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

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Beck, Kent.

Implementation patterns / Kent Beck.

p. cm.

Includes bibliographical references and index.

ISBN-13: 978-0-321-41309-3 (pbk. : alk. paper)

1. Software patterns. 2. Computer software--Development. I. Title.

QA76.76.P37B33 2007

005.1--dc22

2007035166

ISBN-13: 978-0-321-41309-3

ISBN-10: 0-321-41309-1

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, October 2007

Contents

Preface

This is a book about programming—specifically, about programming so other people can understand your code. There is no magic to writing code other people can read. It's like all writing—know your audience, have a clear overall structure in mind, express the details so they contribute to the whole story. Java offers some good ways to communicate. The implementation patterns here are Java programming habits that result in readable code.

Another way to look at implementation patterns is as a way of thinking "What do I want to tell a reader about this code?" Programmers spend so much of their time in their own heads that trying to look at the world from someone else's viewpoint is a big shift. Not just "What will the computer do with this code?" but "How can I communicate what I am thinking to people?" This shift in perspective is healthy and potentially profitable, since so much software development money is spent on understanding existing code.

There is an American game show called Jeopardy in which the host supplies answers and the contestants try to guess the questions. "A word describing being thrown through a window." "What is 'defenestration'?" "Correct."

Coding is like Jeopardy. Java provides answers in the form of its basic constructs. Programmers usually have to figure out for themselves what the questions are, what problems are solved by each language construct. If the answer is "Declare a field as a `Set`." the question might be "How can I tell other programmers that a collection contains no duplicates?" The implementation patterns provide a catalog of the common problems in programming and the features of Java that address those problems.

Scope management is as important in book writing as it is in software development. Here are some things this book is not. It is not a style guide because it contains too much explanation and leaves the final decisions up to the reader. It is not a design book because it is mostly concerned with smaller-scale decisions, the kind programmers make many times a day. It's not a patterns book because the format of the patterns is idiosyncratic and *ad hoc* (literally “built for a particular purpose”). It’s not a language book because, while it covers many Java language features, it assumes readers already know Java.

Actually this book is built on a rather fragile premise: that good code matters. I have seen too much ugly code make too much money to believe that quality of code is either necessary or sufficient for commercial success or widespread use. However, I still believe that quality of code matters even if it doesn’t provide control over the future. Businesses that are able to develop and release with confidence, shift direction in response to opportunities and competition, and maintain positive morale through challenges and setbacks will tend to be more successful than businesses with shoddy, buggy code.

Even if there was no long-term economic impact from careful coding I would still choose to write the best code I could. A seventy-year lifespan contains just over two billion seconds. That’s not enough seconds to waste on work I’m not proud of. Coding well is satisfying, both the act itself and the knowledge that others will be able to understand, appreciate, use, and extend my work.

In the end, then, this is a book about responsibility. As a programmer you have been given time, talent, money, and opportunity. What will you do to make responsible use of these gifts? The pages that follow contain my answer to this question for me: code for others as well as myself and my buddy the CPU.

Acknowledgments

First, last, and always I would like to thank Cynthia Andres, my partner, editor, support, and chief butt-kicker. My friend Paul Petralia got this project going with me and provided encouraging phone calls throughout. My editor Chris Guzikowski and I learned how to work together over the course of this project. He gave me the support I needed from the Pearson side to finish the book. Thank you to the production team at Pearson: Julie Nahil, John Fuller, and Cynthia Kogut. Jennifer Kohnke produced illustrations that combine information and humanity. My reviewers provided clear, timely feedback for my drafts: Erich Gamma, Steve Metsker, Diomidis Spinellis, Tom deMarco, Michael Feathers, Doug Lea, Brad Abrams, Cliff Click, Pekka Abrahamson, Gregor Hohpe and Michele Marchesi. Thank you to David Saff for recognizing the symmetry between state and behavior. My children who remain at home kept reminding me of why I wanted to be finished: Lincoln, Lindsey, Forrest, and Joëlle Andres-Beck.

Chapter 1

Introduction

Here we are together. You've picked up my book (it's yours now). You already write code. You have probably already developed a style of your own through your own experiences.

The goal of this book is to help you communicate your intentions through your code. The book begins with an overview of programming and patterns (chapters 2-4). The remainder of the book (chapters 5-8) is a series of short essays, patterns, on how to use the features of Java to write readable code. It closes with a chapter on how to modify the advice here if you are writing frameworks instead of applications. Throughout, the book is focused on programming techniques that enhance communication.

There are several steps to communicating through code. First I had to become conscious while programming. I had been programming for years when I first started writing implementation patterns. I was astonished to discover that, even though programming decisions came smoothly and quickly to me, I couldn't explain why I was so sure a method should be called such-and-so or that a bit of logic belonged in this object over here. The first step towards communicating was slowing down long enough to become aware of what I was thinking, to stop pretending that I coded by instinct.

The second step was acknowledging the importance of other people. I found programming satisfying, but I am self-centered. Before I could write communicative code I needed to believe that other people were as important as I was. Programming is hardly ever a solitary communion between one man and one machine. Caring about other people is a conscious decision, and one that requires practice.

Which brings me to the third step. Once I had exposed my thinking to sunlight and fresh air and acknowledged that other people had as much right to exist as I did, I needed to demonstrate my new perspective in practice. I use the implementation patterns here to program consciously and for others as well as myself.

You can read this book strictly for technical content—useful tricks with explanations. However, I thought it fair to warn you that there is a whole lot more going on, at least for me.

You can find those technical bits by thumbing through the patterns chapters. One effective strategy for learning this material is to read it just before you need to use it. To read it "just-in-time", I suggest skipping right to chapter 5 and skimming through to the end, then keeping the book by you as you program. After you've used many of the patterns, you can come back to the introductory material for the philosophical background behind the ideas you've been using.

If you are interested in a thorough understanding of the material here, you can read straight through from the beginning. Unlike most of my books, however, the chapters here are quite long, so it will take concentration on your part to read end-to-end.

Most of the material in this book is organized as patterns. Most decisions in programming are similar to decisions that have come before. You might name a million variables in your programming career. You don't come up with a completely novel approach to naming each variable. The general constraints on naming are always the same: you need to convey the purpose, type, and lifetime of the variable to readers, you need to pick a name that's easy to read, you need to pick a name that's easy to write and format. Add to these general constraints the specifics of a particular variable and you come up with a workable name. Naming variables is an example of a pattern: the decision and its constraints repeat even though you might create a different name each time.

I think patterns often need different presentations. Sometimes an argumentative essay best explains a pattern, sometimes a diagram, sometimes a teaching story, sometimes an example. Rather than cram each pattern's description into a rigid format, I have described each in the way I thought best.

This book contains 77 explicitly named patterns, each covering some aspect of writing readable code. In addition, there are many smaller patterns or variants of patterns that I mention in passing. My goal with this book is to offer advice for how to approach most common, daily coding tasks so as to help future readers understand what the code is supposed to do.

This book fits somewhere between *Design Patterns* and a Java language manual. *Design Patterns* talks about decisions you might make a few times a day while developing, typically decisions that regulate the interaction between objects. You apply an implementation pattern every few seconds while programming. While language manuals are good at describing what you can do with Java, they don't talk much about why you would use a certain construct or what someone reading your code is likely to conclude from it.

Part of my philosophy in writing this book has been to stick to topics I know well. Concurrency issues, for example, are not addressed in these implementation patterns, not because concurrency isn't an important issue, but rather because it is not one on which I have a lot to say. My concurrency strategy has always been to isolate as much as possible concurrent parts of my applications. While I am generally successful in doing so, it's not something I can explain. I recommend a book such as *Java Concurrency in Practice* for a practical look at concurrency.

Another topic not addressed in this book is any notion of software process. The advice about communicating through code here is intended to work whether that code is written near the end of a long cycle or seconds after a failing test has been written. Software that costs less overall is good to have, whatever the sociological trappings within which it is written.

I also stop short of the edges of Java. I tend to be conservative in my technology choices because I have been burned too often pushing new features to their limits (it's a fine learning strategy but too risky for most development). So, you'll find here a pedestrian subset of Java. If you are motivated to use the latest features of Java, you can learn them from other sources.

Tour Guide

The book is divided into seven major sections as seen in Figure 1.1. Here they are:

- Introduction—these short chapters describe the importance and value of communicating through code and the philosophy behind patterns.
- Class—patterns describing how and why you might create classes and how classes encode logic.
- State—patterns for storing and retrieving state.
- Behavior—patterns for representing logic, especially alternative paths.
- Method—patterns for writing methods, reminding you what readers are likely to conclude from your choice of method decomposition and names.
- Collections—patterns for choosing and using collections.
- Evolving Frameworks—variations on the preceding patterns when building frameworks instead of applications.

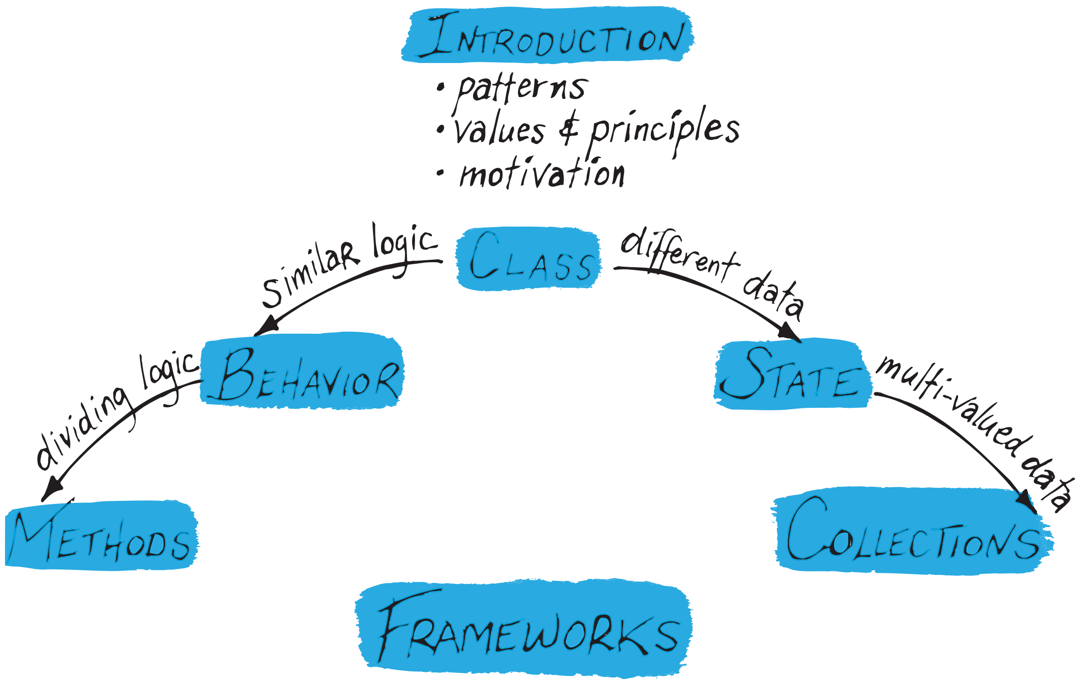

Figure 1.1 *Book overview*

And Now...

...to the meat of the book. If you are reading straight through, just turn the page (I suppose you would have figured that one out for yourself). If you want to browse the patterns themselves, start with chapter 5, page 21. Happy implementing.

Chapter 2

Patterns

Many decisions in programming are unique. How you approach programming a web site will be quite different from how you approach building a pacemaker. However, as the decisions become more and more purely technical, a sense of familiarity sets in. Didn't I just write this code? Programming would be more effective if programmers spent less time on the mundane, repetitive parts of their job so they had more time to spend doing a good job of solving truly unique problems.

Most programs follow a small set of laws:

- Programs are read more often than they are written.
- There is no such thing as "done". Much more investment will be spent modifying programs than developing them initially.
- They are structured using a basic set of state and control flow concepts.
- Readers need to understand programs in detail and in concept. Sometimes they move from detail to concept, sometimes from concept to detail.

Patterns are based on this commonality. For example, every programmer has to decide how to structure iteration. By the time you are thinking about how to write a loop, most of the domain-specific questions have been resolved for the moment and you are left with purely technical issues: the loop should be easy to read, easy to write, easy to verify, easy to modify, and efficient.

This list of concerns is the beginning of a pattern. The constraints or *forces* listed above affect how every loop in a program is written. The forces recur predictably, which is one sense in which a pattern is a pattern: it is a pattern of forces.

There are a few reasonable ways to write a loop. Each of them implies different priorities between the constraints. If performance is more important you might structure the loop one way, while if ease of modification is more important you might structure it differently.

Each pattern illustrates a point of view about the relative priorities of the forces. Most patterns come with a little essay about the alternatives for solving the problem and why the recommended solution is superior. Revealing the reasoning behind the advice in a pattern invites readers to decide for themselves how they want to approach a recurring problem.

As hinted above, each pattern also comes with the seed of a solution. The pattern for loops over a collection might suggest, "Use the Java5 `for` loop to express iteration." Patterns bridge from abstract principles to practice. Patterns help you write code.

Patterns work together. The pattern suggesting a `for` loop introduces the problem of what to name the loop variable. Rather than pack everything into a single pattern, there is another pattern specifically covering variable naming.

The style of presentation for the patterns varies considerably in this book. Sometimes patterns are clearly named, with sections to discuss the forces and solution. Some of the smaller patterns, though, are embedded within a larger pattern. A sentence or two may be all the discussion a little pattern needs.

Working with patterns can feel constraining at times but using a pattern can save time and energy. For example, making a bed takes much less energy to do by habit than if you had to think out each step in the process and work out the correct order each time. You have a set pattern for bed-making which greatly simplifies that chore. If the bed is against the wall or the sheet is too small you adapt your strategy to the situation, but overall bed-making can be done by rote, freeing your mind for more interesting and unique tasks. As a pattern becomes habit I find I appreciate not having to raise a debate just to write a loop. If the whole team becomes dissatisfied with a pattern, they can discuss their options for introducing a new pattern.

No one set of patterns will work in all programming situations. Listed later in this book are the patterns that I use and that I have observed to work well in application development (with a brief foray into framework development). Blindly copying anyone's style is not as effective as thinking about and practicing your own style and discussing and sharing a style within your team.

Patterns work best as aids to human decision making. Some of the implementation patterns will eventually make their way into programming languages just as structured uses of `setjmp()/longjmp()` became today's exception handling system. In the meantime, patterns often require adaptation before they are used.

This chapter opened with the search for a cheaper, quicker, less energy-consuming way to solve the common problems in programming. Using patterns helps programmers write reasonable solutions to common problems, leaving more time, energy, and creativity to apply to the truly unique problems. Each pattern bundles a common problem of programming with a discussion of the factors affecting that problem and provides concrete advice about how to quickly create a satisfactory solution. The result is a better/cheaper/faster job on the boring parts of programs and more time and energy to invest in the unique problems of each program.

The following chapter, “A Theory of Programming”, describes the values and principles underlying the style of programming described by these implementation patterns.

This page intentionally left blank

Chapter 3

A Theory of Programming

No list of patterns, no matter how exhaustive, can cover every situation that comes up while programming. Eventually (or even frequently) you'll come upon a situation where none of the cookie cutters fits. This need for general approaches to unique problems is one reason to study the theory of programming. Another is the sense of mastery that comes of knowing both what to do and why. Conversations about programming are also more interesting when they cover both theory and practice.

Each pattern carries with it a little bit of theory. There are larger and more pervasive forces at work in programming than are covered in individual patterns, however. This section describes these cross-cutting concerns. They are divided here into two types: values and principles. The values are the universal overarching themes of programming. When I am working well, I hold dear the importance of communicating with other people, removing excess complexity from my code, and keeping my options open. These values—communication, simplicity, and flexibility—color every decision I make while programming.

The principles described here aren't as far-reaching or pervasive as the values, but each one is expressed by many of the patterns. The principles bridge between the values, which are universal but often difficult to apply directly, and the patterns, which are clear to apply but specific. I have found it valuable to make the principles explicit for those situations where no pattern applies, or when two mutually exclusive patterns apply equally. Faced with ambiguity, understanding the principles allows me to "make something up" that is consistent with the rest of my practice and likely to turn out well.

These three elements—values, principles, and patterns—form a balanced expression of a style of development. The patterns describe what to do. The values provide motivation. The principles help translate motive into action.

The values, principles, and patterns here are drawn from my own practice, reflection, and conversation with other programmers. We all draw from the experience of previous generations of programmers. The result is *a* style of development, not *the* style of development. Different values and different principles will lead to different styles. One of the advantages of laying out a programming style as values, principles, and practices is that it is easier to have productive conflict about programming this way. If you want to do something one way and I another, we can identify the level of our disagreement and avoid wasting time. If we disagree about principles, arguing about where curly braces belong won't solve the underlying discord.

Values

Three values that are consistent with excellence in programming are communication, simplicity, and flexibility. While these three sometimes conflict, more often they complement each other. The best programs offer many options for future extension, contain no extraneous elements, and are easy to read and understand.

Communication

Code communicates well when a reader can understand it, modify it, or use it. While programming it's tempting to think only of the computer. However, good things happen when I think of others while I program. I get cleaner code that is easier to read, it is more cost-effective, my thinking is clearer, I give myself a fresh perspective, my stress level drops, and I meet some of my social needs. Part of what drew me to programming in the first place was the opportunity to commune with something outside myself. However, I didn't want to deal with sticky, inexplicable, annoying human beings. Programming as if people didn't really exist paled after only a couple of decades. Building ever-more-elaborate sugar castles in my mind became colorless and stale.

One of the early experiences that led me to focus on communication was discovering Knuth's *Literate Programming*: a progam should read like a book. It should have plot, rhythm, and delightful little turns of phrase.

When Ward Cunningham and I first read about literate programs, we decided to try it. We sat down with one of the cleanest pieces of code in the Smalltalk image, the ScrollController, and tried to make it into a story. Hours later we had completely rewritten the code on our way to a reasonable paper. Every time a bit of logic was a little hard to explain, it was easier to rewrite the code than explain why the code was hard to understand. The demands of communication changed our perspective on coding.

There is a sound economic basis for focusing on communication while programming. The majority of the cost of software is incurred after the software has been first deployed. Thinking about my experience of modifying code, I see that I spend much more time reading the existing code than I do writing new code. If I want to make my code cheap, therefore, I should make it easy to read.

Focusing on communication improves thinking by being more realistic. Part of the improvement comes from engaging more of my brain. When I think, "How would someone else see this?" different neurons are firing than when I'm just focused on myself and my computer. I take a step back from my isolated perspective and see my problem and solution anew. Another part of the improvement comes from the reduced stress of knowing that I am taking care of business, doing the right thing. Finally, as a socially oriented species, explicitly accounting for social issues is more realistic than working at pretending they don't exist.

Simplicity

In *The Visual Display of Quantitative Information*, Edward Tufte has an exercise where he takes a graph and starts erasing all the marks that don't add information. The resulting graph is novel and much easier to understand than the original.

Eliminating excess complexity enables those reading, using, and modifying programs to understand them more quickly. Some of the complexity is essential, accurately reflecting the complexity of the problem to be solved. Some of the complexity, though, represents the claw marks our fingernails make as we struggle to get the program to run at all. It is this excess complexity that removes value from software, both by making the software less likely to run correctly and more difficult to change successfully in the future. Part of programming is to look back at what you've done and separate the wheat from the chaff.

Simplicity is in the eye of the beholder. What is simple to an expert programmer, familiar with the power tools of the craft, might be overwhelmingly complex to a beginner. Just as good prose is written with an audience in mind, so good programs are written with an audience in mind. Challenging your audience a little is fine, but too much complexity will lose them.

Computing advances in waves of complexity and simplification. Mainframe architectures became more and more baroque until mini-computers came along. The mini-computer didn't solve all the problems of a mainframe, but it turned out that for many applications those problems weren't all that important. Programming languages, too, go through waves where they get more complex and then simpler. C begets C++, which begets Java, which is now becoming itself more complicated.

Pursuing simplicity enables innovation. JUnit was much simpler than the testing tools it largely replaced. JUnit spawned a variety of look-alikes, add-ons, and new programming/testing techniques. The latest release, JUnit 4, has lost that "bare metal" feel, although I made or concurred with each of the complexifying decisions. Someday someone will come up with a much simpler way for programmers to write tests than JUnit. The new idea will enable a further wave of innovation.

Apply simplicity at all levels. Format code so no code can be deleted without losing information. Design with no extraneous elements. Challenge requirements to find those that are essential. Eliminating excess complexity illuminates the remaining code, giving you a chance to approach it afresh.

Communication and simplicity often work together. The less excess complexity, the easier a system is to understand. The more you focus on communication, the easier it is to see what complexity can be discarded. Sometimes, however, I find a simplification that would make a program harder to understand. I choose communication over simplicity in these cases. Such situations are rare but usually point to some larger-scale simplification I'm not yet seeing.

Flexibility

Of the three values listed here, flexibility is the justification used for the most ineffective coding and design practices. To retrieve a constant, I've seen programs look up an environment variable containing the name of a directory containing a file in which is found the constant value. Why all the complexity? Flexibility. Programs should be flexible, but only in ways they change. If the constant never changes, all that complexity is cost without benefit.

Since most of the cost of a program will be incurred after it is first deployed, programs should be easy to change. The flexibility I imagine will be needed tomorrow, though, is likely to be not what I need when I change the code. That's why the flexibility of simplicity and extensive tests is more effective than the flexibility offered by speculative design.

Choose patterns that encourage flexibility and bring immediate benefits. For patterns with immediate costs and only deferred benefits, often patience is the best strategy. Put them back in the bag until they are needed. Then you can apply them in precisely the way they are needed.

Flexibility can come at the cost of increased complexity. For instance, user-configurable options provide flexibility but add the complexity of a configuration file and the need to take the options into account when programming. Simplicity can encourage flexibility. In the above example, if you can find a way to eliminate the configurable options without losing value, you will have a program that is easier to change later.

Enhancing the communicability of software also adds to flexibility. The more people who can quickly read, understand, and modify the code, the more options your organization has for future change.

The patterns that follow encourage flexibility by helping programmers create simple, understandable applications that can be changed.

Principles

The implementation patterns aren't the way they are "just because". Each one expresses one or more of the values of communication, simplicity, and flexibility. Principles are another level of general ideas, more specific to programming than the values, that also form the foundation of the patterns.

Examining principles is valuable for several reasons. Clear principles can lead to new patterns, just as the periodic table of the elements led to the discovery of new elements. Principles can provide an explanation for the motivation behind a pattern, one connected to general rather than specific ideas. Choices about contradictory patterns are often best discussed in terms of principles rather than the specifics of the patterns involved. Finally, understanding principles provides a guide when encountering novel situations.

For example, when I encounter a new programming language I use my understanding of principles to develop an effective style of programming. I don't have to ape existing styles or, worse, cling to my style in some other programming language (you can write FORTRAN code in any language, but you shouldn't). Understanding principles gives me a chance to learn quickly and act with integrity in novel situations. What follows is the list of principles behind the implementation patterns.

Local Consequences

Structure the code so changes have local consequences. If a change *here* can cause a problem *there*, then the cost of the change rises dramatically. Code with mostly local consequences communicates effectively. It can be understood gradually without first having to assemble an understanding of the whole.

Because keeping the cost of making changes low is a primary motivation behind the implementation patterns, the principle of local consequences is part of the reasoning behind many of the patterns.

Minimize Repetition

A principle that contributes to keeping consequences local is to minimize repetition. When you have the same code in several places, if you change one copy of the code you have to decide whether or not to change all the other copies. Your change is no longer local. The more copies of the code, the more a change will cost.

Copied code is only one form of repetition. Parallel class hierarchies are also repetitive, and break the principle of local consequences. If making one conceptual change requires me to change two or more class hierarchies, then the changes have spreading consequences. Restructuring so the changes are again local would improve the code.

Duplication is not always obvious until after it has been created, and sometimes not for a while even then. Having seen it I can't always think of a good way to eliminate it. Duplication isn't evil, it just raises the cost of making changes.

One of the ways to remove duplication is to break programs up into many small pieces—small statements, small methods, small objects, small packages. Large pieces of logic tend to duplicate parts of other large pieces of logic. This commonality is what makes patterns possible—while there are differences between different pieces of code, there are also many similarities. Clearly communicating which parts of programs are identical, which parts are merely similar, and which parts are completely different makes programs easier to read and cheaper to modify.

Logic and Data Together

Another principle corollary to the principle of local consequences is keeping logic and data together. Put logic and the data it operates on near each other, in the same method if possible, or the same object, or at least the same package. To make a change, the logic and data are likely to have to change at the same time. If they are together, then the consequences of changing them will remain local.

It's not always obvious at first where logic or data should go to satisfy this principle. I may be writing code in A and realize I need data from B. It's only after I have the code working that I notice that it is too far from the data. Then

I need to choose what to do: move the code to the data, move the data to the code, put the code and data together in a helper object, or realize I can't at the moment think of how to bring them together in a way that communicates effectively.

Symmetry

Another principle I use all the time is symmetry. Symmetries abound in programs. An `add()` method is accompanied by a `remove()` method. A group of methods all take the same parameters. All the fields in an object have the same lifetime. Identifying and clearly expressing symmetry makes code easier to read. Once readers understand one half of the symmetry, they can quickly understand the other half.

Symmetry is often discussed in spatial terms: bilateral, rotational, and so on. Symmetry in programs is seldom graphical, it is conceptual. Symmetry in code is where the same idea is expressed the same way everywhere it appears in the code.

Here's an example of code that lacks symmetry:

```
void process() {
  input();
  count++;
  output();
}
```

The second statement is more concrete than the two messages. I would rewrite this on the basis of symmetry, resulting in:

```
void process() {
  input();
  incrementCount();
  output();
}
```

Still this method violates symmetry. The `input()` and `output()` operations are named after intentions, `incrementCount()` after an implementation. Looking for symmetries, I think about *why* I am incrementing the count, perhaps resulting in:

```
void process() {
  input();
  tally();
  output();
}
```

Often, finding and expressing symmetry is a preliminary step to removing duplication. If a similar thought exists in several places in the code, making them symmetrical to each other is a good first step towards unifying them.

Declarative Expression

Another principle behind the implementation patterns is to express as much of my intention as possible declaratively. Imperative programming is powerful and flexible, but to read it requires that you follow the thread of execution. I must build a model in my head of the state of the program and the flow of control and data. For those parts of a program that are more like simple facts, without sequence or conditionals, it is easier to read code that is simply declarative.

For example, in older versions of JUnit, classes could have a static `suite()` method that returned a set of tests to run.

```
public static junit.framework.Test suite() {
  Test result= new TestSuite();
  ...complicated stuff...
  return result;
}
```

Now comes the simple, common question—what tests are going to be run? In most cases, the `suite()` method just aggregates the tests in a bunch of classes. However, because the `suite()` method is general, I have to go read and understand the method if I want to be sure.

JUnit 4, on the other hand, uses the principle of declarative expression to solve the same problem. Instead of a method returning a suite of tests, there is a special test runner that runs the tests in a set of classes (the common case):

```
@RunWith(Suite.class)
@TestClasses({
  SimpleTest.class,
  ComplicatedTest.class
})
class AllTests {
}
```

If I know that tests are being aggregated using this method, I only need to look at the `TestClasses` annotation to see what tests will be run. Because the expression of the suite is declarative I don't need to suspect any tricky exceptions. This solution gives up the power and generality of the original `suite()` method, but the declarative style makes the code easier to read. (The `RunWith` annotation provides even more flexibility for running tests than the `suite()` method, but that's a story for a different book.)

Rate of Change

A final principle is to put logic or data that changes at the same rate together and separate logic or data that changes at different rates. These rates of change are a form of temporal symmetry. Sometimes the rate of change principle applies to changes a programmer makes. For example, if I am writing tax software I will separate code that makes general tax calculations from code that is particular to a given year. The code changes at different rates. When I make changes the following year, I would like to be sure that the code from preceding years still works. Separating them gives me more confidence in the local consequences of my changes.

The rate of change applies to data. All the fields in a single object should change at roughly the same rate. For example, fields that are modified only during the activation of a single method should be local variables. Two fields that change together but out of sync with their neighboring fields probably belong in a helper object. If a financial instrument can have its value and currency change together, then those two fields would probably better be expressed as a helper `Money` object:

```
setAmount(int value, String currency) {
  this.value= value;
  this.currency= currency;
}
```

becomes:

```
setAmount(int value, String currency) {
  this.value= new Money(value, currency);
}
```

and then later:

```
setAmount(Money value) {
  this.value= value;
}
```

The rate of change principle is an application of symmetry, but temporal symmetry. In the example above, the two original fields `value` and `currency` are symmetrical. They change at the same time. However, they are not symmetrical with the other fields in the object. Expressing the symmetry by putting them in their own object communicates their relationship to readers and is likely to set up further opportunities to reduce duplication and further localize consequences later.

Conclusion

This chapter has introduced the theoretical foundations of the implementation patterns. The values of communication, simplicity, and flexibility provide wide-ranging motivation for the patterns. The principles of local consequences, minimize repetition, logic and data together, symmetry, declarative expression, and rate of change help translate the values into action. Now we come to the patterns, specific solutions to repeating tactical problems in programming.

The next chapter, "Motivation", describes the economic factors that make focusing on communication through code a valuable activity.