Game Programming Patterns

Robert Nystrom

gb genever benning

The full text of this book lives online at
gameprogrammingpatterns.com.

This book was lovingly typeset by the author in Sina Nova, Source Sans Pro, and Source Code Pro. Layout is organized around three 1.9 inch columns with a 0.3 inch gutter. Text follows a 3.6 pt baseline grid.

ISBN: 978-0-9905829-0-8

To Megan, for faith and time,
the two essential ingredients.

Contents

Acknowledgements

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.

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 plenty of motivation.

What I didn't lose was a good 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!

— Bob Nystrom, September 6th, 2014

Introduction I.

Chapter 1: Architecture, Performance, and Games

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 (p. 73).

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.

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

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.

How it Relates to Design Patterns

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.

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").

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" (p. 19), 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" (p. 105), "Behavioral Patterns" (p. 153), "Decoupling Patterns" (p. 211), and "Optimization Patterns" (p. 267). 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!

Architecture, Performance, and Games

1

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.

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:

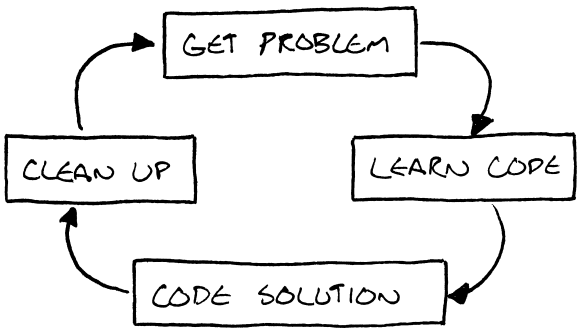

Figure 1.1 – Your workday in a nutshell

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?

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.

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 (p. 43) and messaging (p. 233) 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 (p. 269) 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

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.

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:

- We want nice architecture so the code is easier to understand over the lifetime of the project.
- We want fast runtime performance.
- 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.)

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.

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.

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.
- 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.**

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.

Design Patterns Revisited

II.

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 (p. 73)), while others are underappreciated (Command (p. 21)). A couple are here because I want to explore their relevance to games (Flyweight (p. 33) and Observer (p. 43)). Finally, sometimes I just think it's fun to see how patterns are enmeshed in the larger field of programming (Prototype (p. 59) and State (p. 87)).

Command 2

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

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. Look at it up there.

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.

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

"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.

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*.

never heard it, means "make real". Another term for reifying is making something "first-class".

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:

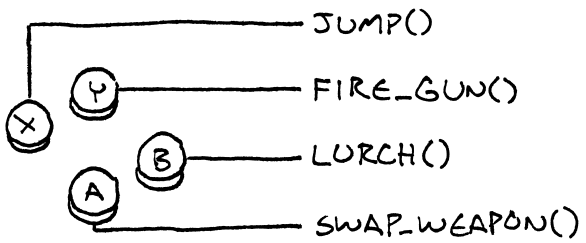

Figure 2.1 – Buttons mapped to game actions

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 (p. 123), 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.

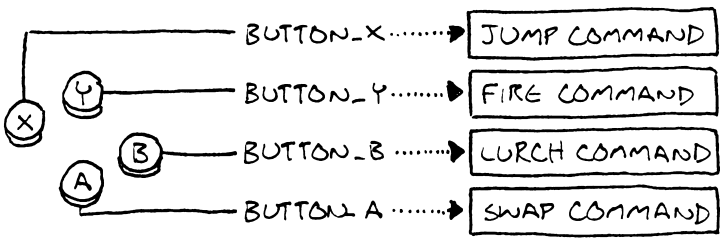

Figure 2.2 – Buttons mapped to assignable commands

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 (p. 233).

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

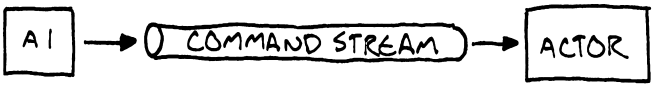

Figure 2.3 – A poorly drawn analogy

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.

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.

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_;
  int 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:

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

```
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;
}
```

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), x_(x), y_(y)
    xBefore_(0), yBefore_(0),
  {}

  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 x_, y_;
  int xBefore_, yBefore_;
};
```

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.

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.

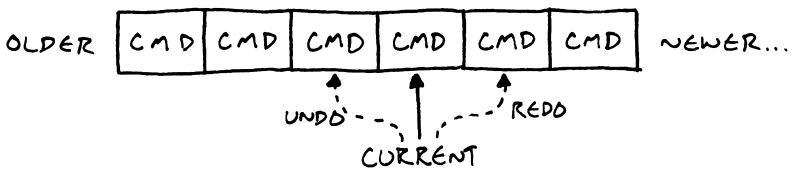

Figure 2.4 – Traversing the undo stack

This seems like a place for the Memento 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.

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 the Subclass Sandbox pattern (p. 181).
- 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 the Chain of Responsibility pattern.
- 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 pattern (p. 33) addresses that.

You could make it a singleton (p. 73) too, but friends don't let friends create singletons.