18

Simple Design

by Kent Beck

This chapter is an abridged and edited excerpt from the book Clean Craftsmanship.

It should be obvious, on the face of it, that the best design for a system is the simplest design that supports all the required features of that system while simultaneously affording the greatest flexibility for change. However, that leaves us to ponder the meaning of simplicity.[1] Simple does not mean easy. Simple means untangled; and untangling things is *hard*.

---

1. In 2012, Rich Hickey gave a wonderful talk titled “Simplicity Made Easy.” I encourage you to listen to it.

What things get tangled in software systems? The most expensive and significant entanglements are those that convolve high-level policies with low-level details. You create terrible complexities when you conjoin SQL

with HTML, or frameworks with core values, or the format of a report with the business rules that calculate the reported values. These entanglements are easy to write, but they make it hard to add new features, hard to fix bugs, and hard to improve and clean the design.

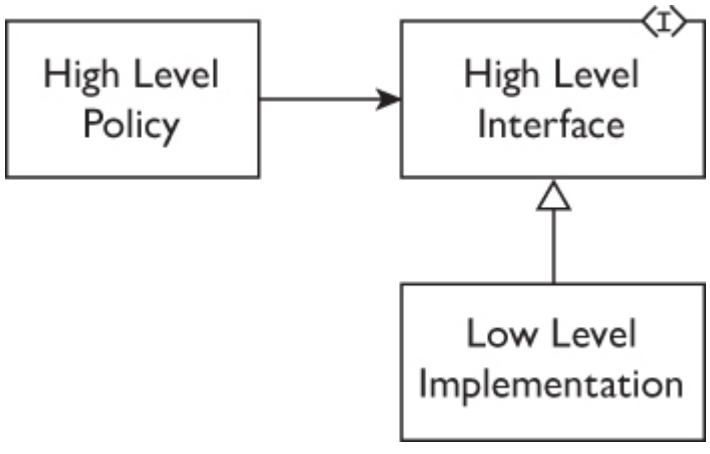
A simple design is a design in which high-level policies are ignorant of low-level details. Those high-level policies are sequestered and isolated from low-level details such that changes to the low-level details have no impact upon the high-level policies.[2]

---

2. I write a great deal about this in [Clean Arch].

The primary means for creating this separation and isolation is *abstraction*. Abstraction is the amplification of the essential and the elimination of the irrelevant. High-level policies are essential, so they are amplified. Low-level details are irrelevant at that level, and so they are isolated and sequestered.

One of the physical means we employ for this abstraction is polymorphism. We arrange high-level policies to use polymorphic interfaces to manage the low-level details. Then, we arrange the low-level details as implementations of those polymorphic interfaces. This keeps all source code dependencies pointing from low-level details to high-level policies and keeps high-level policies ignorant of the implementations of the low-level details. Low-level details can be changed without affecting the high-level policies. This is the essence of the Dependency Inversion Principle (DIP).



Pursuant to the goal of creating good software designs, we will first discuss YAGNI and then Kent Beck's four principles of simple design:

1. Covered by tests
2. Reveals intent
3. Minimizes duplication
4. Minimizes size

YAGNI

What if you aren't gonna need it*?*

In 1999, I was teaching an Extreme Programming course with Martin Fowler, Kent Beck, Ron Jeffries, and a host of others. The topic turned to the dangers of overdesign, and premature generalization. Someone wrote YAGNI on the whiteboard and said, "You aren't gonna need it." Kent Beck interrupted and said something to the effect of you might, in fact, need it; but you should ask yourself: "What if you aren't going to need it?"

That was the original question that YAGNI posed: Every time you thought to yourself "I'm going to need this hook," you then asked yourself what would happen if you didn't put the hook in. If the cost of leaving the hook out was tolerable, then you probably shouldn't put it in. If the cost of carrying the hook in the design year after year would be high while the odds that you'd eventually need that hook were low, you probably shouldn't put that hook in.

It is always wise to think of the future, and there are times when putting a particular hook in is a good idea. But be sure to count the costs and advantages on both sides of the argument. Make sure you ask yourself the question: *What if I'm not gonna need it?*

This kind of discernment is one of the most important disciplines of good software design. If you have a good suite of tests and you are skilled at the discipline of refactoring, then the cost of adding a new feature and updating the design to support that new feature will very often be smaller than the cost of implementing and maintaining all the hooks you might need one day.

Covered by Tests

What is a good test coverage number: 80%? 90%? Many teams treat such numbers as laudable goals. I disagree. The only reasonable coverage goal is 100%.

What other number could possibly make sense? If you are happy with 80% coverage, it means you don't know if 20% of your code works. How could you possibly be happy with that? How could your customers be happy with that?

An Asymptotic Goal

You might complain that 100% is an unreachable goal. I might even agree with you. Achieving 100% line and branch coverage is no mean feat. It may, in fact, be impractical depending on the situation. But that does not mean that the coverage you currently have cannot be improved.

Think of 100% as an asymptotic goal. You may never reach it, but that's no excuse for not trying to get closer and closer with every check-in.

I have personally participated in projects that grew to many tens of thousands of lines of code while constantly keeping the code coverage percentage in the very high 90s.

Design?

But what does high code coverage have to do with simple design? Why is coverage the first rule?

Testable code is decoupled code.

In order to achieve high line and branch coverage of each individual part of the code, each of those parts must be made accessible to the test code. That means those parts must be so well decoupled from the rest of the code that they can be isolated and invoked from an individual test. Therefore, those tests are not only tests of behavior, *they are also tests of decoupling*. The act of writing isolated tests is an act of design, because the code being tested must be designed to be tested.

Tests don't just drive you to create decoupled and robust designs. They also allow you to improve those designs with time. As we have discussed many times before in these pages, a trusted suite of tests vastly reduces the fear of change. If you have such a suite, and if that suite executes quickly, then you can improve the design of the code every time you find a better approach. When the requirements change in a way that the current design does not easily afford, the tests will allow you to fearlessly shift the design to better match those new requirements.

And this is why this rule is the first and most important rule of simple design. Without a suite of tests that covers the system, the other three rules become impractical, because those rules are best applied *after the fact*. Those other three rules are rules that involve refactoring. And refactoring is virtually impossible without a good, comprehensive suite of tests.

Maximize Expression

In the early decades of programming, the code we wrote could not reveal intent. Indeed, the very name "code" suggests that intent is obscured. Back in the late '60s, code looked like this:

```
-------

/ROUTINE TO TYPE A MESSAGE                    PAL8-V10D

             /ROUTINE TO TYPE A MESSAGE
       0200          *200
       7600          MONADR=7600
00200  7300  START,   CLA CLL      /CLEAR ACCUMULATOR
00201  6046           TLS          /CLEAR TERMINAL FL
00202  1216           TAD BUFADR   /SET UP POINTER
00203  3217           DCA PNTR     /FOR GETTING CHARA
00204  604i  NEXT,    TSF          /SKIP IF TERMINAL
00205  5204           JMP .-1      /NO: CHECK AGAIN
00206  1617           TAD I PNTR   /GET A CHARACTER
00207  6046           TLS          /PRINT A CHARACTER
00210  2217           ISZ PNTR   /DONE YET?
00211  7300           CLA CLL      /CLEAR ACCUMULATOR
00212  1617           TAD I PNTR   /GET ANOTHER C8ARA
00213  7640           SZA CLA      /JUMP ON ZERO AND
00214  5204           JMP NEXT     /GET READY TO PRIN
```

```
00215  5631           JMP I MON    /RETURN TO MONITOR
00216  0220  BUFADR,  BUFF         /BUFFER ADDRESS
00217  0220  PNTR,    BUFF         /POINTER
00220  0215  BUFF,    215;212;"H;"E;"L;"L;"O;"!;0
00221  0212
00222  0310
00223  0305
00224  0314
00225  0314
00226  0317
00227  0241
00230  0000
00231  7600  MON,     MONADR       /MONITOR ENTRY POI
```

Notice the ubiquitous comments. These were absolutely necessary because the code itself revealed nothing at all about the intent of the program.

However, we do not work in the 1960s anymore. The languages that we use are *immensely* expressive. With the proper discipline, we can produce code that reads like “well-written prose.”

As an example of such code, consider this little bit of Java:

```
public class RentalCalculator {
  private List<Rental> rentals = new ArrayList<>();

  public void addRental(String title, int days) {
    rentals.add(new Rental(title, days));
  }

  public int getRentalFee() {
    int fee = 0;
    for (Rental rental : rentals)
      fee += rental.getFee();
    return fee;
  }

  public int getRenterPoints() {
    int points = 0;
    for (Rental rental : rentals)
      points += rental.getPoints();
    return points;
```

```
    }
  }
```

If you were not a programmer on this project, you might not understand everything that is going on in this code. However, after even the most cursory glance, the basic intent of the designer is easy to identify. The names of the variables, functions, and types are deeply descriptive. The structure of the algorithm is easy to see. This code is expressive. This code is simple.

The Underlying Abstraction

Lest you think that expressivity is solely a matter of nice names for functions and variables, I should point out that there is another concern: the separation of levels and the exposition of the underlying abstraction.

A software system is expressive if each line of code, each function, and each module lives in a well-defined partition that clearly depicts the level of the code and its place in the overall abstraction.

You may have found that last sentence difficult to parse, so let me be a bit clearer by being much more long-winded.

Imagine an application that has a complex set of requirements. The example I like to use is a payroll system.

- Hourly employees are paid every Friday, based upon the time cards they have submitted. They are paid time-and-a-half for every hour they work after 40 hours in a week.
- Commissioned employees are paid on the first and third Friday of every month. They are paid a base salary plus a commission on the sales receipts they have submitted.
- Salaried employees are paid on the last day of the month. They are paid a fixed monthly salary.

It should not be hard for you to imagine a set of functions with a complex `switch` statement or `if` / `else` chain that captures these requirements. However, such a set of functions is likely to obscure the underlying abstraction. What is that underlying abstraction?

```
public List<Paycheck> run(Database db) {
  Calendar now = SystemTime.getCurrentDate();
  List<Paycheck> paychecks = new ArrayList<>();

  for (Employee e : db.getAllEmployees()) {
    if (e.isPayDay(now))
      paychecks.add(e.calculatePay());
  }

  return paychecks;
}
```

Notice that there is no mention of any of the hideous details that dominate the requirements. The underlying truth of this application is that we need to pay all employees on their payday. Separating the high-level policy from the low-level detail is the most fundamental part of making a design simple and expressive.

Tests: The Other Half of the Problem

When Kent Beck originally wrote this rule, he phrased it this way:

The system (code and tests) must communicate everything you want to communicate.

There is a reason he phrased it that way: No matter how expressive you make the production code, it cannot communicate the context in which it is used. That's the job of the tests.

Every test you write, especially if those tests are isolated and decoupled, is a demonstration of how the production code is intended to be used. Well-written tests are example use cases for the parts of the code that they test.

Thus, *taken together*, the code and the tests are an expression of what each element of the system does and how each element of the system should be used.

What does this have to do with design? Everything, of course. Because the primary goal we wish to achieve with our designs is to make it easy for other programmers to understand, improve, and upgrade our systems. And

there is no better way to achieve that goal than to demonstrate the intended usage of the system's abstractions through a comprehensive suite of executable tests.

Minimize Duplication

As I have told you in several previous chapters, in the earliest days of software, there was no way to call one program from another. If you wanted to do something more than once, you had to duplicate the code. Programmers wrote code snippets into their notebooks so that they could copy them into their programs when needed. They called those snippets *subroutines*.

In the 1950s, it became possible for programs to call subroutines and have those subroutines return to the point of call. In those days, we had no source code editors at all. We wrote our code, using #2 pencils, on preprinted coding forms. We therefore avoided most duplication because it was easier for us to create a single instance of a code snippet and call it. The calls took many CPU cycles, but the lack of duplication saved a lot of memory. It was often a good trade-off.

Then came the source code editors. And with those editors came copy/paste operations. Suddenly it became easy to copy a snippet of code from one routine and paste it into a new routine and then fiddle with it until it worked. With memory becoming cheaper and bigger, the cost of duplication became less of a concern. Thus, as the years went by, more and more systems exhibited massive amounts of duplication in the code.

Duplication is usually problematic. Two or more similar stretches of code will often need to be modified together. Finding those similar stretches is hard. Properly modifying them is even harder because they exist in different contexts. Thus, duplication leads to fragility.

In general, it is best to reduce similar stretches of code into a single instance by abstracting the code into a new function and providing it with appropriate arguments that communicate any differences in context.

Sometimes that strategy is not workable. For example, sometimes the duplication is in code that traverses a complex data structure. Many

different parts of the system may wish to traverse that structure, and they will use the same looping and traversal code only to then operate on the data structure in the body of that code.

As the structure of the data changes over time, programmers will have to find all the duplications of the traversal code and properly update them. The more the traversal code is duplicated, the higher the risk of fragility.

The duplications of the traversal code can be eliminated by encapsulating it in once place, and using lambdas, Command objects, the Strategy pattern, or even the Template Method pattern[3] to pass the necessary operations into the traversal.

---

3. [GOF95].

Accidental Duplication

Not all duplication should be eliminated. There are situations in which two stretches of code may be very similar, even identical, but will change for very different reasons.[4] I call this *accidental duplication*. Accidental duplicates should not be eliminated. The duplication should be allowed to persist. As the requirements change, the duplicates will evolve separately and the accidental duplication will dissolve.

---

4. See "SRP: The Single Responsibility Principle" in Chapter 19, "The SOLID Principles."

It should be clear that managing duplication is nontrivial. Identifying which duplications are real and which are accidental and then encapsulating and isolating the real duplications requires a significant amount of thought and care.

Determining the real duplications from the accidental duplications depends strongly on how well the code expresses its intent. Accidental duplications have divergent intent. Real duplications have convergent intent.

Encapsulating and isolating the real duplications, using abstraction, lambdas, and design patterns, involves a substantial amount of refactoring.

And refactoring requires a good, solid suite of tests.

Therefore, eliminating duplication is third in the priority list of the rules of simple design. First come the tests, then the expression, and *then* the duplication.

Minimize Size

A simple design is composed of as few simple elements as possible without compromising test coverage and expression. Simple elements are small. The last rule of simple design suggests that after you've gotten all the tests to pass, and after you have made the code as expressive as possible, and after you have minimized duplication, *then* you should work to decrease the number of modules, classes, functions, and lines.

Simple Design

Many years back, Kent Beck and I were having a discussion on the principles of design. He said something that has always stuck with me. He said that if you followed these four things as diligently as possible, all other design principles would be satisfied—the principles of design can be reduced to coverage, expression, singularization, and reduction.

I don't know if this is true or not. I don't know if a perfectly covered, expressed, singularized, and reduced program necessarily conforms to the Open–Closed Principle (OCP) or the Single Responsibility Principle (SRP). What I am very sure of, however, is that knowing and studying the principles of good design and good architecture (e.g., the SOLID principles) makes it much easier to create well-partitioned and simple designs.