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

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.
Third printing, January 2008

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. Thank you Bob Martin and Arturo Falco for catching typos. 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.

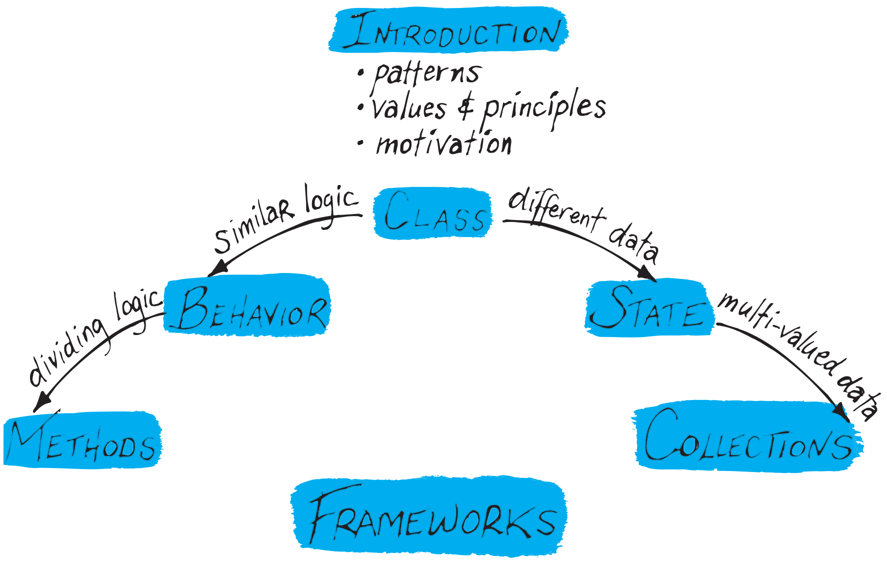

Figure 1.1 *Book overview*

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 7

Behavior

John Von Neumann contributed one of the primary metaphors of computing—a sequence of instructions that are executed one by one. This metaphor permeates most programming languages, Java included. The topic of this chapter is how to express the behavior of a program. The patterns are:

- Control Flow—Express computations as a sequence of steps.
- Main Flow—Clearly express the main flow of control.
- Message—Express control flow by sending a message.
- Choosing Message—Vary the implementors of a message to express choices.
- Double Dispatch—Vary the implementors of messages along two axes to express cascading choices.
- Decomposing Message—Break complicated calculations into cohesive chunks.
- Reversing Message—Make control flows symmetric by sending a sequence of messages to the same receiver.
- Inviting Message—Invite future variation by sending a message that can be implemented in different ways.
- Explaining Message—Send a message to explain the purpose of a clump of logic.
- Exceptional Flow—Express the unusual flows of control as clearly as possible without interfering with the expression of the main flow.
- Guard Clause—Express local exceptional flows by an early return.
- Exception—Express non-local exceptional flows with exceptions.

- Checked Exception—Ensure that exceptions are caught by declaring them explicitly.
- Exception Propagation—Propagate exceptions, transforming them as necessary so the information they contain is appropriate to the catcher.

Control Flow

Why do we have control flow in programs at all? There are languages like Prolog that don't have an explicit notion of a flow of control. Bits of logic float around in a soup, waiting for the right conditions before becoming active.

Java is a member of the family of languages in which the sequence of control is a fundamental organizing principle. Adjacent statements execute one after the other. Conditionals cause code to execute only in certain circumstances. Loops execute code repeatedly. Messages are sent to activate one of several subroutines. Exceptions cause control to jump up the stack.

All of these mechanisms add up to a rich medium for expressing computations. As an author/programmer, you decide whether to express the flow you have in mind as one main flow with exceptions, multiple alternative flows each of which is equally important, or some combination. You group bits of the control flow so they can be understood abstractly at first, for the casual reader, with greater detail available for those who need to understand them. Some groupings are routines in a class, some are by delegating control to another object.

Main Flow

Programmers generally have in mind a main flow of control for their programs. Processing starts here, ends there. There may be decisions and exceptions along the way, but the computation has a path to follow. Use your programming language to clearly express that flow.

Some programs, particularly those that are designed to work reliably in hostile circumstances, don't really have a visible main flow. These programs are in the minority, however. Using the expressive power of your programming language to clearly express little-executed, seldom-changed facts about your program obscures the more highly leveraged part of your program: the part that will be read, understood, and changed frequently. It's not that exceptional conditions are unimportant, just that focusing on expressing the main flow of the computation clearly is more valuable.

Therefore, clearly express the main flow of your program. Use exceptions and guard clauses to express unusual or error conditions.

Message

One of the primary means of expressing logic in Java is the message. Procedural languages use procedure calls as an information hiding mechanism:

```
compute() {
  input();
  process();
  output();
}
```

says, "For purposes of understanding this computation all you need to know is that it consists of these three steps, the details of which are not important at the moment." One of the beauties of programming with objects is that the same procedure also expresses something richer. For every method, there is potentially a whole set of similarly structured computations whose details differ. And, as an extra added bonus, you don't have to nail down the details of all those future variations when you write the invariant part.

Using messages as the fundamental control flow mechanism acknowledges that change is the base state of programs. Every message is a potential place where the receiver of the message can be changed without changing the sender. Rather than saying "There is something out there the details of which aren't important," the message-based version of the procedure says, "At this point in the story something interesting happens around the idea of input. The details may vary." Using this flexibility wisely, making clear and direct expressions of logic where possible and deferring details appropriately, is an important skill if you want to write programs that communicate effectively.

Choosing Message

Sometimes I send a message to choose an implementation, much as a case statement is used in procedural languages. For example, if I am going to display a graphic in one of several ways, I will send a polymorphic message to communicate that a choice will take place at runtime.

```
public void displayShape(Shape subject, Brush brush) {
  brush.display(subject);
}
```

The message `display()` chooses the implementation based on the runtime type of the brush. Then I am free to implement a variety of brushes: `ScreenBrush`, `PostscriptBrush`, and so on.

Liberal use of choosing messages leads to code with few explicit conditionals. Each choosing message is an invitation to later extension. Each explicit conditional is another point in your program that will require explicit modification in order to modify the behavior of the whole program.

Reading code that uses lots of choosing messages requires skill to learn. One of the costs of choosing messages is that a reader may have to look at several classes before understanding the details of a particular path through the computation. As a writer you can help the reader navigate by giving the methods intention-revealing names. Also, be aware of when a choosing message is overkill. If there is no possible variation in a computation, don't introduce a method just to provide the possibility of variation.

Double Dispatch

Choosing messages are good for expressing a single dimension of variability. In the example in "Choosing Message," this dimension was the type of medium on which the shape was to be drawn. If you need to express two independent dimensions of variability, you can cascade two choosing messages.

For example, suppose I wanted to express that a Postscript oval was computed differently than a screen rectangle. First I would decide where I wanted the computations to live. The base computations seeem like they belong in the `Brush`, so I will send a choosing message first to the `Shape`, then to the `Brush`:

```
displayShape(Shape subject, Brush brush) {
  subject.displayWith(brush);
}
```

Now each `Shape` has the opportunity to implement `displayWith()` differently. Rather than do any detailed work, however, they append their type onto the message and defer to the `Brush`:

```
Oval.displayWith(Brush brush) {
  brush.displayOval(this);
}
Rectangle.displayWith(Brush brush) {
  brush.displayRectangle(this);
}
```

Now the different kinds of brushes have the information they need to do their work:

```
PostscriptBrush.displayRectangle(Rectangle subject) {
  writer print(subject.left() +" " +...+ " rect);
}
```

Double dispatch introduces some duplication with a corresponding loss of flexibility. The type names of the receivers of the first choosing message get scattered over the methods in the receiver of the second choosing message. In this example, this means that to add a new Shape, I would have to add methods to all the Brushes. If one dimension is more likely to change than the other, make it the receiver of the second choosing message.

The computer scientist in me wants to generalize to triple, quadruple, quintuple dispatch. However, I've only ever attempted triple dispatch once and it didn't stay for long. I have always found clearer ways to express multi-dimensional logic.

Decomposing (Sequencing) Message

When you have a complicated algorithm composed of many steps, sometimes you can group related steps and send a message to invoke them. The intended purpose of the message isn't to provide a hook for specialization or anything sophisticated like that. It is just old-fashioned functional decomposition. The message is there simply to invoke the sub-sequence of steps in the routine.

Decomposing messages need to be descriptively named. Most readers should be able to gather what they need to know about the purpose of the sub-sequence from the name alone. Only those readers interested in implementation details should have to read the code invoked by the decomposing message.

Difficulty naming a decomposing message is a tip-off that this isn't the right pattern to use. Another tip-off is long parameter lists. If I see these symptoms, I inline the method invoked by the decomposing message and apply a different pattern, like Method Object, to help me communicate the structure of the program.

Reversing Message

Symmetry can improve the readability of code. Consider the following code:

```
void compute() {
  input();
  helper.process(this);
  output();
}
```

While this method is composed of three others, it lacks symmetry. The readability of the method is improved by introducing a helper method that reveals the latent symmetry. Now when reading compute(), I don't have to keep track of who is sent the messages—they all go to this.

```
void process(Helper helper) {
  helper.process(this);
}
void compute() {
  input();
  process(helper);
  output();
}
```

Now the reader can understand how the `compute()` method is structured by reading a single class.

Sometimes the helper method invoked by a reversing message becomes important on its own. Sometimes, overuse of reversing messages can obscure the need to move functionality. If we had the following code:

```
void input(Helper helper) {
  helper.input(this);
}
void output(Helper helper) {
  helper.output(this);
}
```

it would probably be better structured by moving the whole `compute()` method to the `Helper` class:

```
compute() {
  new Helper(this).compute();
}
Helper.compute() {
  input();
  process();
  output();
}
```

Sometimes I feel silly introducing methods “just” to satisfy an “aesthetic” urge like symmetry. Aesthetics go deeper than that. Aesthetics engage more of your brain than strictly linear logical thought. Once you have cultivated your sense of the aesthetics of code, the aesthetic impressions you receive of your code is valuable feedback about the quality of the code. Those feelings that bubble up from below the surface of symbolic thought can be as valuable as your explicitly named and justified patterns.

Inviting Message

Sometimes as you are writing code, you expect that people will want to vary a part of the computation in a subclass. Send an appropriately named message to communicate the possibility of later refinement. The message invites programmers to refine the computation for their own purposes later.

If there is a default implementation of the logic, make it the implementation of the message. If not, declare the method abstract to make the invitation explicit.

Explaining Message

The distinction between intention and implementation has always been important in software development. It is what allows you to understand a computation first in essence and later, if necessary, in detail. You can use messages to make this distinction by sending a message named after the problem you are solving which in turn sends a message named after how the problem is to be solved.

The first example I saw of this was in Smalltalk. Transliterated, the method that caught my eye was this:

```
highlight(Rectangle area) {
  reverse(area);
}
```

I thought, "Why is this useful? Why not just call `reverse()` directly instead of calling the intermediate `highlight()` method?" After some thought, though, I realized that while `highlight()` didn't have a computational purpose, it did serve to communicate an intention. Calling code could be written in terms of what problem they were trying to solve, namely highlighting an area of the screen.

Consider introducing an explaining message when you are tempted to comment a single line of code. When I see:

```
flags|= LOADED_BIT; // Set the loaded bit
```

I would rather read:

```
setLoadedFlag();
```

Even though the implementation of `setLoadedFlag()` is trivial. The one-line method is there to communicate.

```
void setLoadedFlag() {
  flags|= LOADED_BIT;
}
```

Sometimes the helper methods invoked by explaining messages become valuable points for further extension. It's nice to get lucky when you can. However, my main purpose in invoking an explaining message is to communicate my intention more clearly.

Exceptional Flow

Just as programs have a main flow, they can also have one or more exceptional flows. These are paths of computation that are less important to communicate because they are less-frequently executed, less-frequently changed, or conceptually less important than the main flow. Express the main flow clearly, and these exceptional paths as clearly as possible without obscuring the main flow. Guard clauses and exceptions are two ways of expressing exceptional flows.

Programs are easiest to read if the statements execute one after another. Readers can use comfortable and familiar prose-reading skills to understand the intent of the program. Sometimes, though, there are multiple paths through a program. Expressing all paths equally would result in a bowl of worms, with flags set *here* and used *there* and return values with special meanings. Answering the basic question, "What statements are executed?" becomes an exercise in a combination of archaeology and logic. Pick the main flow. Express it clearly. Use exceptions to express other paths.

Guard Clause

While programs have a main flow, some situations require deviations from the main flow. The guard clause is a way to express simple and local exceptional situations with purely local consequences. Compare the following:

```
void initialize() {
  if (!isInitialized()) {
    ...
  }
}
```

with:

```
void initialize() {
  if (isInitialized())
    return;
  ...
}
```

When I read the first version, I make a note to look for an else clause while I am reading the then clause. I mentally put the condition on a stack. All of this is a distraction while I am reading the body of the then clause. The first two lines of the second version simply give me a fact to note: the receiver hasn't been initialized.

If-then-else expresses alternative, equally important control flows. A guard clause is appropriate for expressing a different situation, one in which one of the control flows is more important than the other. In the initialization example above, the important control flow is what happens when the object is initialized. Other than that, there is just a simple fact to notice, that even if an object is asked to initialize multiple times it will only execute the initialization code once.

Back in the old days of programming, a commandment was issued: each routine shall have a single entry and a single exit. This was to prevent the confusion possible when jumping into and out of many locations in the same routine. It made good sense when applied to FORTRAN or assembly language programs written with lots of global data where even understanding which statements were executed was hard work. In Java, with small methods and mostly local data, it is needlessly conservative. However, this bit of programming folklore, thoughtlessly obeyed, prevents the use of guard clauses.

Guard clauses are particularly useful when there are multiple conditions:

```
void compute() {
  Server server= getServer();
  if (server != null) {
    Client client= server.getClient();
    if (client != null) {
      Request current= client.getRequest();
      if (current != null)
        processRequest(current);
    }
  }
}
```

Nested conditionals breed defects. The guard clause version of the same code notes the prerequisites to processing a request without complex control structures:

```
void compute() {
  Server server= getServer();
  if (server == null)
    return;
  Client client= server.getClient();
  if (client == null)
    return;
  Request current= client.getRequest();
  if (current == null)
    return;
  processRequest(current);
}
```

A variant of guard clause is the `continue` statement used in a loop. It says, “Never mind this element. Go on to the next one.”

```
while (line = reader.readline()) {
  if (line.startsWith('#') || line.isEmpty())
    continue;
  // Normal processing here
}
```

Again, the intent is to point out the (strictly local) difference between normal and exceptional processing.

Exception

Exceptions are useful for expressing jumps in program flow that span levels of function invocation. If you realize many levels up on the stack that a problem has occurred—a disk is full or a network connection has been lost—you may only be able to reasonably deal with that fact much lower down on the call stack. Throwing an exception at the point of discovery and catching at the point where it can be handled is much better than cluttering all the intervening code with explicit checks for all the possible exceptional conditions, none of which can be handled.

Exceptions cost. They are a form of design leakage. The fact that the called method throws an exception influences the design and implementation of all possible calling methods until the method is reached that catches the exception. They make it difficult to trace the flow of control, since adjacent statements can be in different methods, objects, or packages. Code that could be written with conditionals and messages, but is implemented with exceptions, is fiendishly difficult to read as you are forever trying to figure out what more is going on than a simple control structure. In short, express control flows with sequence, messages, iteration, and conditionals (in that order) wherever possible. Use exceptions when not doing so would confuse the simply communicated main flow.

Checked Exceptions

One of the dangers of exceptions is what happens if you throw an exception but no one catches it. The program terminates, that's what happens. But you'd like to control when the program terminates unexpectedly, printing out information necessary to diagnose the situation and telling the user what has happened.

Exceptions that are thrown but not caught are an even bigger risk when different people write the code that throws the exception and the code that catches the exception. Any missed communication results in an abrupt and impolite program termination.

To avoid this situation, Java has checked exceptions. These are declared explicitly by the programmer and checked by the compiler. Code that is subject to having a checked exception thrown at it must either catch the exception or pass it along.

Checked exceptions come with considerable costs. First is the cost of the declarations themselves. These can easily add 50% to the length of method declarations and add another thing to read and understand along the levels between the thrower and catcher. Checked exceptions also make changing code more difficult. Refactoring code with checked exceptions is more difficult and tedious than code without, even though modern IDEs reduce the burden.

Exception Propagation

Exceptions occur at different levels of abstraction. Catching and reporting a low-level exception can be confusing to someone who is not expecting it. When a web server shows me an error page with stack trace headed by a `NullPointerException`, I'm not sure what I'm supposed to do with the information. I'd rather see a message that said, "The programmer did not consider the scenario you have just presented." I wouldn't mind if the page also provided a pointer to further information that I could send to a programmer so he could diagnose the problem, but presenting me with untranslated details isn't helpful.

Low-level exceptions often contain valuable information for diagnosing a defect. Wrap the low-level exception in the higher-level exception so that when the exception is printed, on a log for example, enough information is written to help find the defect.

Conclusion

Control flows between methods of a program built from objects. The next chapter describes using methods to express the concepts in a computation.

Index