Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Avoiding Repetition

Martin Fowler

Software design is not easy—not easy to do, teach, or evaluate. Much of software education these days is about products and APIs, yet much of these are transient, whereas good design is eternal—if only we could figure out what *good* design is.

Searching for design principles

One of the best ways to capture and promulgate good design is to learn from the patterns community. Their work, especially the famous book *Design Patterns* (E. Gamma et al., Addison Wesley, Reading, Mass., 1994), has become a cornerstone for many designers of object-oriented software. Patterns are not easy to understand, but they reward the effort of study. We can learn from the specific solutions they convey and from the thinking process that leads to their development. The thinking process is hard to grasp, but understanding it helps us discover principles that often generate these patterns.

Over the last year, I've been struck by one of the underlying principles that leads to better designs: remove duplication. It's also been highlighted by mantras in a couple of recent books: the DRY (don't repeat yourself) principle in the *Pragmatic Programmer* (A. Hunt, and D. Thomas, Addison Wesley, 1999) and "Once and Only Once" from *Extreme Programming Explained: Embrace Change* (K. Beck, Addison Wesley, 1999).

The principle is simple: say anything in your program only once. Stated blandly like that, it hardly bears saying. Yet identifying and removing repetition can lead to many interesting consequences. I have an increasing sense that a pig-headed determination to remove all repetition can lead you a long way toward a good design and can help you apply and understand the patterns that are common in good designs.

A simple case: subroutine calls

Take a simple example: subroutine calls. You use a subroutine when you realize that two blocks of code, in different places, are or will be the same. You define a subroutine and call it from both places. So, if you change what you need to, you don't have to hunt down multiple repetitions to make the change. Granted, sometimes the duplication is just a coincidence, so you wouldn't want a change in one to affect the other—but I find that is rare and easy to spot.

So what if the blocks are similar but not identical? Maybe some data is different in the two cases. In that case, the answer is obvious: you parameterize the data by passing in arguments to a subroutine. One block of code that multiplies by five and another that multiplies by 10 become one block that multiplies by x, and you replace x with the right number.

That's a simple resolution but it illustrates a basic principle that carries over into more complicated cases. Identify what is common and what varies, find a way to isolate the common stuff from the variations, then remove the redundancy in the common stuff. In this case, separating the commonality and the variability is easy. Many times it seems impossible, but the effort of trying leads to good design.

What's the same and what's different?

What if two routines have the same basic flow of behavior but differ in the actual steps (see Figure 1)? These two routines are similar, but not the same. So, what is the same and what is different?

The sameness is the routine's overall structure, and the differences are in the steps. In both cases, the structure is

- print some header for the invoice,
- loop through each item printing a line, and
- print a footer for the invoice.

As Figure 2 shows, we can separate the two by coming up with some kind of printer notion with a common interface for header, footer, and lines and an implementation for the ASCII case. Figure 3a shows that the common part is then the looping structure, so we can wire the pieces together as shown in Figure 3b.

There's nothing earth-shattering about this solution; just apply a polymorphic interface—which is common to any OO or component-based environment that lets you easily plug in multiple implementations of a common interface. *Design Patterns* junkies will recognize the Template Method pattern. If you are a good designer and are familiar with polymorphic interfaces, you could probably come up with this yourself—as many did. Knowing the pattern just gets you there quicker. The point is, the desire to eliminate duplication can lead you to this solution.

Duplication and patterns

Thinking in terms of duplication and its problems also helps you understand the benefits of patterns. Framework folks like patterns because they easily let you define new pluggable behaviors to fit behind the interface. Eliminating duplication helps because as you write a new implementation, you don't have to worry about the common things that need to be. Any common behavior should be in the template method. This lets you concentrate on the new

```
class Invoice...

  String asciiStatement() {
    StringBuffer result = new StringBuffer();
    result.append("Bill for " + customer + "\n");
    Iterator it = items.iterator();
    while(it.hasNext()) {
        LineItem each = (LineItem) it.next();
        result.append("\t" + each.product() + "\t\t"
          + each.amount() + "\n");
    }
    result.append("total owed:" + total + "\n");
    return result.toString();
  }
```

```
  String htmlStatement() {
    StringBuffer result = new StringBuffer();
    result.append("<P>Bill for <I>" + customer + "</I></P>");
    result.append("<table>");
    Iterator it = items.iterator();
    while(it.hasNext()) {
        LineItem each = (LineItem) it.next();
        result.append("<tr><td>" + each.product()
          + "</td><td>" + each.amount() + "</td></tr>");
    }
    result.append("</table>");
    result.append("<P> total owed:<B>" + total + "</B></P>");
    return result.toString();
  }
```

Figure 1. Two similar routines with different steps.

```
  interface Printer {
    String header(Invoice iv);
    String item(LineItem line);
    String footer(Invoice iv);
  }
```

(a)

```
  static class AsciiPrinter implements Printer {
    public String header(Invoice iv) {
      return "Bill for " + iv.customer + "\n";
    }
    public String item(LineItem line) {
      return "\t" + line.product() + "\t\t" + line.amount() + "\n";
    }
    public String footer(Invoice iv) {
      return "total owed:" + iv.total + "\n";
    }
   }
```

(b)

Figure 2. (a) A common interface for header, footer, and lines and (b) an implementation for the ASCII case (try the HTML case as an exercise on your own).

```
class Invoice...
  public String statement(Printer pr) {
    StringBuffer result = new StringBuffer();
    result.append(pr.header(this));
    Iterator it = items.iterator();
    while(it.hasNext()) {
         LineItem each = (LineItem) it.next();
         result.append(pr.item(each));
    }
    result.append(pr.footer(this));
    return result.toString();
  }
```

(a)

```
class Invoice...
  public String asciiStatement2() {
    return statement (new AsciiPrinter());
  }
```

(b)

Figure 3. (a) The common part of the routine and (b) the pieces wired together.

behavior rather than the old.

The principle of duplication also helps you think about when to apply this pattern. As many people know, one of the problems with people who have just read a pattern is that they insist on using it, which often leads to more complicated designs. When you insist on using a pattern, ask, "What repetition is this removing?" Removing repetition makes it more likely that you're making good use of the pattern. If not, perhaps you shouldn't use it.

Often, the hard part of eliminating duplication is spotting it in the first place. In my example, you can spot the two routines easily because they are in the same file and located close to each other. What happens when they are in separate files and written by different people in different millennia?

This question leads us to think about how we construct our software to reduce the chances of this happening. Using abstract data types is a good way of doing this. Because you have to pass data around, you find that people are less likely to duplicate data structures. If you try to place routines next to their data structures, you are more likely to spot any duplication in the routines. Much of the reason why objects are popular is because of this kind of social effect, which works well when reinforced by a culture that encourages people to look around and fix duplications when they do arise.

So, avoiding repetition is a simple principle that leads to good design. I intend to use this column to explore other simple principles that have this effect. Have you noticed simple principles like this in your work? If so, please contact me—I'm always happy to repeat good ideas.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. For a decade, he was an independent consultant pioneering the use of objects in developing business information systems. He's worked with technologies including Smalltalk, C++, object and relational databases, and Enterprise Java with domains including leasing, payroll, derivatives trading, and healthcare. He is particularly known for his work in patterns, UML, lightweight methodologies, and refactoring. He has written four books: *Analysis Patterns, Refactoring, Planning Extreme Programming,* and *UML Distilled.* Contact him at fowler@acm.org.

IEEE Software

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org), or access computer.org/software/author.htm.

Letters to the Editor
Send letters to

Letters Editor
IEEE Software
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
dstrok@computer.org

Please provide an e-mail address or daytime phone number with your letter.

On the Web
Access computer.org/software for information about *IEEE Software.*

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software.*

Membership Change of Address
Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles
For price information or to order reprints, send e-mail to software@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.