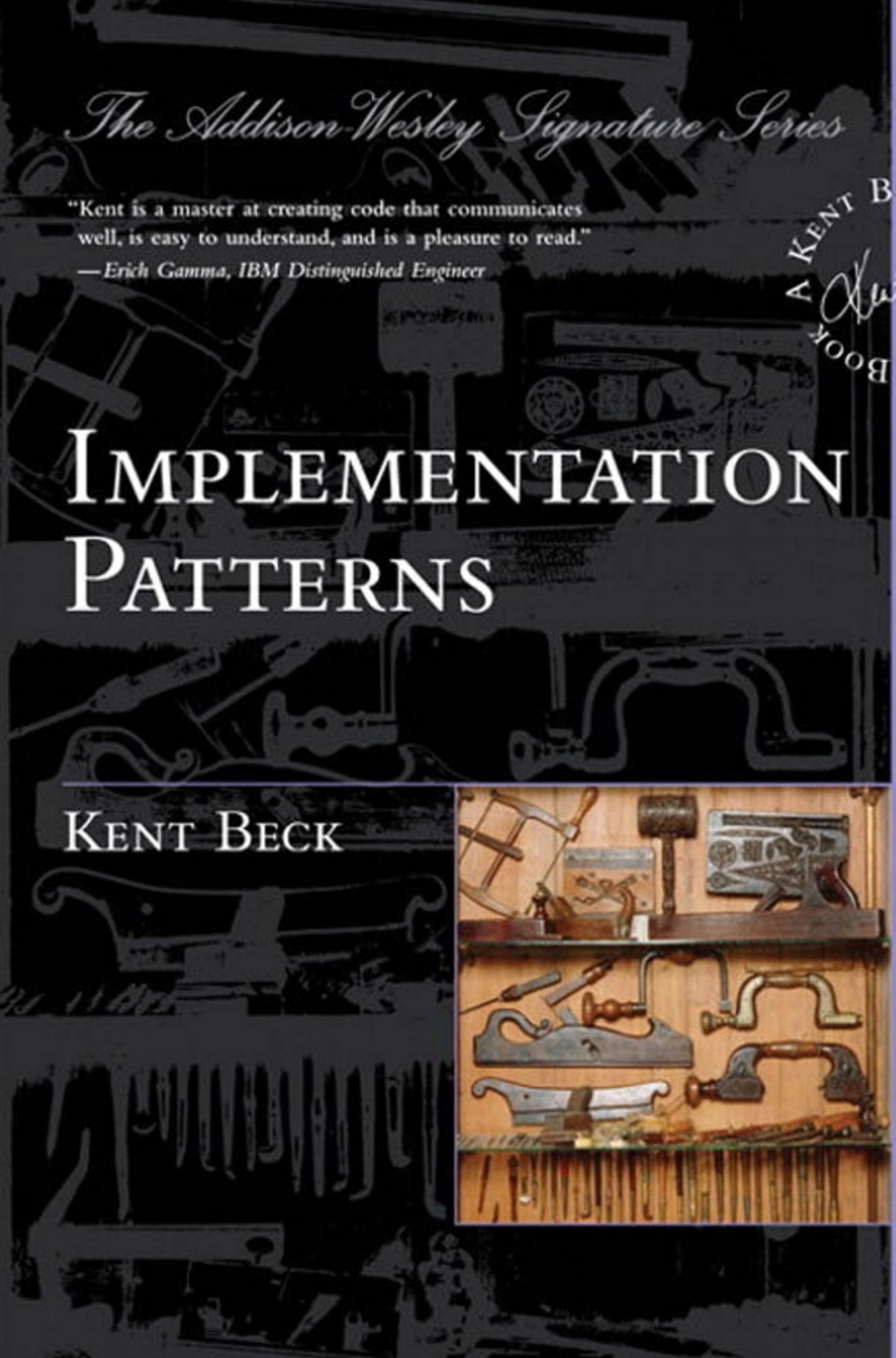

The Addison-Wesley Signature Series

"Kent is a master at creating code that communicates well, is easy to understand, and is a pleasure to read."
—*Erich Gamma, IBM Distinguished Engineer*

A KENT BECK SIGNATURE BOOK

# IMPLEMENTATION PATTERNS

KENT BECK

# Praise for *Implementation Patterns*

"Kent is a master at creating code that communicates well, is easy to understand, and is a pleasure to read. Every chapter of this book contains excellent explanations and insights into the smaller but important decisions we continuously have to make when creating quality code and classes."

—*Erich Gamma, IBM Distinguished Engineer*

"Many teams have a master developer who makes a rapid stream of good decisions all day long. Their code is easy to understand, quick to modify, and feels safe and comfortable to work with. If you ask how they thought to write something the way they did, they always have a good reason. This book will help you become the master developer on your team. The breadth and depth of topics will engage veteran programmers, who will pick up new tricks and improve on old habits, while the clarity makes it accessible to even novice developers."

—*Russ Rufer, Silicon Valley Patterns Group*

"Many people don't realize how readable code can be and how valuable that readability is. Kent has taught me so much, I'm glad this book gives everyone the chance to learn from him."

—*Martin Fowler, chief scientist, ThoughtWorks*

"Code should be worth reading, not just by the compiler, but by humans. Kent Beck distilled his experience into a cohesive collection of implementation patterns. These nuggets of advice will make your code truly worth reading."

—*Gregor Hohpe, author of* Enterprise Integration Patterns

"In this book Kent Beck shows how writing clear and readable code follows from the application of simple principles. *Implementation Patterns* will help developers write intention revealing code that is both easy to understand and flexible towards future extensions. A must read for developers who are serious about their code."

—*Sven Gorts*

"*Implementation Patterns* bridges the gap between design and coding. Beck introduces a new way of thinking about programming by basing his discussion on values and principles."

—*Diomidis Spinellis, author of* Code Reading *and* Code Quality

## Instance-Specific Behavior

In theory, all instances of a class share the same logic. Relaxing this constraint enables new styles of expression. All of these styles, though, come at a cost. When the logic of an object is completely determined by its class, readers can read the code in the class to see what is going to happen. Once you have instances with different behavior, you have to look at live examples or analyze the data flow to understand how a particular object is going to behave.

Another step up in the cost of instance-specific behavior is when the logic changes as the computation progresses. For ease of code reading, try to set instance-specific behavior when an object is created and don't change it afterward.

## Conditional

If/then and switch statements are the simplest form of instance-specific behavior. Using conditionals, different objects will execute different logic based on their data. Conditionals as a form of expression have the advantage that the logic is still all in one class. Readers don't have to go navigating around to find the possible paths for a computation. However, conditionals have the disadvantage that they can't be modified except by modifying the code of the object in question.

Each path of execution through a program has some probability of being correct. Assuming that the probabilities of correctness for the paths are independent, the more paths through a program the less likely the program is to be correct. The probabilities aren't entirely independent, but they are independent enough that programs with many paths are more likely to have defects than those with few paths. The proliferation of conditionals reduces reliability.

This problem is compounded when conditionals are duplicated. Consider a simple graphic editor. The figures will need a `display()` method:

```
public void display() {
  switch (getType()) {
    case RECTANGLE :
      //...
      break;
    case OVAL :
      //...
      break;
    case TEXT :
      //...
      break;
    default :
```

```
      break;
  }
}
```

Figures will also need a method to determine whether a point is contained within them:

```
public boolean contains(Point p) {
  switch (getType()) {
    case RECTANGLE :
      //...
      break;
    case OVAL :
      //...
      break;
    case TEXT :
      //...
      break;
    default :
      break;
  }
}
```

Suppose now you want to add a new kind of figure. First, you must add a clause to every switch statement. Second, to make this change you have to modify the `Figure` class, putting all of the existing functionality at risk. Lastly, everyone who wants to add new figures must coordinate their changes of a single class.

These problems can all be eliminated by converting the conditional logic to messages, either with subclasses or delegation (which technique serves best depends on the code). Duplicate conditional logic or logic where the processing is very different based on which branch of a conditional is taken is generally better expressed as messages instead of explicit logic. Also, conditional logic that changes frequently is better expressed as messages to simplify changing one branch while minimizing the effects on other branches.

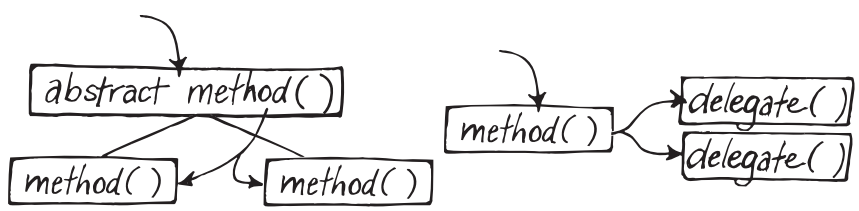


**Figure 5.6** *Conditional logic represented by subclasses and delegation*

In short, the strengths of conditionals—that they are simple and local—become liabilities when they are used too widely.

## Delegation

Another way to execute different logic in different instances is to delegate work to one of several possible kinds of objects. The common logic is held in the referring class, the variations in the delegates.

An example of using a delegate to capture variation is handling user input in a graphical editor. Sometimes a button press means "create a rectangle", sometimes it means "move a figure", and so on.

One way to express the variation between the tools is with conditional logic:

```
public void mouseDown() {
  switch (getTool()) {
    case SELECTING :
      //...
      break;
    case CREATING_RECTANGLE :
      //...
      break;
    case EDITING_TEXT :
      //...
      break;
    default :
      break;
  }
}
```

This has all the problems of conditionals discussed above: adding a new tool requires modifying the code and the duplication of the conditional (in `mouseUp()`, `mouseMove()`, etc.) makes adding new tools complicated.

Subclassing is not an immediate answer either because the editor needs to change tools during its lifetime. Delegation allows that flexibility.

```
public void mouseDown() {
  getTool().mouseDown();
}
```

The code that used to live in the clauses of the switch statement is moved to the various tools. Now new tools can be introduced without modifying the code of the editor or the existing tools. Reading the code requires more navigation, however, because the mouse-down logic is spread over several classes. Understanding how the editor will behave in a given situation requires that you understand what kind of tool it is currently using.

Delegates can be stored in fields (a "pluggable object"), but they can also be computed on the fly. JUnit 4 dynamically computes the object that will run the tests in a given class. If a class contains old-style tests, one delegate is created, but if the class contains new-style tests, a different delegate is created. This is a mix of conditional logic (to create the delegates) and delegation.

Delegation can be used for code sharing as well as instance-specific behavior. An object that delegates to a `Stream` may be involved in instance-specific behavior, if the type of `Stream` can change at runtime, or it may be sharing the implementation of `Stream` with all the other users.

A common twist on delegation is to pass the delegator as a parameter to a delegated method.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown(this);
}

RectangleTool
public void mouseDown(GraphicEditor editor) {
  editor.add(new RectangleFigure());
}
```

If a delegate needs to send a message to itself, "itself" is ambiguous. Sometimes the message should be sent to the delegating object. Sometimes the message should be sent to the delegate. In the example below the `RectangleTool` adds a figure, but to the delegating `GraphicsEditor`, not to itself. The `GraphicsEditor` could have been passed as a parameter to the delegated `mouseDown()` method, but in this case it seemed simpler to store a permanent back-reference in the tool. Passing the `GraphicsEditor` as a parameter makes it possible to use the same tool in multiple editors, but if this isn't important the code with the backpointer may be simpler.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown();
}

RectangleTool
private GraphicEditor editor;
public RectangleTool(GraphicEditor editor) {
  this.editor= editor;
}
public void mouseDown() {
  editor.add(new RectangleFigure());
}
```

## Pluggable Selector

Let's say you need instance-specific behavior, but only for one or two methods, and you don't mind having all the variants of the code live in one class. In this case, store the name of the method to be invoked in a field and invoke the method by reflection.

Originally, each test in JUnit had to be stored in its own class (Figure 5.7). Each subclass only had one method. Classes seemed conceptually heavy as a way to represent a single class.

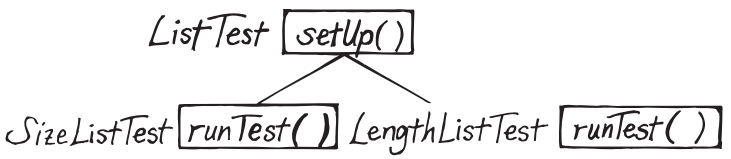


**Figure 5.7** *Trivial subclasses to represent different tests*

By implementing a generic `runTest()`, `ListTests` with different names run different test methods. The name of the test is assumed to also be the name of a method which is retrieved and run when the test is run. Here is the simple version of the code to implement the pluggable selector version of running a test.

```
String name;
public void runTest() throws Exception {
  Class[] noArguments= new Class[0];
  Method method= getClass().getMethod(name, noArguments);
  method.invoke(this, new Object[0]);
}
```

The simplified class hierarchy uses a single class (Figure 5.8). As with all code compression techniques, the modified code is only easy to read if you understand the "trick".

When pluggable selectors first became widely known, people tended to overuse them. You would be looking at some code, decide it couldn't possibly be called, delete, and have the system break because it was invoked by a pluggable selector somewhere. The costs of using pluggable selectors are considerable, but a limited use to solve a difficult problem may justify the cost.

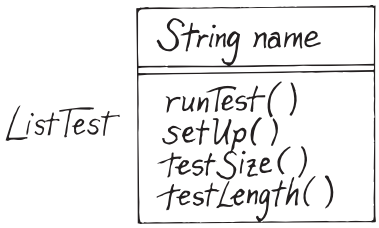


**Figure 5.8** *Pluggable selector helps pack tests into a single class*

## Anonymous Inner Class

Java offers one more alternative for instance-specific behavior, anonymous inner classes. The idea is to create a class that is only used in one place, that can override one or more methods for strictly local purposes. Because it is only used in one place, the class can be referred to implicitly instead of by name.

Effective use of anonymous inner classes relies on having an extremely simple API—like implementing `Runnable` with its one method `run()`—or having a superclass that provides most of the needed implementation so the anonymous inner class can be implemented simply. The code for the anonymous inner class interrupts the presentation of the code in which it is embedded, so it needs to be short so it doesn't distract the reader.

Anonymous inner classes have the limitations that the code to be set in the instance must be known when you write the class (unlike delegates, which can be added later) and it cannot be changed once an instance has been created. Anonymous inner classes are difficult to test directly and so shouldn't contain complicated logic. Because they are un-named, you don't have the opportunity to express your intention for an anonymous inner class with a well-chosen name.

## Library Class

Where do you put functionality that doesn't fit into any object? One solution is to create static methods on an otherwise-empty class. No one is expected to ever create instances of this class. It is just there as a holder for the functions in the library.

While library classes are fairly common, they don't scale well. Putting all the logic into static methods forfeits the biggest advantage of programming with objects: a private namespace of shared data that can be used to help simplify logic. Try to turn library classes into objects whenever possible.