状态模式是一个方法——我们会谈到的——但先从简单的开始。

> 对FSMs我最喜欢的类比是那种老式文字冒险游戏，比如Zork。 你有世界由屋子组成，屋子彼此通过出口相连。你输入像“去北方”的导航指令探索屋子。
>
> 这其实就是状态机：每个屋子都是一个状态。 你现在在的屋子是当前状态。每个屋子的出口是它的转移。 导航指令是输入。

枚举和分支

`Heroine`类的问题在于它不合法地捆绑了一堆布尔量： `isJumping_`和`isDucking_`不会同时为真。 但有些标识同时只能有一个是`true`，这提示是你真正需要的其实是`enum`（枚举）。

在这个例子中的`enum`就是FSM的状态的集合，所以让我们这样定义它：

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

不需要一堆标识，`Heroine`只有一个`state_`状态。 这里我们同时改变了分支顺序。在前面的代码中，我们先在判断输入，然后 判断状态。 这让处理某个按键的代码集中到了一处，但处理某个状态的代码分散到了各处。 我们想让处理状态的代码聚在一起，所以先对状态做分支。这样的话：

```
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;
  }
}
```

这看起来很普通，但是比起前面的代码是个很大的进步。 我们仍有条件分支，但简化了状态变化，将它变成了字段。 处理同一状态的所有代码都聚到了一起。 这是实现状态机最简单的方法，在某些情况下，这也不错。

> 重要的是，英雄不再会处于不合法状态。 使用布尔标识，很多可能存在的值的组合是不合法的。 通过enum，每个值都是合法的。

但是，你的问题也许超过了这个解法的能力范围。 假设我们想增加一个动作动作，英雄可以俯卧一段时间充能，之后释放一次特殊攻击。 当她俯卧时，我们需要追踪充能的持续时间。

我们为Heroine添加了chargeTime_字段，记录充能的时间长度。 假设我们已经有一个每帧都会调用的update()方法。在那里，我们添加：

```
void Heroine::update()
{
  if (state_ == STATE_DUCKING)
  {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
      superBomb();
    }
  }
}
```

> 如果你猜这就是更新方法模式，恭喜你答对了！

我们需要在她开始俯卧的时候重置计时器，所以我们修改handleInput()：

```
void Heroine::handleInput(Input input)
{
  switch (state_)
  {
    case STATE_STANDING:
      if (input == PRESS_DOWN)
      {
        state_ = STATE_DUCKING;
        chargeTime_ = 0;
        setGraphics(IMAGE_DUCK);
      }
      // 处理其他输入……
      break;

      // 其他状态……
  }
}
```

总而言之，为了增加这个充能攻击，我们需要修改两个方法， 添加一个chargeTime_字段到Heroine，哪怕它只在俯卧时有意义。 我们更喜欢的是让所有相关的代码和数据都待在同一个地方。GoF完成了这个。

状态模式

对于那些思维模式深深沉浸在面向对象的人，每个条件分支都是使用动态分配的机会（在C++中叫做虚方法调用）。 我觉得那就太过于复杂化了。有时候一个if就能满足你的需要了。

> 这里有个历史遗留问题。 原先的面向对象传教徒，比如写《设计模式》的GoF和写《重构》的Martin Fowler都使用Smalltalk。 那里，ifThen:是你只是使用条件的方法，该方法在true和false对象中以不同的方式实现。

但是在我们的例子中，面向对象确实是一个更好的方案。 这带领我们走向状态模式。在GoF中这样描述状态模式：

> 允许一个对象在其内部状态 发生变化时改变自己的行为，该对象看起来好像修改了它的类型

这可没太多帮助。我们的switch也完成了这一点。 它们描述的东西应用在英雄的身上实际是：

一个状态接口

首先，我们为状态定义接口。 状态相关的行为——之前用switch的每一处——都成为了接口中的虚方法。 在我们的例子中，那是handleInput()和update()：

```
class HeroineState
```

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

为每个状态写个类

对于每个状态，我们定义一个类实现接口。它的方法定义了英雄在状态的行为。 换言之，从之前的`switch`中取出每个`case`，将它们移动到状态类中。举个例子：

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

  virtual void handleInput(Heroine& heroine, Input input) {
    if (input == RELEASE_DOWN)
    {
      // 改回站立状态……
      heroine.setGraphics(IMAGE_STAND);
    }
  }

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

private:
  int chargeTime_;
};
```

注意我们也将`chargeTime_`移出了`Heroine`，放到了`DuckingState`类中。 这很好——那部分数据只在这个状态有用，现在我们的对象模型显式反映了这一点。

状态委托

接下来，向`Heroine`添加指向当前状态的指针，放弃庞大的`switch`，转向状态委托：

```
class Heroine
{
public:
  virtual void handleInput(Input input)
  {
```

```
    state_->handleInput(*this, input);
  }

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

  // 其他方法……
private:
  HeroineState* state_;
};
```

为了“改变状态”，我们只需要将`state_`声明指向不同的`HeroineState`对象。 这就是状态模式的全部了。

> 这看上去有些像策略 GoF模式和类型对象 模式。 在三者中，你都有一个主对象委托给下属。区别在于意图。
>
> - 在策略模式中，目标是解耦主类和它的部分行为。
> - 在类型对象中，目标是通过共享一个对相同类型对象的引用，让一系列对象行为相近。
> - 在状态模式中，目标是让主对象通过改变委托的对象，来改变它的行为。

状态对象在哪里？

我这里掩掩藏了一些细节。为了改变状态，我们需要声明`state_`指向新的状态， 但那个新状态又是从哪里来呢？ 在`enum`实现中，这都不用过脑子——`enum`实际上就像数字一样。 但是现在状态是类了，意味着我们需要指向实例。通常这有两种方案：

静态状态

如果状态对象没有其他数据字段， 那么它存储的唯一数据就是指向虚方法表的指针，用来调用它的方法。 在这种情况下，没理由产生多个实例。毕竟每个实例都完全一样。

> 如果你的状态没有字段，只有一个虚方法，你可以再简化这个模式。 将每个状态类替换成状态函数——只是一个普通的顶层函数。 然后，主类中的`state_`字段变成一个简单的函数指针。

在那种情况下，你可以用一个静态实例。 哪怕你有一堆FSM同时在同一状态上运行，它们也能指向同一实例，因为状态没有与状态机相关的部分。

> 这是享元 GoF模式。

在哪里放置静态实例取决于你。找一个合理的地方。 没什么特殊的理由，在这里我将它放在状态基类中。

```
class HeroineState
{
public:
```

```
  static StandingState standing;
  static DuckingState ducking;
  static JumpingState jumping;
  static DivingState diving;

  // 其他代码……
};
```

每个静态字段都是游戏状态类的一个实例。为了让英雄跳跃，站立状态会这样做：

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

实例化状态

有时没那么容易。静态状态对俯卧状态不起作用。 它有一个`chargeTime_`字段，与正在俯卧的英雄特定相关。 在游戏中，如果只有一个英雄，那也行，但是如果要添加双人合作，同时在屏幕上有两个英雄，就有麻烦了。

在那种情况下，转换时需要创建状态对象。 这需要每个FSM拥有自己的状态实例。如果我们分配新状态， 那意味着我们需要释放当前的状态。 在这里要小心，由于触发变化的代码是当前状态中的方法，需要删除`this`，因此需要小心从事。

相反，我们允许`HeroineState`中的`handleInput()`返回一个新状态。 如果它那么做了，`Heroine`会删除旧的，然后换成新的，就像这样：

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

这样，直到从之前的状态返回，我们才需要删除它。 现在，站立状态可以通过创建新实例转换为俯卧状态：

```
HeroineState* StandingState::handleInput(Heroine& heroine,
                                         Input input)
{
  if (input == PRESS_DOWN)
  {
    // 其他代码……
    return new DuckingState();
  }
```

```
  // 保持这个状态
  return NULL;
}
```

如果可以，我倾向于使用静态状态，因为它们不会在状态转换时消耗太多的内存和CPU。但是，对于更，额，多状态的事物，需要耗费一些精力来实现。

> 当你为状态动态分配内存时，你也许会担心碎片。 对象池模式可以帮上忙。

入口行为和出口行为

状态模式的目标是将状态的行为和数据封装到单一类中。 我们完成了一部分，但是还有一些未了之事。

当英雄改变状态时，我们也改变她的贴图。 现在，那部分代码在她转换前的状态中。 当她从俯卧转为站立，俯卧状态修改了她的贴图：

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

  // 其他代码……
}
```

我们想做的是，每个状态控制自己的贴图。这可以通过给状态一个入口行为来实现：

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

  // 其他代码……
};
```

在Heroine中，我们将处理状态改变的代码移动到新状态上调用：

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

```
  {
    delete state_;
    state_ = state;

    // 调用新状态的入口行为
    state_->enter(*this);
  }
}
```

这让我们将俯卧代码简化为：

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

  // 其他代码……
}
```

它做的所有事情就是转换到站立状态，站立状态控制贴图。 现在我们的状态真正地封装了。 关于入口行为的好事就是，当你进入状态时，不必关心你是从哪个状态转换来的。

大多数真正的状态图都有转为同一状态的多个转移。 举个例子，英雄在跳跃或跳斩后进入站立状态。 这意味着我们在转换发生的最后重复相同的代码。 入口行为很好地解决了这一点。

我们能，当然，扩展并支持出口行为。 这是在我们离开现有状态，转换到新状态之前调用的方法。

有什么收获？

我花了这么长时间向您推销FSMs，现在我们来捋一捋。 我到现在讲的都是真的，FSM能很好解决一些问题。但它们最大的优点也是它们最大的缺点。

状态机通过使用有约束的结构来理清杂乱的代码。 你只需一个固定状态的集合，单一的当前状态，和一些硬编码的转换。

> 一个有限状态机甚至不是图灵完全的。 自动理论用一系列抽象模型描述计算，每种都比之前的复杂。 图灵机 是其中最具有表现力的模型之一。
>
> “图灵完全”意味着一个系统（通常是编程语言）足以在内部实现一个图灵机， 也就意味着，在某种程度上，所有的图灵完全具有同样的表现力。 FSMs不够灵活，并不在其中。

如果你需要为更复杂的东西使用状态机，比如游戏AI，你会撞到这个模型的限制上。 感谢上天，我们的前辈找到了一些方法来闪避这些限制。我会在这一章的最后简单地浏览一下它们。

并发状态机

我们决定给英雄拿枪的能力。 当她拿着枪的时候，她还是能做她之前的任何事情：跑动，跳跃，跳斩，等等。 但是她在做这些的同时也要能开火。

如果我们执着于FSM，我们需要翻倍现有状态。 对于每个现有状态，我们需要另一个她持枪状态：站立，持枪站立，跳跃，持枪跳跃， 你知道我的意思了吧。

多加几种武器，状态就会指数爆炸。 不但增加了大量的状态，这也增加了大量的冗余： 持枪和不持枪的状态是完全一样的，只是多了一点负责射击的代码。

问题在于我们将两种状态绑定——她做的和她携带的——到了一个状态机上。 为了处理所有可能的组合，我们需要为每一对组合写一个状态。 修复方法很明显：使用两个单独的状态机。

> 如果她在做什么有n个状态，而她携带了什么有m个状态，要塞到一个状态机中，我们需要n × m个状态。使用两个状态机，就只有n + m个。

我们保留之前记录她在做什么的状态机，不用管它。 然后定义她携带了什么的单独状态机。 Heroine将会有两个“状态”引用，每个对应一个状态机，就像这样：

```
class Heroine
{
  // 其他代码……

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

> 为了便于说明，她的装备也使用了状态模式。 在实践中，由于装备只有两个状态，一个布尔标识就够了。

当英雄把输入委托给了状态，两个状态都需要委托：

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

> 功能更完备的系统也许能让状态机销毁输入，这样其他状态机就不会收到了。 这能阻止两个状态机响应同一输入。

每个状态机之后都能响应输入，发生行为，独立于其它机器改变状态。 当两个状态集合几乎没有联系的时候，它工作得不错。

在实践中，你会发现状态有时需要交互。 举个例子，也许她在跳跃时不能开火，或者她在持枪时不能跳斩攻击。 为了完成这个，你也许会在状态的代码中做一些粗糙的`if`测试其他状态来协同， 这不是最优雅的解决方案，但这可以搞定工作。

分层状态机

再充实一下英雄的行为，她可能会有更多相似的状态。 举个例子，她也许有站立，行走，奔跑，和滑铲状态。在这些状态中，按B跳，按下蹲。

如果使用简单的状态机实现，我们在每个状态中的都重复了代码。 如果我们能够实现一次，在多个状态间重用就好了。

如果这是面向对象的代码而不是状态机的，在状态间分享代码的方式是通过继承。 我们可以为“在地面上”定义一个类处理跳跃和速降。 站立，行走，奔跑和滑铲都从它继承，然后增加各自的附加行为。

> 它的影响有好有坏。 继承是一种有力的代码重用工具，但也在两块代码间建立了非常强的耦合。 这是重锤，所以请小心使用。

你会发现，这是个被称为分层状态机的通用结构。 状态可以有父状态（这让它变为子状态）。 当一个事件进来，如果子状态没有处理，它就会交给链上的父状态。 换言之，它像重载的继承方法那样运作。

事实上，如果我们使用状态模式实现FSM，我们可以使用继承来实现层次。 定义一个基类作为父状态：

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

每个子状态继承它：

```
class DuckingState : public OnGroundState
{
public:
  virtual void handleInput(Heroine& heroine, Input input)
  {
    if (input == RELEASE_DOWN)
    {
      // 站起……
    }
```

```
    else
    {
      // 没有处理输入，返回上一层
      OnGroundState::handleInput(heroine, input);
    }
  }
};
```

这当然不是唯一实现层次的方法。 如果你没有使用GoF的状态模式，这可能不会有用。 相反，你可以显式的使用状态栈而不是单一状态来表示当前状态的父状态链。

栈顶的状态是当前状态，在他下面是它的直接父状态， 然后是那个父状态的父状态，以此类推。 当你需要状态的特定行为，你从栈的顶端开始， 然后向下寻找，直到某一个状态处理了它。（如果到底也没找到，就无视它。）

下推自动机

还有一种有限状态机的扩展也用了状态栈。 容易混淆的是，这里的栈表示的是完全不同的事物，被用作解决不同的问题。

要解决的问题是有限状态机没有任何历史的概念。 你记得正在什么状态中，但是不记得曾在什么状态。 没有简单的办法重回上一状态。

举个例子：早先，我们让无畏英雄武装到了牙齿。 当她开火时，我们需要新状态播放开火动画，发射子弹，产生视觉效果。 所以我们拼凑了一个`FiringState`，不管现在是什么状态，都能在按下开火按钮时跳转为这个状态。

> 这个行为在多个状态间重复，也许是用层次状态机重用代码的好地方。

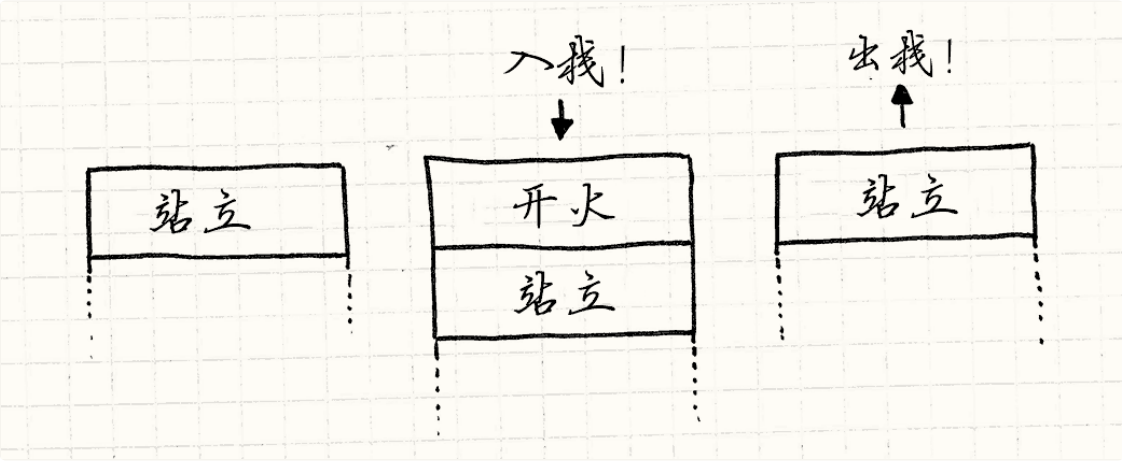
问题在于她射击后转换到的状态。 她可以在站立，奔跑，跳跃，跳斩时射击。 当射击结束，应该转换为她之前的状态。

如果我们固执于纯粹的FSM，我们就已经忘了她之前所处的状态。 为了追踪之前的状态，我们定义了很多几乎完全一样的类——站立开火，跑步开火，跳跃开火，诸如此类—— 每个都有硬编码的转换，用来回到之前的状态。

我们真正想要的是，它会存储开火前所处的状态，之后能回想起来。 自动理论又一次能帮上忙了，相关的数据结构被称为下推自动机。

有限状态机有一个指向状态的指针，下推自动机有一栈指针。 在FSM中，新状态代替了之前的那个状态。 下推自动机不仅能完成那个，还能给你两个额外操作：

1. 你可以将新状态压入栈中。“当前的”状态总是在栈顶，所以你能转到新状态。 但它让之前的状态待在栈中而不是销毁它。
2. 你可以弹出最上面的状态。这个状态会被销毁，它下面的状态成为新状态。



这正是我们开火时需要的。我们创建单一的开火状态。 当开火按钮在其他状态按下时，我们压入开火状态。 当开火动画结束，我们弹出开火状态，然后下推自动机自动转回之前的状态。

所以它们有多有用呢？

即使状态机有这些常见的扩展，它们还是很受限制。 这让今日游戏AI移向了更加激动人心的领域，比如行为树和规划系统 。 如果你关注复杂AI，这一整章只是为了勾起你的食欲。你需要阅读其他书来满足你的欲望。

这不意味着有限状态机，下推自动机，和其他简单的系统没有用。 它们是特定问题的好工具。有限状态机在以下情况有用：

- 你有个实体，它的行为基于一些内在状态。
- 状态可以被严格的分割为相对较少的不相干项目。
- 实体响应一系列输入或事件。

在游戏中，状态机因在AI中使用而闻名，但是它也常用于其他领域， 比如处理玩家输入，导航菜单界面，分析文字，网络协议以及其他异步行为。

← 上一章 ≡ 首页 下一章 →

序列模式

游戏设计模式

电子游戏之所有有趣，很大程度上归功于它们会将我们带到别的地方。 几分钟后（或者，诚实点，可能会更长），我们活在一个虚拟的世界。 创造那样的世界是游戏程序员至上的欢愉。

大多数游戏世界都有的特性是时间——虚构世界以其特定的节奏运行。 作为世界的架构师，我们必须发明时间，制造推动游戏时间运作的齿轮。

这本篇的模式是建构这些的工具。 游戏循环是时钟的中心轴。 对象通过更新方法来聆听时钟的滴答声。 我们可以用双缓冲模式存储快照来隐藏计算机的顺序执行，这样看起来世界可以进行同步更新。

模式

- 双缓冲模式
- 游戏循环
- 更新方法

双缓冲模式

意图

用序列的操作模拟瞬间或者同时发生的事情。

动机

电脑具有强大的序列化处理能力。 它的力量来自于将大的任务分解为小的步骤，这样可以一步接一步的完成。 但是，通常用户需要看到事情发生在瞬间或者让多个任务同时进行。

> 使用线程和多核架构让这种说法不那么正确了，但哪怕使用多核，也只有一些操作可以同步运行。

一个典型的例子，也是每个游戏引擎都得掌控的问题，渲染。 当游戏渲染玩家所见的世界时，它同时需要处理一堆——远处的山，起伏的丘陵，树木，每个都在各自的循环中处理。 如果在用户观察时增量做这些，连续世界的幻觉就会被打破。 场景必须快速流畅地更新，显示一系列完整的帧，每帧都是立即出现的。

双缓冲解决了这个问题，但是为了理解其原理，让我们首先的复习下计算机是如何显示图形的。

计算机图形系统是如何工作的（概述）

在电脑屏幕上显示图像是一次绘制一个像素点。 它从左到右扫描每行像素点，然后移动至下一行。 当抵达了右下角，它退回左上角重新开始。 它做得飞快——每秒六十次——因此我们的眼睛无法察觉。 对我们来说，这是一整张静态的彩色像素——一张图像。

> 这个解释是，额，“简化过的”。 如果你是底层软件开发人员，跳过下一节吧。 你对这章的其余部分已经了解的够多了。 如果你不是，这部分的目标是给你足够的背景知识，理解等要讨论的设计模式。

你可以将整个过程想象为软管向屏幕喷洒像素。 独特的像素从软管的后面流入，然后在屏幕上喷洒，每次对一个像素涂一点颜色。 所以软管怎么知道哪种颜色要喷到哪里？

在大多数电脑上，答案是从帧缓冲中获知这些信息。 帧缓冲是内存中的色素数组，RAM中每两个字节代表表示一个像素点的颜色。 当软管向屏幕喷洒时，它从这个数组中读取颜色

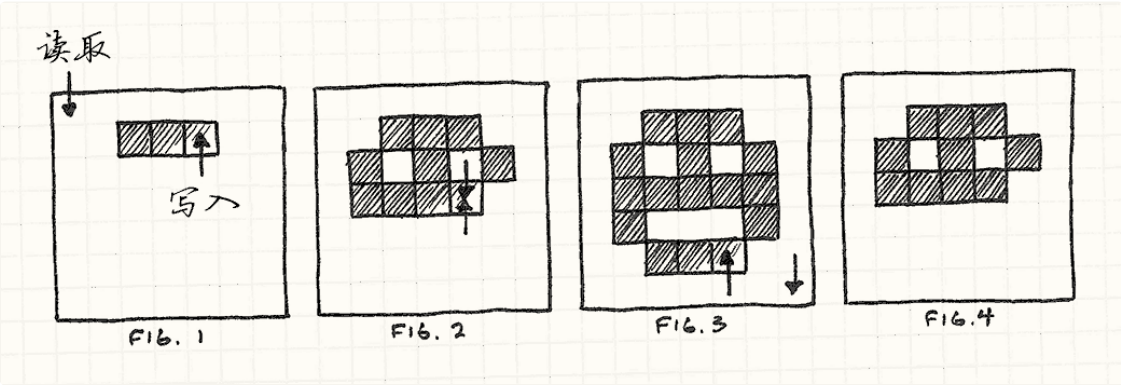
值，每次一个字节。

> 在字节值和颜色之间的映射通常由系统的像素格式和色深来指定。 在今日多数游戏主机上，每个像素都有32位，红绿蓝三个各占八位，剩下的八位保留作其他用途。

最终，为了让游戏显示在屏幕中，我们需要做的就是写入这个数组。 我们疯狂摆弄的图形算法最终都到了这里：设置帧缓冲中的字节值。 但这里有个小问题。

早先，我说过计算机是顺序处理的。 如果机器在运行一块渲染代码，我们不指望它同时还能做些别的什么事。 这通常是没啥问题，但是有些事确实在程序运行时发生。 其中一件是，当游戏运行时，视频输出正在不断从帧缓冲中读取数据。 这可能会为我们带来问题。

假设我们要在屏幕上显示一张笑脸。 程序在帧缓冲上开始循环，为像素点涂色。 我们没有意识到的是，在写入的同时，视频驱动正在读取它。 当它扫描过已写的像素时，笑脸开始浮现，但是之后它进入了未写的部分，就将没有写的像素绘制到了屏幕上。结果就是撕裂，你在屏幕上看到了绘制到一半的图像，这是可见的可怕漏洞。



> 显卡设备读取的缓冲帧正是我们绘制像素的那块(Fig. 1)。 显卡最终追上了渲染器，然后越过它，读取了还没有写入的像素(Fig. 2)。 我们完成了绘制，但驱动没有收到那些新像素。
>
> 结果(Fig. 4)是用户只看到了一半的绘制结果。 我称它为“哭脸”，笑脸看上去下半部是撕裂的。

这就是我们需要这个设计模式的原因。 程序一次渲染一个像素，但是显示需要一次全部看到——在这帧中啥也没有，下一帧笑脸全部出现。 双缓冲解决了这个问题。我会用类比来解释。

表演1，场景1

想象玩家正在观看我们的表演。 在场景一结束而场景二开始时，我们需要改变舞台设置。如果让场务在场景结束后进去拖动东西，在场景的连贯性就被打破了。 我们可以减弱灯光（这是剧院真正所做的），但是观众还是知道有什么在进行，而我们想在场景间毫无跳跃的转换。

消耗一些地皮，我们想到了一个聪明的解决方案：建两个舞台，观众两个都能看到。 每个有它自己的一组灯光。我们称这些舞台为舞台A和舞台B。 场景一在舞台A上。同时场务在处于黑暗之中的舞台B布置场景二。 当场景一完成后，切断场景A的灯光，将打开场景B的灯光。观众看向新舞台，场景二立即开始。

同时，场务到了黑咕隆咚的舞台**A**，收拾了场景一然后布置场景三。 一旦场景二结束，将灯

光转回舞台A。 我们在整场表演中进行这样的活动，使用黑暗的舞台作为布置下一场景的工作区域。 每一次场景转换，只是在两个舞台间切换灯光。 观众获得了连续的体验，场景转换时没有感到任何中断。他们从来没有见到场务。

> 使用单面镜以及其他的巧妙布置，你可以真正的在同一位置布置两个舞台。 随着灯光切换，观众看到了不同的舞台，无需看向不同的地方。 如何这样布置舞台就留给读者做练习吧。

重新回到图形

这就是双缓冲的工作原理， 这就是你看到的几乎每个游戏背后的渲染系统。 不只用一个帧缓冲，我们用两个。其中一个代表现在的帧，即类比中的舞台A，也就是说是显卡读取的那一个。 GPU可以想什么时候扫就什么时候扫。

> 但不是所有的游戏主机都是这么做的。 更老的简单主机中，内存有限，需要小心的同步绘制和渲染。那很需要技巧。

同时，我们的渲染代码正在写入另一个帧缓冲。 即黑暗中的舞台B。当渲染代码完成了场景的绘制，它将通过交换缓存来切换灯光。 这告诉图形硬件开始从第二块缓存中读取而不是第一块。 只要在刷新之前交换，就不会有任何撕裂出现，整个场景都会一下子出现。

这时可以使用以前的帧缓冲了。我们可以将下一帧渲染在它上面了。超棒！

模式

定义缓冲类封装了缓冲：一段可改变的状态。 这个缓冲被增量的修改，但我们想要外部的代码将修改视为单一的原子操作。 为了实现这点，类保存了两个缓冲的实例：下一缓冲和当前缓冲。

当信息从缓冲区中读取，它总是读取当前的缓冲区。 当信息需要写到缓存，它总是在下一缓冲区上操作。 当改变完成后，一个交换操作会立刻将当前缓冲区和下一缓冲区交换， 这样新缓冲区就是公共可见的了。旧的缓冲区成为下一个重用的缓冲区。

何时使用

这是那种你需要它时自然会想起的模式。 如果你有一个系统需要双缓冲，它可能有可见的错误（撕裂之类的）或者行为不正确。 但是，“当你需要时自然会想起”没提提供太多有效信息。 更加特殊的，以下情况都满足时，使用这个模式就很恰当：

- 我们需要维护一些被增量修改的状态。
- 在修改到一半的时候，状态可能会被外部请求。
- 我们想要防止请求状态的外部代码知道内部的工作方式。
- 我们想要读取状态，而且不想等着修改完成。

记住

不像其他较大的架构模式，双缓冲模式位于底层。 正因如此，它对代码库的其他部分影响

较小——大多数游戏甚至不会感到有区别。 尽管这里还是有几个警告。

交换本身需要时间

在状态被修改后，双缓冲需要一个**swap**步骤。 这个操作必须是原子的——在交换时，没有代码可以接触到任何一个状态。 通常，这就是修改一个指针那么快，但是如果交换消耗的时间长于修改状态的时间，那可是毫无助益。

我们得保存两个缓冲区

这个模式的另一个结果是增加了内存的使用。 正如其名，这个模式需要你在内存中一直保留两个状态的拷贝。 在内存受限的设备上，你可能要付出惨痛的代价。 如果你不能接受使用两份内存，你需要使用别的方法保证状态在修改时不会被请求。

示例代码

我们知道了理论，现在看看如何它在实践中如何应用。 我们编写了一个非常基础的图形系统，允许我们在缓冲帧上描绘像素。 在大多数主机和电脑上，显卡驱动提供了这种底层的图形系统， 但是在这里手动实现有助于理解发生了什么。首先是缓冲区本身：

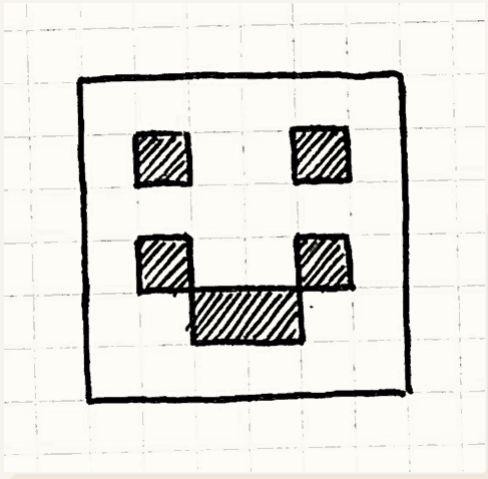
```
class Framebuffer
{
public:
  Framebuffer() { clear(); }

  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];
};
```

它有将整个缓存设置成默认的颜色的操作，也将其一像素设置为特定颜色的操作。 它也有函数`getPixels()`，读取保存像素数据的数组。 虽然在这个例子没有出现，但在实际中，显卡驱动会频繁调用这个函数，将缓存中的数据输送到屏幕上。

我们将整个缓冲区封装在`Scene`类中。渲染某物需要做的是在这块缓冲区上调用一系列`draw()`。

```
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_;
};
```

特别的，它画出来这幅旷世杰作：



每一帧，游戏告诉场景去绘制。场景清空缓冲区然后一个接一个绘制一大堆像素。 它也提供了`getBuffer()`获得缓冲区，这样显卡可以接触到它。

这看起来直截了当，但是如果就这样做，我们会遇到麻烦。 显卡驱动可以在任何时间调用`getBuffer()`，甚至在这个时候：

```
buffer_.draw(1, 1);
buffer_.draw(4, 1);
```

```
// <- 图形驱动从这里读取像素！
buffer_.draw(1, 3);
buffer_.draw(2, 4);
buffer_.draw(3, 4);
buffer_.draw(4, 3);
```

当上面的情况发生时，用户就会看到脸的眼睛，但是这一帧中嘴却消失了。 下一帧，又可能在某些别的地方发生冲突。最终结果是糟糕的闪烁图形。我们会用双缓冲修复这点：

```
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()
  {
    // 只需交换指针
    Framebuffer* temp = current_;
    current_ = next_;
    next_ = temp;
  }

  Framebuffer  buffers_[2];
  Framebuffer* current_;
  Framebuffer* next_;
};
```

现在Scene有存储在buffers_数组中的两个缓冲区，。 我们并不从数组中直接引用它们。而是通过两个成员，next_和current_，指向这个数组。 当绘制时，我们绘制在next_指向的缓冲区上。 当显卡驱动需要获得像素信息时，它总是通过current_获取另一个缓冲区。

通过这种方式，显卡驱动永远看不到我们正在施工的缓冲区。 解决方案的的最后一片碎片就是在场景完成绘制一帧的时候调用swap()。 它通过交换next_和current_的引用完成

这一点。 下一次显卡驱动调用getBuffer()，它会获得我们刚刚完成渲染的新缓冲区， 然后将刚刚描绘好的缓冲区放在屏幕上。没有撕裂，也没有不美观的问题。

不仅是图形

双缓冲解决的核心问题是状态有可能在被修改的同时被请求。 这通常有两种原因。图形的例子覆盖了第一种原因——另一线程的代码或者另一个中断的代码直接访问了状态。

但是，还有一个同样常见的原因：负责修改的 代码试图访问同样正在修改状态。 这可能发生在很多地方，特别是实体的物理部分和AI部分，实体在相互交互。 双缓冲在那里也十分有用。

人工不智能

假设我们正在构建一个关于趣味喜剧的游戏的行为系统。 这个游戏包括一堆跑来跑去寻欢作乐的角色。这里是我们的基础角色：

```
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_;
};
```

每一帧，游戏要在角色身上调用update()，让角色做些事情。 特别的，从玩家的角度，所有的角色都看上去应该同时更新。

> 这是更新方法模式的例子。

角色也可以相互交互，这里的“交互”，我指“可以互相扇对方巴掌”。 当更新时，角色可以在另一个角色身上调用slap()来扇它一巴掌，然后调用wasSlapped()看看自己是不是被扇了。

角色需要一个可以交互的舞台，让我们来布置一下：

```
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];
};
```

Stage允许我们向其中增加角色， 然后使用简单的update()调用来更新每个角色。 在用户看来，角色是同时移动的，但是实际上，它们是依次更新的。

这里需要注意的另一点是，每个角色的“被扇”状态在更新后就立刻被清除。 这样才能保证一个角色对一巴掌只反应一次。

作为一切的开始，让我们定义一个具体的角色子类。 这里的喜剧演员很简单。 他只面向一个角色。当他被扇时——无论是谁扇的他——他的反应是扇他面前的人一巴掌。

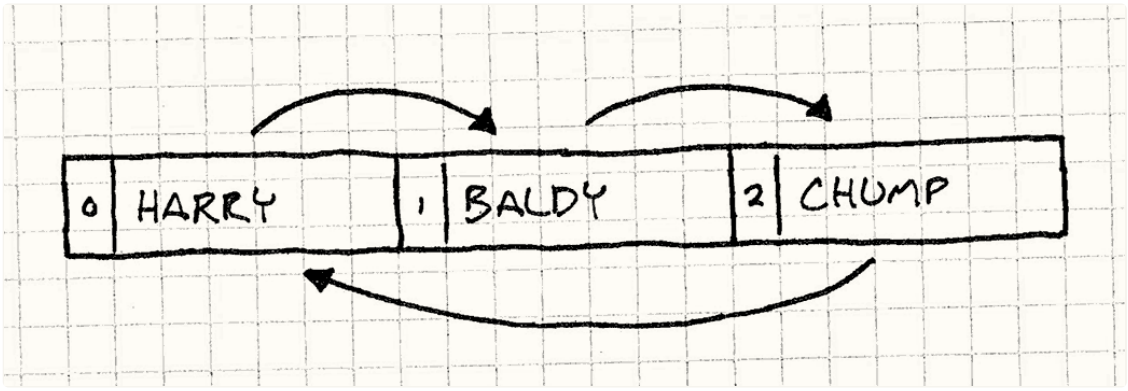
```
class Comedian : public Actor
{
public:
  void face(Actor* actor) { facing_ = actor; }

  virtual void update()
  {
    if (wasSlapped()) facing_->slap();
  }

private:
  Actor* facing_;
};
```

现在我们把一些喜剧演员丢到舞台上看看发生了什么。 我们设置三个演员，第一个面朝第二个，第二个面朝第三个，第三个面对第一个，形成一个环：

```
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);
```

最终舞台布置如下图。箭头代表角色的朝向，然后数字代表角色在舞台数组中的索引。



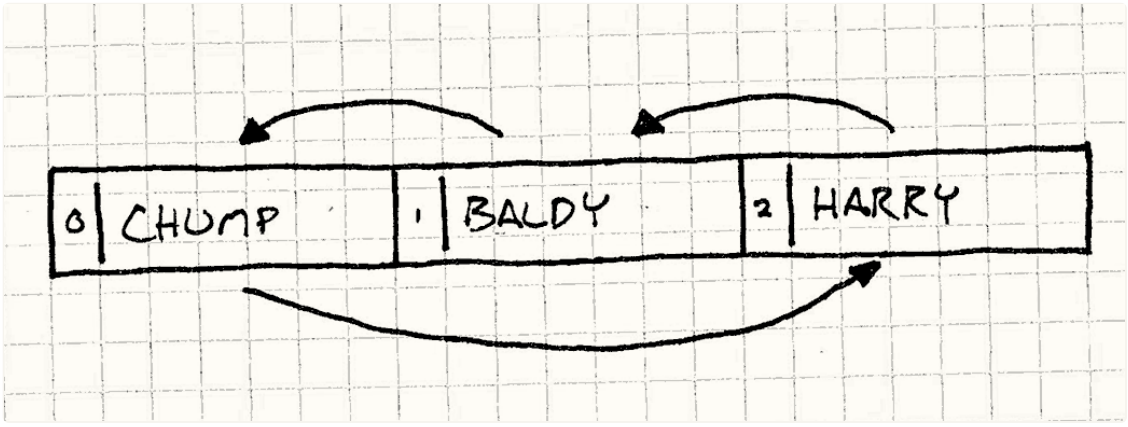
我们扇哈利一巴掌，为表演拉开序幕，看看之后会发生什么：

```
harry->slap();

stage.update();
```

记住Stage中的update()函数轮流更新每个角色， 因此如果检视整个代码，我们会发现事件这样发生：

```
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
```

在单独的一帧中，初始给哈利的一巴掌传给了所有的喜剧演员。 现在，让事物复杂起来，让我们重新排列舞台数组中角色的排序， 但是继续保持面向对方的方式。



我们不动舞台的其余部分，只是将添加角色到舞台的代码块改为如下：

```
stage.add(harry, 2);
```

```
stage.add(baldy, 1);
stage.add(chump, 0);
```

让我们看看再次运行时会发生什么：

```
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
```

哦不。完全不一样了。问题很明显。 更新角色时，我们修改了他们的“被扇”状态，这也是我们在更新时读取的状态。 因此，在更新中早先的状态修改会影响之后同一状态的修改的步骤。

> 如果你继续更新舞台，你会看到巴掌在角色间逐渐传递，每帧传递一个。 在第一帧 Harry扇了Baldy。下一帧，Baldy扇了Chump，如此类推。

而最终的结果是，一个角色对被扇作出反应可能是在被扇的同一帧或者下一帧， 这完全取决于两个角色在舞台上是如何排序的。 这没能满足我角色同时反应的需求——它们在同一帧中更新的顺序不该对结果有影响。

缓存巴掌

幸运的是，双缓冲模式可以帮忙。 这次，不是保存两大块“缓冲”，我们缓冲更小粒度的事物：每个角色的“被扇”状态。

```
class Actor
{
public:
  Actor() : currentSlapped_(false) {}

  virtual ~Actor() {}
  virtual void update() = 0;

  void swap()
  {
    // 交换缓冲区
    currentSlapped_ = nextSlapped_;

    // 清空新的“下一个”缓冲区。.
    nextSlapped_ = false;
  }

  void slap()       { nextSlapped_ = true; }
  bool wasSlapped() { return currentSlapped_; }

private:
  bool currentSlapped_;
```

```
  bool nextSlapped_;
};
```

不再使用一个`slapped_`状态，每个演员现在使用两个。 就像我们之前图形的例子一样，当前状态为读准备，下一状态为写准备。

`reset()`函数被替换为`swap()`。 现在，就在清除交换状态前，它将下一状态拷贝到当前状态上， 使其成为新的当前状态，这还需要在`Stage`中进行小小的改变：

```
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();
  }
}
```

`update()`函数现在更新所有的角色，然后 交换它们的状态。 最终结果是，角色在实际被扇之后的那帧才能看到巴掌。 这样一来，角色无论在舞台数组中如何排列，都会保持相同的行为。 无论外部的代码如何调用，所有的角色在一帧内同时更新。

设计决策

双缓冲很直观，我们上面看到的例子也覆盖了大多数你需要的场景。 使用这个模式之前，还有需要做两个主要的设计决策。

缓冲区是如何被交换的？

交换操作是整个过程的最重要的一步， 因为在其发生时，我们必须锁住两个缓冲区上的读取和修改。 为了让性能最优，我们需要它进行得越快越好。

- 交换缓冲区的指针或者引用： 这是我们图形例子中的做法，这也是大多数双缓冲图形通用的解决方法。
  - 速度快。 不管缓冲区有多大，交换都只需赋值一对指针。很难在速度和简易性上超越它。
  - 外部代码不能存储对缓存的永久指针。 这是主要限制。 由于我们没有真正地移动数据，本质上做的是周期性地通知代码库的其他部分去别处去寻找缓存， 就像前面的舞台类比一样。这就意味着代码库的其他部分不能存储指向缓冲区中数据的指针——它一段时间后可能就指向了错误的部分。

  这会严重误导那些期待缓冲帧永远在内存中的固定地址的显卡驱动。在这种情况下，我们不能这么做。
  - 缓冲区中的数据是两帧之前的数据，而不是上一帧的数据。 接下来的那帧绘制在帧

缓冲区上，而不是在它们之间拷贝数据，就像这样：

```
Frame 1 drawn on buffer A
Frame 2 drawn on buffer B
Frame 3 drawn on buffer A
...
```

你会注意到，当我们绘制第三帧时，缓冲区上的数据是第一帧的，而不是第二帧的。大多数情况下，这不是什么问题——我们通常在绘制之前清空整个帧。但如果想沿用某些缓存中已有的数据，就需要考虑数据其实比期望的更旧。

> 旧帧中缓存数据的经典用法是模拟动态模糊。 当前的帧混合一点之前的帧，看起来更像真实的相机捕获的图景。

- 在缓冲区之间拷贝数据： 如果我们不能重定向到其他缓存，唯一的选项就是将下帧的数据实实在在的拷贝到现在这帧上。 这是我们的扇巴掌喜剧的工作方法。 这种情况下，使用这种方法因为拷贝状态——一个简单的布尔标识——不比修改指向缓存的指针开销大。
  - 下一帧的数据和之前的数据相差一帧。 拷贝数据与在两块缓冲区间跳来跳去正相反。 如果我们需要前一帧的数据，这样我们可以处理更新的数据。
  - 交换也许更花时间。 这个，当然，是最大的缺点。交换操作现在意味着在内存中拷贝整个缓冲区。 如果缓冲区很大，比如一整个缓冲帧，这需要花费可观的时间。 由于交换时没有东西可以读取或者写入任何一个缓冲区，这是一个巨大的限制。

缓冲的粒度如何？

这里另一问题是缓冲区本身是如何组织的——是单个数据块还是散布在对象集合中？ 图形例子是前一种，而角色例子是后一种。

大多数情况下，你缓存的方式自然而然会引导你找到答案，但是这里也有些灵活度。 比如，角色总能将消息存在独立的消息块中，使用索引来引用。

- 如果缓存是一整块：
  - 交换操作更简单。 由于只有一对缓存，一个简单的交换就完成了。 如果可以改变指针来交换，那么不必在意缓冲区大小，只需几部操作就可以交换整个缓冲区。
- 如果很多对象都持有一块数据：
  - 交换操作更慢。 为了交换，需要遍历整个对象集合，通知每个对象交换。

    在喜剧的例子中，这没问题，因为反正需要清除被扇状态 ——每块缓存的数据每帧都需要接触。 如果不需要接触较旧的帧，可以用一个优化在多个对象间分散状态，获得使用整块缓存一样的性能。

    思路是将“当前”和“下一”指针概念，将它们改为对象相关的偏移量。就像这样：

```
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];
};
```

角色使用current_在状态数组中查询，获得当前的被扇状态， 下一状态总是数组中的另一索引，这样可以用next()来计算。 交换状态只需改动current_索引。聪明之处在于swap()现在是静态函数，它只需被调用一次，每个 角色的状态都会被交换。

参见

- 你可以在几乎每个图形API中找到双缓冲模式。举个例子，OpenGL有swapBuffers()，Direct3D有”swap chains”, Microsoft的XNA框架有endDraw()方法。

游戏循环

游戏设计模式 / Sequencing Patterns

意图

将游戏的进行和玩家输入解耦，和处理器速度解耦。

动机

如果本书中有一个不可或缺模式，那非这个模式莫属了。 游戏循环是“游戏设计模式”的精髓。 几乎每个游戏都有，两两不同，而在非游戏的程序几乎没有使用。

为了看看它多有用，让我们快速缅怀一遍往事。 每个编写计算机程序的人都留着胡子的时代，程序像洗碗机一样工作。 你输入一堆代码，按个按钮，等待，然后获得结果，完成。 程序全都是批处理模式的——一旦工作完成，程序就停止了。

> Ada Lovelace和Rear Admiral Grace Hopper是女程序员，并没有胡子。

你在今日仍然能看到这些程序，虽然感谢上天，我们不必在打孔纸上面编写它们了。 终端脚本，命令行程序，甚至将Markdown翻译成这本书的Python脚本都是批处理程序。

采访CPU

最终，程序员意识到将批处理代码留在计算办公室，等几个小时后拿到结果才能开始找程序漏洞的方式实在低效。 他们想要立即的反馈。交互式 程序诞生了。 第一批交互式程序中的就有游戏：

```
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.
```

> 这是Colossal Cave Adventure，史上首个冒险游戏。

你可以和这个程序进行实时交互。 它等待你的输入，然后进行响应。 你再输入，这样一唱一和，就像相声一样。 当轮到你时，它停在那里啥也不做。像这样：

```
while (true)
{
  char* command = readCommand();
  handleCommand(command);
}
```

这程序会永久循环，所以没法退出游戏。 真实的游戏会做些`while (!done)`进行检查，然后通过设置`done`为真来退出游戏。 我省去了那些，保持简明。

事件循环

如果你剥开现代的图形UI的外皮，会惊讶地发现它们与老旧的冒险游戏差不多。 文本处理器通常呆在那里什么也不做，直到你按了个键或者点了什么东西：

```
while (true)
{
  Event* event = waitForEvent();
  dispatchEvent(event);
}
```

这与冒险游戏主要的不同是，程序不是等待文本指令，而是等待用户输入事件——鼠标点击、按键按下之类的。 其他部分还是和以前的老式文本冒险游戏一样，程序阻塞等待用户的输入，这是个问题。

不像其他大多数软件，游戏即使没有玩家输入时也继续运行。 如果你站在那里看着屏幕，游戏不会冻结。动画继续动着。视觉效果继续闪烁。 如果不幸的话，怪物继续吞噬你的英雄。

件循环有“空转”事件，这样你可以无需用户输入间歇地做些事情。 对于闪烁的光标或者进度条已经足够了，但对于游戏就太原始了。

这是真实游戏循环的第一个关键部分：它处理用户输入，但是不等待它。循环总是继续旋转：

```
while (true)
{
  processInput();
  update();
  render();
}
```

我们之后会改善它，但是基本的部分都在这里了。 `processInput()`处理上次调用到现在的任何输入。 然后`update()`让游戏模拟一步。 运行AI和物理（通常是这种顺序）。 最终，`render()`绘制游戏，这样玩家可以看到发生了什么。

就像你可以从名字中猜到的， `update()`是使用更新方法模式的好地方。

时间之外的世界

如果这个循环没有因为输入而阻塞，这就带来了明显的问题，要运转多快呢？ 每次进行游戏循环都会推动一定的游戏状态的发展。 在游戏世界的居民看来，他们手上的表就会滴答一下。

> 运行游戏循环一次的常用术语就是“滴答”(tick)和“帧”(frame)。

同时，玩家的 真实手表也在滴答着。 如果我们用实际时间来测算游戏循环运行的速度，就得到了游戏的“帧率”(FPS)。 如果游戏循环的更快，FPS就更高，游戏运行的更流畅更快。如果循环得过慢，游戏看上去就像是慢动作电影。

我们现在写的这个循环是能转多快转多快，两个因素决定了帧率。 一个是每帧要做多少工作。复杂的物理，众多游戏对象，图形细节都让CPU和GPU繁忙，这决定了需要多久能完成一帧。

另一个是底层平台的速度。 更快的芯片可以在同样的时间里执行更多的代码。 多核，GPU组，独立声卡，以及系统的调度都影响了在一次滴答中能够做多少东西。

每秒的帧数

在早期的视频游戏中，第二个因素是固定的。 如果你为NES或者Apple IIe写游戏，你明确知道游戏运行在什么CPU上。 你可以（也必须）为它特制代码。 你只需担忧第一个因素：每次滴答要做多少工作。

早期的游戏被仔细地编码，一帧只做一定的工作，开发者可以让游戏以想要的速率运行。但是如果你想要在快些或者慢些的机器上运行同一游戏，游戏本身就会加速或减速。

> 这就是为什么老式计算机通常有“turbo”按钮。 新的计算机运行得太快了，无法玩老游戏，因为游戏也会运行的过快。 关闭 turbo按钮，会减慢计算机的运行速度，就可以运行老游戏了。

现在，很少有开发者可以奢侈地知道游戏运行的硬件条件。游戏必须自动适应多种设备。

这就是游戏循环的另一个关键任务：不管潜在的硬件条件，以固定速度运行游戏。

模式

一个游戏循环在游玩中不断运行。 每一次循环，它无阻塞地处理玩家输入，更新游戏状态，渲染游戏。 它追踪时间的消耗并控制游戏的速度。

何时使用

使用错误的模式比不使用模式更糟，所以这节通常告诫你不要过于热衷设计模式。 设计模式的目标不是往代码库里尽可能的塞东西。

但是这个模式有所不同。我可以很自信的说你会使用这个模式。 如果你使用游戏引擎，你不需要自己编写，但是它还在那里。

> 对于我，这是“引擎”与“库”的不同之处。 使用库时，你拥有游戏循环，调用库代码。 使用引擎时，引擎拥有游戏循环，调用你的代码。

你可能认为在做回合制游戏时不需要它。 但是哪怕是那里，就算游戏状态到玩家回合才改变，视觉和听觉 状态仍会改变。 哪怕游戏在“等待”你进行你的回合，动画和音乐也会继续运行。

记住

我们这里谈到的循环是游戏代码中最重要的部分。 有人说程序会花费90%的时间在10%的代码上。 游戏循环代码肯定在这10%中。 你必须小心谨慎，时时注意效率。

> “真正的”工程师，比如机械或电子工程师，不把我们当回事，大概就是因为我们像这样使用统计学是。

你也许需要与平台的事件循环相协调

如果你在操作系统的顶层或者有图形UI和内建事件循环的平台上构建游戏， 那你就有了两个应用循环在同时运作。 它们需要很好的协调。

有时候，你可以进行控制，只运行你的游戏循环。 举个例子，如果舍弃了Windows的珍贵API，`main()`可以只用游戏循环。 其中你可以调用`PeekMessage()`来处理和分发系统的事件。 不像`GetMessage()`，`PeekMessage()`不会阻塞等待用户输入， 因此你的游戏循环会保持运作。

其他的平台不会让你这么轻松地摆脱事件循环。 如果你使用网页浏览器作为平台，事件循环已被内建在浏览器的执行模型深处。 这样，你得用事件循环作为游戏循环。 你会调用`requestAnimationFrame()`之类的函数，它会回调你的代码，保持游戏继续运行。

示例代码

在如此长的介绍之后，游戏循环的代码实际上很直观。 我们会浏览一堆变种，比较它们的好处和坏处。

游戏循环驱动了AI，渲染和其他游戏系统，但这些不是模式的要点， 所以我们会调用虚构的方法。在实现了`render()`，`update()`之后， 剩下的作为给读者练习（挑战！）。

跑，能跑多快跑多快

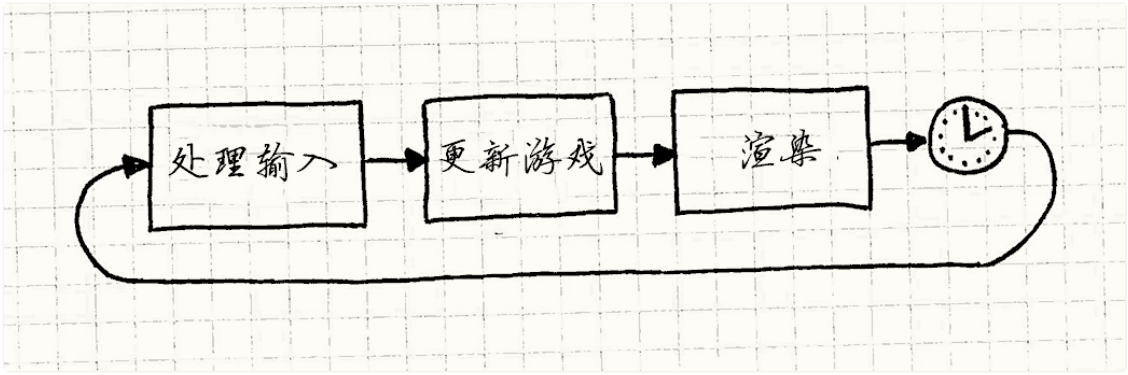
我们已经见过了可能是最简单的游戏循环：

```
while (true)
{
  processInput();
  update();
  render();
}
```

它的问题是你不能控制游戏运行的有多快。 在快速机器上，循环会运行的太快，玩家看不清发生了什么。 在慢速机器上，游戏慢的跟在爬一样。 如果游戏的一部分有大量内容或者做了很多AI或物理运算，游戏就会慢一些。

休息一下

我们看看增加一个简单的小修复如何。 假设你想要你的游戏以60FPS运行。这样每帧大约16毫秒。 只要你用少于这个时长进行游戏所有的处理和渲染，就可以以稳定的帧率运行。 你需要做的就是处理这一帧然后等待，直到处理下一帧的时候，就像这样：



代码看上去像这样：

```
1000 毫秒 / 帧率 = 毫秒每帧.
```

```
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();

  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

如果它很快的处理完一帧，这里的`sleep()`保证了游戏不会运行太快。 如果你的游戏运行太慢，这无济于事。 如果需要超过16ms来更新并渲染一帧，休眠的时间就变成了负的。 如果计算机能回退时间，很多事情就很容易了，但是它不能。

相反，游戏变慢了。 可以通过每帧少做些工作来解决这个问题——减少物理效果和绚丽光影，或者把AI变笨。 但是这影响了那些有快速机器的玩家的游玩体验。

一小步，一大步

让我们尝试一些更加复杂的东西。我们拥有的问题基本上是：

1. 每次更新将游戏时间推动一个固定量。
2. 这消耗一定量的真实时间来处理它。

如果第二步消耗的时间超过第一步，游戏就变慢了。 如果它需要超过16ms来推动游戏时间16ms，那它永远也跟不上。 但是如果一步中推动游戏时间超过16ms，那我们可以减少更新频率，就可以跟得上了。

接着的思路是基于上帧到现在有多少真实时间流逝来选择前进的时间。 这一帧花费的时间越长，游戏的间隔越大。 它总能跟上真实时间，因为它走的步子越来越大。 有人称之为变化的或者流动的时间间隔。它看上去像是：

```
double lastTime = getCurrentTime();
```

```
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - lastTime;
  processInput();
  update(elapsed);
  render();
  lastTime = current;
}
```

每一帧，我们计算上次游戏更新到现在有多少真实时间过去了（即变量`elapsed`）。 当我们更新游戏状态时将其传入。 然后游戏引擎让游戏世界推进一定的时间量。

假设有一颗子弹跨过屏幕。 使用固定的时间间隔，在每一帧中，你根据它的速度移动它。 使用变化的时间间隔，你根据过去的时间拉伸速度。 随着时间间隔增加，子弹在每帧间移动的更远。 无论是二十个快的小间隔还是四个慢的大间隔，子弹在真实时间里移动同样多的距离。 这看上去成功了：

- 游戏在不同的硬件上以固定的速度运行。
- 使用高端机器的玩家获得了更流畅的游戏体验。

但是，悲剧，这里有一个严重的问题： 游戏不再是确定的了，也不再稳定。 这是我们给自己挖的一个坑：

> “确定的”代表每次你运行程序，如果给了它同样的输入，就获得同样的输出。 可以想得到，在确定的程序中追踪漏洞更容易——一旦找到造成漏洞的输入，每次你都能重现之。
>
> 计算机本身是确定的；它们机械地执行程序。 在纷乱的真实世界加入时，非确定性出现了。 例如，网络，系统时钟，和线程调度都依赖于超出程序控制的外部世界。

假设我们有个双人联网游戏，Fred的游戏机是台性能猛兽，而George正在使用他祖母的老爷机。 前面提到的子弹在他们的屏幕上飞行。 在Fred的机器上，游戏跑的超级快，每个时间间隔都很小。 比如，我们塞了50帧在子弹穿过屏幕的那一秒。 可怜的George的机器只能塞进大约5帧。

这就意味着在Fred的机器上，物理引擎每秒更新50次位置，但是George的只更新5次。 大多数游戏使用浮点数，它们有舍入误差。 每次你将两个浮点数加在一起，获得的结果就会有点偏差。 Fred的机器做了10倍的操作，所以他的误差要比George的更大。 同样 的子弹最终在他们的机器上到了不同的位置。

这是使用变化时间可引起的问题之一，还有更多问题呢。 为了实时运行，游戏物理引擎做的是实际机制法则的近似。 为了避免飞天遁地，物理引擎添加了阻尼。 这个阻尼运算被小心地安排成以固定的时间间隔运行。 改变了它，物理就不再稳定。

> “飞天遁地”在这里使用的是它的字面意思。当物理引擎卡住，对象获得了完全错误的速度，就会飞到天上或者掉入地底。

这种不稳定性太糟了，这个例子在这里的唯一原因是作为警示寓言，引领我们到更好的东西……

追逐时间

游戏中渲染通常不会被动态时间间隔影响到。 由于渲染引擎表现的是时间上的一瞬间，它不会计算上次到现在过了多久。 它只是将当前事物渲染在所在的地方。

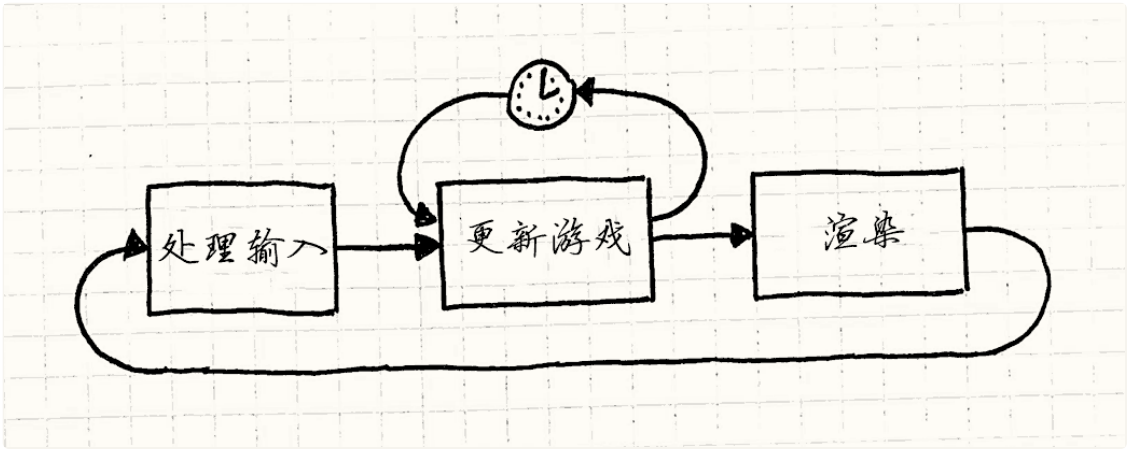
> 这或多或少是成立的。像动态模糊的东西会被时间间隔影响，但如果有一点延迟，玩家通常也不会注意到。

我们可以利用这点。 以固定的时间间隔更新游戏，因为这让所有事情变得简单，物理和AI也更加稳定。 但是我们允许灵活调整渲染的时刻，释放一些处理器时间。

它像这样运作：自上一次游戏循环过去了一定量的真实时间。 需要为游戏的“当前时间”模拟推进相同长度的时间，以追上玩家的时间。 我们使用一系列的固定时间步长。 代码大致如下：

```
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();
}
```

这里有几个部分。 在每帧的开始，根据过去了多少真实的时间，更新`lag`。 这变量表明了游戏世界时钟比真实世界落后了多少，然后我们使用一个固定时间步长的内部循环进行追赶。 一旦我们追上真实时间，我们就渲染然后开始新一轮循环。 你可以将其画成这样：



注意这里的时间步长不是视觉上的帧率了。 `MS_PER_UPDATE`只是我们更新游戏的间隔。

这个间隔越短，就需要越多的处理次数来追上真实时间。 它越长，游戏抖动的越厉害。 理想上，你想要它足够短，通常快过60FPS，这样游戏在高速机器上会有高效的表现。

但是小心不要把它整得太短了。 你需要保证即使在最慢的机器上，这个时间步长也超过处理一次`update()`的时间。 否则，你的游戏就跟不上现实时间了。

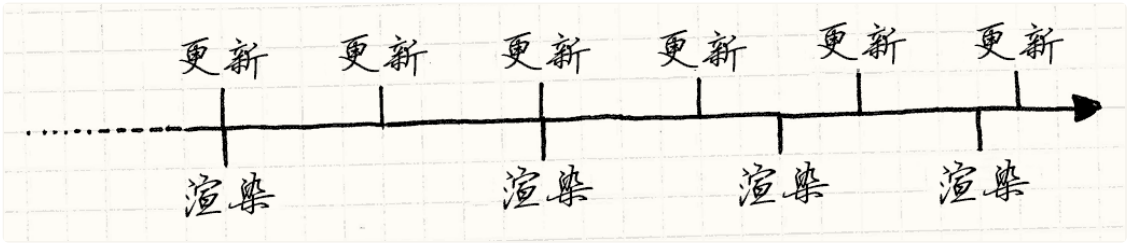
> 我不会详谈这个，但你可以通过限定内层循环最大次数来保证这一点。 游戏会变慢，但是比完全卡死要好。

幸运的是，我们给自己了一些喘息的空间。 技巧在于我们将渲染拉出了更新循环。 这释放了一大块CPU时间。 最终结果是游戏以固定时间步长模拟，该时间步长与硬件不相关。 只是使用低端硬件的玩家看到的会有抖动。

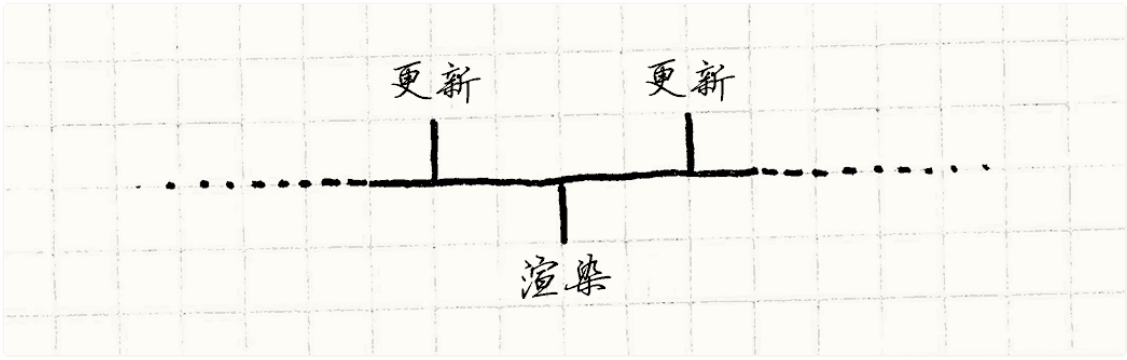
卡在中间

我们还剩一个问题，就是剩下的延迟。 以固定的时间步长更新游戏，在任意时刻渲染。 这就意味着从玩家的角度看，游戏经常在两次更新之间时显示。

这是时间线：



就像你看到的那样，我们以紧凑固定的时间步长进行更新。 同时，我们在任何可能的时候渲染。 它比更新发生的要少，而且也不稳定。 两者都没问题。糟糕的是，我们不总能在正确的时间点渲染。 看看第三次渲染时间。它发生在两次更新之间。



想象一颗子弹飞过屏幕。第一次更新时，它在左边。 第二次更新将它移到了右边。 这个游戏在两次更新之间的时间点渲染，所以玩家期望看到子弹在屏幕的中间。 而现在的实现中，它还在左边。这意味着看上去移动发生了卡顿。

方便的是，我们实际知道渲染时距离两次更新的时间：它被存储在`lag`中。 我们在`lag`比更新时间间隔小时，而不是`lag`是零时，跳出循环进行渲染。 `lag`的剩余量？那就是到下一帧的时间。 n we go to render, we'll pass that in:

当我们要渲染时，我们将它传入：

```
render(lag / MS_PER_UPDATE);
```

```
                MS_PER_UPDATE                                    re
```

我们在这里除以 来归一化值。 不管更新的时间步长是多少，传给nder()的值总在0（恰巧在前一帧）到1.0（恰巧在下一帧）之间。 这样，渲染引擎不必担心帧率。它只需处理0到1的值。

渲染器知道每个游戏对象以及它当前的速度。 假设子弹在屏幕左边200像素的地方，正在以400像素每帧的速度向右移动。 如果在两帧正中渲染，我们会给`render()`传0.5。 它绘制了半帧之前的图形，在200像素，啊哈，平滑的移动。

当然，也许这种推断是错误的。 在我们计算下一帧时，也许会发现子弹碰撞到另一障碍，或者减速，又或者别的什么。 我们只是在上一帧位置和我们认为的下一帧位置之间插值。但只有在完成物理和AI更新后，我们才能知道真正的位置。

所以推断有猜测的成分，有时候结果是错误的。 但是，幸运的，这种修正通常不可感知。最起码，比你不使用推断导致的卡顿更不明显。

设计决策

虽然这章我讲了很多，但是有更多的东西我没讲。 一旦你考虑显示刷新频率的同步，多线程，多GPU，真正的游戏循环会变得更加复杂。 即使在高层，这里还有一些问题你需要回答：

拥有游戏循环的是你，还是平台？

这个选择通常已经是平台决定的。 如果你在做浏览器中的游戏，很可能你不能编写自己的经典游戏循环。 浏览器本身的事件驱动机制阻碍了这一点。 类似的，如果你使用现存的游戏引擎，你很可能依赖于它的游戏循环而不是自己写一个。

- 使用平台的事件循环：
  - 简单。你不必担心编写和优化自己的游戏核心循环。
  - 平台友好。 你不必明确地给平台一段时间让它处理它自己的事件，不必缓存事件，不必管理任何平台输入模型和你的不匹配之处。
  - 你失去了对时间的控制。 平台会在它方便时调用代码。 如果这不如你想要的那样平滑或者频繁，太糟了。 更糟的是，大多数应用的事件循环并未为游戏设计，通常是又慢又卡顿。
- 使用游戏引擎的循环：
  - 不必自己编写。 编写游戏循环非常需要技巧。 由于是每帧都要执行的核心代码，小小的漏洞或者性能问题就对游戏有巨大的影响。 稳固的游戏循环是使用现有引擎的原因之一。
  - 不必自己编写。 当然，硬币的另一面是，如果引擎无法满足你真正的需求，你也没法获得控制权。
- 自己写：
  - 完全的控制。 你可以做任何想做的事情。你可以为游戏的需求订制开发。
  - 你需要与平台交互。 应用框架和操作系统通常需要时间片去处理自己的事件和其他工作。 如果你拥有应用的核心循环，平台就没有这些时间片了。 你得显式定期检查，保证框架没有挂起或者混乱。

如何管理能量消耗？

在五年前这还不是问题。 游戏运行在插到插座上的机器上或者专用的手持设备上。 但是随着智能手机，笔记本以及移动游戏的发展，现在需要关注这个问题了。 画面绚丽，但会耗干三十分钟前充的电，并将手机变成空间加热器的游戏，可不能让人开心。

现在，你需要考虑的不仅仅是让游戏看上去很棒，同时也要尽可能少的使用CPU。 你需要设置一个性能的上限：完成一帧之内所需的工作后，让CPU休眠。

- 尽可能快的运行：

  这是PC游戏的常态（即使越来越多的人在笔记本上运行游戏）。 游戏循环永远不会显式告诉系统休眠。相反，空闲的循环被划在提升FPS或者图像显示效果上了。

  这会给你最好的游戏体验。 但是，也会尽可能多的使用电量。如果玩家在笔记本电脑上游玩，他们就得到了一个很好的加热器。

- 固定帧率

  移动游戏更加注意游戏的体验质量，而不是最大化图像画质。 很多这种游戏都会设置最大帧率（通常是30或60FPS）。 如果游戏循环在分配的时间片消耗完之前完成，剩余的时间它会休眠。

  这给了玩家“足够好的”游戏体验，也让电池轻松了一点。

你如何控制游戏速度？

游戏循环有两个关键部分：不阻塞用户输入和自适应的帧时间步长。 输入部分很直观。关键在于你如何处理时间。 这里有数不尽的游戏可运行的平台， 每个游戏都需要在其中一些平台上运行。 如何适应平台变化就是关键。

> 创作游戏看来是人类的天性，因为每当我们建构可以计算的机器，首先做的就是在上面编游戏。 PDP-1是一个仅有4096字内存的2kHz机器，但是Steve Russell和他的朋友还是在上面创建了Spacewar!。

- 固定时间步长，没有同步：

  见我们第一个样例中的代码。你只需尽可能快地运行游戏。

  - 简单。这是主要的（好吧，唯一的）好处。
  - 游戏速度直接受到硬件和游戏复杂度影响。 主要的缺点是，如果有所变化，会直接影响游戏速度。游戏速度与游戏循环紧密相关。

- 固定时间步长，有同步：

  对复杂度控制的下一步是使用固定的时间间隔，但在循环的末尾增加同步点，保证游戏不会运行得过快。

  - 还是很简单。 这比过于简单不可行的例子只多了一行代码。 在多数游戏循环中，你可能总需要做一些同步。 你可能需要双缓冲图形并将缓冲块与更新显示的频率同步。
  - 电量友好。 这对移动游戏至关重要。你不想消耗不必要的电量。 通过简单的休眠几个毫秒而不是试图每帧塞入更多的处理，你节约了电量。
  - 游戏不会运行的太快。 这解决了固定循环速度的一半问题。

    - 游戏可能运行的太慢。 如果花了太多时间更新和渲染一帧，播放也会减缓。 因为这种方案没有分离更新和渲染，它比更高级的方案更容易遇到这点。 没法扔掉渲染帧来追上真实时间，游戏本身会变慢。

- 动态时间步长：

  我把这个方案放在这里作为问题的解决办法之一，附加警告：大多数我认识的游戏开发者反对它。 不过记住为什么反对它是很有价值的。

    - 能适应并调整，避免运行得太快或者太慢。 如果游戏不能追上真实时间，它用越来越长的时间步长更新，直到追上。
    - 让游戏不确定而且不稳定。 这是真正的问题，当然。在物理和网络部分使用动态时间步长会遇见更多的困难。

- 固定更新时间步长，动态渲染：

  在示例代码中提到的最后一个选项是最复杂的，但是也是最有适应性的。 它以固定时间步长更新，但是如果需要赶上玩家的时间，可以扔掉一些渲染帧。

    - 能适应并调整，避免运行得太快或者太慢。 只要能实时更新，游戏状态就不会落后于真实时间。如果玩家用高端机器，它会回以更平滑的游戏体验。
    - 更复杂。 主要负面问题是需要在实现中写更多东西。 你需要将更新的时间步长调整的尽可能小来适应高端机，同时不至于在低端机上太慢。

参见

- 关于游戏循环的经典文章是Glenn Fiedler的”Fix Your Timestep“。如果没有这篇文章，这章就不会是这个样子。
- Witters关于game loops的文章也值得阅读。
- Unity框架有一个复杂的游戏循环，细节在这里有详尽的解释。

更新方法

意图

通过每次处理一帧的行为模拟一系列独立对象。

动机

玩家操作强大的女武神完成考验：从死亡巫王的栖骨之处偷走华丽的珠宝。 她尝试接近巫王华丽的地宫门口，然后遇到了……啥也没遇到。 没有诅咒雕像向她发射闪电，没有不死战士巡逻入口。 她直捣黄龙，拿走了珠宝。游戏结束。你赢了。

好吧，这可不行。

地宫需要守卫——一些英雄可以杀死的敌人。 首先，我们需要一个骷髅战士在门口前后移动巡逻。 如果无视任何关于游戏编程的知识， 让骷髅蹒跚着来回移动的最简单的代码大概是这样的：

> 如果巫王想表现得更加智慧，它应创造一些仍有脑子的东西。

```
while (true)
{
  // 向右巡逻
  for (double x = 0; x < 100; x++)
  {
    skeleton.setX(x);
  }

  // 向左巡逻
  for (double x = 100; x > 0; x--)
  {
    skeleton.setX(x);
  }
}
```

这里的问题，当然，是骷髅来回打转，可玩家永远看不到。 程序锁死在一个无限循环，那

可不是有趣的游戏体验。 我们事实上想要的是骷髅每帧移动一步。

我们得移除这些循环，依赖外层游戏循环来迭代。 这保证了在卫士来回巡逻时，游戏能响应玩家的输入并进行渲染。如下：

> 当然，游戏循环 是本书的另一个章节。

```
Entity skeleton;
bool patrollingLeft = false;
double x = 0;

// 游戏主循环
while (true)
{
  if (patrollingLeft)
  {
    x--;
    if (x == 0) patrollingLeft = false;
  }
  else
  {
    x++;
    if (x == 100) patrollingLeft = true;
  }

  skeleton.setX(x);

  // 处理用户输入并渲染游戏……
}
```

在这里前后两个版本展示了代码是如何变得复杂的。 左右巡逻需要两个简单的`for`循环。通过指定哪个循环在执行，我们追踪了骷髅在移向哪个方向。 现在我们每帧跳出到外层的游戏循环，然后在跳回继续我们之前所做的，我们使用`patrollingLeft`显式地追踪了方向。

但或多或少这能行，所以我们继续。 一堆无脑的骨头不会对你的女武神提出太多挑战， 下一个我们添加的是魔法雕像。它们一直会向她发射闪电球，这样可让她保持移动。

继续我们的“用最简单的方式编码”的风格，我们得到了：

```
// 骷髅的变量……
Entity leftStatue;
Entity rightStatue;
int leftStatueFrames = 0;
int rightStatueFrames = 0;

// 游戏主循环：
while (true)
{
  // 骷髅的代码……
```

```
  if (++leftStatueFrames == 90)
  {
    leftStatueFrames = 0;
    leftStatue.shootLightning();
  }

  if (++rightStatueFrames == 80)
  {
    rightStatueFrames = 0;
    rightStatue.shootLightning();
  }

  // 处理用户输入，渲染游戏
}
```

你会发现这代码渐渐滑向失控。 变量数目不断增长，代码都在游戏循环中，每段代码处理一个特殊的游戏实体。 为了同时访问并运行它们，我们将它们的代码混杂在了一起。

> 一旦能用“混杂”一词描述你的架构，你就有麻烦了。

你也许已经猜到了修复这个所用的简单模式了： 每个游戏实体应该封装它自己的行为。 这保持了游戏循环的整洁，便于添加和移除实体。

为了做到这点需要抽象层，我们通过定义抽象的`update()`方法来完成。 游戏循环管理对象的集合，但是不知道对象的具体类型。 它只知道这些对象可以被更新。 这样，每个对象的行为与游戏循环分离，与其他对象分离。

每一帧，游戏循环遍历集合，在每个对象上调用`update()`。 这给了我们在每帧上更新一次行为的机会。 在所有对象上每帧调用它，对象就能同时行动。

> 死抠细节的人会在这点上揪着我不放，是的，它们没有真的同步。 当一个对象更新时，其他的都不在更新中。 我们等会再说这点。

游戏循环维护动态的对象集合，所以从关卡添加和移除对象是很容易的——只需要将它们从集合中添加和移除。 不必再用硬编码，我们甚至可以用数据文件构成这个关卡，那正是我们关卡设计者需要的。

模式

游戏世界管理对象集合。 每个对象实现一个更新方法模拟对象在一帧内的行为。每一帧，游戏循环更新集合中的每一个对象。

何时使用

如果游戏循环模式是切片面包， 那么更新方法模式就是它的奶油。 很多玩家交互的游戏实体都以这样或那样的方式实现了这个模式。 如果游戏有太空陆战队，火龙，火星人，鬼魂或者运动员，很有可能它使用了这个模式。

但是如果游戏更加抽象，移动部分不太像活动的角色而更加像棋盘上的棋子， 这个模式通常就不适用了。 在棋类游戏中，你不需要同时模拟所有的部分， 你可能也不需要告诉棋子

每帧都更新它们自己。

> 你也许不需要每帧更新它们的行为，但即使是棋类游戏， 你可能也需要每帧更新动画。 这个设计模式也可以帮到你。

更新方法适应以下情况：

- 你的游戏有很多对象或系统需要同时运行。
- 每个对象的行为都与其他的大部分独立。
- 对象需要跟着时间进行模拟。

记住

这个模式很简单，所以没有太多值得发现的惊喜。当然，每行代码还是有利有弊。

将代码划分到一帧帧中会让它更复杂

当你比较前面两块代码时，第二块看上去更加复杂。 两者都只是让骷髅守卫来回移动，但与此同时，第二块代码将控制权交给了游戏循环的一帧帧中。

几乎 这个改变是游戏循环处理用户输入，渲染等几乎必须要注意的事项，所以第一个例子不大实用。 但是很有必要记住，将你的行为切片会增加很高的复杂性。

> 我在这里说几乎是因为有时候鱼和熊掌可以兼得。 你可以直接为对象编码而不进行返回， 保持很多对象同时运行并与游戏循环保持协调。
>
> 你需要的是允许你同时拥有多个“线程”执行的系统。 如果对象的代码可以在执行中暂停和继续，而不是总得返回， 你可以用更加命令式的方式编码。
>
> 真实的线程太过重量级而不能这么做， 但如果你的语言支持轻量协同架构比如generators，coroutines或者fibers，那你也许可以使用它们。
>
> 字节码 模式是另一个在应用层创建多个线程执行的方法。

当离开每帧时，你需要存储状态，以备将来继续。

在第一个示例代码中，我们不需要用任何变量表明守卫在向左还是向右移动。 这显式的依赖于哪块代码正在运行。

当我们将其变为一次一帧的形式，我们需要创建`patrollingLeft`变量来追踪行走的方向。 当从代码中返回时，就丢失了行走的方向，所以为了下帧继续，我们需要显式存储足够的信息。

状态模式 通常可以在这里帮忙。 状态机在游戏中频繁出现的部分原因是（就像名字暗示的），它能在你离开时为你存储各种你需要的状态。

对象逐帧模拟，但并非真的同步

在这个模式中，游戏遍历对象集合，更新每一个对象。 在`update()`调用中，大多数对象都能够接触到游戏世界的其他部分， 包括现在正在更新的其他对象。这就意味着你更新对象的顺序至关重要。

如果对象更新列表中，A在B之前，当A更新时，它会看到B之前的状态。 但是当B更新时，由于A已经在这帧更新了，它会看见A的新状态。 哪怕按照玩家的视角，所有对象都是同时运转的，游戏的核心还是回合制的。 只是完整的“回合”只有一帧那么长。

> 如果，由于某些原因，你决定不让游戏像这样顺序更新，你需要双缓冲 模式。那么AB更新的顺序就没有关系了，因为双方都会看对方之前那帧的状态。

当关注游戏逻辑时，这通常是件好事。 同时更新所有对象将把你带到一些不愉快的语义角落。 想象如果国际象棋中，黑白双方同时移动会发生什么。 双方都试图同时往同一个空格子中放置棋子。这怎么解决？

序列更新解决了这点——每次更新都让游戏世界从一个合法状态增量更新到下一个，不会出现引发歧义而需要协调的部分。

> 这对在线游戏也有用，因为你有了可以在网上发送的行动指令序列。

在更新时修改对象列表需小心

当你使用这个模式时，很多游戏行为在更新方法中纠缠在一起。 这些行为通常包括增加和删除可更新对象。

举个例子，假设骷髅守卫被杀死时掉落物品。 使用新对象，你通常可以将其增加到列表尾部，而不引起任何问题。 你会继续遍历这张链表，最终找到新的那个，然后也更新了它。

但这确实表明新对象在它产生的那帧就有机会活动，甚至有可能在玩家看到它之前。 如果你不想发生那种情况，简单的修复方法就是在游戏循环中缓存列表对象的数目，然后只更新那么多数目的对象就停止：

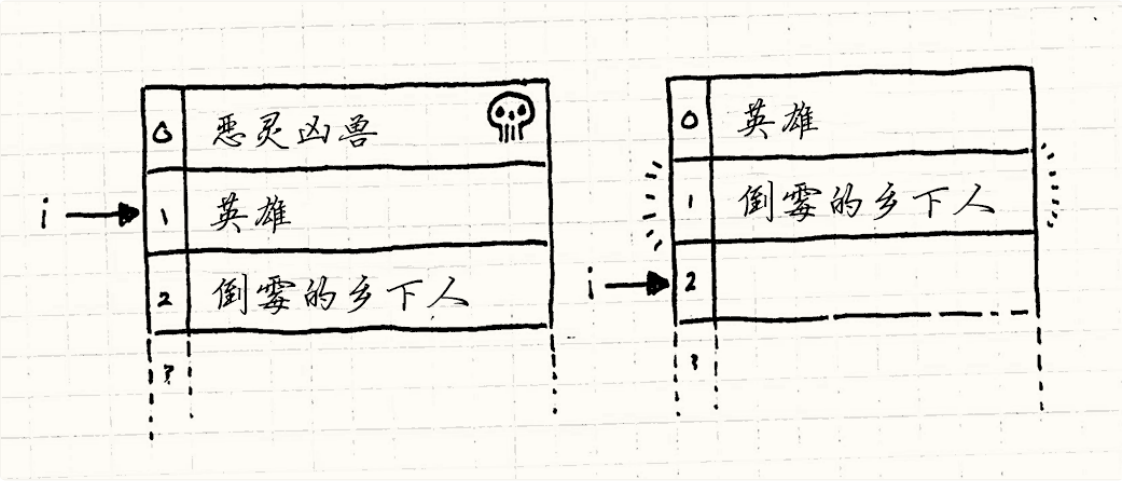
```
int numObjectsThisTurn = numObjects_;
for (int i = 0; i < numObjectsThisTurn; i++)
{
  objects_[i]->update();
}
```

这里，`objects_`是可更新游戏对象的数组，而`numObjects_`是数组的长度。 当添加新对象时，这个数组长度变量就增加。 在循环的一开始，我们在`numObjectsThisTurn`中存储数组的长度， 这样在这帧的遍历循环会停在新添加的对象之前。

一个更麻烦的问题是在遍历时移除对象。 你击败了邪恶的野兽，现在它需要被移出对象列表。 如果它正好位于你当前更新对象之前，你会意外地跳过一个对象：

```
for (int i = 0; i < numObjects_; i++)
{
  objects_[i]->update();
}
```

这个简单的循环通过增加索引值来遍历每个对象。 下图的左侧展示了在我们更新英雄时，数组看上去是什么样的：



我们在更新她时，索引值`i`是1。 邪恶野兽被她杀了，因此需要从数组移除。 英雄移到了位置0，倒霉的乡下人移到了位置1。 在更新英雄之后，`i`增加到了2。 就像你在右图看到的，倒霉的乡下人被跳过了，没有更新。

> 一种简单的解决方案是在更新时从后往前遍历列表。 这种方式只会移动已经被更新的对象。

一种解决方案是小心地移除对象，任何对象被移除时，更新索引。 另一种是在遍历完列表后再移除对象。 将对象标为“死亡”，但是把它放在那里。 在更新时跳过任何死亡的对象。然后，在完成遍历后，遍历列表并删除尸体。

> 如果在更新循环中有多个线程处理对象， 那么你可能更喜欢推迟任何修改，避免更新时同步线程的开销。

示例代码。

这个模式太直观了，代码几乎只是在重复说明要点。 这不意味着这个模式没有用。它因为简单而有用：这是一个无需装饰的干净解决方案。

但是为了让事情更具体些，让我们看看一个基础的实现。 我们会从代表骷髅和雕像的`Entity`类开始：

```
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_;
};
```

我在这里只呈现了我们后面所需东西的最小集合。 可以推断在真实代码中，会有很多图形和物理这样的其他东西。 上面这部分代码最重要的部分是它有抽象的`update()`方法。

游戏管理实体的集合。在我们的示例中，我会把它放在一个代表游戏世界的类中。

```
class World
{
public:
  World()
  : numEntities_(0)
  {}

  void gameLoop();

private:
  Entity* entities_[MAX_ENTITIES];
  int numEntities_;
};
```

> 在真实世界程序中，你可能真的要使用集合类，我在这里使用数组来保持简单

现在，万事俱备，游戏通过每帧更新每个实体来实现模式：

```
void World::gameLoop()
{
  while (true)
  {
    // 处理用户输入……

    // 更新每个实体
    for (int i = 0; i < numEntities_; i++)
    {
      entities_[i]->update();
    }

    // 物理和渲染……
  }
}
```

> 正如其名，这是游戏循环模式的一个例子。

子类化实体？！

有很多读者刚刚起了鸡皮疙瘩，因为我在Entity主类中使用继承来定义不同的行为。 如果你在这里还没有看出问题，我会提供一些线索。

当游戏业界从6502汇编代码和VBLANKs转向面向对象的语言时， 开发者陷入了对软件架构的狂热之中。 其中之一就是使用继承。他们建立了遮天蔽日的高耸的拜占庭式对象层次。

最终证明这是个糟点子，没人可以不拆解它们来管理庞杂的对象层次。 哪怕在1994年的GoF都知道这点，并写道：

> 多用“对象组合”，而非“类继承”。

> 只在你我间聊聊，我认为这已经是一朝被蛇咬十年怕井绳了。 我通常避免使用它，但教条地不用和教条地使用一样糟。 你可以适度使用，不必完全禁用。

当游戏业界都明白了这一点，解决方案是使用组件模式。 使用它，update()是实体的组件而不是在Entity中。 这让你避开了为了定义和重用行为而创建实体所需的复杂类继承层次。相反，你只需混合和组装组件。

如果我真正在做游戏，我也许也会那么做。 但是这章不是关于组件的， 而是关于update()方法，最简单，最少牵连其他部分的介绍方法， 就是把更新方法放在Entity中然后创建一些子类。

> 组件模式在这里。

定义实体

好了，回到任务中。 我们原先的动机是定义巡逻的骷髅守卫和释放闪电的魔法雕像。 让我们从我们的骷髅朋友开始吧。 为了定义它的巡逻行为，我们定义恰当地实现了update()的新实体：

```
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_;
};
```

如你所见，几乎就是从早先的游戏循环中剪切代码，然后粘贴到Skeleton的update()方法中。 唯一的小小不同是patrollingLeft_被定义为字段而不是本地变量。 通过这种方式，它的值在update()两次调用间保持不变。

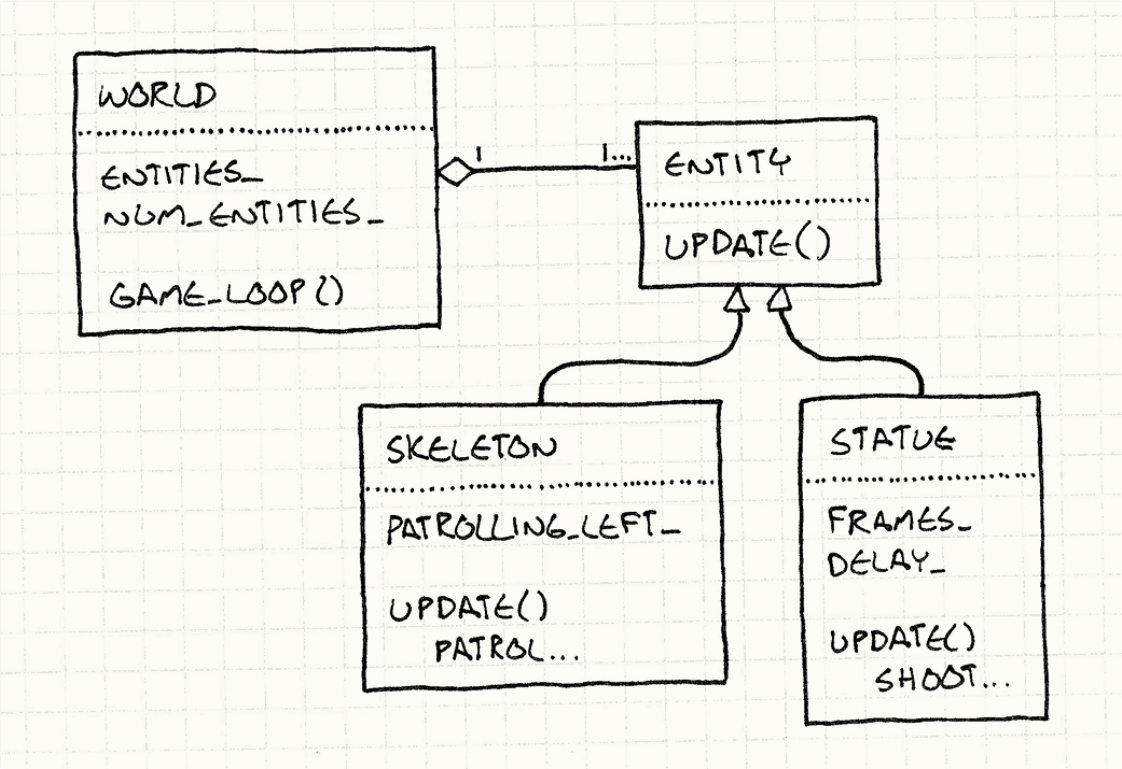
让我们对雕像如法炮制：

```
class Statue : public Entity
{
public:
  Statue(int delay)
  : frames_(0),
    delay_(delay)
  {}

  virtual void update()
  {
    if (++frames_ == delay_)
    {
      shootLightning();

      // 重置计时器
      frames_ = 0;
    }
  }

private:
  int frames_;
  int delay_;

  void shootLightning()
  {
    // 火光效果……
  }
};
```

又一次，大部分改动是将代码从游戏循环中移动到类中，然后重命名一些东西。 但是，在这个例子中，我们真的让代码库变简单了。 先前讨厌的命令式代码中，存在存储每个雕像的帧计数器和开火的速率的分散的本地变量。

现在那些都被移动到了Statue类中，你可以想创建多少就创建多少实例了， 每个实例都有它自己的小计时器。 这是这章背后的真实动机——现在为游戏世界增加新实体会更加简单，因为每个实体都带来了它需要的全部东西。

这个模式让我们分离了游戏世界的构建和实现。 这同样能让我们灵活地使用分散的数据文件或关卡编辑器来构建游戏世界。



还有人关心UML吗？如果还有，那就是我们刚刚建的。

Passing time

传递时间

这是模式的关键，但是我只对常用的部分进行了细化。 到目前为止，我们假设每次对`update()`的调用都推动游戏世界前进一个固定的时间。

我更喜欢那样，但是很多游戏使用可变时间步长。 在那种情况下，每次游戏循环推进的时间长度或长或短， 具体取决于它需要多长时间处理和渲染前一帧。

游戏循环一章讨论了更多关于固定和可变时间步长的优劣。

这意味着每次`update()`调用都需要知道虚拟的时钟转动了多少， 所以你经常可以看到传入消逝的时间。 举个例子，我们可以让骷髅卫士像这样处理变化的时间步长：

```
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);
    }
  }
}
```

现在，骷髅卫士移动的距离随着消逝时间的增长而增长。 也可以看出，处理变化时间步长需要的额外复杂度。 如果一次需要更新的时间步长过长，骷髅卫士也许就超过了其巡逻的范围，因此需要小心的处理。

设计决策

在这样简单的模式中，没有太多的调控之处，但是这里仍有两个你需要决策的地方：

更新方法在哪个类中？

最明显和最重要的决策就是决定将`update()`放在哪个类中。

- 实体类中：

  如果你已经有实体类了，这是最简单的选项， 因为这不会带来额外的类。如果你需要的实体种类不多，这也许可行，但是业界已经逐渐远离这种做法了。

  当类的种类很多时，一有新行为就建`Entity`子类来实现是痛苦的。 当你最终发现你想要用单一继承的方法重用代码时，你就卡住了。

- 组件类：

  如果你已经使用了组件模式，你知道这个该怎么做。 这让每个组件独立更新它自己。更新方法用了同样的方法解耦游戏中的实体，组件让你进一步解耦了单一实体中的各部分。 渲染，物理，AI都可以自顾自了。

- 委托类：

  还可将类的部分行为委托给其他的对象。 状态模式可以这样做，你可以通过改变它委托的对象来改变它的行为。 类型对象模式也这样做了，这样你可以在同“种”实体间分享行为。

  如果你使用了这些模式，将`update()`放在委托类中是很自然的。 在那种情况下，也许主类中仍有`update()`方法，但是它不是虚方法，可以简单的委托给委托对象。就像这样：

  ```
  void Entity::update()
  {
    // 转发给状态对象
    state_->update();
  }
  ```

  这样做让你改变委托对象来定义新行为。就像使用组件，这给了你无须定义全新的子类就能改变行为的灵活性。

如何处理隐藏对象？

游戏中的对象，不管什么原因，可能暂时无需更新。 它们可能是停用了，或者超出了屏幕，或者还没有解锁。 如果状态中的这种对象很多，每帧遍历它们却什么都不做是在浪费CPU循环。

一种方法是管理单独的“活动”对象集合，那些存储真正需要更新的对象。 当一个对象停用时，从那个集合中移除它。当它启用时，再把它添回来。 用这种方式，你只需要迭代那些真正需要更新的东西：

- 如果你使用单个包括了所有不活跃对象的集合：
  - 浪费时间。对于不活跃对象，你要么检查一些“是否启用”的标识，要么调用一些啥都不做的方法。

    > 检查对象启用与否然后跳过它，不但消耗了CPU循环，还报销了你的数据缓存。 CPU通过从RAM上读取数据到缓存上来优化读取。 这样做是基于刚刚读取内存之后的内存部分很可能等会也会被读取到这个假设。
    >
    > 当你跳过对象，你可能越过了缓存的尾部，强迫它从缓慢的主存中再取一块。

- 如果你使用单独的集合保存活动对象：
  - 使用了额外的内存管理第二个集合。 当你需要所有实体时，通常又需要一个巨大的集合。在那种情况下，这集合是多余的。 在速度比内存要求更高的时候（通常如此），这取舍仍是值得的。

    另一个权衡后的选择是使用两个集合，除了活动对象集合的另一个集合只包含不活跃实体而不是全部实体。
  - 得保持集合同步。 当对象创建或完全销毁时（不是暂时停用），你得修改全部对象集合和活跃对象集合。

方法选择的度量标准是不活跃对象的可能数量。 数量越多，用分离的集合，避免在核心游戏循环中用到它们就更有用。

参见

- 这个模式，以及游戏循环模式和组件模式，是构建游戏引擎核心的三位一体。
- 当你关注在每帧中更新实体或组件的缓存性能时，数据局部性模式可以让它跑到更快。
- Unity框架在多个类中使用了这个模式，包括 `MonoBehaviour`。
- 微软的XNA平台在 `Game` 和 `GameComponent` 类中使用了这个模式。
- Quintus，一个JavaScript游戏引擎在它的主`Sprite`类中使用了这个模式。

行为模式

游戏设计模式

一旦做好游戏设定，挂满了角色和道具，剩下的就是启动场景。 为了完成这点，你需要行为——告诉游戏中每个实体做什么的剧本。

当然，所有代码都是“行为”，并且所有软件都是定义行为， 但在游戏中有所不同的是，行为通常很多。 文字处理器也许有很长的特性清单， 但特性的数量与角色扮演游戏中的居民，物品和任务数量相比，那就相形见绌了。

本章的模式帮助快速定义和完善大量的行为。 类型对象定义行为的类别而无需完成真正的类。 子类沙盒定义各种行为的安全原语。 最先进的是字节码，将行为从代码中拖出，放入数据文件中。

模式

- 字节码
- 子类沙箱
- 类型对象

字节码

游戏设计模式 / Behavioral Patterns

意图

将行为编码为虚拟机器上的指令，赋予其数据的灵活性。

动机

制作游戏也许很有趣，但绝不容易。 现代游戏的代码库很是庞杂。 主机厂商和应用市场有严格的质量要求， 小小的崩溃漏洞就能阻止游戏发售。

> 我曾参与制作有六百万行C++代码的游戏。作为对比，控制火星漫游者的软件还没有其一半大小。

与此同时，我们希望榨干平台的最后一点性能。 游戏推动硬件的发展首屈一指，坚持不懈的优化只是为了跟上竞争。

为了保证稳定和性能需求，我们使用如C++这样的重量级编程语言， 它兼容多数硬件的底层结构的同时，还拥有防止漏洞的强类型系统。

我们为此感到自豪，但其亦有代价。 专业程序员需要多年的训练，之后又要对抗代码规模的增长。 构建大型游戏的时间长度在“喝杯咖啡”和 “烤咖啡豆，手磨咖啡豆，弄杯espresso，打奶泡，在拿铁咖啡里拉花。”之间。

除开这些挑战，游戏多了个苛刻的限制：“乐趣”。 玩家需要仔细权衡过的新奇体验。 那需要不断的迭代，但是如果每个调整都需要让工程师调整底层代码，然后等待漫长的编译结束，那就毁掉了创作流程。

法术战斗！

假设我们在完成一个基于法术的格斗游戏。 两个敌对的巫师互相丢法术，直到分出胜负。 我们可以将这些法术都定义在代码中，但这就意味着每当法术修改就会牵扯到工程师。 当设计者想修改几个数字感觉一下效果，就要重新编译整个工程，重启，然后进入战斗。

像现在的许多游戏一样，需要在发售之后更新游戏，修复漏洞或是添加新内容。 如果所有法术都是硬编码的，那么每次修改都意味着发行一个可以运行的游戏版本。

再扯远一点，我们还想支持模组。我们想让玩家创造自己的法术。 如果这些法术都在代码中，那么意味着每个模组制造者都得拥有编译游戏的整套工具链， 我们就得开源代码，如

果他们的自创法术上有个漏洞，那么会把其他人的游戏也搞崩溃。

数据 > 代码

很明显实现引擎的编程语言不是个好选择。 我们需要将法术放在与游戏核心隔绝的沙箱中。 我们想要它们易于修改，易于加载，并与其他可执行部分相隔离。

我不知道你怎么想，但这听上去让我觉得有点像是数据。 如果能在分离的数据文件中定义行为，游戏引擎还能加载并“执行”它们，就可以实现所有目标。

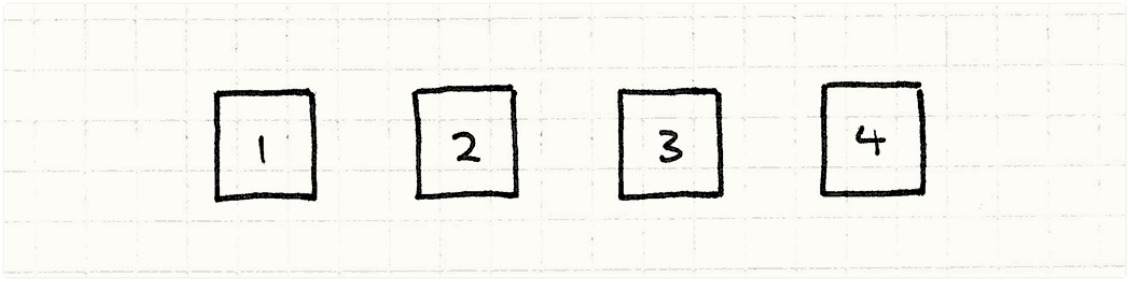
这里需要指出“执行”对于数据的意思。如何让文件中的数据表示为行为呢？这里有几种方式。 与编译模式 GoF 对比着看会好理解些。

解释器模式

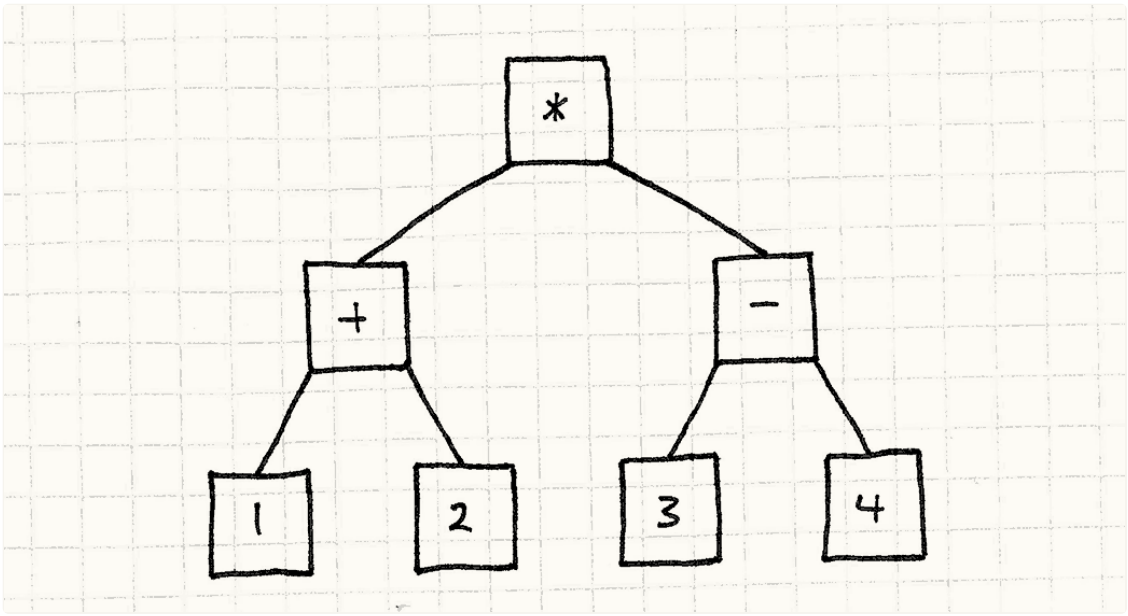
关于这个模式我就能写一章，但是有四个家伙的工作早涵盖了这一切， 所以，这里给一些简短的介绍。 从一种语言开始——想想编程语言——从你想要执行的语言开始。 比如，它支持这样的算术表达式

```
(1 + 2) * (3 - 4)
```

然后，把每块表达式，每条语言规则，都装到对象中去。数字字面量都变成对象：



基本上，它们在原始值上做了个小封装。 运算符也是对象，它们拥有操作的值的引用。 如果你考虑了括号和优先级，那么表达式就魔术般变成这样的小树：



这里的“魔术”是什么？很简单——语法分析。 语法分析器接受一串字符作为输入，将其转为抽象语法树，即一个包含了表示文本语法结构的对象集合。

完成这个你就完成了编译器的一半。

解释器模式与创建这棵树无关，它只关于执行这棵树。　它工作的方式非常巧妙。树中的每个对象都是表达式或子表达式。　用真正面向对象的方式描述，我们会让表达式自己对自己求值。

首先，我们定义所有表达式都实现的基本接口：

```
class Expression
{
public:
  virtual ~Expression() {}
  virtual double evaluate() = 0;
};
```

然后，我们定义一个类，语法中的每种表达式都实现这个接口。最简单的是数字：

```
class NumberExpression : public Expression
{
public:
  NumberExpression(double value)
  : value_(value)
  {}

  virtual double evaluate()
  {
    return value_;
  }

private:
  double value_;
};
```

一个数字表达式就等于它的值。加法和乘法有点复杂，因为它们包含子表达式。　在递归地计算其子表达式之后，才能计算自己的值。像这样：

```
class AdditionExpression : public Expression
{
public:
  AdditionExpression(Expression* left, Expression* right)
  : left_(left),
    right_(right)
  {}

  virtual double evaluate()
  {
    // 计算操作数
    double left = left_->evaluate();
    double right = right_->evaluate();

    // 把它们加起来
    return left + right;
```

```
  }

private:
  Expression* left_;
  Expression* right_;
};
```

> 我确信你知道乘法的实现是什么样的。

整齐漂亮吧？只需几个简单的类，现在我们可以表示和计算任意复杂的算术表达式。 只需要创建正确的对象，并正确连起来。

> Ruby实现这样的东西已经十五年了。在1.9版本，他们转换到了这章描述的字节码。看看我省了你多少时间！

这是个优美简单的模式，但有自己的问题。 看看插图，看到了什么？大量的小盒子，以及它们之间大量的箭头。 代码被表示为小物体组成的巨大分形树。这会带来些令人不快的后果：

- 从磁盘上加载它需要实例化并连接大量这种小对象。
- 这些对象和之间的指针会占据大量的内存。在32位机上，那个小的算术表达式至少要占据68字节，这还没考虑内存对其呢。

  > 如果你想自己算算，别忘了算上虚函数表指针。

- 遍历子表达式的指针是在谋杀数据缓存。同时，虚函数调用是在屠杀指令缓存。

  > 查看数据局部性一章，看看什么是缓存以及它是如何影响游戏性能的。

将这些拼到一起，怎么念？S-L-O-W。 这就是为什么大多数广泛应用的编程语言不基于解释器模式。 太慢了，也太消耗内存了。

虚拟的机器码

想想我们的游戏。玩家电脑在运行时并不会遍历一堆C++语法结构树。 我们提前将其编译成了机器码，CPU基于机器码运行。机器码有什么好处呢？

- 密集的。 它是一块坚实连续的二进制数据块，没有一位被浪费。
- 线性的。 指令被打成包，一条接一条的执行。不会在内存里到处乱跳（除非你控制代码流这么干）。
- 底层的。 每条指令都做一件小事，有趣的行为从组合中诞生。
- 速度快。 在以上所有的要素作用下（当然，还要算上机器码是直接在硬件上实现的），机器码跑得跟风一样快。

这听起来很好，但我们不希望为法术提供真正的机器码。 让玩家提供游戏运行时的机器码简直是在自找麻烦。 我们需要的是机器代码性能和解释器模式的安全性之间的一种妥协方案。

如果不是加载机器码并直接执行，而是定义自己的虚拟机器码呢？ 然后，在游戏中写个小模拟器。 这与机器码类似——密集，线性，相对底层——但也由游戏直接掌控，所以可以放

心地将其放入沙箱。

> 这就是为什么很多游戏主机和iOS不允许程序在运行时生成并加载机器码。 这是一种拖累，因为最快的编程语言实现就是那么做的。 它们包含了一个“即时（just-in-time）”编译器，或者JIT，在运行时将语言翻译成优化的机器码。

我们将小模拟器称为虚拟机（或简称“VM”），它运行的二进制机器码叫做字节码。 它有数据的灵活性和易用性，但比解释器模式有更好的性能。

> 在程序语言编程圈，“虚拟机”和“解释器”是同义词，我在这里交替使用。 当指代GoF的解释器模式，我会加上“模式”来表明区别。

这听起来有点吓人。 这章其余部分的目标是为了展示一下，如果把功能列表缩减下来，它实际上相当通俗易懂。 即使最终没有使用这个模式，你至少对Lua和其他许多语言有更好的了解。

模式

指令集 定义了可执行的底层操作。 一系列的指令被编码为字节序列。 虚拟机 使用 中间值堆栈 依次执行这些指令。 通过组合指令，可以定义复杂的高层行为。

何时使用

这是本书中最复杂的模式，无法轻易地加入游戏中。 当需要定义很多行为，而游戏实现语言因为以下原因不能很好地完成任务时使用它：

- 过于底层，繁琐易错。
- 编译时间长，迭代缓慢。
- 安全性依赖编程者。如果想保证行为不会破坏游戏，你需要将其与代码的其他部分隔开。

当然，该列表描述了一堆特性。谁不希望有更快的迭代循环和更多的安全性？ 然而，世上没有免费的午餐。字节码比本地代码慢，所以不适合引擎的性能攸关部分。

记住

创建自己的语言或者建立系统中的系统是很有趣的。 我在这里做的是小演示，但在现实项目中，这些东西会像藤蔓一样蔓延。

> 对我来说，游戏开发也正因此而有趣。 不管哪种情况，我都创建了虚拟空间让他人游玩。

每当我看到有人定义小语言或脚本系统，他们都说，“别担心，它很小。” 于是，不可避免地，他们增加更多小功能，直到完成了一个完整的语言。 除了，和其它语言不同，它是定制的并拥有棚户区的建筑风格。

例如每一种模板语言。

当然，完成完整的语言并没有什么错。只是保证你是故意这么做的。 否则，小心的控制字节码可以表达的含义。在其失控前为其系上皮带。

你需要一个前端

底层的字节码指令有利于性能，但是二进制的字节码格式不是用户能写的。 我们将行为移出代码的一个原因是想要在高层表示它。 如果说写C++太过底层，那么让用户写汇编——虽然是你设计的——可不是一个改进方案！

> 一个反例的是游戏RoboWar。 在游戏中，玩家 编写类似汇编的语言控制机器人，我们这里也会讨论这种指令集。
>
> 这是我介绍类似汇编的语言的首选。

就像GoF的解释器模式，它假设有一些方法来生成字节码。 通常情况下，用户在更高层编写行为，再用工具将其翻译为虚拟机能理解的字节码。 这里的工具就是编译器。

我知道，这听起来很吓人。丑话说在前头， 如果没有资源制作编辑器，那么字节码不适合你。 但是，接下来你会看到，也可能没你想的那么糟。

你会想念调试器

编程很难。我们知道想要机器做什么，但并不总能正确地传达——所以我们会写出漏洞。 为了查找和解决漏洞，我们已经积累了一堆工具来了解代码做错了什么，以及如何修正。 我们有调试器，静态分析器，反编译工具等。 所有这些工具都是为现有的语言设计的：无论是机器码还是某些更高层次的东西。

当你定义自己的字节码虚拟机时，你离开了这些工具。 当然，可以通过调试器调试虚拟机，但它告诉你虚拟机本身在做什么，而不是字节码被翻译成了什么。 它不能把字节码映射回原先的高层次的形式。

如果你定义的行为很简单，可能无需太多工具帮忙调试就能勉强坚持下来。 但随着内容规模增长，花些时间完成些功能，让用户看到字节码在做什么。 这些功能也许不随游戏发布，但它们至关重要，它们能确保你确实能发布你的游戏。

> 当然，如果你想要让游戏支持mod，那你会发布这些特性，它们就更加重要了。

示例代码

经历了前面几个章节后，你也许会惊讶于它的实现是多么直接。 首先需要为虚拟机设定一套指令集。 在开始考虑字节码之类的东西前，先像思考API一样思考它。

法术的API

如果直接使用C++代码定义法术，代码需要调用何种API呢？ 在游戏引擎中，构成法术的基本操作是什么样的？

大多数法术最终改变一个巫师的状态，因此先从这样的代码开始。

```
void setHealth(int wizard, int amount);
```

```
void setWisdom(int wizard, int amount);
void setAgility(int wizard, int amount);
```

第一个参数指定哪个巫师被影响，0代表玩家而1代表对手。 以这种方式，治愈法术可以治疗玩家的巫师，而伤害法术伤害他的敌人。 这三个小方法覆盖了出人意料多的法术。

如果法术只是默默地调整数据，游戏逻辑就已经完成了， 但玩这样的游戏会让玩家无聊得要哭。让我们修复这点：

```
void playSound(int soundId);
void spawnParticles(int particleType);
```

这并不影响游戏玩法，但它们增强了游戏的体验。 我们可以增加一些镜头晃动，动画之类的，但这已经足以作为开始了。

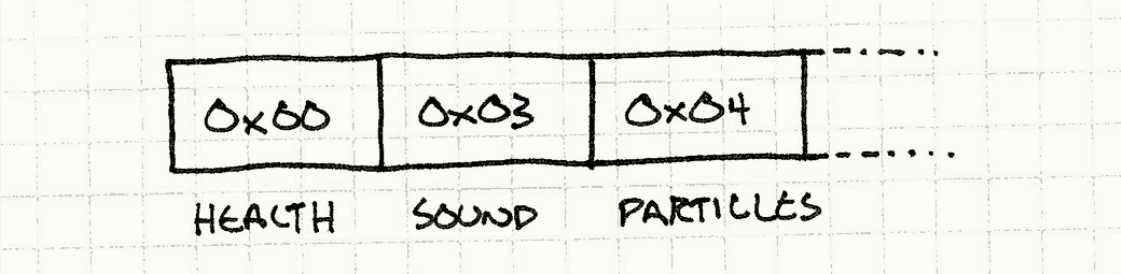
法术指令集

现在让我们把这种程序化的API转化为可被数据控制的东西。 从小处开始，然后慢慢拓展到整体。 现在，要去除方法的所有参数。 假设set__()方法总影响玩家的巫师，总直接将状态设为最大值。 同样，FX操作总是播放一个硬编码的声音和粒子效果。

这样，一个法术就只是一系列指令。 每条指令都代表了想要呈现的操作。我们可以枚举如下：

```
enum Instruction
{
  INST_SET_HEALTH      = 0x00,
  INST_SET_WISDOM      = 0x01,
  INST_SET_AGILITY     = 0x02,
  INST_PLAY_SOUND      = 0x03,
  INST_SPAWN_PARTICLES = 0x04
};
```

为了将法术编码进数据，我们存储了一数组enum值。 只有几个不同的基本操作原语，因此enum值的范围可以存储到一个字节中。 这就意味着法术的代码就是一系列字节——也就是“字节码”。



> 有些字节码虚拟机为每条指令使用多个字节，用更加复杂的规则译解它们。 事实上，在常见芯片上，比如x86上，的机器码更加复杂。
>
> 但单字节对于Java虚拟机和支撑了.NET平台的Common Language Runtime已经足够了，对我们也应该足够了。

为了执行一条指令，我们看看它的基本操作原语是什么，然后调用正确的API方法。

```
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;
}
```

用这种方式，解释器建立了沟通代码世界和数据世界的桥梁。我们可以将执行法术的虚拟机实现如下：

```
class VM
{
public:
  void interpret(char bytecode[], int size)
  {
    for (int i = 0; i < size; i++)
    {
      char instruction = bytecode[i];
      switch (instruction)
      {
        // 每条指令的跳转分支……
      }
    }
  }
};
```

输入它，你就完成你的首个虚拟机。 不幸的是，它并不灵活。 我们不能设定攻击对手的法术，也不能减少状态上限。我们只能播放声音！

为了真正有一点语言的感觉，我们需要在这里引入参数。

栈式机器

要执行复杂的嵌套表达式，得先从最里面的子表达式开始。 计算完里面的，结果向外作为

参数流向包含它们的表达式， 直到得出最终结果，整个表达式就算完了。

解释器模式将其明确的表现为嵌套对象组成的树，但我们需要指令速度达到列表的速度。 我们仍然需要确保子表达式的结果流向正确的表达式。 但由于数据是平面的，我们使用指令的顺序来控制这一点。 用CPU同样的方式完成这点——用栈。

> 这种架构不出所料的被称为栈式计算机。编程语言像Forth，PostScript，和Factor 直接将这点暴露给用户。

```
class VM
{
public:
  VM()
  : stackSize_(0)
  {}

  // 其他代码……

private:
  static const int MAX_STACK = 128;
  int stackSize_;
  int stack_[MAX_STACK];
};
```

虚拟机用内部栈保存值。在例子中，指令交互的值只有一种，那就是数字， 所以可以使用简单的int数组。 每当数据需要从一条指令传到另一条，它得通过栈。

顾名思义，值可以压入栈或者从栈弹出，所以让我们加一对方法。

```
class VM
{
private:
  void push(int value)
  {
    // 检查栈溢出
    assert(stackSize_ < MAX_STACK);
    stack_[stackSize_++] = value;
  }

  int pop()
  {
    // 保证栈不是空的
    assert(stackSize_ > 0);
    return stack_[--stackSize_];
  }

  // 其余的代码
};
```

当一条指令需要接受参数，将参数从栈弹出，如下所示：

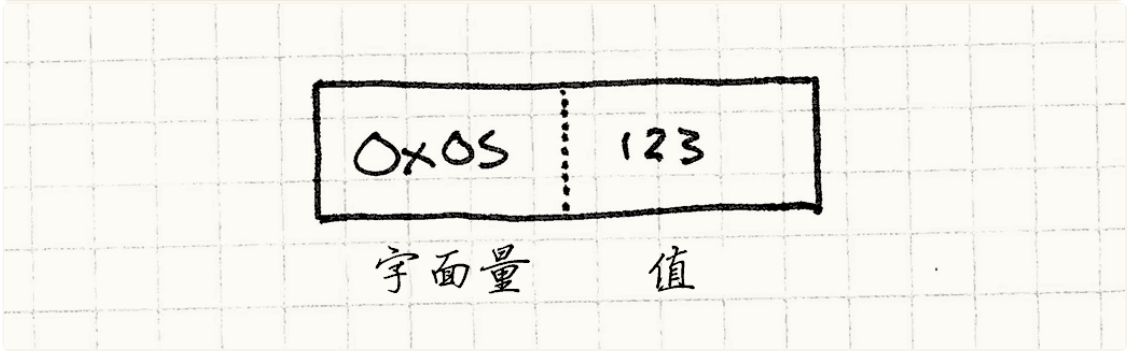
```
switch (instruction)
{
  case INST_SET_HEALTH:
  {
    int amount = pop();
    int wizard = pop();
    setHealth(wizard, amount);
    break;
  }

  case INST_SET_WISDOM:
  case INST_SET_AGILITY:
    // 像上面一样……

  case INST_PLAY_SOUND:
    playSound(pop());
    break;

  case INST_SPAWN_PARTICLES:
    spawnParticles(pop());
    break;
}
```

为了将一些值存入栈中，需要另一条指令：字面量。 它代表了原始的整数值。但是它的值又是从哪里来的呢？ 我们怎么样避免这样追根溯源到无穷无尽呢？

技巧是利用指令是字节序列这一事实——我们可以直接将数值存储在字节数组中。 如下，我们为数值字面量定义了另一条指令类型：

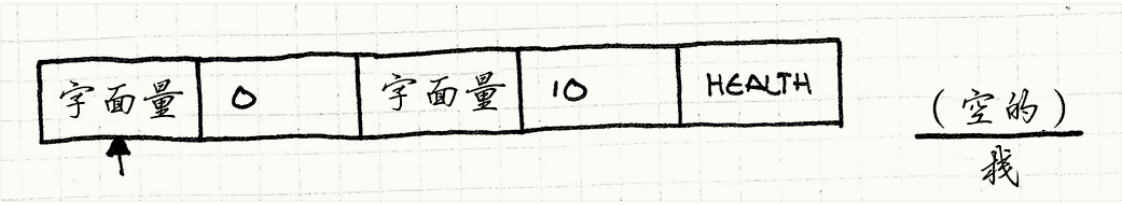
```
case INST_LITERAL:
{
  // 从字节码中读取下一个字节
  int value = bytecode[++i];
  push(value);
  break;
}
```

这里，从单个字节中读取值，从而避免需要解码多字节整数的精巧代码， 但在真实实现中，你会需要支持整个数域的字面量。

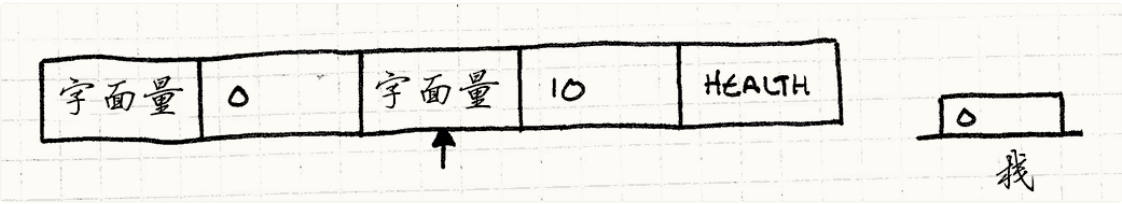


它读取字节码流中的字节作为数值并将其压入栈。

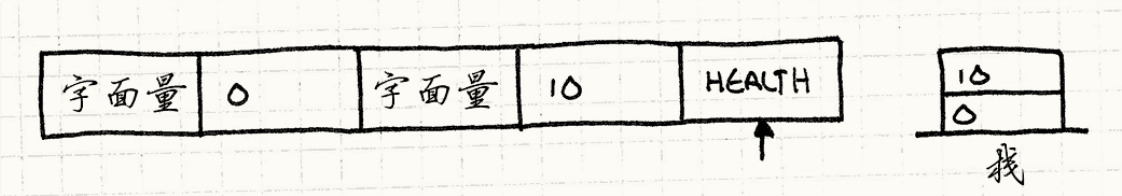
让我们串起来其中的几条指令，看看解释器如何执行它们，感受下栈是如何工作的。 从空栈开始，解释器指向第一个指令：



首先，它执行第一条`INST_LITERAL`，读取字节码流的下一个字节(`0`)并压入栈中。



然后，它执行第二条`INST_LITERAL`，读取`10`然后压入。



最后，执行`INST_SET_HEALTH`。这弹出`10`存进`amount`，弹出`0`存进`wizard`。然后用这两个参数调用`setHealth()`。

完成！我们获得了将玩家巫师血量设为10点的法术。 现在我们拥有了足够的灵活度，来定义修改任一巫师的状态到任意值的法术。 我们还可以放出不同的声音和粒子效果。

但是……这感觉还是像数据格式。比如，不能将巫师的血量提升为他智力的一半。 设计师想要有能力为法术设计规则，而不仅仅是数值。

行为 = 组合

如果我们视小虚拟机为编程语言，现在支持的只有一些内置函数，以及常量参数。 为了让字节码感觉像行为，我们缺少的是组合。

设计师需要能以有趣的方式组合不同的值，来创建表达式。 举个简单的例子，他们想让法术变化一个数值而不是变到一个数值。

这需要考虑到状态的当前值。 我们有指令来修改状态，现在需要添加方法读取状态：

```
case INST_GET_HEALTH:
{
  int wizard = pop();
  push(getHealth(wizard));
  break;
}

case INST_GET_WISDOM:
case INST_GET_AGILITY:
  // 你知道思路了吧……
```

正如你所看到的，这以两种方式与堆栈交互。 弹出一个参数来确定获取哪个巫师的状态，然后查找状态的值并压入栈中。

这允许我们写复制状态的法术。 我们可以创建一个法术，根据巫师的智慧设定敏捷度，或者让巫师的血量等于对方的血量。

有改进，但仍有限制。接下来，我们需要算术。 是时候让小虚拟机学习如何计算1 + 1了，我们将添加更多的指令。 现在，你可能已经知道如何去做，猜到了大概的模样。我只展示加法：

```
case INST_ADD:
{
  int b = pop();
  int a = pop();
  push(a + b);
  break;
}
```

像其他指令一样，它弹出数值，做点工作，然后压入结果。 直到现在，看起来每个新指令是在逐步改善，但其实我们已完成大飞跃。 这并不显而易见，但现在可以处理各种复杂的，深层嵌套的算术表达式。

来看个稍微复杂点的例子。 假设我们希望有个法术，能让巫师的血量增加敏捷和智慧的平均值。 用代码表示如下：

```
setHealth(0, getHealth(0) +
    (getAgility(0) + getWisdom(0)) / 2);
```

你可能会认为我们需要指令来处理括号造成的分组，但栈隐式支持了这一点。可以手算如下：

1. 获取巫师当前的血量并记录。
2. 获取巫师敏捷并记录。
3. 对智慧执行同样的操作。
4. 获取最后两个值，加起来并记录。
5. 除以二并记录。
6. 回想巫师的血量，将它和这结果相加并记录。
7. 取出结果，设置巫师的血量为这一结果。

你看到这些“记录”和“回想”了吗？每个“记录”对应一个压入，“回想”对应弹出。 这意味着可以很容易将其转化为字节码。例如，第一行获得巫师的当前血量：

```
LITERAL 0
GET_HEALTH
```

这些字节码将巫师的血量压入堆栈。 如果我们机械地将每行都这样转化，最终得到一大块等价于原来表达式的字节码。 为了让你感觉这些指令是如何组合的，我在下面给你做个示范。

为了展示堆栈如何随着时间推移而变化，我们举个代码执行的例子。 巫师目前有45点血量

，7点敏捷，和11点智慧。 每条指令的右边是栈在执行指令之后的模样，再右边是解释指令意图的注释：

```
LITERAL 0    [0]            # 巫师索引
LITERAL 0    [0, 0]         # 巫师索引
GET_HEALTH   [0, 45]        # 获取血量()
LITERAL 0    [0, 45, 0]     # 巫师索引
GET_AGILITY  [0, 45, 7]     # 获取敏捷()
LITERAL 0    [0, 45, 7, 0]  # 巫师索引
GET_WISDOM   [0, 45, 7, 11] # 获取智慧()
ADD          [0, 45, 18]    # 将敏捷和智慧加起来
LITERAL 2    [0, 45, 18, 2] # 被除数：2
DIVIDE       [0, 45, 9]     # 计算敏捷和智慧的平均值
ADD          [0, 54]        # 将平均值加到现有血量上。
SET_HEALTH   []             # 将结果设为血量
```

如果你注意每步的栈，你可以看到数据如何魔法一般流动在其中。 我们最开始压入`0`来查找巫师，然后它一直挂在栈的底部，直到最终的`SET_HEALTH`才用到它。

> 也许“魔法”在这里的门槛太低了。

一台虚拟机

我可以继续下去，添加越来越多的指令，但是时候适可而止了。 如上所述，我们已经有了一个可爱的小虚拟机，可以使用简单，紧凑的数据格式，定义开放的行为。 虽然“字节码”和“虚拟机”的听起来很吓人，但你可以看到它们往往简单到只需栈，循环，和switch语句。

还记得我们最初让行为呆在沙盒中的目标吗？ 现在，你已经看到虚拟机是如何实现的，很明显，那个目标已经完成。 字节码不能把恶意触角伸到游戏引擎的其他部分，因为我们只定义了几个与其他部分接触的指令。

我们通过控制栈的大小来控制内存使用量，很小心地确保它不会溢出。 我们甚至可以控制它使用多少时间。 在指令循环里，可以追踪已经执行了多少指令，如果遇到了问题也可以摆脱困境。

> 控制运行时间在例子中没有必要，因为没有任何循环的指令。 可以限制字节码的总体大小来限制运行时间。 这也意味着我们的字节码不是图灵完备的。

这里还有一个问题：创建字节码。 到目前为止，我们使用伪代码，再手工编写为字节码。除非你有很多的空闲时间，否则这种方式并不实用。

语法转换工具

最初的目标是创造更高层方式来控制行为，但是，我们却创造了比C++更底层的东西。 它具有我们想要的运行性能和安全性，但绝对没有对设计师友好的可用性。

为了填补这一空白，我们需要一些工具。 我们需要一个程序，让用户定义法术的高层次行为，然后生成对应的低层栈式机字节码。

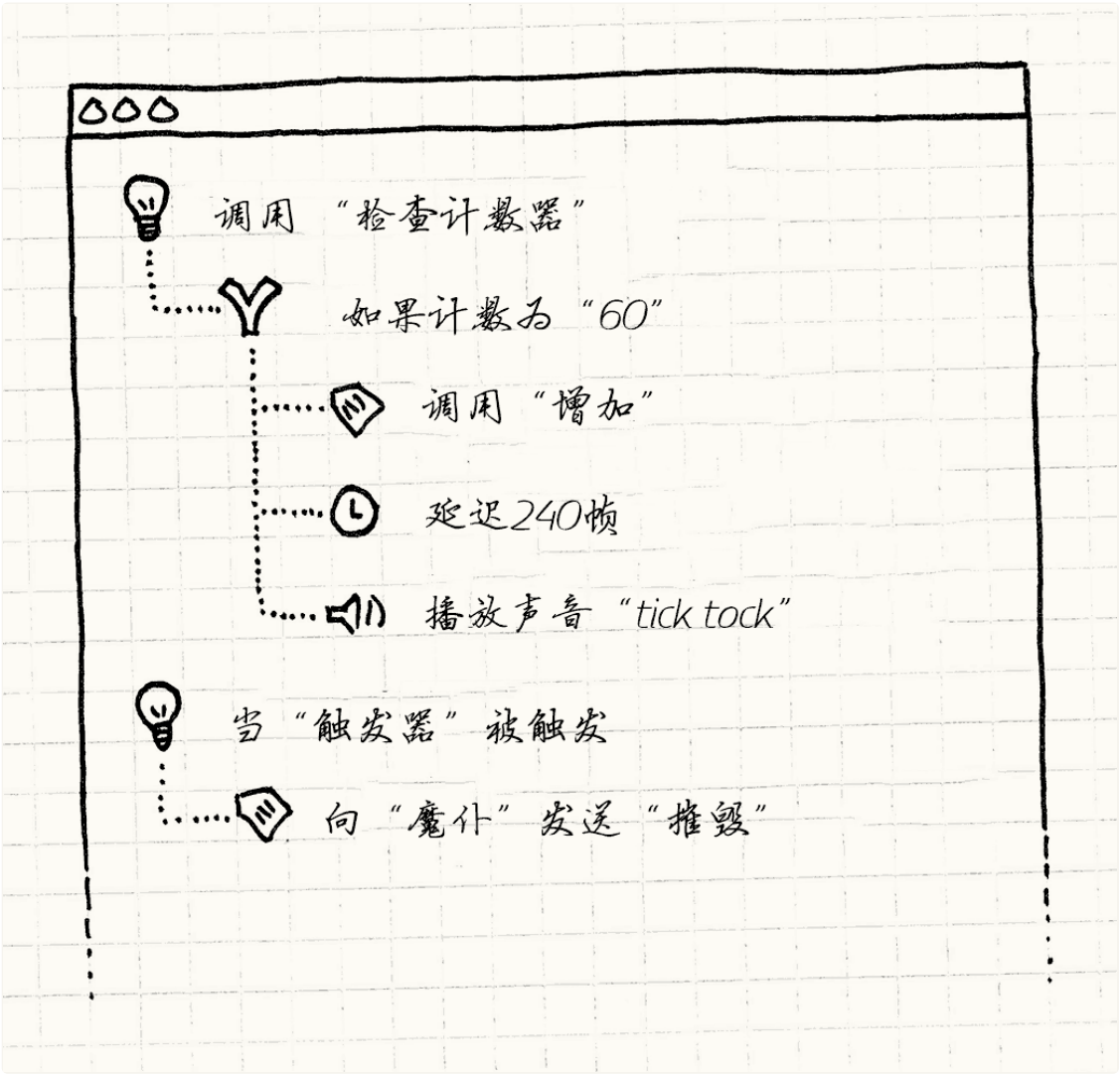
这可能听起来比虚拟机更难。 许多程序员都在大学参加编译器课程，并被龙书或者”lex“和”yacc”引发了PTSD。

> 我指的，当然，是经典教材Compilers: Principles, Techniques, and Tools。

事实上，编译一个基于文本的语言并不那么糟糕，可能有一点糟糕。 那需要补习研究众多话题。但是，你不需要那么做。 我说，我们需要的是工具——它并不一定是个输入格式是文本文件的编译器。

相反，我建议你考虑构建图形界面让用户定义自己的行为， 尤其是使用它的人没有很高的技术。 没有花几年时间习惯编译器怒吼的人很难写出没有语法错误文本。

你可以建立一个应用程序，用户通过单击拖动小盒子，下拉菜单项，或任何有意义的行为创建“脚本”，从而创建行为。



> 我为Henry Hatsworth in the Puzzling Adventure编写的脚本系统就是这么工作的。

这样做的好处是，你的UI可以保证用户无法创建“无效的”程序。 与其向他们喷射错误警告，不如主动关闭按钮或提供默认值， 以确保他们创造的东西在任何时间点上都有效。

> 我想要强调错误处理是多么重要。作为程序员，我们趋向于将人类错误视为想要终结的个人耻辱。
>
> 为了制作用户享受的系统，你需要拥抱人性，包括他们的失败。 制造错误是人们固有的，同时也是创作的固有基础。 用撤销这样的特性优雅地处理它们，这能让用户更有创意，创作出更好的成果。

这免去了设计语法和编写解析器的工作。 但是，我知道，你可能会发现UI设计同样令人不快。 好吧，如果这样，我没啥办法啦。

毕竟，这种模式是关于使用对用户友好的高层方式表达行为。 你必须精心设计用户体验。要有效地执行行为，又需要将其转换成底层形式。这是必做的，但如果你准备好迎接挑战，这终会有所回报。

设计决策

我尽可能让本章简短，但我们真正做的事情是创造语言。 那有开放的设计空间，你可以从中获得很多乐趣，所以别忘了完成你的游戏。

> 这是本书中最长的章节，我看来失败了。

指令如何访问堆栈？

有两种主要的字节码虚拟机：基于栈的和基于寄存器的。 栈式虚拟机中，指令总是操作栈顶，如同我们的示例代码所示。 例如，`INST_ADD`弹出两个值，将它们相加，将结果压入。

基于寄存器的虚拟机也有栈。唯一不同是指令可以从栈的深处读取值。 不像`INST_ADD`始终弹出其操作数， 它在字节码中存储两个索引，指示了从栈的何处读取操作数。

- 基于栈的虚拟机：
  - 指令短小。 由于每个指令隐式认定在栈顶寻找参数，不需要为任何数据编码。 这意味着每条指令可能会非常短，一般只需一个字节。
  - 易于生成代码。 当你需要为生成字节码编写编译器或工具时，你会发现更容易生成基于栈的字节码。 由于每个指令隐式在栈顶工作，你只需要以正确的顺序输出指令就可以在它们之间传递参数。
  - 会生成更多的指令。 每条指令只能看到栈顶。这意味着，产生像`a = b + c`这样的代码， 你需要单独的指令将`b`和`c`压入栈顶，执行操作，再将结果压入`a`。
- 基于寄存器的虚拟机：
  - 指令较长。 由于指令需要参数记录栈偏移量，单个指令需要更多的位。 例如，一个Lua指令——可能是最著名的基于寄存器的虚拟机——占用完整的32位。 它采用6位做指令类型，其余的是参数。

    > Lua作者没有指定Lua的字节码格式，它每个版本都会改变。现在描述的是Lua 5.1。 要深究Lua的内部构造， 读读这个。
  - 指令较少。 由于每个指令可以做更多的工作，你不需要那么多的指令。 有人说，性能会得以提升，因为不需要将值在栈中移来移去了。

所以，应该选一种？我的建议是坚持使用基于栈的虚拟机。 它们更容易实现，也更容易生成代码。 Lua转换为基于寄存器的虚拟机从而变得更快，这为寄存器虚拟机博得了声誉，但是这强烈依赖于实际的指令和虚拟机的其他大量细节。

你有什么指令？

指令集定义了在字节码中可以干什么，不能干什么，对虚拟机性能也有很大的影响。 这里有个清单，记录了不同种类的，你可能需要的指令：

- 外部基本操作原语。 这是虚拟机与引擎其他部分交互，影响玩家所见的部分。 它们控制了字节码可以表达的真实行为。 如果没有这些，你的虚拟机除了消耗CPU循环以外一无所得。
- 内部基本操作原语 这些语句在虚拟机内操作数值——文字，算术，比较操作，以及操纵栈的指令。
- 控制流。 我们的例子没有包含这些，但当你需要有条件执行或循环执行，你需要控制流。 在字节码这样底层的语言，它们出奇的简单：跳转。

  在我们的指令循环中，需要索引来跟踪到了字节码的哪里。 跳转指令做的是修改这个索引并改变将要执行的。 换言之，这是`goto`。你可以基于它制定各种更高级别的控制流。
- 抽象。 如果用户开始在数据中定义很多的东西，最终要重用字节码的部分位，而不是复制和粘贴。 你也许需要可调用过程这样的东西。

  最简单的形式中，过程并不比跳转复杂。 唯一不同的是，虚拟机需要管理另一个返回的栈。 当执行“call”指令时，将当前指令索引压入栈中，然后跳转到被调用的字节码。 当它到了“return”，虚拟机从堆栈弹出索引，然后跳回索引指示的位置。

数值是如何表示的？

我们的虚拟机示例只与一种数值打交道：整数。 回答这个问题很简单——栈只是一栈的`int`。 更加完整的虚拟机支持不同的数据类型：字符串，对象，列表等。 你必须决定在内部如何存储这些值。

- 单一数据类型：
  - 简单易用 你不必担心标记，转换，或类型检查。
  - 无法使用不同的数据类型。 这是明显的缺点。将不同类型成塞进单一的表示方式——比如将数字存储为字符串——这是找打。
- 带标记的类型：

  这是动态类型语言中常见的表示法。 所有的值有两部分。 第一部分是类型标识——一个`enum`——标识存储了数据的类型。这些位的其余部分会被解释为这种类型：

```
enum ValueType
{
  TYPE_INT,
  TYPE_DOUBLE,
  TYPE_STRING
};

struct Value
{
  ValueType type;
  union
  {
    int    intValue;
    double doubleValue;
    char*  stringValue;
```

```
  };
};
```

- 数值知道其类型。 这个表示法的好处是可在运行时检查值的类型。 这对动态调用很重要，可以确保没有在类型上面执行其不支持的操作。
- 消耗更多内存。 每个值都要带一些额外的位来标识类型。在像虚拟机这样的底层，这里几位，那里几位，总量就会快速增加。

- 无标识的**union**：

像前面一样使用union，但是没有类型标识。 你可以将这些位表示为不同的类型，由你确保没有搞错值的类型。

这是静态类型语言在内存中表示事物的方式。 由于类型系统在编译时保证没弄错值的类型，不需要在运行时对其进行验证。

> 这也是无类型语言，像汇编和Forth存储值的方式。 这些语言让用户保证不会写出误认值类型的代码。毫无服务态度！

- 结构紧凑。 找不到比只存储需要的值更加有效率的存储方式。
- 速度快。 没有类型标识意味着在运行时无需消耗周期检查它们的类型。这是静态类型语言往往比动态类型语言快的原因之一。
- 不安全。 这是真正的代价。一块错误的字节码，会让你误解一个值，把数字误解为指针，会破坏游戏安全导致崩溃。

> 如果你的字节码是由静态类型语言编译而来，你也许认为它是安全的，因为编译不会生成不安全的字节码。 那也许是真的，但记住恶意用户也许会手写恶意代码而不经过你的编译器。
>
> 这就是为什么，举个例子，Java虚拟机在加载程序时要做字节码验证。

- 接口：

多种类型值的面向对象解决方案是通过多态。接口为不同的类型测试和转换提供虚方法，如下：

```
class Value
{
public:
  virtual ~Value() {}

  virtual ValueType type() = 0;

  virtual int asInt() {
    // 只能在int上调用
    assert(false);
    return 0;
  }

  // 其他转换方法……
};
```

然后你为每个特定的数据类型设计特定的类，如：

```
class IntValue : public Value
{
public:
  IntValue(int value)
  : value_(value)
  {}

  virtual ValueType type() { return TYPE_INT; }
  virtual int asInt() { return value_; }

private:
  int value_;
};
```

- 开放。 可在虚拟机的核心之外定义新的值类型，只要它们实现了基本接口就行。
- 面向对象。 如果你坚持OOP原则，这是做事情“正确”的方式，为特定类型使用多态分配行为，而不是在标签上做switch之类的。
- 冗长。 必须定义单独的类，包含了每个数据类型的相关行为。 注意在前面的例子中，这样的类定义了所有的类型。在这里，只包含了一个！
- 低效。 为了使用多态，必须使用指针，这意味着即使是短小的值，如布尔和数字，也得裹在堆中的对象里。 每使用一个值，你就得做一次虚方法调用。

  在虚拟机核心之类的地方，像这样的性能影响会迅速叠加。 事实上，这引起了许多我们试图在解释器模式中避免的问题。 只是现在的问题不在代码中，而是在值中。

我的建议是，如果你可以只用单一数据类型，那就这么做。 除此以外，使用带标识的union。这是世界上几乎每个语言解释器做的事情。

如何生成字节码？

我将最重要的问题留到最后。我们已经完成了消耗和解释字节码， 但需要你写制造字节码的工具。 典型的解决方案是写个编译器，但它不是唯一的选择。

- 如果你定义基于文本的语言：
  - 必须定义语法。 业余和专业的语言设计师小看这件事情的难度。让解析器快乐很简单，让用户快乐很难。

    语法设计是用户界面设计，当你将用户界面限制到字符构成的字符串，这可没把事情变简单。
  - 必须实现解析器。 不管名声如何，这部分其实非常简单。无论使用ANTLR或Bison，还是——像我一样——手写递归下降，都可以完成。
  - 必须处理语法错误。 这是最重要和最困难的部分。 当用户制造了语法和语义错误——他们总会这么干——引导他们返回到正确的道路是你的任务。 解析器只知道接到了意外的符号，给予有用的的反馈并不容易。
  - 可能会对非技术用户关上大门。 我们程序员喜欢文本文件。结合强大的命令行工具

，我们把它们当作计算机的乐高积木——简单，有百万种方式组合。

大部分非程序员不这样想。 对他们来说，输入文本文件就像为愤怒机器人审核员填写税表，如果忘记了一个分号就会遭到痛斥。

- 如果你定义了一个图形化创作工具：
  - 必须实现用户界面。 按钮，点击，拖动，诸如此类。 有些人畏惧它，但我喜欢它。如果沿着这条路走下去，设计用户界面和工作核心部分同等重要——而不是硬着头皮完成的乱七八糟工作。

    每点额外工作都会让工具更容易更舒适地使用，并直接导致了游戏中更好的内容。如果你看看很多游戏制作过程的内部解密，经常会发现制作有趣的创造工具是秘诀之一。
  - 有较少的错误情况。 由于用户通过交互式一步一步地设计行为，应用程序可以尽快引导他们走出错误。

    而使用基于文本的语言时，直到用户输完整个文件才能看到用户的内容，更难预防和处理错误。
  - 更难移植。 文本编译器的好处是，文本文件是通用的。编译器简单地读入文件并写出。跨平台移植的工作实在微不足道。

    > 除了换行符。还有编码。

    当你构建用户界面，你必须选择要使用的架构，其中很多是基于某个操作系统。 也有跨平台的用户界面工具包，但他们往往要为对所有平台同样适用付出代价——对所有的平台上同样不适用。

参见

- 这一章节的近亲是GoF的解释器模式[GoF]。两种方式都能让你用数据组合行为。

  事实上，最终你两种模式都会使用。你用来构造字节码的工具会有内部的对象树。这也是解释器模式所能做的。

  为了编译到字节码，你需要递归回溯整棵树，就像用解释器模式去解释它一样。 唯一的不同在于，不是立即执行一段行为，而是生成整个字节码再执行。
- Lua是游戏中最广泛应用的脚本语言。 它的内部被实现为一个非常紧凑的，基于寄存器的字节码虚拟机。
- Kismet是个可视化脚本编辑工具，应用于Unreal引擎的编辑器UnrealEd。
- 我的脚本语言Wren，是一个简单的，基于栈的字节码解释器。

子类沙箱

游戏设计模式 / Behavioral Patterns

意图

用一系列由基类提供的操作定义子类中的行为。

动机

每个孩子都梦想过变成超级英雄，但是不幸的是，高能射线在地球上很短缺。 游戏是让你扮演超级英雄最简单的方法。 因为我们的游戏设计者从来没有学会说“不”，我们的 超级英雄游戏中有成百上千种不同的超级能力可供选择。

我们的计划是创建一个`Superpower`基类。然后由它派生出各种超级能力的实现类。 我们在程序员队伍中分发设计文档，然后开始编程。 当我们完成时，我们就会有上百种超级能力类。

> 当你发现像这个例子一样有很多子类时，那通常意味着数据驱动的方式更好。 不再用代码定义不同的能力，用数据吧。
>
> 像类型对象，字节码，和解释器 GoF 模式都能帮忙。

我们想让玩家处于拥有无限可能的世界中。无论他们在孩童时想象过什么能力，我们都要在游戏中展现。 这就意味着这些超能力子类需要做任何事情： 播放声音，产生视觉刺激，与AI交互，创建和销毁其他游戏实体，与物理打交道。没有哪处代码是它们不会接触的。

假设我们让团队信马由缰地写超能力类。会发生什么？

- 会有很多冗余代码。 当超能力种类繁多，我们可以预期有很多重叠。 很多超能力都会用相同的方式产生视觉效果并播放声音。 当你坐下来看看，冷冻光线，热能光线，芥末酱光线都很相似。 如果人们实现这些的时候没有协同，那就会有很多冗余的代码和重复劳动。
- 游戏引擎中的每一部分都会与这些类耦合。 没有深入了解的话，任何人都能写出直接调用子系统的代码，但子系统从来没打算直接与超能力类绑定。 就算渲染系统被好好组织成多个层次，只有一个能被外部的图形引擎使用， 我们可以打赌，最终超级能力代码会与每一个接触。
- 当外部代码需要改变时，一些随机超能力代码有很大几率会损坏。 一旦我们有了不同的

超能力类绑定到游戏引擎的多个部分，改变那些部分必然影响超能力类。 这可不合理，因为图形，音频，UI程序员很可能不想也成为玩法程序员。

- 很难定义所有超能力遵守的不变量。 假设我们想保证超能力播放的所有音频都有正确的顺序和优先级。 如果我们几百个类都直接调用音频引擎，就没什么好办法来完成这点。

我们要的是给每个实现超能力的玩法程序员一系列可使用的基本单元。 你想要播放声音？这是你的`playSound()`函数。 你想要粒子效果？这是你的`spawnParticles()`函数。 我们保证了这些操作覆盖了你要做的事情，所以你不需要`#include`随机的头文件，干扰到代码库的其他部分。

我们实现的方法是通过定义这些操作为`Superpower`基类的**protected**方法。 将它们放在基类给了每个子类直接便捷的途径获取方法。 让它们为protected（很可能不是虚方法）方法暗示了它们存在就是为了被子类调用。

一旦有了这些东西来使用，我们需要一个地方使用他们。 为了做到那点，我们定义沙箱方法，这是子类必须实现的抽象的protected方法。 有了那些，要实现一种新的能力，你需要：

1. 创建从`Superpower`继承的新类。
2. 重载沙箱方法`activate()`。
3. 通过调用`Superpower`提供的protected方法实现主体。

我们现在可以使用这些高层次的操作来解决冗余代码问题了。 当我们看到代码在多个子类间重复，我们总可以将其打包到`Superpower`中，作为它们都可以使用的新操作。

我们通过将耦合约束到一个地方解决了耦合问题。 `Superpower`最终与不同的系统耦合，但是继承它的几百个类不会。 相反，它们只耦合基类。 当游戏系统的某部分改变时，修改`Superpower`也许是必须的，但是众多的子类不需要修改。

这个模式带来浅层但是广泛的类层次。 你的继承链不深，但是有很多类与`Superpower`挂钩。 通过使用有很多直接子类的基类，我们在代码库中创造了一个支撑点。 我们投入到`Superpower`的时间和爱可以让游戏中众多类获益。

> 最近，你发现很多人批评面向对象语言中的继承。 继承是有问题——在代码库中没有比父类子类之间的耦合更深的了——但我发现扁平的继承树比起深的继承树更好处理。

模式

基类定义抽象的沙箱方法和几个提供的操作。 将操作标为protected，表明它们只为子类所使用。 每个推导出的沙箱子类用提供的操作实现了沙箱函数。

何时使用

子类沙箱模式是潜伏在代码库中简单常用的模式，哪怕是在游戏之外的地方亦有应用。 如果你有一个非虚的protected方法，你可能已在用类似的东西了。 沙箱方法在以下情况适用：

- 你有一个能推导很多子类的基类。

- 基类可以提供子类需要的所有操作。
- 在子类中有行为重复，你想要更容易的在它们间分享代码。
- 你想要最小化子类和程序的其他部分的耦合。

记住

“继承”在很多编程圈子近来为人诟病，原因之一是基类趋向于增加越来越多的代码　这个模式特别容易染上这个毛病。

由于子类通过基类接触游戏的剩余部分，基类最后和子类需要的每个系统耦合。　当然，子类也紧密的与基类相绑定。这种蛛网耦合让你很难在不破坏什么的情况下改变基类——你得到了（脆弱的基类问题）brittle base class problem。

硬币的另一面是由于你耦合的大部分都被推到了基类，子类现在与世界的其他部分分离。理想的情况下，你大多数的行为都载子类中。这意味着你的代码库大部分是孤立的，很容易管理。

如果你发现这个模式正把你的基类变成一锅代码糊糊，　考虑将它提供的一些操作放入分离的类中，这样基类可以分散它的责任。组件模式可以在这里帮上忙。

示例代码

因为这个模式太简单了，示例代码中没有太多东西。 这不是说它没用——这个模式关键在于“意图”，而不是它实现的复杂度。

我们从`Superpower`基类开始：

```
class Superpower
{
public:
  virtual ~Superpower() {}

protected:
  virtual void activate() = 0;

  void move(double x, double y, double z)
  {
    // 实现代码……
  }

  void playSound(SoundId sound, double volume)
  {
    // 实现代码……
  }

  void spawnParticles(ParticleType type, int count)
  {
    // 实现代码……
  }
```

```
};
```

activate()方法是沙箱方法。由于它是抽象虚函数，子类必须重载它。 这让那些需要创建子类的人知道要做哪些工作。

其他的protected函数move()，playSound()，和spawnParticles()都是提供的操作。它们是子类在实现activate()要调用的。

在这个例子中，我们没有实现提供的操作，但真正的游戏在那里有真正的代码。 那些代码中，Superpower与游戏中其他部分的耦合——move()也许调用物理代码，playSound()会与音频引擎交互，等等。 由于这都在基类的实现中，保证了耦合封闭在Superpower中。

好了，拿出我们的放射蜘蛛，创建个能力。像这样：

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    // 空中滑行
    playSound(SOUND_SPROING, 1.0f);
    spawnParticles(PARTICLE_DUST, 10);
    move(0, 0, 20);
  }
};
```

> 好吧，也许跳跃不是超级能力，但我在这里讲的是基础知识。

这种能力将超级英雄射向天空，播放合适的声音，扬起尘土。 如果所有的超能力都这样简单——只是声音，粒子效果，动作的组合——那么就根本不需要这个模式了。 相反，Superpower有内置的activate()能获取声音ID，粒子类型和运动的字段。 但是这只在所有能力运行方式相同，只在数据上不同时才可行。让我们精细一些：

```
class Superpower
{
protected:
  double getHeroX()
  {
    // 实现代码……
  }

  double getHeroY()
  {
    // 实现代码……
  }

  double getHeroZ()
  {
    // 实现代码……
  }
```

```
  // 退出之类的……
};
```

这里我们增加了些方法获取英雄的位置。我们的SkyLaunch现在可以使用它们了：

```
class SkyLaunch : public Superpower
{
protected:
  virtual void activate()
  {
    if (getHeroZ() == 0)
    {
      // 在地面上，冲向空中
      playSound(SOUND_SPROING, 1.0f);
      spawnParticles(PARTICLE_DUST, 10);
      move(0, 0, 20);
    }
    else if (getHeroZ() < 10.0f)
    {
      // 接近地面，再跳一次
      playSound(SOUND_SWOOP, 1.0f);
      move(0, 0, getHeroZ() + 20);
    }
    else
    {
      // 正在空中，跳劈攻击
      playSound(SOUND_DIVE, 0.7f);
      spawnParticles(PARTICLE_SPARKLES, 1);
      move(0, 0, -getHeroZ());
    }
  }
};
```

由于我们现在可以访问状态，沙箱方法可以做有用有趣的控制流了。 这还需要几个简单的if声明， 但你可以做任何你想做东西。 使用包含任意代码的成熟沙箱方法，天高任鸟飞了。

> 早先，我建议以数据驱动的方式建立超能力。 这里是你可能不想那么做的原因之一。 如果你的行为复杂而使用命令式风格，它更难在数据中定义。

设计决策

如你所见，子类沙箱是一个“软”模式。它表述了一个基本思路，但是没有很多细节机制。这意味着每次使用都面临着一些有趣的选择。这里是一些需要思考的问题。

应该提供什么操作？

这是最大的问题。这深深影响了模式感觉上和实际上有多好。 在一种极端中，基类几乎不

提供任何操作。只有一个沙箱方法。 为了实现功能，总是需要调用基类外部的系统。如果你这样做，很难说你在使用这个模式。

另一个极端，基类提供了所有子类也许需要的操作。 子类只与基类耦合，不调用任何外部系统的东西。

> 具体来说，这意味着每个子类的源文件只需要`#include`它的基类头文件。

在这两极端之间，操作由基类提供还是向外部直接调用有很大的操作余地。 你提供的操作越多，外部系统与子类耦合越少，但是与基类耦合越多。 从子类中移除了耦合是通过将耦合推给基类完成的。

如果你有一堆与外部系统耦合的子类的话，这很好。 通过将耦合移到提供的操作中，你将其移动到了一个地方：基类。但是你越这么做，基类就越大越难管理。

所以分界线在哪里？这里是一些首要原则：

- 如果提供的操作只被一个或几个子类使用，将操作加入基类获益不会太多。 你向基类添加了会影响所有事物的复杂性，但是只有少数几个类受益。

  让该操作与其他提供的操作保持一致或许有价值，但让使用操作的子类直接调用外部系统也许更简单明了。

- 当你调用游戏中其他地方的方法，如果方法没有修改状态就有更少的干扰。 它仍然制造耦合，但是这是“安全的”耦合，因为它没有破坏游戏中的任何东西。

  > “安全的”在这里打了引号是因为严格来说，接触数据也能造成问题。 如果你的游戏是多线程的，读取的数据可能正在被修改。如果你不小心，就会读入错误的数据。
  >
  > 另一个不愉快的情况是，如果你的游戏状态是严格确定性的（很多在线游戏为了保持玩家同步都是这样的）。 接触了游戏同步状态之外的东西会造成极糟的不确定性漏洞。

  另一方面，修改状态的调用会和代码库的其他方面紧密绑定，你需要三思。打包他们成基类提供的操作是个好的候选项。

- 如果操作只是增加了向外部系统的转发调用，那它就没增加太多价值。那种情况下，也许直接调用外部系统的方法更简单。

  但是，简单的转发也是有用的——那些方法接触了基类不想直接暴露给子类的状态。 举个例子，假设`Superpower`提供这个：

  ```
  void playSound(SoundId sound, double volume)
  {
    soundEngine_.play(sound, volume);
  }
  ```

  它只是转发调用给`Superpower`中`soundEngine_`字段。 但是，好处是将字段封装在`Superpower`中，避免子类接触。

方法应该直接提供，还是包在对象中提供？

这个模式的挑战是基类中最终加入了很多方法。 你可以将一些方法移到其他类中来缓和。

基类通过返回对象提供方法。

举个例子，为了让超能力播放声音，我们可以直接将它们加到Superpower中：

```
class Superpower
{
protected:
  void playSound(SoundId sound, double volume)
  {
    // 实现代码……
  }

  void stopSound(SoundId sound)
  {
    // 实现代码……
  }

  void setVolume(SoundId sound)
  {
    // 实现代码……
  }

  // 沙盒方法和其他操作……
};
```

但是如果Superpower已经很庞杂了，我们也许想要避免这样。 取而代之的是创建SoundPlayer类暴露该函数：

```
class SoundPlayer
{
  void playSound(SoundId sound, double volume)
  {
    // 实现代码……
  }

  void stopSound(SoundId sound)
  {
    // 实现代码……
  }

  void setVolume(SoundId sound)
  {
    // 实现代码……
  }
};
```

Superpower提供了对其的接触：

```
class Superpower
{
protected:
```

```
  SoundPlayer& getSoundPlayer()
  {
    return soundPlayer_;
  }

  // 沙箱方法和其他操作……

private:
  SoundPlayer soundPlayer_;
};
```

将提供的操作分流到辅助类可以为你做一些事情：

- 减少了基类中的方法。 在这里的例子中，将三个方法变成了一个简单的获取函数。
- 在辅助类中的代码通常更好管理。 像Superpower的核心基类，不管意图如何好，它被太多的类依赖而很难改变。 通过将函数移到耦合较少的次要类，代码变得更容易被使用而不破坏任何东西。
- 减少了基类和其他系统的耦合度。 当playSound()方法直接在Superpower时，基类与SoundId以及其他涉及音频代码直接绑定。 将它移动到SoundPlayer中，减少了Superpower与SoundPlayer类的耦合，这就封装了它其他的依赖。

基类如何获得它需要的状态？

你的基类经常需要将对子类隐藏的数据封装起来。 在第一个例子中，Superpower类提过了spawnParticles()方法。 如果方法的实现需要一些粒子系统对象，怎么获得呢？

- 将它传给基类构造器：

  最简单的解决方案是让基类将其作为构造器变量：

```
class Superpower
{
public:
  Superpower(ParticleSystem* particles)
  : particles_(particles)
  {}

  // 沙箱方法和其他操作……

private:
  ParticleSystem* particles_;
};
```

  这安全地保证了每个超能力在构造时能得到粒子系统。但让我们看看子类：

```
class SkyLaunch : public Superpower
{
public:
  SkyLaunch(ParticleSystem* particles)
  : Superpower(particles)
  {}
```

```
};
```

我们在这看到了问题。每个子类都需要构造器调用基类构造器并传递变量。这让子类接触了我们不想要它知道的状态。

这也让造成了维护的负担。如果我们后续向基类添加了状态，每个子类都需要修改并传递这个状态。

- 使用两阶初始化：

  为了避免通过构造器传递所有东西，我们可以将初始化划分为两个部分。 构造器不接受任何参数，只是创建对象。然后，我们调用定义在基类的分离方法传入必要的数据：

  ```
  Superpower* power = new SkyLaunch();
  power->init(particles);
  ```

  注意我们没有为SkyLaunch的构造器传入任何东西，它与Superpower中想要保持私有的任何东西都不耦合。 这种方法的问题在于，你要保证永远记得调用init()，如果忘了，你会获得处于半完成的，无法运行的超能力。

  你可以将整个过程封装到一个函数中来修复这一点，就像这样：

  ```
  Superpower* createSkyLaunch(ParticleSystem* particles)
  {
    Superpower* power = new SkyLaunch();
    power->init(particles);
    return power;
  }
  ```

  > 使用一点像私有构造器和友类的技巧，你可以保证createSkylaunch()函数是唯一能够创建能力的函数。 这样，你不会忘记任何初始化步骤。

- 让状态静态化：

  在先前的例子中，我们用粒子系统初始化每一个Superpower实例。 在每个能力都需要自己独特的状态时这是有意义的。但是如果粒子系统是单例 ，那么每个能力都会分享相同的状态。

  如果是这样，我们可以让状态是基类私有而静态的。 游戏仍然要保证初始化状态，但是它只需要为整个游戏初始化Superpower类一遍，而不是为每个实例初始化一遍。

  > 记住单例仍然有很多问题。你在很多对象中分享了状态（所有的Superpower实例）。 粒子系统被封装了，因此它不是全局可见的，这很好，但它们都访问同一对象，这让分析起来更加困难了。

  ```
  class Superpower
  {
  public:
    static void init(ParticleSystem* particles)
    {
      particles_ = particles;
  ```

```
  }

  // 沙箱方法和其他操作……

private:
  static ParticleSystem* particles_;
};
```

注意这里的`init()`和`particles_`都是静态的。 只要游戏早先调用过一次`Superpower::init()`，每种能力都能接触粒子系统。 同时，可以调用正确的推导类构造器来自由创建`Superpower`实例。

更棒的是，现在`particles_`是静态变量， 我们不需要在每个`Superpower`中存储它，这样我们的类占据的内存更少了。

- 使用服务定位器：

  前一选项中，外部代码要在基类请求前压入基类需要的全部状态。 初始化的责任交给了周围的代码。另一选项是让基类拉取它需要的状态。 而做到这点的一种实现的方法是使用服务定位器模式：

```
class Superpower
{
protected:
  void spawnParticles(ParticleType type, int count)
  {
    ParticleSystem& particles = Locator::getParticles();
    particles.spawn(type, count);
  }

  // 沙箱方法和其他操作……
};
```

  这儿，`spawnParticles()`需要粒子系统，不是外部系统给它，而是它自己从服务定位器中拿了一个。

参见

- 当你使用更新模式时，你的更新函数通常也是沙箱方法。
- 这个模式与模板方法 GoF正相反。 两种模式中，都使用一系列受限操作实现方法。 使用子类沙箱时，方法在推导类中，受限操作在基类中。 使用模板方法时，基类 有方法，而受限操作在推导类中。
- 你也可以认为这个模式是外观 GoF模式的变形。 外观模式将一系列不同系统藏在简化的API后。使用子类沙箱，基类起到了在子类前隐藏整个游戏引擎的作用。

类型对象

游戏设计模式 / Behavioral Patterns

意图

创造一个类**A**来允许灵活的创造新**“类型”**，类**A**的每个实例都代表了不同的对象类型。

动机

想象我们在制作一个奇幻RPG游戏。 我们的任务是为一群想要杀死英雄的恶毒怪物编写代码。 怪物有多个的属性：生命值，攻击力，图形效果，声音表现，等等。 但是为了说明介绍的目的我们先只考虑前面两个。

游戏中的每个怪物都有当前血值。 开始时是满的，每次怪物受伤，它就下降。 怪物也有一个攻击字符串。 当怪物攻击我们的英雄，那个文本就会以某种方式展示给用户。 （我们不在乎这里怎样实现。）

设计者告诉我们怪物有不同品种，像“龙”或者“巨魔”。 每个品种都描述了一种存在于游戏中的怪物，同时可能有多个同种怪物在地牢里游荡。

品种决定了怪物的初始健康——龙开始的血量比巨魔多，它们更难被杀死。 这也决定了攻击字符——同种的所有怪物都以相同的方式进行攻击。

传统的面向对象方案

想着这样的设计方案，我们启动了文本编辑器开始编程。 根据设计，龙是一种怪物，巨魔是另一种，其他品种的也一样。 用面向对象的方式思考，这引导我们创建`Monster`基类。

> 这是一种“是某物”的关系。 在传统OOP思路中，由于龙“是”怪物，我们用`Dragon`是`Monster`的子类来描述这点。 如我们将看到的，继承是一种将这种关系表示为代码的方法。

```
class Monster
{
public:
  virtual ~Monster() {}
  virtual const char* getAttack() = 0;
```

```
protected:
  Monster(int startingHealth)
  : health_(startingHealth)
  {}

private:
  int health_; // 当前血值
};
```

在怪物攻击英雄时，公开的`getAttack()`函数让战斗代码能获得需要显示的文字。 每个子类都需要重载它来提供不同的消息。

构造器是protected的，需要传入怪物的初始血量。 每个品种的子类的公共构造器调用这个构造器，传入对于该品种适合的起始血量。

现在让我们看看两个品种子类：

```
class Dragon : public Monster
{
public:
  Dragon() : Monster(230) {}

  virtual const char* getAttack()
  {
    return "The dragon breathes fire!";
  }
};

class Troll : public Monster
{
public:
  Troll() : Monster(48) {}

  virtual const char* getAttack()
  {
    return "The troll clubs you!";
  }
};
```

感叹号让所有事情都更刺激！

每个从`Monster`派生出来的类都传入起始血量，重载`getAttack()`返回那个品种的攻击字符串。 所有事情都一如所料的运行，不久以后，我们的英雄就可以跑来跑去杀死各种野兽了。 我们继续编程，在意识到之前，我们就有了酸泥怪到僵尸羊的众多怪物子类。

然后，很奇怪，事情陷入了困境。 设计者最终想要几百个品种，但是我们发现所有的时间都花费在写这些只有七行长的子类和重新编译上。 这会继续变糟——设计者想要协调已经编码的品种。我们之前富有产出的工作日退化成了：

1. 收到设计者将巨魔的血量从48改到52的邮件。
2. 签出并修改`Troll.h`。

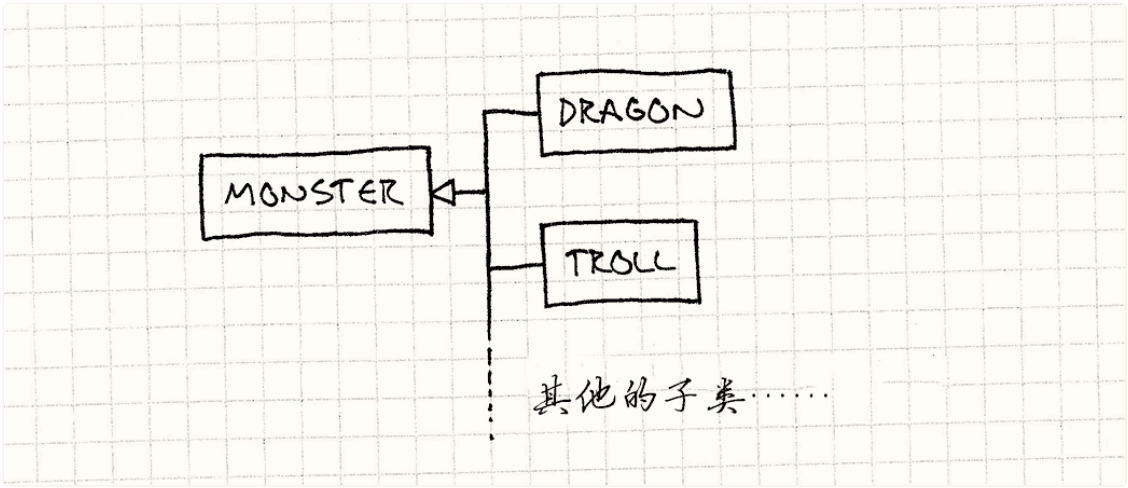
3. 重新编译游戏。
4. 签入修改。
5. 回复邮件。
6. 重复。

我们度过了失意的一天，因为我们变成了填数据的猴子。 设计者也感到挫败，因为修改一个数据就要老久。 我们需要的是一种无需每次重新编译游戏就能修改品种的状态。 如果设计者创建和修改品种时无需任何程序员的介入那就更好了。

为类型建类

从较高的层次看来，我们试图解决的问题非常简单。 游戏中有很多不同的怪物，我们想要在它们之间分享属性。 一大群怪物在攻击英雄，我们想要它们中的一些使用相同的攻击文本。 我们声明这些怪物是相同的“品种”，而品种决定了攻击字符串。

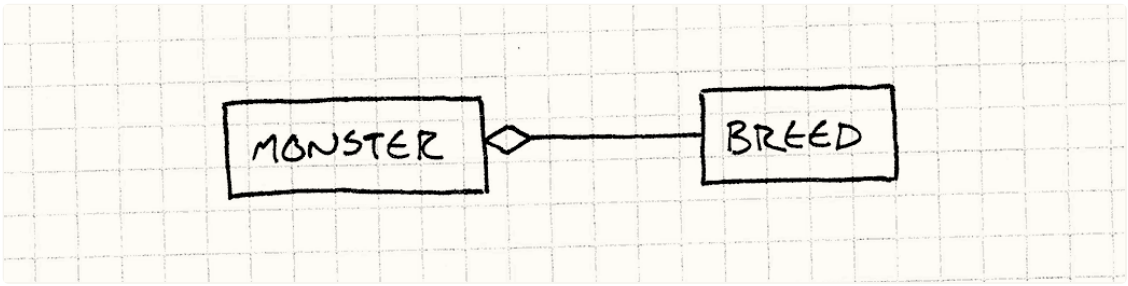
这种情况下我们很容易想到类，那就试试吧。 龙是怪物，每条龙都是龙“类”的实例。 定义每个品种为抽象基类`Monster` 的子类，让游戏中每个怪物都是子类的实例反映了那点。最终的类层次是这样的：



> 这里的  意为“从……继承”。

每个怪物的实例属于摸个继承怪物类的类型。 我们有的品种越多，类层次越高。 这当然是问题：添加新品种就需要添加新代码，而每个品种都需要被编译为它自己的类型。

这可行，但不是唯一的选项。 我们也可以重构代码让每个怪物有品种。 不是让每个品种继承`Monster`，我们现在有单一的`Monster`类和`Breed`类。



> 这里  意为“被……引用”。

这就成了，就两个类。注意这里完全没有继承。 通过这个系统，游戏中的每个怪物都是`Monster`的实例。 `Breed`类包含了在不同品种怪物间分享的信息：开始血量和攻击字符串。

为了将怪物与品种相关联，我们给了每个`Monster`实例对包含品种信息的`Breed`对象的引

用。 为了获得攻击字符串，一个怪兽可以调用它品种的方法。 `Breed`类本质上定义了一个怪物的类型，这就是为啥这个模式叫做类型对象。

这个模式特别有用的一点是，我们现在可以定义全新的类型而无需搅乱代码库。 我们本质上将部分的类型系统从硬编码的继承结构中拉出，放到可以在运行时定义的数据中去。

我们可以通过用不同值实例化`Monster`来创建成百上千的新品种。 如果从配置文件读取不同的数据初始化品种，我们就有能力完全靠数据定义新怪物品种。 这么容易，设计者也可以做到！

模式

定义类型对象类和有类型的对象类。每个类型对象实例代表一种不同的逻辑类型。 每种有类型的对象保存对描述它类型的类型对象的引用。

实例相关的数据被存储在有类型对象的实例中，被同种类分享的数据或者行为存储在类型对象中。 引用同一类型对象的对象将会像同一类型一样运作。 这让我们在一组相同的对象间分享行为和数据，就像子类让我们做的那样，但没有固定的硬编码子类集合。

何时使用

在任何你需要定义不同“种”事物，但是语言自身的类型系统过于僵硬的时候使用该模式。尤其是下面两者之一成立时：

- 你不知道你后面还需要什么类型。（举个例子，如果你的游戏需要支持资料包，而资料包有新的怪物品种呢？）
- 修改或添加新类型不想改变代码或者重新编译就能。

记住

这个模型是关于将“类型”的定义从命令式僵硬的语言世界移到灵活但是缺少行为的对象内存世界。 灵活性很好，但是将类型提到数据丧失了一些东西。

需要手动追踪类型对象

使用像C++类型系统这样的好处之一就是编译器自动记录类的注册。 定义类的数据自动编译到可执行的静态内存段然后就运作起来了。

使用类型对象模式，我们现在不但要负责管理内存中的怪物，同时要管理它们的类型 —— 我们要保证，只要我的怪物需要，所有的品种对象都能实例化并保存在内存中。 无论何时创建新的怪物，由我们来保证能初始化为有含有品种的引用。

我们从编译器的限制中解放了自己，但是代价是需要重新实现一些它以前为我们做的事情。

> C++内部使用了“虚函数表”（“vtable”）实现虚方法。 虚函数表是个简单的`struct`，包含了一集合函数指针，每个对应一个类中的虚方法。 在内存中每个类有一个虚函数表。每个类的实例有一个指针指向它类的虚函数表。
>
> 当你调用一个虚函数，代码首先在虚函数表中查找对象，然后调用表中函数指针

指向的函数。

听起来很熟悉？虚函数表就是个品种对象，而指向虚函数表的指针是怪物保留的、指向品种的引用。 C++的类是C中的类型对象，由编译器自动处理。

更难为每种类型定义行为

使用子类派生，你可以重载方法，然后做你想做的事——用程序计算值，调用其他代码，等等。 天高任鸟飞。如果我们想的话，可以定义一个怪物子类，根据月亮的阶段改变它的攻击字符串。（我觉得就像狼人。）

当我们使用类型对象模式时，我们将重载的方法替换成了成员变量。 不再让怪物的子类重载方法，用不同的代码来计算攻击字符串，我们的品种对象而是在不同的变量中存储攻击字符串。

这让使用类型对象定义类型相关的数据变得容易，但是定义类型相关的行为变得困难。 如果，举个例子，不同品种的怪物需要使用不同的AI算法，使用这个模式就面临着挑战。

有很多方式可以让我们跨越这个限制。 一个简单的方式是使用预先定义的固定行为， 然后类型对象中的数据简单的选择它们中的一个。 举例，假设我们的怪物AI总是处于“站着不动”、“追逐英雄”或者“恐惧地呜咽颤抖”（嘿，他们不可能都是强势的龙）。 我们可以定义函数来实现每种行为。 然后，我们在方法中存储合适函数的引用，将AI算法与品种相关联。

听起来很熟悉？这是在我们的类型对象中实现虚函数表。

另一个更加彻底的解决方案是真正地在数据中支持定义行为。 解释器 [GoF]模式和字节码 [模式让我们定义有行为的对象。 如果我们读取数据文件并用上面两种模式之一构建数据结构，我们就将行为完全从代码中移出，放入了数据之中。

时过境迁，游戏越来越多的由数据驱动。 硬件更为强大，我们发现比起能榨干多少硬件的性能，瓶颈更多在能完成多少内容。 使用64K的软盘的时代，挑战是将游戏塞入其中。 而在使用双面DVD的时代，挑战是用游戏填满它。

脚本语言和其他高层定义游戏行为的方式能给我们提供必要的生产力，同时只消耗可预期的运行时性能。 由于硬件越来越好，而大脑并非如此，这种交换越来越有意义。

示例代码

在第一遍实现中，让我们从简单的开始，只构建动机那节提到的基础系统。 我们从`Breed`类开始：

```
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_; // 初始血值
  const char* attack_;
};
```

很简单。它基本上只是两个数据字段的容器：起始血量和攻击字符串。 让我们看看怪物怎么使用它：

```
class Monster
{
public:
  Monster(Breed& breed)
  : health_(breed.getHealth()),
    breed_(breed)
  {}

  const char* getAttack()
  {
    return breed_.getAttack();
  }

private:
  int    health_; // 当前血值
  Breed& breed_;
};
```

当我们建构怪物时，我们给它一个品种对象的引用。 它定义了怪物的品种，取代了之前的子类。 在构造函数中，Monster使用的品种决定了起始血量。 为了获得攻击字符串，怪物简单的将调用转发给它的品种。

这段非常简单的代码是这章的核心思路。剩下的任何东西都是红利。

让类型对象更像类型：构造器

现在，我们可以直接构造怪物并负责传入它的品种。 和常用的OOP语言实现的对象相比这有些退步——我们通常不会分配一块空白内存，然后赋予它类型。 相反，我们根据类调用构造器，它负责创建一个新实例。

我们可以在类型对象上应用同样的模式。

```
class Breed
{
public:
  Monster* newMonster() { return new Monster(*this); }

  // Previous Breed code...
};
```

“模式”一词用在这里正合适。我们讨论的是设计模式中经典的模式：工厂方法 GoF 。

在一些语言中，这个模式被用来构造所有的对象。 在Ruby，Smalltalk，Objective-C以及其他类是对象的语言中，你通过在类对象本身上调用方法来构建实例。

以及那个使用它们的类：

```
class Monster
{
  friend class Breed;

public:
  const char* getAttack() { return breed_.getAttack(); }

private:
  Monster(Breed& breed)
  : health_(breed.getHealth()),
    breed_(breed)
  {}

  int health_; // 当前血值
  Breed& breed_;
};
```

不同的关键点在于Breed中的newMonster()。 这是我们的“构造器”工厂方法。使用我们原先的实现，就像这样创建怪物：

这里还有一个小小的不同。 因为样例代码由C++写就，我们可以使用一个小小的特性：友类。

我们让Monster的构造器成为私有，防止了任何人直接调用它。 友类放松了这个限制，Breed仍可接触它。 这意味着构造怪物的唯一方法是通过newMonster()。

```
Monster* monster = new Monster(someBreed);
```

在我们改动后，它看上去是这样：

```
Monster* monster = someBreed.newMonster();
```

所以，为什么这么做？创建一个对象分为两步：内存分配和初始化。 Monster的构造器让我们做完了所有需要的初始化。 在例子中，那只存储了类型，但是在完整的游戏中，那需要加载图形，初始化怪物AI以及做其他的设置工作。

但是，那都发生在内存分配之后。 在构造器调用前，我们已经找到了内存放置怪物。 在游戏中，我们通常也想控制对象创造这一环节： 我们通常使用自定义的分配器或者对象池 模式来控制对象最终在内存中的位置。

在Breed中定义“构造器”函数给了我们地方实现这些逻辑。 不是简单的调用new,newMonster()函数可以在将控制权传递给Monster初始化之前，从池中或堆中获取内存。 通过在唯

一有能力创建怪物的Breed函数中放置这些逻辑， 我们保证了所有怪物变量遵守了内存管理规范。

通过继承分享数据

我们现在已经实现了能完美服务的类型对象系统，但是它非常基础。 我们的游戏最终有上百种不同品种，每种都有成打的特性。 如果设计者想要协调30中不同的巨魔，让它们变得强壮一点，他会的处理很多数据。

能帮上忙的是在不同品种间分享属性的能力，一如品种在不同的怪物间分享属性的能力。就像我们在之前OOP方案中做的那样，我们可以使用派生完成这点。 只是，这次，不使用语言的继承机制，我们用类型对象实现它。

简单起见，我们只支持单继承。 就像类可以有一个父类，我们允许品种有一个父品种：

```
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_; // 初始血值
  const char* attack_;
};
```

当我们构建一个品种，我们先传入它继承的父品种。 我们可以为基础品种传入NULL表明它没有祖先。

为了让这有用，子品种需要控制它从父品种继承了哪些属性，以及哪些属性需要重载并由自己指定。 在我们的示例系统中，我们可以说品种用非零值重载了怪物的健康，用非空字符串重载了攻击字符串。 否则，这些属性要从它的父品种里继承。

实现方式有两种。 一种是每次属性被请求时动态处理委托，就像这样：

```
int Breed::getHealth()
{
  // 重载
  if (health_ != 0 || parent_ == NULL) return health_;

  // 继承
  return parent_->getHealth();
}

const char* Breed::getAttack()
```

```
{
  // 重载
  if (attack_ != NULL || parent_ == NULL) return attack_;

  // 继承
  return parent_->getAttack();
}
```

如果品种在运行时修改种类，不再重载，或者不再继承某些属性时，这能保证做正确的事。另一方面，这要更多的内存（它需要保存指向它的父品种的指针）而且更慢。 每次你查找属性都需要回溯继承链。

如果我们可以保证品种的属性不变，一个更快的解决方案是在构造时使用继承。 这被称为“复制”委托，因为在创建对象时，我们复制继承的属性到推导的类型。它看上去是这样的：

```
Breed(Breed* parent, int health, const char* attack)
: health_(health),
  attack_(attack)
{
  // 继承没有重载的属性
  if (parent != NULL)
  {
    if (health == 0) health_ = parent->getHealth();
    if (attack == NULL) attack_ = parent->getAttack();
  }
}
```

注意现在我们不再需要给父品种的字段了。 一旦构造器完成，我们可以忘了父品种，因为我们已经拷贝了它的所有属性。 为了获得品种的属性，我们现在直接返回字段：

```
int         getHealth() { return health_; }
const char* getAttack() { return attack_; }
```

又好又快！

假设游戏引擎从品种的JSON文件加载设置然后创建类型。它看上去是这样的：

```
{
  "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!"
  }
}

:::json
{
  "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!"
  }
}
```

我们有一段代码读取每个品种，用新数据实例化品种实例。 就像你从`"parent": "Troll"`字段看到的，`Troll Archer`和`Troll Wizard`品种都由基础`Troll`品种继承而来。

由于派生类的初始血量都是0，所以该值从基础`Troll`品种继承。 这意味着无论怎么调整`Troll`的血量，三个品种的血量都会被更新。 随着品种的数量和属性的数量增加，这节约了很多时间。 现在，通过一小块代码，系统给了设计者控制权，让他们能好好利用时间。 与此同时，我们可以回去编码其他特性了。

设计决策

类型对象模式让我们建立类型系统，就好像在设计自己的编程语言。 设计空间是开放的，我们可以做很多有趣的事情。

在实践中，有些东西打破了我们的幻想。 时间和可维护性阻止我们创建特别复杂的东西。更重要的是，无论如何设计类型系统，用户（通常不是程序员）要能轻松地理解它。 我们将其做的越简单，它就越有用。 所以我们在这里谈到的是已经反复探索的领域，开辟新路就留给学者和探索者吧。

类型对象是封装的还是暴露的？

在我们的简单实现中，`Monster`有一个对品种的引用，但是它没有显式暴露这个引用。 外部代码不能直接获取怪物的品种。 从代码库的角度看来，怪物事实上是没有类型的，事实上它们拥有品种只是个实现细节。

我们可以很容易的改变这点，让`Monster`返回它的`Breed`：

```
class Monster
{
public:
```

```
  Breed& getBreed() { return breed_; }

  // 当前的代码……
};
```

> 在本书的另一个例子中，我们遵守了惯例，返还对象的引用而不是对象的指针，保证了永远不会返回NULL。

这样做改变了Monster的设计。 事实是所有怪物都拥有品种是API的可见部分了，下面是这两者各自的好处：

- 如果类型对象是封装的：
  - 类型对象模式的复杂性对代码库的其他部分是隐藏的。 它成为了只有有类型的对象才需要考虑的实现细节。
  - 有类型的对象可以选择性地修改类型对象的重载行为。 假设我们想要怪物在它接近死亡时改变它的攻击字符串。 由于攻击字符串总是通过Monster获取的，我们有一个方便的地方放置代码：

    ```
    const char* Monster::getAttack()
    {
      if (health_ < LOW_HEALTH)
      {
        return "The monster flails weakly.";
      }

      return breed_.getAttack();
    }
    ```

    如果外部代码直接调用品种的getAttack()，我们就没有机会能插入逻辑。
  - 我们得为每个类型对象暴露的方法写转发。 这是这个设计的冗长之处。如果类型对象有很多方法，对象类也得为每一个方法建立属于自己的公共可见方法。
- 如果类型对象是暴露的：
  - 外部代码可以与类型对象直接交互，无需拥有类型对象的实例。 如果类型对象是封装的，那么没有一个拥有它的对象就没法使用它。 这阻止我们使用构造器模式这样的方法，在品种上调用方法来创建新怪物。 如果用户不能直接获得品种，他们就没办法调用它。
  - 类型对象现在是对象公共**API**的一部分了。 大体上，窄接口比宽接口更容易掌控——你暴露给代码库其他部分的越少，你需要处理的复杂度和维护工作就越少。 通过暴露类型对象，我们扩宽了对象的API，包含了所有类型对象提供的东西。

有类型的对象是如何创建的？

使用这个模式，每个“对象”现在都是一对对象：主对象和它的类型对象。 所以我们怎样创建并绑定两者呢？

- 构造对象然后传入类型对象：
  - 外部代码可以控制分配。 由于调用代码也是构建对象的代码，它可以控制其的内存

位置。 如果我们想要UI在多种内存场景中使用（不同的分配器，在栈中，等等），这给了完成它的灵活性。

- 在类型对象上调用“构造器”函数：
  - 类型对象控制了内存分配。 这是硬币的另一面。如果我们不想让用户选择在内存中何处创建对象， 在类型对象上调用工厂方法可以达到这一点。 如果我们想保证所有的对象都来自具体的对象池 或者其他的内存分配器时也有用。

能改变类型吗？

到目前为止，我们假设一旦对象创建并绑定到类型对象上，这永远不会改变。 对象创建时的类型就是它销毁时的类型。这其实没有必要。 我们可以允许对象随着时间改变它的类型。

让我们回想下我们的例子。 当怪物死去时，设计者告诉我们，有时它的尸体会复活成僵尸。 我们可以通过在怪物死亡时产生僵尸类型的新怪兽，但另一个选项是拿到现有的怪物，然后将它的品种改为僵尸。

- 如果类型不改变：
  - 编码和理解都更容易。 在概念上，大多数人不期望“类型”会改变。这符合大多数人的理解。
  - 更容易查找漏洞。 如果我们试图追踪怪物进入奇怪状态时的漏洞，现在 看到的品种就是怪物始终保持的品种可以大大简化工作。
- 如果类型可以改变：
  - 需要创建的对象更少。 在我们的例子中，如果类型不能改变，我们需要消耗CPU循环创建新的僵尸怪物对象， 把原先对象中需要保留的属性都拷贝过来，然后删除它。 如果我们可以改变类型，所有的工作都被一个简单的声明取代。
  - 我们需要小心地做约束。 在对象和它的类型间有强耦合是很自然的事情。 举个例子，一个品种也许假设怪物当前的血量永远高于品种中的初始血量。

    如果我们允许品种改变，我们需要确保已存对象满足新品种的需求。 当我们改变类型时，我们也许需要执行一些验证代码保证对象现在的状态对新类型是有意义的。

它支持何种继承？

- 没有继承：
  - 简单。 最简单的通常是最好的。如果你在类型对象间没有大量数据共享，为什么要为难自己呢？
  - 这会带来重复的工作。 我从未见过看到哪个编码系统中设计者不想要继承的。 当你有十五种不同的精灵时，协调血量就要修改十五处同样的数字真是糟透了。
- 单继承：
  - 还是相对简单。 它易于实现，但是，更重要的是，也易于理解。如果非技术用户正在使用这个系统，要操作的部分越少越好。 这就是很多编程语言只支持单继承的原因。这看起来是能力和简洁之间的平衡点。
  - 查询属性更慢。 为了在类型对象中获取一块数据，我们也许需要回溯继承链寻找是哪一个类型最终决定了值。 在性能攸关的代码上，我们也许不想花时间在这上面。

- 多重继承：
  - 可以避免绝大多数代码重复。 使用优良的多继承系统，用户可以为类型对象建立几乎没有冗余的层次。 改变数值时，我们可以避免很多复制和粘贴。
  - 复杂。 不幸的是，它的好处更多的是理论上的而非实际上的。多重继承很难理解。

    如果僵尸龙继承僵尸和龙，哪些属性来自僵尸，哪些来自于龙？ 为了使用系统，用户需要理解如何继承图的遍历，还需要有设计优秀层次的远见。

    我看到的大多数C++编码标准趋向于禁止多重继承，Java和C#完全移除了它。 这承认了一个悲伤的事实：它太难掌握了，最好根本不要用。 尽管值得考虑，但你很少想要在类型对象上实现多重继承。就像往常一样，简单的总是最好的。

参见

- 这个模式处理的高层问题是在多个对象间分享数据和行为。 另一个用另一种方式解决了相同问题的模式是原型 GoF模式。
- 类型对象是享元 GoF模式的近亲。 两者都让你在实例间分享代码。使用享元，意图是节约内存，而分享的数据也许不代表任何概念上对象的“类型”。 使用类型对象模式，焦点在组织性和灵活性。
- 这个模式和状态 GoF模式有很多相似之处。 两者都委托对象的部分定义给另外一个对象。 通过类型对象，我们通常委托是什么：不变的数据概括描述对象。 通过状态，我们委托现在是什么：暂时描述对象当前状态的数据。

  当我们讨论对象改变它的类型时，你可以将其认为类型对象起到了和状态相似的职责。

解耦模式

游戏设计模式

一旦你掌握了编程语言，编写想要写的东西就会变得相当容易。 困难的是编写适应需求变化的代码，在我们用文本编辑器开火之前，通常没有完美的特性表供我们使用。

能让我们更好适应变化的工具是解耦。 当我们说两块代码“解耦”时，是指修改一块代码一般不会需要修改另一块代码。 当我们修改游戏中的特性时，需要修改的代码越少，就越容易。

组件模式将一个实体拆成多个，解耦不同的领域。 事件序列解耦了两个互相通信的事物，稳定而且及时。 服务定位器让代码使用服务而无需绑定到提供服务的代码。

模式

- 组件模式
- 事件序列
- 服务定位器

组件模式

意图

允许单一的实体跨越多个领域而不会导致这些领域彼此耦合。

动机

让我们假设我们正构建平台跳跃游戏。 意大利水管工已经有人做了，因此我们将出动丹麦面包师，Bj?rn。 照理说，会有一个类来表示友好的糕点厨师，包含他在比赛中做的一切。

> 像这样的游戏创意导致了我是程序员而不是设计师。

由于玩家控制他，这意味着需要读取控制器的输入然后转化为动作。 而且，当然，他需要与关卡相互作用，所以要引入物理和碰撞。 一旦这样做了，他必须在屏幕上出现，所以要引入动画和渲染。 他可能还会播放一些声音。

等一下，这在失控。软件体系结构101课程告诉我们，程序的不同领域应保持分离。 如果我们做一个文字处理器，处理打印的代码不应该受加载和保存文件的代码影响。 游戏和企业应用程序的领域不尽相同，但该规则仍然适用。

我们希望AI，物理，渲染，声音和其他领域域尽可能相互不了解， 但现在我们将所有这一切挤在一个类中。 我们已经看到了这条路通往何处：5000行的巨大代码文件，哪怕是你们团队中最勇敢的程序员也不敢打开。

这工作对能驯服他的少数人是有趣的，但对其他人是地狱。 这么大的类意味着，即使是看似微不足道的变化亦可有深远的影响。 很快，为类添加错误的速度明显快于添加功能的速度。

快刀斩乱麻

比起单纯的规模问题，更糟糕的是耦合。 在游戏中所有不同的系统被绑成了一个巨大的代码球：

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
  playSound(HIT_FLOOR);
```

```
}
```

任何试图改变上面代码的程序员，都需要物理，图形和声音的相关知识，以确保没破坏什么。

> 这样的耦合在任何游戏中出现都是个问题，但是在使用并发的现代游戏中尤其糟糕。 在多核硬件上，让代码同时在多个线程上运行是至关重要的。 将游戏分割到线程的一种通用方法是通过领域划分——在一个核上运行AI代码，在另一个上播放声音，在第三个上渲染，等等。
>
> 一旦你这么做了，在领域间保持解耦就是至关重要的，这是为了避免死锁或者其他恶魔般的并发问题。 如果某个函数从一个线程上调用`UpdateSounds()`方法，从另一个线程上调用`RenderGraphics()`方法，那它是在自找麻烦。

这两个问题互相混合;这个类涉及太多的域，每个程序员都得接触它， 但它又太过巨大，这就变成了一场噩梦。 如果变得够糟糕，程序员会黑入代码库的其他部分，仅仅为了躲开这个像毛球一样的`Bjorn`类。

快刀斩乱麻

我们可以像亚历山大大帝一样解决这个问题——快刀斩乱麻。 按领域将`Bjorn`类切片成相互独立的部分。 例如，我们抽出所有处理用户输入的代码，将其移动到一个单独的`InputComponent`类。 `Bjorn`拥有这个部件的一个实例。我们将对`Bjorn`接触的每个领域重复这一过程。

当完成后，我们将`Bjorn`大多数的东西都抽走了。 剩下的是一个薄壳包着所有的组件。 通过将类划分为多个小类，我们已经解决了这个问题。但我们完成了远远不止这些。

宽松的结果

我们的组件类现在解耦了。 尽管`Bjorn`有`PhysicsComponent`和`GraphicsComponent`，但这两部分都不知道对方的存在。 这意味着处理物理的人可以修改组件而不需要了解图形，反之亦然。

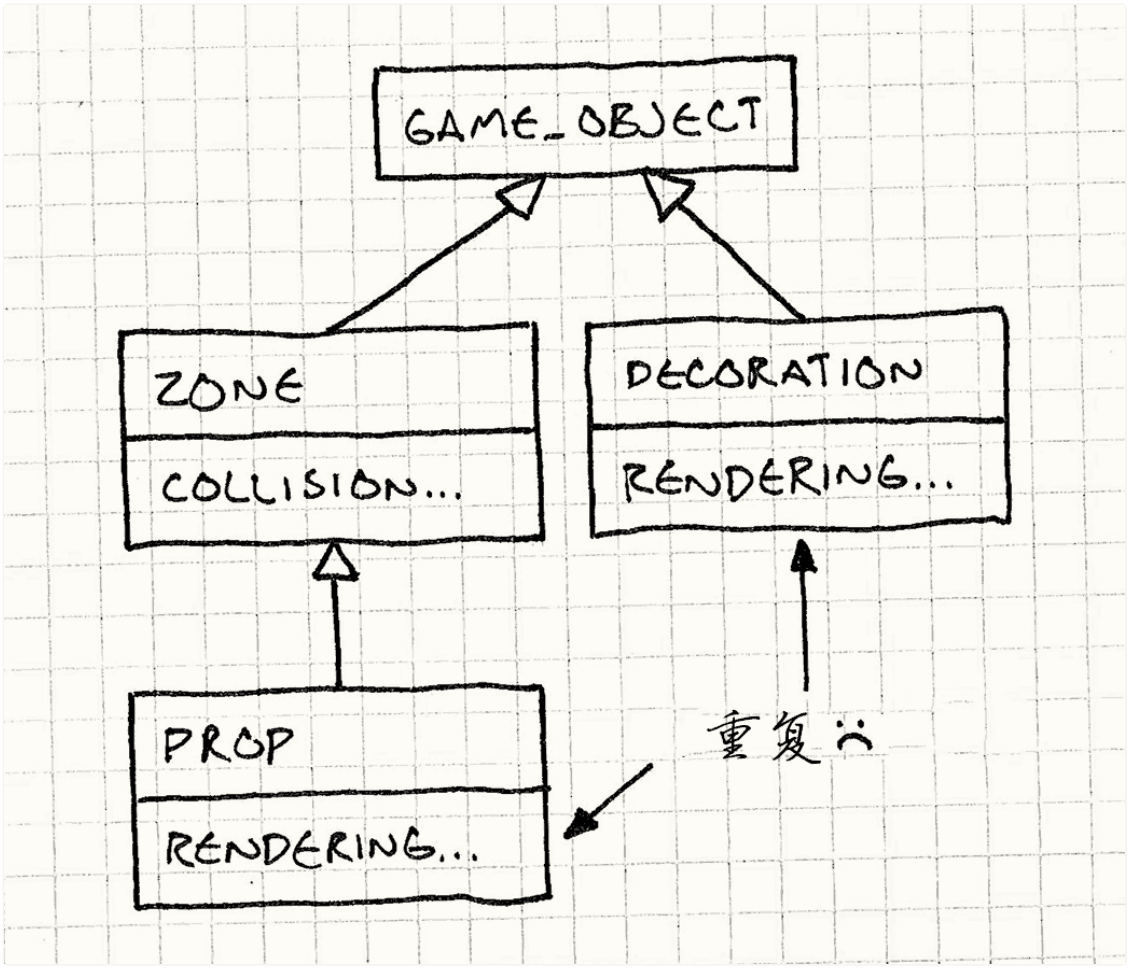
在实践中，这些部件需要在它们之间有一些相互作用。 例如，AI组件可能需要告诉物理组件Bj?rn试图去哪里。 然而，我们可以限制这种交互在确实需要交互的组件之间， 而不是把它们围在同一个围栏里。

绑到一起

这种设计的另一特性是，组件现在是可复用的包。 到目前为止，我们专注于面包师，但是让我们考虑几个游戏世界中其他类型的对象。 装饰 是玩家看到但不能交互的事物：灌木，杂物等视觉细节。 道具 像装饰，但可以交互：箱，巨石，树木。 区域 与装饰相反——无形但可互动。 它们是很好的触发器，比如在Bj?rn进入区域时触发过场动画。

> 当面向对象语言第一次接触这个场景时，继承是它箱子里最闪耀的工具。 它被认为是代码无限重用之锤，编程者常常挥舞着它。 然而我们痛苦地学到，事实上它是一把重锤。 继承有它的用处，但对简单的代码重用太过复杂。
>
> 相反，在今日软件设计的趋势是尽可能使用组件代替继承。 不是让两个类继承同一类来分享代码，而是让它们拥有同一个类的实例。

现在，考虑如果不用组件，我们将如何建立这些类的继承层次。第一遍可能是这样的：



我们有GameObject基类，包含位置和方向之类的通用部分。 Zone继承它，增加了碰撞检测。 同样，Decoration继承GameObject，并增加了渲染。 Prop继承Zone，因此它可以重用碰撞代码。 然而，Prop不能同时继承Decoration来重用渲染， 否则就会造成致命菱形结构。

> “致命菱形”发生在类继承了多个类，而这多个类中有两个继承同一基类时。 介绍它造成的痛苦超过了本书的范围，但它被称为“致命”是有原因的。

我们可以反过来让Prop继承Decoration，但随后不得不重复碰撞检测代码。 无论哪种方式，没有干净的方式重用碰撞和渲染代码而不诉诸多重继承。 唯一的其他选择是一切都继承GameObject， 但随后Zone会浪费内存在并不需要的渲染数据上， Decoration在物理效果上有同样的浪费。

现在，让我们尝试用组件。子类将彻底消失。 取而代之的是一个GameObject类和两个组件类：PhysicsComponent和GraphicsComponent。 装饰是个简单的GameObject，包含GraphicsComponent但没有PhysicsComponent。 区域与其恰好相反，而道具包含两种组件。 没有代码重复，没有多重继承，只有三个类，而不是四个。

> 可以拿饭店菜单打比方。如果每个实体是一个类，那就只能订套餐。 我们需要为每种可能的组合定义各自的类。 为了满足每位用户，我们需要十几种套餐。
>
> 组件是照单点菜——每位顾客都可以选他们想要的，菜单记录可选的菜式。

对对象而言，组件是即插即用的。 将不同的可重用部件插入对象，我们就能构建复杂具有丰富的行为实体。 就像软件中的战神金刚。

模式

单一实体跨越了多个领域。为了保持领域之间相互分离，将每部分代码放入各自的组件类中。 实体被简化为组件的容器。

> “组件”，就像“对象”，在编程中意味任何东西也不意味任何东西。 正因如此，它被用来描述一些概念。 在商业软件中，“组件”设计模式描述通过网络解耦的服务。
>
> 我试图从游戏中找到无关这个设计模式的另一个名字，但“组件”看来是最常用的术语。 由于设计模式是关于记录已存的实践，我没有创建新术语的余地。 所以，跟着XNA，Delta3D和其他人的脚步，我称之为“组件”。

何时使用

组件通常在定义游戏实体的核心部分中使用，但它们在其他地方也有用。 这个模式在如下情况中可以使用：

- 有一个涉及了多个领域的类，而你想保持这些领域互相隔离。
- 一个类正在变大而且越来越难以使用。
- 想要能定义一系列分享不同能力的类，但是使用继承无法让你精确选取要重用的部分。

记住

组件模式比简单地向类中添加代码增加了一点点复杂性。 每个概念上的“对象”要组成真正对象需要实例化，初始化，然后正确的连接。 不同组件间沟通会有些困难，而控制它们如何使用内存就更加复杂。

对于大型代码库，为了解耦和重用而付出这样的复杂度是值得的。 但是在使用这种模式之前，保证你没有为了不存在的问题而“过度设计”。

使用组件的另一后果是，需要多一层跳转才能做要做的事。 拿到容器对象，获得相应的组件，然后 你才能做想做的事情。 在性能攸关的内部循环中，这种跳转也许会导致糟糕的性能。

> 这是硬币的两面。组件模式通常可以增进性能和缓存一致性。 组件让使用数据局部性 模式的CPU更容易地组织数据。

示例代码

我写这本书的最大挑战之一就是搞明白如何隔离各个模式。 许多设计模式包含了不属于这种模式的代码。 为了将提取模式的本质，我尽可能的消减代码， 但是在某种程度上，这就像没有衣服还要说明如何整理衣柜。

说明组件模式尤其困难。 如果看不到它解耦的各个领域的代码，你就不能获得正确的体会，因此我会多写一些有关于Bj?rn的代码。 这个模式事实上只关于将组件变为类，但类中的代码可以帮助表明类是做什么用的。 它是伪代码——它调用了其他不存在的类——但这应该

可以让你理解我们正在做什么。

单块类

为了清晰的看到这个模式是如何应用的， 我们先展示一个Bjorn类， 它包含了所有我们需要的事物，但是没有使用这个模式：

> 我应指出在代码使用角色的名字总是个坏主意。市场部有在发售之前改名字的坏习惯。 “焦点测试表明，在11岁到15岁的男性不喜欢‘Bj?rn’，请改为‘Sven&rsquo”。
>
> 这就是为什么很多软件项目使用内部代码名。 而且比起告诉人们你在完成“Photoshop的下一版本”，告诉他们你在完成“大电猫”更有趣。

```
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_;
};
```

Bjorn有个每帧调用的update()方法。

```
void Bjorn::update(World& world, Graphics& graphics)
{
  // 根据用户输入修改英雄的速度
  switch (Controller::getJoystickDirection())
  {
    case DIR_LEFT:
      velocity_ -= WALK_ACCELERATION;
      break;

    case DIR_RIGHT:
      velocity_ += WALK_ACCELERATION;
```

```
      break;
  }

  // 根据速度修改位置
  x_ += velocity_;
  world.resolveCollision(volume_, x_, y_, velocity_);

  // 绘制合适的图形
  Sprite* sprite = &spriteStand_;
  if (velocity_ < 0)
  {
    sprite = &spriteWalkLeft_;
  }
  else if (velocity_ > 0)
  {
    sprite = &spriteWalkRight_;
  }

  graphics.draw(*sprite, x_, y_);
}
```

它读取操纵杆以确定如何加速面包师。 然后，用物理引擎解析新位置。 最后，将Bj?rn渲染至屏幕。

这里的示例实现平凡而简单。 没有重力，动画，或任何让人物有趣的其他细节。 即便如此，我们可以看到，已经出现了同时消耗多个程序员时间的函数，而它开始变得有点混乱。 想象增加到一千行，你就知道这会有多难受了。

分离领域

从一个领域开始，将Bjorn的代码去除一部分，归入分离的组件类。 我们从首个执行的领域开始：输入。 Bjorn做的头件事就是读取玩家的输入，然后基于此调整它的速度。 让我们将这部分逻辑移入一个分离的类：

```
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;
};
```

很简单吧。我们将Bjorn的update()的第一部分取出，放入这个类中。 对Bjorn的改变也很直接：

```
class Bjorn
{
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics)
  {
    input_.update(*this);

    // 根据速度修改位置
    x += velocity;
    world.resolveCollision(volume_, x, y, velocity);

    // 绘制合适的图形
    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_;
};
```

Bjorn现在拥有了一个InputComponent对象。 之前它在update()方法中直接处理用户输入，现在委托给组件：

```
input_.update(*this);
```

我们才刚开始，但已经摆脱了一些耦合——Bjorn主体现在已经与Controller无关了。这会派上用场的。

将剩下的分割出来

现在让我们对物理和图像代码继续这种剪切粘贴的工作。 这是我们新的 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_;
};
```

为了将物理行为移出Bjorn类，你可以看到我们也移出了数据：Volume对象已经是组件的一部分了。

最后，这是现在的渲染代码：

```
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_;
};
```

我们几乎将所有的东西都移出来了，所以面包师还剩下什么？没什么了：

```
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_;
};
```

Bjorn类现在基本上就做两件事：拥有定义它的组件，以及在不同域间分享的数据。 有两个原因导致位置和速度仍然在Bjorn的核心类中： 首先，它们是“泛领域”状态——几乎每个组件都需要使用它们， 所以我们想要提取它出来时，哪个组件应该拥有它们并不明确。

第二，也是更重要的一点，它给了我们无需让组件耦合就能沟通的简易方法。 让我们看看能不能利用这一点。

机器人Bj?rn

到目前为止，我们将行为归入了不同的组件类，但还没将行为抽象出来。 Bjorn仍知道每个类的具体定义的行为。让我们改变这一点。

取出处理输入的部件，将其藏在接口之后，将InputComponent变为抽象基类。

```
class InputComponent
{
public:
  virtual ~InputComponent() {}
  virtual void update(Bjorn& bjorn) = 0;
};
```

然后，将现有的处理输入的代码取出，放进一个实现接口的类中。

```
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;
};
```

我们将Bjorn改为只拥有一个指向输入组件的指针，而不是拥有一个内联的实例。

```
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_;
};
```

现在当我们实例化Bjorn，我们可以传入输入组件使用，就像下面这样：

```
Bjorn* bjorn = new Bjorn(new PlayerInputComponent());
```

这个实例可以是任何实现了抽象InputComponent接口的类型。 我们为此付出了代价——update()现在是虚方法调用了，这会慢一些。这一代价的回报是什么？

大多数的主机需要游戏支持“演示模式”。 如果玩家停在主菜单没有做任何事情，游戏就会自动开始运行，直到接入一个玩家。 这让屏幕上的主菜单看上去更有生机，同时也是销售商店里很好的展示。

隐藏在输入组件后的类帮我们实现了这点， 我们已经有了具体的PlayerInputComponen

t供在玩游戏时使用。 现在让我们完成另一个：

```
class DemoInputComponent : public InputComponent
{
public:
  virtual void update(Bjorn& bjorn)
  {
    // 自动控制Bjorn的AI……
  }
};
```

当游戏进入演示模式，不像之前演示的那样构造Bj?rn， 我们将它和一个新组件连接起来：

```
Bjorn* bjorn = new Bjorn(new DemoInputComponent());
```

现在，只需要更改组件，我们就有了为演示模式而设计的电脑控制的玩家。 我们可以重用所有Bj?rn的代码——物理和图像都不知道这里有了变化。 也许我有些奇怪，但这就是每天能让我从起床的事物。

> 那个，还有咖啡。热气腾腾的咖啡。

删掉Bj?rn？

如果你看看现在的Bj?rn类，你会意识到那里完全没有“Bj?rn”——那只是个组件包。 事实上，它是个好候选人，能够作为每个游戏中的对象都能继承的“游戏对象”基类。 我们可以像弗兰肯斯坦一样，通过挑选拼装部件构建任何对象。

让我们将剩下的两个具体组件——物理和图像——像输入那样藏到接口之后。

```
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;
};
```

然后将Bj?rn改为使用这些接口的通用GameObject类。

```
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_;
};
```

有些人走的更远。 不使用包含组件的GameObject，游戏实体只是一个ID，一个数字。 每个组件都知道它们连接的实体ID，然后管理分离的组件。

这些实体组件系统将组件发挥到了极致，让你向实体添加组件而无需通知实体。 数据局部性一章有更多细节。

我们现有的具体类被重命名并实现这些接口：

```
class BjornPhysicsComponent : public PhysicsComponent
{
public:
  virtual void update(GameObject& obj, World& world)
  {
    // 物理代码……
  }
};

class BjornGraphicsComponent : public GraphicsComponent
{
public:
  virtual void update(GameObject& obj, Graphics& graphics)
  {
    // 图形代码……
  }
};
```

现在我们无需为Bj?rn建立具体类，就能构建拥有所有Bj?rn行为的对象。

```
GameObject* createBjorn()
```

```
{
  return new GameObject(new PlayerInputComponent(),
                        new BjornPhysicsComponent(),
                        new BjornGraphicsComponent());
}
```

这个`createBjorn()`函数是，当然，经典GoF工厂模式[GoF]的例子。

通过用不同组件实例化`GameObject`，我们可以构建游戏需要的任何对象。

设计决策

这章中你最需要回答的设计问题是“我需要什么样的组件？” 回答取决于你游戏的需求和风格。 引擎越大越复杂，你就想将组件划分得更细。

除此之外，还有几个更具体的选项要回答：

对象如何获取组件？

一旦将单块对象分割为多个分离的组件，就需要决定谁将它们拼到一起。

- 如果对象创建组件：
  - 这保证了对象总是能拿到需要的组件。 你永远不必担心某人忘记连接正确的组件然后破坏了整个游戏。容器类自己会处理这个问题。
  - 重新设置对象比较困难。 这个模式的强力特性之一就是只需重新组合组件就可以创建新的对象。 如果对象总是用硬编码的组件组装自己，我们就无法利用这个特性。
- 如果外部代码提供组件：
  - 对象更加灵活。 我们可以提供不同的组件，这样就能改变对象的行为。 通过共用组件，对象变成了组件容器，我们可以为不同目的一遍又一遍重用它。
  - 对象可以与具体的组件类型解耦。 如果我们允许外部代码提供组件，好处是也可以传递派生的组件类型。 这样，对象只知道组件接口而不知道组件的具体类型。这是一个很好的封装结构。

组件之间如何通信？

完美解耦的组件不需要考虑这个问题，但在真正的实践中行不通。 事实上组件属于同一对象暗示它们属于需要相互协同的更大整体的一部分。 这就意味着通信。

所以组件如何相互通信呢？ 这里有很多选项，但不像这本书中其他的“选项”，它们相互并不冲突——你可以在一个设计中支持多种方案。

- 通过修改容器对象的状态：
  - 这保持组件解耦。 当我们的`InputComponent`设置了Bj?rn的速度，而后`PhysicsComponent`使用它， 这两个组件都不知道对方的存在。在它们的理解中，Bj?rn的速度是被黑魔法改变的。
  - 需要将组件分享的任何数据存储在容器类中。 通常状态只在几个组件间共享。比如，动画组件和渲染组件需要共享图形专用的信息。 将信息存入容器类会让所有组件都获得这样的信息。

更糟的是，如果我们为不同组件配置使用相同的容器类，最终会浪费内存存储不被任何对象组件需要的状态。 如果我们将渲染专用的数据放入容器对象中，任何隐形对象都会无益的消耗内存。

- 这让组件的通信基于组件运行的顺序。 在同样的代码中，原先一整块的`update()`代码小心地排列这些操作。 玩家的输入修改了速度，速度被物理代码使用并修改位置，位置被渲染代码使用将Bj?rn绘到该有的地方。 当我们将这些代码划入组件时，还是得小心翼翼的保持这种操作顺序。

  如果我们不那么做，就引入了 微妙而难以追踪的漏洞。 比如，我们先更新图形组件，就错误地将Bj?rn渲染在他上一帧而不是这一帧所处的位置上。 如果你考虑更多的组件和更多的代码，那你可以想象要避免这样的错误有多么困难了。

  > 大量这样共享状态的代码很难正确完成读写的相同数据。 这就是为什么学术界花时间研究完全函数式语言，比如Haskell，那里根本没有可变状态。

- 通过它们之间相互引用：

  这里的思路是组件有要交流的组件的引用，这样它们直接交流，无需通过容器类。

  假设我们想让Bj?rn跳跃。图形代码想知道它需要用跳跃图像还是不用。 这可以通过询问物理引擎它当前是否在地上来确定。一种简单的方式是图形组件直接知道物理组件的存在：

```
class BjornGraphicsComponent
{
public:
  BjornGraphicsComponent(BjornPhysicsComponent* physics)
  : physics_(physics)
  {}

  void Update(GameObject& obj, Graphics& graphics)
  {
    Sprite* sprite;
    if (!physics_->isOnGround())
    {
      sprite = &spriteJump_;
    }
    else
    {
      // 现存的图形代码……
    }

    graphics.draw(*sprite, obj.x, obj.y);
  }

private:
  BjornPhysicsComponent* physics_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
```

```
  Sprite spriteJump_;
};
```

当构建Bj?rn的GraphicsComponent时，我们给它相应的PhysicsComponent引用。

- 简单快捷。 通信是一个对象到另一个的直接方法调用。组件可以调用任一引用对象的方法。做什么都可以。
- 两个组件紧绑在一起。 这是什么都可以的坏处。我们向使用整块类又退回了一步。这比只用单一类好一点，至少我们现在只是把需要通信的类绑在一起。

- 通过发送消息：
  - 这是最复杂的选项。我们可以在容器类中建小小的消息系统，允许组件相互发送消息。

    下面是一种可能的实现。我们从每个组件都会实现的Component接口开始：

    ```
    class Component
    {
    public:
      virtual ~Component() {}
      virtual void receive(int message) = 0;
    };
    ```

    它有一个简单的receive()方法，每个需要接受消息的组件类都要实现它。 这里，我们使用一个int来定义消息，但更完整的消息实现应该可以附加数据。

    然后，向容器类添加发送消息的方法。

    ```
    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];
    };
    ```

    现在，如果组件能够接触容器，它就能向容器发送消息，直接向所有的组件广播。（包括了原先发送消息的组件，小心别陷入消息的无限循环中！）这会造成一些结果：

> 如果你真的乐意，甚至可以将消息存储在队列中，晚些发送。 要知道更多，看看事件队列[queue]。

- 同级组件解耦。 通过父级容器对象，就像共享状态的方案一样，我们保证了组件之间仍然是解耦的。 使用了这套系统，组件之间唯一的耦合是它们发送的消息。

  > GoF称之为中介[GoF]模式——两个或更多的对象通过中介对象通信。 现在这种情况下，容器对象本身就是中介。

- 容器类很简单。 不像使用共享状态那样，容器类无需知道组件使用了什么数据，它只是将消息发送出去。 这可以让组件发送领域特有的数据而无需打扰容器对象。

不出意料的，这里没有最好的答案。这些方法你最终可能都会使用一些。 共享状态对于每个对象都有的数据是很好用的——比如位置和大小。

有些不同领域仍然紧密相关。想想动画和渲染，输入和AI，或物理和粒子。 如果你有这样一对分离的组件，你会发现直接相互引用也许更加容易。

消息对于“不那么重要”的通信很有用。 发送后不管的特性对物理组件发现事物碰撞后发送消息，让音乐组件播放声音这种事情是很有效的。

就像以前一样，我建议你从简单的开始，然后如果需要的话，加入其他的通信路径。

See Also

参见

- Unity核心架构中`GameObject`类完全根据这样的原则设计components。
- 开源的Delta3D引擎有`GameActor`基类通过`ActorComponent`实现了这种模式。
- 微软的XNA游戏框架有一个核心的`Game`类。它拥有一系列`GameComponent`对象。我们在游戏实体层使用组件，XNA在游戏主对象上实现了这种模式，但意图是一样的。
- 这种模式与GoF的策略模式[GoF]类似。 两种模式都是将对象的行为取出，划入单独的重述对象。 与对象模式不同的是，分离的策略模式通常是无状态的——它封装了算法，而没有数据。 它定义了对象如何行动，但没有定义对象是什么。

  组件更加重要。它们经常保存了对象的状态，这有助于确定其真正的身份。 但是，这条界限很模糊。有一些组件也许根本没有任何状态。 在这种情况下，你可以在不同的容器对象中使用相同的组件实例。这样看来，它的行为确实更像一种策略。

事件队列

意图

解耦发出消息或事件的时间和处理它的时间。

动机

除非还呆在一两个没有互联网接入的犄角旮旯，否则你很可能已经听说过“事件序列”了。 如果没有，也许“消息队列”或“事件循环”或“消息泵”可以让你想起些什么。 为了唤醒你的记忆，让我们了解几个此模式的常见应用吧。

> 这章的大部分里，我交替使用“事件”和“消息”。 在两者的意义有区别时，我会表明的。

GUI事件循环

如果你曾做过任何用户界面编程，你就会很熟悉事件。 每当用户与你的程序交互——点击按钮，拉出菜单，或者按个键——操作系统就会生成一个事件。 它会将这个对象扔给你的应用程序，你的工作就是获取它然后将其与有趣的行为相挂钩。

> 这个程序风格非常普遍，被认为是一种编程范式：事件驱动编程。

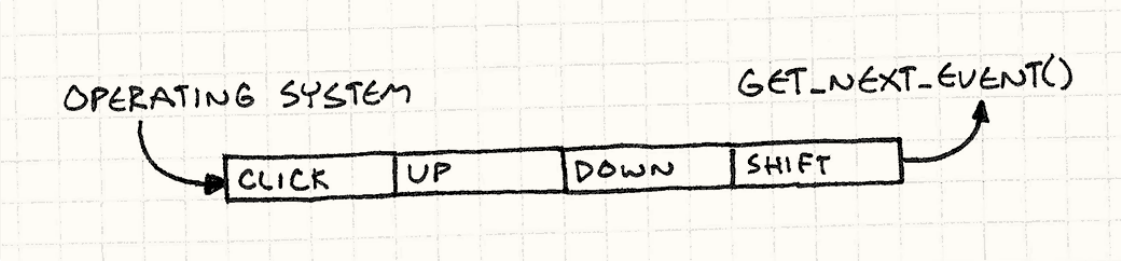
为了获取这些事件，代码底层是事件循环。它大体上是这样的：

```
while (running)
{
  Event event = getNextEvent();
  // 处理事件……
}
```

对`getNextEvent()`的调用将一堆未处理的用户输出传到应用程序中。 你将它导向事件处理器，之后应用魔术般获得了生命。 有趣的部分是应用在它想要的时候获取事件。 操作系统在用户操作时不是直接跳转到你应用的某处代码。

> 相反，操作系统的中断确实是直接跳转的。 当中断发生时，操作系统中断应用在做的事，强制它跳到中断处理。 这种唐突的做法是中断很难使用的原因。

这就意味着当用户输入进来时，它需要到某处去， 这样操作系统在设备驱动报告输入和应用去调用`getNextEvent()`之间不会漏掉它。 这个“某处”是一个队列。



当用户输入抵达时，操作系统将其添加到未处理事件的队列中。 当你调用`getNextEvent()`时，它从队列中获取最旧的事件然后交给应用程序。

中心事件总线

大多数游戏不是像这样事件驱动的，但是在游戏中使用事件循环来支撑中枢系统是很常见的。 你通常听到用“中心”“全局”“主体”描述它。 它通常被用于想要相互保持解耦的高层模块间通信。

> 如果你想知道为什么它们不是事件驱动的，看看游戏循环一章。

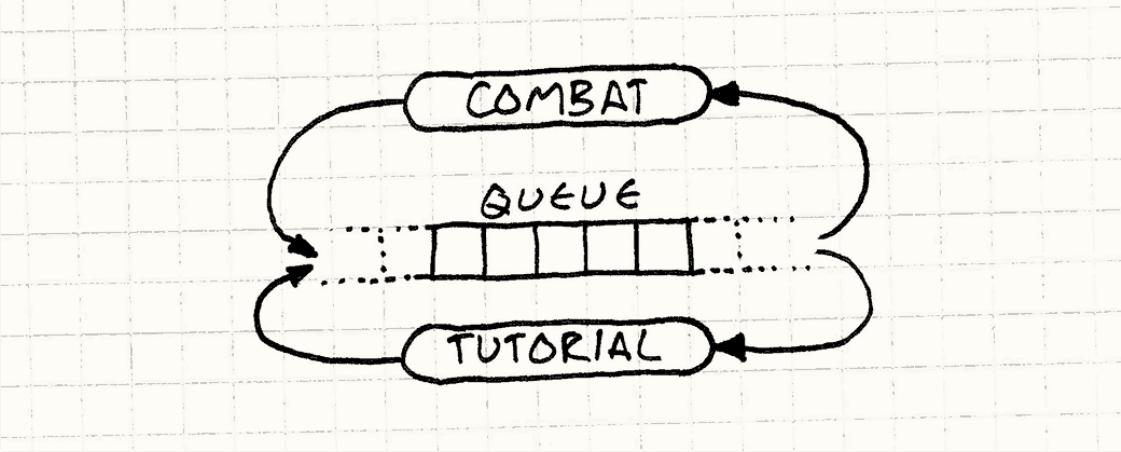
假设游戏有新手教程系统，在某些特定游戏事件后显示帮助框。 举个例子，当玩家第一次击败了邪恶野兽，你想要一个显示着“按X拿起战利品！”的小气泡。

> 新手教程系统很难优雅地实现，大多数玩家很少使用游戏内的帮助，所以这感觉上吃力不讨好。 但对那些使用教程的玩家，这是无价之物。

游戏玩法和战斗代码也许像上面一样复杂。 你最不想做的就是检查一堆教程的触发器。 相反，你可以使用中心事件队列。 任何游戏系统都可以发事件给队列，这样战斗代码可以在砍倒敌人时发出“敌人死亡”事件。

类似的，任何游戏系统都能从队列接受事件。 教程引擎在队列中注册自己，然后表明它想要收到“敌人死亡”事件。 用这种方式，敌人死了的消息从战斗系统传到了教程引擎，而不需要这两个系统直接知道对方的存在。

> 实体可以发送和收到消息的模型很像AI界的blackboard systems。



我本想将这个作为这章其他部分的例子，但是我真不喜欢这样巨大的全局系统。 事件队列不需要在整个游戏引擎中沟通。在一个类或者领域中沟通就足够有用了。

你说什么？

所以说点别的，让我们给游戏添加一些声音。 人类是视觉动物，但是听觉强烈影响到情感系统和空间感觉。 正确模拟的回声可以让漆黑的屏幕感觉上是巨大的洞穴，而适时的小提琴慢板可以让心弦拉响同样的旋律。

为了获得优秀的音效表现，我们从最简单的解决方法开始，看看结果如何。 添加一个“声音引擎”，其中有使用标识符和音量就可以播放音乐的API：

> 我总是离单例模式 GoF远远的。 这是少数它可以使用的领域，因为机器通常只有一个声源系统。 我使用更简单的方法，直接将方法定为静态。

```
class Audio
{
public:
  static void playSound(SoundId id, int volume);
};
```

它负责加载合适的声音资源，找到可靠的播放频道，然后启动它。 这章不是关于某个平台真实的音频API，所以我会假设在其他某处魔术般实现了一个。 使用它，我们像这样写方法：

```
void Audio::playSound(SoundId id, int volume)
{
  ResourceId resource = loadSound(id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, volume);
}
```

我们签入以上代码，创建一些声音文件，然后在代码中加入一些对`playSound()`的调用。举个例子，在UI代码中，我们在选择菜单项变化时播放一点小音效：

```
class Menu
{
public:
  void onSelect(int index)
  {
    Audio::playSound(SOUND_BLOOP, VOL_MAX);
    // 其他代码……
  }
};
```

这样做了之后，注意到有时候你改变菜单项目，整个屏幕就会冻住几帧。 我们遇到了第一个问题：

- 问题一：**API**在音频引擎完成对请求的处理前阻塞了调用者。

我们的playSound()方法是同步的——它在从播放器放出声音前不会返回调用者。 如果声音文件要从光盘上加载，那就得花费一定时间。 与此同时，游戏的其他部分被卡住了。

现在忽视这一点，我们继续。 在AI代码中，我们增加了一个调用，在敌人承受玩家伤害时发出痛苦的低号。 没有什么比在虚拟的生物身上施加痛苦更能温暖玩家心灵的了。

这能行，但是有时玩家打出暴击，他在同一帧可打到两个敌人。 这让游戏同时要播放两遍哀嚎。 如果你了解一些音频的知识，那么就知道要把两个不同的声音混合在一起，就要加和它们的波形。 当这两个是同一波形时，它与一个声音播放两倍响是一样的。那会很刺耳。

> 我在完成Henry Hatsworth in the Puzzling Adventure遇到了同样的问题。解决方法和这里的很相似。

在Boss战中有个相关的问题，当有一堆小怪跑动制造伤害时。 硬件只能同时播放一定数量的音频。当数量超过限度时，声音就被忽视或者切断了。

为了处理这些问题，我们需要获得音频调用的整个集合，用来整合和排序。 不幸的是，音频API独立处理每一个playSound()调用。 看起来这些请求是从针眼穿过一样，一次只能有一个。

- 问题二：请求无法合并处理。

这个问题与下面的问题相比只是小烦恼。 现在，我们在很多不同的游戏系统中散布了playSound()调用。 但是游戏引擎是在现代多核机器上运行的。 为了使用多核带来的优势，我们将系统分散在不同线程上——渲染在一个，AI在另一个，诸如此类。

由于我们的API是同步的，它在调用者的线程上运行。 当从不同的游戏系统调用时，我们从多个线程同时使用API。 看看示例代码，看到任何线程同步性吗？我也没看到。

当我们想要分配一个单独的线程给音频，这个问题就更加严重。 当其他线程都忙于互相跟随和制造事物，它只是傻傻待在那里。

- 问题三：请求在错误的线程上执行。

音频引擎调用playSound()意味着，“放下任何东西，现在就播放声音！”立即 就是问题。游戏系统在它们方便时调用playSound()，但是音频引擎不一定能方便去处理这个请求。为了解决这点，我们需要将接受请求和处理请求解耦。

模式

事件队列在队列中按先入先出的顺序存储一系列通知或请求。 发送通知时，将请求放入队列并返回。 处理请求的系统之后稍晚从队列中获取请求并处理。 这解耦了发送者和接收者，既静态又及时。

何时使用

如果你只是想解耦接收者和发送者，像观察者模式 和命令模式都可以用较小的复杂度进行处理。 在解耦某些需要及时处理的东西时使用队列。

> 我在之前的几乎每章都提到了，但这值得反复提。 复杂度会拖慢你，所以要将简单视为珍贵的财宝。

用推和拉来考虑。 有一块代码A需要另一块代码B去做些事情。 对A自然的处理方式是将请求推给B。

同时，对B自然的处理方式是在**B**方便时将请求拉入。 当一端有推模型另一端有拉模型，你需要在它们间放缓存。 这就是队列比简单的解耦模式多提供的部分。

队列给了代码对拉取的控制权——接收者可以延迟处理，合并或者忽视请求。 但队列做这些事是通过将控制权从发送者那里拿走完成的。 发送者能做的就是向队列发送请求然后祈祷。 当发送者需要回复时，队列不是好选择。

记住

不像这书中的其他模式，事件队列很复杂，会对游戏架构引起广泛影响。 这就意味着你仔细考虑如何——或者要不要——使用它。

中心事件队列是一个全局变量

这个模式的常用方法是一个大的交换站，游戏中的每个部分都能将消息送过这里。 这是很有用的基础架构，但是有用并不代表好用。

可能要走一些弯路，但是我们中的大多数最终学到了全局变量是不好的。 当有一小片状态，程序的每部分都能接触到，会产生各种微妙的相关性。 这个模式将状态封装在协议中，但是它还是全局的，仍然有全局变量引发的全部危险。

世界的状态可以因你改变

假设在虚拟的小怪结束它一生时，一些AI代码将“实体死亡”事件发送到队列中。 这个事件在队列中等待了谁知有多少帧后才排到了前面，得以处理。

同时，经验系统想要追踪英雄的杀敌数，并对他的效率加以奖励。 它接受每个“实体死亡”事件，然后决定英雄击杀了何种怪物，以及击杀的难易程度，最终计算出合适的奖励。

这需要游戏世界的多种不同状态。 我们需要死亡的实体以获取击杀它的难度。 我们也许要看看英雄的周围有什么其他的障碍物或者怪物。 但是如果事件没有及时处理，这些东西都会消失。 实体可能被清除，周围的东西也有可能移开。

当你接到事件时，得小心，不能假设现在的状态反映了事件发生时的世界。 这就意味着队列中的事件比同步系统中的事件需要存储更多数据。 在后者中，通知只需说“某事发生了”然后接收者可以找到细节。 使用队列时，这些短暂的细节必须在事件发送时就被捕获，以方便之后使用。

会陷于反馈系统环路中

任何事件系统和消息系统都得担心环路：

1. A发送了一个事件
2. B接收然后发送事件作为回应。
3. 这个事件恰好是A关注的，所以它收到了。为了回应，它发送了一个事件。
4. 回到2.

当消息系统是同步的，你很快就能找到环路——它们造成了栈溢出并让游戏崩溃。 使用队列，异步地使用栈，即使虚假事件晃来晃去时，游戏仍然可以继续运行。 避免这个的通用方法就是避免在处理事件的代码中发送事件。

> 在你的事件系统中加一个小小的漏洞日志也是一个好主意。

示例代码

我们已经看到一些代码了。它不完美，但是有基本的正确功能——公用的API和正确的底层音频调用。 剩下需要做的就是修复它的问题。

第一个问题是我们的API是阻塞的。 当代码播放声音时，它不能做任何其他事情，直到`playSound()`加载完音频然后真正地开始播放。

我们想要推迟这项工作，这样 `playSound()` 可以很快的返回。 为了达到这一点，我们需要具体化播放声音的请求。 我们需要一个小结构存储发送请求时的细节，这样我们晚些时候可以使用：

```
struct PlayMessage
{
  SoundId id;
  int volume;
};
```

下面我们需要给`Audio`一些存储空间来追踪正在播放的声音。 现在，你的算法专家也许会告诉你使用激动人心的数据结构， 比如Fibonacci heap或者skip list或者最起码链表。 但是在实践中，存储一堆同类事物最好的办法是使用一个平凡无奇的经典数组：

> 算法研究者通过发表新奇数据结构的研究获得收入。 他们不鼓励使用基本的结构。

- 没有动态分配。
- 没有为记录信息造成额外的开销或者多余的指针。
- 对缓存友好的连续存储空间。

> 更多“缓存友好”的内容，见数据局部性一章。

所以让我们开干吧：

```
class Audio
{
public:
  static void init()
  {
    numPending_ = 0;
  }

  // 其他代码……
private:
  static const int MAX_PENDING = 16;

  static PlayMessage pending_[MAX_PENDING];
```

```
  static int numPending_;
};
```

我们可以将数组大小设置为最糟情况下的大小。 为了播放声音，简单的将新消息插到最后：

```
void Audio::playSound(SoundId id, int volume)
{
  assert(numPending_ < MAX_PENDING);

  pending_[numPending_].id = id;
  pending_[numPending_].volume = volume;
  numPending_++;
}
```

这让`playSound()`几乎是立即返回，当然我们仍得播放声音。 那块代码在某处，即`update()`方法中：

```
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;
  }

  // 其他代码……
};
```

> 就像名字暗示的，这是更新方法模式。

现在我们需要从方便时候调用。 这个“方便”取决于你的游戏。 它也许要从主游戏循环中或者专注于音频的线程中调用。

这可行，但是这假定了我们在对`update()`的单一调用中可以处理每个声音请求。 如果你做了像在声音资源加载后处理异步请求的事情，这就没法工作了。 `update()`一次处理一个请求，它需要有完成一个请求后从缓存中再拉取一个请求的能力。 换言之，我们需要一个真实的队列。

环状缓存

有很多种方式能实现队列，但我最喜欢的是环状缓存。 它保留了数组的所有优点，同时能

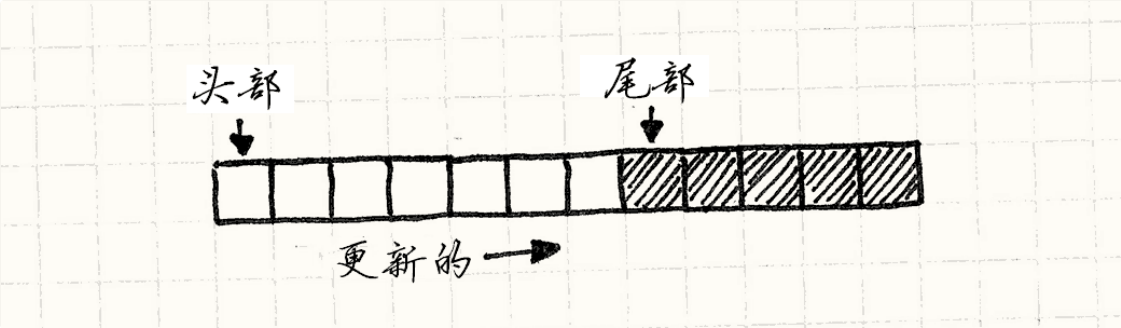
让我们不断从队列的前方移除事物。

现在，我知道你在想什么。 如果我们从数组的前方移除东西，不是需要将所有剩下的部分都移动一次吗？这不是很慢吗？

这就是为什么要学习链表——你可以从中移除一个节点，而无需移动东西。 好吧，其实你可以用数组实现一个队列而无需移动东西。 我会展示给你看，但是首先预习一些术语：

- 队列的头部是读取请求的地方。头部存储最早发出的请求。
- 尾部 是另一端。它是数组中下个写入请求的地方。注意它指向队列终点的下一个。你可以将其理解为一个半开半闭区间，如果这有帮助的话。

由于 `playSound()` 向数组的末尾添加了新的请求，头部开始指向元素0而尾部向右增长。



让我们开始编码。首先，我们显式定义这两个标记在类中的意义：

```
class Audio
{
public:
  static void init()
  {
    head_ = 0;
    tail_ = 0;
  }

  // 方法……
private:
  static int head_;
  static int tail_;

  // 数组……
};
```

在 `playSound()` 的实现中，`numPending_`被`tail_`取代，但是其他都是一样的：

```
void Audio::playSound(SoundId id, int volume)
{
  assert(tail_ < MAX_PENDING);

  // Add to the end of the list.
  pending_[tail_].id = id;
  pending_[tail_].volume = volume;
  tail_++;
```

```
}
```

更有趣的变化在`update()`中：

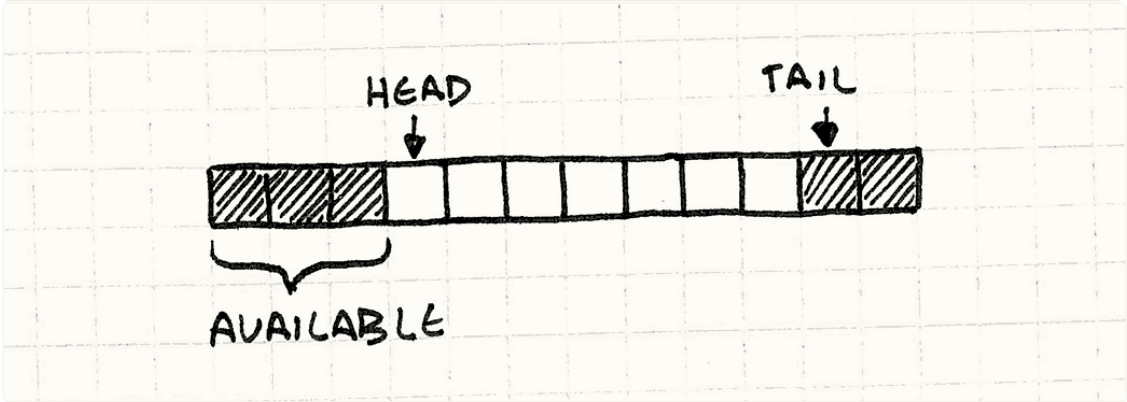
```
void Audio::update()
{
  // 如果这里没有待处理的请求
  // 那就什么也不做。
  if (head_ == tail_) return;

  ResourceId resource = loadSound(pending_[head_].id);
  int channel = findOpenChannel();
  if (channel == -1) return;
  startSound(resource, channel, pending_[head_].volume);

  head_++;
}
```

我们在头部处理，然后通过将头部指针向右移动来消除它。 我们定义头尾之间没有距离的队列为空队列。

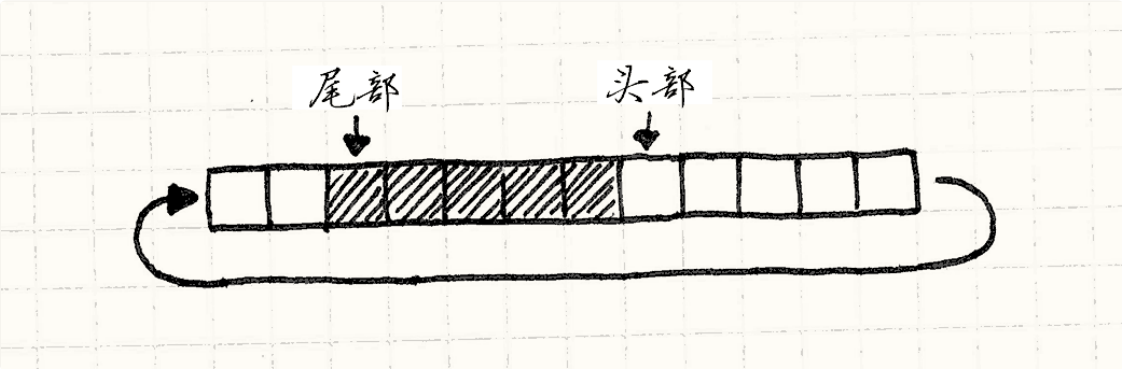
> 这就是为什么我们让尾部指向最后元素之后的那个位置。 这意味着头尾相等则队列为空。

现在，我们获得了一个队列——我们可以向尾部添加元素，从头部移除元素。 这里有很明显的问题。在我们让队列跑起来后，头部和尾部继续向右移动。 最终`tail_`碰到了数组的尾部，欢乐时光结束了。 接下来是这个方法的灵巧之处。

> 你想结束欢乐时光吗？不，你不想。



注意当尾部移动时，头部 也是如此。 这就意味着在数组开始部分的元素不再被使用了。 所以我们做的就是，当抵达末尾时，将尾部回折到数组的头部。 这就是为什么它被称为环状缓存，它表现的像是一个环状的数组。



这个的实现非常简单。 当我们入队一个事物时，只需要保证尾部在抵达结束的时候回折到数组的开头：

```
void Audio::playSound(SoundId id, int volume)
{
  assert((tail_ + 1) % MAX_PENDING != head_);

  // 添加到列表的尾部
  pending_[tail_].id = id;
  pending_[tail_].volume = volume;
  tail_ = (tail_ + 1) % MAX_PENDING;
}
```

替代`tail++`，将增量设为数组长度的模，这样可将尾部回折回来。 另一个改变是断言。我们得保证队列不会溢出。 只要这里有少于`MAX_PENDING`的请求在队列中，在头部和尾部之间就有没有使用的间隔。 如果队列满了，那就不会有了，就像古怪的衔尾蛇一样，尾部会遇到头部然后覆盖它。 断言保证了这不会发生。

在`update()`中，头部也回折了：

```
void Audio::update()
{
  // 如果没有待处理的请求，就啥也不做
  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;
}
```

这样就好——没有动态分配，没有数据拷贝，缓存友好的简单数组实现的队列完成了。

> 如果最大容量影响了你，你可以使用增长的数组。 当队列满了后，分配一块当前数组两倍大的数组（或者更多倍），然后将对象拷进去。
>
> 哪怕你在队列增长时拷贝，入队仍然有常数级的摊销复杂度。

合并请求

现在有队列了，我们可以转向其他问题了。 首先来解决多重请求播放同一音频，最终导致音量过大的问题。 由于我们知道哪些请求在等待处理，需要做的所有事就是将请求和早先等待处理的请求合并：

```
void Audio::playSound(SoundId id, int volume)
{
  // 遍历待处理的请求
  for (int i = head_; i != tail_;
       i = (i + 1) % MAX_PENDING)
  {
    if (pending_[i].id == id)
    {
      // 使用较大的音量
      pending_[i].volume = max(volume, pending_[i].volume);

      // 无需入队
      return;
    }
  }

  // 之前的代码……
}
```

当有两个请求播放同一音频时，我们将它们合并成只保留声音最大的请求。 这一“合并”非常简陋，但是我们可以用同样的方法做很多有趣的合并。

注意在请求入队时合并，而不是处理时。 在队列中处理更加容易，因为不需要在最终会被合并的多余请求上浪费时间。 这也更加容易被实现。

但是，这确实将处理的职责放在了调用者肩上。 对`playSound()`的调用返回前会遍历整个队列。 如果队列很长，那么会很慢。 在`update()`中合并也许更加合理。

> 避免O(n) 的队列扫描代价的另一种方式是使用不同的数据结构。 如果我们将`SoundId`作为哈希表的键，那么我们就可以在常量时间内检查重复。

这里有些要记住的要点。 我们能够合并的“同步”请求窗口只有队列长度那么大。 如果我们快速处理请求，队列长度就会保持较短，我们就有更少的机会合并东西。 同样的，如果处理慢了，队列满了，我们能找到更多的东西合并。

这个模式隔离了请求者和何时请求被处理，但如果你将整个队列交互视为与数组结构交互，那么发出请求和处理它之间的延迟会显式的影响行为。 确认这么做之前保证这不会造成问题。

分离线程

最终，最险恶的问题。 使用同步的音频API，调用`playSound()`的线程就是处理请求的线程。 这通常不是我们想要的。

在今日的多核硬件上，你需要不止一个线程来最大程度使用芯片。 有无数的编程范式在线程间分散代码，但是最通用的策略是将每个独立的领域分散到一个线程——音频，渲染，AI等等。

> 单线程代码同时只在一个核心上运行。 如果你不使用线程，哪怕做了流行的异步风格编程，能做的极限就是让一个核心繁忙，那也只发挥CPU能力的一小部分。
>
> 服务器程序员将他们的程序分割成多个独立进程作为弥补。 这让系统在不同的核上同时运行它们。 游戏几乎总是单进程的，所以增加线程真的有用。

我们很容易就能做到这一点是因为三个关键点：

1. 请求音频的代码与播放音频的代码解耦。
2. 有队列在两者之间整理它们。
3. 队列与程序其他部分是隔离的。

剩下要做的事情就是写修改队列的方法——`playSound()`和`update()`——使之线程安全。通常，我会写一写具体代码完成之，但是由于这是一本关于架构的书，我不想着眼于一些特定的API或者锁机制。

从高层看来，我们只需保证队列不是同时被修改的。 由于`playSound()`只做了一点点事情——基本上就是声明字段。——不会阻塞线程太长时间。 在`update()`中，我们等待条件变量之类的东西，直到有请求需要处理时才会消耗CPU循环。

设计决策

很多游戏使用事件队列作为交流结构的关键部分，你可以花很多时间设计各种复杂的路径和消息过滤器。 但是在构建洛杉矶电话交换机之类的东西之前，我推荐你从简单开始。这里是几个需要在开始时思考的问题：

队列中存储了什么？

到目前为止，我交替使用“事件”和“消息”，因为大多时候两者的区别并不重要。 无论你在队列中塞了什么都可以获得解耦和合并的能力，但是还是有几个地方不同。

- 如果你存储事件：

  “事件”或者“通知”描绘已经发生的事情，比如“怪物死了”。 你入队它，这样其他对象可以对这个事件作出回应，有点像异步的观察者 [GoF]模式。

  - 很可能允许多个监听者。 由于队列包含的是已经发生的事情，发送者可能不关心谁接受它。 从这个层面来说，事件发生在过去，早已被遗忘。
  - 访问队列的模块更广。 事件队列通常广播事件到任何感兴趣的部分。为了最大程度允许哪些部分能兴趣，队列一般是全局可见的。

- 如果你存储消息：

  “消息”或“请求”描绘了想要发生在未来的事情，比如“播放声音”。可以将其视为服务的异步API。

  > 另一个描述“请求”的词是“命令”，就像在命令模式 [GoF]中那样，队列也可以在那里使用。

  - 更可能只有一个监听者。 在这个例子中，存储的消息只请求音频**API**播放声音。如果引擎的随便什么部分都能从队列中拿走消息，那可不好。

> 我在这里说“更可能”，因为只要像期望的那样处理消息，消息入队时可以不必担心哪块代码处理它。 这样的话，你在做的事情类似于服务定位器。

谁能从队列中读取？

在例子中，队列是密封的，只有`Audio`类可以从中读取。 在用户交互的事件系统中，你可以在核心内容中注册监听器。 有时可以听到术语“单播”和“广播”来描述它，两者都很有用。

- 单播队列：

  在队列是类API的一部分时是，单播很自然的。 就像我们的音频例子，从调用者的角度，它们只能看到可以调用的`playSound()`方法。

  - 队列变成了读取者的实现细节。 发送者知道的所有事就是发条消息。
  - 队列更封装。 其他都一样时，越多封装越方便。
  - 无须担心监听者之间的竞争。 使用多个监听者，你需要决定每个队列中的事物一对多分给全部的监听者（广播） 还是每个队列中的事物一对一分给单独的监听者（更加像工作队列）。

    在两种情况下，监听者最终要么做了多余的事情要么在相互干扰，你得谨慎考虑想要的行为。 使用单一的监听者，这种复杂性消失了。

- 广播队列：

  这是大多数“事件”系统工作的方法。如果你有十个监听者，一个事件进来，所有监听者都能看到这个事件。

  - 事件可能无人接收。 前面那点的必然推论就是如果有零个*监听者，没有谁能看到这个事件。 在大多数广播系统中，如果处理事件时没有监听者，事件就消失了。
  - 也许需要过滤事件。 广播队列经常对程序的所有部分可见，最终你会获得一系列监听者。 很多事件乘以很多监听者，你会获取一大堆事件处理器。

    为了削减大小，大多数广播事件系统让监听者筛出其需要接受的事件。 比如，可能它们只想要接受鼠标事件或者在某一UI区域内的事件。

- 工作队列：

  类似广播队列，有多个监听器。不同之处在于每个队列中的东西只会投到监听器其中的一个。 常应用于将工作打包给同时运行的线程池。

  - 你得规划。 由于一个事物只有一个监听器，队列逻辑需要指出最好的选项。 这也许像round robin算法或者乱序选择一样简单，或者可以使用更加复杂的优先度系统。

谁能写入队列？

这是前一个设计决策的另一面。 这个模式兼容所有可能的读/写设置：一对一，一对多，多对一，多对多。

> 你有时听到用“扇入”描述多对一的沟通系统，而用“扇出”描述一对多的沟通系统。

- 使用单个写入器：

  这种风格和同步的观察者 GoF模式很像。 有特定对象收集所有可接受的事件。

- 你隐式知道事件是从哪里来的。 由于这里只有一个对象可向队列添加事件，任何监听器都可以安全的假设那就是发送者。
- 通常允许多个读取者。 你可以使用单发送者对单接收者的队列，但是这样沟通系统更像纯粹的队列数据结构。

- 使用多个写入器：

  这是例子中音频引擎工作的方式。 由于`playSound()`是公开的方法，代码库的任何部分都能给队列添加请求。“全局”或“中心”事件总线像这样工作。

  - 得更小心环路。 由于任何东西都有可能向队列中添加东西，这更容易意外地在处理事件时添加事件。 如果你不小心，那可能会触发反馈循环。
  - 很可能需要在事件中添加对发送者的引用。 当监听者接到事件时，它不知道是谁发送的，因为可能是任何人。 如果它确实需要知道发送者，你得将发送者打包到事件对象中去，这样监听者才可以使用它。

对象在队列中的生命周期如何？

使用同步的通知，当所有的接收者完成了消息处理才会返回发送者。 这意味着消息本身可以安全的存在栈的局部变量中。 使用队列，消息比让它入队的调用活得更久。

如果你是用有垃圾回收的语言，你无需过度担心这个。 消息存到队列中，会一直存到需要它的时候。 而在C或C++中，得由你来保证对象活的足够长。

- 传递所有权：

  这是手动管理内存的传统方法。当消息入队时，队列拥有了它，发送者不再拥有它。 当它被处理时，接收者获取了所有权，负责销毁他。

  > 在C++中，`unique_ptr<T>`给了你同样的语义。

- 共享所有权：

  现在，甚至C++程序员都更适应垃圾回收了，分享所有权更加可接受。 这样，消息只要有东西对其有引用就会存在，当被遗忘时自动释放。

  > 同样的，C++的风格是使用`shared_ptr<T>`。

- 队列拥有它：

  另一个选项是让消息永远存在于队列中。 发送者不再自己分配消息的内存，它向内存请求一个“新的”消息。 队列返回一个队列中已经在内存的消息的引用，接收者引用队列中相同的消息。

  > 换言之，队列存储的背后是一个对象池[1]模式。

参见

- 我在之前提到了几次，很大程度上， 这个模式是广为人知的观察者[GoF]模式的异步实现。
- 就像其他很多模式一样，事件队列有很多别名。 其中一个是“消息队列”。这通常指代一

个更高层次的实现。 事件队列在应用中，消息队列通常在应用间交流。

另一个术语是“发布/提交”，有时被缩写为“pubsub”。 就像“消息队列”一样，这通常指代更大的分布式系统，而不是现在关注的这个模式。

- 确定状态机，很像GoF的状态模式 [GoF]，需要一个输入流。如果想要异步响应，可以考虑队列存储它们。

  当你有一对状态机相互发送消息时，每个状态机都有一个小小的未处理队列（被称为一个信箱）， 然后你需要重新发明actor model。

- Go语言内建的“通道”类型本质上是事件队列或消息队列。

← 上一章 ≡ 首页 下一章 →

服务定位器

游戏设计模式 / Decoupling Patterns

提供服务的全局接入点，避免使用者和实现服务的具体类耦合。

动机

一些游戏中的对象或者系统几乎出现在程序库中的每一个角落。 很难找到游戏中的哪部分永远不需要内存分配，记录日志，或者随机数字。 像这样的东西可以被视为整个游戏都需要的服务。

我们考虑音频作为例子。 它不需要接触像内存分配这么底层的东西，但是仍然要接触一大堆游戏系统。 滚石撞击地面（物理）。 NPC狙击手开了一枪，射出子弹（AI）。 用户选择菜单项需要响一声确认（用户界面）。

每处都需要用像下面这样的东西调用音频系统：

```
// 使用静态类？
AudioSystem::playSound(VERY_LOUD_BANG);

// 还是使用单例？
AudioSystem::instance()->playSound(VERY_LOUD_BANG);
```

尽管每种都能获得想要的结果，但是我们会绊倒在一些微妙的耦合上。 每个调用音频系统的游戏部分直接引用了具体的`AudioSystem`类，和访问它的机制——是静态类还是一个单例。 GoF

这些调用点，当然，需要耦合到某些东西上来播放声音， 但是直接接触到具体的音频实现，就好像给了一百个陌生人你家的地址，只是为了让他们在门口放一封信。 这不仅仅有隐私问题，在你搬家后，需要告诉每个人新地址是个更加痛苦的问题。

有个更好的解决办法：一本电话薄。 需要联系我们的人可以在上面查找并找到现在的地址。 当我们搬家时，我们通知电话公司。 他们更新电话薄，每个人都知道了新地址。 事实上，我们甚至无需给出真实的地址。 我们可以列一个转发信箱或者其他“代表”我们的东西。 通过让调用者查询电话薄找我们，我们获得了一个控制找我们的方法的方便地方。

这就是服务定位模式的简短介绍——它解耦了需要服务的代码和服务由谁提供（那个具体的实现类）以及服务在哪里（我们如何获得它的实例）。

模式

服务 类定义了一堆操作的抽象接口。 具体的服务提供者实现这个接口。 分离的服务定位器提供了通过查询获取服务的方法，同时隐藏了服务提供者的具体细节和定位它的过程。

何时使用

你需要让某物在程序的各处都能被访问时，你就是在找麻烦。 这是单例 GoF模式的主要问题，这个模式也没有什么不同。 我对何时使用服务定位器的最简单建议是：少用。

与其使用全局机制让某些代码接触到它，首先考虑将它传给代码。 这超简单，也明显保持了解耦，能覆盖你大部分的需求。

但是**……** 有时候手动传入对象是不可能的或者会让代码难以阅读。 有些系统，比如日志或内存管理，不该是模块公开API的一部分。 传给渲染代码的参数应该与渲染相关，而不是与日志之类的相关。

同样，代表外设的系统通常只存在一个。 你的游戏可能只有一个音频设备或者显示设备。这是周围环境的属性，所以将它传过十个函数让一个底层调用能够使用它会为代码增加不必要的复杂度。

如果是那样，这个模式可以帮忙。 就像我们将看到的那样，它是更加灵活，更加可配置的单例模式。 如果用得好，它能以很小的运行时开销，换取很大的灵活性。

> 相反，如果用得不好，它会带来单例模式的所有缺点以及更多的运行时开销。

记住

使用服务定位器的核心难点是它将依赖——在两块代码之间的一点耦合——推迟到运行时再连接。 这有了更大的灵活度，但是代价是更难在阅读代码时理解你依赖的是什么。

服务必须真的可定位

如果使用单例或者静态类，我们需要的实例不可能不可用。 调用代码保证了它就在那里。但是由于这个模式是在定位服务，我们也许要处理失败的情况。 幸运的是，我们之后会介绍一种处理它的策略，保证我们在需要时总能获得某些服务。

服务不知道谁在定位它

由于定位器是全局可访问的，任何游戏中的代码都可以请求服务，然后使用它。 这就意味着服务必须在任何环境下正确工作。 举个例子，如果一个类只能在游戏循环的模拟部分使用，而不能在渲染部分使用，那它不适合作为服务——我们不能保证在正确的时间使用它。所以，如果你的类只期望在特定上下文中使用，避免模式将它暴露给整个世界更安全。

示例代码

重回我们的音频系统问题，让我们通过服务定位器将代码暴露给代码库的剩余部分。

服务

我们从音频API开始。这是我们服务要暴露的接口：

```
class Audio
{
public:
  virtual ~Audio() {}
  virtual void playSound(int soundID) = 0;
  virtual void stopSound(int soundID) = 0;
  virtual void stopAllSounds() = 0;
};
```

当然，一个真实的音频引擎比这复杂的多，但这展示了基本的理念。 要点在于它是个没有实现绑定的抽象接口类。

服务提供者

只靠它自己，我们的音频接口不是很有用。 我们需要具体的实现。这本书不是关于如何为游戏主机写音频代码，所以你得想象这些函数中有实际的代码，了解原理就好：

```
class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID)
  {
    // 使用主机音频API播放声音……
  }

  virtual void stopSound(int soundID)
  {
    // 使用主机音频API停止声音……
  }

  virtual void stopAllSounds()
  {
    // 使用主机音频API停止所有声音……
  }
};
```

现在我们有接口和实现了。 剩下的部分是服务定位器——那个将两者绑在一起的类

一个简单的定位器

下面的实现是你可以定义的最简单的服务定位器：

```
class Locator
{
public:
  static Audio* getAudio() { return service_; }

  static void provide(Audio* service)
```

```
  {
    service_ = service;
  }

private:
  static Audio* service_;
};
```

这里用的技术被称为依赖注入，一个简单思路的复杂行话表示。 假设你有一个类依赖另一个。 在例子中，是我们的Locator类需要Audio的实例。 通常，定位器负责构造实例。 依赖注入与之相反，它指外部代码负责向对象注入它需要的依赖。

静态函数getAudio()完成了定位工作。 我们可以从代码库的任何地方调用它，它会给我们一个Audio服务实例使用：

```
Audio *audio = Locator::getAudio();
audio->playSound(VERY_LOUD_BANG);
```

它“定位”的方式十分简单——依靠一些外部代码在任何东西使用服务前已注册了服务提供者。 当游戏开始时，它调用一些这样的代码：

```
ConsoleAudio *audio = new ConsoleAudio();
Locator::provide(audio);
```

这里需要注意的关键部分是调用playSound()的代码没有意识到任何具体的ConsoleAudio类； 它只知道抽象的Audio接口。 同样重要的是，定位器 类没有与具体的服务提供者耦合。 代码中只有初始化代码唯一知道哪个具体类提供了服务。

这里有更高层次的解耦： Audio接口没有意识到它在通过服务定位器来接受访问。 据它所知，它只是常见的抽象基类。 这很有用，因为这意味着我们可以将这个模式应用到现有的类上，而那些类无需为此特殊设计。 这与单例 GoF形成了对比，那个会影响“服务”类本身的设计。

一个空服务

我们现在的实现很简单，而且也很灵活。 但是它有巨大的缺点：如果我们在服务提供者注册前使用服务，它会返回NULL。 如果调用代码没有检查，游戏就崩溃了。

我有时听说这被称为“时序耦合”——两块分离的代码必须以正确的顺序调用，才能让程序正确运行。 有状态的软件某种程度都有这种情况，但是就像其他耦合一样，减少时序耦合让代码库更容易管理。

幸运的是，还有一种设计模式叫做“空对象”，我们可用它处理这个。 基本思路是在我们没能找到服务或者程序没以正确的顺序调用时，不返回NULL， 而是返回一个特定的，实现了请求对象一样接口的对象。 它的实现什么也不做，但是它保证调用服务的代码能获取到对象，保证代码就像收到了“真的”服务对象一样安全运行。

为了使用它，我们定义另一个“空”服务提供者：

```
class NullAudio: public Audio
{
public:
  virtual void playSound(int soundID) { /* 什么也不做 */ }
  virtual void stopSound(int soundID) { /* 什么也不做 */ }
  virtual void stopAllSounds()        { /* 什么也不做 */ }
};
```

就像你看到的那样，它实现了服务接口，但是没有干任何实事。 现在，我们将服务定位器改成这样：

```
class Locator
{
public:
  static void initialize() { service_ = &nullService_; }

  static Audio& getAudio() { return *service_; }

  static void provide(Audio* service)
  {
    if (service == NULL)
    {
      // 退回空服务
      service_ = &nullService_;
    }
    else
    {
      service_ = service;
    }
  }

private:
  static Audio* service_;
  static NullAudio nullService_;
};
```

你也许注意我们用引用而非指针返回服务。 由于C++中的引用（理论上）永远不是NULL，返回引用是提示用户：总可以期待获得一个合法的对象。

另一件值得注意的事是我们在provide()而不是访问者中检查NULL。 那需要我们早早调用initialize()，保证定位器可以正确找到默认的空服务提供者。 作为回报，它将分支移出了getAudio()，这在每次使用服务时节约了检查开销。

调用代码永远不知道“真正的”服务没找到，也不必担心处理NULL。 这保证了它永远能获得有效的对象。

这对故意找不到服务也很有用。 如果我们想暂时停用系统，现在有更简单的方式来实现这点了： 很简单，不要在定位器中注册服务，定位器会默认使用空服务提供器。

在开发中能关闭音频是很便利的。它释放了一些内存和CPU循环。 更重要的，当你使用debugger时正好爆发巨响，它能防止你的鼓膜爆裂。 没有什么东西比二

十毫秒的最高音量尖叫循环更能让你血液逆流的了。

日志装饰器

现在我们的系统非常强健了，让我们讨论这个模式允许的另一个好处——装饰服务。 我会举例说明。

在开发过程中，记录有趣事情发生的小小日志系统可助你查出游戏引擎正处于何种状态。 如果你在处理AI，你要知道哪个实体改变了AI状态。 如果你是音频程序员，你也许想记录每个播放的声音，这样你可以检查它们是否是以正确的顺序触发。

通常的解决方案是向代码中丢些对`log()`函数的调用。 不幸的是，这是用一个问题取代了另一个——现在我们有太多日志了。 AI程序员不关心什么时候声音在播放，声音程序员也不在乎AI状态转换，但是现在都得在对方的日志中跋涉。

理念上，我们应该可以选择性的为关心的事物启动日志，而游戏成品中，不应该有任何日志。 如果将不同的系统条件日志改写为服务，那么我们就可以用装饰器 GoF模式。 让我们定义另一个音频服务提供者的实现：

```
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)
  {
    // 记录日志的代码……
  }

  Audio &wrapped_;
};
```

如你所见，它包装了另一个音频提供者，暴露同样的接口。 它将实际的音频行为转发给内部的提供者，但它也同时记录每个音频调用。 如果程序员需要启动音频日志，他们可以这样调用：

```
void enableAudioLogging()
{
  // 装饰现有的服务
  Audio *service = new LoggedAudio(Locator::getAudio());

  // 将它换进来
  Locator::provide(service);
}
```

现在，对音频服务的任何调用在运行前都会记录下去。 同时，当然，它和我们的空服务也能很好的相处，你能启用音频，也能继续记录如果音频被启用时将会播放的声音。

设计决策

我们讨论了一种典型的实现，但是对核心问题的不同回答有着不同的实现方式：

服务是如何被定位的？

- 外部代码注册：

  这是样例代码中定位服务使用的机制，这也是我在游戏中最常见的设计方式：

  - 简单快捷。 `getAudio()`函数简单的返回指针。这通常会被编译器内联，所以我们几乎没有付出性能损失就获得了很好的抽象层。
  - 可以控制如何构建提供者。 想想一个接触游戏控制器的服务。我们使用两个具体的提供者：一个是给常规游戏，另一个给在线游戏。 在线游戏跨过网络提供控制器的输入，这样，对游戏的其他部分，远程玩家好像是在使用本地控制器。

    为了能正常工作，在线的服务提供者需要知道其他远程玩家的IP。 如果定位器本身构建对象，它怎么知道传进来什么？ `Locator`类对在线的情况一无所知，更不用说其他用户的IP地址了。

    外部注册的提供者闪避了这个问题。定位器不再构造类，游戏的网络代码实例化特定的在线服务提供器， 传给它需要的IP地址。然后把服务提供给定位器，而定位器只知道服务的抽象接口。
  - 可以在游戏运行时改变服务。 我们也许在最终的游戏版本中不会用到这个，但是这是个在开发过程中有效的技巧。 举个例子，在测试时，即使游戏正在运行，我们也可以切换音频服务为早先提到的空服务来临时地关闭声音。
  - 定位器依赖外部代码。 这是缺点。任何访问服务的代码必须假定在某处的代码已经注册过服务了。 如果没有做初始化，要么游戏会崩溃，要么服务会神秘的不工作。
- 在编译时绑定：

  这里的思路是使用预处理器，在编译时间处理“定位”。就像这样：

```
class Locator
{
public:
  static Audio& getAudio() { return service_; }

private:
  #if DEBUG
    static DebugAudio service_;
  #else
    static ReleaseAudio service_;
  #endif
};
```

像这样定位服务暗示了一些事情：

- 快速。 所有的工作都在编译时完成，在运行时无需完成任何东西。 编译器很可能会内联`getAudio()`调用，这是我们能达到的最快方案。
- 能保证服务是可用的。 由于定位器现在拥有服务，在编译时就进行了定位，我们可以保证游戏如果能完成编译，就不必担心服务不可用。
- 无法轻易改变服务。 这是主要的缺点。由于绑定发生在编译时，任何时候你想要改变服务，都得重新编译并重启游戏。

- 在运行时设置：

企业级软件中，如果你说“服务定位器”，他们脑中第一反应就是这个方法。 当服务被请求时，定位器在运行时做一些魔法般的事情来追踪请求的真实实现。

> 反射 是一些编程语言在运行时与类型系统打交道的能力。 举个例子，我们可以通过名字找到类，找到它的构造器，然后创建实例。
>
> 像Lisp，Smalltalk和Python这样的动态类型语言自然有这样的特性，但新的静态语言比如C#和Java同样支持它。

通常而言，这意味着加载设置文件确认提供者，然后使用反射在运行时实例化这个类。这为我们做了一些事情：

- 我们可以更换服务而无需重新编译。 这比编译时绑定多了小小的灵活性，但是不像注册那样灵活，那里你可以真正的在运行游戏的时候改变服务。
- 非程序员也可改变服务。 这对于设计者师很好的，他们想要开关某项游戏特性，但修改源代码并不舒服。（或者，更可能的，编程者 对设计者介入感到不舒服。）
- 同样的代码库可以同时支持多种设置。 由于从代码库中完全移出了定位处理，我们可以使用相同的代码来同时支持多种服务设置。

  这就是这个模型在企业网站上广泛应用的原因之一： 只需要修改设置，你就可以在不同的服务器上发布相同的应用。 历史上看来，这在游戏中没什么用，因为主机硬件本身是好好标准化了的， 但是很多游戏的目标是大杂烩般的移动设备，这点就很有关系了。
- 复杂。 不像前面的解决方案，这个方案是重量级的。 你得创建设置系统，也许要写代码来加载和粘贴文件，通常要做些事情来定位服务。 花时间写这些代码上，就没法花时间写其他的游戏特性。

- 加载服务需要时间。 现在你眉会头紧蹙了。在运行时设置意味着你在消耗CPU循环加载服务。 缓存可以最小化消耗，但是仍暗示着在首次使用服务时，游戏需要暂停花点时间完成。 游戏开发者讨厌消耗CPU循环在不能提高游戏体验的地方。

如果服务不能被定位怎么办？

- 让使用者处理它：

  最简单的解决方案就是把责任推回去。如果定位器不能找到服务，只需返回NULL。这暗示着：

  - 让使用者决定如何掌控失败。 使用者也许在收到找不到服务的关键错误时应该暂停游戏。 其他时候可能可以安全地忽视并继续。 如果定位器不能定义全面的政策应对所有的情况，那么就将失败传回去，让每个使用者决定什么是正确的回应。
  - 使用服务的用户必须处理失败。 当然，这个的必然结果是每个使用者都必须检查服务的失败。 如果它们都以相同方式来处理，在代码库中就有很多重复的代码。 如果一百个中有一个忘了检查，游戏就会崩溃。

- 挂起游戏：

  我说过，我们不能保证服务在编译时总是可用的，但是不意味着我们不能声明可用性是游戏定位器运行的一部分。 最简答的方法就是使用断言：

```
class Locator
{
public:
  static Audio& getAudio()
  {
    Audio* service = NULL;

    // Code here to locate service...

    assert(service != NULL);
    return *service;
  }
};
```

  如果服务没有被找到，游戏停在试图使用它的后续代码之前。 这里的assert()调用没有解决无法定位服务的问题，但是它确实明确了问题是什么。 通过这里的断言，我们表明，“无法定位服务是定位器的漏洞。”

> 如果你没见过assert()函数，单例□模式一章中有解释

  那么这为我们做了什么呢？

  - 使用者不必处理缺失的服务。 也许简单的服务在成百上千的地方使用，这节约了很多代码。 通过声明定位器永远能够提供服务，我们节约了使用者处理它的精力。
  - 如果服务没有找到，游戏会挂起。 在极少的情况下，服务真的找不到，游戏就会挂起。 强迫我们解决定位服务的漏洞是好事（比如一些本该调用的初始化代码没有被调用）， 但被阻塞的所有人都得等到漏洞修复时。与大型开发团队工作时，当这种事情发生时，会增加痛苦的停工时间。

- 返回空服务：

  我们在样例中实现中展示了这种修复。使用它意味着：

  - 使用者不必处理缺失的服务。 就像前面的选项一样，我们保证了总是会返回可用的服务，简化了使用服务的代码。
  - 如果服务不可用，游戏仍将继续。 这有利有弊。让我们在没有服务的情况下依然能运行游戏是很有用的。 在大团队中，当我们工作依赖的其他特性或者依赖的其他系统还没有就位时，这也是很有用的。

    缺点在于，较难查找无意缺失服务的漏洞。 假设游戏用服务去获取数据，然后基于数据做出决策。 如果我们无法注册真正的服务，代码获得了空服务，游戏也许不会像期望的那样行动。 需要在这个问题上花一些时间，才能发现我们以为可用的服务是不存在的。

    > 我们可以让空服务被调用时打印一些debug信息来缓和这点。

在这些选项中，我看到最常使用的是会找到服务的简单断言。 在游戏发布的时候，它经历了严格的测试，会在可信赖的硬件上运行。 无法找到服务的机会非常小。

在更大的团队中，我推荐使用空服务。 这不会花太多时间实现，可以减少开发中服务不可用的缺陷。 这也给你了一个简单的方式去关闭服务，无论它是有漏洞还是干扰到了现在的工作。

服务的服务范围有多大？

到目前为止，我们假设定位器给任何需要服务的地方提供服务。 当然这是这个模式的典型的使用方式，另一选项是服务范围限制到类和它的依赖类中，就像这样：

```
class Base
{
  // 定位和设置服务的代码……

protected:
  // 派生类可以使用服务
  static Audio& getAudio() { return *service_; }

private:
  static Audio* service_;
};
```

通过这样，对服务的访问被收缩到了继承Base的类。这两种各有千秋：

- 如果全局可访问：

  - 鼓励整个代码库使用同样的服务。 大多数服务都被设计成单一的。 通过允许整个代码库接触到相同的服务，我们可以避免代码因不能获取“真正的”服务而到处实例化提供者，。
  - 我们失去了何时何地使用服务的控制权。 这是让某物全局化的明显代价——任何东西都能接触它。单例 GoF模式一章讲了全局变量是多么的糟糕。

- 如果接触被限制在某个类中：

    - 我们控制了耦合。 这是主要的优点。通过显式限制服务到继承树的一个分支上，应该解耦的系统保持了解耦。
    - 可能导致重复的付出。 潜在的缺点是如果一对无关的类确实需要接触服务，每个类都要拥有服务的引用。 无论是谁定位或者注册服务，它也需要在这些类之间重复处理。

      另一个选项是改变类的继承层次，给这些类一个公共的基类，但这引起的麻烦也许多于收益。)

我的通用准则是，如果服务局限在游戏的一个领域中，那么限制它的服务范围在一个类上面。 举个例子，获取网络接口的服务可能限制于在线联网类中。 像日志这样更加广泛应用的服务应该是全局的。

参见

- 服务定位模式在很多方面是单例 [GoF]模式的兄弟，在应用前值得看看哪个更适合你的需求。
- Unity框架在它的`GetComponent()`方法中使用这个模式，协调它的组件 模式
- 微软的XNA游戏开发框架在它的核心`Game`类中内建了这种模式。 每个实体都有一个`GameServices`对象可以用来注册和定位任何种类的服务。

优化模式

游戏设计模式

虽然越来越快的硬件解除了大部分软件在性能上的顾虑，对游戏却并非如此。 玩家总是想要更丰富的，更真实的，更激动人心的体验。 到处都是争抢玩家注意力——还有金钱——的游戏，能将硬件的功能发挥至极致的游戏往往获胜。

优化游戏性能是一门高深的艺术，接触到软件的各个层面。 底层程序员掌握硬件架构的种种特质。同时，算法研究者争先恐后地证明谁的过程是最有效率的。

这里，我描述了几个加速游戏的中间层模式。 数据局部性介绍了计算机的存储层次以及如何使用其以获得优势。 脏标识帮你避开不必要的计算。 对象池帮你避开不必要的内存分配。 空间分区加速了虚拟世界和其中元素的空间布局。

模式

- 数据局部性
- 脏标识
- 对象池
- 空间分区

数据局部性

游戏设计模式 / Optimization Patterns

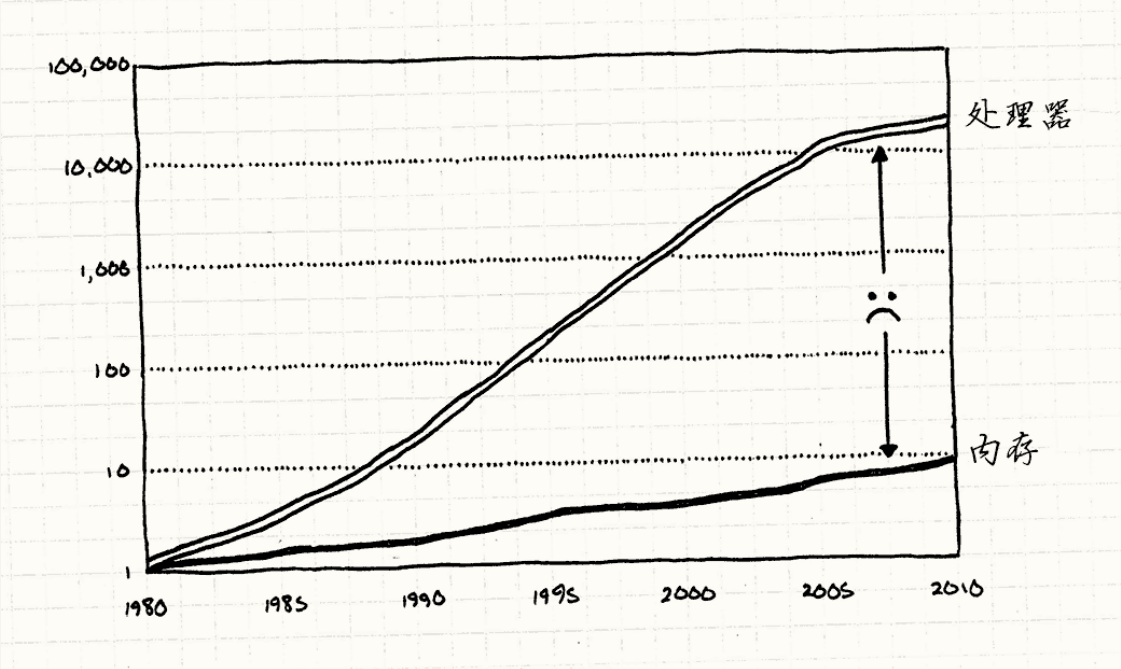
意图

合理组织数据，充分使用**CPU**的缓存来加速内存读取。

动机

我们被欺骗了。 他们一直向我们展示CPU速度每年递增的图表，就好像摩尔定律不是观察历史的结果，而是某种定理。 无需吹灰之力，软件凭借着新硬件就可以奇迹般的加速。

芯片确实越来越快（就算现在增长的速度放缓了），但硬件巨头没有提到某些事情。 是的，我们可以更快地处理数据，但不能更快地获得数据。



处理器和RAM发展速度从1980开始不同。如你所见，CPU飞跃式发展，RAM读取速度被远远甩到了后面。

这个数据来自Computer Architecture: A Quantitative Approach 由John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau基于Tony Albrecht的“Pitfalls of Object-Oriented Programming”写就。

为了让你超高速的CPU刮起指令风暴， 它需要从内存获取数据加载到寄存器。 如你所知，RAM没有紧跟CPU的速度增长，差远了。

借助现代的硬件，要消耗上百个周期才能从RAM获得一比特的数据。 如果大部分指令需要的数据都需要上百个周期去获取， 那么为什么我们的CPU没有在99%的时间空转着等待数据？

事实上，等待内存读取确实消耗很长时间，但是没有那么糟糕。 为了解释为什么，让我们看一看这一长串类比.....

> 它被称为“乱序存储器（RAM，random access memory）”是因为， 不像光驱，理论上你从某块获取数据的速度和从其他块获取数据的速度是一样的。你不需要像光盘那样考虑连续读取。
>
> 或者，你现在不需要。就像接下来看到的，RAM不是那么乱序地读取。

数据仓库

想象一下，你是小办公室里的会计。 你的任务是拿盒文件，然后做一些会计工作——把数据加起来什么的。 你必须根据一堆只有会计能懂的晦涩逻辑，取出特定标记的文件盒并工作。

> 我也许不应该在例子中用我一无所知的职业打比方。

感谢辛勤地工作，天生的悟性，还有进取心，你可以在一分钟内处理一个文件盒。 但是这里有一个小小的问题。所有这些文件盒都存储在分离的仓库中。 想要拿到一个文件盒，需要让仓库管理员带给你。 他开着叉车在传送带周围移动，直到找到你要的文件盒。

这会消耗他，认真的，一整天才能完成。 不像你，他下个月就不会被雇佣了。 这就意味着无论你有多快，一天只能拿到一个文件盒。 剩下的时间，你只能坐在那里，质疑自己怎么选了这个折磨灵魂的工作。

一天，一组工业设计师出现了。 他们的任务是提高操作的效率——比如让传送带跑得更快。在看着你工作几天后，他们发现了几件事情：

- 通常，当你处理文件盒时，下一个需要处理的文件盒就在仓库同一个架子上。
- 叉车一次只取一个文件盒太愚蠢了。
- 在你的办公室角落里还是有些空余空间的。

> 描述访问刚刚访问事物旁边的位置的术语是引用局部性。

他们想出来一个巧妙的办法。 无论何时你问仓库要一个盒子，他都会带给你一托盘的盒子。 他给你想要的盒子，以及它周围的盒子。 他不知道你是不是想要这些（而且，根据工作条件，他根本不在乎）； 他只是尽可能的塞满托盘，然后带给你。

无视工作场地的安全，他直接将叉车开到你的办公室，然后将托盘放在办公室的角落。

当你需要新盒子，你需要做的第一件事就是看看它在不在办公室角落的托盘里。 如果在，很好！你只需要几分钟拿起它然后继续计算数据。 如果一个托盘中有五十个盒子，而幸运的你需要所有盒子，你可以以五十倍的速度工作。

但是如果你需要的盒子不在托盘上，就要一新托盘的盒子。 由于你的办公室里只能放一托

盘的盒子，你的仓库朋友只能将旧的拿走，带给你一托盘全新的盒子。

CPU的托盘

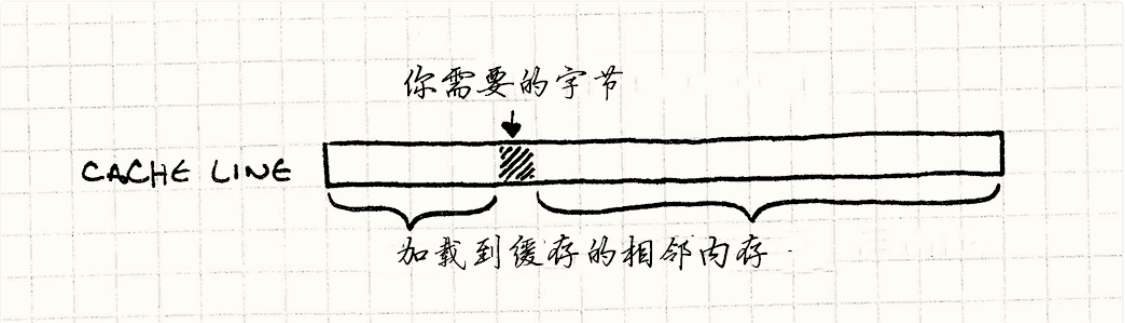
奇怪的是，这就是现代CPU运转的方式。如果还不够明显，你是CPU。 你的桌子是CPU的寄存器，一盒文件就是寄存器能放下的数据。 仓库是机器的RAM，那个烦人的仓库管理员是从主存加载数据到寄存器的总线。

如果我在三十年前写这一章，这个比喻就到此为止了。 但是芯片越来越快，而RAM，好吧，“没有跟上”，硬件工程师开始寻找解决方案。 他们想到的是**CPU**缓存技术。

现代的电脑在芯片内部有一小块存储器。 CPU从那上面取数据比从内存取数据快得多。 它很小，因为需要放在芯片上，它很快，因为使用的（静态RAM，或称SRAM）内存更贵。

> 现代硬件有多层缓存，就是你听到的“L1”，“L2”，“L3”之类的。 每一层都更大也更慢。在这章里，我们不关心内存是不是多层的，但了解一下还是很有必要的。

这一小片内存被称为缓存（特别的，芯片上的被称为**L1**级缓存）， 在比喻中，它是由托盘扮演的。 无论何时芯片需要从RAM取一字节的数据，它自动将一整块内存读入——通常是64到128字节——然后将其放入缓存。 这些一次性传输的字节被称为**cache line**。



如果你需要的下一字节数据就在这块上， CPU从缓存中直接读取，比从RAM中读取快得多。 成功从缓存中找到数据被称为“缓存命中”。 如果不能从中获得而得去主存里取，这就是一次缓存不命中。

> 我在类比中掩盖了（至少）一个细节。在办公室里，只能放一个托盘，或者一个cache line。 真实的缓存包含多个cache line。关于这点的细节超出了本书的范围，搜索“缓存关联性”来了解相关内容。

当缓存不命中时，CPU空转——它不能执行下一条指令，因为它没有数据。 它坐在那里，无聊的等待几百个周期直到取到数据。 我们的任务是避免这一点。想象你在优化一块性能攸关的游戏代码，长得像这样：

```
for (int i = 0; i < NUM_THINGS; i++)
{
  sleepFor500Cycles();
  things[i].doStuff();
}
```

你会做的第一个改动是什么？对了。 将那个无用的，代价巨大的函数调用拿出来。 这个调用等价于一次缓存不命中的代价。 每次跳到内存，都会延误你的代码。

等等，数据是性能？

当着手写这一章时，我花费了一些时间制作一个类似游戏的小程序，用于收集缓存使用最好情况和最坏情况。 我想要缓存速度的基准，这样可以得到缓存失效造成的性能浪费。

当看到一些工作的结果时，我惊到了。 我知道这是一个大问题，但眼见为实。 两个程序完成完全相同的计算，唯一的区别是它们会造成缓存不命中的数量。 较慢的程序比较快的程序慢五十倍。

> 这里有很多警告。特别的，不同的计算机有不同的缓存设置，所以我的机器可能和你的不同， 专用的游戏主机与个人电脑不同，而二者都与移动设备不同。
>
> 你得自己测测看。

这让我大开眼界。我一直从代码的角度考虑性能，而不是数据。 一个字节没有快慢，它是静态的。但是因为缓存的存在，组织数据的方式直接影响了性能。

现在真正的挑战是将这些打包成一章内可以讲完的东西。 优化缓存使用是一个很大的话题。 我还没有谈到指令缓存呢。 记住，代码也在内存上，而且在执行前需要加载到CPU上。有些更熟悉这个主题的人可以就这个问题写一整本书。

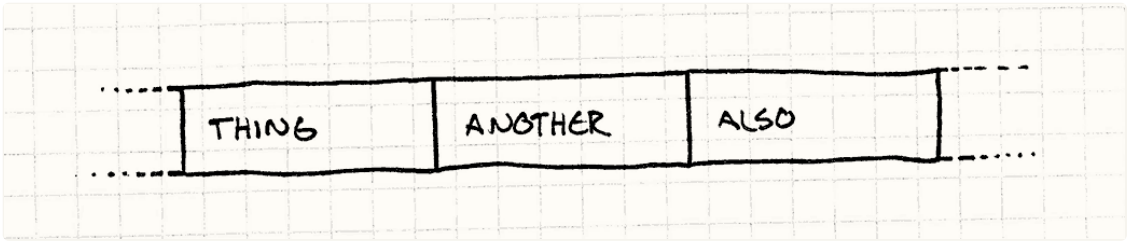
> 事实上，有人确实写了一本书：Data-Oriented Design，作者Richard Fabian.

既然你已经在阅读这本书了， 我有几个基本技术帮你考虑数据结构是如何影响性能的。

这可以归结成很简单的事情：芯片读内存时总是获得一整块cache line。 你能从cache line读到越多你要的东西，跑得就越快。 所以目标是组织数据结构，让要处理的数据紧紧相邻。

> 这里有一个核心假设：单线程。 如果在多线程上处理邻近数据，让它在多个不同的cache line上更好。 如果两个线程争夺同一cache line上的数据，两个核都得花些时间同步缓存。

换言之，如果你正处理`Thing`，然后`Another`然后`Also`，你需要它们这样呆在内存里：



注意，不是`Thing`，`Another`，和`Also`的指针，而是一个接一个的真实数据，。 CPU读到`Thing`，也会读取`Another`和`Also`（取决于数据的大小和cache line的大小）。 当你开始下一个时，它们已经在缓存上了。芯片很高兴，你也很高兴。

模式

现代的CPU有缓存来加速内存读取。 它可以更快的读取最近访问过的内存的毗邻内存。 通过提高内存局部性来提高性能——保证数据以处理顺序排列在连续内存上。

何时使用

就像大多数优化方案，使用数据局部性的第一准则是在遇到性能问题时使用。 不要将其应

用在代码库不经常使用的角落上。 优化代码不会让你过得更轻松，因为其结果往往更加复杂，更加缺乏灵活性。

就本模式而言，还得确认你的性能问题确实由缓存不命中引发。 如果代码是因为其他原因而缓慢，这个模帮不上忙。

简单的估算方案是手动添加指令，检查代码中两点间消耗的时间，寄希望于精确的计时器。为了找到糟糕的缓存使用，你需要使用更加复杂的东西。 你想要知道有多少缓存不命中，又是在哪里发生的。

幸运的是，有现成的工具做这些。 在数据结构上做大手术前，花一些时间了解这些工具是如何工作， 理解它们抛出的一大堆数据（令人惊讶的复杂）是很有意义的。

> 不幸的是，这些工具大部分不便宜。如果在一个主机开发团队，你可能已经有了使用它们的证书。
>
> 如果没有，一个极好的替代选项是Cachegrind。 它在模拟的CPU和缓存结构上运行你的程序，然后报告所有的缓存交互。

话虽这么说，缓存不命中仍会影响游戏的性能。 虽然不应该花费大量时间提前优化缓存的使用，但是在设计过程中仍要思考数据结构是不是对缓存友好。

记住。

软件体系结构的特点之一是抽象。 这本书很多的章节都在谈论如何解耦代码块，这样可以更容易的进行改变。 在面向对象的语言中，这几乎总是意味着接口。

在C++中，使用接口意味着通过指针或者引用访问对象。 但是使用指针就意味在内存中跳跃，这就带来了这章想要避免的缓存不命中。

> 接口的另一半是虚方法调用。 这需要CPU查找对象的虚函数表，找到调用方法的真实指针。 所以，你又一次追踪指针，造成缓存不命中。

为了讨好这个模式，你需要牺牲一些宝贵的抽象。 你越围绕数据局部性设计程序，就越放弃继承、接口和它们带来的好处。 没有银弹，只有挑战性的权衡。这就是乐趣所在！

示例代码

如果你真的要一探数据局部性优化的深处，那么你会发现有无数的方法去分割数据结构，将其切为CPU更好处理的小块。 为了热热身，我会先从一些最通用的分割方法开始。 我们会在游戏引擎的特定部分介绍它们， 但是（像其他章节一样）记住这些通用方法也能在其他部分使用。

连续数组

让我们从处理一系列游戏实体的游戏循环开始。 实体使用了组件模式，被分解到不同的领域——AI，物理，渲染。 这里是GmaeEntity类。

```
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_;
};
```

每个组件都有相对较少的状态，也许只有几个向量或一个矩阵， 然后会有方法去更新它。这里的细节无关紧要，但是想象一下，大概是这样的：

> 就像名字暗示的，这些是更新方法模式的例子。 甚至`render()`也是这个模式，只是换了个名字。

```
class AIComponent
{
public:
  void update() { /* 处理并修改状态…… */ }

private:
  // 目标，情绪，等等……
};

class PhysicsComponent
{
public:
  void update() { /* 处理并修改状态…… */ }

private:
  // 刚体，速度，质量，等等……
};

class RenderComponent
{
public:
  void render() { /* 处理并修改状态…… */ }

private:
  // 网格，纹理，着色器，等等……
};
```

游戏循环管理游戏世界中一大堆实体的指针数组。每个游戏循环，我们都要做如下事情：

1. 为每个实体更新他们的AI组件。
2. 为每个实体更新他们的物理组件。
3. 为每个实体更新他们的渲染组件。

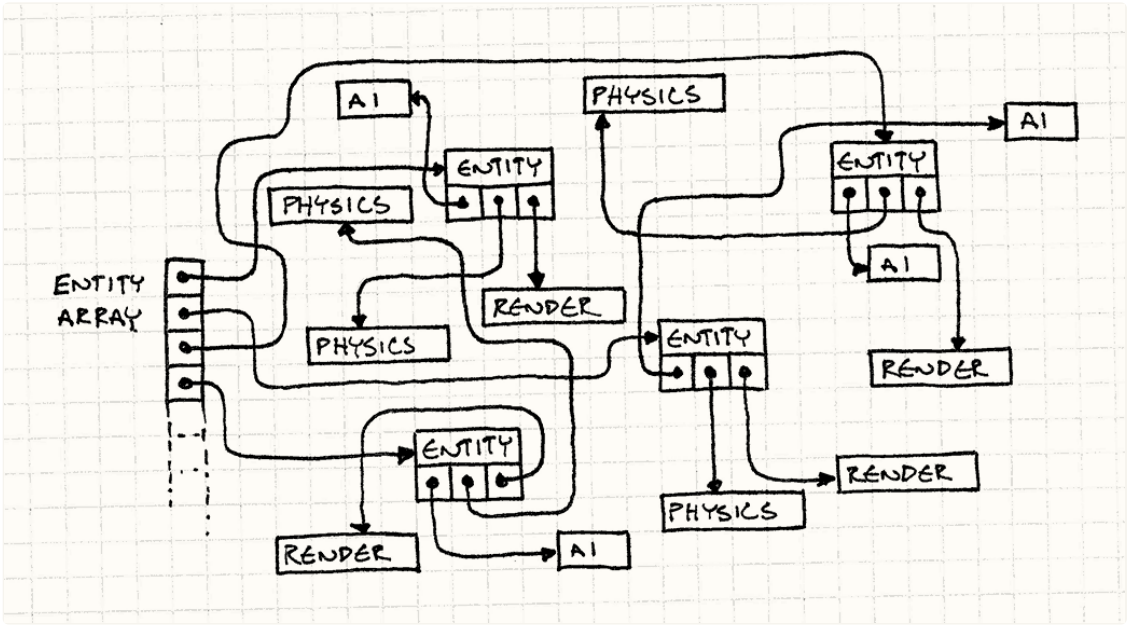
很多游戏引擎以这种方式实现：

```
while (!gameOver)
{
  // 处理AI
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->ai()->update();
  }

  // 更新物理
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->physics()->update();
  }

  // 绘制屏幕
  for (int i = 0; i < numEntities; i++)
  {
    entities[i]->render()->render();
  }

  // 其他和时间有关的游戏循环机制……
}
```

在你听说CPU缓存之前，这些看上去完全无害。 但是现在，你得看到这里有隐藏着的不对之处。 这不是在颠簸缓存，这是在四处乱晃然后猛烈地敲击。看看它做了什么：

1. 游戏实体的数组存储的是指针，所以为了获取游戏实体，我们得转换指针。缓存不命中。
2. 然后游戏实体有组件的指针。又一次缓存不命中。
3. 然后我们更新组件。
4. 再然后我们退回第一步，为游戏中的每个实体做这件事。

令人害怕的是，我们不知道这些对象是如何在内存中布局的。 我们完全任由内存管理器摆布。 随着实体的分配和释放，堆会组织更加乱。



每一帧，游戏循环得追踪这些指针来获取数据。

如果我们目标是在游戏地址空间中四处乱转，完成“256MB内存四晚廉价游”，这也许是一个很好的决定。 但是我们的目标是让游戏跑得尽可能快，而在主存四处乱逛不是一个好办法。 记得`sleepFor500Cycles()`函数吗？上面的代码效率和它也差不多了。

描述浪费时间转换指针这一行为的术语是“追逐指针”，那并不像听上去那么有趣。

我们能做得更好。 第一个发现是，之所以跟着指针去寻找游戏实体，是因为可以立刻跟着另一个指针去获得组件。 `GameEntity`本身没有有意义的状态和有用的方法。组件 才是游戏循环需要的。

众多实体和组件不能像星星一样散落在黑暗的天空中，我们得脚踏实地。 我们将每种组件存入巨大的数组：一个数组给AI组件，一个给物理，另一个给渲染。

就像这样：

```
AIComponent* aiComponents =
    new AIComponent[MAX_ENTITIES];
PhysicsComponent* physicsComponents =
    new PhysicsComponent[MAX_ENTITIES];
RenderComponent* renderComponents =
    new RenderComponent[MAX_ENTITIES];
```

使用组件时，我最不喜欢的就是组件这个单词的长度。

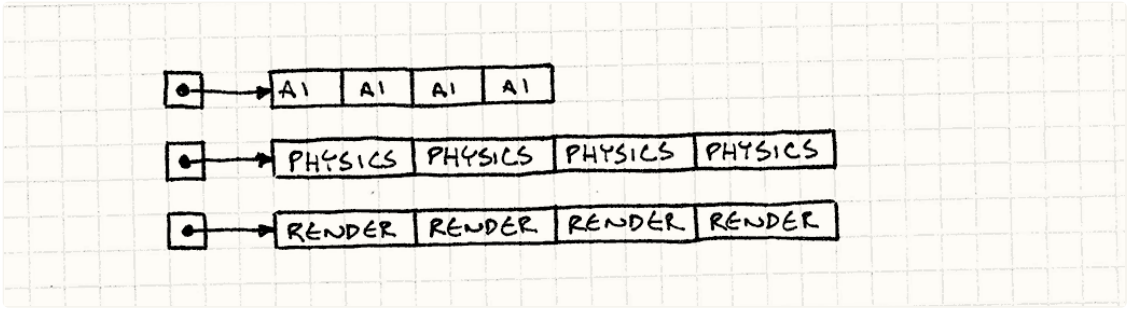
让我强调一点，这些都是组件的数组，而不是指向组件的指针。数据都在那里一个接着一个排列。 游戏循环现在可以直接遍历它们了。

```
while (!gameOver)
{
  // 处理AI
  for (int i = 0; i < numEntities; i++)
  {
```

```
    aiComponents[i].update();
  }

  // 更新物理
  for (int i = 0; i < numEntities; i++)
  {
    physicsComponents[i].update();
  }

  // 绘制屏幕
  for (int i = 0; i < numEntities; i++)
  {
    renderComponents[i].render();
  }

  // 其他和时间有关的游戏循环机制……
}
```

在这里做的更好的一条提示是新代码中有更少的->操作符。 如果你想要提高数据局部性，找找哪些你可以摆脱的间接跳转。

我们消除了所有的指针追逐。不在内存中跳来跳去，而是直接在三个数组中做直线遍历。



这将一股字节流直接泵到了CPU饥饿的肚子里。 在我的测试中，这个改写后的更新循环是之前性能的**50**倍。

有趣的是，我们并没有在这里放弃太多的封装。 是的，游戏循环直接更新的游戏组件而不是通过游戏实体，但在此之前它已经确保了以正确的顺序运行。 即使如此，每个组件的内部还是具有很好的封装性。 它们的封装性取决于自身的数据和方法。我们只是改变了使用它们的方法。

这也不意味着我们摆脱了`GameEntity`。它拥有它组件的指针这一状态仍然得以保持。 它的组件指针现在只是指到了这个数组之中。 对游戏的其他部分，如果你还是想传递一个“游戏实体”，一切照旧。 重要的是性能攸关的游戏循环部分回避了这点，直接获取数据。

打包数据

假设我们在做粒子系统。 根据上节的建议，将所有的粒子放在巨大的连续数组中。让我们用管理类封装它。

`ParticleSystem`类是对象池的一个例子，通常为单一类型对象构建。

```
class Particle
```

```
{
public:
  void update() { /* 重力，等等…… */ }
  // 位置，速度，等等……
};

class ParticleSystem
{
public:
  ParticleSystem()
  : numParticles_(0)
  {}

  void update();
private:
  static const int MAX_PARTICLES = 100000;

  int numParticles_;
  Particle particles_[MAX_PARTICLES];
};
```

系统中的基本更新方法看起来是这样的：

```
void ParticleSystem::update()
{
  for (int i = 0; i < numParticles_; i++)
  {
    particles_[i].update();
  }
}
```

但实际上不需要同时更新所有的粒子。 粒子系统维护固定大小的对象池，但是粒子通常不是同时在屏幕上活跃。 最简单的解决方案是这样的：

```
for (int i = 0; i < numParticles_; i++)
{
  if (particles_[i].isActive())
  {
    particles_[i].update();
  }
}
```

我们给Particle一个标志位来追踪其是否在使用状态。 在更新循环时，我们检查每个粒子的这个标志位。 这会将粒子其他部分的数据也加载到缓存中。 如果粒子没有在使用，那么跳过它去检查下一个。 这时粒子加载到内存中的其他数据都是浪费。

活跃的粒子越少，要在内存中跳过的部分就越多。 越这样做，在两次活跃粒子有效更新之间发生的缓存不命中就越多。 如果数组很大又有很多不活跃的粒子，我们又在颠簸缓存了。

如果连续数组中的对象不是连续处理的，实际上这个办法也没有太多效果。 如果有太多不

活跃的对象需要跳过，就又回到了问题的起点。

> 理解底层代码的程序员也可以看出这里的问题。 使用`if`为每个粒子检查会引起分支预测错误和流水线暂停。 在现代CPU中，一条简单的“指令”实际消耗多个时钟周期。 为了保持CPU繁忙，指令流水线化，在前面的指令处理完成之前就开始处理。
>
> 为了实现流水线，CPU需要猜测接下来要执行哪一条指令。 在顺序结构的代码中，这很简单，但是加入控制流，就难了。 当它为`if`执行指令，它是猜粒子是活跃的然后执行`update()`调用，还是猜它不活跃呢？
>
> 为了回答这一点，芯片做分支预测——它看看之前的代码选择了哪条分支然后照做。 但是当循环不断在活跃和不活跃粒子之间转换，预测就失败了。
>
> 当它失败，CPU取消它推测的代码（流水线更新），重头开始。 这在机器上波及广泛，这是为什么有时候你看到开发者在热点代码避免控制流。

鉴于本节的标题，你大概可以猜出答案是什么了。 我们不监测活跃与否的标签，我们根据标签排序粒子。 将所有活跃的粒子放在列表的前头。 如果知道了这些粒子都是活跃的，就不必再检查这些标识位了。

还可以很容易地追踪有多少活跃的粒子。这样，更新循环变成了这种美丽的东西：

```
for (int i = 0; i < numActive_; i++)
{
  particles[i].update();
}
```

现在没有跳过任何数据。 加载入缓存的每一字节都是需要处理的粒子的一部分。

当然，我可没说每帧都要对整个数组做快排。 这将抵消这里的收益。我们想要的是保持数组的顺序。

假设数组已经排好序了——开始时确实如此，因为所有粒子都不活跃——它变成未排序的时候即是粒子被激活或者被反激活时。 我们可以很轻易的处理这两种情况。 当一个粒子激活时，我们让它占据第一个不活跃粒子的位置， 将不活跃粒子移动到激活序列的尾端，完成一次交换：

```
void ParticleSystem::activateParticle(int index)
{
  // 不应该已被激活！
  assert(index >= numActive_);

  // 将它和第一个未激活的粒子交换
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;

  // 现在多了一个激活粒子
  numActive_++;
}
```

为了反激活粒子，只需做相反的事情：

```
void ParticleSystem::deactivateParticle(int index)
{
  // 不应该已被激活！
  assert(index < numActive_);

  // 现在少了一个激活粒子
  numActive_--;

  // 将它和最后一个激活粒子交换
  Particle temp = particles_[numActive_];
  particles_[numActive_] = particles_[index];
  particles_[index] = temp;
}
```

很多程序员（包括我在内）已经对于在内存中移动数据过敏了。 将一堆数据移来移去感觉比发送指针要消耗大得多。 但是如果你加上了解析指针的代价，有时候这种估算是错误的。 在有些情况下，如果能够保证缓存命中，在内存中移动数据消耗更小。

> 在你做这种决策前要记得验证这点。

将粒子根据激活状态保持排序——就不需要给每个粒子都添加激活标志位了。 这可以由它在数组中的位置和`numActive_`计数器推断而得。 这让粒子对象更小，意味着在cache lines中能够打包更多数据，能跑得更快。

但是并非完事如意。 你可以从API看出，我们放弃了一定的面向对象思想。 `Particle`类不再控制其激活状态了。 你不能在它上面调用`activate()`因为它不知道自己的索引。 相反，任何想要激活粒子的代码都需要接触到粒子系统。

在这个例子中，将`ParticleSystem`和`Particle`这样牢牢绑一起没有问题。 我将它们视为两个物理类的的组合概念。 这意味着粒子只在特定的粒子系统中有意义。 在这种情况下，很可能是粒子系统在复制和销毁粒子。

冷/热 分割

这里是最后一种取悦缓存的技术例子。 假设某些游戏实体有AI控件。 其中包括一些状态——现在正在播放的动画，正在前往的方向，能量等级，等等——这些东西每帧都会发生变化。 就像这样：

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;
};
```

但它也有一些罕见事件的状态。 它存储了一些数据，描述它遭到猎枪痛击后会掉落什么战利品。 掉落数据在实体的整个生命周期只会使用一次，就在它结束的前一霎那：

```
class AIComponent
{
public:
  void update() { /* ... */ }

private:
  // 之前的字段……
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

假设我们遵循前面的章节，那么当我们更新AI组件时，就穿过了一序列打包好的连续数组。 那个数据包含所有掉落物的信息。 这让每个组件都变得更大了，这就减少了我们能够加载到cache line中的组件个数。 每帧的每个组件都会将战利品数据加载到内存中去，即使我们根本不会去使用它。

这里的解决方案被称为“冷/热分割”。这个点子源于将数据结构划分为两个分离的部分。 第一部分保存“热”数据，那些每帧都要调用的数据。 剩下的片段被称为“冷”数据，在那里的存储使用的次数较少的数据。

这里的热部分是AI组件的主体。 它是使用最多的部分，所以我们不希望解析指针去找到它。 冷组件可以被归到一边去，但是我们还是需要访问它，因此我们在热组件中包含一个指向它的指针，就像这样：

```
class AIComponent
{
public:
  // 方法……
private:
  Animation* animation_;
  double energy_;
  Vector goalPos_;

  LootDrop* loot_;
};

class LootDrop
{
  friend class AIComponent;
  LootType drop_;
  int minDrops_;
  int maxDrops_;
  double chanceOfDrop_;
};
```

现在我们每帧都要遍历AI组件，加载到内存的数据只包含必需的数据（以及那个指向冷数据的指针）。

> 我们可以继续去除指针，为冷热数据维护平行数组。 仍能够为组件找到冷数据，因为两者在各自数组中索引值是相同的。

你可以看到事情是怎么变得模棱两可的。 在我的例子中，哪些是冷数据，哪些是热数据是很明确的，但是在真实的游戏中一般很少可以这么明显的分割。 如果你有一部分数据，实体在一种状态下会经常使用，另一种状态则不会，那该怎么办？ 如果实体只在特定关卡时使用一块特定的数据，又该怎么办？

做这种优化有时就是在走钢丝。 很容易陷入其中，消耗无尽的时间把数据挪来挪去看看性能如何。 需要通过实践来掌握在哪里付出努力。

设计决策

这章更接近于介绍一种思维定势——将数据的组织模式作为游戏性能的关键部分。 实际上具体的设计空间是开放的。 你可以让数据局部性影响整个架构，或者只在局部几个核心数据结构上使用这模式。

最需要关心的问题是在何时何地使用这个模式，但是这里还有其他几个问题需要回答。

> Noel Llopis的著名文章让很多人围绕缓存设计游戏，他称之为“面向数据的设计”。

你如何处理多态？

到了现在，我们避开了子类和虚方法。 我们假设有打包好的同类对象。 这种情况下，我们知道它们有同样的大小。 但是多态和动态调用也是有用的工具。我们如何调和呢？

- 别这么干

  最简单的解决方案是避免子类，至少在做内存优化的部分避免使用。 无论如何，软件工程师文化已经和大量使用继承渐行渐远了。

  > 一种保持多态的灵活性而不使用子类的方法是通过类型对象模式。

  - 简洁安全。 你知道在处理什么类，所有的对象都是同样大小。
  - 更快 动态调用意味着在跳转表中寻找方法，然后跟着指针寻找特定的代码。 这种消耗在不同硬件区别很大，但动态调用总会带来一些代价。

  > 就像往常一样，万事无绝对。 在大多数情况下，虚方法调用中C++编译器需要一次重定向。 但是在某些情况下，如果可以知道接受者的具体类型，编译器可以去虚拟化，然后静态地调用正确的方法。 去虚拟化在一些just-in-time虚拟机比如Java和JavaScript中更为常见。

  - 不灵活 当然，使用动态调用的原因就是它给了在不同对象间展示不同的行为的强大能力。 如果游戏想要不同的实体使用独特的渲染、移动或攻击，虚方法是处理它的好方法。 把它换成包含巨大的`switch`的非虚方法会超级慢。
- 为每种类型使用分离的数组：

我们使用多态，这样即使不知道对象的类型，也能引入行为。 换言之，有了一包混合的东西，我们想要其中每个对象在接到通知时去做自己的事情。

但是这提出来一个问题：为什么开始的时候要把它们混在一起呢？ 取而代之，为什么不为每种类型保持一个单独的集合呢？

- 对象被紧密的排列着。 每个数组只包含同类的对象，这里没有填充或者其他的古怪。
- 静态调度。 一旦获得了对象的类型，你不必在所有时候使用多态。你可以使用常规的非虚方法调用。
- 得追踪每个集合。 如果你有很多不同类型，这种管理每种类型分别的数组可是件苦差事。
- 得明了每一种类型。 由于你为每种类型管理分离的集合，你无法解耦类型集合。 多态的魔力之一在于它是开放的——与一个接口交互的代码可以与实现此接口的众多类型解耦。

- 使用指针的集合：

  如果你不太担心缓存，这是自然的解法。 只要一个指针数组指向基类或者接口类型，你就获得了想要的多态，以及想多大多大的对象。

  - 灵活。这样构建集合的代码可以与任何支持接口的类工作。完全开放。
  - 对缓存不友好。 当然，我们在这里讨论其他方案的原因就是指针跳转导致的缓存不友好。 但是，记住，如果代码不是性能攸关的，这很有可能是行得通的。

游戏实体是如何定义的？

如果与组件[1]模式串联使用此模式， 你会获得多个数组，包含组成游戏实体的组件。 游戏循环会在那里直接遍历它们，所以实体本身就不是那么重要了， 但是在其他你想要与“实体”交互的代码库部分，一个概念上的实体还是很有用的。

这里的问题是它该如何被表示？如何追踪这些组件？

- 如果游戏实体是拥有它组件指针的类：

  这是第一个例子中的情况。纯OOP解决方案。 你得到了`GameEntity`类，以及指向它拥有的组件的指针。 由于它们只是指针，并不知道这些组件是如何在内存中组织的了。

  - 你可以将实体存储到连续数组中。 既然游戏实体不在乎组件在哪里，你可以将组件好好打包，组织数组中，来优化遍历。
  - 拿到一个实体，可以轻易的获得它的组件。 就在一次指针跳转后的位置。
  - 在内存中移动组件很难。 当组件启用或者关闭时，你可能想要在数组中移动它们，保证启用的组件位于前列。 如果在实体中有指针指向组件时直接移动该组件，一不小心指针就会损毁。你得保证同时更新指向组件的指针。

- 如果游戏实体是拥有组件**ID**的类：

  使用裸指针的挑战在于内存中移动它很难。你可以使用更加直接的方案：使用ID或者索引来查找组件。

  ID的实际查找过程是由你决定的，它可能很简单，只需为每个实体保存独特的ID，然后遍历数组查找， 或者更加复杂地使用哈希表，将ID映射到组件现有位置。

- 更复杂。 ID系统不是高科技，但是还是需要比指针多做些事情。你得实现它然后排除漏洞，这里需要消耗内存。
- 更慢。 很难比直接使用指针更快。需要使用搜索或者哈希来帮助实体找到它的组件。
- 你需要访问组件**“管理器”**。 基本思路是用抽象的ID标识组件。你可以使用它来获得对应组件对象的引用。 但是为了做到这点，你需要让ID有办法找到对应的组件。正是包裹着整个连续组件数组的对所要做的。

  通过裸指针，如果你有游戏实体，你可以直接找到组件，而这种方式你需要接触游戏实体和组件注册器。

  > 你也许在想，“我会把它做成单例！问题解决！”好吧，在某种程度上是这样的。 不过，你也许想要先看看这章。

- 如果游戏实体本身就是一个**ID**：

  这是某些游戏引擎使用的新方式。一旦实体的行为和状态被移出放入组件，还剩什么呢？ 事实上，没什么了。实体干的唯一事情就是将组件连接在一起。 它的存在只是为了说明：这个AI组件和这个物理组件还有这个 渲染组件合起来， 定义了一个存在于游戏世界中的实体。

  这很重要，因为组件要相互交互。 渲染组件需要知道实体位于何处，而位置信息也许是物理组件的属性。 AI组件想要移动实体，因此它需要对物理组件施加力。每个组件都需要以某种方式获得同一实体中的其他组件。

  有些聪明人意识到你需要的唯一东西就是ID。不是实体知道组件，而是组件知道实体。每个组件都知道拥有它实体的ID。当AI组件需要它所属实体的物理组件时，它只需要找到那个拥有同样ID的物理组件。

  你的实体类整个消失了，取而代之的是围绕数字的华丽包装。

  - 实体很小。当你想要传递游戏实体的引用时，只需一个简单的值。
  - 实体是空的。当然，将所有东西移出实体的代价是，你必须将所有东西移出。 不能再拥有组件独有的状态和行为，这样更加依赖于组件模式。
  - 不必管理实体的生命周期。 由于实体只是内置值类型，不需要被显式分配和释放。当它所有的组件都被释放时，对象就隐式“死亡”了。
  - 查找实体的某一组件也许会很慢。 这和前一方案有相同的问题，但在另一个方向上。 为了找某个实体的组件，你需要给ID做对象映射。这一过程消耗也许很大。

    但是，这一次是性能攸关的。 在更新时，组件经常与它的兄弟组件交互，因此你需要经常的查找组件。 解法是让组件在数组中的索引作为实体的“ID”。

    如果每个实体都是拥有相同组件的集合，那么组件数组就是完全同步的。 组件数组三号位的AI组件与在物理组件数组三号位的组件相关联。

    但是，记住，这强迫你保持这些数组平行。 如果你想要用不同的方式排序或者打包它们就会变得很难。 你也许需要一些没有物理组件或者没有渲染组件的实体。 而它们仍保证与其他组件同步，没有办法独自排序物理组件数组和渲染组件数组。

参见

- 这一章大部分围绕着组件模式。 这种模式的数据结构绝对是为缓存优化的最常见例子。事实上，使用组件模式让这种优化变得容易了。 由于实体是按“领域”（AI，物理，等等）更新的，将它们划出去变成组件，更容易将它们保存为对缓存友好的合适大小。

  但是这不意味你只能为组件使用这个模式！ 任何需要接触很多数据的关键代码，考虑数据局部性都是很重要的。

- Tony Albrecht的《Pitfalls of Object-Oriented Programming》PDF 也许是最广为人知的内存友好游戏设计指南。它让很多人（包括我！）明白了数据结构对性能是多么重要。
- 几乎同时，Noel Llopis关于同一话题写了一篇 非常有影响力的博客。
- 这一模式几乎完全得益于同类对象的连续存储数组。 随着时间的推移，你也许需要向那个数组增加或删除对象。 对象池模式正是关于这一点。
- 游戏引擎Artemis是首个也是最著名的为游戏实体使用简单ID的游戏框架。

脏标识模式

意图

将工作延期至需要其结果时才去执行，避免不必要的工作。

动机

很多游戏有场景图。 那是一个巨大的数据结构，包含了游戏世界中所有的对象。 渲染引擎使用它决定在屏幕哪里画东西。

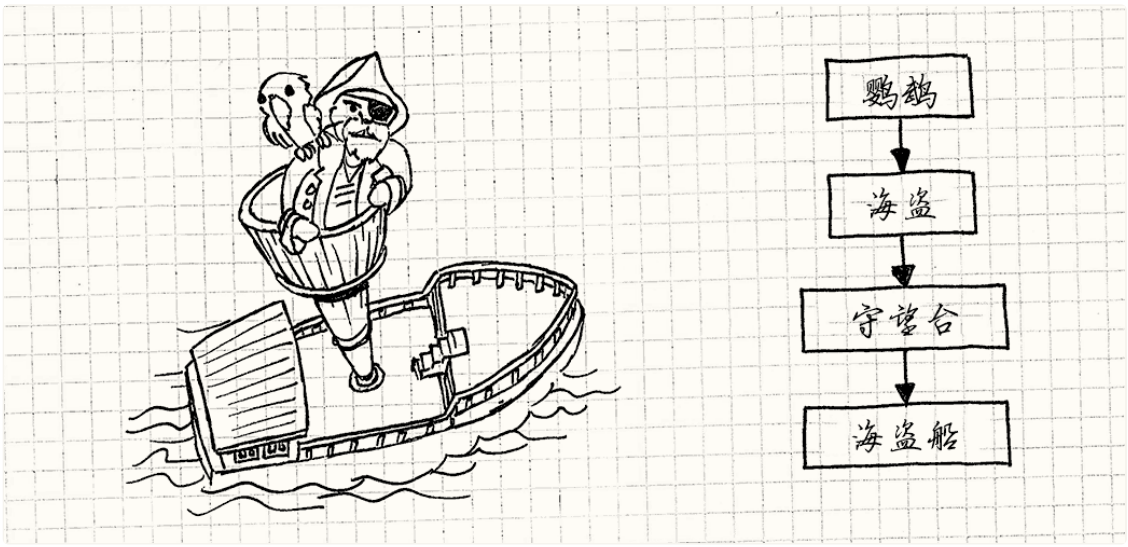
最简单的实现中，场景图只是对象列表。 每个对象都有模型，或者其他的原始图形，以及变换。 变换描述了对象在世界中的位置，方向，拉伸。 为了移动或者旋转对象，只需简单地改变它的变换。

> 如何 存储和操作变换的内容很不幸超出了本书讨论范围。 简单的总结下，是个4x4的矩阵。 你可以通过矩阵相乘来组合两个变换，获得单一变换——举个例子，平移之后旋转对象。
>
> 它如何工作，以及为什么那样工作是留给读者的练习。

当渲染系统描绘对象，它取出对象的模型，对其应用变换，然后将渲染到游戏世界中。 如果我们有场景包而不是场景图，那就是这样了，生活很简单。

但是，大多数场景图都是分层的。 场景图中的对象也许拥有锚定的父对象。 这种情况下，它的变换依赖于父对象的位置，而不是游戏世界上的绝对位置。

举个例子，想象游戏世界中有一艘海上的海盗船。 桅杆的顶端有瞭望塔，瞭望塔中有海盗，海盗肩上有鹦鹉。 船本身的变换定位船在海上的位置。瞭望塔的变换定位它在船上的位置，诸如此类。



> 编程艺术！

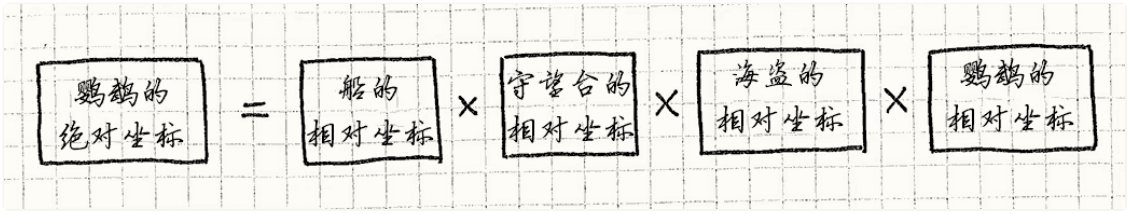
这样的话，当父对象移动时，子节点也自动的跟着移动。 如果改变了船的自身变换，瞭望塔，海盗和鹦鹉都会随之变动。 如果当船移动时，就得手动调整每个对象的变换来防止滑动，那可相当令人头疼。

> 老实说，当你在海上，你确实需要手动调整姿势来防止滑动。 也许我应该选一个不会滑动的例子。

但是为了在屏幕上真正的描绘鹦鹉，我需要知道它在世界上的绝对位置。 我会调用父节点相关的变换对对象的自身变换进行变换。 为了渲染对象，我需要知道对象的世界变换。

自身变换和世界变换

计算对象的世界变换很直接——从它的父根节点一直追踪到对象，将经过的所有变换绑在一起。 换言之，鹦鹉的世界变换如下：



> 如果对象没有父对象，它的自身变换和世界变换是一样的。

我们每帧需要为游戏世界的每个对象计算世界变换，因此哪怕每个模型只有一部分矩阵乘法， 它也是代码影响性能的关键所在。 保持它们及时更新是有技巧的，因为当父对象移动时，它影响自己的世界变换，并递归影响所有子节点。

最简单的方法是在渲染时计算变换。 每一帧，我们从最高层递归遍历整个场景图。 我们计算每个对象的世界变换然后立刻绘制它。

但这完全是在浪费CPU！ 很多游戏世界的对象不是在每帧都移动。 想想那些构成关卡的静态几何图形。 在没有改变的情况下每帧计算它们的世界变换是一种浪费。

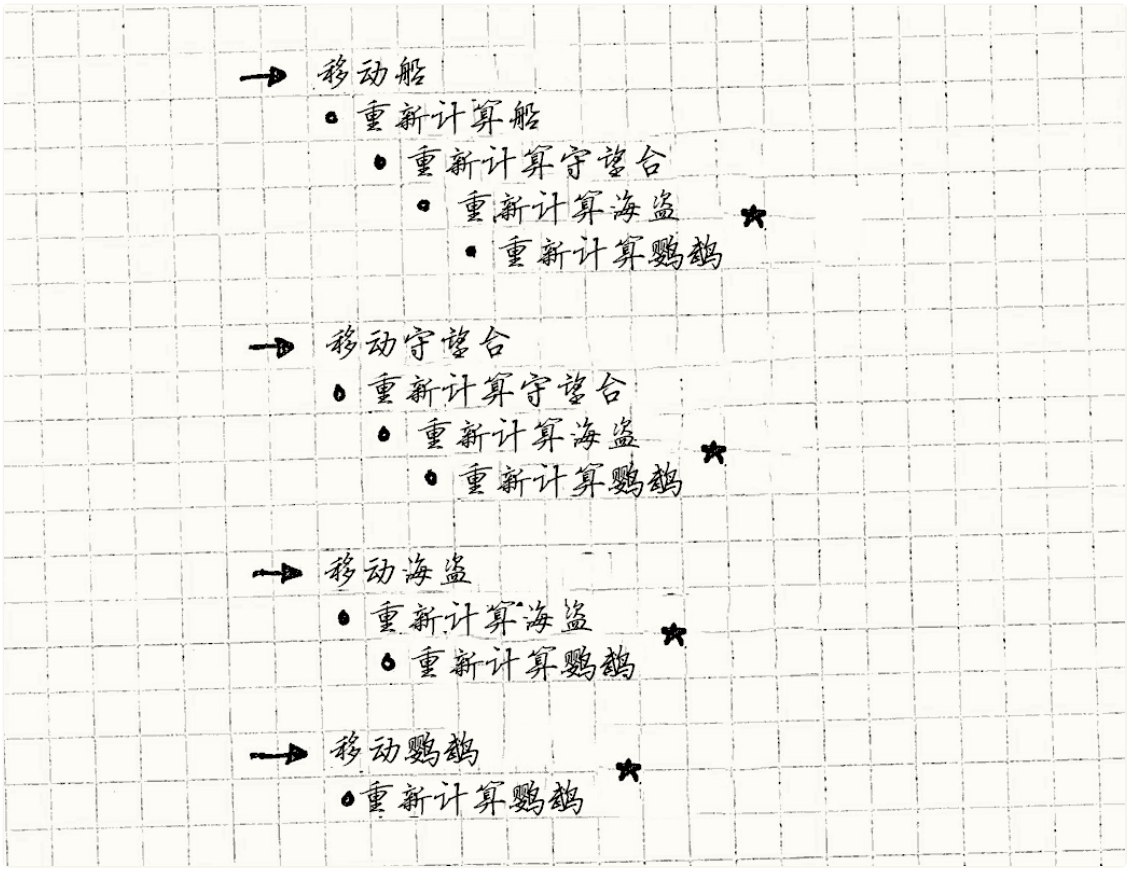
缓存世界变换

明显的解决方案是缓存它。 在每个对象中，我们存储它的自身变换和世界变换。 当我们渲

染时使用预计算的世界变换。 如果对象从未移动，缓存的变换永远是最新的变换，每个人都很开心。

当一个对象确实移动了，简单的解决方式是之后就更新世界变换。 但是不要忘记层次性！当父节点移动时，我们得计算它的世界变换并递归计算它所有的子对象。

想象游戏中忙碌的时刻。 在一帧中，船在海上颠簸，瞭望塔在风中摇晃，海盗被甩到了边缘，而鹦鹉撞上了他的脑袋。 我们改变了四个自身变换。如果每次自身变换都立即更新世界变换，会发生什么？



> 你可以看到在标记了★的行上，我们重复计算了四次鹦鹉的世界变换，但我们只需要最后的那次。

我只移动四个对象，但我们做了十次世界变换计算。 那就有六次在被渲染器使用前浪费了。 我们计算了鹦鹉的世界变换四次，但它只需渲染一次。

问题在于世界变换也许会依赖于多个自身变换。 由于我们每次变化就立即重新计算，当自身变换依赖的多个世界变换在同一帧发生变化时， 我们就对同一变换做了多次重新计算。

延期重计算

我们会通过解耦自身变换和世界变换的更新来解决这个问题。 这让我们先在一次批处理中改变自身变换，在这些改变完成之后，在渲染它之前，重新计算它们世界变换。

> 有趣的是，不少软件架构是故意稍微偏离了一点。

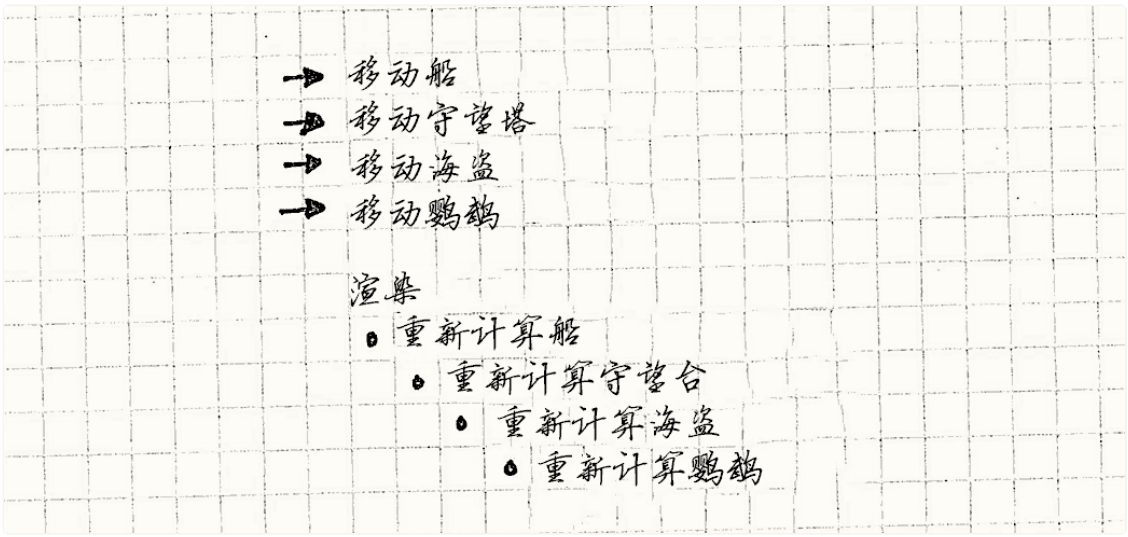
为了做到这点，我们为图中的每个对象添加标识。 “标识”和“位”在编程中密切相关——都代表处在两种状态之一的一小块数据。 我们称之为“真”和“假”，或者有时称为“设置”和“清除”。 我之后会交替使用它们。

当自身变换改变了，我们设置它。 当我们需要对象的世界变换时，我们检查这位。 如果它

被设置了，计算世界变换然后清除标识。 那个标识代表着，“世界变换过时了吗？” 由于它们没有被清除，这种“过时的杂乱”被称为“脏”。 也就是脏标识。“脏位”也是这模式通常使用的名字，但是我决定使用不那么下流的称呼。

> 维基百科的编辑者没有我这样的自制力，使用了dirty bit.

如果我们运用这个模式，然后移动之前例子中所有对象，那么游戏最终是这样的：



这就是你能希望得到的最好结果了——每个受影响对象的世界变换只被计算一次。 使用仅仅一位数据，这个模式为我们做了以下事情：

- 它将对象的父节点链上的众多的自身变换变化归并成对象上的一次计算。
- 它避免了在没有移动的对象上重新计算。
- 还有一个小小的意外收获：如果对象在渲染前被删除了，不必再计算它的世界变换。

模式

一组原始数据随着时间变化而改变。 使用昂贵的过程推定一组导出数据。 一个**“脏”**标识追踪导出数据是否与原始数据保持一致。 它在原始数据改变时被设置。 如果导出数据被请求时，该标识被设置了，那么重新计算并清除标识 否则的话，使用之前缓存的导出数据。

何时使用

与这本书中的其他模式相比，这个模式解决了一个非常特殊的问题。 同时，就像其他优化一样，只在性能问题足够大时，再使用这一模式增加代码的复杂度。

脏标识在两种任务上应用：“计算”和“同步”。 在两种情况下，从原始数据变换到导出数据消耗很多时间，或者有很多其他方面的消耗。

在我们的场景图例子中，这个过程非常缓慢是因为需要执行很多数学运算。 在同步上使用这个模式是另一个应用场景， 导出数据在别的地方——在磁盘上或者在网络另一头的终端机上——从点A传输到点B消耗很大。

这里是一些其他的应用场景：

- 原始数据的变化速度远高于导出数据的使用速度。 避免在导出数据使用前原始数据多次

变化带来的不必要计算。 如果你总在原始数据变化后立即使用导出数据，这个模式无法帮忙。

- 增量更新十分困难。 假设海盗船只能携带特定数量的战利品。我们需要获取携带事物的总重量。 我们可以使用这个模式，然后为总重量设脏标识。每次添加或者移除一些战利品，我们设置这个标识。 当我们需要总量时，将所有战利品重量加起来，然后清除标识。

  但是更简单的解决方法是保存计算总量。 当我们添加或删除事物，直接从现在的总重量添加或者删除它的重量。 如果我们可以承担得起消耗，保持导出数据的更新，那么更好的选择是不用这个模式， 每次需要时重新计算导出数据。

这听起来脏标识很少有能使用的时候，但你总会找到一两个部分它能帮得上忙。 直接在你的游戏代码库中搜索“dirty”，通常会发现这个模式的使用之处。

> 根据我的研究，也能找到很多对“dirty”黑魔法的抱怨注释。

记住

哪怕是在说服自己这个模式在这里很恰当之后，这里还有一些瑕疵可能会让你不爽。

延期太久是有代价的

这个模式将某些耗时的工作延期到真正需要结果的时候，但是当它要的时候，通常是现在就要。 但是我们使用这个模式的原因是计算很耗时！

在例子中，这不是问题，因为我们还是可以在一帧之内计算世界坐标，但是可以想象其他情况下，工作需要消耗可观时间。 如果玩家想要结果时才开始计算，这会引起不愉快的卡顿。

延期的另一个问题是，如果有东西出错了，你可能根本无法弥补。 当你使用这个模式将状态持久化时，问题更加突出。

举个例子，文本编辑器知道文档有“没保存的修改”。 在文件标题栏的小点或者星号就是可见的脏标识。 原始数据是在内存中打开的文档，推导数据是在磁盘上的文件。

很多程序直到文档关闭或者应用退出才保存到磁盘上。 在大多数情况下这很好，但是如果一不小心踢到了插线板，你的主要工作也就随风而逝了。

在后台自动保存备份的编辑器弥补了这一失误。 自动保存的频率保持在崩溃时不丢失太多数据和频繁保存文件之间。

> 这反映了自动内存管理系统的不同垃圾回收策略。 引用计数在不需要内存时立即释放它，但每次引用改变时都会更新引用计数，那消耗了大量CPU时间。
>
> 简单的垃圾回收器将回收内存拖延到需要内存时，但是代价是可怕的， “垃圾回收

过程”会冻住整个游戏，直到回收器完成了对堆的处理。

在两者之间是更复杂的系统，像延时引用计数和增量的垃圾回收， 比纯粹的引用计数回收要消极，但比冻住游戏的回收系统更积极。

每次 状态改变你都得保证设置标识。

由于推导数据是从原始数据推导而来的，它本质上是缓存。 无论何时缓存了数据，都是需要保证缓存一致性——在缓存与原始数据不同步时通知之。 在这个模式上，这意味着在任何原始数据变化时设置脏标识。

> Phil Karlton有句名言：“计算机科学中只有两件难事：缓存一致性和命名。”

一处遗漏，你的程序就使用了错误的推导数据。 这引起了玩家的困惑和非常难以追踪的漏洞。 当使用这个模式时，你也得注意，任何修改了原始数据的代码都得设置脏标识。

一种解决它的方法是将原始数据的修改隐藏在接口之后。 任何想要改变状态的代码都要通过API，你可以在API那里设置脏标识来保证不会遗漏。

得将之前的推导数据保存在内存中。

当推导数据被请求而脏标识没有设置，就使用之前计算出的数据。 这很明显，但这需要在内存中保存推导数据，以防之后再次使用。

> 如果你用这个模式将原始状态同步到其他地方，这不是问题。 那样的话，推导数据通常不在内存里。

如果你没有使用这个模式，可在需要时计算推导数据，使用完后释放。 这避免将其存储回内存的开销，而代价是每次使用都需要重新计算。

就像很多优化一样，这种模式以内存换速度。 通过在内存中保存之前计算的结果，避免了在它没有改变的情况下重新计算。 这种交易在内存便宜而计算昂贵时是划算的。 当你手头有更多空闲的时间而不是内存的时候，最好在需求时重新计算。

> 相反，压缩算法做了反向的交易： 它们优化空间，代价是解压时额外的处理时间。

示例代码

假设我们满足了超长的需求列表，看看在代码中是如何应用这个模式的。 就像我之前提到的那样，矩阵运算背后的数学知识超出了本书的范围，因此我将其封装在类中，假设在某处已经实现了：

```
class Transform
{
public:
  static Transform origin();

  Transform combine(Transform& other);
};
```

这里我们唯一需要的操作就是combine()， 这样可以将父节点链上所有的自身变换组合起来获得对象的世界变换。 同样有办法来获得原点变换——通常使用一个单位矩阵，表示没有平移，旋转，或者拉伸。

下面，我们勾勒出场景图中的对象类。这是在应用模式之前最低限度所需的东西：

```
class GraphNode
{
public:
  GraphNode(Mesh* mesh)
  : mesh_(mesh),
    local_(Transform::origin())
  {}

private:
  Transform local_;
  Mesh* mesh_;

  GraphNode* children_[MAX_CHILDREN];
  int numChildren_;
};
```

每个节点都有自身变换描述了它和父节点之间的关系。 它有代表对象图形的真实网格。（将mesh_置为NULL来处理子节点的不可见节点。） 最终，每个节点都包含一个有可能为空的子节点集合。

通过这样，“场景图”只是简单的GraphNode，它是所有的子节点（以及孙子节点）的根。

```
GraphNode* graph_ = new GraphNode(NULL);
// 向根图节点增加子节点……
```

为了渲染场景图，我们需要的就是从根节点开始遍历节点树，然后使用正确的世界变换为每个节点的网格调用函数：

```
void renderMesh(Mesh* mesh, Transform transform);
```

我们不会直接在这里实现，但真正的实现中也是做渲染需要的事，将网格绘制在世界上给定的位置。 如果对场景图中的每个节点都正确有效地调用，这就愉快地完成了。

尚未优化的遍历

让我们开始吧，我们做一个简单的遍历，在渲染需要时去计算所有的世界位置。 这没有优化，但它很简单。我们添加一个新方法给GraphNode：

```
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);
  }
}
```

使用parentWorld将父节点的世界变换传入节点。 这样，需要获得这个节点的世界变换只需要将其和节点的自身变换相结合。 不需要向上遍历父节点去计算世界变换，因为我们可以在向下遍历时计算。

我们计算了节点的世界变换，将其存储到world，如果有网格，渲染它。 最后我们遍历进入子节点，传入这个节点的世界变换。 无论如何，这是一个紧密的，简单的遍历方法。

为了绘制整个场景图，我们从根节点开始整个过程。

```
graph_->render(Transform::origin());
```

让我们变脏

所以代码做了正确的事情——它在正确的地方渲染正确的网格——但是它没有高效地完成。它每帧在图中的每个节点上调用local_.combine(parentWorld)。 让我们看看这个模式是如何修复这一点的。首先，我们给GraphNode添加两个字段：

```
class GraphNode
{
public:
  GraphNode(Mesh* mesh)
  : mesh_(mesh),
    local_(Transform::origin()),
    dirty_(true)
  {}

  // 其他方法……

private:
  Transform world_;
  bool dirty_;
  // 其他字段……
};
```

world_字段缓存了上一次计算出来的世界变换，和dirty_这个脏标识字段。 注意标识初始为true。当我们创建新节点时，我们还没有计算它的世界变换。 初始时，它与自身变换不是同步的。

我们需要这个模式的唯一原因是对象可以移动，因此让我们添加对这点的支持：

```
void GraphNode::setTransform(Transform local)
{
  local_ = local;
  dirty_ = true;
}
```

这里重要的部分是同时设置脏标识。我们忘了什么吗？是的——子节点！

当父节点移动时，它所有子节点的世界坐标也改变了。 但是这里，我们不设置它们的脏标识。 我们可以那样做，但是那要递归，很缓慢。我们可以在渲染时做点更聪明的事。让我们看看：

```
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);
  }
}
```

这里有一个微妙的假设：`if`检查比矩阵乘法快。直观上，你当然会这么想，检测一位当然比一堆浮点计算要快。

但是，现代CPU超级复杂。它们严重依赖于流水线——入队的一系列连续指令。像我们这里的`if`造成的分支会引发分支预测失败，强迫CPU消耗周期在填满流水线上。

数据局部性 一章有更多现代CPU是如何试图加快运行的细节， 以及如何避免这样颠簸它们。

这与原先的原始实现很相似。 关键改变是我们在计算世界变换之前去检查节点是不是脏的，然后将结果存在字段中而不是本地变量中。 如果节点是干净的，我们完全跳过了`combine()`，使用老的但是正确的`world_`值。

这里的技巧是`dirty`参数。 如果父节点链上有任何节点是脏的，那么就是`true`。 当我们顺着层次遍历下来时，`parentWorld`用同样的方式更新它的世界变换，`dirty`追踪父节点链的是否有脏。

这让我们避免递归地调用`setTransform()`标注每个子节点的`dirty_`标识。 相反，我们在渲染时将父节点的脏标识传递给子节点，然后看看是否需要重新计算它的世界变换。

这里结果正是我们需要的： 改变节点的自身变换只是一些声明，渲染世界时只计算从上一帧以来所需的最小数量的世界变换。

注意这个技巧能有用是因为`render()`是`GraphNode`中唯一需要最新世界变换的。 如果其他东西也要获取，我们就得做点不同的事。

设计决策

这种模式非常具体，所以只需注意几点：

什么时候清空脏标识？

- 当结果被请求时？
  - 如果不需要结果，可以完全避免计算。 如果原始数据变化的速度比推导数据获取的速度快得多，这效果很明显。
  - 如果计算消耗大量时间，这会造成可察觉的卡顿。 将工作推迟到玩家想要结果的时候会严重影响游戏体验。 这部分工作一般足够快，不会构成问题，但是如果构成问题，你就需要提前做这些工作。
- 在精心设计的检查点处：

  有时候，某个时间点或在游戏过程中很自然的需要推迟处理。 例如，只有海盗船驶入港口才会去保存游戏。 如果同步点不是游戏的机制，我们会将这些工作隐藏在加载画面或者过场动画之后。
  - 这种工作不会影响到玩家体验。 不像前一个选项，游戏在紧张运行时，你总能转移玩家的注意力。
  - 在工作时丧失了控制权。 这和前一个选项相反。你在处理时能进行微观控制，确保有效优雅地处理它。

    你不能保证玩家真的到了检查点或者满足了定义的条件。 如果他们在游戏中迷失了，或者游戏进入了奇怪的状态，最终工作会推迟得超乎预料的晚。
- 在后台处理：

  通常情况下，你为第一次更改时启动固定时长的计时器，然后在计时器到时间后处理之间的所有变化。

  > 在人机交互界，用术语**hysteresis**描述程序接受用户的输入和响应之间的故意延迟。
  - 可以控制工作进行的频率。 通过调节计时器，可以保证它发生的像预期一样频繁（或者不频繁）。
  - 更多的冗余工作。 如果原始状态在计时器运行之间只改变了很少的部分，最终大部分处理的都是没有改变的数据。
  - 需要支持异步工作。 在“后台”处理数据意味着玩家可以同时继续做事。 这就意味着你将会需要线程或者其他并行支持，这样游戏在处理数据的同时仍然可以继续游玩。

    由于玩家很可能与处理中的状态交互，你也需要考虑保持并行修改的安全性。

脏追踪的粒度有多细？

假设我们的海盗游戏允许玩家建造并个性化自己的船。 船在线时会自动保存，这样玩家可以离线后恢复。 我们使用脏标识记录船的哪块甲板被修改了并需要发送到服务器。 每一块发送给服务器的数据都包括了修改的数据和一些描述改动发生在何处的元数据。

- 如果粒度更细：

  假设你为甲板上的每个小木板都拍上一个脏标识。

  - 你只需处理真正改变的数据。 你只将船上修改了的数据发送到服务器。
- 如果粒度更粗：

  或者，我们可以为每层甲板关联一个脏标识。改变它上面的任何东西都会让整个甲板变脏。

  > 我可以说些这里不合时宜的糟糕笑话，但我克制住了。

  - 最终需要处理没有变化的数据。 在甲板上添加一个桶，就要将整层甲板发送到服务器。
  - 用在存储脏标识上的内存更少。 为甲板上添加十个桶只需要一位来追踪。
  - 固定开销花费的时间更少。 当处理某些修改后的数据时，通常处理数据之前有些固定的工作要做。 在这个例子中，是确认船上改动在哪里的元数据。 处理的块越大，那么要处理的数量就越少，这就意味着有更小的开销。

参见

- 在游戏之外，这个模式在像Angular的浏览器方向框架中是很常见的。 它们使用脏标识来追踪哪个数据在浏览器中被改变了，需要将其推向服务器。
- 物理引擎追踪哪些对象在运动中哪些在休息。 由于休息的骨骼直到有力施加在上面才会移动，在被碰到时才会需要处理。 “正在移动”就是一个脏标识，标注哪个对象上面有力施加并需要物理解析。

对象池模式

意图

放弃单独地分配和释放对象，从固定的池子中重用对象，以提高性能和内存使用率

动机

我们在处理游戏的视觉效果。 当英雄释放了法术，我们想要在屏幕上爆发闪光。 这需要调用粒子系统，产生动态的闪烁图形，显示动画直到图形消失。

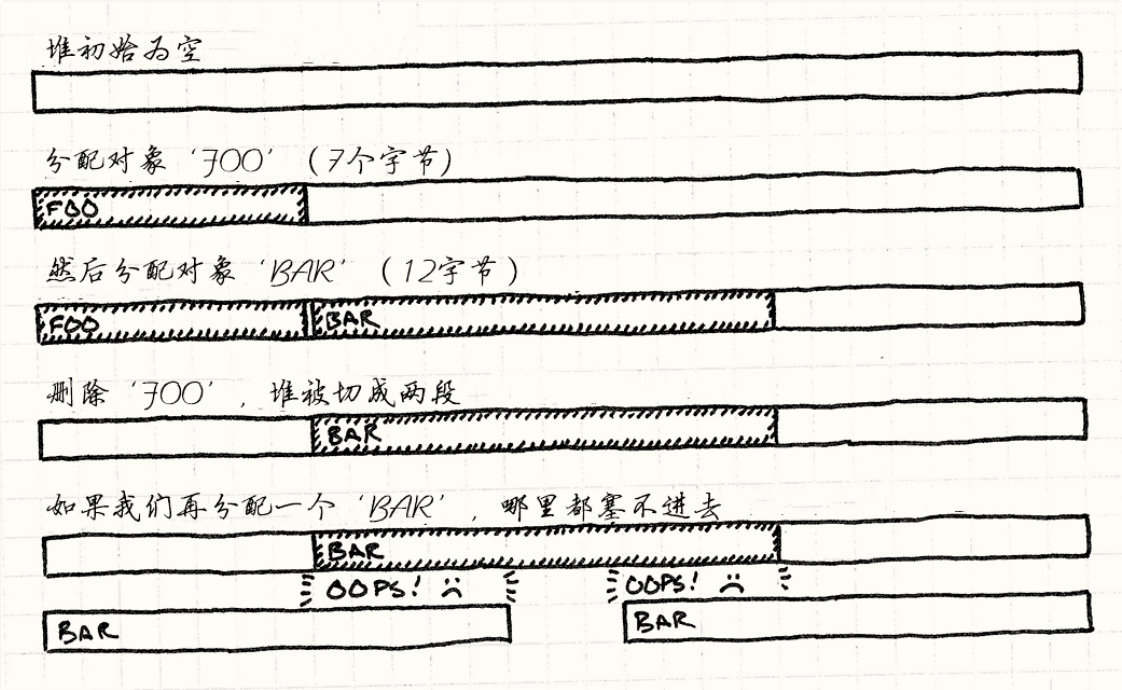
由于一次简单的魔杖挥舞就能产生成百上千的粒子，系统需要能够快速生成它们。 更重要的是，我们需要保证创建和销毁这些粒子不会造成内存碎片。

碎片的诅咒

为游戏主机或者移动设备编程在许多方面比为普通的计算机编程更像是嵌入式编程。 内存紧张，玩家希望游戏能如岩石般稳定运行，压缩内存的管理器很难有效。 在这种环境下，内存碎片是致命的。

碎片意味着在堆中的空余空间被打碎成了很多小的内存碎片，而不是大的连续内存块。 总共的 可用内存也许很大，但是最长的连续空间可能难以忍受的小。 假设我们有十四个空余字节，但是被一块正在使用的内存分割成了两个七字节的碎片。 而我们尝试分配十二字节的对象，那么就会失败。屏幕上不会有更多的闪烁火花了。

> 这有点像在已经停了很多车的繁忙街道上停车。 如果它们挤在一起，那就有地方停，但空闲地带变成了车之间的碎片空间。



这里展现了堆是怎么碎片化的，以及即使在理论上有足够的可用内存，内存也会分配失败。

哪怕碎片化发生得不频繁，它也仍会逐渐把堆变成了有空洞和裂隙的不可用泡沫，最终完全无法运行游戏。

大多数主机游戏制作商要求游戏通过“浸泡测试”，即让游戏在demo模式运行上几天。 如果游戏崩溃了，他们不允许游戏发售。 浸泡测试失败有时是因为发生罕见的漏洞，但碎片增长或者内存泄露是造成游戏停止的大部分原因。

兼得鱼和熊掌

由于碎片化和可能很慢的内存分配，游戏中何时何处管理内存通常需要十分小心。 一个简单又有效的办法是——游戏开始时取一大块内存，然后直到游戏结束才去释放它。 但是这对要在游戏运行时创建和销毁事物的系统是痛苦的。

使用对象池能让我们兼得鱼和熊掌。 对内存管理器，我们只需要将一大块内存分出来，保持在游戏运行时不释放它。 对于池的使用者，我们可以简单的构造析构我们想要的内容对象。

模式

定义一个池对象，其包含了一组可重用对象*。 其中每个可重用对象都支持查询“使用中”状态**，说明它是不是“正在使用”。 池被初始化时，它就创建了整个对象集合（通常使用一次连续的分配），然后初始化所有对象到“不在使用中”状态。

当你需要新对象，向池子要一个。 它找到一个可用对象，初始化为“使用中”然后返回。 当对象不再被需要，它被设置回“不在使用中”。 通过这种方式，可以轻易的创建和销毁对象而不必分配内存或其他资源。

何时使用

这个模式广泛应用于可见的事物上，比如游戏实体和视觉效果， 但是它也可在不那么视觉化的数据结构上使用，比如正在播放的声音。 在以下情况中使用对象池：

- 需要频繁创建和销毁对象。
- 对象大小相仿。
- 在堆上进行对象内存分配十分缓慢或者会导致内存碎片。
- 每个对象都封装了像数据库或者网络连接这样很昂贵又可以重用的资源。

记住

你通常依赖垃圾回收机制或者`new`和`delete`来处理内存管理。 通过使用对象池，你是在说，“我知道如何更好地处理这些字节。”这就意味着处理内存的责任落到了你头上。

池可能在不需要的对象上浪费内存

对象池的大小需要根据游戏的需求设置。 当池子太小时，很明显需要调整（没有什么比崩溃更能获得你的注意力了）。 但是小心池子没有太大。更小的池子提供了空余的内存做其他有趣的事情。

同时只能激活固定数量的对象

在某种程度上这是好事。 将内存按不同的对象类型划分单独池子保证了这点。 举个例子，一连串爆炸不会让粒子系统消耗掉所有可用内存，然后阻碍创建新敌人这样的关键事件。

尽管如此，这也意味着试图从池子重用对象可能会失败，因为它们都在使用中。 这里有几个常见对策：

- 完全阻止这点。 这是通常的“修复”：增加对象池的大小，这样无论用户做什么，它们都不会溢出。 对于重要对象，比如敌人或游戏道具，这通常是正确的选择。 也许没有“正确的”方法来处理玩家抵达关底时创建巨大Boss内存不足的问题，所以最聪明的办法就是保证这不发生。

  这个的副作用是强迫你为那些只在一两个罕见情况下需要的对象分配过多的内存。 因此，固定大小的对象池也许不对所有的游戏状态都适用。 举个例子，某些关卡也许需要更多的效果而其他的需要声音。 在这种情况下，考虑为每个场景调整对象池的大小。

- 就不要创建对象了。 这听起来很糟，但是对于像粒子系统这样的情况很有道理。 如果所有的粒子都在使用，那么屏幕已经充满了闪动的图形。 用户不会注意到下个爆炸不如现在的这个一样引人注目。

- 强制干掉一个已有的对象。 想想正在播放声音的内存池，假设需要播放新声音而对象池满了。 你不想简单地忽视新声音——用户会注意到魔法剑有时会发出戏剧般的声音，有时顽固的一声不吭。 更好的解决方法是找到播放中最轻的声音，然后用新声音替代之。 新声音会覆盖掉前一个声音。

  大体上，如果已有对象的消失要比新对象的出现更不引人察觉，这也许是正确的选择。

- 增加池的大小。 如果游戏允许你使用一点内存上的灵活性，我们也许会在运行时增加池

子的大小或者创建新的溢出池。 如果用这种方式获取内存，考虑下在增加的内存不再需要时，池子是否需要缩回原来的大小。

每个对象的内存大小是固定的

多数对象池将对象存储在一个数组中。 如果你所有的对象都是同样的类型，这很好。 但是，如果你想要在同一个对象池中存储不同类型的对象，或者存储子类的实例， 你需要保证池中的每个位置对最大的可能对象都有足够的内存。 否则，超过预期大小对象会占据下一个对象的内存空间，导致内存崩坏。

同时，如果对象大小是变化的，你是在浪费内存。 每个槽都需要能存储最大的对象。 如果对象很少那么大，每放进去一个小对象都是在浪费内存。 这很像是通过机场安检时，使用最大允许尺寸的箱子，而里面只放了钥匙和钱包。

当你发现在用这种方式浪费内存，思考将池根据对象的大小分割为分离的池子 ——大箱子给大行李，小箱子给口袋里东西。

> 这是一种实现有效率的内存管理的常用模式。 管理者拥有一系列池子，池子的块大小不相同。 当你申请分配一块，它会从合适块大小的池子中取出一块，然后分配给你。

重用对象不会自动清除。

很多内存管理系统拥有debug特性，会清除或释放所有内存成特定的值，比如`0xdeadbeef`。 这帮助你找到使用未初始化变量或使用已被释放内存造成的痛苦漏洞。

由于对象池重用对象不再经过内存管理系统，我们失去了这层安全网。 更糟的是，为“新”对象使用的内存之前存储的是同样类型的对象。 这使你很难分辨出创建新对象时的未初始化问题： 那个存储新对象的内存已经保存了来自于上个生命周期中几乎完全正确的数据。

由于这点，特别注意在池里初始化对象的代码，保证它完全地初始化了对象。 甚至很值得加个在对象回收时清空对象槽的debug选项。

> 如果你将其清空为`0x1deadb0b`，我会很荣幸的。

未使用的对象会保留在内存中

对象池在支持垃圾回收的系统中很少见，因为内存管理系统通常会为你处理这些碎片。 但是对象池仍然是避免构建和析构的有用手段，特别是在有更慢CPU和更简陋垃圾回收系统的移动设备上。

如果你使用有垃圾回收的对象池系统，注意潜在的冲突。 由于池不会在对象不再使用时真正的析构它们，如果对象仍然保留任何对其他对象的引用，也会阻止垃圾回收器回收它。为了避免这点，当池中对象不再使用，清除它对其他对象的所有引用。

示例代码

现实世界的粒子系统通常应用重力，风，摩擦，和其他物理效果。 我们简陋的例子只在直线上特定帧移动粒子，然后销毁粒子。 这不是工业级的代码，但足够说明如何使用对象池。

我们应该从最简单的可能实现开始。首先是小小的粒子类：

```
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_;
};
```

默认的构造器将粒子初始化为“不在使用中”。之后对`init()`的调用初始化粒子到活跃状态。 粒子随着时间播放动画，一帧调用一次`animate()`函数。

对象池需要知道哪个粒子可以被重用。它通过粒子的`inUse()`函数获知这点。 这个函数利用了粒子生命时间有限这点，并使用变量`framesLeft_`来决定哪些粒子在被使用，无需存储分离的标识。

对象池类也很简单：

```
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];
};
```

`create()`函数允许外部代码创建新粒子。 游戏每帧调用`animate()`一次，让对象池中的粒子轮流显示动画。

> `animate()`方法是更新方法模式的一个例子。

粒子本身被存储在对象池类中一个固定大小的数组里。 在这个简单的实现中，池的大小在类声明时被硬编码了，但是也可以使用动态大小的数组或使用由外部定义的模板变量。

创建新粒子很直观：

```
void ParticlePool::create(double x, double y,
                          double xVel, double yVel,
                          int lifetime)
{
  // 找到一个可用粒子
  for (int i = 0; i < POOL_SIZE; i++)
  {
    if (!particles_[i].inUse())
    {
      particles_[i].init(x, y, xVel, yVel, lifetime);
      return;
    }
  }
}
```

我们遍历对象池找到第一个可用粒子。 当我们找到后，初始化它然后就完成了。 注意在这个实现中，如果这里没有找到任何可用的粒子，就不创建新的粒子。

做一个简单粒子系统的所有东西都在这里了，当然，没有包含渲染粒子。 我们现在可以创建对象池然后使用它创建粒子。当时间到了，粒子会自动失效。

这足够承载一个游戏了，但是敏锐的读者也许会注意到创建新粒子（可能）需要遍历整个集合，直到找到一个空闲槽。 如果池子很大很满，这可能很慢。 让我们看看可以怎样改进这一点。

> 创建一个粒子的复杂度是O(n)，上过算法课的人都知道。

空闲列表

如果不想浪费时间在查找空闲粒子上，明显的解决方案是不要失去对它们的追踪。 我们可以存储指向每个未使用粒子的单独指针列表。 然后，当需要创建粒子时，我们从列表中移除第一个指针，然后重用它指向的粒子。

不幸的是，这回要我们管理一个和对象池同样大小的单独数组。 无论如何，在我们创建池时，所有的 粒子都未被使用，所以列表初始会包含池中每个对象的指针。

如果无需牺牲任何内存就能修复性能问题那就好了。 方便的是，这里已经有可以借用的内存了——那些未使用粒子自身的内存。

当粒子未被使用时，它的大部分的状态都是无关紧要的。 它的位置和速度没有被使用。唯一需要的表示自身是否激活的状态。 在我们的例子中，那是`framesLeft_`成员。 其他的所有位都可以被重用。这里是改进后的粒子：

```
class Particle
{
public:
  // ...

  Particle* getNext() const { return state_.next; }
  void setNext(Particle* next) { state_.next = next; }

private:
  int framesLeft_;

  union
  {
    // 使用时的状态
    struct
    {
      double x, y;
      double xVel, yVel;
    } live;

    // 可重用时的状态
    Particle* next;
  } state_;
};
```

我们将除`framesLeft_`外的所有成员变量移到`live`结构中，而该结构存储在union`state_`中。 这个结构保存粒子在播放动画时的状态。 当粒子被重用时，union的其他部分，`next`成员被使用了。 它保留了一个指向这个粒子后面的可用粒子的指针。

> Unions近些年不那么常见了，所以你可能不熟悉这些语法。 如果你在游戏团队中，你可能会遇见“内存大师”，当游戏遇到不可避免的内存耗尽问题时，他们就挺身而出。 问问他们关于unions的事。 他们知道所有有关union的事情，还有其他有趣的位压缩技巧。

我们可以使用这些指针构建链表，将池中每个未使用粒子都连在一起。 我们有可用粒子的列表，而且无需使用额外的内存。 我们使用了死亡粒子本身的内存来存储列表。

这种聪明的技术被称为**freelist**。 为了让其工作，我们需要保证指针正确的初始化，在粒子创建和销毁时好好被管理了。 并且，当然，我们要追踪列表的头指针：

```
class ParticlePool
{
  // ...
private:
  Particle* firstAvailable_;
};
```

当首次创建对象池时，所有的 粒子都是可用的，所以空余列表应该贯穿整个对象池。对象池构造器设置了这些：

```
ParticlePool::ParticlePool()
{
  // 第一个可用的粒子
  firstAvailable_ = &particles_[0];

  // 每个粒子指向下一个
  for (int i = 0; i < POOL_SIZE - 1; i++)
  {
    particles_[i].setNext(&particles_[i + 1]);
  }

  // 最后一个终结的列表
  particles_[POOL_SIZE - 1].setNext(NULL);
}
```

现在为了创建新粒子，我们直接跳到首个可用的粒子：

```
O(1) 复杂度，孩子！这才叫编码！
```

```
void ParticlePool::create(double x, double y,
                          double xVel, double yVel,
                          int lifetime)
{
  // 保证池没有满
  assert(firstAvailable_ != NULL);

  // 将它从可用粒子列表中移除
  Particle* newParticle = firstAvailable_;
  firstAvailable_ = newParticle->getNext();

  newParticle->init(x, y, xVel, yVel, lifetime);
}
```

我们需要知道粒子何时死亡，这样可将其放回到空闲列表中， 所以我们将`animate()`改为在粒子不再活跃时返回`true`：

```
bool Particle::animate()
{
  if (!inUse()) return false;
```

```
  framesLeft_--;
  x_ += xVel_;
  y_ += yVel_;

  return framesLeft_ == 0;
}
```

当那发生时，简单地将其放回列表：

```
void ParticlePool::animate()
{
  for (int i = 0; i < POOL_SIZE; i++)
  {
    if (particles_[i].animate())
    {
      // 将粒子加到列表的前部
      particles_[i].setNext(firstAvailable_);
      firstAvailable_ = &particles_[i];
    }
  }
}
```

这样就成了，一个小对象池，拥有常量时间的构造和删除。

设计决策

如你所见，对象池最简单实现非常平凡：创建对象数组，在需要它们时重新初始化。 实际的代码很少会那么简单，这里还有很多方式让池更加的通用，安全，或容易管理。 在游戏中实现对象池时，你需要回答以下问题：

对象和池耦合吗？

第一个写对象池时需要思考的问题：是否对象本身需要知道它们在池子中。 大多数情况下它们需要，但是你不大可能写一个通用对象池类来保存任意对象。

- 如果对象与池耦合：
  - 实现更简单。 你可以在对象中简单地放个“在使用中”标识或者函数，就完成了。
  - 你可以保证对象只能被对象池创建。 在C++中，做这事最简单的方法是让池对象是对象类的友类，将对象的构造器设为私有。

```
class Particle
{
  friend class ParticlePool;

private:
  Particle()
  : inUse_(false)
  {}
```

```
  bool inUse_;
};

class ParticlePool
{
  Particle pool_[100];
};
```

在类间保持这种关系来确保使用者无法创建对象池没有追踪的对象。

- 你也许可以避免显式存储**“使用中”**的标识。 很多对象已经保存了可以告诉外界它有没有在使用的状态。 举个例子，粒子的位置如果不在屏幕上，也许它就可以被重用。 如果对象类知道它在对象池中，那它可以提供一个`inUse()`来查询这个状态。 这省下了对象池存储“在使用中”标识的多余内存。

- 如果对象没有和对象池耦合：
  - 可以保存多种类型的对象。 这是最大的好处。通过解耦对象和对象池，你可以实现通用的、可重用的对象池类。
  - 必须在对象的外部追踪**“使用中”**状态。 做这点最简单的方式是创建分离的位字段：

```
template <class TObject>
class GenericPool
{
private:
  static const int POOL_SIZE = 100;

  TObject pool_[POOL_SIZE];
  bool    inUse_[POOL_SIZE];
};
```

谁负责初始化重用对象？

为了重用一个已经存在的对象，它必须用新状态重新初始化。 这里的关键问题是你需要在对象池的内部还是外部重新初始化。

- 如果在对象池的内部重新初始化：
  - 对象池可以完全封装管理对象。 取决于对象需要的其他能力，你可以让它们完全处于池子的内部。 这保证了其外部代码不会引用到已重用的对象。
  - 对象池与对象是如何初始化的相绑定。 池中对象也许提供了不同的初始化函数。 如果对象池控制了初始化，它的接口需要支持所有的初始化函数，然后转发给对象。

```
class Particle
{
  // 多种初始化方式……
  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)
  {
    // 转发给粒子……
  }

  void create(double x, double y, double angle)
  {
    // 转发给粒子……
  }

  void create(double x, double y, double xVel, double yVel)
  {
    // 转发给粒子……
  }
};
```

- 如果外部代码初始化对象：
  - 对象池的接口更简单。 无需提供覆盖每种对象初始化的多种函数，对象池只需要返回新对象的引用：

```
class Particle
{
public:
  // 多种初始化方法
  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()
  {
    // 返回可用粒子的引用……
  }
private:
  Particle pool_[100];
};
```

调用者可以使用对象暴露的任何方法进行初始化：

```
ParticlePool pool;

pool.create()->init(1, 2);
pool.create()->init(1, 2, 0.3);
pool.create()->init(1, 2, 3.3, 4.4);
```

- 外部代码需要处理无法创建新对象的失败。 前面的例子假设`create()`总能成功地返回一个指向对象的指针。 但如果对象池已经满了，返回的会是`NULL`。安全起见，你需要在初始化之前检查这一点。

```
Particle* particle = pool.create();
if (particle != NULL) particle->init(1, 2);
```

参见

- 这看上去很像是享元 GoF模式。 两者都控制了一系列可重用的对象。不同在于“重用”的含义。 享元对象分享实例间同时拥有的相同部分。享元模式在不同上下文中使用相同对象避免了重复内存使用。

  对象池中的对象也被重用了，但是是在不同的时间点上被重用的。 “重用”在对象池中意味着对象在原先的对象用完之后分配内存。 对象池没有期待对象会在它的生命周期中分享什么。

- 将内存中同样类型的对象进行整合，能确保在遍历对象时CPU缓存总是满的。 数据局部性 模式介绍了这一点。

空间分区

意图

将对象根据它们的位置存储在数据结构中，来高效地定位对象。

动机

游戏让我们能拜访其他世界，但这些世界通常和我们的世界没有太多不同。 它们通常有和我们宇宙同样的基础物理和可理解性。 这就是我们为什么会认为这些由比特和像素构建的东西是真实的。

我们这里注意的虚拟事实是位置。游戏世界有空间感，对象都在空间的某处。 它用很多种方式证明了这点。最明显的是物理——对象移动，碰撞，交互——但是还有其他方式。 音频引擎也许会考虑声源和玩家的距离，越远的声音响声越小。 在线交流也许局限在较近的玩家之间。

这意味着游戏引擎通常需要回答这个问题，“哪些对象在这个位置周围？” 如果每帧都需要回答这个问题，这就会变成性能瓶颈。

在战场上的单位

假设我们在做实时战略游戏。双方成百上千的单位在战场上撞在一起。 战士需要挥舞刀锋向最近的那个敌人砍去。 最简单的处理方法是检查每对单位，然后看看它们互相之间的距离：

```
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]);
      }
    }
```

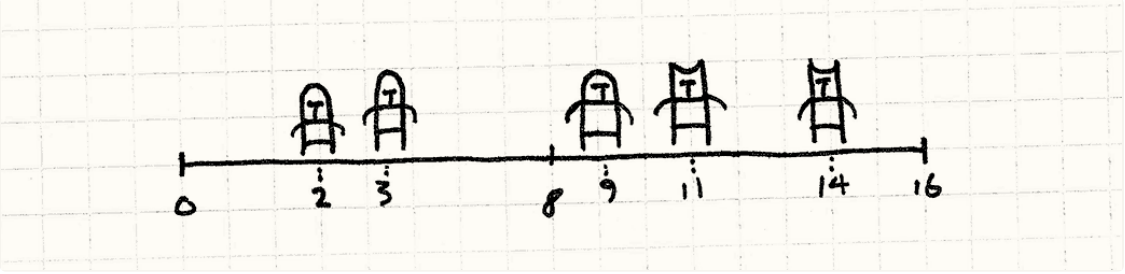
```
    }
}
```

这里使用的是双重循环，每个循环都会遍历战场上的所有单位。 这就是意味着每帧进行的检测对数会随着单位数量的平方增长。 每个附加单位都需要和之前所有单位的进行检查。如果有大量单位，这就完全失控了。

> 内层循环实际没有遍历所有的单位。 它只遍历那些外部循环还没有拜访的对象。这避免了比较一对单位两次：A与B一次，B与A一次。 如果我们已经处理了A和B之间的碰撞，我们不必为B和A再做一次。
>
> 用大O术语，这还是O(n²) 的。

描绘战线

我们这里碰到的问题是没有指明数组中潜藏的对象顺序。 为了在某个位置附近找到单位，我们需要遍历整个数组。 现在，我们简化一下游戏。 不使用二维的战场，想象这是个一维的战线。



在这种情况下，我们可以通过根据单位在战线上的位置排序数组元素来简化问题。 一旦我们那样做，我们可以使用像二分查找之类的东西找到最近的对象而不必扫描整个数组。

> 二分查找有O(log n) 的复杂度，意味着找所有战斗单位的复杂度从O(n²)降到O(n log n)。 像pigeonhole sort可将其降至O(n) 。

这里的经验很明显：如果我们根据位置存储对象在数据结构中，就可以更快的找到它们。这个模式便是将这个思路应用到多维空间上。

模式

对于一系列对象，每个对象都有空间上的位置。 将它们存储在根据位置组织对象的空间数据结构中，让你有效查询在某处或者某处附近的对象。 当对象的位置改变时，更新空间数据结构，这样它可以继续找到对象。

何时使用

这是存储活跃的，移动的游戏对象的常用模式，也可用于静态美术和世界地理。 复杂的游戏中，不同的内容有不同的空间分区。

这个模式的基本要求是一系列有位置的对象，而你做了太多的通过位置寻找对象的查询，导致性能下降。

记住

空间分区的存在是为了将**O(n)**或者**O(n²)** 的操作降到更加可控的数量级。 你拥有的对象越多，这就越重要。相反的，如果**n**足够小，也许不需要担心这个。

由于这个模式需要通过位置组织对象，可以改变位置的对象更难处理。 你需要重新组织数据结构来追踪在新位置的对象，这添加了更多的复杂性并消耗CPU循环。 保证这种交易是值得的。

> 想象一下哈希表，其中对象的键可以自动改变，那你就知道为什么这难以处理。

空间分区也会因为记录划分的数据结构而使用额外的内存。 就像很多优化一样，它用内存换速度。如果内存比时钟周期更短缺，这回是个错误的选择。

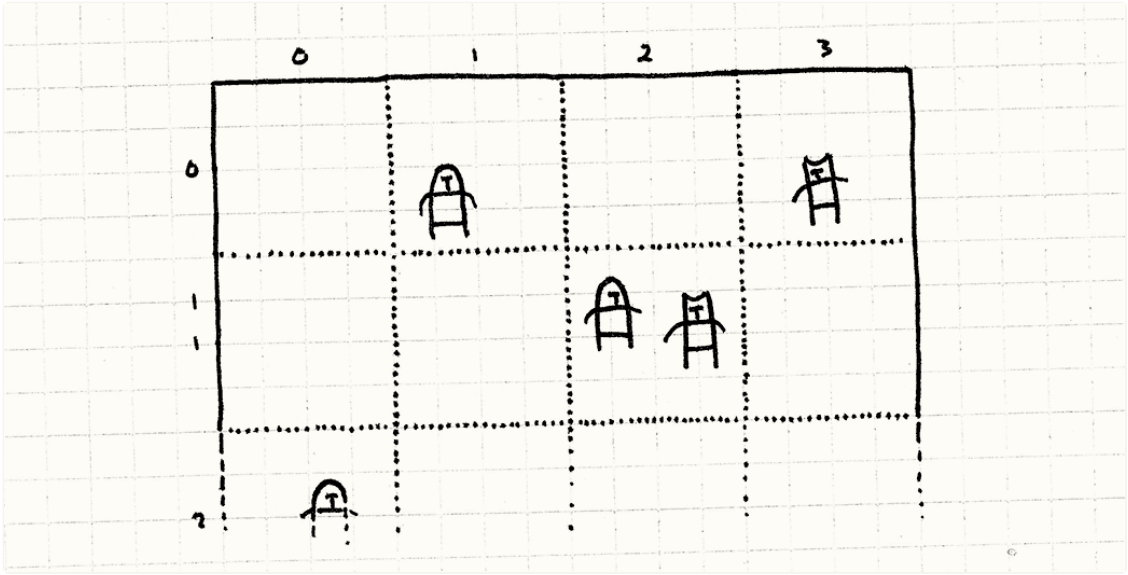
示例代码

模式总会变化——每种实现都略有不同，空间分区也不例外。 不像其他的模式，它的每种变化都很好地被记录下来了。 学术界发表文章证明各种变化各自的性能优势。 由于我只关注模式背后的观念，我会给你展示最简单的空间分区：固定网格。

> 看看本章的最后一节，那里有游戏中常用的空间分区方法列表。

一张网格纸

想象整个战场。现在，叠加一张方格大小固定的网格在上面，就好像一张网格纸。 不是在单独的数组中存储我们的对象，我们将它们存到网格的格子中。 每个格子存储一组单位，它们的位置在格子的边界内部。



当我们处理战斗时，我们只需考虑在同一格子中的单位。 不是将每个游戏中的单位与其他所有单位比较，我们将战场划分为多个小战场，每个格子中的单位都较少。

一网格相邻单位

好了，让我们编码把。首先，一些准备工作。这是我们的基础`Unit`类。

```
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_;
};
```

每个单位都有位置（2D表示），以及一个指针指向它存在的Grid。 我们让Grid成为一个friend类， 因为，就像将要看到的，当单位的位置改变时，它需要和网格做复杂的交互，以确保所有事情都正确的更新了。

这里是网格的表示：

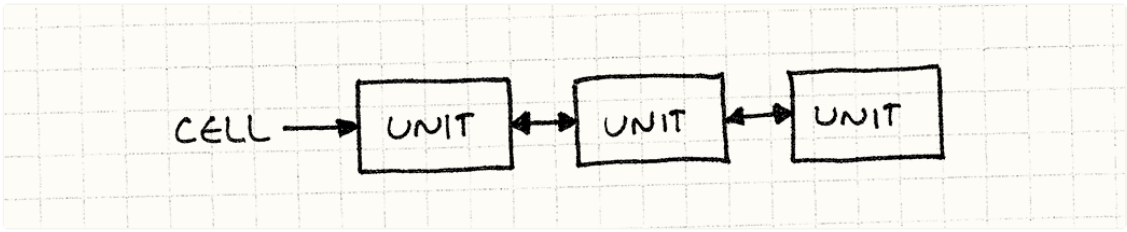
```
class Grid
{
public:
  Grid()
  {
    // 清空网格
    for (int x = 0; x < NUM_CELLS; x++)
    {
      for (int y = 0; y < NUM_CELLS; y++)
      {
        cells_[x][y] = NULL;
      }
    }
  }

  static const int NUM_CELLS = 10;
  static const int CELL_SIZE = 20;
private:
  Unit* cells_[NUM_CELLS][NUM_CELLS];
};
```

注意每个格子都是一个指向单位的指针。 下面我们扩展Unit，增加next和prev指针：

```
class Unit
{
  // 之前的代码……
private:
```

```
  Unit* prev_;
  Unit* next_;
};
```

这让我们将对象组织为双向链表，而不是数组。



每个网格中的指针都指向网格中的元素列表的第一个， 每个对象都有个指针指向它前面的对象，以及另一个指针指向它后面的对象。 我们很快会知道为什么要这么做。

> 在这本书中，我避免使用任何C++标准库内建的集合类型。 我想让理解例子的所需知识越小越好，然后，就像魔术师的“我的袖子里什么也没有”， 我想明晰代码中确实在发生什么。 细节很重要，特别是那些与性能相关的模式。
>
> 但这是我解释模式的方式。 如果你在真实代码中使用它们，使用内建在几乎每种程序语言集合避免麻烦。 人生苦短，不要浪费在编写链表上。

进入战场

我们需要做的第一件事就是保证新单位创建时被放置到了网格中。 我们让`Unit`在它的构造函数中处理这个：

```
Unit::Unit(Grid* grid, double x, double y)
: grid_(grid),
  x_(x),
  y_(y),
  prev_(NULL),
  next_(NULL)
{
  grid_->add(this);
}
```

`add()`方法像这样定义：

```
void Grid::add(Unit* unit)
{
  // 检测它在哪个网格中
  int cellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int cellY = (int)(unit->y_ / Grid::CELL_SIZE);

  // 加到网格的对象列表前段
  unit->prev_ = NULL;
  unit->next_ = cells_[cellX][cellY];
  cells_[cellX][cellY] = unit;

  if (unit->next_ != NULL)
```

```
  {
    unit->next_->prev_ = unit;
  }
}
```

世界坐标除以网格大小转换到了网格空间。 然后，缩短为`int`消去了分数部分，这样可以获得网格索引。

除了链表带来的繁琐，基本思路是非常简单的。 我们找到单位所在的网格，然后将它添加到列表前部。 如果那已经存在有列表单位了，我们把新单位链接到旧单位的后面。

刀剑碰撞

一旦所有的单位都放入网格中，我们可以让它们开始互相交互。 使用这个新网格，处理战斗的主要方法看上去是这样的：

```
void Grid::handleMelee()
{
  for (int x = 0; x < NUM_CELLS; x++)
  {
    for (int y = 0; y < NUM_CELLS; y++)
    {
      handleCell(cells_[x][y]);
    }
  }
}
```

它在每个网格它上面遍历并调用`handleCell()`。 就像你看到的那样，我们真的已经将战场分割为分离的小冲突。 每个网格之后像这样处理它的战斗：

```
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_;
  }
}
```

除了遍历链表的指针把戏，注意它和我们原先处理战斗的原始方法完全一样。 它对比每对

单位，看看它们是否在同一位置。

不同之处是，我们不必再互相比较战场上所有的单位——只与那些近在一个格子中的相比较。 这就是优化的核心。

> 简单分析一下，似乎我们让性能更糟了。 我们从对单位的双重循环变成了对格子内单位的三重循环。 这里的技巧是内部循环现在只在很少的单位上运行，这足够抵消在格子上的外部循环的代价。
>
> 但是，这依赖于我们格子的粒度。如果它们太小，外部循环确实会造成影响。

冲锋陷阵

我们解决了性能问题，但同时创建了新问题。 单位现在陷在它的格子中。 如果将单位移出了包含它的格子，格子中的单位就再也看不到它了，但其他单位也看不到它。 我们的战场有点过度划分了。

为了解决这点，需要每次单位移动时都做些工作。 如果它跨越了格子的边界，我们需要将它从原来的格子中删除，添加到新的格子中。 首先，我们给`Unit`添加一个改变位置的方法：

```
void Unit::move(double x, double y)
{
  grid_->move(this, x, y);
}
```

显而易见，AI代码可以调用它来控制电脑的单位，玩家也可以输入代码调用它来控制玩家的单位。 它做的只是交换格子的控制权，之后：

```
void Grid::move(Unit* unit, double x, double y)
{
  // 看看它现在在哪个网格中
  int oldCellX = (int)(unit->x_ / Grid::CELL_SIZE);
  int oldCellY = (int)(unit->y_ / Grid::CELL_SIZE);

  // 看看它移动向哪个网格
  int cellX = (int)(x / Grid::CELL_SIZE);
  int cellY = (int)(y / Grid::CELL_SIZE);

  unit->x_ = x;
  unit->y_ = y;

  // 如果它没有改变网格，就到此为止
  if (oldCellX == cellX && oldCellY == cellY) return;

  // 将它从老网格的列表中移除
  if (unit->prev_ != NULL)
  {
    unit->prev_->next_ = unit->next_;
  }
```

```
  if (unit->next_ != NULL)
  {
    unit->next_->prev_ = unit->prev_;
  }

  // 如果它是列表的头，移除它
  if (cells_[oldCellX][oldCellY] == unit)
  {
    cells_[oldCellX][oldCellY] = unit->next_;
  }

  // 加到新网格的对象列表末尾
  add(unit);
}
```

这块代码很长但也很直观。 第一步检查我们是否穿越了格子的边界。 如果没有，需要做的事情就是更新单位的位置，搞定。

如果单位已经离开了现在的格子，我们从格子的链表中移除它，然后再添加到网格中。 就像添加一个新单位，它会插入新格子的链表中。

这就是为什么我们使用双向链表——我们可以通过设置一些指针飞快地添加和删除单位。 每帧都有很多单位移动时，这就很重要了。

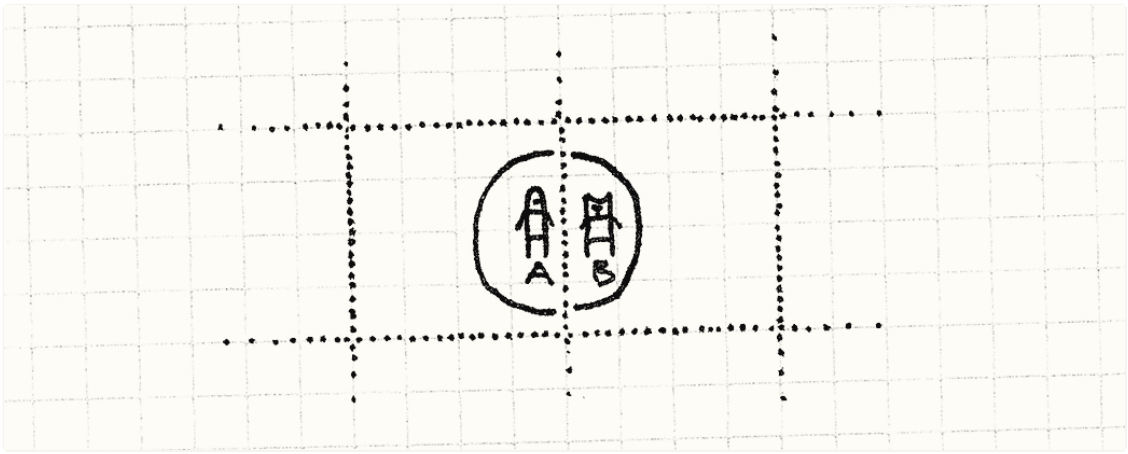
短兵相接

这看起来很简单，但我们某种程度上作弊了。 在我展示的例子中，单位在它们有完全相同的位置时才进行交互。 西洋棋和国际象棋中这是真的，但是对于更加实际的游戏就不那么准确了。 它们通常需要将攻击距离引入考虑。

这个模式仍然可以好好工作，与检查位置匹配不同，我们这样做：

```
if (distance(unit, other) < ATTACK_DISTANCE)
{
  handleAttack(unit, other);
}
```

当范围被牵扯进来，需要考虑一个边界情况： 在不同网格的单位也许仍然足够接近，可以相互交互。



这里，B在A的攻击半径内，即使中心点在不同的网格。 为了处理这种情况，我们不仅需要

比较同一网格的单位，同时需要比较邻近网格的对象。 为了达到这点，首先我们让内层循环摆脱handleCell()：

```
void Grid::handleUnit(Unit* unit, Unit* other)
{
  while (other != NULL)
  {
    if (distance(unit, other) < ATTACK_DISTANCE)
    {
      handleAttack(unit, other);
    }

    other = other->next_;
  }
}
```

现在有函数接受一个单位和一列表的其他单位看看有没有碰撞。 让handleCell()使这个函数：

```
void Grid::handleCell(int x, int y)
{
  Unit* unit = cells_[x][y];
  while (unit != NULL)
  {
    // 处理同一网格中的其他单位
    handleUnit(unit, unit->next_);

    unit = unit->next_;
  }
}
```

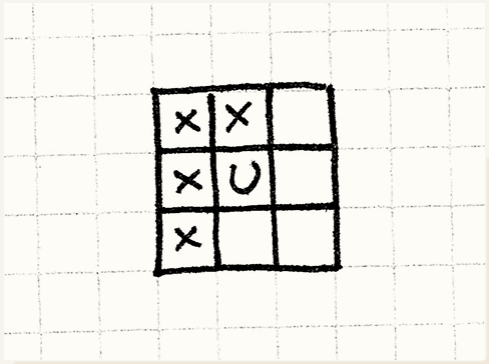
注意我们同样传入了网格的坐标，而不仅仅是对象列表。 现在，这也许和前面的例子没有什么区别，但是我们会稍微扩展一下：

```
void Grid::handleCell(int x, int y)
{
  Unit* unit = cells_[x][y];
  while (unit != NULL)
  {
    // 处理同一网格中的其他单位
    handleUnit(unit, unit->next_);

    // 同样检测近邻网格
    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_;
    }
  }
```

这些新增`handleCell()`调用在八个邻近格子中的四个格子中寻找这个单位是否与它们有任何对抗。 如果任何邻近格子的单位离边缘近到单位的攻击半径内，就找到碰撞了。

有单位的格子是U，它查找的邻近格子是X。



我们只查询一半的近邻格子，这原因和之前是一样的：内层循环从当前单位之后的单位开始——避免每对单位比较两次。 考虑如果我们检查全部八个近邻格子会发生什么。

假设我们有两个在邻近格子的单位近到可以互相攻击，就像前一个例子。 这是我们检查全部8个格子会发生的事情：

1. 当找谁打了A时，我们检查它的右边找到了B。所以记录一次A和B之间的攻击。
2. 当找谁打了B时，我们检查它的左边找到了A。所以记录第二次A和B之间的攻击。

只检查一半的近邻格子修复了这点。检查哪一半倒无关紧要。

我们还需要考虑另外的边界情况。 这里，我们假设最大攻击距离小于一个格子。 如果我们有较小的小格子和较长的攻击距离，我们也许需要扫描几行外的近邻格子。

设计决策

空间划分的优秀数据结构相对较少，可以一一列举进行介绍。 但是，我试图根据它们的本质特性来组织。 我期望当你学习四叉树和二分空间查找（BSPs）之类的时， 可以帮助你理解它们是如何工作，为什么 工作，以帮你选择。

划分是层次的还是平面的？

我们的网格例子将空间划分成平面格子的集合。 相反，层次空间划分将空间分成几个区域。 然后，如果其中一个区域还包含多个对象，再划分它。 这个过程递归进行，直到每个区域都有少于最大数量的对象在其中。

它们通常分为2,4,8块——程序员很熟悉这些数值。

- 如果是平面划分：
  - 更简单。 平面数据结构更容易想到也更容易实现。

    我在每章中都提到了这个设计要点，这是有理由的。尽可能使用简单的选

项。 大多数软件工程都是与复杂度做斗争。

- 内存使用量确定。 由于添加新对象不需要添加新划分，空间分区的内存使用量通常在之前就可以确定。
- 在对象改变位置时更新的更快。 当对象移动，数据结构需要更新，找到它的新位置。 使用层次空间分区，可能需要在多层间调整层次结构。

- 如果是层次性的：
  - 能更有效率的处理空的区域。 考虑之前的例子，如果战场的一边是空的。 我们需要分配一堆空白格子，这些格子浪费内存，每帧还要遍历它们。

    由于层次空间分区不再分割空区域，大的空区域保存在单个划分上。不需要遍历很多小空间，那里只有一个大的。
  - 它处理密集空间更有效率。 这是硬币的另一面：如果你有一堆对象堆在一起，无层次的划分很没有效率。 你最终将所有对象都划分到了一起，就跟没有划分一样。 层次空间分区会自适应地划成小块，让你同时只需考虑少数对象。

划分依赖于对象集合吗？

在示例代码中，网格空间大小事先被固定了，我们在格子里追踪单位。 另外的划分策略是自适应的——它们根据现有的对象集合在世界中的位置划分边界。

目标是均匀地划分，每个区域拥有相同的单位数量，以获得最好性能。 考虑网格的例子，如果所有的单位都挤在战场的一个角落里。 它们都会在同一格子中，找寻单位间攻击的代码退化为原来的$\mathbf{O(n^2)}$问题。

- 如果划分与对象无关：
  - 对象可以增量添加。 添加对象意味着找到正确的划分然后放入，这点可以一次性完成，没有任何性能问题。
  - 对象移动的更快。 通过固定的划分，移动单位意味着从格子移除然后添加到另一个。 如果划分它们的边界跟着集合而改变，那么移动对象会引起边界移动，导致很多其他对象也要移到其他划分。

    > 这可与如红黑树或AVL树这样的二叉搜索树相类比： 当你添加事物时，你也许最终需要重排树，并重排一堆节点。
  - 划分也许不均匀。 当然，固定的缺点就是对划分缺少控制。如果对象挤在一起，你就在空区域上浪费了内存，这会造成更糟的性能。

- 如果划分适应对象集合：

  像BSPs和k-d树这样的空间划分切分世界，让每部分都包含接近相同数目的对象。 为了做到这点，划分边界时，你需要计算每边各有多少对象。 层次包围盒是另外一种为特定集合对象优化的空间分区。
  - 你可以保证划分是平衡的。 这不仅提供了优良的性能表现，还提供了稳定的性能表现： 如果每个区域的对象数量保持一致，你可以保证游戏世界中的所有查询都会消耗同样的时间。 一旦你需要固定帧率，这种一致性也许比性能本身更重要。
  - 一次性划分一组对象更加有效率。 当对象集合影响了边界的位置，最好在划分前都出现所有对象。 这就是为什么美术和地理更多的使用这种划分。

- 如果划分与对象无关，但层次与对象相关：

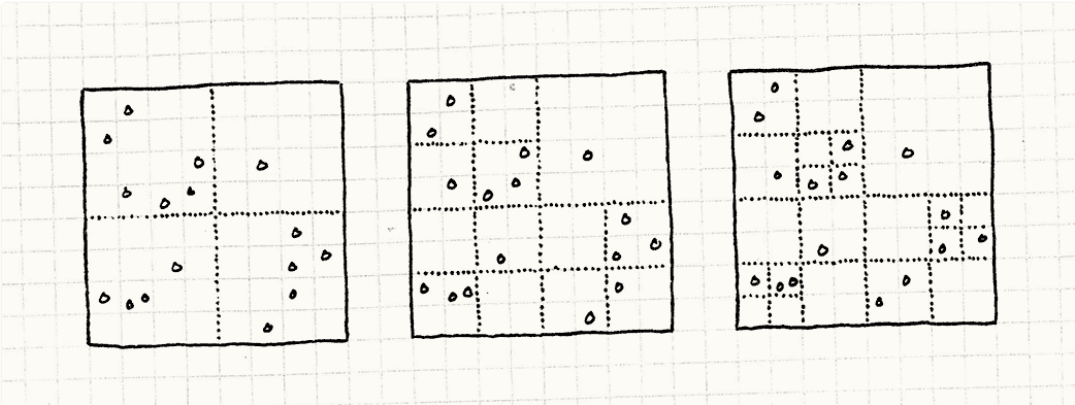
  有一种空间分区需要特殊注意，因为它拥有固定分区和适应分区两者的优点：四叉树。

  > 四叉树划分二维空间。它的三维实现是八叉树，获取“空间”，分割为8个正方体。 除了有额外的维度，它和平面划分一样工作。

  四叉树开始时将整个空间视为单一的划分。 如果空间中对象数目超过了临界值，它将其切为四小块。 这些块的边界是确定的：它们总是将空间一切为二。

  然后，对于四个区域中的每一个，我们递归地做相同的事情，直到每个区域都有较少数目的对象在其中。 由于我们递归地分割有较多对象的区域，这种划分适应了对象集合，但是划分本身没有移动。

  你可以从这里从左向右看到分区的过程：

  - 对象可以增量增加。 添加新对象意味着找到并添加到正确的区域。 如果区域中的对象数目超过了最大限度，就划分区域。 区域中的其他对象也划分到新的小区域中。 这需要一些小小的工作，但是工作总量是固定的： 你需要移动的对象数目总是少于数目临界值。添加对象从来不会引发超过一次划分。

    删除对象也同样简单。 你从它的格子中移除对象，如果它的父格子中的计数少于临界值，你可以合并这些子分区。

  - 移动对象很快。 当然，如上所述，“移动”对象只是添加和移除，两者在四叉树中都很快。

  - 分区是平衡的。 由于任何给定的区域的对象数目都少于最大的对象数量，哪怕对象都堆在一起，你也不会有包含太多对象的分区。

对象只存储在分区中吗？

你可将空间分区作为在游戏中对象存储的唯一地方，或者将其作为更快查找的二级缓存，使用另一个集合包含对象。

- 如果它是对象唯一存储的地方：

  - 这避免了内存开销和两个集合带来的复杂度。 当然，存储对象一遍总比存两遍来的轻松。 同样，如果你有两个集合，你需要保证它们同步。 每当添加或删除对象，都得从两者中添加或删除对象。

- 如果其他集合保存对象：

  - 遍历所有的对象更快。 如果所有对象都是“活的”，而且它们需要做些处理，也许会发现你需要频繁拜访每个对象而并不在乎它的位置。 想想看，早先的例子中，大多

数格子都是空的。访问那些空的格子是对时间的浪费。

存储对象的第二集合给了你直接遍历对象的方法。 你有两个数据结构，每种为各种的用况优化。

参见

- 在这里，我试图不讨论特定的空间分区结构细节来保证这章的高层概况性（而且节约篇幅！），但你的下一步应该是学习一下常见的结构。尽管名字很恐怖，它们都令人惊讶的直观。常见的有：
  - Grid
  - Quadtree
  - BSP
  - k-d tree
  - Bounding volume hierarchy
- 每种空间分区数据结构基本上都是将一维数据结构扩展成更高维度的数据结构。 知道它的相互关系有助于分辨它是不是问题的好解答：
  - 网格是连续的桶排序。
  - BSPs，k-d trees和包围盒是线性搜索树。
  - 四叉树和八叉树是多叉树。