13

Clean Classes

by Jeff Langr

In Chapter 7, "Clean Functions," you learned how to best organize code statements and expressions in the form of functions or methods. Hopefully you learned these important things about clean functions:

- Functions provide a way to declare, invoke, and reuse pieces of logical behavior, or concepts.
- While nothing prevents you from aggregating multiple concepts into a single function, you're much better off if you compose methods that capture small, discrete concepts.

In this chapter, you'll hear echoes of the advice from Chapter 7 at the next-higher-up organizational construct: classes or modules.

Your programming language allows you to group concepts into a single construct—a class, if you're coding with an object-oriented language. If you program in a primarily functional language like JavaScript, Elixir, or F#, you might refer to a module instead.

Classes and Modules versus Files

In some object-oriented languages, such as Java, a file can declare only a single, publicly available class. Other languages, such as C++, TypeScript, and Ruby, allow you to declare multiple classes within a single file, or even split a single class between multiple files.

To make matters more confusing, some languages (both OO and functional) allow you to declare namespaces. These can help you avoid name clashes across a larger system. They can also help you organize related concepts.

Most functional languages—for example, Clojure, Elixir, and Haskell—allow you to organize functions into multiple, separate modules within a file. Even for languages that don't directly support modules/namespaces within the language (JavaScript comes to mind), you can organize multiple "virtual" modules within a single source file.

Just because you can doesn't mean you should. Generally, stick to a single module or class per source file. You'll create fewer headaches for yourself and your dev team.

Everything is a trade-off in software development, of course, and thus, exceptions always exist. You might find it useful to declare other very closely related classes within the same source file, such as small structs (e.g., records in Java) or exception types. Perhaps they are consumed only by other types within the same file, or they always appear in client code that also involves the file's primary type.

(And, of course, all this means little to those devious Smalltalk programmers who have the audacity to not organize their code into files at all.)

But, yeah, for the most part, class <=> file, or module <=> file, or whatever. Consider the file part of things mostly not relevant in our discussions in this chapter. We're going to focus instead on how to properly group concepts in classes and modules, irrespective of how they are deployed into files.

What Should a Class Contain?

You could conceivably lump together your system's entire set of methods into a single class—a monolith. You could, similarly perversely, create a separate class for each and every method. Both are outrageous ideas, though I've done the former for very small systems, and perhaps the second isn't so outrageous for some Smalltalkers.

Classes provide ways to help you keep your sanity as your system grows. Without sensible organization, you will

- Have a harder time finding the code you're looking for.

- Bloat your system with considerable redundancy; doubling it in size is a likely possibility.
- Create defects around inconsistent behaviors, due to the redundancies.
- Create defects due to increasing complexity.
- Spend extensive amounts of time writing automated tests for it.

All those outcomes increase in likelihood with growth, in turn increasing your costs and frustrations.

We'll talk about clean class design throughout this chapter. In your head, translate that to "doing all the things that should make our development lives easier."

"Clean" isn't a ding at anyone. Our notion of "unclean" is a way of characterizing code that demands more effort from the average developer to understand or maintain.

Characterizing Class Design

The quality of a design can only truly be assessed in the face of change; change that occurs each time the system must take on new features. Is it getting increasingly costly to introduce new features as the system grows and ages? Is the number of defects increasing?

Across seven decades of software development, we've gathered empirical data from our experiences with accommodating change. We've gleaned valuable insights from that data, and we've continually extrapolated those learnings into larger and newer contexts.

In the late 1940s, a typical system was implemented like our degenerate example, with the entire set of code essentially stored as a single function to be read and executed top to bottom. Let's call this a *monolith*.

John von Neumann and Herman Goldstine's article, "Planning and Coding of Problems for an Electronic Computing Instrument,"[1] explored the premise of a subroutine:

[1] H. H. Goldstine and J. von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," Institute for Advanced Study,

Princeton, New Jersey, 1947, archived at https://www.ias.edu/sites/default/files/library/pdfs/ecp/planningcodingof0103inst.pdf.

"We call the coded sequence of a problem a routine, and one which is formed with the purpose of possible substitution into other routines, a subroutine."

The premise was that subroutines (particularly for smaller, common operations) would cut down on the amount of code by allowing reuse, and would also help reduce defects.

Pioneers Maurice Wilkes and David Wheeler, inspired by ideas found in the seminal article, implemented the concept of subroutines in the EDSAC computer in 1951.[2] In short, they believed that code modularization brought about by subroutines would lead to less costly software development. They didn't wait for research (though it did come, slowly, across later decades).

[2]. [PPEDC].

Since the advent of subroutines, we've extrapolated the associated claimed value of modularization to larger contexts. We learned to organize collections of subroutines—functions or methods—into modules or classes. That, in turn, presented new challenges around the composition of such modules and classes, as well as the interrelationships (dependencies) between them. We began to derive various heuristics to guide other developers.

Heuristics and Characteristics

Heuristics help you identify the steps to take, and also the steps to avoid, to arrive at well-designed classes.

- Positive heuristics—steps to take. Kent Beck's four rules of emergent design,[3] for example, tell you to do the following.

[3]. See Chapter 18, "Simple Design." See also https://martinfowler.com/bliki/BeckDesignRules.html.

    - Ensure that all code is testable as you go.
    - Eliminate logical redundancies.
    - Ensure that all programmatic elements are clearly and concisely named.
    - Minimize the number of elements.
- Negative heuristics—steps to avoid. Antipatterns describe bad paths to take while developing code. Copy-paste programming, for example, tells us to not habitually create new logic by first duplicating existing code and then changing it. (A more memorable name for this antipattern might be "grab and stab." Or "snatch and patch." Or "copy and corrupt." Or … oh, that's enough for now. Just avoid it.)

Characteristics help you assess the current quality of your classes.

- Positive characteristics—desired traits. Bob Martin's collection of five class design principles, known as *SOLID*, have withstood around three decades of scrutiny, and can be practically applied to functional code as well.
- Negative characteristics—undesired traits. Martin Fowler's code smells[4] are a collection of things to avoid in code, each identified with a memorable name. Shotgun surgery, for example, means that your code forces you to make updates to numerous classes in order to effect simple changes.

[4]. [Refactoring] and https://martinfowler.com/bliki/CodeSmell.html.

Over time, folks in the software development arena have codified numerous collections of heuristics and characteristics. We'll refer to any of these as a design perspective.

Design perspectives can describe design at both the method (micro) level and the class (macro) level (or even higher levels). You might be able to consider some characteristics as heuristics, and vice versa; the categorization isn't terribly important.

For much of this chapter, we'll focus on the characteristics of an ideally sized class, because the idea is that you can glean them from looking at the code.

Such an ideal class is typically small. Its name concisely summarizes the small numbers of behaviors gathered within. And it is defined cohesively: Its methods are closely related and focused, and it performs a single, well-defined task (or set of related tasks) effectively. As a result, the class exhibits the characteristic of adhering to the Single Responsibility Principle (SRP): It has one reason to change.

Design perspectives most certainly overlap. Notably, outcomes for the more overarching perspectives—SOLID, Kent Beck's simple design, and Martin Fowler's code smells, for example—are comparable. They all foster small, cohesive modules and functions.

In nearly all of these perspectives, there's an acknowledgment, whether stated outright or implied, of the need for balance: Virtually no principle is absolute. Kent Beck's four rules of simple design, for example, drive us toward small, single responsibility classes. The resulting classes are easier to test (rule 1), they can help us eliminate redundancies more easily (rule 2), and they promote clear, intention-revealing names (rule 3). But rule 4—minimize the number of elements, for example, modules and functions—suggests that we can go too far and create tiny classes that add no advantages to the system.

Future Bob:

Indeed, this is one of John Ousterhout's fears regarding the "One Thing" rule.

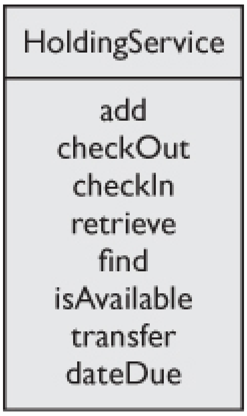
Most of us needn't worry: The typical system has moved too far in the wrong direction, containing large, muddled, multipurpose modules and functions. Still, keep in mind that there are trade-offs for every choice in software. Choose the design perspective(s) you like, and focus on the outcome of creating code that is easy to maintain.

When Is a Class Too Large?

Let's take a look at the `HoldingService` class, used to manage the holdings (materials, such as books or physical audio recordings) of a library system and its patrons. Here are the public behaviors defined on `HoldingService`:

```
boolean isAvailable(String barCode)
Holding find(String barCode)
String add(String sourceId, String branchId)
void transfer(String barcode, String branchScanCode)
Date dateDue(String barCode)
void checkOut(String patronId, String barCode, Date c
int checkIn(String barCode, Date date, String branchS
```

The following figure depicts that `HoldingService` interface (we'll be looking at more pictures later), using Unified Modeling Language (UML) diagramming guidelines:



The method parameters aren't usually interesting when we start looking at class-level design. We'll omit them until they become useful.

Our picture shows that the `HoldingService` allows holdings to be added to the system, borrowed ("checked out") by a patron, returned ("checked in") to a library location (branch), and transferred from one branch to another. The `HoldingService` also supports a few queries, including the ability to retrieve holding details, answer whether it's available for a patron to check out, and determine the date when the material is due to be returned.

The picture suggests that the core (single?) responsibility of the `HoldingService` is to manage the disposition of holdings. Each of the behaviors depicted directly supports that interest.

The SRP tells us to be a bit pickier, however: An SRP-compliant class should have only one reason to change. It's the opposite of a monolith, in which every single change requires updating the sole class in the system.

Why might a single-responsibility class be desirable?

- Its name can concisely reflect the behaviors contained within, making those behaviors easier to locate.
- It creates real possibilities for reuse.
- Its likelihood of adhering to the Open–Closed Principle (OCP; see Chapter 19), and thus being closed to future changes, increases.
- It's easier to test.
- It makes your system more flexible.

Each of the behaviors in `HoldingService` implements a policy—a set of rules established by the library system and implemented in the code. The `dateDue` policy, for example, specifies that books are due 21 days after they are checked out and that DVDs are due seven days after checkout.

The `checkIn` policy is more involved: When a book is checked in, it's marked as available for other patrons to check out. The patron returning the book is assessed a fine if material is late, that is, when it is returned after its due date. The fine amount varies depending on the material type and other possible factors.

Policies do change, and it's possible that the policy for check-outs might need to change exclusively from the policy for check-ins. Such independent variance suggests implementing each of the policies in its own class—`CheckOutService`, `CheckInService`, and so on—at least if you want to be SRP compliant. At that point, the only job of the `HoldingService` would be as a consolidated interface for holdings interests that delegated to other objects to do the work.

Future Jeff:

Creating new classes is surprisingly easy in an IDE, yet we often resist as developers. It takes only moments and has few downsides, but we often

pair with developers who do everything they can to avoid clicking a New Class menu item. Just do it.
We might not want a “ravioli” system where every class contains only one method, but there’s nothing wrong with tiny classes containing three or two methods, or even one.

While the SRP tells us to be picky, it also doesn’t tell us to speculate about the nexus of change. Your systems probably rampantly violate the SRP. Rather than go find code to change, wait for the next demand for change and ensure that you take that opportunity to shape your system to be more compliant.

Policies in Code

Let’s assume that the core library policies for check-in and check-out are stable. It’s easy to think of numerous reasons for `HoldingService` to potentially change. An example: The library wants to allow patrons to rent DVDs for a longer period—14 days instead of seven—because they’re an older technology and are in low demand nowadays.

Will `HoldingService` be impacted by that change? While looking at UML models can be very helpful, we can’t spot all change reasons by looking at UML alone.

The interface of a class only tells you about the behaviors it directly publicizes. To understand fully the extent to which a class violates the SRP, you have to open up the source file and peruse the code. It doesn’t take long to spot more reasons why a class might need to change.

Here’s the `dateDue` method:

```
public Date dateDue(String barCode) {
   var holding = find(barCode);
   if (holding == null)
      throw new HoldingNotFoundException();
   return holding.dateDue();
}
```

The service delegates the responsibility for calculating a due date to the `Holding` class. As such, the DVD rental period change won't impact the `HoldingService` class, but it will impact the `Holding` class (or maybe another class it depends on).

"Managing dates due" is a responsibility that could disappear, because some libraries have actually eliminated fines. If and when that occurs, we'll look to shield `HoldingService` from similar, subsequent changes.

Where Reasons to Change Hide

Large classes tend to contain and obscure numerous responsibilities. Often, change reasons are buried within the body of a long method. Let's look at an example.

The `HoldingService` method `checkIn` does a pretty good job of declaring the policy for returning materials:

```
public int checkIn(String barCode, Date date, String
   var branch = new BranchService().find(branchScanCc
   var holding = find(barCode);
   if (holding == null)
      throw new HoldingNotFoundException();

   holding.checkIn(date, branch);

   var foundPatron = findPatronWith(holding);
   removeBookFromPatron(foundPatron, holding);

   if (holding.dateLastCheckedIn().after(holding.date
      foundPatron.addFine(calculateLateFine(holding))
      return holding.daysLate();
   }
   return 0;
}
```

While rich in policy, the `checkIn` method still begs for a bit of cleanup. One line of glaring implementation detail stands out as a potential problem —the `if` statement that determines whether the check-in is late:

```
if (holding.dateLastCheckedIn().after(holding.dateDue
```

That conditional represents a policy decision: A holding is late if it's returned after the date due. However, it's possible that the policy will become considerably more complex. Librarians might imagine no end of new policy changes: VIPs (very important patrons) might be exempt from fines, the library might offer grace periods, and so on. As a result, that seemingly innocuous single line of code represents another reason for `HoldingService` to change.

Because it exposes implementation detail, the conditional also slows comprehension time for readers. The detail demands closer inspection: We must carefully read it, then mentally assemble it into a singular concept.

A comment might help, but it's not the right solution, and can be a lie anyway. It's best if we can glean intent from scanning the code itself, rather than having to look elsewhere. Comments are the distracting footnotes of code.

Fixing the Problem

We want to move the specifics of "is it late" out of the `HoldingService` class. Where to? We could move it to a new class such as `LateHoldingService`, but the obvious short-term candidate appears to be the `Holding` class.

Step 1, however, is abstracting the concept of "is it late" by creating a new method that contains just the conditional. Yep—this notion of extracting methods should be familiar from Chapter 10, "One Thing." Extract till you drop!

```
public int checkIn(String barCode, Date date, String
   // …
   if (isLate(holding)) {
      foundPatron.addFine(calculateLateFine(holding))
      return holding.daysLate();
   }
   return 0;
}
```

```
private boolean isLate(Holding holding) {
   return holding.dateLastCheckedIn().after(holding.c
}
```

The silly line-level comment `//late?` has disappeared. The new query method's name, `isLate`, precisely imparts the same information. The implementation detail is encapsulated within.

Going forward, if you feel you need a comment to guide readers through a line or five of related code, consider extracting it into its own method. The "guiding" comment often provides the basis for the new method's name, as it does here. A clear, concise method name obviates the need for a comment.

Predicates are particularly ripe for extraction. In fact, when faced with the challenge of trying to understand a long, daunting method, one of the best first steps you can take is to ensure that you understand the various paths through it (sometimes so that you can know which smaller piece you can focus on). Often the conditionals that control these pathways are complex, depending on combinations of method calls and variables— `if (x(a) && y || !z(c, d)`, for example. Those details don't make it clear why we're heading into a certain piece of code. An extracted predicate's method name can impart immediate understanding.

Once isolated within `HoldingService`, the `isLate` method reveals a glaring code smell known as *feature envy*[5]: The method, apparently envious of the `Holding` class, asks it multiple questions ("What's the date checked in? What's the date due?") in order to compute a result. The `isLate` method also shows disinterest in the `HoldingService` class on which it's defined.

We can soothe the method's envy by moving it to the `Holding` class, where it can talk directly to its new peers:

```
public class HoldingService {
// …
```

[5] [Refactoring].

```
   public int checkIn(
      String barCode, Date date, String branchScanCoc
      // …
      if (holding.isLate()) {
         foundPatron.addFine(calculateLateFine(holdir
         return holding.daysLate();
      }
      return 0;
   }
}
public class Holding {
   // …
   public boolean isLate() {
      return dateLastCheckedIn().after(dateDue());
   }
}
```

The calling code (in the `checkIn` method) now concisely states a piece of the overall policy:

```
if (holding.isLate())
```

The conditional is now immediately digestible. It doesn’t require us to stop and pause. Instead, it helps us quickly understand the method’s control flow, which, in turn, helps us know where we need to look next.

After moving the `checkIn` method to `Holding`, we look again at the method in its new context. Does the `Holding` class now contain too many reasons to change? Should we move the method again, perhaps this time to a brand-new class? What can be simplified within the method itself?

With every change, we continue to ask these kinds of questions as part of continuous design.

It might seem like we just passed the buck by moving a problem from `HoldingService` into `Holding`. But toward our primary interest in SRP-compliant classes, we’ve removed one reason to change from `HoldingService`. We’ve also replaced unnecessary detail with an abstraction, which, in turn, helps us more easily consider the remainder of `HoldingService`.

An Overly Open Implementation

Opportunities to extract and move are rampant in a typical codebase. Extracting implementation detail from a method often makes it obvious that the code belongs elsewhere. Do this enough, and eventually most code makes it to the right place, and each such place is reasonably small, cohesive, and easy to work with. Another line in the `checkIn` method similarly suggests feature envy:

```
foundPatron.addFine(calculateLateFine(holding));
```

We're calling a local method, `calculateFine`, that interacts with code defined in `Holding`, again asking it questions in order to calculate a fine. We move the method into `Holding` (with only two keystrokes in IDEA![6]):

```
foundPatron.addFine(holding.calculateLateFine());
```

Here's what `calculateLateFine` looks like in its new home:

```
public int calculateLateFine() {
   var daysLate = daysLate();
   var fineBasis = getMaterial().materialType().daily

   var fine = 0;
   switch (getMaterial().materialType()) {
      case BOOK, NEW_RELEASE_DVD:
         fine = fineBasis * daysLate;
         break;

      case AUDIO_CASSETTE, VINYL_RECORDING, MICRO_FIC
           AUDIO_CD, SOFTWARE_CD, DVD, BLU_RAY, VIDEC
         fine = Math.min(1000, 100 + fineBasis * days
         break;
   }
```

[6] The IDE from IntelliJ.

```
      return fine;
   }
```

Ugh, a `switch` statement, and an old-school one at that—the kind where control flow can accidentally drip from one case to the next if you're careless.

Even after being moved to `Holding`, the method still doesn't look at home. While `calculateLateFine` does interact directly with the `Holding` (asking it for `daysLate`), it predominantly interacts with the `Material` object returned by `getMaterial`.

The `calculateLateFine` method is the sort that will likely suffer many changes over time. It has at least three reasons to change: the addition of new material types (e-books, puzzles, STEM kits, etc.), new schemes to encourage patrons to return materials in high demand, and new rates to cover material price increases.

Such a method violates another SOLID principle—we want to minimize "opening up" existing classes to make changes, and instead find ways to enhance a system by adding new, single-purpose classes. This is the OCP.

Open classes introduce risk and cost. `Holding` is open. Changes to any of the three responsibilities in `calculateLateFine` carry the risk of breaking existing behaviors in `Holding`. With a drippy `switch` statement in the mix, anything can happen.

Should We Do Anything Now?

Typical systems being what they are, we already missed most of the opportunities to take advantage of the OCP. Our codebase is unashamed about its openness. Virtually no classes are closed. Rampant change greets everyone who must touch the code. If change is a given, the best approach is to wait for it. Speculative cleanup creates unnecessary risk from improvements no one yet needs. Let's wait for the next change.

What About Now?

OK, that didn't take too long. We were told, moments ago (between the prior paragraph and this one), that indeed we must support a couple of new material types. Specifically, we need to allow jigsaw puzzles and board games to be borrowed. They’ll fall under a new fine scheme the library folks want to try.

We’ll prepare for the new requirement by first factoring the code (prefactoring it) so that the change effort is smoother and has minimal impact.

Each of the two `switch` branches calculates an appropriate fine value using one of two strategies. We can extract the tiny bits of calculation logic to a couple of Strategy[7] classes, each implementing a common interface:

```
public interface LateStrategy {
   int calculateFine(int daysLate);
}

public class ConstrainedFineStrategy implements LateS
   public static final short BASE_FEE = 100;
   public static final short MAX_FINE = 1000;

   private final int fineBasis;

   public ConstrainedFineStrategy(int fineBasis) {
      this.fineBasis = fineBasis;
   }

   @Override
   public int calculateFine(int daysLate) {
      return Math.min(MAX_FINE, BASE_FEE + fineBasis
   }
}

public class DaysLateStrategy implements LateStrategy
   private final int fineBasis;

   public DaysLateStrategy(int fineBasis) {
      this.fineBasis = fineBasis;
   }
```

[7] [GOF95].

```
   @Override
   public int calculateFine(int daysLate) {
      return fineBasis * daysLate;
   }
}
```

What cute, tiny little classes! Here’s the updated `calculateLateFine` method.

```
public int calculateLateFine() {
   var daysLate = daysLate();
   var fineBasis = getMaterial().materialType().daily

   var fine = 0;
   switch (getMaterial().materialType()) {
      case BOOK, NEW_RELEASE_DVD:
         fine =
            new DaysLateStrategy(fineBasis).calculate
         break;

      case AUDIO_CASSETTE, VINYL_RECORDING, MICRO_FIC
           AUDIO_CD, SOFTWARE_CD, DVD, BLU_RAY, VIDEO
         fine = new ConstrainedFineStrategy(fineBasis
                   .calculateFine(daysLate);
         break;
   }
   return fine;
}
```

Note that `calculateLateFine` first retrieves the material type (e.g., `BOOK`) from the material, in order to decide which strategy to apply.

Here’s what `MaterialType` looks like:

```
public enum MaterialType {
   BOOK(21, 10),
   AUDIO_CASSETTE(14, 10),
   VINYL_RECORDING(14, 10),
   MICRO_FICHE(7, 200),
   AUDIO_CD(7, 100),
   SOFTWARE_CD(7, 500),
   DVD(3, 100),
   NEW_RELEASE_DVD(1, 200),
   BLU_RAY(3, 200),
   VIDEO_CASSETTE(7, 10);

   private final int checkoutPeriod;
   private final int dailyFine;

   MaterialType(int checkoutPeriod, int dailyFine) {
      this.checkoutPeriod = checkoutPeriod;
      this.dailyFine = dailyFine;
   }

   public int dailyFine() {
      return dailyFine;
   }

   public int checkoutPeriod() {
      return checkoutPeriod;
   }
}
```

If each material type can be initialized with the checkout period and late fine amounts, it can also be initialized with a Strategy object:

```
import domain.core.ConstrainedFineStrategy;
import domain.core.DaysLateStrategy;
import domain.core.LateStrategy;

public enum MaterialType {
   BOOK(21, new DaysLateStrategy(10)),
   AUDIO_CASSETTE(14, new ConstrainedFineStrategy(10)
   VINYL_RECORDING(14, new ConstrainedFineStrategy(1
   MICRO_FICHE(7, new ConstrainedFineStrategy(200)),
   AUDIO_CD(7, new ConstrainedFineStrategy(100)),
   SOFTWARE_CD(7, new ConstrainedFineStrategy(500)),
   DVD(3, new ConstrainedFineStrategy(100)),
   NEW_RELEASE_DVD(1, new DaysLateStrategy(200)),
   BLU_RAY(3, new ConstrainedFineStrategy(200)),
   VIDEO_CASSETTE(7, new ConstrainedFineStrategy(10))

   private final int checkoutPeriod;
   private final LateStrategy lateStrategy;

   MaterialType(int checkoutPeriod, LateStrategy late
      this.checkoutPeriod = checkoutPeriod;
      this.lateStrategy = lateStrategy;
   }

   public int checkoutPeriod() {
      return checkoutPeriod;
   }

   public int calculateFine(int daysLate) {
      return lateStrategy.calculateFine(daysLate);
   }
}
```

(Yes we could use Singletons[8] for each `LateStrategy` derivative, particularly since the subclasses contain no data. We could also specify a class type for each `enum` value, then use reflection to instantiate the Strategy object. But why make things more complicated?)

[8]. [GOF95].

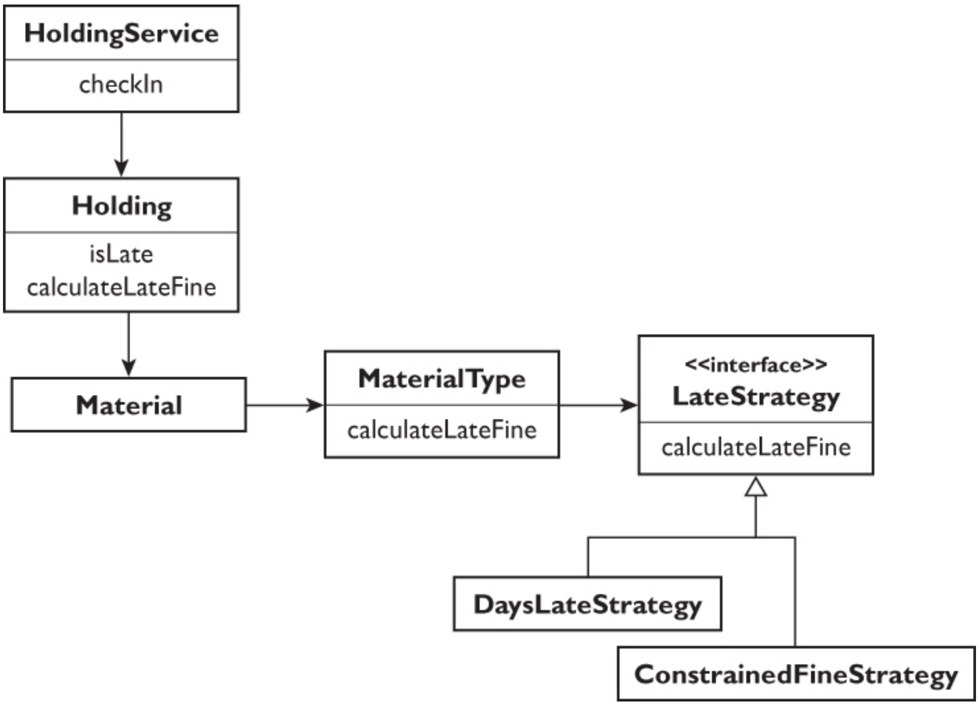
`MaterialType` can then supply a `calculateFine` implementation that delegates to the `lateStrategy` object:

```
public int calculateFine(int daysLate) {
   return lateStrategy.calculateFine(dailyFine, daysL
}
```

The best part? At this point, the `Holding` method `calculateLateFine` can simplify to a single line of code that delegates to `MaterialType`:

```
public int calculateLateFine() {
   return getMaterial().materialType().calculateFine(
}
```

The following shows a picture of the updated solution:



Closed, Cohesive, Single-Responsibility Classes

We've been tasked with supporting a new feature. For jigsaw puzzles and board games, the library board has decided to test out a new fine scheme. We test-drive the corresponding Strategy class:

```
public class DegradingFineStrategy implements LateStr
   private final int fineBasis;
   private final double degradationRate;

   public DegradingFineStrategy(
      int fineBasis, double degradationRate) {
      this.fineBasis = fineBasis;
      this.degradationRate = degradationRate;
   }

   @Override
   public int calculateFine(int daysLate) {
      double total = fineBasis *
         (1 - Math.pow(1 - degradationRate, daysLate)
         degradationRate;
      return (int) Math.round(total);
   }
}
```

Here are the tests, in case you thought we were only pretending to write them:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assert

class DegradingFineStrategyTest {
   int initialFine = 1000;
   double degradationRate = 0.1d;
   DegradingFineStrategy strategy =
      new DegradingFineStrategy(initialFine, degradat

   @Test
   void firstDayIsFineBasis() {
      assertEquals(initialFine, strategy.calculateFir
   }

   @Test
   void nextDayReducedUsingDegradationRate() {
      assertEquals(1000 + 900, strategy.calculateFine
   }

   @Test
   void multipleDays() {
      assertEquals(1000 + 900 + 810 + 729 + 656,
         strategy.calculateFine(5));
   }
}
```

For now, we must add a line to the `MaterialType` class to accommodate the new category:

```
public enum MaterialType {
   GAMES_AND_PUZZLES(21, new DegradingFineStrategy(5(
   BOOK(21, new DaysLateStrategy(10)),
   AUDIO_CASSETTE(14, new ConstrainedFineStrategy(10)
   // …
}
```

As a result of having to modify an existing class, our system isn't quite fully closed with respect to accommodating new fine schemes. But we realize that it wouldn't be difficult to replace the `MaterialType enum` with a class that reads the necessary information from a database. Once built, our system would be fully OCP compliant: Code a new class, seed the database with sufficient configuration data, and go—no changes to existing classes required!

When Policies Change

Many service classes must orchestrate numerous disparate behaviors. They interact with business logic regarding various domains (holdings, patrons, branches), they might persist or retrieve data, and they might interact with external services. The `HoldingServiceClass` is one example of an orchestrating class.

Some services might have minimal orchestration needs. An example is a collection of behaviors concerned with date calculations.

A well-defined SRP-compliant service, as such, should either:

- Contain only orchestration logic that declares policy and delegates to other classes for implementation specifics.
- Contain only implementation specifics for related domain or utility behaviors.

In other words, don't mix policy with implementation detail. Doing so is costlier to understand, navigate, and maintain. It's also not SRP compliant: A policy change is one reason to make a code change, and implementation details about a step in the policy is another reason.

Is This Overengineering?

What would happen if we took the SRP and the OCP to the extreme? Virtually everything in our system would be "plug and play." In a service, we wouldn't have a single controller class with a half dozen routes; we'd create a separate class for each route. It would register itself (perhaps like what an `@PostMapping` annotation accomplishes in Spring Boot). Its sole job would be to delegate requests to a similarly single-purpose service class (or to reject them as a result of validation, which is likewise implemented elsewhere). Each microservice facet would represent either a policy—whereby all specifics are delegated to—or a chunk of code that accomplished a singular need.

And so on for each class down the line. Pretty much all conditionals would be implemented polymorphically; that is, both branches would be implemented in separate classes, much like states in a State[9] pattern. Each step in a policy (a workflow) would be implemented in a separate class as well.

Ridiculous? It may sound like it, but such a system would truly support the OCP, whereby all new behavior was accomplished by dropping a new class in place. All classes would be small and easily tested in isolation. You’d write tests for each new class and perhaps would update an end-to-end test. Otherwise, you’d make no changes to the system.

Almost all systems don’t look like this, and they are harder to work with as well as riskier.

A highly SRP/OCP-compliant system can be initially challenging for developers not familiar with the idea—it takes a while to figure out where things are at first. (“There’s all these objects and none of ‘em are doing anything!”) Long term, however, it dramatically streamlines the change process.

It’s not a new idea. We worked with systems approaching this ideal 30 years ago in Smalltalk. Most classes were small, and the majority of methods were one to three lines of code each. Life was bliss.

As with most principles, consider both the SRP and the OCP as ideals: You want to almost always move in their direction, not away from them. It might be OK that you’ll never hit the ideal. The typical system, however, moves in the opposite direction, degrading toward larger, more monolithic classes. That’s why they’re harder to work with as they grow.

For now, forgive your system for its flaws. Moving forward, assess each change you make—how many existing classes did you have to open for change? Think of a reshaped, SRP/OCP-compliant system that would have instead accommodated that new feature as an extension of the existing code. Expend at least some effort to move your system in that direction.

[9] [GOF95].

Seek to create smaller, single-purpose classes. They're easier to work with (test, understand, change), and they're easier to close.

Seek to close classes. They're something we can consider "out of sight, out of mind." We seek to have less code that we must concern ourselves with.

Simpler Testing

You could choose to retain whatever tests you had in place for `HoldingService`. When adding the new behavior for fines, you'd add to these tests. That's one acceptable route.

However, you can also distribute the testing as well. Yes, the individual fine policies aren't publicly exposed to the ultimate client, and might be considered "private behavior." But the policies are also isolated, closed concepts that can be reused—and more easily tested than within the larger context of `HoldingService`. They are also public in the sense that they could conceivably be consumed by a different client.

Creating unit tests focused directly on a now-public fine policy Strategy class makes it officially documented. Such focused unit tests describe all the intentional behavior that developers coded in the class. Clean (clear, concise, and well-abstracted) unit tests provide far more useful documentation than method-level doc comments: We can trust them, as long as the tests pass, and we can see exactly how to interact with the class as a direct client.

We avoid testing implementation details. We test the outcomes of (small) behavioral concepts instead. Small, closed units should be concepts that can be moved about and work in any context where they're useful. A days-late fine policy can be tested in isolation without exposing any implementation specifics. It is a "sub-behavioral unit."

Corollary: If you extract to a small, single-purpose class, but don't view it as useful in any other context, don't make it publicly accessible. You'll still need to write tests that indirectly involve these classes at a higher level. Test such private details directly only as a last resort.

Our new fine strategy took less than 20 minutes to test-drive—we created both unit tests and the production code in that time. We didn't have to worry about changing tests for, or adding tests to, an existing class.

Enter AI

We (humans) are and have been advancing by orders of magnitude in many ways, technologically, every five to ten years or so for the past 90 years. AI will likely continue that amazing trend, accelerating us toward the singularity by the 2040s. Per Ray Kurzweil, "We are now entering the steep part of the exponential."[10]

[10] [Singularity], pp. 164–165.

Today, we can ask an LLM to produce code at the class level, and it will do an OK job, maybe getting 80% of the code correct. (There's that good, old 80/20 rule again.[11]) We can help an LLM do a better job by:

[11] https://www.investopedia.com/terms/1/80-20-rule.asp

- Providing it with examples
- Providing it with a simple style that tells it to (a) produce small, single-purpose modules/classes and small, single-purpose functions/methods, (b) produce more-functional solutions with less state management and side effects, and (c) emphasize clarity

What's cool is that this simple style matches how we've been told to code all along. It's also the style that has the broadest support, with few serious detractors. We have concepts like SOLID and an exhaustively examined body of work in design patterns; these things have been around for decades now.

It Will Be Wrong

Unfortunately, we cannot trust that an LLM will get 100% of the code correct. The easy solution is to have it generate tests from the examples provided, then vet the tests for fidelity with our examples, then run them.

In fact, comprehensive testing for behavioral intent is *essential* when using AI. AI will quickly generate more code than has been produced by humans to this point in history. It should create fear in everyone that 20% of that code is likely to be defective.

When the AI gets the code wrong—and it will—we need to have a conversation with it. That conversation is far more effective with a clean design: The source of the problem will be easier to pinpoint, and the resolution can involve focusing on that one small class or method. (If it's a method, sometimes the best route is to have the AI extract what's known as a *method object*—a new class that embodies the single troubled method.[12])

While the current trend is to view AI as most useful as a line-by-line coding assistant, it's already at the point where it's more effective to have it produce code at a modular level. At this point, the code content of the module matters far less. Need to change a class? Pull up the list of examples, update them, and regenerate both the tests and production code. That's not so easy if the modules aren't SRP compliant.

Don't get us wrong. We love mucking with code on a line-by-line basis. We are gratified by streamlining functions to make them elegant—that is, clear and concise. But ultimately, the contents of any well-composed function are irrelevant once we begin generating wholesale modules.

Our focus begins to shift. We concern ourselves with the overall architecture/design of the system, and we seek to create a system based on the OCP and SRP. When we need new behavior, we generate the appropriate module using AI, we vet the tests it generates, and we drop it into place.

Our system designs once again take center stage. We find value in employing time-honored design patterns that directly speak toward pluggable designs: Functional Pipeline, Command, Factory, Decorator, Strategy, Composite, Chain of Responsibility, and Bridge.

[12] [WELC].

Our focus on clean code remains important for the near future. At some point, AI will be able to completely implement an entire system from a set of examples/tests. We’ll be happy to have retired by then. You’ll still have opportunities as someone who knows how to design and deliver a system as an orchestration of well-tested, AI-implemented pieces.