Code, without tests, is not clean. No matter how elegant it is, no matter how readable and accessible, if it hath not tests, it be unclean.

In priority order, simple code:

• Runs all the tests;

• Contains no duplication;

• Expresses all the design ideas that are in the system;

• Minimizes the number of entities such as classes, methods, functions, and the like.

**Meaningful Names**

### **USE INTENTION-REVEALING NAMES**

Choosing names that reveal intent can make it much easier to understand and change code.

### **AVOID DISINFORMATION**

We should avoid words whose entrenched meanings vary from our intended meaning. Spelling similar concepts similarly is information. Using inconsistent spellings is disinformation. A truly awful example of disinformative names would be the use of lower-case L or uppercase O as variable names, especially in combination.

### **MAKE MEANINGFUL DISTINCTIONS**

or example, because you can’t use the same name to refer to two different things in the same scope, you might be tempted to change one name in an arbitrary way. Sometimes this is done by misspelling one, leading to the surprising situation where correcting spelling errors leads to an inability to compile.

It is not sufficient to add number series or noise words, even though the compiler is satisfied. If names must be different, then they should also mean something different.

In the absence of specific conventions, the variable moneyAmount is indistinguishable from money, customerInfo is indistinguishable from customer, accountData is indistinguishable from account, and theMessage is indistinguishable from message. Distinguish names in such a way that the reader knows what the differences offer.

### **USE PRONOUNCEABLE NAMES**

If you can’t pronounce it, you can’t discuss it without sounding like an idiot.

### **USE SEARCHABLE NAMES**

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

### **AVOID ENCODINGS**

We have enough encodings to deal with without adding more to our burden. Encoding type or scope information into names simply adds an extra burden of deciphering. It hardly seems reasonable to require each new employee to learn yet another encoding “language” in addition to learning the (usually considerable) body of code that they’ll be working in. It is an unnecessary mental burden when trying to solve a problem. Encoded names are seldom pronounceable and are easy to mis-type.

#### **Hungarian Notation**

In days of old, when we worked in name-length-challenged languages, we violated this rule out of necessity, and with regret. Fortran forced encodings by making the first letter a code for the type. Early versions of BASIC allowed only a letter plus one digit. Hungarian Notation (HN) took this to a whole new level.

#### **Member Prefixes**

You also don’t need to prefix member variables with m\_ anymore. Your classes and functions should be small enough that you don’t need them. And you should be using an editing environment that highlights or colorizes members to make them distinct.

#### **Interfaces and Implementations**

These are sometimes a special case for encodings. For example, say you are building an ABSTRACTFACTORY for the creation of shapes. This factory will be an interface and will be implemented by a concrete class. What should you name them? IShapeFactory and ShapeFactory? I prefer to leave interfaces unadorned. The preceding I, so common in today’s legacy wads, is a distraction at best and too much information at worst. I don’t want my users knowing that I’m handing them an interface. I just want them to know that it’s a ShapeFactory. So if I must encode either the interface or the implementation, I choose the implementation. Calling it ShapeFactoryImp, or even the hideous CShapeFactory, is preferable to encoding the interface.

### **AVOID MENTAL MAPPING**

This is a problem with single-letter variable names. Certainly a loop counter may be named i or j or k(though never l!) if its scope is very small and no other names can conflict with it.

### **CLASS NAMES**

Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.

### **METHOD NAMES**

Methods should have verb or verb phrase names like postPayment, deletePage, or save. Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the javabean standard.

When constructors are overloaded, use static factory methods with names that describe the arguments. For example,

   Complex fulcrumPoint = Complex.FromRealNumber(23.0);

is generally better than

   Complex fulcrumPoint = new Complex(23.0);

Consider enforcing their use by making the corresponding constructors private.

### **DON’T BE CUTE**

If names are too clever, they will be memorable only to people who share the author’s sense of humor, and only as long as these people remember the joke.

### **PICK ONE WORD PER CONCEPT**

Pick one word for one abstract concept and stick with it. For instance, it’s confusing to have fetch, retrieve, and get as equivalent methods of different classes. How do you remember which method name goes with which class? Sadly, you often have to remember which company, group, or individual wrote the library or class in order to remember which term was used. Otherwise, you spend an awful lot of time browsing through headers and previous code samples.

A consistent lexicon is a great boon to the programmers who must use your code.

### **DON’T PUN**

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.

### **USE SOLUTION DOMAIN NAMES**

Remember that the people who read your code will be programmers. So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth. It is not wise to draw every name from the problem domain because we don’t want our coworkers to have to run back and forth to the customer asking what every name means when they already know the concept by a different name.

### **USE PROBLEM DOMAIN NAMES**

When there is no “programmer-eese” for what you’re doing, use the name from the problem domain. At least the programmer who maintains your code can ask a domain expert what it means.

Separating solution and problem domain concepts is part of the job of a good programmer and designer. The code that has more to do with problem domain concepts should have names drawn from the problem domain.

### **ADD MEANINGFUL CONTEXT**

There are a few names which are meaningful in and of themselves—most are not.

Imagine that you have variables named firstName, lastName, street, houseNumber, city, state, and zipcode. Taken together it’s pretty clear that they form an address. But what if you just saw the state variable being used alone in a method? Would you automatically infer that it was part of an address?

You can add context by using prefixes: addrFirstName, addrLastName, addrState, and so on. At least readers will understand that these variables are part of a larger structure. Of course, a better solution is to create a class named Address. Then, even the compiler knows that the variables belong to a bigger concept.

### **DON’T ADD GRATUITOUS CONTEXT**

Shorter names are generally better than longer ones, so long as they are clear. Add no more context to a name than is necessary.

### **FINAL WORDS**

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don’t learn to do it very well.

## ****Functions****

### **SMALL!**

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

#### **Blocks and Indenting**

This implies that the blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.

This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand.

### **DO ONE THING**

***FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.***

#### **Sections within Functions**

Look at [Listing 4-7](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter04.html#ch4lt1) on page [71](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter04.html#page_71). Notice that the generatePrimes function is divided into sections such as declarations, initializations, and sieve. This is an obvious symptom of doing more than one thing. Functions that do one thing cannot be reasonably divided into sections.

### **ONE LEVEL OF ABSTRACTION PER FUNCTION**

In order to make sure our functions are doing “one thing,” we need to make sure that the statements within our function are all at the same level of abstraction. It is easy to see how [Listing 3-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lt1) violates this rule. There are concepts in there that are at a very high level of abstraction, such as getHtml(); others that are at an intermediate level of abstraction, such as: String pagePathName = PathParser.render(pagePath); and still others that are remarkably low level, such as: .append(”\n”).

### **SWITCH STATEMENTS**

It’s hard to make a small switch statement.6 Even a switch statement with only two cases is larger than I’d like a single block or function to be. It’s also hard to make a switch statement that does one thing. By their nature, switch statements always do N things. Unfortunately we can’t always avoid switch statements, but we can make sure that each switch statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.

The solution to this problem (see [Listing 3-5](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lt5)) is to bury the switch statement in the basement of an ABSTRACT FACTORY,9 and never let anyone see it.

public Money calculatePay(Employee e)   
   throws InvalidEmployeeType {  
       switch (e.type) {  
         case COMMISSIONED:  
           return calculateCommissionedPay(e);  
         case HOURLY:  
           return calculateHourlyPay(e);  
         case SALARIED:  
           return calculateSalariedPay(e);  
         default:  
           throw new InvalidEmployeeType(e.type);  
       }  
     }

change to:

public abstract class Employee {  
     public abstract boolean isPayday();  
     public abstract Money calculatePay();  
     public abstract void deliverPay(Money pay);  
   }  
   -----------------  
   public interface EmployeeFactory {  
     public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;  
   }  
   -----------------  
   public class EmployeeFactoryImpl implements  
          EmployeeFactory {  
     public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {  
       switch (r.type) {  
         case COMMISSIONED:  
           return new CommissionedEmployee(r) ;  
         case HOURLY:  
           return new HourlyEmployee(r);  
         case SALARIED:  
           return new SalariedEmploye(r);  
         default:  
           throw new InvalidEmployeeType(r.type);  
       }  
     }  
   }

### **USE DESCRIPTIVE NAMES**

Don’t be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment. Use a naming convention that allows multiple words to be easily read in the function names, and then make use of those multiple words to give the function a name that says what it does.

Don’t be afraid to spend time choosing a name. Indeed, you should try several different names and read the code with each in place.

Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.

Be consistent in your names. Use the same phrases, nouns, and verbs in the function names you choose for your modules. Consider, for example, the names includeSetup-AndTeardownPages, includeSetupPages, includeSuiteSetupPage, and includeSetupPage. The similar phraseology in those names allows the sequence to tell a story.

### **FUNCTION ARGUMENTS**

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn’t be used anyway.

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going *in* to the function through arguments and *out* through the return value. We don’t usually expect information to be going out through the arguments. So output arguments often cause us to do a double-take.

One input argument is the next best thing to no arguments. SetupTeardown-Includer.render(pageData) is pretty easy to understand. Clearly we are going to *render* the data in the pageData object.

#### **Common Monadic Forms**

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in boolean fileExists(“MyFile”). Or you may be operating on that argument, transforming it into something else and returning it. For example, InputStreamfileOpen(“MyFile”) transforms a file name String into an InputStream return value. These two uses are what readers expect when they see a function.

A somewhat less common, but still very useful form for a single argument function, is an event. In this form there is an input argument but no output argument. For example, void passwordAttemptFailedNtimes(int attempts). Use this form with care. It should be very clear to the reader that this is an event. Choose names and contexts carefully.

Using an output argument instead of a return value for a transformation is confusing. If a function is going to transform its input argument, the transformation should appear as the return value. Indeed, StringBuffer transform(StringBuffer in) is better than void transform(StringBuffer out), even if the implementation in the first case simply returns the input argument. At least it still follows the form of a transformation.

#### **Flag Arguments**

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!

#### **Dyadic Functions**

A function with two arguments is harder to understand than a monadic function. For example, writeField(name) is easier to understand than writeField(output-Stream, name).10Though the meaning of both is clear.

There are times, of course, where two arguments are appropriate.

Dyads aren’t evil, and you will certainly have to write them. However, you should be aware that they come at a cost and should take advantage of what mechanisms may be available to you to convert them into monads.

#### **Triads**

Functions that take three arguments are significantly harder to understand than dyads. The issues of ordering, pausing, and ignoring are more than doubled. I suggest you think very carefully before creating a triad.

#### **Argument Objects**

 Circle makeCircle(double x, double y, double radius);  
   Circle makeCircle(Point center, double radius);

Reducing the number of arguments by creating objects out of them may seem like cheating, but it’s not. When groups of variables are passed together, the way x and y are in the example above, they are likely part of a concept that deserves a name of its own.

#### **Argument Lists**

Sometimes we want to pass a variable number of arguments into a function. Consider, for example, the String.format method:

   String.format(”%s worked %.2f hours.”, name, hours);

If the variable arguments are all treated identically, as they are in the example above, then they are equivalent to a single argument of type List. By that reasoning, String.format is actually dyadic. Indeed, the declaration of String.format as shown below is clearly dyadic.

   public String format(String format, Object… args)

#### **Verbs and Keywords**

Choosing good names for a function can go a long way toward explaining the intent of the function and the order and intent of the arguments. In the case of a monad, the function and argument should form a very nice verb/noun pair. For example, write(name) is very evocative. Whatever this “name” thing is, it is being “written.” An even better name might be writeField(name), which tells us that the “name” thing is a “field.”

### **HAVE NO SIDE EFFECTS**

Side effects are lies. Your function promises to do one thing, but it also does other hidden things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies.

Consider, for example, the seemingly innocuous function in [Listing 3-6](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lt6). This function uses a standard algorithm to match a userName to a password. It returns true if they match and false if anything goes wrong. But it also has a side effect. Can you spot it?

**Listing 3-6 UserValidator.java**

   public class UserValidator {  
     private Cryptographer cryptographer;  
  
     public boolean checkPassword(String userName, String password) {  
       User user = UserGateway.findByName(userName);  
       if (user != User.NULL) {  
         String codedPhrase = user.  
         getPhraseEncodedByPassword();  
         String phrase = cryptographer.decrypt(codedPhrase, password);  
         if ("Valid Password".equals(phrase)) {  
           Session.initialize();  
           return true;  
         }  
       }  
       return false;  
     }  
   }

The side effect is the call to Session.initialize(), of course. The checkPassword function, by its name, says that it checks the password. The name does not imply that it initializes the session. So a caller who believes what the name of the function says runs the risk of erasing the existing session data when he or she decides to check the validity of the user.

This side effect creates a temporal coupling. That is, checkPassword can only be called at certain times (in other words, when it is safe to initialize the session). If it is called out of order, session data may be inadvertently lost. Temporal couplings are confusing, especially when hidden as a side effect. If you must have a temporal coupling, you should make it clear in the name of the function. In this case we might rename the function checkPasswordAndInitializeSession, though that certainly violates “Do one thing.”

#### **Output Arguments**

For example:

   appendFooter(s);

Does this function append s as the footer to something? Or does it append some footer to s? Is s an input or an output? It doesn’t take long to look at the function signature and see:

   public void appendFooter(StringBuffer report)

This clarifies the issue, but only at the expense of checking the declaration of the function. Anything that forces you to check the function signature is equivalent to a double-take. It’s a cognitive break and should be avoided.

report.appendFooter();

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

### **COMMAND QUERY SEPARATION**

This leads to odd statements like this:

   if (set(”username”, ”unclebob”))…

Imagine this from the point of view of the reader. What does it mean? Is it asking whether the “username” attribute was previously set to “unclebob”? Or is it asking whether the “username” attribute was successfully set to “unclebob”? It’s hard to infer the meaning from the call because it’s not clear whether the word “set” is a verb or an adjective.

The real solution is to separate the command from the query so that the ambiguity cannot occur.

   if (attributeExists(”username”)) {  
     setAttribute(”username”, ”unclebob”);  
     …  
   }

### **PREFER EXCEPTIONS TO RETURNING ERROR CODES**

Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of if statements.

 When you return an error code, you create the problem that the caller must deal with the error immediately.

   if (deletePage(page) == E\_OK) {  
     if (registry.deleteReference(page.name) == E\_OK) {  
       if (configKeys.deleteKey(page.name.makeKey()) == E\_OK){  
         logger.log("page deleted");  
       } else {  
         logger.log("configKey not deleted");  
       }  
     } else {  
       logger.log("deleteReference from registry failed");  
     }  
   } else {  
     logger.log("delete failed");  
     return E\_ERROR;  
   }

On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code and can be simplified:

   try {  
     deletePage(page);  
     registry.deleteReference(page.name);  
     configKeys.deleteKey(page.name.makeKey());  
   }  
   catch (Exception e) {  
     logger.log(e.getMessage());  
   }

#### **Extract Try/Catch Blocks**

Try/catch blocks are ugly in their own right. They confuse the structure of the code and mix error processing with normal processing. So it is better to extract the bodies of the try and catch blocks out into functions of their own.

   public void delete(Page page) {  
     try {  
       deletePageAndAllReferences(page);  
     }  
     catch (Exception e) {  
       logError(e);  
     }  
   }  
  
   private void deletePageAndAllReferences(Page page) throws Exception {  
     deletePage(page);  
     registry.deleteReference(page.name);  
     configKeys.deleteKey(page.name.makeKey());  
   }  
  
   private void logError(Exception e) {  
     logger.log(e.getMessage());  
   }

#### **Error Handling Is One Thing**

Functions should do one thing. Error handing is one thing. Thus, a function that handles errors should do nothing else.

When you use exceptions rather than error codes, then new exceptions are derivatives of the exception class. They can be added without forcing any recompilation or redeployment.

### **DON’T REPEAT YOURSELF**

Duplication may be the root of all evil in software. Many principles and practices have been created for the purpose of controlling or eliminating it.

### **STRUCTURED PROGRAMMING**

Some programmers follow Edsger Dijkstra’s rules of structured programming.14 Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one return statement in a function, no break or continuestatements in a loop, and never, ever, any goto statements.

## ****Comments****

*Don’t comment bad code—rewrite it.*

So when you find yourself in a position where you need to write a comment, think it through and see whether there isn’t some way to turn the tables and express yourself in code. Every time you express yourself in code, you should pat yourself on the back. Every time you write a comment, you should grimace and feel the failure of your ability of expression.

Inaccurate comments are far worse than no comments at all. They delude and mislead. They set expectations that will never be fulfilled. They lay down old rules that need not, or should not, be followed any longer.

### **COMMENTS DO NOT MAKE UP FOR BAD CODE**

Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.

### **EXPLAIN YOURSELF IN CODE**

Which would you rather see? This:

   // Check to see if the employee is eligible for full benefits  
   if ((employee.flags & HOURLY\_FLAG) &&  
       (employee.age > 65))

Or this?

   if (employee.isEligibleForFullBenefits())

It takes only a few seconds of thought to explain most of your intent in code. In many cases it’s simply a matter of creating a function that says the same thing as the comment you want to write.

### **GOOD COMMENTS**

#### **Legal Comments**

Sometimes our corporate coding standards force us to write certain comments for legal reasons. For example, copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

#### **Informative Comments**

It is sometimes useful to provide basic information with a comment. For example, consider this comment that explains the return value of an abstract method:

   // format matched kk:mm:ss EEE, MMM dd, yyyy  
   Pattern timeMatcher = Pattern.compile(  
     “\\d\*:\\d\*:\\d\* \\w\*, \\w\* \\d\*, \\d\*”);

#### **Explanation of Intent**

You might not agree with the programmer’s solution to the problem, but at least you know what he was trying to do.

   public void testConcurrentAddWidgets() throws Exception {  
     WidgetBuilder widgetBuilder =  
       new WidgetBuilder(new Class[]{BoldWidget.class});  
       String text = ”’’’bold text’’’”;  
       ParentWidget parent =  
         new BoldWidget(new MockWidgetRoot(), ”’’’bold text’’’”);  
       AtomicBoolean failFlag = new AtomicBoolean();  
       failFlag.set(false);  
     
       **//This is our best attempt to get a race condition  
       //by creating large number of threads.**  
       for (int i = 0; i < 25000; i++) {  
         WidgetBuilderThread widgetBuilderThread =  
           new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);  
         Thread thread = new Thread(widgetBuilderThread);  
         thread.start();  
       }  
       assertEquals(false, failFlag.get());  
     }

#### **Clarification**

Sometimes it is just helpful to translate the meaning of some obscure argument or return value into something that’s readable.

   public void testCompareTo() throws Exception  
   {  
     WikiPagePath a = PathParser.parse("PageA");  
     WikiPagePath ab = PathParser.parse("PageA.PageB");  
     WikiPagePath b = PathParser.parse("PageB");  
     WikiPagePath aa = PathParser.parse("PageA.PageA");  
     WikiPagePath bb = PathParser.parse("PageB.PageB");  
     WikiPagePath ba = PathParser.parse("PageB.PageA");  
   
     assertTrue(a.compareTo(a) == 0);    // a == a  
     assertTrue(a.compareTo(b) != 0);    // a != b  
     assertTrue(ab.compareTo(ab) == 0);  // ab == ab  
     assertTrue(a.compareTo(b) == -1);   // a < b  
     assertTrue(aa.compareTo(ab) == -1); // aa < ab  
     assertTrue(ba.compareTo(bb) == -1); // ba < bb  
     assertTrue(b.compareTo(a) == 1);    // b > a  
     assertTrue(ab.compareTo(aa) == 1);  // ab > aa  
     assertTrue(bb.compareTo(ba) == 1);  // bb > ba  
   }

#### **Warning of Consequences**

Sometimes it is useful to warn other programmers about certain consequences.

**// Don't run unless you   
   // have some time to kill.**  
   public void \_testWithReallyBigFile()  
   {  
     writeLinesToFile(10000000);  
     
     response.setBody(testFile);  
     response.readyToSend(this);  
     String responseString = output.toString();  
     assertSubString("Content-Length: 1000000000", responseString);  
     assertTrue(bytesSent > 1000000000);  
   }

#### **TODO Comments**

TODOs are jobs that the programmer thinks should be done, but for some reason can’t do at the moment. It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem. It might be a request for someone else to think of a better name or a reminder to make a change that is dependent on a planned event. Whatever else a TODO might be, it is not an excuse to leave bad code in the system.

#### **Amplification**

A comment may be used to amplify the importance of something that may otherwise seem inconsequential.

   String listItemContent = match.group(3).trim();  
   **// the trim is real important.  It removes the starting  
   // spaces that could cause the item to be recognized  
   // as another list.**  
   new ListItemWidget(this, listItemContent, this.level + 1);  
   return buildList(text.substring(match.end()));

#### **Javadocs in Public APIs**

There is nothing quite so helpful and satisfying as a well-described public API. The java-docs for the standard Java library are a case in point. It would be difficult, at best, to write Java programs without them.

### **BAD COMMENTS**

#### **Mumbling**

   public void loadProperties()  
   {  
     try  
     {  
      String propertiesPath = propertiesLocation +  
       ”/” + PROPERTIES\_FILE;  
      FileInputStream propertiesStream = new  
       FileInputStream(propertiesPath);  
      loadedProperties.load(propertiesStream);  
     }  
     catch(IOException e)  
     {  
       **// No properties files means all defaults are loaded**  
     }  
   }

#### **Redundant Comments**

 The comment probably takes longer to read than the code itself.

#### **Misleading Comments**

This subtle bit of misinformation, couched in a comment that is harder to read than the body of the code, could cause another programmer to blithely call this function in the expectation that it will return as soon as this.closed becomes true. That poor programmer would then find himself in a debugging session trying to figure out why his code executed so slowly.

#### **Mandated Comments**

It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment.

#### **Journal Comments**

Sometimes people add a comment to the start of a module every time they edit it. These comments accumulate as a kind of journal, or log, of every change that has ever been made. I have seen some modules with dozens of pages of these run-on journal entries.

#### **Noise Comments**

Sometimes you see comments that are nothing but noise. They restate the obvious and provide no new information.

#### **Scary Noise**

   /\*\* The name. \*/  
   private String name;

#### **Don’t Use a Comment When You Can Use a Function or a Variable**

Consider the following stretch of code:

   // does the module from the global list <mod> depend on the  
   // subsystem we are part of?  
   if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))

This could be rephrased without the comment as

   ArrayList moduleDependees = smodule.getDependSubsystems();  
   String ourSubSystem = subSysMod.getSubSystem();  
   if (moduleDependees.contains(ourSubSystem))

#### **Position Markers**

Sometimes programmers like to mark a particular position in a source file.

 A banner is startling and obvious if you don’t see banners very often. So use them very sparingly, and only when the benefit is significant. If you overuse banners, they’ll fall into the background noise and be ignored.

#### **Closing Brace Comments**

So if you find yourself wanting to mark your closing braces, try to shorten your functions instead.

#### **Attributions and Bylines**

 /\* Added by Rick \*/

You might think that such comments would be useful in order to help others know who to talk to about the code. But the reality is that they tend to stay around for years and years, getting less and less accurate and relevant.

#### **Commented-Out Code**

Few practices are as odious as commenting-out code. Don’t do this!

#### **HTML Comments**

It makes the comments hard to read in the one place where they should be easy to read—the editor/IDE.

#### **Nonlocal Information**

If you must write a comment, then make sure it describes the code it appears near.

#### **Too Much Information**

Don’t put interesting historical discussions or irrelevant descriptions of details into your comments.

#### **Inobvious Connection**

The connection between a comment and the code it describes should be obvious. If you are going to the trouble to write a comment, then at least you’d like the reader to be able to look at the comment and the code and understand what the comment is talking about.

#### **Function Headers**

Short functions don’t need much description. A well-chosen name for a small function that does one thing is usually better than a comment header.

#### **Javadocs in Nonpublic Code**

Generating javadoc pages for the classes and functions inside a system is not generally useful, and the extra formality of the javadoc comments amounts to little more than cruft and distraction.

## ****Formatting****

### **VERTICAL FORMATTING**

#### **The Newspaper Metaphor**

Think of a well-written newspaper article. You read it vertically. At the top you expect a headline that will tell you what the story is about and allows you to decide whether it is something you want to read. The first paragraph gives you a synopsis of the whole story, hiding all the details while giving you the broad-brush concepts. As you continue downward, the details increase until you have all the dates, names, quotes, claims, and other minutia.

We would like a source file to be like a newspaper article. The name should be simple but explanatory. The name, by itself, should be sufficient to tell us whether we are in the right module or not. The topmost parts of the source file should provide the high-level concepts and algorithms. Detail should increase as we move downward, until at the end we find the lowest level functions and details in the source file.

It appears to be possible to build significant systems (FitNesse is close to 50,000 lines) out of files that are typically 200 lines long, with an upper limit of 500. Although this should not be a hard and fast rule, it should be considered very desirable. Small files are usually easier to understand than large files ar

#### **Vertical Openness Between Concepts**

There are blank lines that separate the package declaration, the import(s), and each of the functions. This extremely simple rule has a profound effect on the visual layout of the code. Each blank line is a visual cue that identifies a new and separate concept. As you scan down the listing, your eye is drawn to the first line that follows a blank line.

This effect is even more pronounced when you unfocus your eyes. In the first example the different groupings of lines pop out at you, whereas the second example looks like a muddle. The difference between these two listings is a bit of vertical openness.

#### **Vertical Density**

 So lines of code that are tightly related should appear vertically dense.

#### **Vertical Distance**

Concepts that are closely related should be kept vertically close to each other [G10]. Clearly this rule doesn’t work for concepts that belong in separate files. But then closely related concepts should not be separated into different files unless you have a very good reason. Indeed, this is one of the reasons that protected variables should be avoided.

**Variable Declarations.** Variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear a the top of each function, as in this longish function from Junit4.3.1.

Control variables for loops should usually be declared within the loop statement, as in this cute little function from the same source.

**Instance variables,** on the other hand, should be declared at the top of the class. This should not increase the vertical distance of these variables, because in a well-designed class, they are used by many, if not all, of the methods of the class.

**Dependent Functions.** If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use.

**Conceptual Affinity.** Certain bits of code want to be near other bits. They have a certain conceptual affinity. The stronger that affinity, the less vertical distance there should be between them.

As we have seen, this affinity might be based on a direct dependence, such as one function calling another, or a function using a variable. But there are other possible causes of affinity. Affinity might be caused because a group of functions perform a similar operation.

#### **Vertical Ordering**

In general we want function call dependencies to point in the downward direction.

### **HORIZONTAL FORMATTING**

I personally set my limit at 120.

#### **Horizontal Openness and Density**

On the other hand, I didn’t put spaces between the function names and the opening parenthesis. This is because the function and its arguments are closely related. Separating them makes them appear disjoined instead of conjoined. I separate arguments within the function call parenthesis to accentuate the comma and show that the arguments are separate.

#### **Horizontal Alignment**

Nowadays I prefer unaligned declarations and assignments, as shown below, because they point out an important deficiency. If I have long lists that need to be aligned, the problem is the length of the lists, not the lack of alignment.

#### **Indentation**

Programmers rely heavily on this indentation scheme. They visually line up lines on the left to see what scope they appear in. This allows them to quickly hop over scopes, such as implementations of if or while statements, that are not relevant to their current situation. They scan the left for new method declarations, new variables, and even new classes. Without indentation, programs would be virtually unreadable by humans.

**Breaking Indentation.** It is sometimes tempting to break the indentation rule for short ifstatements, short while loops, or short functions. Whenever I have succumbed to this temptation, I have almost always gone back and put the indentation back in.

#### **Dummy Scopes**

Sometimes the body of a while or for statement is a dummy, as shown below. I don’t like these kinds of structures and try to avoid them. When I can’t avoid them, I make sure that the dummy body is properly indented and surrounded by braces. I can’t tell you how many times I’ve been fooled by a semicolon silently sitting at the end of a while loop on the same line. Unless you make that semicolon visible by indenting it on it’s own line, it’s just too hard to see.

   while (dis.read(buf, 0, readBufferSize) != -1)     ;

### **TEAM RULES**

These were not the rules that I prefer; they were rules decided by the team.

## ****Objects and Data Structures****

### **DATA ABSTRACTION**

Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.

### **DATA/OBJECT ANTI-SYMMETRY**

Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions.

This exposes the fundamental dichotomy between objects and data structures:

Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.

The complement is also true:

Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.

Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.

### **THE LAW OF DEMETER**

There is a well-known heuristic called the *Law of Demeter*2 that says a module should not know about the innards of the *objects* it manipulates.

More precisely, the Law of Demeter says that a method f of a class C should only call the methods of these:

• C

• An object created by f

• An object passed as an argument to f

• An object held in an instance variable of C

#### **Train Wrecks**

This kind of code is often called a train wreck because it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided [G36]. It is usually best to split them up as follows:

   Options opts = ctxt.getOptions();  
   File scratchDir = opts.getScratchDir();  
   final String outputDir = scratchDir.getAbsolutePath();

Whether this is a violation of Demeter depends on whether or not ctxt, Options, and ScratchDirare objects or data structures. If they are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law of Demeter. On the other hand, if ctxt, Options, and ScratchDir are just data structures with no behavior, then they naturally expose their internal structure, and so Demeter does not apply.

The use of accessor functions confuses the issue. If the code had been written as follows, then we probably wouldn’t be asking about Demeter violations.

   final String outputDir = ctxt.options.scratchDir.absolutePath;

#### **Hybrids**

This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure.

#### **Hiding Structure**

What if ctxt, options, and scratchDir are objects with real behavior? Then, because objects are supposed to hide their internal structure, we should not be able to navigate through them.

   ctxt.getAbsolutePathOfScratchDirectoryOption();

### **DATA TRANSFER OBJECTS**

#### **Active Record**

Active Records are special forms of DTOs. They are data structures with public (or bean-accessed) variables; but they typically have navigational methods like save and find.

Unfortunately we often find that developers try to treat these data structures as though they were objects by putting business rule methods in them.

The solution, of course, is to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instances of the Active Record).

 Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.

## ****Error Handling****

### **USE EXCEPTIONS RATHER THAN RETURN CODES**

The problem with these approaches is that they clutter the caller. The caller must check for errors immediately after the call. Unfortunately, it’s easy to forget. For this reason it is better to throw an exception when you encounter an error. The calling code is cleaner. Its logic is not obscured by error handling.

### **WRITE YOUR** TRY-CATCH-FINALLY **STATEMENT FIRST**

In a way, try blocks are like transactions. Your catch has to leave your program in a consistent state, no matter what happens in the try. For this reason it is good practice to start with a try-catch-finally statement when you are writing code that could throw exceptions. This helps you define what the user of that code should expect, no matter what goes wrong with the code that is executed in the try.

### **USE UNCHECKED EXCEPTIONS**

### **PROVIDE CONTEXT WITH EXCEPTIONS**

Each exception that you throw should provide enough context to determine the source and location of an error. In Java, you can get a stack trace from any exception; however, a stack trace can’t tell you the intent of the operation that failed.

Create informative error messages and pass them along with your exceptions. Mention the operation that failed and the type of failure. If you are logging in your application, pass along enough information to be able to log the error in your catch.

### **DEFINE EXCEPTION CLASSES IN TERMS OF A CALLER’S NEEDS**

There are many ways to classify errors. We can classify them by their source: Did they come from one component or another? Or their type: Are they device failures, network failures, or programming errors? However, when we define exception classes in an application, our most important concern should be how they are caught.

### **DEFINE THE NORMAL FLOW**

 You wrap external APIs so that you can throw your own exceptions, and you define a handler above your code so that you can deal with any aborted computation. Most of the time this is a great approach, but there are some times when you may not want to abort. Let’s take a look at an example. Here is some awkward code that sums expenses in a billing application:

   try {  
     MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
     m\_total += expenses.getTotal();  
   } catch(MealExpensesNotFound e) {  
     m\_total += getMealPerDiem();  
   }

In this business, if meals are expensed, they become part of the total. If they aren’t, the employee gets a meal per diem amount for that day. The exception clutters the logic. Wouldn’t it be better if we didn’t have to deal with the special case? If we didn’t, our code would look much simpler. It would look like this:

   MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());  
   m\_total += expenses.getTotal();

Can we make the code that simple? It turns out that we can. We can change the ExpenseReportDAOso that it always returns a MealExpense object. If there are no meal expenses, it returns a MealExpense object that returns the per diem as its total:

   public class PerDiemMealExpenses implements MealExpenses {  
     public int getTotal() {  
       // return the per diem default  
     }  
   }

This is called the SPECIAL CASE PATTERN [Fowler]. You create a class or configure an object so that it handles a special case for you. When you do, the client code doesn’t have to deal with exceptional behavior. That behavior is encapsulated in the special case object.

### **DON’T RETURN NULL**

Here is some example code:

   public void registerItem(Item item) {  
     if (item != null) {  
       ItemRegistry registry = peristentStore.getItemRegistry();  
       if (registry != null) {  
         Item existing = registry.getItem(item.getID());  
         if (existing.getBillingPeriod().hasRetailOwner()) {  
           existing.register(item);  
         }  
       }  
     }  
   }

It’s easy to say that the problem with the code above is that it is missing a null check, but in actuality, the problem is that it has too many. If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead. If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

### **DON’T PASS NULL**

Returning null from methods is bad, but passing null into methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.

In most programming languages there is no good way to deal with a null that is passed by a caller accidentally. Because this is the case, the rational approach is to forbid passing null by default. When you do, you can code with the knowledge that a null in an argument list is an indication of a problem, and end up with far fewer careless mistakes.

## ****Boundaries****

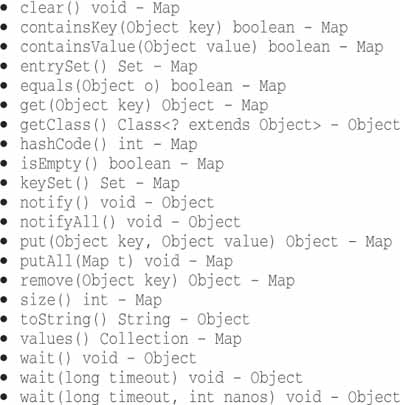
We seldom control all the software in our systems. Sometimes we buy third-party packages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code with our own. In this chapter we look at practices and techniques to keep the boundaries of our software clean.

### **USING THIRD-PARTY CODE**

There is a natural tension between the provider of an interface and the user of an interface. Providers of third-party packages and frameworks strive for broad applicability so they can work in many environments and appeal to a wide audience. Users, on the other hand, want an interface that is focused on their particular needs. This tension can cause problems at the boundaries of our systems.

Let’s look at java.util.Map as an example. As you can see by examining [Figure 8-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter08.html#ch8fig1), Maps have a very broad interface with plenty of capabilities. Certainly this power and flexibility is useful, but it can also be a liability. For instance, our application might build up a Map and pass it around. Our intention might be that none of the recipients of our Map delete anything in the map. But right there at the top of the list is the clear() method. Any user of the Map has the power to clear it. Or maybe our design convention is that only particular types of objects can be stored in the Map, but Maps do not reliably constrain the types of objects placed within them. Any determined user can add items of any type to any Map.

**Figure 8-1** The methods of Map



If our application needs a Map of Sensors, you might find the sensors set up like this:

   Map sensors = new HashMap();

Then, when some other part of the code needs to access the sensor, you see this code:

   Sensor s = (Sensor)sensors.get(sensorId );

We don’t just see it once, but over and over again throughout the code. The client of this code carries the responsibility of getting an Object from the Map and casting it to the right type. This works, but it’s not clean code. Also, this code does not tell its story as well as it could. The readability of this code can be greatly improved by using generics, as shown below:

       Map<Sensor> sensors = new HashMap<Sensor>();  
   …  
       Sensor s = sensors.get(sensorId );

However, this doesn’t solve the problem that Map<Sensor> provides more capability than we need or want.

Passing an instance of Map<Sensor> liberally around the system means that there will be a lot of places to fix if the interface to Map ever changes. You might think such a change to be unlikely, but remember that it changed when generics support was added in Java 5. Indeed, we’ve seen systems that are inhibited from using generics because of the sheer magnitude of changes needed to make up for the liberal use of Maps.

A cleaner way to use Map might look like the following. No user of Sensors would care one bit if generics were used or not. That choice has become (and always should be) an implementation detail.

   public class Sensors {  
     private Map sensors = new HashMap();  
  
     public Sensor getById(String id) {  
       return (Sensor) sensors.get(id);  
     }  
  
     //snip  
   }

The interface at the boundary (Map) is hidden. It is able to evolve with very little impact on the rest of the application. The use of generics is no longer a big issue because the casting and type management is handled inside the Sensors class.

This interface is also tailored and constrained to meet the needs of the application. It results in code that is easier to understand and harder to misuse. The Sensors class can enforce design and business rules.

We are not suggesting that every use of Map be encapsulated in this form. Rather, we are advising you not to pass Maps (or any other interface at a boundary) around your system. If you use a boundary interface like Map, keep it inside the class, or close family of classes, where it is used. Avoid returning it from, or accepting it as an argument to, public APIs.

### **EXPLORING AND LEARNING BOUNDARIES**

Third-party code helps us get more functionality delivered in less time. Where do we start when we want to utilize some third-party package? It’s not our job to test the third-party code, but it may be in our best interest to write tests for the third-party code we use.

Suppose it is not clear how to use our third-party library. We might spend a day or two (or more) reading the documentation and deciding how we are going to use it. Then we might write our code to use the third-party code and see whether it does what we think. We would not be surprised to find ourselves bogged down in long debugging sessions trying to figure out whether the bugs we are experiencing are in our code or theirs.

Learning the third-party code is hard. Integrating the third-party code is hard too. Doing both at the same time is doubly hard. What if we took a different approach? Instead of experimenting and trying out the new stuff in our production code, we could write some tests to explore our understanding of the third-party code. Jim Newkirk calls such tests learning tests.1

1. [[BeckTDD](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter08.html#BeckTDD)], pp. [136](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter10.html#page_136)–137.

In learning tests we call the third-party API, as we expect to use it in our application. We’re essentially doing controlled experiments that check our understanding of that API. The tests focus on what we want out of the API.

### **LEARNING** LOG4J

Let’s say we want to use the apache log4j package rather than our own custom-built logger. We download it and open the introductory documentation page. Without too much reading we write our first test case, expecting it to write “hello” to the console.

   @Test  
   public void testLogCreate() {  
     Logger logger = Logger.getLogger(“MyLogger”);  
     logger.info(“hello”);  
   }

When we run it, the logger produces an error that tells us we need something called an Appender. After a little more reading we find that there is a ConsoleAppender. So we create a ConsoleAppender and see whether we have unlocked the secrets of logging to the console.

   @Test  
   public void testLogAddAppender() {  
     Logger logger = Logger.getLogger(“MyLogger”);  
     ConsoleAppender appender = new ConsoleAppender();  
     logger.addAppender(appender);  
     logger.info(“hello”);  
   }

This time we find that the Appender has no output stream. Odd—it seems logical that it’d have one. After a little help from Google, we try the following:

   @Test  
   public void testLogAddAppender() {  
     Logger logger = Logger.getLogger(“MyLogger”);  
     logger.removeAllAppenders();  
     logger.addAppender(new ConsoleAppender(  
          new PatternLayout(“%p %t %m%n”),  
          ConsoleAppender.SYSTEM\_OUT));  
     logger.info(“hello”);  
   }

That worked; a log message that includes “hello” came out on the console! It seems odd that we have to tell the ConsoleAppender that it writes to the console.

Interestingly enough, when we remove the ConsoleAppender.SystemOut argument, we see that “hello” is still printed. But when we take out the PatternLayout, it once again complains about the lack of an output stream. This is very strange behavior.

Looking a little more carefully at the documentation, we see that the default ConsoleAppenderconstructor is “unconfigured,” which does not seem too obvious or useful. This feels like a bug, or at least an inconsistency, in log4j.

A bit more googling, reading, and testing, and we eventually wind up with [Listing 8-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter08.html#ch8lt1). We’ve discovered a great deal about the way that log4j works, and we’ve encoded that knowledge into a set of simple unit tests.

**Listing 8-1 LogTest.java**

   public class LogTest {  
       private Logger logger;  
         
       @Before  
       public void initialize() {  
           logger = Logger.getLogger(“logger”);  
           logger.removeAllAppenders();  
           Logger.getRootLogger().removeAllAppenders();  
       }  
       @Test  
       public void basicLogger() {  
           BasicConfigurator.configure();  
           logger.info(“basicLogger”);  
       }  
       @Test  
       public void addAppenderWithStream() {  
       logger.addAppender(new ConsoleAppender(  
           new PatternLayout(“%p %t %m%n”),  
           ConsoleAppender.SYSTEM\_OUT));  
          logger.info(“addAppenderWithStream”);  
   }  
          @Test  
          public void addAppenderWithoutStream() {  
           logger.addAppender(new ConsoleAppender(  
                new PatternLayout(“%p %t %m%n”)));  
              logger.info(“addAppenderWithoutStream”);  
           }  
   }

Now we know how to get a simple console logger initialized, and we can encapsulate that knowledge into our own logger class so that the rest of our application is isolated from the log4j boundary interface.

### **LEARNING TESTS ARE BETTER THAN FREE**

The learning tests end up costing nothing. We had to learn the API anyway, and writing those tests was an easy and isolated way to get that knowledge. The learning tests were precise experiments that helped increase our understanding.

Not only are learning tests free, they have a positive return on investment. When there are new releases of the third-party package, we run the learning tests to see whether there are behavioral differences.

Learning tests verify that the third-party packages we are using work the way we expect them to. Once integrated, there are no guarantees that the third-party code will stay compatible with our needs. The original authors will have pressures to change their code to meet new needs of their own. They will fix bugs and add new capabilities. With each release comes new risk. If the third-party package changes in some way incompatible with our tests, we will find out right away.

Whether you need the learning provided by the learning tests or not, a clean boundary should be supported by a set of outbound tests that exercise the interface the same way the production code does. Without these boundary tests to ease the migration, we might be tempted to stay with the old version longer than we should.

### **USING CODE THAT DOES NOT YET EXIST**

There is another kind of boundary, one that separates the known from the unknown. There are often places in the code where our knowledge seems to drop off the edge. Sometimes what is on the other side of the boundary is unknowable (at least right now). Sometimes we choose to look no farther than the boundary.

A number of years back I was part of a team developing software for a radio communications system. There was a subsystem, the “Transmitter,” that we knew little about, and the people responsible for the subsystem had not gotten to the point of defining their interface. We did not want to be blocked, so we started our work far away from the unknown part of the code.

We had a pretty good idea of where our world ended and the new world began. As we worked, we sometimes bumped up against this boundary. Though mists and clouds of ignorance obscured our view beyond the boundary, our work made us aware of what we wanted the boundary interface to be. We wanted to tell the transmitter something like this:

Key the transmitter on the provided frequency and emit an analog representation of the data coming from this stream.

We had no idea how that would be done because the API had not been designed yet. So we decided to work out the details later.

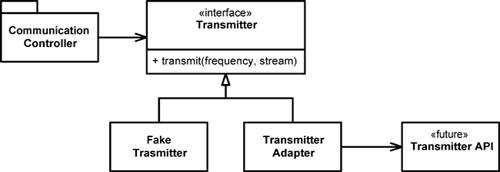
To keep from being blocked, we defined our own interface. We called it something catchy, like Transmitter. We gave it a method called transmit that took a frequency and a data stream. This was the interface we wished we had.

One good thing about writing the interface we wish we had is that it’s under our control. This helps keep client code more readable and focused on what it is trying to accomplish.

In [Figure 8-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter08.html#ch8fig2), you can see that we insulated the CommunicationsController classes from the transmitter API (which was out of our control and undefined). By using our own application specific interface, we kept our CommunicationsController code clean and expressive. Once the transmitter API was defined, we wrote the TransmitterAdapter to bridge the gap. The ADAPTER2 encapsulated the interaction with the API and provides a single place to change when the API evolves.

2. See the Adapter pattern in [[GOF](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter08.html#GOF)].

**Figure 8-2** Predicting the transmitter



This design also gives us a very convenient seam3 in the code for testing. Using a suitable FakeTransmitter, we can test the CommunicationsController classes. We can also create boundary tests once we have the TransmitterAPI that make sure we are using the API correctly.

3. See more about seams in [[WELC](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter08.html#WELC)].

### **CLEAN BOUNDARIES**

Interesting things happen at boundaries. Change is one of those things. Good software designs accommodate change without huge investments and rework. When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly.

Code at the boundaries needs clear separation and tests that define expectations. We should avoid letting too much of our code know about the third-party particulars. It’s better to depend on something you control than on something you don’t control, lest it end up controlling you.

We manage third-party boundaries by having very few places in the code that refer to them. We may wrap them as we did with Map, or we may use an ADAPTER to convert from our perfect interface to the provided interface. Either way our code speaks to us better, promotes internally consistent usage across the boundary, and has fewer maintenance points when the third-party code changes.

## ****Unit Tests****

### **THE THREE LAWS OF TDD**

By now everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg. Consider the following three laws:1

**First Law** You may not write production code until you have written a failing unit test.

**Second Law** You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

**Third Law** You may not write more production code than is sufficient to pass the currently failing test.

These three laws lock you into a cycle that is perhaps thirty seconds long. The tests and the production code are written together, with the tests just a few seconds ahead of the production code.

If we work this way, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. If we work this way, those tests will cover virtually all of our production code. The sheer bulk of those tests, which can rival the size of the production code itself, can present a daunting management problem.

### **KEEPING TESTS CLEAN**

Having dirty tests is equivalent to, if not worse than, having no tests. The problem is that tests must change as the production code evolves.

The moral of the story is simple: Test code is just as important as production code. It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code.

#### **Tests Enable the -ilities**

Without tests every change is a possible bug. The higher your test coverage, the less your fear.

### **CLEAN TESTS**

What makes a clean test? Readability is perhaps even more important in unit tests than it is in production code. What makes tests readable? The same thing that makes all code readable: clarity, simplicity, and density of expression. In a test you want to say a lot with as few expressions as possible.

The BUILD-OPERATE-CHECK2 pattern is made obvious by the structure of these tests. Each of the tests is clearly split into three parts. The first part builds up the test data, the second part operates on that test data, and the third part checks that the operation yielded the expected results.

#### **Domain-Specific Testing Language**

Rather than using the APIs that programmers use to manipulate the system, we build up a set of functions and utilities that make use of those APIs and that make the tests more convenient to write and easier to read. These functions and utilities become a specialized API used by the tests. They are a testing language that programmers use to help themselves to write their tests and to help those who must read those tests later on.

#### **A Dual Standard**

The code within the testing API does have a different set of engineering standards than production code. It must still be simple, succinct, and expressive, but it need not be as efficient as production code. After all, it runs in a test environment, not a production environment, and those two environment have very different needs.

### **ONE ASSERT PER TEST**

There is a school of thought4 that says that every test function in a JUnit test should have one and only one assert statement.

We can eliminate the duplication by using the TEMPLATE METHOD6 pattern and putting the given/when parts in the base class, and the then parts in different derivatives. Or we could create a completely separate test class and put the given and when parts in the @Before function, and the when parts in each @Test function. But this seems like too much mechanism for such a minor issue. In the end, I prefer the multiple asserts in [Listing 9-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter09.html#ch9lt2).

#### **Single Concept per Test**

Perhaps a better rule is that we want to test a single concept in each test function.

**Independent** Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

**Repeatable** Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren’t repeatable in any environment, then you’ll always have an excuse for why they fail. You’ll also find yourself unable to run the tests when the environment isn’t available.

**Self-Validating** The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren’t self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

**Timely** The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.

## ****Classes****

### **CLASS ORGANIZATION**

#### **Encapsulation**

### **CLASSES SHOULD BE SMALL!**

No, we’re not going to repeat the exact same text from the Functions chapter. But as with functions, smaller is the primary rule when it comes to designing classes.

With functions we measured size by counting physical lines. With classes we use a different measure. We count responsibilities.

The name of a class should describe what responsibilities it fulfills. In fact, naming is probably the first way of helping determine class size. If we cannot derive a concise name for a class, then it’s likely too large. The more ambiguous the class name, the more likely it has too many responsibilities.

We should also be able to write a brief description of the class in about 25 words, without using the words “if,” “and,” “or,” or “but.”

#### **The Single Responsibility Principle**

The Single Responsibility Principle (SRP)2 states that a class or module should have one, and only one, reason to change. This principle gives us both a definition of responsibility, and a guidelines for class size. Classes should have one responsibility—one reason to change.

We want our systems to be composed of many small classes, not a few large ones. Each small class encapsulates a single responsibility, has a single reason to change, and collaborates with a few others to achieve the desired system behaviors.

#### **Cohesion**

Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables.

The strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive.

#### **Maintaining Cohesion Results in Many Small Classes**

So breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well. This gives our program a much better organization and a more transparent structure.

### **ORGANIZING FOR CHANGE**

## ****Systems****

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

—Ray Ozzie, CTO, Microsoft Corporation

### **HOW WOULD YOU BUILD A CITY?**

Could you manage all the details yourself? Probably not. Even managing an existing city is too much for one person. Yet, cities work (most of the time). They work because cities have teams of people who manage particular parts of the city, the water systems, power systems, traffic, law enforcement, building codes, and so forth. Some of those people are responsible for the big picture, while others focus on the details.

Cities also work because they have evolved appropriate levels of abstraction and modularity that make it possible for individuals and the “components” they manage to work effectively, even without understanding the big picture.

Although software teams are often organized like that too, the systems they work on often don’t have the same separation of concerns and levels of abstraction. Clean code helps us achieve this at the lower levels of abstraction. In this chapter let us consider how to stay clean at higher levels of abstraction, the system level.

### **SEPARATE CONSTRUCTING A SYSTEM FROM USING IT**

First, consider that construction is a very different process from use. As I write this, there is a new hotel under construction that I see out my window in Chicago. Today it is a bare concrete box with a construction crane and elevator bolted to the outside. The busy people there all wear hard hats and work clothes. In a year or so the hotel will be finished. The crane and elevator will be gone. The building will be clean, encased in glass window walls and attractive paint. The people working and staying there will look a lot different too.

Software systems should separate the startup process, when the application objects are constructed and the dependencies are “wired” together, from the runtime logic that takes over after startup.

The startup process is a concern that any application must address. It is the first concern that we will examine in this chapter. The separation of concerns is one of the oldest and most important design techniques in our craft.

Unfortunately, most applications don’t separate this concern. The code for the startup process is ad hoc and it is mixed in with the runtime logic. Here is a typical example:

   public Service getService() {  
     if (service == null)  
       service = new MyServiceImpl(…); // Good enough default for most cases?  
     return service;  
   }

This is the LAZY INITIALIZATION/EVALUATION idiom, and it has several merits. We don’t incur the overhead of construction unless we actually use the object, and our startup times can be faster as a result. We also ensure that null is never returned.

However, we now have a hard-coded dependency on MyServiceImpl and everything its constructor requires (which I have elided). We can’t compile without resolving these dependencies, even if we never actually use an object of this type at runtime!

Testing can be a problem. If MyServiceImpl is a heavyweight object, we will need to make sure that an appropriate TEST DOUBLE1 or MOCK OBJECT gets assigned to the service field before this method is called during unit testing. Because we have construction logic mixed in with normal runtime processing, we should test all execution paths (for example, the null test and its block). Having both of these responsibilities means that the method is doing more than one thing, so we are breaking the Single Responsibility Principle in a small way.

1. [[Mezzaros07](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Mezzaros07)].

Perhaps worst of all, we do not know whether MyServiceImpl is the right object in all cases. I implied as much in the comment. Why does the class with this method have to know the global context? Can we ever really know the right object to use here? Is it even possible for one type to be right for all possible contexts?

One occurrence of LAZY-INITIALIZATION isn’t a serious problem, of course. However, there are normally many instances of little setup idioms like this in applications. Hence, the global setup strategy (if there is one) is scattered across the application, with little modularity and often significant duplication.

If we are diligent about building well-formed and robust systems, we should never let little, convenient idioms lead to modularity breakdown. The startup process of object construction and wiring is no exception. We should modularize this process separately from the normal runtime logic and we should make sure that we have a global, consistent strategy for resolving our major dependencies.

#### **Separation of Main**

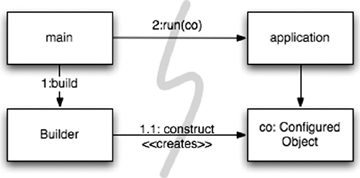
One way to separate construction from use is simply to move all aspects of construction to main, or modules called by main, and to design the rest of the system assuming that all objects have been constructed and wired up appropriately. (See [Figure 11-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11fig1).)

The flow of control is easy to follow. The main function builds the objects necessary for the system, then passes them to the application, which simply uses them. Notice the direction of the dependency arrows crossing the barrier between main and the application. They all go one direction, pointing away from main. This means that the application has no knowledge of main or of the construction process. It simply expects that everything has been built properly.

#### **Factories**

Sometimes, of course, we need to make the application responsible for when an object gets created. For example, in an order processing system the application must create the

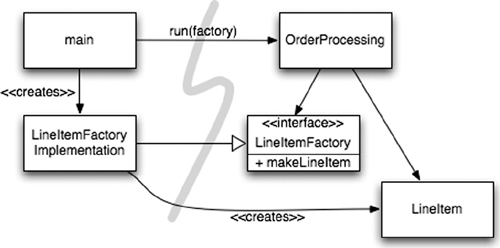
**Figure 11-1** Separating construction in main()



LineItem instances to add to an Order. In this case we can use the ABSTRACT FACTORY2 pattern to give the application control of when to build the LineItems, but keep the details of that construction separate from the application code. (See [Figure 11-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11fig2).)

2. [[GOF](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#GOF)].

**Figure 11-2** Separation construction with factory



Again notice that all the dependencies point from main toward the OrderProcessing application. This means that the application is decoupled from the details of how to build a LineItem. That capability is held in the LineItemFactoryImplementation, which is on the main side of the line. And yet the application is in complete control of when the LineItem instances get built and can even provide application-specific constructor arguments.

#### **Dependency Injection**

A powerful mechanism for separating construction from use is Dependency Injection (DI), the application of Inversion of Control (IoC) to dependency management.3 Inversion of Control moves secondary responsibilities from an object to other objects that are dedicated to the purpose, thereby supporting the Single Responsibility Principle. In the context of dependency management, an object should not take responsibility for instantiating dependencies itself. Instead, it should pass this responsibility to another “authoritative” mechanism, thereby inverting the control. Because setup is a global concern, this authoritative mechanism will usually be either the “main” routine or a special-purpose container.

3. See, for example, [[Fowler](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Fowler)].

JNDI lookups are a “partial” implementation of DI, where an object asks a directory server to provide a “service” matching a particular name.

   MyService myService = (MyService)(jndiContext.lookup(“NameOfMyService”));

The invoking object doesn’t control what kind of object is actually returned (as long it implements the appropriate interface, of course), but the invoking object still actively resolves the dependency.

True Dependency Injection goes one step further. The class takes no direct steps to resolve its dependencies; it is completely passive. Instead, it provides setter methods or constructor arguments (or both) that are used to inject the dependencies. During the construction process, the DI container instantiates the required objects (usually on demand) and uses the constructor arguments or setter methods provided to wire together the dependencies. Which dependent objects are actually used is specified through a configuration file or programmatically in a special-purpose construction module.

The Spring Framework provides the best known DI container for Java.4 You define which objects to wire together in an XML configuration file, then you ask for particular objects by name in Java code. We will look at an example shortly.

4. See [[Spring](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Spring)]. There is also a Spring.NET framework.

But what about the virtues of LAZY-INITIALIZATION? This idiom is still sometimes useful with DI. First, most DI containers won’t construct an object until needed. Second, many of these containers provide mechanisms for invoking factories or for constructing proxies, which could be used for LAZY-EVALUATIONand similar optimizations.5

5. Don’t forget that lazy instantiation/evaluation is just an optimization and perhaps premature!

### **SCALING UP**

Cities grow from towns, which grow from settlements. At first the roads are narrow and practically nonexistent, then they are paved, then widened over time. Small buildings and empty plots are filled with larger buildings, some of which will eventually be replaced with skyscrapers.

At first there are no services like power, water, sewage, and the Internet (gasp!). These services are also added as the population and building densities increase.

This growth is not without pain. How many times have you driven, bumper to bumper through a road “improvement” project and asked yourself, “Why didn’t they build it wide enough the first time!?”

But it couldn’t have happened any other way. Who can justify the expense of a six-lane highway through the middle of a small town that anticipates growth? Who would want such a road through their town?

It is a myth that we can get systems “right the first time.” Instead, we should implement only today’s stories, then refactor and expand the system to implement new stories tomorrow. This is the essence of iterative and incremental agility. Test-driven development, refactoring, and the clean code they produce make this work at the code level.

But what about at the system level? Doesn’t the system architecture require preplanning? Certainly, it can’t grow incrementally from simple to complex, can it?

Software systems are unique compared to physical systems. Their architectures can grow incrementally, **if**we maintain the proper separation of concerns.

The ephemeral nature of software systems makes this possible, as we will see. Let us first consider a counterexample of an architecture that doesn’t separate concerns adequately.

The original EJB1 and EJB2 architectures did not separate concerns appropriately and thereby imposed unnecessary barriers to organic growth. Consider an Entity Bean for a persistent Bank class. An entity bean is an in-memory representation of relational data, in other words, a table row.

First, you had to define a local (in process) or remote (separate JVM) interface, which clients would use. [Listing 11-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11lt1) shows a possible local interface:

**Listing 11-1 An EJB2 local interface for a Bank EJB**

   package com.example.banking;  
   import java.util.Collections;  
   import javax.ejb.\*;  
     
   public interface BankLocal extends java.ejb.EJBLocalObject {  
     String getStreetAddr1() throws EJBException;  
     String getStreetAddr2() throws EJBException;  
     String getCity() throws EJBException;  
     String getState() throws EJBException;  
     String getZipCode() throws EJBException;  
     void setStreetAddr1(String street1) throws EJBException;  
     void setStreetAddr2(String street2) throws EJBException;  
     void setCity(String city) throws EJBException;  
     void setState(String state) throws EJBException;  
     void setZipCode(String zip) throws EJBException;  
     Collection getAccounts() throws EJBException;  
     void setAccounts(Collection accounts) throws EJBException;  
     void addAccount(AccountDTO accountDTO) throws EJBException;  
   }

I have shown several attributes for the Bank’s address and a collection of accounts that the bank owns, each of which would have its data handled by a separate Account EJB. [Listing 11-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11lt2) shows the corresponding implementation class for the Bank bean.

**Listing 11-2 The corresponding EJB2 Entity Bean Implementation**

   package com.example.banking;  
   import java.util.Collections;  
   import javax.ejb.\*;  
     
   public abstract class Bank implements javax.ejb.EntityBean {  
     // Business logic…  
     public abstract String getStreetAddr1();  
     public abstract String getStreetAddr2();  
     public abstract String getCity();  
     public abstract String getState();  
     public abstract String getZipCode();  
     public abstract void setStreetAddr1(String street1);  
     public abstract void setStreetAddr2(String street2);  
     public abstract void setCity(String city);  
     public abstract void setState(String state);  
     public abstract void setZipCode(String zip);  
     public abstract Collection getAccounts();  
     public abstract void setAccounts(Collection accounts);  
     public void addAccount(AccountDTO accountDTO) {  
       InitialContext context = new InitialContext();  
       AccountHomeLocal accountHome = context.lookup(”AccountHomeLocal”);  
       AccountLocal account = accountHome.create(accountDTO);  
       Collection accounts = getAccounts();  
       accounts.add(account);  
     }  
     // EJB container logic  
     public abstract void setId(Integer id);  
     public abstract Integer getId();  
     public Integer ejbCreate(Integer id) { … }  
     public void ejbPostCreate(Integer id) { … }  
     // The rest had to be implemented but were usually empty:  
     public void setEntityContext(EntityContext ctx) {}   
     public void unsetEntityContext() {}  
     public void ejbActivate() {}  
     public void ejbPassivate() {}  
     public void ejbLoad() {}  
     public void ejbStore() {}  
     public void ejbRemove() {}  
   }

I haven’t shown the corresponding LocalHome interface, essentially a factory used to create objects, nor any of the possible Bank finder (query) methods you might add.

Finally, you had to write one or more XML deployment descriptors that specify the object-relational mapping details to a persistence store, the desired transactional behavior, security constraints, and so on.

The business logic is tightly coupled to the EJB2 application “container.” You must subclass container types and you must provide many lifecycle methods that are required by the container.

Because of this coupling to the heavyweight container, isolated unit testing is difficult. It is necessary to mock out the container, which is hard, or waste a lot of time deploying EJBs and tests to a real server. Reuse outside of the EJB2 architecture is effectively impossible, due to the tight coupling.

Finally, even object-oriented programming is undermined. One bean cannot inherit from another bean. Notice the logic for adding a new account. It is common in EJB2 beans to define “data transfer objects” (DTOs) that are essentially “structs” with no behavior. This usually leads to redundant types holding essentially the same data, and it requires boilerplate code to copy data from one object to another.

#### **Cross-Cutting Concerns**

The EJB2 architecture comes close to true separation of concerns in some areas. For example, the desired transactional, security, and some of the persistence behaviors are declared in the deployment descriptors, independently of the source code.

Note that concerns like persistence tend to cut across the natural object boundaries of a domain. You want to persist all your objects using generally the same strategy, for example, using a particular DBMS6 versus flat files, following certain naming conventions for tables and columns, using consistent transactional semantics, and so on.

6. Database management system.

In principle, you can reason about your persistence strategy in a modular, encapsulated way. Yet, in practice, you have to spread essentially the same code that implements the persistence strategy across many objects. We use the term cross-cutting concerns for concerns like these. Again, the persistence framework might be modular and our domain logic, in isolation, might be modular. The problem is the fine-grained intersection of these domains.

In fact, the way the EJB architecture handled persistence, security, and transactions, “anticipated” aspect-oriented programming (AOP),7 which is a general-purpose approach to restoring modularity for cross-cutting concerns.

7. See [[AOSD](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#AOSD)] for general information on aspects and [[AspectJ](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#AspectJ)]] and [[Colyer](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Colyer)] for AspectJ-specific information.

In AOP, modular constructs called aspects specify which points in the system should have their behavior modified in some consistent way to support a particular concern. This specification is done using a succinct declarative or programmatic mechanism.

Using persistence as an example, you would declare which objects and attributes (or patterns thereof) should be persisted and then delegate the persistence tasks to your persistence framework. The behavior modifications are made noninvasively8 to the target code by the AOP framework. Let us look at three aspects or aspect-like mechanisms in Java.

8. Meaning no manual editing of the target source code is required.

### **JAVA PROXIES**

Java proxies are suitable for simple situations, such as wrapping method calls in individual objects or classes. However, the dynamic proxies provided in the JDK only work with interfaces. To proxy classes, you have to use a byte-code manipulation library, such as CGLIB, ASM, or Javassist.9

9. See [[CGLIB](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#CGLIB)], [[ASM](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ASM)], and [[Javassist](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Javassist)].

[Listing 11-3](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11lt3) shows the skeleton for a JDK proxy to provide persistence support for our Bankapplication, covering only the methods for getting and setting the list of accounts.

**Listing 11-3 JDK Proxy Example**

   // Bank.java (suppressing package names…)  
   import java.utils.\*;  
     
   // The abstraction of a bank.  
   public interface Bank {  
     Collection<Account> getAccounts();  
     void setAccounts(Collection<Account> accounts);  
   }  
   // BankImpl.java  
   import java.utils.\*;  
  
   // The “Plain Old Java Object” (POJO) implementing the abstraction.  
   public class BankImpl implements Bank {  
     private List<Account> accounts;  
  
     public Collection<Account> getAccounts() {   
       return accounts;   
     }  
     public void setAccounts(Collection<Account> accounts) {   
       this.accounts = new ArrayList<Account>();   
       for (Account account: accounts) {  
         this.accounts.add(account);  
       }  
     }  
   }  
   // BankProxyHandler.java  
   import java.lang.reflect.\*;  
   import java.util.\*;  
   // “InvocationHandler” required by the proxy API.  
   public class BankProxyHandler implements InvocationHandler {  
     private Bank bank;  
       
     public BankHandler (Bank bank) {  
       this.bank = bank;  
     }  
     // Method defined in InvocationHandler  
     public Object invoke(Object proxy, Method method, Object[] args)   
         throws Throwable {  
     String methodName = method.getName();  
     if (methodName.equals(”getAccounts”)) {  
       bank.setAccounts(getAccountsFromDatabase());  
       return bank.getAccounts();  
     } else if (methodName.equals(”setAccounts”)) {  
       bank.setAccounts((Collection<Account>) args[0]);  
       setAccountsToDatabase(bank.getAccounts());  
       return null;  
     } else {  
       …  
     }  
   }  
   // Lots of details here:  
   protected Collection<Account> getAccountsFromDatabase() { … }  
   protected void setAccountsToDatabase(Collection<Account> accounts) { … }  
   }  
  
   // Somewhere else…  
  
   Bank bank = (Bank) Proxy.newProxyInstance(  
     Bank.class.getClassLoader(),   
     new Class[] { Bank.class },  
     new BankProxyHandler(new BankImpl()));

We defined an interface Bank, which will be wrapped by the proxy, and a Plain-Old Java Object (POJO), BankImpl, that implements the business logic. (We will revisit POJOs shortly.)

The Proxy API requires an InvocationHandler object that it calls to implement any Bank method calls made to the proxy. Our BankProxyHandler uses the Java reflection API to map the generic method invocations to the corresponding methods in BankImpl, and so on.

There is a lot of code here and it is relatively complicated, even for this simple case.10 Using one of the byte-manipulation libraries is similarly challenging. This code “volume”

10. For more detailed examples of the Proxy API and examples of its use, see, for example, [[Goetz](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Goetz)].

and complexity are two of the drawbacks of proxies. They make it hard to create clean code! Also, proxies don’t provide a mechanism for specifying system-wide execution “points” of interest, which is needed for a true AOP solution.11

11. AOP is sometimes confused with techniques used to implement it, such as method interception and “wrapping” through proxies. The real value of an AOP system is the ability to specify systemic behaviors in a concise and modular way.

### **PURE JAVA AOP FRAMEWORKS**

Fortunately, most of the proxy boilerplate can be handled automatically by tools. Proxies are used internally in several Java frameworks, for example, Spring AOP and JBoss AOP, to implement aspects in pure Java.12 In Spring, you write your business logic as Plain-Old Java Objects. POJOs are purely focused on their domain. They have no dependencies on enterprise frameworks (or any other domains). Hence, they are conceptually simpler and easier to test drive. The relative simplicity makes it easier to ensure that you are implementing the corresponding user stories correctly and to maintain and evolve the code for future stories.

12. See [[Spring](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Spring)] and [[JBoss](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#JBoss)]. “Pure Java” means without the use of AspectJ.

You incorporate the required application infrastructure, including cross-cutting concerns like persistence, transactions, security, caching, failover, and so on, using declarative configuration files or APIs. In many cases, you are actually specifying Spring or JBoss library aspects, where the framework handles the mechanics of using Java proxies or byte-code libraries transparently to the user. These declarations drive the dependency injection (DI) container, which instantiates the major objects and wires them together on demand.

[Listing 11-4](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11lt4) shows a typical fragment of a Spring V2.5 configuration file, app.xml13:

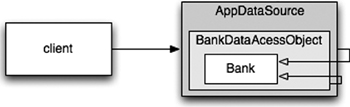
13. Adapted from [http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25](http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring).

**Listing 11-4 Spring 2.X configuration file**

   <beans>  
     …  
     <bean id=”appDataSource”  
     class=”org.apache.commons.dbcp.BasicDataSource”  
     destroy-method=”close”  
     p:driverClassName=”com.mysql.jdbc.Driver”  
     p:url=”jdbc:mysql://localhost:3306/mydb”  
     p:username=”me”/>  
  
     <bean id=”bankDataAccessObject”  
     class=”com.example.banking.persistence.BankDataAccessObject”  
     p:dataSource-ref=”appDataSource”/>  
  
     <bean id=”bank”  
   class=”com.example.banking.model.Bank”  
   p:dataAccessObject-ref=”bankDataAccessObject”/>  
   …  
</beans>

Each “bean” is like one part of a nested “Russian doll,” with a domain object for a Bank proxied (wrapped) by a data accessor object (DAO), which is itself proxied by a JDBC driver data source. (See [Figure 11-3](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11fig3).)

**Figure 11-3** The “Russian doll” of decorators



The client believes it is invoking getAccounts() on a Bank object, but it is actually talking to the outermost of a set of nested DECORATOR14 objects that extend the basic behavior of the Bank POJO. We could add other decorators for transactions, caching, and so forth.

14. [[GOF](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#GOF)].

In the application, a few lines are needed to ask the DI container for the top-level objects in the system, as specified in the XML file.

   XmlBeanFactory bf =  
     new XmlBeanFactory(new ClassPathResource(”app.xml”, getClass()));  
   Bank bank = (Bank) bf.getBean(”bank”);

Because so few lines of Spring-specific Java code are required, the application is almost completely decoupled from Spring, eliminating all the tight-coupling problems of systems like EJB2.

Although XML can be verbose and hard to read,15 the “policy” specified in these configuration files is simpler than the complicated proxy and aspect logic that is hidden from view and created automatically. This type of architecture is so compelling that frameworks like Spring led to a complete overhaul of the EJB standard for version 3. EJB3

15. The example can be simplified using mechanisms that exploit convention over configuration and Java 5 annotations to reduce the amount of explicit “wiring” logic required.

largely follows the Spring model of declaratively supporting cross-cutting concerns using XML configuration files and/or Java 5 annotations.

[Listing 11-5](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#ch11lt5) shows our Bank object rewritten in EJB316.

16. Adapted from <http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html>

**Listing 11-5 An EBJ3 Bank EJB**

   package com.example.banking.model;  
   import javax.persistence.\*;  
   import java.util.ArrayList;  
   import java.util.Collection;  
  
   @Entity  
   @Table(name = “BANKS”)  
   public class Bank implements java.io.Serializable {  
      @Id @GeneratedValue(strategy=GenerationType.AUTO)  
      private int id;  
  
      @Embeddable // An object “inlined” in Bank’s DB row  
      public class Address {  
         protected String streetAddr1;  
         protected String streetAddr2;  
         protected String city;  
         protected String state;  
         protected String zipCode;  
      }  
      @Embedded  
      private Address address;  
  
      @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,  
                 mappedBy=”bank”)  
      private Collection<Account> accounts = new ArrayList<Account>();  
  
      public int getId() {  
         return id;  
      }  
  
      public void setId(int id) {  
         this.id = id;  
      }  
  
      public void addAccount(Account account) {  
         account.setBank(this);  
         accounts.add(account);  
      }  
      public Collection<Account> getAccounts() {  
         return accounts;  
      }  
   public void setAccounts(Collection<Account> accounts) {  
      this.accounts = accounts;  
   }  
}

This code is much cleaner than the original EJB2 code. Some of the entity details are still here, contained in the annotations. However, because none of that information is outside of the annotations, the code is clean, clear, and hence easy to test drive, maintain, and so on.

Some or all of the persistence information in the annotations can be moved to XML deployment descriptors, if desired, leaving a truly pure POJO. If the persistence mapping details won’t change frequently, many teams may choose to keep the annotations, but with far fewer harmful drawbacks compared to the EJB2 invasiveness.

### **ASPECTJ ASPECTS**

Finally, the most full-featured tool for separating concerns through aspects is the AspectJ language,17an extension of Java that provides “first-class” support for aspects as modularity constructs. The pure Java approaches provided by Spring AOP and JBoss AOP are sufficient for 80–90 percent of the cases where aspects are most useful. However, AspectJ provides a very rich and powerful tool set for separating concerns. The drawback of AspectJ is the need to adopt several new tools and to learn new language constructs and usage idioms.

17. See [[AspectJ](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#AspectJ)] and [[Colyer](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Colyer)].

The adoption issues have been partially mitigated by a recently introduced “annotation form” of AspectJ, where Java 5 annotations are used to define aspects using pure Java code. Also, the Spring Framework has a number of features that make incorporation of annotation-based aspects much easier for a team with limited AspectJ experience.

A full discussion of AspectJ is beyond the scope of this book. See [[AspectJ](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#AspectJ)], [[Colyer](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Colyer)], and [[Spring](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Spring)] for more information.

### **TEST DRIVE THE SYSTEM ARCHITECTURE**

The power of separating concerns through aspect-like approaches can’t be overstated. If you can write your application’s domain logic using POJOs, decoupled from any architecture concerns at the code level, then it is possible to truly test drive your architecture. You can evolve it from simple to sophisticated, as needed, by adopting new technologies on demand. It is not necessary to do a Big Design Up Front18 (BDUF). In fact, BDUF is even harmful because it inhibits adapting to change, due to the psychological resistance to discarding prior effort and because of the way architecture choices influence subsequent thinking about the design.

18. Not to be confused with the good practice of up-front design, BDUF is the practice of designing everything up front before implementing anything at all.

Building architects have to do BDUF because it is not feasible to make radical architectural changes to a large physical structure once construction is well underway.19 Although software has its own physics,20 it is economically feasible to make radical change, if the structure of the software separates its concerns effectively.

19. There is still a significant amount of iterative exploration and discussion of details, even after construction starts.

20. The term software physics was first used by [[Kolence](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Kolence)].

This means we can start a software project with a “naively simple” but nicely decoupled architecture, delivering working user stories quickly, then adding more infrastructure as we scale up. Some of the world’s largest Web sites have achieved very high availability and performance, using sophisticated data caching, security, virtualization, and so forth, all done efficiently and flexibly because the minimally coupled designs are appropriately simple at each level of abstraction and scope.

Of course, this does not mean that we go into a project “rudderless.” We have some expectations of the general scope, goals, and schedule for the project, as well as the general structure of the resulting system. However, we must maintain the ability to change course in response to evolving circumstances.

The early EJB architecture is but one of many well-known APIs that are over-engineered and that compromise separation of concerns. Even well-designed APIs can be overkill when they aren’t really needed. A good API should largely disappear from view most of the time, so the team expends the majority of its creative efforts focused on the user stories being implemented. If not, then the architectural constraints will inhibit the efficient delivery of optimal value to the customer.

To recap this long discussion,

An optimal system architecture consists of modularized domains of concern, each of which is implemented with Plain Old Java (or other) Objects. The different domains are integrated together with minimally invasive Aspects or Aspect-like tools. This architecture can be test-driven, just like the code.

### **OPTIMIZE DECISION MAKING**

Modularity and separation of concerns make decentralized management and decision making possible. In a sufficiently large system, whether it is a city or a software project, no one person can make all the decisions.

We all know it is best to give responsibilities to the most qualified persons. We often forget that it is also best to postpone decisions until the last possible moment. This isn’t lazy or irresponsible; it lets us make informed choices with the best possible information. A premature decision is a decision made with suboptimal knowledge. We will have that much less customer feedback, mental reflection on the project, and experience with our implementation choices if we decide too soon.

The agility provided by a POJO system with modularized concerns allows us to make optimal, just-in-time decisions, based on the most recent knowledge. The complexity of these decisions is also reduced.

### **USE STANDARDS WISELY, WHEN THEY ADD** DEMONSTRABLE **VALUE**

Building construction is a marvel to watch because of the pace at which new buildings are built (even in the dead of winter) and because of the extraordinary designs that are possible with today’s technology. Construction is a mature industry with highly optimized parts, methods, and standards that have evolved under pressure for centuries.

Many teams used the EJB2 architecture because it was a standard, even when lighter-weight and more straightforward designs would have been sufficient. I have seen teams become obsessed with various strongly hyped standards and lose focus on implementing value for their customers.

Standards make it easier to reuse ideas and components, recruit people with relevant experience, encapsulate good ideas, and wire components together. However, the process of creating standards can sometimes take too long for industry to wait, and some standards lose touch with the real needs of the adopters they are intended to serve.

### **SYSTEMS NEED DOMAIN-SPECIFIC LANGUAGES**

Building construction, like most domains, has developed a rich language with a vocabulary, idioms, and patterns21 that convey essential information clearly and concisely. In software, there has been renewed interest recently in creating Domain-Specific Languages (DSLs),22 which are separate, small scripting languages or APIs in standard languages that permit code to be written so that it reads like a structured form of prose that a domain expert might write.

21. The work of [[Alexander](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#Alexander)] has been particularly influential on the software community.

22. See, for example, [[DSL](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#DSL)]. [[JMock](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter11.html#JMock)] is a good example of a Java API that creates a DSL.

A good DSL minimizes the “communication gap” between a domain concept and the code that implements it, just as agile practices optimize the communications within a team and with the project’s stakeholders. If you are implementing domain logic in the same language that a domain expert uses, there is less risk that you will incorrectly translate the domain into the implementation.

DSLs, when used effectively, raise the abstraction level above code idioms and design patterns. They allow the developer to reveal the intent of the code at the appropriate level of abstraction.

Domain-Specific Languages allow all levels of abstraction and all domains in the application to be expressed as POJOs, from high-level policy to low-level details.

### **CONCLUSION**

Systems must be clean too. An invasive architecture overwhelms the domain logic and impacts agility. When the domain logic is obscured, quality suffers because bugs find it easier to hide and stories become harder to implement. If agility is compromised, productivity suffers and the benefits of TDD are lost.

At all levels of abstraction, the intent should be clear. This will only happen if you write POJOs and you use aspect-like mechanisms to incorporate other implementation concerns noninvasively.

Whether you are designing systems or individual modules, never forget to *use the simplest thing that can possibly work*.

## ****Emergence****

### **GETTING CLEAN VIA EMERGENT DESIGN**

What if there were four simple rules that you could follow that would help you create good designs as you worked? What if by following these rules you gained insights into the structure and design of your code, making it easier to apply principles such as SRP and DIP? What if these four rules facilitated the emergence of good designs?

Many of us feel that Kent Beck’s four rules of Simple Design1 are of significant help in creating well-designed software.

1. [[XPE](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter12.html#XPE)].

According to Kent, a design is “simple” if it follows these rules:

• Runs all the tests

• Contains no duplication

• Expresses the intent of the programmer

• Minimizes the number of classes and methods

The rules are given in order of importance.

### **SIMPLE DESIGN RULE 1: RUNS ALL THE TESTS**

First and foremost, a design must produce a system that acts as intended. A system might have a perfect design on paper, but if there is no simple way to verify that the system actually works as intended, then all the paper effort is questionable.

A system that is comprehensively tested and passes all of its tests all of the time is a testable system. That’s an obvious statement, but an important one. Systems that aren’t testable aren’t verifiable. Arguably, a system that cannot be verified should never be deployed.

Fortunately, making our systems testable pushes us toward a design where our classes are small and single purpose. It’s just easier to test classes that conform to the SRP. The more tests we write, the more we’ll continue to push toward things that are simpler to test. So making sure our system is fully testable helps us create better designs.

Tight coupling makes it difficult to write tests. So, similarly, the more tests we write, the more we use principles like DIP and tools like dependency injection, interfaces, and abstraction to minimize coupling. Our designs improve even more.

Remarkably, following a simple and obvious rule that says we need to have tests and run them continuously impacts our system’s adherence to the primary OO goals of low coupling and high cohesion. Writing tests leads to better designs.

### **SIMPLE DESIGN RULES 2–4: REFACTORING**

Once we have tests, we are empowered to keep our code and classes clean. We do this by incrementally refactoring the code. For each few lines of code we add, we pause and reflect on the new design. Did we just degrade it? If so, we clean it up and run our tests to demonstrate that we haven’t broken anything. The fact that we have these tests eliminates the fear that cleaning up the code will break it!

During this refactoring step, we can apply anything from the entire body of knowledge about good software design. We can increase cohesion, decrease coupling, separate concerns, modularize system concerns, shrink our functions and classes, choose better names, and so on. This is also where we apply the final three rules of simple design: Eliminate duplication, ensure expressiveness, and minimize the number of classes and methods.

### **NO DUPLICATION**

Duplication is the primary enemy of a well-designed system. It represents additional work, additional risk, and additional unnecessary complexity. Duplication manifests itself in many forms. Lines of code that look exactly alike are, of course, duplication. Lines of code that are similar can often be massaged to look even more alike so that they can be more easily refactored. And duplication can exist in other forms such as duplication of implementation. For example, we might have two methods in a collection class:

   int size() {}  
   boolean isEmpty() {}

We could have separate implementations for each method. The isEmpty method could track a boolean, while size could track a counter. Or, we can eliminate this duplication by tying isEmpty to the definition of size:

   boolean isEmpty() {  
      return 0 == size();  
   }

Creating a clean system requires the will to eliminate duplication, even in just a few lines of code. For example, consider the following code:

   public void scaleToOneDimension(  
        float desiredDimension, float imageDimension) {  
     if (Math.abs(desiredDimension - imageDimension) < errorThreshold)  
        return;  
     float scalingFactor = desiredDimension / imageDimension;  
     scalingFactor = (float)(Math.floor(scalingFactor \* 100) \* 0.01f);  
  
     RenderedOp newImage = ImageUtilities.getScaledImage(  
        image, scalingFactor, scalingFactor);  
     image.dispose();  
     System.gc();  
     image = newImage;  
   }  
   public synchronized void rotate(int degrees) {  
      RenderedOp newImage = ImageUtilities.getRotatedImage(  
         image, degrees);  
      image.dispose();  
      System.gc();  
      image = newImage;  
   }

To keep this system clean, we should eliminate the small amount of duplication between the scaleToOneDimension and rotate methods:

   public void scaleToOneDimension(  
        float desiredDimension, float imageDimension) {  
     if (Math.abs(desiredDimension - imageDimension) < errorThreshold)  
        return;  
     float scalingFactor = desiredDimension / imageDimension;  
     scalingFactor = (float)(Math.floor(scalingFactor \* 100) \* 0.01f);  
     **replaceImage(ImageUtilities.getScaledImage(  
        image, scalingFactor, scalingFactor));**  
   }  
   public synchronized void rotate(int degrees) {  
      **replaceImage(ImageUtilities.getRotatedImage(image, degrees));**  
   }  
   **privatex void replaceImage(RenderedOp newImage) {**  
      **image.dispose();**  
      **System.gc();**  
   **image = newImage;**  
   }

As we extract commonality at this very tiny level, we start to recognize violations of SRP. So we might move a newly extracted method to another class. That elevates its visibility. Someone else on the team may recognize the opportunity to further abstract the new method and reuse it in a different context. This “reuse in the small” can cause system complexity to shrink dramatically. Understanding how to achieve reuse in the small is essential to achieving reuse in the large.

The TEMPLATE METHOD2 pattern is a common technique for removing higher-level duplication. For example:

   public class VacationPolicy {  
      public void accrueUSDivisionVacation() {  
         // code to calculate vacation based on hours worked to date  
         // …  
         // code to ensure vacation meets US minimums  
         // …  
         // code to apply vaction to payroll record  
         // …  
      }  
  
      public void accrueEUDivisionVacation() {  
         // code to calculate vacation based on hours worked to date  
         // …  
         // code to ensure vacation meets EU minimums  
         // …  
         // code to apply vaction to payroll record  
         // …  
      }  
   }

The code across accrueUSDivisionVacation and accrueEuropeanDivisionVacation is largely the same, with the exception of calculating legal minimums. That bit of the algorithm changes based on the employee type.

We can eliminate the obvious duplication by applying the TEMPLATE METHOD pattern.

   abstract public class VacationPolicy {  
      public void accrueVacation() {  
         **calculateBaseVacationHours();**

         **alterForLegalMinimums();**  
         **applyToPayroll();**  
      }  
  
      private void calculateBaseVacationHours() { /\* … \*/ };  
      abstract protected void alterForLegalMinimums();  
      private void applyToPayroll() { /\* … \*/ };  
   }  
   public class USVacationPolicy extends VacationPolicy {  
      @Override protected void alterForLegalMinimums() {  
          // US specific logic  
      }  
   }  
  
   public class EUVacationPolicy extends VacationPolicy {  
      @Override protected void alterForLegalMinimums() {  
          // EU specific logic  
      }  
   }

The subclasses fill in the “hole” in the accrueVacation algorithm, supplying the only bits of information that are not duplicated.

### **EXPRESSIVE**

Most of us have had the experience of working on convoluted code. Many of us have produced some convoluted code ourselves. It’s easy to write code that we understand, because at the time we write it we’re deep in an understanding of the problem we’re trying to solve. Other maintainers of the code aren’t going to have so deep an understanding.

The majority of the cost of a software project is in long-term maintenance. In order to minimize the potential for defects as we introduce change, it’s critical for us to be able to understand what a system does. As systems become more complex, they take more and more time for a developer to understand, and there is an ever greater opportunity for a misunderstanding. Therefore, code should clearly express the intent of its author. The clearer the author can make the code, the less time others will have to spend understanding it. This will reduce defects and shrink the cost of maintenance.

You can express yourself by choosing good names. We want to be able to hear a class or function name and not be surprised when we discover its responsibilities.

You can also express yourself by keeping your functions and classes small. Small classes and functions are usually easy to name, easy to write, and easy to understand.

You can also express yourself by using standard nomenclature. Design patterns, for example, are largely about communication and expressiveness. By using the standard pattern names, such as COMMAND or VISITOR, in the names of the classes that implement those patterns, you can succinctly describe your design to other developers.

Well-written unit tests are also expressive. A primary goal of tests is to act as documentation by example. Someone reading our tests should be able to get a quick understanding of what a class is all about.

But the most important way to be expressive is to try. All too often we get our code working and then move on to the next problem without giving sufficient thought to making that code easy for the next person to read. Remember, the most likely next person to read the code will be you.

So take a little pride in your workmanship. Spend a little time with each of your functions and classes. Choose better names, split large functions into smaller functions, and generally just take care of what you’ve created. Care is a precious resource.

### **MINIMAL CLASSES AND METHODS**

Even concepts as fundamental as elimination of duplication, code expressiveness, and the SRP can be taken too far. In an effort to make our classes and methods small, we might create too many tiny classes and methods. So this rule suggests that we also keep our function and class counts low.

High class and method counts are sometimes the result of pointless dogmatism. Consider, for example, a coding standard that insists on creating an interface for each and every class. Or consider developers who insist that fields and behavior must always be separated into data classes and behavior classes. Such dogma should be resisted and a more pragmatic approach adopted.

Our goal is to keep our overall system small while we are also keeping our functions and classes small. Remember, however, that this rule is the lowest priority of the four rules of Simple Design. So, although it’s important to keep class and function count low, it’s more important to have tests, eliminate duplication, and express yourself.

### **CONCLUSION**

Is there a set of simple practices that can replace experience? Clearly not. On the other hand, the practices described in this chapter and in this book are a crystallized form of the many decades of experience enjoyed by the authors. Following the practice of simple design can and does encourage and enable developers to adhere to good principles and patterns that otherwise take years to learn.

## ****Concurrency****

“Objects are abstractions of processing. Threads are abstractions of schedule.”

—James O. Coplien1

1. Private correspondence.

Writing clean concurrent programs is hard—very hard. It is much easier to write code that executes in a single thread. It is also easy to write multithreaded code that looks fine on the surface but is broken at a deeper level. Such code works fine until the system is placed under stress.

In this chapter we discuss the need for concurrent programming, and the difficulties it presents. We then present several recommendations for dealing with those difficulties, and writing clean concurrent code. Finally, we conclude with issues related to testing concurrent code.

Clean Concurrency is a complex topic, worthy of a book by itself. Our strategy in this book is to present an overview here and provide a more detailed tutorial in “Concurrency II” on page [317](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_317). If you are just curious about concurrency, then this chapter will suffice for you now. If you have a need to understand concurrency at a deeper level, then you should read through the tutorial as well.

### **WHY CONCURRENCY?**

Concurrency is a decoupling strategy. It helps us decouple what gets done from when it gets done. In single-threaded applications what and when are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and know the state of the system by which breakpoints are hit.

Decoupling what from when can dramatically improve both the throughput and structures of an application. From a structural point of view the application looks like many little collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.

Consider, for example, the standard “Servlet” model of Web applications. These systems run under the umbrella of a Web or EJB container that partially manages concurrency for you. The servlets are executed asynchronously whenever Web requests come in. The servlet programmer does not have to manage all the incoming requests. In principle, each servlet execution lives in its own little world and is decoupled from all the other servlet executions.

Of course if it were that easy, this chapter wouldn’t be necessary. In fact, the decoupling provided by Web containers is far less than perfect. Servlet programmers have to be very aware, and very careful, to make sure their concurrent programs are correct. Still, the structural benefits of the servlet model are significant.

But structure is not the only motive for adopting concurrency. Some systems have response time and throughput constraints that require hand-coded concurrent solutions. For example, consider a single-threaded information aggregator that acquires information from many different Web sites and merges that information into a daily summary. Because this system is single threaded, it hits each Web site in turn, always finishing one before starting the next. The daily run needs to execute in less than 24 hours. However, as more and more Web sites are added, the time grows until it takes more than 24 hours to gather all the data. The single-thread involves a lot of waiting at Web sockets for I/O to complete. We could improve the performance by using a multithreaded algorithm that hits more than one Web site at a time.

Or consider a system that handles one user at a time and requires only one second of time per user. This system is fairly responsive for a few users, but as the number of users increases, the system’s response time increases. No user wants to get in line behind 150 others! We could improve the response time of this system by handling many users concurrently.

Or consider a system that interprets large data sets but can only give a complete solution after processing all of them. Perhaps each data set could be processed on a different computer, so that many data sets are being processed in parallel.

#### **Myths and Misconceptions**

And so there are compelling reasons to adopt concurrency. However, as we said before, concurrency is hard. If you aren’t very careful, you can create some very nasty situations. Consider these common myths and misconceptions:

• Concurrency always improves performance.  
Concurrency can sometimes improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situation is trivial.

• Design does not change when writing concurrent programs.  
In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of what from when usually has a huge effect on the structure of the system.

• Understanding concurrency issues is not important when working with a container such as a Web or EJB container.  
In fact, you’d better know just what your container is doing and how to guard against the issues of concurrent update and deadlock described later in this chapter.

Here are a few more balanced sound bites regarding writing concurrent software:

• Concurrency incurs some overhead, both in performance as well as writing additional code.

• Correct concurrency is complex, even for simple problems.

• Concurrency bugs aren’t usually repeatable, so they are often ignored as one-offs2 instead of the true defects they are.

2. Cosmic-rays, glitches, and so on.

• Concurrency often requires a fundamental change in design strategy.

### **CHALLENGES**

What makes concurrent programming so difficult? Consider the following trivial class:

   public class X {  
      private int lastIdUsed;  
  
      public int getNextId() {  
           return ++lastIdUsed;  
       }  
   }

Let’s say we create an instance of X, set the lastIdUsed field to 42, and then share the instance between two threads. Now suppose that both of those threads call the method getNextId(); there are three possible outcomes:

• Thread one gets the value 43, thread two gets the value 44, lastIdUsed is 44.

• Thread one gets the value 44, thread two gets the value 43, lastIdUsed is 44.

• Thread one gets the value 43, thread two gets the value 43, lastIdUsed is 43.

The surprising third result3 occurs when the two threads step on each other. This happens because there are many possible paths that the two threads can take through that one line of Java code, and some of those paths generate incorrect results. How many different paths are there? To really answer that question, we need to understand what the Just-In-Time Compiler does with the generated byte-code, and understand what the Java memory model considers to be atomic.

3. See “[Digging Deeper](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lev2sec6)” on page [323](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_323).

A quick answer, working with just the generated byte-code, is that there are 12,870 different possible execution paths4 for those two threads executing within the getNextId method. If the type of lastIdUsed is changed from int to long, the number of possible paths increases to 2,704,156. Of course most of those paths generate valid results. The problem is that some of them don’t.

4. See “[Possible Paths of Execution](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lev1sec2)” on page [321](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_321).

### **CONCURRENCY DEFENSE PRINCIPLES**

What follows is a series of principles and techniques for defending your systems from the problems of concurrent code.

#### **Single Responsibility Principle**

The SRP5 states that a given method/class/component should have a single reason to change. Concurrency design is complex enough to be a reason to change in it’s own right and therefore deserves to be separated from the rest of the code. Unfortunately, it is all too common for concurrency implementation details to be embedded directly into other production code. Here are a few things to consider:

5. [[PPP](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter13.html#PPP)]

• Concurrency-related code has its own life cycle of development, change, and tuning.

• Concurrency-related code has its own challenges, which are different from and often more difficult than nonconcurrency-related code.

• The number of ways in which miswritten concurrency-based code can fail makes it challenging enough without the added burden of surrounding application code.

**Recommendation**: Keep your concurrency-related code separate from other code.6

6. See “[Client/Server Example](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lev1sec1)” on page [317](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_317).

#### **Corollary: Limit the Scope of Data**

As we saw, two threads modifying the same field of a shared object can interfere with each other, causing unexpected behavior. One solution is to use the synchronized keyword to protect a critical section in the code that uses the shared object. It is important to restrict the number of such critical sections. The more places shared data can get updated, the more likely:

• You will forget to protect one or more of those places—effectively breaking all code that modifies that shared data.

• There will be duplication of effort required to make sure everything is effectively guarded (violation of DRY7).

7. [[PRAG](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter13.html#PRAG)].

• It will be difficult to determine the source of failures, which are already hard enough to find.

**Recommendation**: Take data encapsulation to heart; severely limit the access of any data that may be shared.

#### **Corollary: Use Copies of Data**

A good way to avoid shared data is to avoid sharing the data in the first place. In some situations it is possible to copy objects and treat them as read-only. In other cases it might be possible to copy objects, collect results from multiple threads in these copies and then merge the results in a single thread.

If there is an easy way to avoid sharing objects, the resulting code will be far less likely to cause problems. You might be concerned about the cost of all the extra object creation. It is worth experimenting to find out if this is in fact a problem. However, if using copies of objects allows the code to avoid synchronizing, the savings in avoiding the intrinsic lock will likely make up for the additional creation and garbage collection overhead.

#### **Corollary: Threads Should Be as Independent as Possible**

Consider writing your threaded code such that each thread exists in its own world, sharing no data with any other thread. Each thread processes one client request, with all of its required data coming from an unshared source and stored as local variables. This makes each of those threads behave as if it were the only thread in the world and there were no synchronization requirements.

For example, classes that subclass from HttpServlet receive all of their information as parameters passed in to the doGet and doPost methods. This makes each Servlet act as if it has its own machine. So long as the code in the Servlet uses only local variables, there is no chance that the Servlet will cause synchronization problems. Of course, most applications using Servletseventually run into shared resources such as database connections.

**Recommendation**: Attempt to partition data into independent subsets than can be operated on by independent threads, possibly in different processors.

### **KNOW YOUR LIBRARY**

Java 5 offers many improvements for concurrent development over previous versions. There are several things to consider when writing threaded code in Java 5:

• Use the provided thread-safe collections.

• Use the executor framework for executing unrelated tasks.

• Use nonblocking solutions when possible.

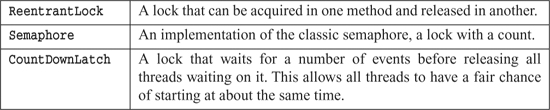
• Several library classes are not thread safe.

#### **Thread-Safe Collections**

When Java was young, Doug Lea wrote the seminal book8 Concurrent Programming in Java. Along with the book he developed several thread-safe collections, which later became part of the JDK in the java.util.concurrent package. The collections in that package are safe for multithreaded situations and they perform well. In fact, the ConcurrentHashMap implementation performs better than HashMap in nearly all situations. It also allows for simultaneous concurrent reads and writes, and it has methods supporting common composite operations that are otherwise not thread safe. If Java 5 is the deployment environment, start with ConcurrentHashMap.

8. [[Lea99](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter13.html#Lea99)].

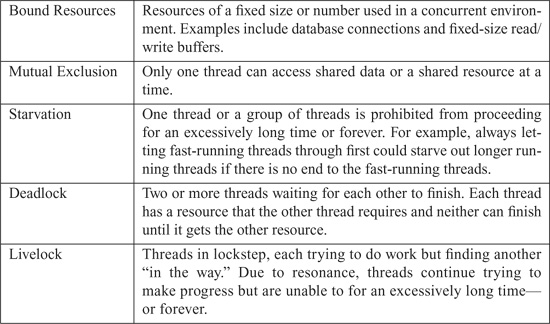
There are several other kinds of classes added to support advanced concurrency design. Here are a few examples:



**Recommendation**: Review the classes available to you. In the case of Java, become familiar with java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.

### **KNOW YOUR EXECUTION MODELS**

There are several different ways to partition behavior in a concurrent application. To discuss them we need to understand some basic definitions.



Given these definitions, we can now discuss the various execution models used in concurrent programming.

#### **Producer-Consumer9**

9. <http://en.wikipedia.org/wiki/Producer-consumer>

One or more producer threads create some work and place it in a buffer or queue. One or more consumer threads acquire that work from the queue and complete it. The queue between the producers and consumers is a bound resource. This means producers must wait for free space in the queue before writing and consumers must wait until there is something in the queue to consume. Coordination between the producers and consumers via the queue involves producers and consumers signaling each other. The producers write to the queue and signal that the queue is no longer empty. Consumers read from the queue and signal that the queue is no longer full. Both potentially wait to be notified when they can continue.

#### **Readers-Writers10**

10. <http://en.wikipedia.org/wiki/Readers-writers_problem>

When you have a shared resource that primarily serves as a source of information for readers, but which is occasionally updated by writers, throughput is an issue. Emphasizing throughput can cause starvation and the accumulation of stale information. Allowing updates can impact throughput. Coordinating readers so they do not read something a writer is updating and vice versa is a tough balancing act. Writers tend to block many readers for a long period of time, thus causing throughput issues.

The challenge is to balance the needs of both readers and writers to satisfy correct operation, provide reasonable throughput and avoiding starvation. A simple strategy makes writers wait until there are no readers before allowing the writer to perform an update. If there are continuous readers, however, the writers will be starved. On the other hand, if there are frequent writers and they are given priority, throughput will suffer. Finding that balance and avoiding concurrent update issues is what the problem addresses.

#### **Dining Philosophers11**

11. <http://en.wikipedia.org/wiki/Dining_philosophers_problem>

Imagine a number of philosophers sitting around a circular table. A fork is placed to the left of each philosopher. There is a big bowl of spaghetti in the center of the table. The philosophers spend their time thinking unless they get hungry. Once hungry, they pick up the forks on either side of them and eat. A philosopher cannot eat unless he is holding two forks. If the philosopher to his right or left is already using one of the forks he needs, he must wait until that philosopher finishes eating and puts the forks back down. Once a philosopher eats, he puts both his forks back down on the table and waits until he is hungry again.

Replace philosophers with threads and forks with resources and this problem is similar to many enterprise applications in which processes compete for resources. Unless carefully designed, systems that compete in this way can experience deadlock, livelock, throughput, and efficiency degradation.

Most concurrent problems you will likely encounter will be some variation of these three problems. Study these algorithms and write solutions using them on your own so that when you come across concurrent problems, you’ll be more prepared to solve the problem.

**Recommendation**: Learn these basic algorithms and understand their solutions.

### **BEWARE DEPENDENCIES BETWEEN SYNCHRONIZED METHODS**

Dependencies between synchronized methods cause subtle bugs in concurrent code. The Java language has the notion of synchronized, which protects an individual method. However, if there is more than one synchronized method on the same shared class, then your system may be written incorrectly.12

12. See “[Dependencies Between Methods Can Break Concurrent Code](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lev1sec4)” on page [329](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_329).

**Recommendation**: Avoid using more than one method on a shared object.

There will be times when you must use more than one method on a shared object. When this is the case, there are three ways to make the code correct:

• **Client-Based Locking**—Have the client lock the server before calling the first method and make sure the lock’s extent includes code calling the last method.

• **Server-Based Locking**—Within the server create a method that locks the server, calls all the methods, and then unlocks. Have the client call the new method.

• **Adapted Server**—create an intermediary that performs the locking. This is an example of server-based locking, where the original server cannot be changed.

### **KEEP SYNCHRONIZED SECTIONS SMALL**

The synchronized keyword introduces a lock. All sections of code guarded by the same lock are guaranteed to have only one thread executing through them at any given time. Locks are expensive because they create delays and add overhead. So we don’t want to litter our code with synchronizedstatements. On the other hand, critical sections13 must be guarded. So we want to design our code with as few critical sections as possible.

13. A critical section is any section of code that must be protected from simultaneous use for the program to be correct.

Some naive programmers try to achieve this by making their critical sections very large. However, extending synchronization beyond the minimal critical section increases contention and degrades performance.14

14. See “[Increasing Throughput](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lev1sec5)” on page [333](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_333).

**Recommendation**: Keep your synchronized sections as small as possible.

### **WRITING CORRECT SHUT-DOWN CODE IS HARD**

Writing a system that is meant to stay live and run forever is different from writing something that works for awhile and then shuts down gracefully.

Graceful shutdown can be hard to get correct. Common problems involve deadlock,15 with threads waiting for a signal to continue that never comes.

15. See “[Deadlock](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lev1sec6)” on page [335](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_335).

For example, imagine a system with a parent thread that spawns several child threads and then waits for them all to finish before it releases its resources and shuts down. What if one of the spawned threads is deadlocked? The parent will wait forever, and the system will never shut down.

Or consider a similar system that has been instructed to shut down. The parent tells all the spawned children to abandon their tasks and finish. But what if two of the children were operating as a producer/consumer pair. Suppose the producer receives the signal from the parent and quickly shuts down. The consumer might have been expecting a message from the producer and be blocked in a state where it cannot receive the shutdown signal. It could get stuck waiting for the producer and never finish, preventing the parent from finishing as well.

Situations like this are not at all uncommon. So if you must write concurrent code that involves shutting down gracefully, expect to spend much of your time getting the shutdown to happen correctly.

**Recommendation**: Think about shut-down early and get it working early. It’s going to take longer than you expect. Review existing algorithms because this is probably harder than you think.

### **TESTING THREADED CODE**

Proving that code is correct is impractical. Testing does not guarantee correctness. However, good testing can minimize risk. This is all true in a single-threaded solution. As soon as there are two or more threads using the same code and working with shared data, things get substantially more complex.

**Recommendation**: Write tests that have the potential to expose problems and then run them frequently, with different programatic configurations and system configurations and load. If tests ever fail, track down the failure. Don’t ignore a failure just because the tests pass on a subsequent run.

That is a whole lot to take into consideration. Here are a few more fine-grained recommendations:

• Treat spurious failures as candidate threading issues.

• Get your nonthreaded code working first.

• Make your threaded code pluggable.

• Make your threaded code tunable.

• Run with more threads than processors.

• Run on different platforms.

• Instrument your code to try and force failures.

#### **Treat Spurious Failures as Candidate Threading Issues**

Threaded code causes things to fail that “simply cannot fail.” Most developers do not have an intuitive feel for how threading interacts with other code (authors included). Bugs in threaded code might exhibit their symptoms once in a thousand, or a million, executions. Attempts to repeat the systems can be frustratingly. This often leads developers to write off the failure as a cosmic ray, a hardware glitch, or some other kind of “one-off.” It is best to assume that one-offs do not exist. The longer these “one-offs” are ignored, the more code is built on top of a potentially faulty approach.

**Recommendation**: *Do not ignore system failures as one-offs.*

#### **Get Your Nonthreaded Code Working First**

This may seem obvious, but it doesn’t hurt to reinforce it. Make sure code works outside of its use in threads. Generally, this means creating POJOs that are called by your threads. The POJOs are not thread aware, and can therefore be tested outside of the threaded environment. The more of your system you can place in such POJOs, the better.

**Recommendation**: Do not try to chase down nonthreading bugs and threading bugs at the same time. Make sure your code works outside of threads.

#### **Make Your Threaded Code Pluggable**

Write the concurrency-supporting code such that it can be run in several configurations:

• One thread, several threads, varied as it executes

• Threaded code interacts with something that can be both real or a test double.

• Execute with test doubles that run quickly, slowly, variable.

• Configure tests so they can run for a number of iterations.

**Recommendation**: Make your thread-based code especially pluggable so that you can run it in various configurations.

#### **Make Your Threaded Code Tunable**

Getting the right balance of threads typically requires trial an error. Early on, find ways to time the performance of your system under different configurations. Allow the number of threads to be easily tuned. Consider allowing it to change while the system is running. Consider allowing self-tuning based on throughput and system utilization.

#### **Run with More Threads Than Processors**

Things happen when the system switches between tasks. To encourage task swapping, run with more threads than processors or cores. The more frequently your tasks swap, the more likely you’ll encounter code that is missing a critical section or causes deadlock.

#### **Run on Different Platforms**

In the middle of 2007 we developed a course on concurrent programming. The course development ensued primarily under OS X. The class was presented using Windows XP running under a VM. Tests written to demonstrate failure conditions did not fail as frequently in an XP environment as they did running on OS X.

In all cases the code under test was known to be incorrect. This just reinforced the fact that different operating systems have different threading policies, each of which impacts the code’s execution. Multithreaded code behaves differently in different environments.16 You should run your tests in every potential deployment environment.

16. Did you know that the threading model in Java does not guarantee preemptive threading? Modern OS’s support preemptive threading, so you get that “for free.” Even so, it not guaranteed by the JVM.

**Recommendation**: Run your threaded code on all target platforms early and often.

#### **Instrument Your Code to Try and Force Failures**

It is normal for flaws in concurrent code to hide. Simple tests often don’t expose them. Indeed, they often hide during normal processing. They might show up once every few hours, or days, or weeks!

The reason that threading bugs can be infrequent, sporadic, and hard to repeat, is that only a very few pathways out of the many thousands of possible pathways through a vulnerable section actually fail. So the probability that a failing pathway is taken can be star-tlingly low. This makes detection and debugging very difficult.

How might you increase your chances of catching such rare occurrences? You can instrument your code and force it to run in different orderings by adding calls to methods like Object.wait(), Object.sleep(), Object.yield() and Object.priority().

Each of these methods can affect the order of execution, thereby increasing the odds of detecting a flaw. It’s better when broken code fails as early and as often as possible.

There are two options for code instrumentation:

• Hand-coded

• Automated

#### **Hand-Coded**

You can insert calls to wait(), sleep(), yield(), and priority() in your code by hand. It might be just the thing to do when you’re testing a particularly thorny piece of code.

Here is an example of doing just that:

    public synchronized String nextUrlOrNull() {  
        if(hasNext()) {  
            String url = urlGenerator.next();  
            Thread.yield(); // inserted for testing.  
            updateHasNext();  
            return url;  
        }  
        return null;  
    }

The inserted call to yield() will change the execution pathways taken by the code and possibly cause the code to fail where it did not fail before. If the code does break, it was not because you added a call to yield().17 Rather, your code was broken and this simply made the failure evident.

17. This is not strictly the case. Since the JVM does not guarantee preemptive threading, a particular algorithm might always work on an OS that does not preempt threads. The reverse is also possible but for different reasons.

There are many problems with this approach:

• You have to manually find appropriate places to do this.

• How do you know where to put the call and what kind of call to use?

• Leaving such code in a production environment unnecessarily slows the code down.

• It’s a shotgun approach. You may or may not find flaws. Indeed, the odds aren’t with you.

What we need is a way to do this during testing but not in production. We also need to easily mix up configurations between different runs, which results in increased chances of finding errors in the aggregate.

Clearly, if we divide our system up into POJOs that know nothing of threading and classes that control the threading, it will be easier to find appropriate places to instrument the code. Moreover, we could create many different test jigs that invoke the POJOs under different regimes of calls to sleep, yield, and so on.

#### **Automated**

You could use tools like an Aspect-Oriented Framework, CGLIB, or ASM to programmatically instrument your code. For example, you could use a class with a single method:

   public class ThreadJigglePoint {  
       public static void jiggle() {  
       }  
   }

You can add calls to this in various places within your code:

   public synchronized String nextUrlOrNull() {  
     if(hasNext()) {  
         ThreadJiglePoint.jiggle();  
         String url = urlGenerator.next();  
         ThreadJiglePoint.jiggle();  
         updateHasNext();  
         ThreadJiglePoint.jiggle();  
         return url;  
     }   
     return null;  
   }

Now you use a simple aspect that randomly selects among doing nothing, sleeping, or yielding.

Or imagine that the ThreadJigglePoint class has two implementations. The first implements jiggle to do nothing and is used in production. The second generates a random number to choose between sleeping, yielding, or just falling through. If you run your tests a thousand times with random jiggling, you may root out some flaws. If the tests pass, at least you can say you’ve done due diligence. Though a bit simplistic, this could be a reasonable option in lieu of a more sophisticated tool.

There is a tool called ConTest,18 developed by IBM that does something similar, but it does so with quite a bit more sophistication.

18. <http://www.alphaworks.ibm.com/tech/contest>

The point is to jiggle the code so that threads run in different orderings at different times. The combination of well-written tests and jiggling can dramatically increase the chance finding errors.

**Recommendation**: Use jiggling strategies to ferret out errors.

### **CONCLUSION**

Concurrent code is difficult to get right. Code that is simple to follow can become nightmarish when multiple threads and shared data get into the mix. If you are faced with writing concurrent code, you need to write clean code with rigor or else face subtle and infrequent failures.

First and foremost, follow the Single Responsibility Principle. Break your system into POJOs that separate thread-aware code from thread-ignorant code. Make sure when you are testing your thread-aware code, you are only testing it and nothing else. This suggests that your thread-aware code should be small and focused.

Know the possible sources of concurrency issues: multiple threads operating on shared data, or using a common resource pool. Boundary cases, such as shutting down cleanly or finishing the iteration of a loop, can be especially thorny.

Learn your library and know the fundamental algorithms. Understand how some of the features offered by the library support solving problems similar to the fundamental algorithms.

Learn how to find regions of code that must be locked and lock them. Do not lock regions of code that do not need to be locked. Avoid calling one locked section from another. This requires a deep understanding of whether something is or is not shared. Keep the amount of shared objects and the scope of the sharing as narrow as possible. Change designs of the objects with shared data to accommodate clients rather than forcing clients to manage shared state.

Issues will crop up. The ones that do not crop up early are often written off as a onetime occurrence. These so-called one-offs typically only happen under load or at seemingly random times. Therefore, you need to be able to run your thread-related code in many configurations on many platforms repeatedly and continuously. Testability, which comes naturally from following the Three Laws of TDD, implies some level of plug-ability, which offers the support necessary to run code in a wider range of configurations.

You will greatly improve your chances of finding erroneous code if you take the time to instrument your code. You can either do so by hand or using some kind of automated technology. Invest in this early. You want to be running your thread-based code as long as possible before you put it into production.

If you take a clean approach, your chances of getting it right increase drastically.

## ****Successive Refinement****

This chapter is a case study in successive refinement. You will see a module that started well but did not scale. Then you will see how the module was refactored and cleaned.

Most of us have had to parse command-line arguments from time to time. If we don’t have a convenient utility, then we simply walk the array of strings that is passed into the main function. There are several good utilities available from various sources, but none of them do exactly what I want. So, of course, I decided to write my own. I call it: Args.

Args is very simple to use. You simply construct the Args class with the input arguments and a format string, and then query the Args instance for the values of the arguments. Consider the following simple example:

**Listing 14-1 Simple use of Args**

   public static void main(String[] args)   {  
     try {  
       Args arg = new Args(“l,p#,d\*”, args);  
       boolean logging = arg.getBoolean(’l’);  
       int port = arg.getInt(’p’);  
       String directory = arg.getString(’d’);  
       executeApplication(logging, port, directory);  
     }  catch (ArgsException e) {  
         System.out.printf(“Argument error: %s\n”, e.errorMessage());  
     }  
   }

You can see how simple this is. We just create an instance of the Args class with two parameters. The first parameter is the format, or schema, string: “l,p#,d\*.” It defines three command-line arguments. The first, -l, is a boolean argument. The second, -p, is an integer argument. The third, -d, is a string argument. The second parameter to the Args constructor is simply the array of command-line argument passed into main.

If the constructor returns without throwing an ArgsException, then the incoming command-line was parsed, and the Args instance is ready to be queried. Methods like getBoolean, getInteger,and getString allow us to access the values of the arguments by their names.

If there is a problem, either in the format string or in the command-line arguments themselves, an ArgsException will be thrown. A convenient description of what went wrong can be retrieved from the errorMessage method of the exception.

### **ARGS IMPLEMENTATION**

[Listing 14-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt2) is the implementation of the Args class. Please read it very carefully. I worked hard on the style and structure and hope it is worth emulating.

**Listing 14-2 Args.java**

   package com.objectmentor.utilities.args;  
  
   import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;  
   import java.util.\*;  
  
   public class Args {  
     private Map<Character, ArgumentMarshaler> marshalers;  
   private Set<Character> argsFound;  
   private ListIterator<String> currentArgument;  
    
   public Args(String schema, String[] args) throws ArgsException {  
     marshalers = new HashMap<Character, ArgumentMarshaler>();  
     argsFound = new HashSet<Character>();  
      
     parseSchema(schema);  
     parseArgumentStrings(Arrays.asList(args));  
   }  
     
   private void parseSchema(String schema) throws ArgsException {  
     for (String element : schema.split(“,”))  
       if (element.length() > 0)  
         parseSchemaElement(element.trim());  
   }  
   private void parseSchemaElement(String element) throws ArgsException {  
     char elementId = element.charAt(0);  
     String elementTail = element.substring(1);  
     validateSchemaElementId(elementId);  
     if (elementTail.length() == 0)  
       marshalers.put(elementId, new BooleanArgumentMarshaler());  
     else if (elementTail.equals(“\*”))  
       marshalers.put(elementId, new StringArgumentMarshaler());  
     else if (elementTail.equals(“#”))  
       marshalers.put(elementId, new IntegerArgumentMarshaler());  
     else if (elementTail.equals(“##”))  
       marshalers.put(elementId, new DoubleArgumentMarshaler());  
     else if (elementTail.equals(“[\*]”))  
       marshalers.put(elementId, new StringArrayArgumentMarshaler());  
     else  
       throw new ArgsException(INVALID\_ARGUMENT\_FORMAT, elementId, elementTail);  
   }  
   private void validateSchemaElementId(char elementId) throws ArgsException {  
     if (!Character.isLetter(elementId))  
       throw new ArgsException(INVALID\_ARGUMENT\_NAME, elementId, null);  
   }  
   private void parseArgumentStrings(List<String> argsList) throws ArgsException   
   {  
     for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)   
     {  
       String argString = currentArgument.next();  
       if (argString.startsWith(“-”)) {  
         parseArgumentCharacters(argString.substring(1));  
       } else {  
         currentArgument.previous();  
         break;  
       }  
     }  
   }  
     private void parseArgumentCharacters(String argChars) throws ArgsException {  
       for (int i = 0; i < argChars.length(); i++)  
         parseArgumentCharacter(argChars.charAt(i));  
     }  
  
     private void parseArgumentCharacter(char argChar) throws ArgsException {  
       ArgumentMarshaler m = marshalers.get(argChar);  
       if (m == null) {  
         throw new ArgsException(UNEXPECTED\_ARGUMENT, argChar, null);  
       } else {  
         argsFound.add(argChar);  
         try {  
           m.set(currentArgument);  
         } catch (ArgsException e) {  
           e.setErrorArgumentId(argChar);  
           throw e;  
         }  
       }  
    }  
    public boolean has(char arg) {  
      return argsFound.contains(arg);  
    }  
  
    public int nextArgument() {  
      return currentArgument.nextIndex();  
    }  
    public boolean getBoolean(char arg) {  
      return BooleanArgumentMarshaler.getValue(marshalers.get(arg));  
    }  
  
    public String getString(char arg) {  
      return StringArgumentMarshaler.getValue(marshalers.get(arg));  
    }  
  
    public int getInt(char arg) {  
      return IntegerArgumentMarshaler.getValue(marshalers.get(arg));  
    }  
    public double getDouble(char arg) {  
      return DoubleArgumentMarshaler.getValue(marshalers.get(arg));  
    }  
    public String[] getStringArray(char arg) {  
      return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));  
    }  
   }

Notice that you can read this code from the top to the bottom without a lot of jumping around or looking ahead. The one thing you may have had to look ahead for is the definition of ArgumentMarshaler, which I left out intentionally. Having read this code carefully, you should understand what the ArgumentMarshaler interface is and what its derivatives do. I’ll show a few of them to you now ([Listing 14-3](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt3) through [Listing 14-6](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt6)).

**Listing 14-3 ArgumentMarshaler.java**

   public interface ArgumentMarshaler {  
     void set(Iterator<String> currentArgument) throws ArgsException;  
   }

**Listing 14-4 BooleanArgumentMarshaler.java**

public class BooleanArgumentMarshaler implements ArgumentMarshaler {  
  private boolean booleanValue = false;  
    
  public void set(Iterator<String> currentArgument) throws ArgsException {  
    booleanValue = true;  
  }  
  
  public static boolean getValue(ArgumentMarshaler am) {  
    if (am != null && am instanceof BooleanArgumentMarshaler)  
      return ((BooleanArgumentMarshaler) am).booleanValue;  
    else  
      return false;  
  }  
}

**Listing 14-5 StringArgumentMarshaler.java**

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;  
  
public class StringArgumentMarshaler implements ArgumentMarshaler {  
  private String stringValue =   
  
  public void set(Iterator<String> currentArgument) throws ArgsException {  
    try {  
      stringValue = currentArgument.next();  
    } catch (NoSuchElementException e) {  
      throw new ArgsException(MISSING\_STRING);  
    }  
  }  
  
  public static String getValue(ArgumentMarshaler am) {  
    if (am != null && am instanceof StringArgumentMarshaler)  
      return ((StringArgumentMarshaler) am).stringValue;  
    else  
      return ””;  
  }  
}

The other ArgumentMarshaler derivatives simply replicate this pattern for doubles and Stringarrays and would serve to clutter this chapter. I’ll leave them to you as an exercise.

One other bit of information might be troubling you: the definition of the error code constants. They are in the ArgsException class ([Listing 14-7](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt7)).

**Listing 14-6 IntegerArgumentMarshaler.java**

   import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;  
  
   public class IntegerArgumentMarshaler implements ArgumentMarshaler {  
     private int intValue = 0;  
    
     public void set(Iterator<String> currentArgument) throws ArgsException {  
       String parameter = null;  
       try {  
         parameter = currentArgument.next();  
         intValue = Integer.parseInt(parameter);  
       } catch (NoSuchElementException e) {  
         throw new ArgsException(MISSING\_INTEGER);  
       } catch (NumberFormatException e) {  
         throw new ArgsException(INVALID\_INTEGER, parameter);  
       }  
     }  
    
     public static int getValue(ArgumentMarshaler am) {  
       if (am != null && am instanceof IntegerArgumentMarshaler)  
         return ((IntegerArgumentMarshaler) am).intValue;  
       else  
         return 0;  
     }  
   }

**Listing 14-7 ArgsException.java**

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.\*;  
  
public class ArgsException extends Exception {  
  private char errorArgumentId = ’\0’;  
  private String errorParameter = null;  
  private ErrorCode errorCode = OK;  
  
  public ArgsException() {}  
  
  public ArgsException(String message) {super(message);}  
    
  public ArgsException(ErrorCode errorCode) {  
    this.errorCode = errorCode;  
  }  
    
  public ArgsException(ErrorCode errorCode, String errorParameter) {  
    this.errorCode = errorCode;  
    this.errorParameter = errorParameter;  
  }  
  
  public ArgsException(ErrorCode errorCode,   
                       char errorArgumentId, String errorParameter) {  
    this.errorCode = errorCode;  
    this.errorParameter = errorParameter;  
    this.errorArgumentId = errorArgumentId;  
  }  
  
  public char getErrorArgumentId() {  
    return errorArgumentId;  
  }  
  
  public void setErrorArgumentId(char errorArgumentId) {  
    this.errorArgumentId = errorArgumentId;  
  }  
     
  public String getErrorParameter() {  
    return errorParameter;  
  }  
  
  public void setErrorParameter(String errorParameter) {  
    this.errorParameter = errorParameter;  
  }  
    
  public ErrorCode getErrorCode() {  
    return errorCode;  
  }  
    
  public void setErrorCode(ErrorCode errorCode) {  
    this.errorCode = errorCode;  
  }  
    
  public String errorMessage() {  
    switch (errorCode) {  
      case OK:  
        return “TILT: Should not get here.”;  
      case UNEXPECTED\_ARGUMENT:  
        return String.format(“Argument -%c unexpected.”, errorArgumentId);  
      case MISSING\_STRING:  
        return String.format(“Could not find string parameter for -%c.”,   
                              errorArgumentId);  
      case INVALID\_INTEGER:  
        return String.format(“Argument -%c expects an integer but was ’%s’.”,   
                              errorArgumentId, errorParameter);  
      case MISSING\_INTEGER:  
        return String.format(“Could not find integer parameter for -%c.”,  
                              errorArgumentId);  
      case INVALID\_DOUBLE:  
        return String.format(“Argument -%c expects a double but was ’%s’.”,   
                              errorArgumentId, errorParameter);  
      case MISSING\_DOUBLE:  
        return String.format(“Could not find double parameter for -%c.”,   
                              errorArgumentId);  
      case INVALID\_ARGUMENT\_NAME:  
        return String.format(“’%c” is not a valid argument name.”,   
                              errorArgumentId);  
      case INVALID\_ARGUMENT\_FORMAT:  
        return String.format(“’%s” is not a valid argument format.”,   
                              errorParameter);  
    }  
    return ””;  
  }  
  
  public enum ErrorCode {  
    OK, INVALID\_ARGUMENT\_FORMAT, UNEXPECTED\_ARGUMENT, INVALID\_ARGUMENT\_NAME,  
    MISSING\_STRING,  
    MISSING\_INTEGER, INVALID\_INTEGER,  
    MISSING\_DOUBLE, INVALID\_DOUBLE}  
}

It’s remarkable how much code is required to flesh out the details of this simple concept. One of the reasons for this is that we are using a particularly wordy language. Java, being a statically typed language, requires a lot of words in order to satisfy the type system. In a language like Ruby, Python, or Smalltalk, this program is much smaller.1

1. I recently rewrote this module in Ruby. It was 1/7th the size and had a subtly better structure.

Please read the code over one more time. Pay special attention to the way things are named, the size of the functions, and the formatting of the code. If you are an experienced programmer, you may have some quibbles here and there with various parts of the style or structure. Overall, however, I hope you conclude that this program is nicely written and has a clean structure.

For example, it should be obvious how you would add a new argument type, such as a date argument or a complex number argument, and that such an addition would require a trivial amount of effort. In short, it would simply require a new derivative of Argument-Marshaler, a new getXXX function, and a new case statement in the parseSchemaElement function. There would also probably be a new ArgsException.ErrorCode and a new error message.

#### **How Did I Do This?**

Let me set your mind at rest. I did not simply write this program from beginning to end in its current form. More importantly, I am not expecting you to be able to write clean and elegant programs in one pass. If we have learned anything over the last couple of decades, it is that programming is a craft more than it is a science. To write clean code, you must first write dirty code *and then clean it*.

This should not be a surprise to you. We learned this truth in grade school when our teachers tried (usually in vain) to get us to write rough drafts of our compositions. The process, they told us, was that we should write a rough draft, then a second draft, then several subsequent drafts until we had our final version. Writing clean compositions, they tried to tell us, is a matter of successive refinement.

Most freshman programmers (like most grade-schoolers) don’t follow this advice particularly well. They believe that the primary goal is to get the program working. Once it’s “working,” they move on to the next task, leaving the “working” program in whatever state they finally got it to “work.” Most seasoned programmers know that this is professional suicide.

### **ARGS: THE ROUGH DRAFT**

[Listing 14-8](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt8) shows an earlier version of the Args class. It “works.” And it’s messy.

**Listing 14-8 Args.java (first draft)**

import java.text.ParseException;  
import java.util.\*;  
  
public class Args {  
  private String schema;  
  private String[] args;  
  private boolean valid = true;  
  private Set<Character> unexpectedArguments = new TreeSet<Character>();  
  private Map<Character, Boolean> booleanArgs =   
    new HashMap  
       <Character, Boolean>();  
  private Map<Character, String> stringArgs = new HashMap  
       <Character, String>();  
  private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();  
  private Set<Character> argsFound = new HashSet<Character>();  
  private int currentArgument;  
  private char errorArgumentId = ’\0’;  
  private String errorParameter = “TILT”;  
  private ErrorCode errorCode = ErrorCode.OK;  
  
  private enum ErrorCode {  
    OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT}  
  
  public Args(String schema, String[] args) throws ParseException {  
    this.schema = schema;  
    this.args = args;  
    valid = parse();  
  }  
  
  private boolean parse() throws ParseException {  
    if (schema.length() == 0 && args.length == 0)  
      return true;  
    parseSchema();  
    try {  
      parseArguments();  
    } catch (ArgsException e) {  
    }  
    return valid;  
  }  
  
  private boolean parseSchema() throws ParseException {  
    for (String element : schema.split(“,”)) {  
      if (element.length() > 0) {  
        String trimmedElement = element.trim();  
        parseSchemaElement(trimmedElement);  
      }  
    }  
    return true;  
  }  
  
  private void parseSchemaElement(String element) throws ParseException {  
    char elementId = element.charAt(0);  
    String elementTail = element.substring(1);  
    validateSchemaElementId(elementId);  
    if (isBooleanSchemaElement(elementTail))  
      parseBooleanSchemaElement(elementId);  
    else if (isStringSchemaElement(elementTail))  
      parseStringSchemaElement(elementId);  
    else if (isIntegerSchemaElement(elementTail)) {  
      parseIntegerSchemaElement(elementId);  
    } else {  
      throw new ParseException(  
        String.format(“Argument: %c has invalid format: %s.”,   
                     elementId, elementTail), 0);  
    }  
  }  
  
  private void validateSchemaElementId(char elementId) throws ParseException {  
    if (!Character.isLetter(elementId)) {  
      throw new ParseException(  
        “Bad character:” + elementId + “in Args format: ” + schema, 0);  
    }  
  }  
  
  private void parseBooleanSchemaElement(char elementId) {  
    booleanArgs.put(elementId, false);  
  }  
  
  private void parseIntegerSchemaElement(char elementId) {  
    intArgs.put(elementId, 0);  
  }  
  
  private void parseStringSchemaElement(char elementId) {  
    stringArgs.put(elementId, ””);  
  }  
  
  private boolean isStringSchemaElement(String elementTail) {  
    return elementTail.equals(”\*”);  
  }  
  
  private boolean isBooleanSchemaElement(String elementTail) {  
    return elementTail.length() == 0;  
  }  
  
  private boolean isIntegerSchemaElement(String elementTail) {  
    return elementTail.equals(”#”);  
}  
  
  private boolean parseArguments() throws ArgsException {  
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)   
    {  
      String arg = args[currentArgument];  
      parseArgument(arg);  
    }  
    return true;  
  }  
    
  private void parseArgument(String arg) throws ArgsException {  
    if (arg.startsWith(”-”))  
      parseElements(arg);  
  }  
  
  private void parseElements(String arg) throws ArgsException {  
    for (int i = 1; i < arg.length(); i++)  
      parseElement(arg.charAt(i));  
  }  
  
  private void parseElement(char argChar) throws ArgsException {  
    if (setArgument(argChar))  
      argsFound.add(argChar);  
    else {  
      unexpectedArguments.add(argChar);  
      errorCode = ErrorCode.UNEXPECTED\_ARGUMENT;  
      valid = false;  
    }  
  }  
  
  private boolean setArgument(char argChar) throws ArgsException {  
    if (isBooleanArg(argChar))  
      setBooleanArg(argChar, true);  
    else if (isStringArg(argChar))  
      setStringArg(argChar);  
    else if (isIntArg(argChar))  
      setIntArg(argChar);  
    else  
      return false;  
  
    return true;  
  }  
  
  private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}  
    
  private void setIntArg(char argChar) throws ArgsException {  
    currentArgument++;  
    String parameter = null;  
    try {  
      parameter = args[currentArgument];  
      intArgs.put(argChar, new Integer(parameter));  
    } catch (ArrayIndexOutOfBoundsException e) {  
      valid = false;  
      errorArgumentId = argChar;  
      errorCode = ErrorCode.MISSING\_INTEGER;  
  
      throw new ArgsException();  
    } catch (NumberFormatException e) {  
      valid = false;  
      errorArgumentId = argChar;  
      errorParameter = parameter;  
      errorCode = ErrorCode.INVALID\_INTEGER;  
      throw new ArgsException();  
    }  
  }  
  
  private void setStringArg(char argChar) throws ArgsException {  
    currentArgument++;  
    try {  
      stringArgs.put(argChar, args[currentArgument]);  
    } catch (ArrayIndexOutOfBoundsException e) {  
      valid = false;  
      errorArgumentId = argChar;  
      errorCode = ErrorCode.MISSING\_STRING;  
      throw new ArgsException();  
    }  
  }  
  
  private boolean isStringArg(char argChar) {  
    return stringArgs.containsKey(argChar);  
  }  
  
  private void setBooleanArg(char argChar, boolean value) {  
    booleanArgs.put(argChar, value);  
  }  
  
  private boolean isBooleanArg(char argChar) {  
    return booleanArgs.containsKey(argChar);  
  }  
  
  public int cardinality() {  
    return argsFound.size();  
  }  
  
  public String usage() {  
    if (schema.length() > 0)  
      return “-[” + schema + “]”;  
    else  
      return ””;  
  }  
  
  public String errorMessage() throws Exception {  
    switch (errorCode) {  
      case OK:  
        throw new Exception(“TILT: Should not get here.”);  
      case UNEXPECTED\_ARGUMENT:  
        return unexpectedArgumentMessage();  
      case MISSING\_STRING:  
        return String.format(“Could not find string parameter for -%c.”,  
                            errorArgumentId);  
  
      case INVALID\_INTEGER:  
        return String.format(“Argument -%c expects an integer but was ’%s’.”,   
                            errorArgumentId, errorParameter);  
      case MISSING\_INTEGER:  
        return String.format(“Could not find integer parameter for -%c.”,   
                            errorArgumentId);  
    }  
    return ””;  
  }  
  
  private String unexpectedArgumentMessage() {  
    StringBuffer message = new StringBuffer(“Argument(s) -”);  
    for (char c : unexpectedArguments) {  
      message.append(c);  
    }  
    message.append(“ unexpected.”);  
  
    return message.toString();  
  }  
  
  private boolean falseIfNull(Boolean b) {  
    return b != null && b;  
  }  
  
  private int zeroIfNull(Integer i) {  
    return i == null ? 0 : i;  
  }  
  
  private String blankIfNull(String s) {  
    return s == null ? ”” : s;  
  }  
  
  public String getString(char arg) {  
    return blankIfNull(stringArgs.get(arg));  
  }  
  
  public int getInt(char arg) {  
    return zeroIfNull(intArgs.get(arg));  
  }  
  
  public boolean getBoolean(char arg) {  
    return falseIfNull(booleanArgs.get(arg));  
  }  
  
  public boolean has(char arg) {  
    return argsFound.contains(arg);  
  }  
    
  public boolean isValid() {  
    return valid;  
  }  
  
  private class ArgsException extends Exception {  
  }  
}

I hope your initial reaction to this mass of code is “I’m certainly glad he didn’t leave it like that!” If you feel like this, then remember that’s how other people are going to feel about code that you leave in rough-draft form.

Actually “rough draft” is probably the kindest thing you can say about this code. It’s clearly a work in progress. The sheer number of instance variables is daunting. The odd strings like “TILT,” the HashSets and TreeSets, and the try-catch-catch blocks all add up to a festering pile.

I had not wanted to write a festering pile. Indeed, I was trying to keep things reasonably well organized. You can probably tell that from my choice of function and variable names and the fact that there is a crude structure to the program. But, clearly, I had let the problem get away from me.

The mess built gradually. Earlier versions had not been nearly so nasty. For example, [Listing 14-9](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt9)shows an earlier version in which only Boolean arguments were working.

**Listing 14-9 Args.java (Boolean only)**

package com.objectmentor.utilities.getopts;  
  
import java.util.\*;  
  
public class Args {  
  private String schema;  
  private String[] args;  
  private boolean valid;  
  private Set<Character> unexpectedArguments = new TreeSet<Character>();  
  private Map<Character, Boolean> booleanArgs =   
    new HashMap<Character, Boolean>();  
  private int numberOfArguments = 0;  
  
  public Args(String schema, String[] args) {  
    this.schema = schema;  
    this.args = args;  
    valid = parse();  
  }  
  
  public boolean isValid() {  
    return valid;  
  }  
  
  private boolean parse() {  
    if (schema.length() == 0 && args.length == 0)  
      return true;  
    parseSchema();  
    parseArguments();  
    return unexpectedArguments.size() == 0;  
  }  
  
  private boolean parseSchema() {  
    for (String element : schema.split(”,”)) {  
      parseSchemaElement(element);  
    }  
    return true;  
  }  
    
  private void parseSchemaElement(String element) {  
    if (element.length() == 1) {  
      parseBooleanSchemaElement(element);  
    }  
  }  
    
  private void parseBooleanSchemaElement(String element) {  
    char c = element.charAt(0);  
    if (Character.isLetter(c)) {  
      booleanArgs.put(c, false);  
    }  
  }  
  
  private boolean parseArguments() {  
    for (String arg : args)  
      parseArgument(arg);  
    return true;  
  }  
  
  private void parseArgument(String arg) {  
    if (arg.startsWith(”-”))  
      parseElements(arg);  
  }  
  
  private void parseElements(String arg) {  
    for (int i = 1; i < arg.length(); i++)  
      parseElement(arg.charAt(i));  
  }  
    
  private void parseElement(char argChar) {  
    if (isBoolean(argChar)) {  
      numberOfArguments++;  
      setBooleanArg(argChar, true);  
    } else  
      unexpectedArguments.add(argChar);  
  }  
    
  private void setBooleanArg(char argChar, boolean value) {  
    booleanArgs.put(argChar, value);  
  }  
    
  private boolean isBoolean(char argChar) {  
    return booleanArgs.containsKey(argChar);  
  }  
    
  public int cardinality() {  
    return numberOfArguments;  
  }  
  
  public String usage() {  
    if (schema.length() > 0)  
       return ”-[“+schema+”]”;    else  
      return ””;  
  }  
    
  public String errorMessage() {  
    if (unexpectedArguments.size() > 0) {  
      return unexpectedArgumentMessage();  
    } else  
      return ””;  
  }  
    
  private String unexpectedArgumentMessage() {  
    StringBuffer message = new StringBuffer(“Argument(s) -”);  
    for (char c : unexpectedArguments) {  
      message.append(c);  
    }  
    message.append(“ unexpected.”);  
  
    return message.toString();  
  }  
  public boolean getBoolean(char arg) {  
    return booleanArgs.get(arg);  
  }  
}

Although you can find plenty to complain about in this code, it’s really not that bad. It’s compact and simple and easy to understand. However, within this code it is easy to see the seeds of the later festering pile. It’s quite clear how this grew into the latter mess.

Notice that the latter mess has only two more argument types than this: String and integer. The addition of just two more argument types had a massively negative impact on the code. It converted it from something that would have been reasonably maintainable into something that I would expect to become riddled with bugs and warts.

I added the two argument types incrementally. First, I added the String argument, which yielded this:

**Listing 14-10 Args.java (Boolean and String)**

   package com.objectmentor.utilities.getopts;  
  
   import java.text.ParseException;  
   import java.util.\*;  
  
   public class Args {  
     private String schema;  
     private String[] args;  
     private boolean valid = true;  
     private Set<Character> unexpectedArguments = new TreeSet<Character>();  
     private Map<Character, Boolean> booleanArgs =  
       new HashMap<Character, Boolean>();  
     private Map<Character, String> stringArgs =  
       new HashMap<Character, String>();  
     private Set<Character> argsFound = new HashSet<Character>();  
     private int currentArgument;  
     private char errorArgument = '\0';  
  
     enum ErrorCode {  
       OK, MISSING\_STRING}  
  
     private ErrorCode errorCode = ErrorCode.OK;  
  
     public Args(String schema, String[] args) throws ParseException {  
       this.schema = schema;  
       this.args = args;  
       valid = parse();  
     }  
  
     private boolean parse() throws ParseException {  
       if (schema.length() == 0 && args.length == 0)  
         return true;  
       parseSchema();  
       parseArguments();  
       return valid;  
     }  
  
     private boolean parseSchema() throws ParseException {  
       for (String element : schema.split(“,”)) {  
         if (element.length() > 0) {  
           String trimmedElement = element.trim();  
           parseSchemaElement(trimmedElement);  
         }  
       }  
       return true;  
     }  
  
     private void parseSchemaElement(String element) throws ParseException {  
       char elementId = element.charAt(0);  
       String elementTail = element.substring(1);  
       validateSchemaElementId(elementId);  
       if (isBooleanSchemaElement(elementTail))  
         parseBooleanSchemaElement(elementId);  
       else if (isStringSchemaElement(elementTail))  
         parseStringSchemaElement(elementId);  
     }  
  
     private void validateSchemaElementId(char elementId) throws ParseException {  
       if (!Character.isLetter(elementId)) {  
         throw new ParseException(  
           “Bad character:” + elementId + “in Args format: ” + schema, 0);  
       }  
     }  
  
     private void parseStringSchemaElement(char elementId) {  
       stringArgs.put(elementId, “ ”);  
     }  
  
  
     private boolean isStringSchemaElement(String elementTail) {  
       return elementTail.equals(“\*”);  
     }  
  
     private boolean isBooleanSchemaElement(String elementTail) {  
       return elementTail.length() == 0;  
     }  
  
     private void parseBooleanSchemaElement(char elementId) {  
       booleanArgs.put(elementId, false);  
     }  
  
     private boolean parseArguments() {  
       for (currentArgument = 0; currentArgument < args.length; currentArgument++)  
       {  
         String arg = args[currentArgument];  
         parseArgument(arg);  
       }  
       return true;  
     }  
  
     private void parseArgument(String arg) {  
       if (arg.startsWith(“-”))  
         parseElements(arg);  
     }  
  
     private void parseElements(String arg) {  
       for (int i = 1; i < arg.length(); i++)  
         parseElement(arg.charAt(i));  
     }  
  
     private void parseElement(char argChar) {  
       if (setArgument(argChar))  
         argsFound.add(argChar);  
       else {  
         unexpectedArguments.add(argChar);  
         valid = false;  
       }  
     }  
  
     private boolean setArgument(char argChar) {  
       boolean set = true;  
       if (isBoolean(argChar))  
         setBooleanArg(argChar, true);  
       else if (isString(argChar))  
         setStringArg(argChar, “ ”);  
       else  
         set = false;  
  
       return set;  
     }  
  
     private void setStringArg(char argChar, String s) {  
       currentArgument++;  
       try {  
         stringArgs.put(argChar, args[currentArgument]);  
       } catch (ArrayIndexOutOfBoundsException e) {  
         valid = false;  
         errorArgument = argChar;  
         errorCode = ErrorCode.MISSING\_STRING;  
       }  
     }  
  
     private boolean isString(char argChar) {  
       return stringArgs.containsKey(argChar);  
     }  
  
     private void setBooleanArg(char argChar, boolean value) {  
       booleanArgs.put(argChar, value);  
     }  
  
     private boolean isBoolean(char argChar) {  
       return booleanArgs.containsKey(argChar);  
     }  
  
     public int cardinality() {  
       return argsFound.size();  
     }  
  
     public String usage() {  
       if (schema.length() > 0)  
         return “-[“ + schema + ”]”;  
       else  
         return “ ”;  
     }  
  
     public String errorMessage() throws Exception {  
       if (unexpectedArguments.size() > 0) {  
         return unexpectedArgumentMessage();  
       } else  
         switch (errorCode) {  
           case MISSING\_STRING:  
             return String.format(“Could not find string parameter for -%c.”, errorArgument);  
           case OK:  
             throw new Exception(“TILT: Should not get here.”);  
         }  
       return “ ”;  
     }  
  
     private String unexpectedArgumentMessage() {  
       StringBuffer message = new StringBuffer(“Argument(s) -”);  
       for (char c : unexpectedArguments) {  
         message.append(c);  
       }  
       message.append(“ unexpected.”);  
  
       return message.toString();  
     }  
  
     public boolean getBoolean(char arg) {  
       return falseIfNull(booleanArgs.get(arg));  
     }  
  
     private boolean falseIfNull(Boolean b) {  
       return b == null ? false : b;  
     }  
  
     public String getString(char arg) {  
       return blankIfNull(stringArgs.get(arg));  
     }  
  
     private String blankIfNull(String s) {  
       return s == null ? “ ” : s;  
     }  
  
     public boolean has(char arg) {  
       return argsFound.contains(arg);  
     }  
  
     public boolean isValid() {  
       return valid;  
     }  
   }

You can see that this is starting to get out of hand. It’s still not horrible, but the mess is certainly starting to grow. It’s a pile, but it’s not festering quite yet. It took the addition of the integer argument type to get this pile really fermenting and festering.

#### **So I Stopped**

I had at least two more argument types to add, and I could tell that they would make things much worse. If I bulldozed my way forward, I could probably get them to work, but I’d leave behind a mess that was too large to fix. If the structure of this code was ever going to be maintainable, now was the time to fix it.

So I stopped adding features and started refactoring. Having just added the String and integerarguments, I knew that each argument type required new code in three major places. First, each argument type required some way to parse its schema element in order to select the HashMap for that type. Next, each argument type needed to be parsed in the command-line strings and converted to its true type. Finally, each argument type needed a getXXX method so that it could be returned to the caller as its true type.

Many different types, all with similar methods—that sounds like a class to me. And so the ArgumentMarshaler concept was born.

#### **On Incrementalism**

One of the best ways to ruin a program is to make massive changes to its structure in the name of improvement. Some programs never recover from such “improvements.” The problem is that it’s very hard to get the program working the same way it worked before the “improvement.”

### **ARGS: THE ROUGH DRAFT**

To avoid this, I use the discipline of Test-Driven Development (TDD). One of the central doctrines of this approach is to keep the system running at all times. In other words, using TDD, I am not allowed to make a change to the system that breaks that system. Every change I make must keep the system working as it worked before.

To achieve this, I need a suite of automated tests that I can run on a whim and that verifies that the behavior of the system is unchanged. For the Args class I had created a suite of unit and acceptance tests while I was building the festering pile. The unit tests were written in Java and administered by JUnit. The acceptance tests were written as wiki pages in FitNesse. I could run these tests any time I wanted, and if they passed, I was confident that the system was working as I specified.

So I proceeded to make a large number of very tiny changes. Each change moved the structure of the system toward the ArgumentMarshaler concept. And yet each change kept the system working. The first change I made was to add the skeleton of the ArgumentMarshaller to the end of the festering pile ([Listing 14-11](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt11)).

**Listing 14-11 ArgumentMarshaller appended to Args.java**

   private class ArgumentMarshaler }  
       private boolean booleanValue = false;  
  
  
       public void setBoolean(boolean value) {  
         booleanValue = value;  
       }  
  
       public boolean getBoolean() {return booleanValue;}  
     }  
  
     private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
     }  
  
     private class StringArgumentMarshaler extends ArgumentMarshaler {  
     }  
  
     private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
     }  
   }

Clearly, this wasn’t going to break anything. So then I made the simplest modification I could, one that would break as little as possible. I changed the HashMap for the Boolean arguments to take an ArgumentMarshaler.

   private Map<Character, **ArgumentMarshaler**> booleanArgs =  
     new HashMap<Character, **ArgumentMarshaler**>();

This broke a few statements, which I quickly fixed.

   …  
     private void parseBooleanSchemaElement(char elementId) {  
       booleanArgs.put(elementId, **new BooleanArgumentMarshaler()**);  
     }  
   ..  
     private void setBooleanArg(char argChar, boolean value) {  
       booleanArgs.**get**(argChar)**.setBoolean**(value);  
     }  
   …  
     public boolean getBoolean(char arg) {  
       return falseIfNull(booleanArgs.get(arg).**getBoolean()**);  
   }

Notice how these changes are in exactly the areas that I mentioned before: the parse, set, and getfor the argument type. Unfortunately, small as this change was, some of the tests started failing. If you look carefully at getBoolean, you’ll see that if you call it with 'y,' but there is no y argument, then booleanArgs.get('y') will return null, and the function will throw a NullPointerException. The falseIfNull function had been used to protect against this, but the change I made caused that function to become irrelevant.

Incrementalism demanded that I get this working quickly before making any other changes. Indeed, the fix was not too difficult. I just had to move the check for null. It was no longer the boolean being null that I needed to check; it was the ArgumentMarshaller.

First, I removed the falseIfNull call in the getBoolean function. It was useless now, so I also eliminated the function itself. The tests still failed in the same way, so I was confident that I hadn’t introduced any new errors.

   public boolean getBoolean(char arg) {  
     return booleanArgs.get(arg).getBoolean();  
   }

Next, I split the function into two lines and put the ArgumentMarshaller into its own variable named argumentMarshaller. I didn’t care for the long variable name; it was badly redundant and cluttered up the function. So I shortened it to am [N5].

   public boolean getBoolean(char arg) {  
     Args.ArgumentMarshaler **am** = booleanArgs.get(arg);  
     return **am**.getBoolean();  
   }

And then I put in the null detection logic.

   public boolean getBoolean(char arg) {  
     Args.ArgumentMarshaler am = booleanArgs.get(arg);  
     return **am != null &&** am.getBoolean();  
   }

### **STRING ARGUMENTS**

Addin\_g String arguments was very similar to adding boolean arguments. I had to change the HashMap and get the parse, set, and get functions working. There shouldn’t be any surprises in what follows except, perhaps, that I seem to be putting all the marshalling implementation in the ArgumentMarshaller base class instead of distributing it to the derivatives.

   private Map<Character, **ArgumentMarshaler**> stringArgs =  
       new HashMap<Character, **ArgumentMarshaler**>();  
  …  
   private void parseStringSchemaElement(char elementId) {  
     stringArgs.put(elementId**, new StringArgumentMarshaler()**);  
   }  
  …  
   private void setStringArg(char argChar) throws ArgsException {  
    currentArgument++;  
    try {  
      stringArgs**.get**(argChar)**.setString**(args[currentArgument]);  
    } catch (ArrayIndexOutOfBoundsException e) {  
      valid = false;  
      errorArgumentId = argChar;  
      errorCode = ErrorCode.MISSING\_STRING;  
      throw new ArgsException();  
    }  
   }  
  …  
   public String getString(char arg) {  
     **Args.ArgumentMarshaler am =** stringArgs.get(arg);  
     return **am == null ? “ ”** : am.getString();  
   }  
  …  
  private class ArgumentMarshaler {  
    private boolean booleanValue = false;  
    **private String stringValue;**  
  
    public void setBoolean(boolean value) {  
      booleanValue = value;  
    }  
  
    public boolean getBoolean() {  
      return booleanValue;  
    }  
  
    **public void setString(String s) {**  
      **stringValue = s;**  
    }  
    **public String getString() {**  
      **return stringValue == null ? “ ” : stringValue;**  
    **}**  
  }

Again, these changes were made one at a time and in such a way that the tests kept running, if not passing. When a test broke, I made sure to get it passing again before continuing with the next change.

By now you should be able to see my intent. Once I get all the current marshalling behavior into the ArgumentMarshaler base class, I’m going to start pushing that behavior down into the derivatives. This will allow me to keep everything running while I gradually change the shape of this program.

The obvious next step was to move the int argument functionality into the ArgumentMarshaler. Again, there weren’t any surprises.

   private Map<Character, **ArgumentMarshaler**> intArgs =  
        new HashMap<Character, **ArgumentMarshaler**>();  
   …  
   private void parseIntegerSchemaElement(char elementId) {  
     intArgs.put(elementId, **new IntegerArgumentMarshaler()**);  
   }  
  …  
  
   private void setIntArg(char argChar) throws ArgsException {  
     currentArgument++;  
     String parameter = null;  
     try {  
       parameter = args[currentArgument];  
       intArgs**.get**(argChar)**.setInteger**(Integer.parseInt(parameter));  
     } catch (ArrayIndexOutOfBoundsException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       errorCode = ErrorCode.MISSING\_INTEGER;  
       throw new ArgsException();  
     } catch (NumberFormatException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       errorParameter = parameter;  
       errorCode = ErrorCode.INVALID\_INTEGER;  
       throw new ArgsException();  
     }  
   }  
…  
   public int getInt(char arg) {  
     **Args.ArgumentMarshaler am =** intArgs.get(arg);  
     return **am == null ? 0** : am.getInteger();  
   }  
  …  
   private class ArgumentMarshaler {  
     private boolean booleanValue = false;  
     private String stringValue;  
     **private int integerValue;**  
     public void setBoolean(boolean value) {  
       booleanValue = value;  
     }  
  
     public boolean getBoolean() {  
      return booleanValue;  
     }  
  
     public void setString(String s) {  
       stringValue = s;  
     }  
    
     public String getString() {  
       return stringValue == null ? “ ”: stringValue;  
     }  
     **public void setInteger(int i) {**  
       **integerValue = i;**  
     **}**  
     **public int getInteger() {**  
       **return integerValue;**  
     **}**  
   }

With all the marshalling moved to the ArgumentMarshaler, I started pushing functionality into the derivatives. The first step was to move the setBoolean function into the BooleanArgumentMarshaller and make sure it got called correctly. So I created an abstract setmethod.

   private **abstract** class ArgumentMarshaler {  
     **protected** boolean booleanValue = false;  
     private String stringValue;  
     private int integerValue;  
  
     public void setBoolean(boolean value) {  
       booleanValue = value;  
     }  
     public boolean getBoolean() {  
       return booleanValue;  
     }  
  
     public void setString(String s) {  
       stringValue = s;  
     }  
  
     public String getString() {  
       return stringValue == null ? “ ” : stringValue;  
     }  
  
     public void setInteger(int i) {  
       integerValue = i;  
     }  
  
     public int getInteger() {  
       return integerValue;  
     }  
  
     **public abstract void set(String s);**  
   **}**

Then I implemented the set method in BooleanArgumentMarshaller.

   private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
     **public void set(String s) {**  
       **booleanValue = true;**  
     **}**  
   }

And finally I replaced the call to setBoolean with a call to set.

   private void setBooleanArg(char argChar, boolean value) {  
       booleanArgs.get(argChar) **.set(“true”);**  
      }

The tests all still passed. Because this change caused set to be deployed to the Boolean-ArgumentMarshaler, I removed the setBoolean method from the ArgumentMarshaler base class.

Notice that the abstract set function takes a String argument, but the implementation in the BooleanArgumentMarshaller does not use it. I put that argument in there because I knew that the StringArgumentMarshaller and IntegerArgumentMarshaller would use it.

Next, I wanted to deploy the get method into BooleanArgumentMarshaler. Deploying getfunctions is always ugly because the return type has to be Object, and in this case needs to be cast to a Boolean.

   public boolean getBoolean(char arg) {  
       Args.ArgumentMarshaler am = booleanArgs.get(arg);  
       return am != null && **(Boolean)**am.**get**();  
     }

Just to get this to compile, I added the get function to the ArgumentMarshaler.

   private abstract class ArgumentMarshaler {  
       …  
  
       **public Object get() {**  
         **return null;**  
       **}**  
   }

This compiled and obviously failed the tests. Getting the tests working again was simply a matter of making get abstract and implementing it in BooleanAgumentMarshaler.

   private abstract class ArgumentMarshaler {  
       protected boolean booleanValue = false;  
       …  
  
       public **abstract** Object get();  
     }  
  
     private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
       public void set(String s) {  
         booleanValue = true;  
       }  
  
       **public Object get() {**  
         **return booleanValue;**  
       **}**  
   }

Once again the tests passed. So both get and set deploy to the BooleanArgumentMarshaler! This allowed me to remove the old getBoolean function from ArgumentMarshaler, move the protected booleanValue variable down to BooleanArgumentMarshaler, and make it private.

I did the same pattern of changes for Strings. I deployed both set and get, deleted the unused functions, and moved the variables.

   private void setStringArg(char argChar) throws ArgsException {  
     currentArgument++;  
     try {  
       stringArgs.get(argChar).**set**(args[currentArgument]);  
     } catch (ArrayIndexOutOfBoundsException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       errorCode = ErrorCode.MISSING\_STRING;  
       throw new ArgsException();  
     }  
   }  
   …  
     public String getString(char arg) {  
       Args.ArgumentMarshaler am = stringArgs.get(arg);  
       return am == null ? “ ” : **(String)** am.**get**();  
     }  
   …  
     private abstract class ArgumentMarshaler {  
       private int integerValue;  
  
       public void setInteger(int i) {  
         integerValue = i;  
       }  
  
       public int getInteger() {  
         return integerValue;  
       }  
  
       public abstract void set(String s);  
  
       public abstract Object get();  
     }  
  
     private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
       **private boolean booleanValue = false;**  
  
       public void set(String s) {  
         booleanValue = true;  
       }  
  
       public Object get() {  
         return booleanValue;  
       }  
     }  
  
     private class StringArgumentMarshaler extends ArgumentMarshaler {  
       **private String stringValue = “ ”;**  
  
       public void set(String s) {  
         **stringValue = s;**  
       }  
  
       public Object get() {  
         **return stringValue;**  
       }  
     }  
  
     private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
  
       public void set(String s) {  
       }  
  
       public Object get() {  
         return null;  
       }  
     }  
   }

Finally, I repeated the process for integers. This was just a little more complicated because integers needed to be parsed, and the parse operation can throw an exception. But the result is better because the whole concept of NumberFormatException got buried in the IntegerArgumentMarshaler.

   private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}  
  
     private void setIntArg(char argChar) throws ArgsException {  
       currentArgument++;  
       String parameter = null;  
       try {  
         parameter = args[currentArgument];  
         intArgs.get(argChar).**set**(parameter);  
       } catch (ArrayIndexOutOfBoundsException e) {  
         valid = false;  
         errorArgumentId = argChar;  
         errorCode = ErrorCode.MISSING\_INTEGER;  
         throw new ArgsException();  
       } catch (**ArgsException** e) {  
         valid = false;  
         errorArgumentId = argChar;  
         errorParameter = parameter;  
         errorCode = ErrorCode.INVALID\_INTEGER;  
         throw **e**;  
       }  
     }  
   …  
     private void setBooleanArg(char argChar) {  
       **try {**  
                 booleanArgs.get(argChar).set(“true”);  
       **} catch (ArgsException e) {**  
       **}**  
     }  
   …  
     public int getInt(char arg) {  
       Args.ArgumentMarshaler am = intArgs.get(arg);  
       return am == null ? 0 : **(Integer)** am.**get**();  
     }  
   …  
     private abstract class ArgumentMarshaler {  
       public abstract void set(String s) throws ArgsException;  
       public abstract Object get();  
     }  
   …  
     private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
       **private int intValue = 0;**  
  
       public void set(String s) **throws ArgsException {**  
         **try {**  
           **intValue = Integer.parseInt(s);**  
         **} catch (NumberFormatException e) {**  
           **throw new ArgsException();**  
         **}**  
       }  
  
       public Object get() {  
         **return intValue;**  
       }  
     }

Of course, the tests continued to pass. Next, I got rid of the three different maps up at the top of the algorithm. This made the whole system much more generic. However, I couldn’t get rid of them just by deleting them because that would break the system. Instead, I added a new Map for the ArgumentMarshaler and then one by one changed the methods to use it instead of the three original maps.

   public class Args {  
   …  
     private Map<Character, ArgumentMarshaler> booleanArgs =  
       new HashMap<Character, ArgumentMarshaler>();  
     private Map<Character, ArgumentMarshaler> stringArgs =  
       new HashMap<Character, ArgumentMarshaler>();  
     private Map<Character, ArgumentMarshaler> intArgs =  
       new HashMap<Character, ArgumentMarshaler>();  
     **private Map<Character, ArgumentMarshaler> marshalers =**   
       **new HashMap<Character, ArgumentMarshaler>();**  
   …  
     private void parseBooleanSchemaElement(char elementId) {  
       **ArgumentMarshaler m = new BooleanArgumentMarshaler();**  
       booleanArgs.put(elementId, m);  
       **marshalers.put(elementId, m);**  
     }  
  
     private void parseIntegerSchemaElement(char elementId) {  
       **ArgumentMarshaler m = new IntegerArgumentMarshaler();**  
       intArgs.put(elementId, m);  
       **marshalers.put(elementId, m);**  
     }  
  
     private void parseStringSchemaElement(char elementId) {  
       **ArgumentMarshaler m = new StringArgumentMarshaler();**  
       stringArgs.put(elementId, m);  
       **marshalers.put(elementId, m);**  
     }

Of course the tests all still passed. Next, I changed isBooleanArg from this:

   private boolean isBooleanArg(char argChar) {  
     return booleanArgs.containsKey(argChar);  
   }

to this:

   private boolean isBooleanArg(char argChar) {  
     **ArgumentMarshaler m = marshalers.get(argChar);**  
     **return m instanceof BooleanArgumentMarshaler;**  
   }

The tests still passed. So I made the same change to isIntArg and isStringArg.

   private boolean isIntArg(char argChar) {  
     **ArgumentMarshaler m = marshalers.get(argChar);**  
     **return m instanceof IntegerArgumentMarshaler;**  
   }  
  
   private boolean isStringArg(char argChar) {  
     **ArgumentMarshaler m = marshalers.get(argChar);**  
     **return m instanceof StringArgumentMarshaler;**  
   }

The tests still passed. So I eliminated all the duplicate calls to marshalers.get as follows:

   private boolean setArgument(char argChar) throws ArgsException {  
     **ArgumentMarshaler m = marshalers.get(argChar);**  
     if (isBooleanArg(**m**))  
       setBooleanArg(argChar);  
     else if (isStringArg(**m**))  
       setStringArg(argChar);  
     else if (isIntArg(**m**))  
       setIntArg(argChar);  
     else  
       return false;  
  
     return true;  
   }  
  
   private boolean isIntArg(**ArgumentMarshaler m**) {  
     return m instanceof IntegerArgumentMarshaler;  
   }  
  
   private boolean isStringArg(**ArgumentMarshaler m**) {  
     return m instanceof StringArgumentMarshaler;  
   }  
  
   private boolean isBooleanArg(**ArgumentMarshaler m**) {  
     return m instanceof BooleanArgumentMarshaler;  
   }

This left no good reason for the three isxxxArg methods. So I inlined them:

   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     if (**m instanceof BooleanArgumentMarshaler**)  
       setBooleanArg(argChar);  
     else if (**m instanceof StringArgumentMarshaler**)  
       setStringArg(argChar);  
     else if (**m instanceof IntegerArgumentMarshaler**)  
       setIntArg(argChar);  
     else  
       return false;  
  
     return true;  
   }

Next, I started using the marshalers map in the set functions, breaking the use of the other three maps. I started with the booleans.

   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     if (m instanceof BooleanArgumentMarshaler)  
       setBooleanArg(**m**);  
     else if (m instanceof StringArgumentMarshaler)  
       setStringArg(argChar);  
     else if (m instanceof IntegerArgumentMarshaler)  
       setIntArg(argChar);  
     else  
       return false;  
       return true;  
     }  
   …  
     private void setBooleanArg(**ArgumentMarshaler m**) {  
       try {  
         **m**.set(“true”); **// was: booleanArgs.get(argChar).set(“true”);**  
       } catch (ArgsException e) {  
       }  
     }

The tests still passed, so I did the same with Strings and Integers. This allowed me to integrate some of the ugly exception management code into the setArgument function.

   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     **try {**  
       if (m instanceof BooleanArgumentMarshaler)  
         setBooleanArg(m);  
       else if (m instanceof StringArgumentMarshaler)  
         setStringArg(**m**);  
       else if (m instanceof IntegerArgumentMarshaler)  
         setIntArg(**m**);  
       else  
         return false;  
     **} catch (ArgsException e) {**  
       **valid = false;**  
       **errorArgumentId = argChar;**  
       **throw e;**  
     **}**  
     return true;  
   }  
  
   private void setIntArg(**ArgumentMarshaler m**) throws ArgsException {  
     currentArgument++;  
     String parameter = null;  
     try {  
       parameter = args[currentArgument];  
       **m**.set(parameter);  
     } catch (ArrayIndexOutOfBoundsException e) {  
       errorCode = ErrorCode.MISSING\_INTEGER;  
       throw new ArgsException();  
     } catch (ArgsException e) {  
       errorParameter = parameter;  
       errorCode = ErrorCode.INVALID\_INTEGER;  
       throw e;  
     }  
   }  
  
   private void setStringArg(**ArgumentMarshaler m**) throws ArgsException {  
     currentArgument++;  
     try {  
       **m**.set(args[currentArgument]);  
     } catch (ArrayIndexOutOfBoundsException e) {  
       errorCode = ErrorCode.MISSING\_STRING;  
       throw new ArgsException();  
     }  
   }

I was close to being able to remove the three old maps. First, I needed to change the getBooleanfunction from this:

   public boolean getBoolean(char arg) {  
     Args.ArgumentMarshaler am = booleanArgs.get(arg);  
     return am != null && (Boolean) am.get();  
   }

to this:

   public boolean getBoolean(char arg) {  
       Args.ArgumentMarshaler am = marshalers.get(arg);  
       **boolean b = false;**  
       **try {**  
         **b =**   am != null && (Boolean) am.get();  
       **} catch (ClassCastException e) {**  
  
         **b = false;**  
       **}**  
       **return b;**  
   }

This last change might have been a surprise. Why did I suddenly decide to deal with the ClassCastException? The reason is that I have a set of unit tests and a separate set of acceptance tests written in FitNesse. It turns out that the FitNesse tests made sure that if you called getBooleanon a nonboolean argument, you got a false. The unit tests did not. Up to this point I had only been running the unit tests.2

2. To prevent further surprises of this kind, I added a new unit test that invoked all the FitNesse tests.

This last change allowed me to pull out another use of the boolean map:

   private void parseBooleanSchemaElement(char elementId) {  
     ArgumentMarshaler m = new BooleanArgumentMarshaler();  
     **booleanArgs.put(elementId, m);**  
     marshalers.put(elementId, m);  
   }

And now we can delete the boolean map.

   public class Args {  
   …  
     **private Map<Character, ArgumentMarshaler> booleanArgs  
     = new HashMap<Character, ArgumentMarshaler>();**  
     private Map<Character, ArgumentMarshaler> stringArgs =  
     new HashMap<Character, ArgumentMarshaler>();  
     private Map<Character, ArgumentMarshaler> intArgs =  
     new HashMap<Character, ArgumentMarshaler>();  
     private Map<Character, ArgumentMarshaler> marshalers =  
     new HashMap<Character, ArgumentMarshaler>();  
   …

Next, I migrated the String and Integer arguments in the same manner and did a little cleanup with the booleans.

   private void parseBooleanSchemaElement(char elementId) {  
     marshalers.put(elementId, **new BooleanArgumentMarshaler()**);  
   }  
   private void parseIntegerSchemaElement(char elementId) {  
     marshalers.put(elementId, **new IntegerArgumentMarshaler()**);  
   }  
  
   private void parseStringSchemaElement(char elementId) {  
     marshalers.put(elementId, **new StringArgumentMarshaler()**);  
   }  
  …  
   public String getString(char arg) {  
     Args.ArgumentMarshaler am = **marshalers**.get(arg);  
     **try {**  
       return am == null ? “ ” : (String) am.get();  
     **} catch (ClassCastException e) {**  
       **return “ ”;**  
     **}**  
   }  
  
   public int getInt(char arg) {  
     Args.ArgumentMarshaler am = **marshalers**.get(arg);  
     **try {**  
       return am == null ? 0 : (Integer) am.get();  
     **} catch (Exception e) {**  
       **return 0;**  
     **}**  
   }  
…  
public class Args {  
…  
   **private Map<Character, ArgumentMarshaler> stringArgs =**  
   **new HashMap<Character, ArgumentMarshaler>();**  
   **private Map<Character, ArgumentMarshaler> intArgs =**  
   **new HashMap<Character, ArgumentMarshaler>();**  
   private Map<Character, ArgumentMarshaler> marshalers =  
   new HashMap<Character, ArgumentMarshaler>();  
  …

Next, I inlined the three parse methods because they didn’t do much anymore:

   private void parseSchemaElement(String element) throws ParseException {  
     char elementId = element.charAt(0);  
     String elementTail = element.substring(1);  
     validateSchemaElementId(elementId);  
     if (isBooleanSchemaElement(elementTail))  
       **marshalers.put(elementId, new BooleanArgumentMarshaler());**  
     else if (isStringSchemaElement(elementTail))  
       **marshalers.put(elementId, new StringArgumentMarshaler());**  
     else if (isIntegerSchemaElement(elementTail)) {  
       **marshalers.put(elementId, new IntegerArgumentMarshaler());**  
     } else {  
       throw new ParseException(String.format(  
       “Argument: %c has invalid format: %s.”, elementId, elementTail), 0);  
     }  
   }

Okay, so now let’s look at the whole picture again. [Listing 14-12](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt12) shows the current form of the Argsclass.

**Listing 14-12 Args.java (After first refactoring)**

   package com.objectmentor.utilities.getopts;  
  
  
   import java.text.ParseException;  
   import java.util.\*;  
  
   public class Args {  
     private String schema;  
     private String[] args;  
     private boolean valid = true;  
     private Set<Character> unexpectedArguments = new TreeSet<Character>();  
     private Map<Character, ArgumentMarshaler> marshalers =  
      new HashMap<Character, ArgumentMarshaler>();  
     private Set<Character> argsFound = new HashSet<Character>();  
     private int currentArgument;  
     private char errorArgumentId = '\0';  
     private String errorParameter = “TILT”;  
     private ErrorCode errorCode = ErrorCode.OK;  
  
  
     private enum ErrorCode {  
       OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER,  
         UNEXPECTED\_ARGUMENT}  
  
     public Args(String schema, String[] args) throws ParseException {  
       this.schema = schema;  
       this.args = args;  
       valid = parse();  
     }  
  
   private boolean parse() throws ParseException {  
     if (schema.length() == 0 && args.length == 0)  
       return true;  
     parseSchema();  
     try {  
       parseArguments();  
     } catch (ArgsException e) {  
     }  
     return valid;  
   }  
  
   private boolean parseSchema() throws ParseException {  
     for (String element : schema.split(“,”)) {  
       if (element.length() > 0) {  
         String trimmedElement = element.trim();  
         parseSchemaElement(trimmedElement);  
       }  
     }  
     return true;  
   }  
  
   private void parseSchemaElement(String element) throws ParseException {  
     char elementId = element.charAt(0);  
     String elementTail = element.substring(1);  
     validateSchemaElementId(elementId);  
     if (isBooleanSchemaElement(elementTail))  
       marshalers.put(elementId, new BooleanArgumentMarshaler());  
     else if (isStringSchemaElement(elementTail))  
       marshalers.put(elementId, new StringArgumentMarshaler());  
     else if (isIntegerSchemaElement(elementTail)) {  
       marshalers.put(elementId, new IntegerArgumentMarshaler());  
     } else {  
       throw new ParseException(String.format(  
       “Argument: %c has invalid format: %s.”, elementId, elementTail), 0);  
     }  
   }  
  
   private void validateSchemaElementId(char elementId) throws ParseException {  
     if (!Character.isLetter(elementId)) {  
       throw new ParseException(  
       “Bad character:” + elementId + “in Args format: ” + schema, 0);  
     }  
   }  
  
   private boolean isStringSchemaElement(String elementTail) {  
     return elementTail.equals(“\*”);  
   }  
  
   private boolean isBooleanSchemaElement(String elementTail) {  
     return elementTail.length() == 0;  
   }  
  
   private boolean isIntegerSchemaElement(String elementTail) {  
     return elementTail.equals(“-”);  
   }  
  
   private boolean parseArguments() throws ArgsException {  
     for (currentArgument=0; currentArgument<args.length; currentArgument++) {  
       String arg = args[currentArgument];  
       parseArgument(arg);  
     }  
     return true;  
   }  
  
   private void parseArgument(String arg) throws ArgsException {  
     if (arg.startsWith(“-”))  
       parseElements(arg);  
   }  
  
   private void parseElements(String arg) throws ArgsException {  
     for (int i = 1; i < arg.length(); i++)  
       parseElement(arg.charAt(i));  
   }  
  
   private void parseElement(char argChar) throws ArgsException {  
     if (setArgument(argChar))  
       argsFound.add(argChar);  
     else {  
       unexpectedArguments.add(argChar);  
       errorCode = ErrorCode.UNEXPECTED\_ARGUMENT;  
       valid = false;  
     }  
   }  
  
   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     try {  
       if (m instanceof BooleanArgumentMarshaler)  
         setBooleanArg(m);  
       else if (m instanceof StringArgumentMarshaler)  
          setStringArg(m);  
       else if (m instanceof IntegerArgumentMarshaler)  
          setIntArg(m);  
       else  
          return false;  
     } catch (ArgsException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       throw e;  
     }  
     return true;  
   }  
  
   private void setIntArg(ArgumentMarshaler m) throws ArgsException {  
     currentArgument++;  
     String parameter = null;  
     try {  
       parameter = args[currentArgument];  
       m.set(parameter);  
     } catch (ArrayIndexOutOfBoundsException e) {  
       errorCode = ErrorCode.MISSING\_INTEGER;  
       throw new ArgsException();  
     } catch (ArgsException e) {  
       errorParameter = parameter;  
       errorCode = ErrorCode.INVALID\_INTEGER;  
       throw e;  
     }  
   }  
  
   private void setStringArg(ArgumentMarshaler m) throws ArgsException {  
     currentArgument++;  
     try {  
       m.set(args[currentArgument]);  
     } catch (ArrayIndexOutOfBoundsException e) {  
       errorCode = ErrorCode.MISSING\_STRING;  
       throw new ArgsException();  
     }  
   }  
  
   private void setBooleanArg(ArgumentMarshaler m) {  
     try {  
       m.set(“true”);  
     } catch (ArgsException e) {  
     }  
   }  
  
   public int cardinality() {  
     return argsFound.size();  
   }  
  
   public String usage() {  
     if (schema.length() > 0)  
       return “-[“ + schema + ”]”;  
     else  
       return “ ”;  
   }  
  
   public String errorMessage() throws Exception {  
     switch (errorCode) {  
       case OK:  
         throw new Exception(“TILT: Should not get here.”);  
       case UNEXPECTED\_ARGUMENT:  
         return unexpectedArgumentMessage();  
       case MISSING\_STRING:  
         return String.format(“Could not find string parameter for -%c.”,  
     errorArgumentId);  
       case INVALID\_INTEGER:  
         return String.format(“Argument -%c expects an integer but was '%s'.”,  
     errorArgumentId, errorParameter);  
       case MISSING\_INTEGER:  
         return String.format(“Could not find integer parameter for -%c.”,  
     errorArgumentId);  
     }  
     return “ ”;  
   }  
  
   private String unexpectedArgumentMessage() {  
     StringBuffer message = new StringBuffer(“Argument(s) -”);  
     for (char c : unexpectedArguments) {  
       message.append(c);  
     }  
     message.append(“ unexpected.”);  
  
     return message.toString();  
   }  
  
   public boolean getBoolean(char arg) {  
     Args.ArgumentMarshaler am = marshalers.get(arg);  
     boolean b = false;  
     try {  
       b = am != null && (Boolean) am.get();  
     } catch (ClassCastException e) {  
       b = false;  
     }  
     return b;  
   }  
  
   public String getString(char arg) {  
     Args.ArgumentMarshaler am = marshalers.get(arg);  
     try {  
       return am == null ? “ ” : (String) am.get();  
     } catch (ClassCastException e) {  
       return “ ”;  
     }  
   }  
  
   public int getInt(char arg) {  
     Args.ArgumentMarshaler am = marshalers.get(arg);  
     try {  
       return am == null ? 0 : (Integer) am.get();  
     } catch (Exception e) {  
       return 0;  
     }  
   }  
  
   public boolean has(char arg) {  
     return argsFound.contains(arg);  
   }  
  
   public boolean isValid() {  
     return valid;  
   }  
  
   private class ArgsException extends Exception {  
   }  
  
   private abstract class ArgumentMarshaler {  
     public abstract void set(String s) throws ArgsException;  
     public abstract Object get();  
   }  
  
   private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
     private boolean booleanValue = false;  
  
     
     public void set(String s) {  
       booleanValue = true;  
     }  
     public Object get() {  
       return booleanValue;  
     }  
   }  
  
   private class StringArgumentMarshaler extends ArgumentMarshaler {  
     private String stringValue = “ ”;  
  
     public void set(String s) {  
       stringValue = s;  
     }  
  
     public Object get() {  
       return stringValue;  
     }  
   }  
  
   private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
     private int intValue = 0;  
  
     public void set(String s) throws ArgsException {  
       try {  
         intValue = Integer.parseInt(s);  
         } catch (NumberFormatException e) {  
             throw new ArgsException();  
         }  
       }  
  
       public Object get() {  
         return intValue;  
       }  
     }  
   }

After all that work, this is a bit disappointing. The structure is a bit better, but we still have all those variables up at the top; there’s still a horrible type-case in setArgument; and all those set functions are really ugly. Not to mention all the error processing. We still have a lot of work ahead of us.

I’d really like to get rid of that type-case up in setArgument [[G23](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#ch17lev2sec34)]. What I’d like in setArgument is a single call to ArgumentMarshaler.set. This means I need to push setIntArg, setStringArg, and setBooleanArg down into the appropriate ArgumentMarshaler derivatives. But there is a problem.

If you look closely at setIntArg, you’ll notice that it uses two instance variables: args and currentArg. To move setIntArg down into BooleanArgumentMarshaler, I’ll have to pass both args and currentArgs as function arguments. That’s dirty [F1]. I’d rather pass one argument instead of two. Fortunately, there is a simple solution. We can convert the args array into a list and pass an Iterator down to the set functions. The following took me ten steps, passing all the tests after each. But I’ll just show you the result. You should be able to figure out what most of the tiny little steps were.

public class Args {  
  private String schema;  
  **private String[] args;**  
  private boolean valid = true;  
  private Set<Character> unexpectedArguments = new TreeSet<Character>();  
  private Map<Character, ArgumentMarshaler> marshalers =  
   new HashMap<Character, ArgumentMarshaler>();  
  private Set<Character> argsFound = new HashSet<Character>();  
  private **Iterator<String>** currentArgument;  
  private char errorArgumentId = ’\0’;  
  private String errorParameter = “TILT”;  
  private ErrorCode errorCode = ErrorCode.OK;  
  **private List<String> argsList;**  
  
  private enum ErrorCode {  
    OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER,  
              UNEXPECTED\_ARGUMENT}  
  
  public Args(String schema, String[] args) throws ParseException {  
    this.schema = schema;  
    **argsList = Arrays.asList(args);**  
    valid = parse();  
  }  
  private boolean parse() throws ParseException {  
    if (schema.length() == 0 && **argsList.size()** == 0)  
      return true;  
    parseSchema();  
    try {  
      parseArguments();  
    } catch (ArgsException e) {  
    }  
    return valid;  
  }  
---  
  private boolean parseArguments() throws ArgsException {  
    for (currentArgument = **argsList.iterator()**; currentArgument.**hasNext()**;) {  
      String arg = currentArgument.**next()**;  
      parseArgument(arg);  
    }  
  
    return true;  
   }  
---  
  private void setIntArg(ArgumentMarshaler m) throws ArgsException {  
    String parameter = null;  
    try {  
      parameter = currentArgument.**next()**;  
      m.set(parameter);  
    } catch (**NoSuchElementException** e) {  
      errorCode = ErrorCode.MISSING\_INTEGER;  
      throw new ArgsException();  
    } catch (ArgsException e) {  
      errorParameter = parameter;  
      errorCode = ErrorCode.INVALID\_INTEGER;  
      throw e;  
    }  
  }  
  
  private void setStringArg(ArgumentMarshaler m) throws ArgsException {  
    try {  
      m.set(currentArgument**.next()**);  
    } catch (**NoSuchElementException** e) {  
      errorCode = ErrorCode.MISSING\_STRING;  
      throw new ArgsException();  
    }  
  }

These were simple changes that kept all the tests passing. Now we can start moving the set functions down into the appropriate derivatives. First, I need to make the following change in setArgument:

   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     **if (m == null)**  
       **return false;**  
     try {  
       if (m instanceof BooleanArgumentMarshaler)  
         setBooleanArg(m);  
       else if (m instanceof StringArgumentMarshaler)  
         setStringArg(m);  
       else if (m instanceof IntegerArgumentMarshaler)  
         setIntArg(m);  
       **else**  
         **return false;**  
     } catch (ArgsException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       throw e;  
     }  
     return true;  
   }

This change is important because we want to completely eliminate the if-else chain. Therefore, we needed to get the error condition out of it.

Now we can start to move the set functions. The setBooleanArg function is trivial, so we’ll prepare that one first. Our goal is to change the setBooleanArg function to simply forward to the BooleanArgumentMarshaler.

   private boolean setArgument(char argChar) throws ArgsException {  
       ArgumentMarshaler m = marshalers.get(argChar);  
       if (m == null)  
         return false;  
       try {  
         if (m instanceof BooleanArgumentMarshaler)  
           setBooleanArg(m, **currentArgument**);  
         else if (m instanceof StringArgumentMarshaler)  
           setStringArg(m);  
         else if (m instanceof IntegerArgumentMarshaler)  
           setIntArg(m);  
       } catch (ArgsException e) {  
         valid = false;  
         errorArgumentId = argChar;  
         throw e;  
       }  
       return true;  
     }  
   ---  
     private void setBooleanArg(ArgumentMarshaler m,   
                                **Iterator<String> currentArgument)**  
                                throws ArgsException {  
     **try {**  
        m.set(”true”);  
     **catch (ArgsException e) {**  
   }  
   }

Didn’t we just put that exception processing in? Putting things in so you can take them out again is pretty common in refactoring. The smallness of the steps and the need to keep the tests running means that you move things around a lot. Refactoring is a lot like solving a Rubik’s cube. There are lots of little steps required to achieve a large goal. Each step enables the next.

Why did we pass that iterator when setBooleanArg certainly doesn’t need it? Because setIntArg and setStringArg will! And because I want to deploy all three of these functions through an abstract method in ArgumentMarshaller, I need to pass it to setBooleanArg.

So now setBooleanArg is useless. If there were a set function in ArgumentMarshaler, we could call it directly. So it’s time to make that function! The first step is to add the new abstract method to ArgumentMarshaler.

   private abstract class ArgumentMarshaler {  
     **public abstract void set(Iterator<String> currentArgument)**  
                          **throws ArgsException;**  
     public abstract void set(String s) throws ArgsException;  
     public abstract Object get();  
   }

Of course this breaks all the derivatives. So let’s implement the new method in each.

   private class BooleanArgumentMarshaler extends ArgumentMarshaler {  
     private boolean booleanValue = false;  
  
     **public void set(Iterator<String> currentArgument) throws ArgsException {**  
       **booleanValue = true;**  
     }  
  
     public void set(String s) {  
**booleanValue = true;**  
     }  
  
     public Object get() {  
       return booleanValue;  
     }  
   }  
   private class StringArgumentMarshaler extends ArgumentMarshaler {  
     private String stringValue = ””;  
  
     **public void set(Iterator<String> currentArgument) throws ArgsException {**  
**}**       
     public void set(String s) {  
       stringValue = s;  
     }  
     public Object get() {  
       return stringValue;  
     }  
   }  
  
   private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
     private int intValue = 0;  
  
     **public void set(Iterator<String> currentArgument) throws ArgsException {**  
     **}**  
     public void set(String s) throws ArgsException {  
       try {  
         intValue = Integer.parseInt(s);  
       } catch (NumberFormatException e) {  
         throw new ArgsException();  
       }  
     }      public Object get() {  
       return intValue;  
     }  
   }

And now we can eliminate setBooleanArg!

   private boolean setArgument(char argChar) throws ArgsException {  
       ArgumentMarshaler m = marshalers.get(argChar);  
       if (m == null)  
         return false;  
       try {  
         if (m instanceof BooleanArgumentMarshaler)  
           **m.set**(currentArgument);  
         else if (m instanceof StringArgumentMarshaler)  
           setStringArg(m);  
         else if (m instanceof IntegerArgumentMarshaler)  
           setIntArg(m);  
  
       } catch (ArgsException e) {  
         valid = false;  
         errorArgumentId = argChar;  
         throw e;  
       }  
       return true;  
   }

The tests all pass, and the set function is deploying to BooleanArgumentMarshaler! Now we can do the same for Strings and Integers.

   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     if (m == null)  
       return false;  
     try {  
       if (m instanceof BooleanArgumentMarshaler)  
         m.set(currentArgument);  
       else if (m instanceof StringArgumentMarshaler)  
         **m.set(currentArgument);**  
       else if (m instanceof IntegerArgumentMarshaler)  
         **m.set(currentArgument);**    
     } catch (ArgsException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       throw e;  
     }  
     return true;  
   }  
---  
   private class StringArgumentMarshaler extends ArgumentMarshaler {  
     private String stringValue = ””;  
  
     public void set(Iterator<String> currentArgument) throws ArgsException {  
       **try {**  
         **stringValue = currentArgument.next();**  
       **} catch (NoSuchElementException e) {**  
         **errorCode = ErrorCode.MISSING\_STRING;**  
         **throw new ArgsException();**  
       **}**  
     }  
  
     public void set(String s) {  
     }  
  
     public Object get() {  
       return stringValue;  
     }  
   }  
  
   private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
     private int intValue = 0;  
  
   public void set(Iterator<String> currentArgument) throws ArgsException {  
     **String parameter = null;**  
     **try {**  
       **parameter = currentArgument.next();**  
       **set(parameter);**  
     **} catch (NoSuchElementException e) {**  
       **errorCode = ErrorCode.MISSING\_INTEGER;**  
       **throw new ArgsException();**  
     **} catch (ArgsException e) {**  
       **errorParameter = parameter;**  
       **errorCode = ErrorCode.INVALID\_INTEGER;**  
       **throw e;**  
     **}**  
   }  
  
     public void set(String s) throws ArgsException {  
       try {  
         intValue = Integer.parseInt(s);  
       } catch (NumberFormatException e) {  
         throw new ArgsException();  
       }  
     }  
  
     public Object get() {  
       return intValue;  
     }  
   }

And so the coup de grace: The type-case can be removed! Touche!

   private boolean setArgument(char argChar) throws ArgsException {  
     ArgumentMarshaler m = marshalers.get(argChar);  
     if (m == null)  
       return false;  
     try {  
       m.set(currentArgument);  
       return true;  
     } catch (ArgsException e) {  
       valid = false;  
       errorArgumentId = argChar;  
       throw e;  
     }  
   }

Now we can get rid of some crufty functions in IntegerArgumentMarshaler and clean it up a bit.

   private class IntegerArgumentMarshaler extends ArgumentMarshaler {  
     private int intValue = 0  
  
     public void set(Iterator<String> currentArgument) throws ArgsException {  
       String parameter = null;  
       try {  
         parameter = currentArgument.next();  
         **intValue = Integer.parseInt**(parameter);  
       } catch (NoSuchElementException e) {  
         errorCode = ErrorCode.MISSING\_INTEGER;  
         throw new ArgsException();  
       } catch (**NumberFormatException** e) {  
         errorParameter = parameter;  
         errorCode = ErrorCode.INVALID\_INTEGER;  
         throw new ArgsException();  
       }  
     }  
  
     public Object get() {  
       return intValue;  
     }  
   }

We can also turn **ArgumentMarshaler** into an interface.

   private **interface** ArgumentMarshaler {  
     void set(Iterator<String> currentArgument) throws ArgsException;  
     Object get();  
   }

So now let’s see how easy it is to add a new argument type to our structure. It should require very few changes, and those changes should be isolated. First, we begin by adding a new test case to check that the double argument works correctly.

   public void testSimpleDoublePresent() throws Exception {  
     Args args = new Args(”x##”, new String[] {”-x”,”42.3”});  
     assertTrue(args.isValid());  
     assertEquals(1, args.cardinality());  
     assertTrue(args.has(’x’));  
     assertEquals(42.3, args.getDouble(’x’), .001);  
   }

Now we clean up the schema parsing code and add the ## detection for the double argument type.

   private void parseSchemaElement(String element) throws ParseException {  
     char elementId = element.charAt(0);  
     String elementTail = element.substring(1);  
     validateSchemaElementId(elementId);  
     if (elementTail.**length() == 0**)  
       marshalers.put(elementId, new BooleanArgumentMarshaler());  
     else if (elementTail.**equals(”\*”)**)  
       marshalers.put(elementId, new StringArgumentMarshaler());  
     else if (elementTail.**equals(”#”)**)  
       marshalers.put(elementId, new IntegerArgumentMarshaler());   
     **else if (elementTail.equals(”##”))**  
       **marshalers.put(elementId, new DoubleArgumentMarshaler());**  
     else  
       throw new ParseException(String.format(  
         ”Argument: %c has invalid format: %s.”, elementId, elementTail), 0);  
   }

Next, we write the DoubleArgumentMarshaler class.

**private class DoubleArgumentMarshaler implements ArgumentMarshaler {**  
     **private double doubleValue = 0;**  
  
     **public void set(Iterator<String> currentArgument) throws ArgsException {**  
       **String parameter = null;**  
       **try {**  
         **parameter = currentArgument.next();**  
         **doubleValue = Double.parseDouble(parameter);**  
       **} catch (NoSuchElementException e) {**  
         **errorCode = ErrorCode.MISSING\_DOUBLE;**  
         **throw new ArgsException();**  
       **} catch (NumberFormatException e) {**  
         **errorParameter = parameter;**  
         **errorCode = ErrorCode.INVALID\_DOUBLE;**  
         **throw new ArgsException();**  
       **}**  
     **}**  
  
     **public Object get() {**  
       **return doubleValue;**  
     **}**  
   **}**

This forces us to add a new ErrorCode.

   private enum ErrorCode {  
       OK, MISSING\_STRING, MISSING\_INTEGER, INVALID\_INTEGER, UNEXPECTED\_ARGUMENT,  
       **MISSING\_DOUBLE, INVALID\_DOUBLE**}

And we need a getDouble function.

**public double getDouble(char arg) {**  
     **Args.ArgumentMarshaler am = marshalers.get(arg);**  
     **try {**  
       **return am == null ? 0 : (Double) am.get();**  
     **} catch (Exception e) {**  
       **return 0.0;**  
     **}**  
   **}**

And all the tests pass! That was pretty painless. So now let’s make sure all the error processing works correctly. The next test case checks that an error is declared if an unparseable string is fed to a ##argument.

   public void testInvalidDouble() throws Exception {  
     Args args = new Args(”x##”, new String[] {”-x”,”Forty two”});  
     assertFalse(args.isValid());  
     assertEquals(0, args.cardinality());  
     assertFalse(args.has(’x’));  
     assertEquals(0, args.getInt(’x’));  
       assertEquals(”Argument -x expects a double but was ‘Forty two’.”,  
                    args.errorMessage());  
   }  
---  
   public String errorMessage() throws Exception {  
     switch (errorCode) {  
       case OK:  
         throw new Exception(”TILT: Should not get here.”);  
       case UNEXPECTED\_ARGUMENT:  
         return unexpectedArgumentMessage();  
       case MISSING\_STRING:  
         return String.format(”Could not find string parameter for -%c.”,  
                              errorArgumentId);  
       case INVALID\_INTEGER:  
         return String.format(”Argument -%c expects an integer but was ‘%s’.”,  
                              errorArgumentId, errorParameter);  
       case MISSING\_INTEGER:  
         return String.format(”Could not find integer parameter for -%c.”,  
                              errorArgumentId);  
       **case INVALID\_DOUBLE:**  
         **return String.format(”Argument -%c expects a double but was ‘%s’.”,**  
                              **errorArgumentId, errorParameter);**  
       **case MISSING\_DOUBLE:**  
         **return String.format(”Could not find double parameter for -%c.”**,  
                              **errorArgumentId);**  
     }  
     return””;  
   }

And the tests pass. The next test makes sure we detect a missing double argument properly.

**public void testMissingDouble() throws Exception {**  
     **Args args = new Args(”x##”, new String[]{”-x”});**  
     **assertFalse(args.isValid());**  
     **assertEquals(0, args.cardinality());**  
     **assertFalse(args.has(’x’));**  
     **assertEquals(0.0, args.getDouble(’x’), 0.01);**  
     **assertEquals(”Could not find double parameter for -x.”,**  
                  **args.errorMessage());**  
   **}**

This passes as expected. We wrote it simply for completeness.

The exception code is pretty ugly and doesn’t really belong in the Args class. We are also throwing out ParseException, which doesn’t really belong to us. So let’s merge all the exceptions into a single ArgsException class and move it into its own module.

**public class ArgsException extends Exception {**  
     **private char errorArgumentId = ’\0’;**  
     **private String errorParameter = ”TILT”;**  
     **private ErrorCode errorCode = ErrorCode.OK;**  
  
     **public ArgsException() {}**  
  
     **public ArgsException(String message) {super(message);}**  
  
     **public enum ErrorCode {**  
       **OK, MISSING\_STRING, MISSING\_INTEGER,  
           INVALID\_INTEGER, UNEXPECTED\_ARGUMENT,**  
         **MISSING\_DOUBLE, INVALID\_DOUBLE}**  
   }  
   ---  
   public class Args {  
     …  
     private char errorArgumentId = ’\0’;  
     private String errorParameter = ”TILT”;  
     private **ArgsException**.ErrorCode errorCode = **ArgsException**.ErrorCode.OK;  
     private List<String> argsList;  
  
     public Args(String schema, String[] args) throws **ArgsException** {  
       this.schema = schema;  
       argsList = Arrays.asList(args);  
       valid = parse();  
     }  
  
     private boolean parse() throws ArgsException {  
       if (schema.length() == 0 && argsList.size() == 0)  
         return true;  
       parseSchema();  
       try {  
         parseArguments();  
       } catch (**ArgsException** e) {  
       }  
       return valid;  
     }  
     private boolean parseSchema() throws **ArgsException** {  
       …  
     }  
  
     private void parseSchemaElement(String element) throws **ArgsException** {  
       …  
       else  
         throw new **ArgsException**(  
           String.format(”Argument: %c has invalid format: %s.”,  
                         elementId,elementTail));  
     }  
     private void validateSchemaElementId(char elementId) throws **ArgsException** {  
       if (!Character.isLetter(elementId)) {  
         throw new **ArgsException**(  
           ”Bad character:” + elementId + ”in Args format: ” + schema);  
       }  
     }  
  
     …  
  
     private void parseElement(char argChar) throws **ArgsException** {  
       if (setArgument(argChar))  
         argsFound.add(argChar);  
       else {  
         unexpectedArguments.add(argChar);  
         errorCode = **ArgsException**.ErrorCode.UNEXPECTED\_ARGUMENT;  
         valid = false;  
       }  
     }  
  
     …  
     private class StringArgumentMarshaler implements ArgumentMarshaler {  
       private String stringValue = ””;  
       public void set(Iterator<String> currentArgument) throws ArgsException {  
         try {  
           stringValue = currentArgument.next();  
         } catch (NoSuchElementException e) {  
           errorCode = **ArgsException**.ErrorCode.MISSING\_STRING;  
           throw new ArgsException();  
         }  
       }  
       public Object get() {  
         return stringValue;  
       }  
     }  
  
     private class IntegerArgumentMarshaler implements ArgumentMarshaler {  
       private int intValue = 0;  
  
       public void set(Iterator<String> currentArgument) throws **ArgsException** {  
         String parameter = null;  
         try {  
           parameter = currentArgument.next();  
           intValue = Integer.parseInt(parameter);  
         } catch (NoSuchElementException e) {  
           errorCode = ArgsException.ErrorCode.MISSING\_INTEGER;  
           throw new **ArgsException**();  
         } catch (NumberFormatException e) {  
           errorParameter = parameter;  
           errorCode = **ArgsException**.ErrorCode.INVALID\_INTEGER;  
           throw new **ArgsException**();  
         }  
       }  
  
       public Object get() {  
         return intValue;  
       }  
     }  
  
     private class DoubleArgumentMarshaler implements ArgumentMarshaler {  
       private double doubleValue = 0;  
  
       public void set(Iterator<String> currentArgument) throws ArgsException {  
         String parameter = null;  
         try {  
           parameter = currentArgument.next();  
           doubleValue = Double.parseDouble(parameter);  
         } catch (NoSuchElementException e) {  
           errorCode = **ArgsException**.ErrorCode.MISSING\_DOUBLE;  
           throw new ArgsException();  
         } catch (NumberFormatException e) {  
           errorParameter = parameter;  
           errorCode = **ArgsException**.ErrorCode.INVALID\_DOUBLE;  
           throw new ArgsException();  
         }  
       }  
       public Object get() {  
         return doubleValue;  
       }  
     }  
   }

This is nice. Now the only exception thrown by Args is ArgsException. Moving ArgsExceptioninto its own module means that we can move a lot of the miscellaneous error support code into that module and out of the Args module. It provides a natural and obvious place to put all that code and will really help us clean up the Args module going forward.

So now we have completely separated the exception and error code from the Args module. (See [Listing 14-13](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt13) through [Listing 14-16](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter14.html#ch14lt16).) This was achieved through a series of about 30 tiny steps, keeping the tests passing between each step.

**Listing 14-13 ArgsTest.java**

   package com.objectmentor.utilities.args;  
  
   import junit.framework.TestCase;  
  
   public class ArgsTest extends TestCase {  
     public void testCreateWithNoSchemaOrArguments() throws Exception {  
       Args args = new Args(“”, new String[0]);  
       assertEquals(0, args.cardinality());  
     }  
  
     public void testWithNoSchemaButWithOneArgument() throws Exception {  
       try {  
         new Args(“”, new String[]{“-x”});  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT,  
                      e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
       }  
     }  
  
     public void testWithNoSchemaButWithMultipleArguments() throws Exception {  
       try {  
         new Args(“”, new String[]{“-x”, “-y”});  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT,  
                      e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
       }  
  
     }  
  
     public void testNonLetterSchema() throws Exception {  
       try {  
         new Args(“\*”, new String[]{});  
         fail(“Args constructor should have thrown exception”);  
       } catch (ArgsException e) {  
  
         assertEquals(ArgsException.ErrorCode.INVALID\_ARGUMENT\_NAME,  
                      e.getErrorCode());  
         assertEquals(‘\*’, e.getErrorArgumentId());  
       }  
     }  
  
     public void testInvalidArgumentFormat() throws Exception {  
       try {  
         new Args(“f~”, new String[]{});  
         fail(“Args constructor should have throws exception”);  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.INVALID\_FORMAT, e.getErrorCode());  
         assertEquals(‘f’, e.getErrorArgumentId());  
       }  
     }  
  
     public void testSimpleBooleanPresent() throws Exception {  
       Args args = new Args(“x”, new String[]{“-x”});  
       assertEquals(1, args.cardinality());  
       assertEquals(true, args.getBoolean(‘x’));  
     }  
  
     public void testSimpleStringPresent() throws Exception {  
       Args args = new Args(“x\*”, new String[]{“-x”, “param”});  
       assertEquals(1, args.cardinality());  
       assertTrue(args.has(‘x’));  
       assertEquals(“param”, args.getString(‘x’));  
     }  
  
     public void testMissingStringArgument() throws Exception {  
       try {  
         new Args(“x\*”, new String[]{“-x”});  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.MISSING\_STRING, e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
       }  
     }  
  
     public void testSpacesInFormat() throws Exception {  
       Args args = new Args(“x, y”, new String[]{“-xy”});  
       assertEquals(2, args.cardinality());  
       assertTrue(args.has(‘x’));  
       assertTrue(args.has(‘y’));  
     }  
  
     public void testSimpleIntPresent() throws Exception {  
       Args args = new Args(“x#”, new String[]{“-x”, “42”});  
       assertEquals(1, args.cardinality());  
       assertTrue(args.has(‘x’));  
       assertEquals(42, args.getInt(‘x’));  
     }  
  
     public void testInvalidInteger() throws Exception {  
       try {  
         new Args(“x#”, new String[]{“-x”, “Forty two”});  
  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.INVALID\_INTEGER, e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
         assertEquals(”Forty two”, e.getErrorParameter());  
       }  
  
     }  
  
     public void testMissingInteger() throws Exception {  
       try {  
         new Args(“x#”, new String[]{“-x”});  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.MISSING\_INTEGER, e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
       }  
     }  
  
     public void testSimpleDoublePresent() throws Exception {  
       Args args = new Args(“x##”, new String[]{“-x”, “42.3”});  
       assertEquals(1, args.cardinality());  
       assertTrue(args.has(‘x’));  
       assertEquals(42.3, args.getDouble(‘x’), .001);  
     }  
  
     public void testInvalidDouble() throws Exception {  
       try {  
         new Args(“x##”, new String[]{“-x”, “Forty two”});  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.INVALID\_DOUBLE, e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
         assertEquals(“Forty two”, e.getErrorParameter());  
       }  
     }  
  
     public void testMissingDouble() throws Exception {  
       try {  
         new Args(“x##”, new String[]{“-x”});  
         fail();  
       } catch (ArgsException e) {  
         assertEquals(ArgsException.ErrorCode.MISSING\_DOUBLE, e.getErrorCode());  
         assertEquals(‘x’, e.getErrorArgumentId());  
       }  
     }  
   }

**Listing 14-14 ArgsExceptionTest.java**

   public class ArgsExceptionTest extends TestCase {  
     public void testUnexpectedMessage() throws Exception {  
       ArgsException e =  
  
         new ArgsException(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT,  
                           ‘x’, null);  
       assertEquals(“Argument -x unexpected.”, e.errorMessage());  
     }  
  
     public void testMissingStringMessage() throws Exception {  
       ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING\_STRING,  
                                           ‘x’, null);  
       assertEquals(“Could not find string parameter for -x.”, e.errorMessage());  
     }  
  
     public void testInvalidIntegerMessage() throws Exception {  
       ArgsException e =  
         new ArgsException(ArgsException.ErrorCode.INVALID\_INTEGER,  
                           ‘x’, “Forty two”);  
       assertEquals(“Argument -x expects an integer but was ‘Forty two’.“,  
                    e.errorMessage());  
     }  
  
     public void testMissingIntegerMessage() throws Exception {  
       ArgsException e =  
         new ArgsException(ArgsException.ErrorCode.MISSING\_INTEGER, ‘x’, null);  
       assertEquals(“Could not find integer parameter for -x.”, e.errorMessage());  
     }  
  
     public void testInvalidDoubleMessage() throws Exception {  
       ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID\_DOUBLE,  
                                           ‘x’, “Forty two”);  
       assertEquals(“Argument -x expects a double but was ‘Forty two’.”,  
                    e.errorMessage());  
     }  
  
     public void testMissingDoubleMessage() throws Exception {  
       ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING\_DOUBLE,  
                                           ‘x’, null);  
       assertEquals(“Could not find double parameter for -x.”, e.errorMessage());  
     }  
   }

**Listing 14-15 ArgsException.java**

   public class ArgsException extends Exception {  
     private char errorArgumentId = ‘\0’;  
     private String errorParameter = “TILT”;  
     private ErrorCode errorCode = ErrorCode.OK;  
  
     public ArgsException() {}  
  
     public ArgsException(String message) {super(message);}  
  
     public ArgsException(ErrorCode errorCode) {  
       this.errorCode = errorCode;  
     }  
  
     public ArgsException(ErrorCode errorCode, String errorParameter) {  
       this.errorCode = errorCode;  
       this.errorParameter = errorParameter;  
     }  
  
     public ArgsException(ErrorCode errorCode, char errorArgumentId,  
                          String errorParameter) {  
       this.errorCode = errorCode;  
       this.errorParameter = errorParameter;  
       this.errorArgumentId = errorArgumentId;  
     }  
  
     public char getErrorArgumentId() {  
       return errorArgumentId;  
     }  
  
     public void setErrorArgumentId(char errorArgumentId) {  
       this.errorArgumentId = errorArgumentId;  
     }  
  
     public String getErrorParameter() {  
       return errorParameter;  
     }  
  
     public void setErrorParameter(String errorParameter) {  
       this.errorParameter = errorParameter;  
     }  
  
     public ErrorCode getErrorCode() {  
       return errorCode;  
     }  
  
     public void setErrorCode(ErrorCode errorCode) {  
       this.errorCode = errorCode;  
     }  
  
     public String errorMessage() throws Exception {  
       switch (errorCode) {  
         case OK:  
           throw new Exception(“TILT: Should not get here.”);  
         case UNEXPECTED\_ARGUMENT:  
           return String.format(“Argument -%c unexpected.”, errorArgumentId);  
         case MISSING\_STRING:  
           return String.format(“Could not find string parameter for -%c.”,  
                                errorArgumentId);  
         case INVALID\_INTEGER:  
           return String.format(“Argument -%c expects an integer but was ‘%s’.”,  
                                errorArgumentId, errorParameter);  
         case MISSING\_INTEGER:  
           return String.format(“Could not find integer parameter for -%c.”,  
                                errorArgumentId);  
         case INVALID\_DOUBLE:  
           return String.format(“Argument -%c expects a double but was ‘%s’.”,  
                                errorArgumentId, errorParameter);  
  
         case MISSING\_DOUBLE:  
           return String.format(“Could not find double parameter for -%c.”,  
                                errorArgumentId);  
       }  
       return “”;  
     }  
  
     public enum ErrorCode {  
       OK, INVALID\_FORMAT, UNEXPECTED\_ARGUMENT, INVALID\_ARGUMENT\_NAME,  
       MISSING\_STRING,  
       MISSING\_INTEGER, INVALID\_INTEGER,  
       MISSING\_DOUBLE, INVALID\_DOUBLE}  
   }

**Listing 14-16 Args.java**

   public class Args {  
     private String schema;  
     private Map<Character, ArgumentMarshaler> marshalers =  
       new HashMap<Character, ArgumentMarshaler>();  
     private Set<Character> argsFound = new HashSet<Character>();  
     private Iterator<String> currentArgument;  
     private List<String> argsList;  
  
     public Args(String schema, String[] args) throws ArgsException {  
       this.schema = schema;  
       argsList = Arrays.asList(args);  
       parse();  
     }  
  
     private void parse() throws ArgsException {  
       parseSchema();  
       parseArguments();  
     }  
  
     private boolean parseSchema() throws ArgsException {  
       for (String element : schema.split(“,”)) {  
         if (element.length() > 0) {  
           parseSchemaElement(element.trim());  
         }  
       }  
       return true;  
     }  
  
     private void parseSchemaElement(String element) throws ArgsException {  
       char elementId = element.charAt(0);  
       String elementTail = element.substring(1);  
       validateSchemaElementId(elementId);  
       if (elementTail.length() == 0)  
         marshalers.put(elementId, new BooleanArgumentMarshaler());  
       else if (elementTail.equals(“\*”))  
         marshalers.put(elementId, new StringArgumentMarshaler());  
  
       else if (elementTail.equals(“#”))  
         marshalers.put(elementId, new IntegerArgumentMarshaler());  
       else if (elementTail.equals(“##”))  
         marshalers.put(elementId, new DoubleArgumentMarshaler());  
       else  
         throw new ArgsException(ArgsException.ErrorCode.INVALID\_FORMAT,  
                                 elementId, elementTail);  
     }  
  
     private void validateSchemaElementId(char elementId) throws ArgsException {  
       if (!Character.isLetter(elementId)) {  
         throw new ArgsException(ArgsException.ErrorCode.INVALID\_ARGUMENT\_NAME,  
                                 elementId, null);  
       }  
     }  
  
     private void parseArguments() throws ArgsException {  
       for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {  
         String arg = currentArgument.next();  
         parseArgument(arg);  
       }  
     }  
  
     private void parseArgument(String arg) throws ArgsException {  
       if (arg.startsWith(“-”))  
         parseElements(arg);  
     }  
  
     private void parseElements(String arg) throws ArgsException {  
       for (int i = 1; i < arg.length(); i++)  
         parseElement(arg.charAt(i));  
     }  
  
     private void parseElement(char argChar) throws ArgsException {  
       if (setArgument(argChar))  
         argsFound.add(argChar);  
       else {  
         throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED\_ARGUMENT,  
                                 argChar, null);  
       }  
     }  
  
     private boolean setArgument(char argChar) throws ArgsException {  
       ArgumentMarshaler m = marshalers.get(argChar);  
       if (m == null)  
         return false;  
       try {  
         m.set(currentArgument);  
         return true;  
       } catch (ArgsException e) {  
         e.setErrorArgumentId(argChar);  
         throw e;  
       }  
     }  
  
     public int cardinality() {  
       return argsFound.size();  
     }  
  
     public String usage() {  
       if (schema.length() > 0)  
         return “-[” + schema + “]”;  
       else  
         return “”;  
     }  
  
     public boolean getBoolean(char arg) {  
       ArgumentMarshaler am = marshalers.get(arg);  
       boolean b = false;  
       try {  
         b = am != null && (Boolean) am.get();  
       } catch (ClassCastException e) {  
         b = false;  
       }  
       return b;  
     }  
  
     public String getString(char arg) {  
       ArgumentMarshaler am = marshalers.get(arg);  
       try {  
         return am == null ? “” : (String) am.get();  
       } catch (ClassCastException e) {  
         return “”;  
       }  
     }  
  
     public int getInt(char arg) {  
       ArgumentMarshaler am = marshalers.get(arg);  
       try {  
         return am == null ? 0 : (Integer) am.get();  
       } catch (Exception e) {  
         return 0;  
       }  
     }  
  
  
     public double getDouble(char arg) {  
       ArgumentMarshaler am = marshalers.get(arg);  
       try {  
         return am == null ? 0 : (Double) am.get();  
       } catch (Exception e) {  
         return 0.0;  
       }  
     }  
  
     public boolean has(char arg) {  
       return argsFound.contains(arg);  
     }  
   }

The majority of the changes to the Args class were deletions. A lot of code just got moved out of Argsand put into ArgsException. Nice. We also moved all the ArgumentMarshaller s into their own files. Nicer!

Much of good software design is simply about partitioning—creating appropriate places to put different kinds of code. This separation of concerns makes the code much simpler to understand and maintain.

Of special interest is the errorMessage method of ArgsException. Clearly it was a violation of the SRP to put the error message formatting into Args. Args should be about the processing of arguments, not about the format of the error messages. However, does it really make sense to put the error message formatting code into ArgsException?

Frankly, it’s a compromise. Users who don’t like the error messages supplied by ArgsException will have to write their own. But the convenience of having canned error messages already prepared for you is not insignificant.

By now it should be clear that we are within striking distance of the final solution that appeared at the start of this chapter. I’ll leave the final transformations to you as an exercise.

### **CONCLUSION**

It is not enough for code to work. Code that works is often badly broken. Programmers who satisfy themselves with merely working code are behaving unprofessionally. They may fear that they don’t have time to improve the structure and design of their code, but I disagree. Nothing has a more profound and long-term degrading effect upon a development project than bad code. Bad schedules can be redone, bad requirements can be redefined. Bad team dynamics can be repaired. But bad code rots and ferments, becoming an inexorable weight that drags the team down. Time and time again I have seen teams grind to a crawl because, in their haste, they created a malignant morass of code that forever thereafter dominated their destiny.

Of course bad code can be cleaned up. But it’s very expensive. As code rots, the modules insinuate themselves into each other, creating lots of hidden and tangled dependencies. Finding and breaking old dependencies is a long and arduous task. On the other hand, keeping code clean is relatively easy. If you made a mess in a module in the morning, it is easy to clean it up in the afternoon. Better yet, if you made a mess five minutes ago, it’s very easy to clean it up right now.

So the solution is to continuously keep your code as clean and simple as it can be. Never let the rot get started.

## ****JUnit Internals****

JUnit is one of the most famous of all Java frameworks. As frameworks go, it is simple in conception, precise in definition, and elegant in implementation. But what does the code look like? In this chapter we’ll critique an example drawn from the JUnit framework.

### **THE JUNIT FRAMEWORK**

JUnit has had many authors, but it began with Kent Beck and Eric Gamma together on a plane to Atlanta. Kent wanted to learn Java, and Eric wanted to learn about Kent’s Smalltalk testing framework. “What could be more natural to a couple of geeks in cramped quarters than to pull out our laptops and start coding?”1 After three hours of high-altitude work, they had written the basics of JUnit.

1. JUnit Pocket Guide, Kent Beck, O’Reilly, 2004, p. [43](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#page_43).

The module we’ll look at is the clever bit of code that helps identify string comparison errors. This module is called ComparisonCompactor. Given two strings that differ, such as ABCDE and ABXDE, it will expose the difference by generating a string such as <…B[X]D…>.

I could explain it further, but the test cases do a better job. So take a look at [Listing 15-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter15.html#ch15lt1) and you will understand the requirements of this module in depth. While you are at it, critique the structure of the tests. Could they be simpler or more obvious?

**Listing 15-1 ComparisonCompactorTest.java**

   package junit.tests.framework;  
   import junit.framework.ComparisonCompactor;  
   import junit.framework.TestCase;  
  
   public class ComparisonCompactorTest extends TestCase {  
  
     public void testMessage() {  
       String failure= new ComparisonCompactor(0, “b”, “c”).compact(“a”);  
       assertTrue(“a expected:<[b]> but was:<[c]>”.equals(failure));  
     }  
  
     public void testStartSame() {  
       String failure= new ComparisonCompactor(1, “ba”, “bc”).compact(null);  
       assertEquals(“expected:<b[a]> but was:<b[c]>”, failure);  
     }  
  
     public void testEndSame() {  
       String failure= new ComparisonCompactor(1, “ab”, “cb”).compact(null);  
       assertEquals(“expected:<[a]b> but was:<[c]b>”, failure);  
     }  
  
     public void testSame() {  
       String failure= new ComparisonCompactor(1, “ab”, “ab”).compact(null);  
       assertEquals(“expected:<ab> but was:<ab>”, failure);  
     }  
  
     public void testNoContextStartAndEndSame() {  
       String failure= new ComparisonCompactor(0, “abc”, “adc”).compact(null);  
       assertEquals(“expected:<…[b]…> but was:<…[d]…>”, failure);  
     }  
     public void testStartAndEndContext() {  
       String failure= new ComparisonCompactor(1, “abc”, “adc”).compact(null);  
       assertEquals(“expected:<a[b]c> but was:<a[d]c>”, failure);  
     }  
    
     public void testStartAndEndContextWithEllipses() {  
       String failure=   
    new ComparisonCompactor(1, “abcde”, “abfde”).compact(null);  
       assertEquals(“expected:<…b[c]d…> but was:<…b[f]d…>”, failure);  
     }  
  
     public void testComparisonErrorStartSameComplete() {  
       String failure= new ComparisonCompactor(2, “ab”, “abc”).compact(null);  
       assertEquals(“expected:<ab[]> but was:<ab[c]>”, failure);  
     }  
  
     public void testComparisonErrorEndSameComplete() {  
       String failure= new ComparisonCompactor(0, “bc”, “abc”).compact(null);  
       assertEquals(“expected:<[]…> but was:<[a]…>”, failure);  
     }  
  
     public void testComparisonErrorEndSameCompleteContext() {  
       String failure= new ComparisonCompactor(2, “bc”, “abc”).compact(null);  
       assertEquals(“expected:<[]bc> but was:<[a]bc>”, failure);  
     }  
  
     public void testComparisonErrorOverlapingMatches() {  
       String failure= new ComparisonCompactor(0, “abc”, “abbc”).compact(null);  
       assertEquals(“expected:<…[]…> but was:<…[b]…>”, failure);  
   }  
  
     public void testComparisonErrorOverlapingMatchesContext() {  
       String failure= new ComparisonCompactor(2, “abc”, “abbc”).compact(null);  
       assertEquals(“expected:<ab[]c> but was:<ab[b]c>”, failure);  
     }  
  
     public void testComparisonErrorOverlapingMatches2() {  
       String failure= new ComparisonCompactor(0, “abcdde”,  
“abcde”).compact(null);  
       assertEquals(“expected:<…[d]…> but was:<…[]…>”, failure);  
     }  
     public void testComparisonErrorOverlapingMatches2Context() {  
       String failure=   
      new ComparisonCompactor(2, “abcdde”, “abcde”).compact(null);  
       assertEquals(“expected:<…cd[d]e> but was:<…cd[]e>”, failure);  
     }  
  
     public void testComparisonErrorWithActualNull() {  
       String failure= new ComparisonCompactor(0, “a”, null).compact(null);  
       assertEquals(“expected:<a> but was:<null>”, failure);  
     }  
    
     public void testComparisonErrorWithActualNullContext() {  
       String failure= new ComparisonCompactor(2, “a”, null).compact(null);  
       assertEquals(“expected:<a> but was:<null>”, failure);  
    }  
  
    public void testComparisonErrorWithExpectedNull() {  
      String failure= new ComparisonCompactor(0, null, “a”).compact(null);  
      assertEquals(“expected:<null> but was:<a>”, failure);  
    }  
  
    public void testComparisonErrorWithExpectedNullContext() {  
      String failure= new ComparisonCompactor(2, null, “a”).compact(null);  
      assertEquals(“expected:<null> but was:<a>”, failure);  
    }  
    public void testBug609972() {  
      String failure= new ComparisonCompactor(10, “S&P500”, “0”).compact(null);  
      assertEquals(“expected:<[S&P50]0> but was:<[]0>”, failure);  
    }  
   }

I ran a code coverage analysis on the ComparisonCompactor using these tests. The code is 100 percent covered. Every line of code, every if statement and for loop, is executed by the tests. This gives me a high degree of confidence that the code works and a high degree of respect for the craftsmanship of the authors.

The code for ComparisonCompactor is in [Listing 15-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter15.html#ch15lt2). Take a moment to look over this code. I think you’ll find it to be nicely partitioned, reasonably expressive, and simple in structure. Once you are done, then we’ll pick the nits together.

**Listing 15-2 ComparisonCompactor.java (Original)**

   package junit.framework;  
  
   public class ComparisonCompactor {  
  
     private static final String ELLIPSIS = “…”;  
     private static final String DELTA\_END = “]”;  
     private static final String DELTA\_START = “[”;  
  
     private int fContextLength;  
     private String fExpected;  
     private String fActual;  
     private int fPrefix;  
     private int fSuffix;  
  
     public ComparisonCompactor(int contextLength,  
                                String expected,  
                                  String actual) {  
       fContextLength = contextLength;  
       fExpected = expected;  
       fActual = actual;  
     }  
  
     public String compact(String message) {  
       if (fExpected == null || fActual == null || areStringsEqual())  
         return Assert.format(message, fExpected, fActual);  
  
       findCommonPrefix();  
       findCommonSuffix();  
       String expected = compactString(fExpected);  
       String actual = compactString(fActual);  
       return Assert.format(message, expected, actual);  
     }  
  
     private String compactString(String source) {  
       String result = DELTA\_START +   
                         source.substring(fPrefix, source.length() - fSuffix + 1) + DELTA\_END;  
       if (fPrefix > 0)  
         result = computeCommonPrefix() + result;  
       if (fSuffix > 0)  
         result = result + computeCommonSuffix();  
       return result;  
   }  
   private void findCommonPrefix() {  
     fPrefix = 0;  
     int end = Math.min(fExpected.length(), fActual.length());  
     for (; fPrefix < end; fPrefix++) {  
       if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))  
         break;  
     }  
   }  
  
   private void findCommonSuffix() {  
     int expectedSuffix = fExpected.length() - 1;  
     int actualSuffix = fActual.length() - 1;  
     for (;  
          actualSuffix >= fPrefix && expectedSuffix >= fPrefix;  
           actualSuffix--, expectedSuffix--) {  
       if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))  
         break;  
     }  
     fSuffix = fExpected.length() - expectedSuffix;  
   }  
  
   private String computeCommonPrefix() {  
     return (fPrefix > fContextLength ? ELLIPSIS : “”) +  
              fExpected.substring(Math.max(0, fPrefix - fContextLength),  
                                     fPrefix);  
   }  
   private String computeCommonSuffix() {  
     int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,  
                          fExpected.length());  
     return fExpected.substring(fExpected.length() - fSuffix + 1, end) +  
            (fExpected.length() - fSuffix + 1 < fExpected.length() -  fContextLength ? ELLIPSIS : “”);  
  
   }  
  
   private boolean areStringsEqual() {  
     return fExpected.equals(fActual);  
   }  
  }

You might have a few complaints about this module. There are some long expressions and some strange +1s and so forth. But overall this module is pretty good. After all, it might have looked like [Listing 15-3](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter15.html#ch15lt3).

**Listing 15-3 ComparisonCompator.java (defactored)**

   package junit.framework;  
   public class  ComparisonCompactor {  
     private int ctxt;  
     private String s1;  
     private String s2;  
     private int pfx;  
     private int sfx;  
     public ComparisonCompactor(int ctxt, String s1, String s2) {  
       this.ctxt = ctxt;  
       this.s1 = s1;  
       this.s2 = s2;  
     }  
  
     public String compact(String msg) {  
       if (s1 == null || s2 == null || s1.equals(s2))  
         return Assert.format(msg, s1, s2);  
  
       pfx = 0;  
       for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {  
         if (s1.charAt(pfx) != s2.charAt(pfx))  
           break;  
       }  
       int sfx1 = s1.length() - 1;  
       int sfx2 = s2.length() - 1;  
       for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {  
         if (s1.charAt(sfx1) != s2.charAt(sfx2))  
         break;  
     }  
       sfx = s1.length() - sfx1;  
       String cmp1 = compactString(s1);  
       String cmp2 = compactString(s2);  
       return Assert.format(msg, cmp1, cmp2);  
   }  
   private String compactString(String s) {  
     String result =  
      “[“ + s.substring(pfx, s.length() - sfx + 1) + “]”;  
     if (pfx > 0)  
       result = (pfx > ctxt ? “…” : “”) +  
         s1.substring(Math.max(0, pfx - ctxt), pfx) + result;  
     if (sfx > 0) {  
       int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());  
       result = result + (s1.substring(s1.length() - sfx + 1, end) +  
         (s1.length() - sfx + 1 < s1.length() - ctxt ? “…” : “”));  
     }  
     return result;  
    }  
  }

Even though the authors left this module in very good shape, the Boy Scout Rule2 tells us we should leave it cleaner than we found it. So, how can we improve on the original code in Listing 15-2?

2. See “[The Boy Scout Rule](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter01.html#ch1lev1sec6)” on page [14](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter01.html#page_14).

The first thing I don’t care for is the f prefix for the member variables [N6]. Today’s environments make this kind of scope encoding redundant. So let’s eliminate all the f’s.

   private int contextLength;  
   private String expected;  
   private String actual;  
   private int prefix;  
   private int suffix;

Next, we have an unencapsulated conditional at the beginning of the compact function [G28].

   public String compact(String message) {  
     **if (expected == null || actual == null || areStringsEqual())**  
       return Assert.format(message, expected, actual);  
     findCommonPrefix();  
     findCommonSuffix();  
     String expected = compactString(this.expected);  
     String actual = compactString(this.actual);  
     return Assert.format(message, expected, actual);  
   }

This conditional should be encapsulated to make our intent clear. So let’s extract a method that explains it.

     public String compact(String message) {  
       if (**shouldNotCompact()**)  
         return Assert.format(message, expected, actual);  
       findCommonPrefix();  
       findCommonSuffix();  
       String expected = compactString(this.expected);  
       String actual = compactString(this.actual);  
       return Assert.format(message, expected, actual);  
     }  
**private boolean shouldNotCompact() {  
       return expected == null || actual == null || areStringsEqual();  
     }**

I don’t much care for the this.expected and this.actual notation in the compact function. This happened when we changed the name of fExpected to expected. Why are there variables in this function that have the same names as the member variables? Don’t they represent something else [N4]? We should make the names unambiguous.

   String **compactExpected** = compactString(**expected**);   String **compactActual** = compactString(**actual**);

Negatives are slightly harder to understand than positives [G29]. So let’s turn that if statement on its head and invert the sense of the conditional.

   public String compact(String message) {  
     if (**canBeCompacted**()) {  
       findCommonPrefix();  
       findCommonSuffix();  
       String compactExpected = compactString(expected);  
       String compactActual = compactString(actual);  
       return Assert.format(message, compactExpected, compactActual);  
     } else {  
       return Assert.format(message, expected, actual);  
     }  
   }  
   private boolean **canBeCompacted**() {  
     return expected **!=** null **&&** actual **!=** null **&& !** areStringsEqual();  
   }

The name of the function is strange [N7]. Although it does compact the strings, it actually might not compact the strings if canBeCompacted returns false. So naming this function compact hides the side effect of the error check. Notice also that the function returns a formatted message, not just the compacted strings. So the name of the function should really be formatCompactedComparison. That makes it read a lot better when taken with the function argument:

   public String formatCompactedComparison(String message) {

The body of the if statement is where the true compacting of the expected and actual strings is done. We should extract that as a method named compactExpectedAndActual. However, we want the formatCompactedComparison function to do all the formatting. The compact… function should do nothing but compacting [G30]. So let’s split it up as follows:

  …  
**private String compactExpected;**  
**private String compactActual;**   
  …  
    public String formatCompactedComparison(String message) {  
      if (canBeCompacted()) {  
  **compactExpectedAndActual();**  
        return Assert.format(message, compactExpected, compactActual);  
      } else {   
  
        return Assert.format(message, expected, actual);  
      }  
    }  
    private void **compactExpectedAndActual**() {  
      findCommonPrefix();  
      findCommonSuffix();  
      compactExpected = compactString(expected);  
      compactActual = compactString(actual);  
    }

Notice that this required us to promote compactExpected and compactActual to member variables. I don’t like the way that the last two lines of the new function return variables, but the first two don’t. They aren’t using consistent conventions [G11]. So we should change findCommonPrefixand findCommonSuffix to return the prefix and suffix values.

  private void compactExpectedAndActual() {  
**prefixIndex =** findCommonPrefix();  
    **suffixIndex =** findCommonSuffix();  
    compactExpected = compactString(expected);  
    compactActual = compactString(actual);  
  }  
  private **int** findCommonPrefix() {  
    **int** prefix**Index** = 0;  
    int end = Math.min(expected.length(), actual.length());  
    for (; prefix**Index** < end; prefix**Index**++) {  
      if (expected.charAt(prefix**Index**) != actual.charAt(prefix**Index**))  
        break;  
    }  
    **return prefixIndex;**  
  }  
  private **int** findCommonSuffix() {    
    int expectedSuffix = expected.length() - 1;  
    int actualSuffix = actual.length() - 1;  
    for (; actualSuffix >= prefix**Index** && expectedSuffix >= prefixIndex;  
         actualSuffix--, expectedSuffix--) {  
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))  
        break;  
    }  
    **return** expected.length() - expectedSuffix;  
  }

We should also change the names of the member variables to be a little more accurate [N1]; after all, they are both indices.

Careful inspection of findCommonSuffix exposes a hidden temporal coupling [G31]; it depends on the fact that prefixIndex is calculated by findCommonPrefix. If these two functions were called out of order, there would be a difficult debugging session ahead. So, to expose this temporal coupling, let’s have findCommonSuffix take the prefixIndex as an argument.

   private void compactExpectedAndActual() {  
     prefixIndex = findCommonPrefix();  
     suffixIndex = findCommonSuffix(**prefixIndex**);  
     compactExpected = compactString(expected);  
     compactActual = compactString(actual);  
   }  
   private int findCommonSuffix(**int prefixIndex**) {  
     int expectedSuffix = expected.length() - 1;  
     int actualSuffix = actual.length() - 1;  
     for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;  
          actualSuffix--, expectedSuffix--) {  
       if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))  
         break;  
     }  
     return expected.length() - expectedSuffix;  
   }

I’m not really happy with this. The passing of the prefixIndex argument is a bit arbitrary [G32]. It works to establish the ordering but does nothing to explain the need for that ordering. Another programmer might undo what we have done because there’s no indication that the parameter is really needed. So let’s take a different tack.

   private void compactExpectedAndActual() {  
     **findCommonPrefixAndSuffix();**  
     compactExpected = compactString(expected);  
     compactActual = compactString(actual);  
   }  
   private **void** **findCommonPrefixAndSuffix**() {  
     **findCommonPrefix();**  
     int expectedSuffix = expected.length() - 1;  
     int actualSuffix = actual.length() - 1;  
     for (;  
          actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;  
          actualSuffix--, expectedSuffix--   
       )  {  
  
       if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))  
         break;  
     }  
     suffixIndex = expected.length() - expectedSuffix;  
   }  
   private **void** findCommonPrefix() {  
     prefixIndex = 0;  
     int end = Math.min(expected.length(), actual.length());  
     for (;   prefixIndex < end; prefixIndex++)  
       if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))  
         break;  
   }

We put findCommonPrefix and findCommonSuffix back the way they were, changing the name of findCommonSuffix to findCommon**PrefixAnd** Suffix and having it call findCommon-Prefixbefore doing anything else. That establishes the temporal nature of the two functions in a much more dramatic way than the previous solution. It also points out how ugly findCommonPrefixAndSuffixis. Let’s clean it up now.

   private void findCommonPrefixAndSuffix() {  
     findCommonPrefix();  
     int suffixLength = 1;   
     for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {  
       if (charFromEnd(expected, suffixLength) !=  
            charFromEnd(actual, suffixLength))  
         break;  
     }  
     suffixIndex = suffixLength;  
   }  
   private char charFromEnd(String s, int i) {  
     return s.charAt(s.length()-i);}  
   private boolean suffixOverlapsPrefix(int suffixLength) {  
     return actual.length() - suffixLength < prefixLength ||  
       expected.length() - suffixLength < prefixLength;  
   }

This is much better. It exposes that the suffixIndex is really the length of the suffix and is not well named. The same is true of the prefixIndex, though in that case “index” and “length” are synonymous. Even so, it is more consistent to use “length.” The problem is that the suffixIndexvariable is not zero based; it is 1 based and so is not a true length. This is also the reason that there are all those +1s in computeCommonSuffix [G33]. So let’s fix that. The result is in [Listing 15-4](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter15.html#ch15lt4).

**Listing 15-4 ComparisonCompactor.java (interim)**

   public class ComparisonCompactor {  
   …  
     private int **suffixLength**;  
   …  
     private void findCommonPrefixAndSuffix() {  
       findCommonPrefix();  
       **suffixLength = 0;**  
       for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {  
         if (charFromEnd(expected, suffixLength) !=  
             charFromEnd(actual, suffixLength))  
           break;  
       }  
    }  
    private char charFromEnd(String s, int i) {  
      return s.charAt(s.length() - i **- 1**);  
    }  
    private boolean suffixOverlapsPrefix(int suffixLength) {  
      return actual.length() - suffixLength **<=** prefixLength ||  
        expected.length() - suffixLength **<=** prefixLength;  
    }  
…  
    private String compactString(String source) {  
      String result =  
        DELTA\_START +  
        source.substring(prefixLength, source.length() - **suffixLength**) +  
        DELTA\_END;  
      if (prefixLength > 0)  
        result = computeCommonPrefix() + result;   
      if (**suffixLength** > 0)  
        result = result + computeCommonSuffix();  
      return result;  
    }  
  …  
    private String computeCommonSuffix() {  
      int end = Math.min(expected.length() - **suffixLength** +  
        contextLength, expected.length()  
      );  
      return  
        expected.substring(expected.length() - **suffixLength**, end) +  
        (expected.length() - **suffixLength** <  
          expected.length() - contextLength ?  
          ELLIPSIS : “”);  
    }

We replaced the +1s in computeCommonSuffix with a -1 in charFromEnd, where it makes perfect sense, and two <= operators in suffixOverlapsPrefix, where they also make perfect sense. This allowed us to change the name of suffixIndex to suffixLength, greatly enhancing the readability of the code.

There is a problem however. As I was eliminating the +1s, I noticed the following line in compactString:

  if (suffixLength > 0)

Take a look at it in [Listing 15-4](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter15.html#ch15lt4). By rights, because suffixLength is now one less than it used to be, I should change the > operator to a >= operator. But that makes no sense. It makes sense now! This means that it didn’t use to make sense and was probably a bug. Well, not quite a bug. Upon further analysis we see that the if statement now prevents a zero length suffix from being appended. Before we made the change, the if statement was nonfunctional because suffixIndex could never be less than one!

This calls into question both if statements in compactString! It looks as though they could both be eliminated. So let’s comment them out and run the tests. They passed! So let’s restructure compactString to eliminate the extraneous if statements and make the function much simpler [G9].

   private String compactString(String source) {  
       return  
         computeCommonPrefix() +  
         DELTA\_START +  
         source.substring(prefixLength, source.length() - suffixLength) +  
         DELTA\_END +  
         computeCommonSuffix();  
   }

This is much better! Now we see that the compactString function is simply composing the fragments together. We can probably make this even clearer. Indeed, there are lots of little cleanups we could do. But rather than drag you through the rest of the changes, I’ll just show you the result in [Listing 15-5](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter15.html#ch15lt5).

**Listing 15-5 ComparisonCompactor.java (final)**

   package junit.framework;  
   public class ComparisonCompactor {  
  
     private static final String ELLIPSIS = “…”;  
     private static final String DELTA\_END = “]”;  
     private static final String DELTA\_START = “[”;  
     private int contextLength;  
     private String expected;  
     private String actual;  
     private int prefixLength;  
     private int suffixLength;  
  
     public ComparisonCompactor(  
       int contextLength, String expected, String actual  
     ) {  
       this.contextLength = contextLength;  
       this.expected = expected;  
       this.actual = actual;  
     }  
  
     public String formatCompactedComparison(String message) {  
       String compactExpected = expected;  
       String compactActual = actual;  
       if (shouldBeCompacted()) {  
         findCommonPrefixAndSuffix();  
         compactExpected = compact(expected);  
         compactActual = compact(actual);  
      }   
      return Assert.format(message, compactExpected, compactActual);  
    }  
  
    private boolean shouldBeCompacted() {  
      return !shouldNotBeCompacted();  
    }  
  
   private boolean shouldNotBeCompacted() {  
     return expected == null ||  
            actual == null ||  
            expected.equals(actual);  
   }  
  
   private void findCommonPrefixAndSuffix() {  
     findCommonPrefix();  
     suffixLength = 0;  
     for (; !suffixOverlapsPrefix(); suffixLength++) {  
       if (charFromEnd(expected, suffixLength) !=  
           charFromEnd(actual, suffixLength)  
       )  
  
         break;  
      }  
   }  
   private char charFromEnd(String s, int i) {  
     return s.charAt(s.length() - i - 1);  
   }  
   private boolean suffixOverlapsPrefix() {  
     return actual.length() - suffixLength <= prefixLength ||  
       expected.length() - suffixLength <= prefixLength;  
   }  
  
   private void findCommonPrefix() {  
     prefixLength = 0;  
     int end = Math.min(expected.length(), actual.length());  
     for (; prefixLength < end; prefixLength++)  
       if (expected.charAt(prefixLength) != actual.charAt(prefixLength))  
         break;  
   }  
  
   private String compact(String s) {  
     return new StringBuilder()  
       .append(startingEllipsis())  
       .append(startingContext())  
       .append(DELTA\_START)  
       .append(delta(s))  
       .append(DELTA\_END)  
       .append(endingContext())  
       .append(endingEllipsis())  
       .toString();  
   }  
   private String startingEllipsis() {  
     return prefixLength > contextLength ? ELLIPSIS : “”;  
   }  
   private String startingContext() {  
     int contextStart = Math.max(0, prefixLength - contextLength);  
     int contextEnd = prefixLength;  
     return expected.substring(contextStart, contextEnd);  
   }  
   private String delta(String s) {  
     int deltaStart = prefixLength;  
     int deltaEnd = s.length() - suffixLength;  
     return s.substring(deltaStart, deltaEnd);  
   }  
   private String endingContext() {  
     int contextStart = expected.length() - suffixLength;  
     int contextEnd =  
       Math.min(contextStart + contextLength, expected.length());  
     return expected.substring(contextStart, contextEnd);  
   }  
   private String endingEllipsis() {  
     return (suffixLength > contextLength ? ELLIPSIS : “”);  
   }  
  }

This is actually quite pretty. The module is separated into a group of analysis functions and another group of synthesis functions. They are topologically sorted so that the definition of each function appears just after it is used. All the analysis functions appear first, and all the synthesis functions appear last.

If you look carefully, you will notice that I reversed several of the decisions I made earlier in this chapter. For example, I inlined some extracted methods back into formatCompactedComparison, and I changed the sense of the shouldNotBeCompacted expression. This is typical. Often one refactoring leads to another that leads to the undoing of the first. Refactoring is an iterative process full of trial and error, inevitably converging on something that we feel is worthy of a professional.

### **CONCLUSION**

And so we have satisfied the Boy Scout Rule. We have left this module a bit cleaner than we found it. Not that it wasn’t clean already. The authors had done an excellent job with it. But no module is immune from improvement, and each of us has the responsibility to leave the code a little better than we found it.

## ****Refactoring**** SerialDate

If you go to http://www.jfree.org/jcommon/index.php, you will find the JCommon library. Deep within that library there is a package named org.jfree.date. Within that package there is a class named SerialDate. We are going to explore that class.

The author of SerialDate is David Gilbert. David is clearly an experienced and competent programmer. As we shall see, he shows a significant degree of professionalism and discipline within his code. For all intents and purposes, this is “good code.” And I am going to rip it to pieces.

This is not an activity of malice. Nor do I think that I am so much better than David that I somehow have a right to pass judgment on his code. Indeed, if you were to find some of my code, I’m sure you could find plenty of things to complain about.

No, this is not an activity of nastiness or arrogance. What I am about to do is nothing more and nothing less than a professional review. It is something that we should all be comfortable doing. And it is something we should welcome when it is done for us. It is only through critiques like these that we will learn. Doctors do it. Pilots do it. Lawyers do it. And we programmers need to learn how to do it too.

One more thing about David Gilbert: David is more than just a good programmer. David had the courage and good will to offer his code to the community at large for free. He placed it out in the open for all to see and invited public usage and public scrutiny. This was well done!

SerialDate ([Listing B-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt1), page [349](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_349)) is a class that represents a date in Java. Why have a class that represents a date, when Java already has java.util.Date and java.util.Calendar, and others? The author wrote this class in response to a pain that I have often felt myself. The comment in his opening Javadoc (line 67) explains it well. We could quibble about his intention, but I have certainly had to deal with this issue, and I welcome a class that is about dates instead of times.

### **FIRST, MAKE IT WORK**

There are some unit tests in a class named SerialDateTests ([Listing B-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt2), page [366](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_366)). The tests all pass. Unfortunately a quick inspection of the tests shows that they don’t test everything [T1]. For example, doing a “Find Usages” search on the method MonthCodeToQuarter (line 334) indicates that it is not used [F4]. Therefore, the unit tests don’t test it.

So I fired up Clover to see what the unit tests covered and what they didn’t. Clover reported that the unit tests executed only 91 of the 185 executable statements in SerialDate (~50 percent) [T2]. The coverage map looks like a patchwork quilt, with big gobs of unexecuted code littered all through the class.

It was my goal to completely understand and also refactor this class. I couldn’t do that without much greater test coverage. So I wrote my own suite of completely independent unit tests ([Listing B-4](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt4), page [374](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_374)).

As you look through these tests, you will note that many of them are commented out. These tests didn’t pass. They represent behavior that I think SerialDate should have. So as I refactor SerialDate, I’ll be working to make these tests pass too.

Even with some of the tests commented out, Clover reports that the new unit tests are executing 170 (92 percent) out of the 185 executable statements. This is pretty good, and I think we’ll be able to get this number higher.

The first few commented-out tests (lines 23-63) were a bit of conceit on my part. The program was not designed to pass these tests, but the behavior seemed obvious [G2] to me. I’m not sure why the testWeekdayCodeToString method was written in the first place, but because it is there, it seems obvious that it should not be case sensitive. Writing these tests was trivial [T3]. Making them pass was even easier; I just changed lines 259 and 263 to use equalsIgnoreCase.

I left the tests at line 32 and line 45 commented out because it’s not clear to me that the “tues” and “thurs” abbreviations ought to be supported.

The tests on line 153 and line 154 don’t pass. Clearly, they should [G2]. We can easily fix this, and the tests on line 163 through line 213, by making the following changes to the stringToMonthCodefunction.

   457     if ((result < 1) || (result > 12)) {  
               result = -1;  
   458         for (int i = 0; i < monthNames.length; i++) {  
   459             if (s.equalsIgnoreCase(shortMonthNames[i])) {  
   460                 result = i + 1;  
   461                 break;  
   462             }  
   463             if (s.equalsIgnoreCase(monthNames[i])) {  
   464                 result = i + 1;  
   465                 break;  
   466             }  
   467         }  
   468     }

The commented test on line 318 exposes a bug in the getFollowingDayOfWeek method (line 672). December 25th, 2004, was a Saturday. The following Saturday was January 1st, 2005. However, when we run the test, we see that getFollowingDayOfWeek returns December 25th as the Saturday that follows December 25th. Clearly, this is wrong [G3],[T1]. We see the problem in line 685. It is a typical boundary condition error [T5]. It should read as follows:

   685     if (baseDOW >= targetWeekday) {

It is interesting to note that this function was the target of an earlier repair. The change history (line 43) shows that “bugs” were fixed in getPreviousDayOfWeek, getFollowingDayOfWeek, and getNearestDayOfWeek [T6].

The testGetNearestDayOfWeek unit test (line 329), which tests the getNearestDayOfWeekmethod (line 705), did not start out as long and exhaustive as it currently is. I added a lot of test cases to it because my initial test cases did not all pass [T6]. You can see the pattern of failure by looking at which test cases are commented out. That pattern is revealing [T7]. It shows that the algorithm fails if the nearest day is in the future. Clearly there is some kind of boundary condition error [T5].

The pattern of test coverage reported by Clover is also interesting [T8]. Line 719 never gets executed! This means that the if statement in line 718 is always false. Sure enough, a look at the code shows that this must be true. The adjust variable is always negative and so cannot be greater or equal to 4. So this algorithm is just wrong.

The right algorithm is shown below:

     int delta = targetDOW - base.getDayOfWeek();  
     int positiveDelta = delta + 7;  
     int adjust = positiveDelta % 7;  
     if (adjust > 3)  
       adjust -= 7;  
  
     return SerialDate.addDays(adjust, base);

Finally, the tests at line 417 and line 429 can be made to pass simply by throwing an IllegalArgumentException instead of returning an error string from weekInMonthToStringand relativeToString.

With these changes all the unit tests pass, and I believe SerialDate now works. So now it’s time to make it “right.”

### **THEN MAKE IT RIGHT**

We are going to walk from the top to the bottom of SerialDate, improving it as we go along. Although you won’t see this in the discussion, I will be running all of the JCommon unit tests, including my improved unit test for SerialDate, after every change I make. So rest assured that every change you see here works for all of JCommon.

Starting at line 1, we see a ream of comments with license information, copyrights, authors, and change history. I acknowledge that there are certain legalities that need to be addressed, and so the copyrights and licenses must stay. On the other hand, the change history is a leftover from the 1960s. We have source code control tools that do this for us now. This history should be deleted [C1].

The import list starting at line 61 could be shortened by using java.text.\* and java.util.\*. [J1]

I wince at the HTML formatting in the Javadoc (line 67). Having a source file with more than one language in it troubles me. This comment has four languages in it: Java, English, Javadoc, and html [G1]. With that many languages in use, it’s hard to keep things straight. For example, the nice positioning of line 71 and line 72 are lost when the Javadoc is generated, and yet who wants to see <ul>and <li> in the source code? A better strategy might be to just surround the whole comment with <pre> so that the formatting that is apparent in the source code is preserved within the Javadoc.1

1. An even better solution would have been for Javadoc to present all comments as preformatted, so that comments appear the same in both code and document.

Line 86 is the class declaration. Why is this class named SerialDate? What is the significance of the word “serial”? Is it because the class is derived from Serializable? That doesn’t seem likely.

I won’t keep you guessing. I know why (or at least I think I know why) the word “serial” was used. The clue is in the constants SERIAL\_LOWER\_BOUND and SERIAL\_UPPER\_BOUND on line 98 and line 101. An even better clue is in the comment that begins on line 830. This class is named SerialDatebecause it is implemented using a “serial number,” which happens to be the number of days since December 30th, 1899.

I have two problems with this. First, the term “serial number” is not really correct. This may be a quibble, but the representation is more of a relative offset than a serial number. The term “serial number” has more to do with product identification markers than dates. So I don’t find this name particularly descriptive [N1]. A more descriptive term might be “ordinal.”

The second problem is more significant. The name SerialDate implies an implementation. This class is an abstract class. There is no need to imply anything at all about the implementation. Indeed, there is good reason to hide the implementation! So I find this name to be at the wrong level of abstraction [N2]. In my opinion, the name of this class should simply be Date.

Unfortunately, there are already too many classes in the Java library named Date, so this is probably not the best name to choose. Because this class is all about days, instead of time, I considered naming it Day, but this name is also heavily used in other places. In the end, I chose DayDate as the best compromise.

From now on in this discussion I will use the term DayDate. I leave it to you to remember that the listings you are looking at still use SerialDate.

I understand why DayDate inherits from Comparable and Serializable. But why does it inherit from MonthConstants? The class MonthConstants ([Listing B-3](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt3), page [372](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_372)) is just a bunch of static final constants that define the months. Inheriting from classes with constants is an old trick that Java programmers used so that they could avoid using expressions like MonthConstants.January, but it’s a bad idea [J2]. MonthConstants should really be an enum.

   public abstract class DayDate implements Comparable,  
                                            Serializable {  
     public static enum Month {  
       JANUARY(1),  
       FEBRUARY(2),  
       MARCH(3),  
       APRIL(4),  
       MAY(5),  
       JUNE(6),  
       JULY(7),  
       AUGUST(8),  
       SEPTEMBER(9),  
       OCTOBER(10),  
       NOVEMBER(11),  
       DECEMBER(12);  
  
       Month(int index) {  
         this.index = index;  
       }  
  
       public static Month make(int monthIndex) {  
         for (Month m : Month.values()) {  
           if (m.index == monthIndex)  
             return m;  
         }  
         throw new IllegalArgumentException(“Invalid month index ” + monthIndex);  
       }  
       public final int index;  
     }

Changing MonthConstants to this enum forces quite a few changes to the DayDate class and all it’s users. It took me an hour to make all the changes. However, any function that used to take an int for a month, now takes a Month enumerator. This means we can get rid of the isValidMonthCode method (line 326), and all the month code error checking such as that in monthCodeToQuarter (line 356) [G5].

Next, we have line 91, serialVersionUID. This variable is used to control the serializer. If we change it, then any DayDate written with an older version of the software won’t be readable anymore and will result in an InvalidClassException. If you don’t declare the serialVersionUID variable, then the compiler automatically generates one for you, and it will be different every time you make a change to the module. I know that all the documents recommend manual control of this variable, but it seems to me that automatic control of serialization is a lot safer [G4]. After all, I’d much rather debug an InvalidClassException than the odd behavior that would ensue if I forgot to change the serialVersionUID. So I’m going to delete the variable—at least for the time being.2

2. Several of the reviewers of this text have taken exception to this decision. They contend that in an open source framework it is better to assert manual control over the serial ID so that minor changes to the software don’t cause old serialized dates to be invalid. This is a fair point. However, at least the failure, inconvenient though it might be, has a clear-cut cause. On the other hand, if the author of the class forgets to update the ID, then the failure mode is undefined and might very well be silent. I think the real moral of this story is that you should not expect to deserialize across versions.

I find the comment on line 93 redundant. Redundant comments are just places to collect lies and misinformation [C2]. So I’m going to get rid of it and its ilk.

The comments at line 97 and line 100 talk about serial numbers, which I discussed earlier [C1]. The variables they describe are the earliest and latest possible dates that DayDate can describe. This can be made a bit clearer [N1].

   public static final int EARLIEST\_DATE\_ORDINAL = 2;     // 1/1/1900  
   public static final int LATEST\_DATE\_ORDINAL = 2958465; // 12/31/9999

It’s not clear to me why EARLIEST\_DATE\_ORDINAL is 2 instead of 0. There is a hint in the comment on line 829 that suggests that this has something to do with the way dates are represented in Microsoft Excel. There is a much deeper insight provided in a derivative of DayDate called SpreadsheetDate([Listing B-5](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt5), page [382](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_382)). The comment on line 71 describes the issue nicely.

The problem I have with this is that the issue seems to be related to the implementation of SpreadsheetDate and has nothing to do with DayDate. I conclude from this that EARLIEST\_DATE\_ORDINAL and LATEST\_DATE\_ORDINAL do not really belong in DayDate and should be moved to SpreadsheetDate [G6].

Indeed, a search of the code shows that these variables are used only within SpreadsheetDate. Nothing in DayDate, nor in any other class in the JCommon framework, uses them. Therefore, I’ll move them down into SpreadsheetDate.

The next variables, MINIMUM\_YEAR\_SUPPORTED, and MAXIMUM\_YEAR\_SUPPORTED (line 104 and line 107), provide something of a dilemma. It seems clear that if DayDate is an abstract class that provides no foreshadowing of implementation, then it should not inform us about a minimum or maximum year. Again, I am tempted to move these variables down into SpreadsheetDate [G6]. However, a quick search of the users of these variables shows that one other class uses them: RelativeDayOfWeekRule ([Listing B-6](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt6), page [390](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_390)). We see that usage at line 177 and line 178 in the getDate function, where they are used to check that the argument to getDate is a valid year. The dilemma is that a user of an abstract class needs information about its implementation.

What we need to do is provide this information without polluting DayDate itself. Usually, we would get implementation information from an instance of a derivative. However, the getDate function is not passed an instance of a DayDate. It does, however, return such an instance, which means that somewhere it must be creating it. Line 187 through line 205 provide the hint. The DayDate instance is being created by one of the three functions, getPreviousDayOfWeek, getNearestDayOfWeek, or getFollowingDayOfWeek. Looking back at the DayDate listing, we see that these functions (lines 638–724) all return a date created by addDays (line 571), which calls createInstance (line 808), which creates a SpreadsheetDate! [G7].

It’s generally a bad idea for base classes to know about their derivatives. To fix this, we should use the ABSTRACT FACTORY3 pattern and create a DayDateFactory. This factory will create the instances of DayDate that we need and can also answer questions about the implementation, such as the maximum and minimum dates.

3. [[GOF](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#GOF)].

   public abstract class DayDateFactory {  
     private static DayDateFactory factory = new SpreadsheetDateFactory();  
     public static void setInstance(DayDateFactory factory) {  
       DayDateFactory.factory = factory;  
     }  
  
     protected abstract DayDate \_makeDate(int ordinal);  
     protected abstract DayDate \_makeDate(int day, DayDate.Month month, int year);  
     protected abstract DayDate \_makeDate(int day, int month, int year);  
     protected abstract DayDate \_makeDate(java.util.Date date);  
     protected abstract int \_getMinimumYear();  
     protected abstract int \_getMaximumYear();  
  
     public static DayDate makeDate(int ordinal) {  
       return factory.\_makeDate(ordinal);  
     }  
     public static DayDate makeDate(int day, DayDate.Month month, int year) {  
       return factory.\_makeDate(day, month, year);  
     }  
  
     public static DayDate makeDate(int day, int month, int year) {  
       return factory.\_makeDate(day, month, year);  
     }  
  
     public static DayDate makeDate(java.util.Date date) {  
       return factory.\_makeDate(date);  
     }  
  
     public static int getMinimumYear() {  
       return factory.\_getMinimumYear();  
     }  
  
     public static int getMaximumYear() {  
       return factory.\_getMaximumYear();  
     }  
   }

This factory class replaces the createInstance methods with makeDate methods, which improves the names quite a bit [N1]. It defaults to a SpreadsheetDateFactory but can be changed at any time to use a different factory. The static methods that delegate to abstract methods use a combination of the SINGLETON,4 DECORATOR,5 and ABSTRACT FACTORY patterns that I have found to be useful.

4. Ibid.

5. Ibid.

The SpreadsheetDateFactory looks like this.

   public class SpreadsheetDateFactory extends DayDateFactory {  
     public DayDate \_makeDate(int ordinal) {  
       return new SpreadsheetDate(ordinal);  
     }  
  
     public DayDate \_makeDate(int day, DayDate.Month month, int year) {  
       return new SpreadsheetDate(day, month, year);  
     }  
  
     public DayDate \_makeDate(int day, int month, int year) {  
       return new SpreadsheetDate(day, month, year);  
     }  
  
     public DayDate \_makeDate(Date date) {  
       final GregorianCalendar calendar = new GregorianCalendar();  
       calendar.setTime(date);  
       return new SpreadsheetDate(  
         calendar.get(Calendar.DATE),  
         DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),  
         calendar.get(Calendar.YEAR));  
     }  
  
     protected int \_getMinimumYear() {  
       return SpreadsheetDate.MINIMUM\_YEAR\_SUPPORTED;  
     }  
  
     protected int \_getMaximumYear() {  
       return SpreadsheetDate.MAXIMUM\_YEAR\_SUPPORTED;  
     }  
   }

As you can see, I have already moved the MINIMUM\_YEAR\_SUPPORTED and MAXIMUM\_YEAR\_SUPPORTED variables into SpreadsheetDate, where they belong [G6].

The next issue in DayDate are the day constants beginning at line 109. These should really be another enum [J3]. We’ve seen this pattern before, so I won’t repeat it here. You’ll see it in the final listings.

Next, we see a series of tables starting with LAST\_DAY\_OF\_MONTH at line 140. My first issue with these tables is that the comments that describe them are redundant [C3]. Their names are sufficient. So I’m going to delete the comments.

There seems to be no good reason that this table isn’t private [G8], because there is a static function lastDayOfMonth that provides the same data.

The next table, AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH, is a bit more mysterious because it is not used anywhere in the JCommon framework [G9]. So I deleted it.

The same goes for LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.

The next table, AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH, is used only in Spread-sheetDate (line 434 and line 473). This begs the question of whether it should be moved to SpreadsheetDate. The argument for not moving it is that the table is not specific to any particular implementation [G6]. On the other hand, no implementation other than SpreadsheetDate actually exists, and so the table should be moved close to where it is used [G10].

What settles the argument for me is that to be consistent [G11], we should make the table private and expose it through a function like julianDateOfLastDayOfMonth. Nobody seems to need a function like that. Moreover, the table can be moved back to DayDate easily if any new implementation of DayDate needs it. So I moved it.

The same goes for the table, LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.

Next, we see three sets of constants that can be turned into enums (lines 162–205). The first of the three selects a week within a month. I changed it into an enum named WeekInMonth.

   public enum WeekInMonth {  
       FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);  
       public final int index;  
  
       WeekInMonth(int index) {  
         this.index = index;  
       }  
     }

The second set of constants (lines 177–187) is a bit more obscure. The INCLUDE\_NONE, INCLUDE\_FIRST, INCLUDE\_SECOND, and INCLUDE\_BOTH constants are used to describe whether the defining end-point dates of a range should be included in that range. Mathematically, this is described using the terms “open interval,” “half-open interval,” and “closed interval.” I think it is clearer using the mathematical nomenclature [N3], so I changed it to an enum named DateIntervalwith CLOSED, CLOSED\_LEFT, CLOSED\_RIGHT, and OPEN enumerators.

The third set of constants (lines 189–205) describe whether a search for a particular day of the week should result in the last, next, or nearest instance. Deciding what to call this is difficult at best. In the end, I settled for WeekdayRange with LAST, NEXT, and NEAREST enumerators.

You might not agree with the names I’ve chosen. They make sense to me, but they may not make sense to you. The point is that they are now in a form that makes them easy to change [J3]. They aren’t passed as integers anymore; they are passed as symbols. I can use the “change name” function of my IDE to change the names, or the types, without worrying that I missed some -1 or 2 somewhere in the code or that some int argument declaration is left poorly described.

The description field at line 208 does not seem to be used by anyone. I deleted it along with its accessor and mutator [G9].

I also deleted the degenerate default constructor at line 213 [G12]. The compiler will generate it for us.

We can skip over the isValidWeekdayCode method (lines 216–238) because we deleted it when we created the Day enumeration.

This brings us to the stringToWeekdayCode method (lines 242–270). Javadocs that don’t add much to the method signature are just clutter [C3],[G12]. The only value this Javadoc adds is the description of the -1 return value. However, because we changed to the Day enumeration, the comment is actually wrong [C2]. The method now throws an IllegalArgumentException. So I deleted the Javadoc.

I also deleted all the final keywords in arguments and variable declarations. As far as I could tell, they added no real value but did add to the clutter [G12]. Eliminating final flies in the face of some conventional wisdom. For example, Robert Simmons6 strongly recommends us to “. . . spread finalall over your code.” Clearly I disagree. I think that there are a few good uses for final, such as the occasional final constant, but otherwise the keyword adds little value and creates a lot of clutter. Perhaps I feel this way because the kinds of errors that final might catch are already caught by the unit tests I write.

6. [[Simmons04](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#Simmons04)], p. [73](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter04.html#page_73).

I didn’t care for the duplicate if statements [G5] inside the for loop (line 259 and line 263), so I connected them into a single if statement using the || operator. I also used the Day enumeration to direct the for loop and made a few other cosmetic changes.

It occurred to me that this method does not really belong in DayDate. It’s really the parse function of Day. So I moved it into the Day enumeration. However, that made the Day enumeration pretty large. Because the concept of Day does not depend on DayDate, I moved the Day enumeration outside of the DayDate class into its own source file [G13].

I also moved the next function, weekdayCodeToString (lines 272–286) into the Day enumeration and called it toString.

   public enum Day {  
     MONDAY(Calendar.MONDAY),  
     TUESDAY(Calendar.TUESDAY),  
     WEDNESDAY(Calendar.WEDNESDAY),s  
     THURSDAY(Calendar.THURSDAY),  
     FRIDAY(Calendar.FRIDAY),  
     SATURDAY(Calendar.SATURDAY),  
     SUNDAY(Calendar.SUNDAY);  
  
     public final int index;  
     private static DateFormatSymbols dateSymbols = new DateFormatSymbols();  
  
     Day(int day) {  
       index = day;  
     }  
  
     public static Day make(int index) throws IllegalArgumentException {  
       for (Day d : Day.values())  
         if (d.index == index)  
           return d;  
       throw new IllegalArgumentException(  
         String.format(“Illegal day index: %d.”, index));  
     }  
  
     public static Day parse(String s) throws IllegalArgumentException {  
       String[] shortWeekdayNames =  
         dateSymbols.getShortWeekdays();  
       String[] weekDayNames =  
         dateSymbols.getWeekdays();  
  
       s = s.trim();  
       for (Day day : Day.values()) {  
         if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||  
             s.equalsIgnoreCase(weekDayNames[day.index])) {  
           return day;  
         }  
       }  
       throw new IllegalArgumentException(  
         String.format(“%s is not a valid weekday string”, s));  
      }  
  
      public String toString() {  
        return dateSymbols.getWeekdays()[index];  
      }  
    }

There are two getMonths functions (lines 288–316). The first calls the second. The second is never called by anyone but the first. Therefore, I collapsed the two into one and vastly simplified them [G9],[G12],[F4]. Finally, I changed the name to be a bit more self-descriptive [N1].

   public static String[] getMonthNames() {  
     return dateFormatSymbols.getMonths();  
   }

The isValidMonthCode function (lines 326–346) was made irrelevant by the Month enum, so I deleted it [G9].

The monthCodeToQuarter function (lines 356–375) smells of FEATURE ENVY7 [G14] and probably belongs in the Month enum as a method named quarter. So I replaced it.

7. [[Refactoring](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#Refactoring)].

   public int quarter() {  
     return 1 + (index-1)/3;  
   }

This made the Month enum big enough to be in its own class. So I moved it out of DayDate to be consistent with the Day enum [G11],[G13].

The next two methods are named monthCodeToString (lines 377–426). Again, we see the pattern of one method calling its twin with a flag. It is usually a bad idea to pass a flag as an argument to a function, especially when that flag simply selects the format of the output [G15]. I renamed, simplified, and restructured these functions and moved them into the Month enum [N1],[N3],[C3],[G14].

   public String toString() {  
     return dateFormatSymbols.getMonths()[index - 1];  
   }  
  
   public String toShortString() {  
     return dateFormatSymbols.getShortMonths()[index - 1];  
   }

The next method is stringToMonthCode (lines 428–472). I renamed it, moved it into the Monthenum, and simplified it [N1],[N3],[C3],[G14],[G12].

   public static Month parse(String s) {  
     s = s.trim();  
     for (Month m : Month.values())  
       if (m.matches(s))  
         return m;  
  
     try {  
       return make(Integer.parseInt(s));  
     }  
     catch (NumberFormatException e) {}  
     throw new IllegalArgumentException(“Invalid month ” + s);  
   }  
  
   private boolean matches(String s) {  
     return s.equalsIgnoreCase(toString()) ||  
            s.equalsIgnoreCase(toShortString());  
   }

The isLeapYear method (lines 495–517) can be made a bit more expressive [G16].

   public static boolean isLeapYear(int year) {  
     boolean fourth = year % 4 == 0;  
     boolean hundredth = year % 100 == 0;  
     boolean fourHundredth = year % 400 == 0;  
     return fourth && (!hundredth || fourHundredth);  
   }

The next function, leapYearCount (lines 519–536) doesn’t really belong in DayDate. Nobody calls it except for two methods in SpreadsheetDate. So I pushed it down [G6].

The lastDayOfMonth function (lines 538–560) makes use of the LAST\_DAY\_OF\_MONTH array. This array really belongs in the Month enum [G17], so I moved it there. I also simplified the function and made it a bit more expressive [G16].

   public static int lastDayOfMonth(Month month, int year) {  
     if (month == Month.FEBRUARY && isLeapYear(year))  
       return month.lastDay() + 1;  
      else  
       return month.lastDay();  
   }

Now things start to get a bit more interesting. The next function is addDays (lines 562–576). First of all, because this function operates on the variables of DayDate, it should not be static [G18]. So I changed it to an instance method. Second, it calls the function toSerial. This function should be renamed toOrdinal [N1]. Finally, the method can be simplified.

   public DayDate addDays(int days) {  
     return DayDateFactory.makeDate(toOrdinal() + days);  
   }

The same goes for addMonths (lines 578–602). It should be an instance method [G18]. The algorithm is a bit complicated, so I used EXPLAINING TEMPORARY VARIABLES8 [G19] to make it more transparent. I also renamed the method getYYY to getYear [N1].

8. [[Beck97](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#Beck97)].

   public DayDate addMonths(int months) {  
     int thisMonthAsOrdinal = 12 \* getYear() + getMonth().index - 1;  
     int resultMonthAsOrdinal = thisMonthAsOrdinal + months;  
     int resultYear = resultMonthAsOrdinal / 12;  
     Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);  
  
     int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);  
     int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);  
     return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);  
   }

The addYears function (lines 604–626) provides no surprises over the others.

   public DayDate plusYears(int years) {  
     int resultYear = getYear() + years;  
     int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);  
     int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);  
     return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);  
   }

There is a little itch at the back of my mind that is bothering me about changing these methods from static to instance. Does the expression date.addDays(5) make it clear that the date object does not change and that a new instance of DayDate is returned? Or does it erroneously imply that we are adding five days to the date object? You might not think that is a big problem, but a bit of code that looks like the following can be very deceiving [G20].

   DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);  
   date.addDays(7); // bump date by one week.

Someone reading this code would very likely just accept that addDays is changing the date object. So we need a name that breaks this ambiguity [N4]. So I changed the names to plusDays and plusMonths. It seems to me that the intent of the method is captured nicely by

   DayDate date = oldDate.plusDays(5);

whereas the following doesn’t read fluidly enough for a reader to simply accept that the date object is changed:

   date.plusDays(5);

The algorithms continue to get more interesting. getPreviousDayOfWeek (lines 628–660) works but is a bit complicated. After some thought about what was really going on [G21], I was able to simplify it and use EXPLAINING TEMPORARY VARIABLES [G19] to make it clearer. I also changed it from a static method to an instance method [G18], and got rid of the duplicate instance method [G5] (lines 997–1008).

   public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {  
     int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;  
     if (offsetToTarget >= 0)  
       offsetToTarget -= 7;  
     return plusDays(offsetToTarget);  
   }

The exact same analysis and result occurred for getFollowingDayOfWeek (lines 662–693).

   public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {  
       int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;  
       if (offsetToTarget <= 0)  
  
         offsetToTarget += 7;  
       return plusDays(offsetToTarget);  
     }

The next function is getNearestDayOfWeek (lines 695–726), which we corrected back on page [270](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#page_270). But the changes I made back then aren’t consistent with the current pattern in the last two functions [G11]. So I made it consistent and used some EXPLAINING TEMPORARY VARIABLES [G19] to clarify the algorithm.

   public DayDate getNearestDayOfWeek(final Day targetDay) {  
       int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;  
       int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;  
       int offsetToPreviousTarget = offsetToFutureTarget - 7;  
  
       if (offsetToFutureTarget > 3)  
         return plusDays(offsetToPreviousTarget);  
       else  
         return plusDays(offsetToFutureTarget);  
     }

The getEndOfCurrentMonth method (lines 728–740) is a little strange because it is an instance method that envies [G14] its own class by taking a DayDate argument. I made it a true instance method and clarified a few names.

   public DayDate getEndOfMonth() {  
       Month month = getMonth();  
       int year = getYear();  
       int lastDay = lastDayOfMonth(month, year);  
       return DayDateFactory.makeDate(lastDay, month, year);  
     }

Refactoring weekInMonthToString (lines 742–761) turned out to be very interesting indeed. Using the refactoring tools of my IDE, I first moved the method to the WeekInMonth enum that I created back on page [275](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#page_275). Then I renamed the method to toString. Next, I changed it from a static method to an instance method. All the tests still passed. (Can you guess where I am going?)

Next, I deleted the method entirely! Five asserts failed (lines 411–415, [Listing B-4](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt4), page [374](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_374)). I changed these lines to use the names of the enumerators (FIRST, SECOND, …). All the tests passed. Can you see why? Can you also see why each of these steps was necessary? The refactoring tool made sure that all previous callers of weekInMonthToString now called toString on the weekInMonthenumerator because all enumerators implement toString to simply return their names.…

Unfortunately, I was a bit too clever. As elegant as that wonderful chain of refactorings was, I finally realized that the only users of this function were the tests I had just modified, so I deleted the tests.

Fool me once, shame on you. Fool me twice, shame on me! So after determining that nobody other than the tests called relativeToString (lines 765–781), I simply deleted the function and its tests.

We have finally made it to the abstract methods of this abstract class. And the first one is as appropriate as they come: toSerial (lines 838–844). Back on page [279](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#page_279) I had changed the name to toOrdinal. Having looked at it in this context, I decided the name should be changed to getOrdinalDay.

The next abstract method is toDate (lines 838–844). It converts a DayDate to a java.util.Date. Why is this method abstract? If we look at its implementation in SpreadsheetDate (lines 198–207, [Listing B-5](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt5), page [382](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_382)), we see that it doesn’t depend on anything in the implementation of that class [G6]. So I pushed it up.

The getYYYY, getMonth, and getDayOfMonth methods are nicely abstract. However, the getDayOfWeek method is another one that should be pulled up from SpreadSheetDate because it doesn’t depend on anything that can’t be found in DayDate [G6]. Or does it?

If you look carefully (line 247, [Listing B-5](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt5), page [382](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_382)), you’ll see that the algorithm implicitly depends on the origin of the ordinal day (in other words, the day of the week of day 0). So even though this function has no physical dependencies that couldn’t be moved to DayDate, it does have a logical dependency.

Logical dependencies like this bother me [G22]. If something logical depends on the implementation, then something physical should too. Also, it seems to me that the algorithm itself could be generic with a much smaller portion of it dependent on the implementation [G6].

So I created an abstract method in DayDate named getDayOfWeekForOrdinalZero and implemented it in SpreadsheetDate to return Day.SATURDAY. Then I moved the getDayOfWeekmethod up to DayDate and changed it to call getOrdinalDay and getDayOfWeekForOrdinal-Zero.

   public Day getDayOfWeek() {  
       Day startingDay = getDayOfWeekForOrdinalZero();  
       int startingOffset = startingDay.index - Day.SUNDAY.index;  
       return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);  
     }

As a side note, look carefully at the comment on line 895 through line 899. Was this repetition really necessary? As usual, I deleted this comment along with all the others.

The next method is compare (lines 902–913). Again, this method is inappropriately abstract [G6], so I pulled the implementation up into DayDate. Also, the name does not communicate enough [N1]. This method actually returns the difference in days since the argument. So I changed the name to daysSince. Also, I noted that there weren’t any tests for this method, so I wrote them.

The next six functions (lines 915–980) are all abstract methods that should be implemented in DayDate. So I pulled them all up from SpreadsheetDate.

The last function, isInRange (lines 982–995) also needs to be pulled up and refactored. The switchstatement is a bit ugly [G23] and can be replaced by moving the cases into the DateInterval enum.

   public enum DateInterval {  
       OPEN {  
         public boolean isIn(int d, int left, int right) {  
           return d > left && d < right;  
         }  
       },  
       CLOSED\_LEFT {  
         public boolean isIn(int d, int left, int right) {  
           return d >= left && d < right;  
         }  
       },  
       CLOSED\_RIGHT {  
         public boolean isIn(int d, int left, int right) {  
           return d > left && d <= right;  
         }  
       },  
       CLOSED {  
         public boolean isIn(int d, int left, int right) {  
           return d >= left && d <= right;  
         }  
       };  
  
       public abstract boolean isIn(int d, int left, int right);  
     }

   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {  
       int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());  
       int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());  
       return interval.isIn(getOrdinalDay(), left, right);  
     }

That brings us to the end of DayDate. So now we’ll make one more pass over the whole class to see how well it flows.

First, the opening comment is long out of date, so I shortened and improved it [C2].

Next, I moved all the remaining enums out into their own files [G12].

Next, I moved the static variable (dateFormatSymbols) and three static methods (getMonthNames, isLeapYear, lastDayOfMonth) into a new class named DateUtil [G6].

I moved the abstract methods up to the top where they belong [G24].

I changed Month.make to Month.fromInt [N1] and did the same for all the other enums. I also created a toInt() accessor for all the enums and made the index field private.

There was some interesting duplication [G5] in plusYears and plusMonths that I was able to eliminate by extracting a new method named correctLastDayOfMonth, making the all three methods much clearer.

I got rid of the magic number 1 [G25], replacing it with Month.JANUARY.toInt() or Day.SUNDAY.toInt(), as appropriate. I spent a little time with SpreadsheetDate, cleaning up the algorithms a bit. The end result is contained in [Listing B-7](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt7), page [394](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_394), through [Listing B-16](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt16), page [405](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_405).

Interestingly the code coverage in DayDate has decreased to 84.9 percent! This is not because less functionality is being tested; rather it is because the class has shrunk so much that the few uncovered lines have a greater weight. DayDate now has 45 out of 53 executable statements covered by tests. The uncovered lines are so trivial that they weren’t worth testing.

### **CONCLUSION**

So once again we’ve followed the Boy Scout Rule. We’ve checked the code in a bit cleaner than when we checked it out. It took a little time, but it was worth it. Test coverage was increased, some bugs were fixed, the code was clarified and shrunk. The next person to look at this code will hopefully find it easier to deal with than we did. That person will also probably be able to clean it up a bit more than we did.

## ****Smells and Heuristics****

In his wonderful book Refactoring,1 Martin Fowler identified many different “Code Smells.” The list that follows includes many of Martin’s smells and adds many more of my own. It also includes other pearls and heuristics that I use to practice my trade.

1. [[Refactoring](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#Refactoring)].

I compiled this list by walking through several different programs and refactoring them. As I made each change, I asked myself why I made that change and then wrote the reason down here. The result is a rather long list of things that smell bad to me when I read code.

This list is meant to be read from top to bottom and also to be used as a reference. There is a cross-reference for each heuristic that shows you where it is referenced in the rest of the text in “Appendix C” on page [409](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app03.html#page_409).

### **COMMENTS**

#### **C1:** Inappropriate Information

It is inappropriate for a comment to hold information better held in a different kind of system such as your source code control system, your issue tracking system, or any other record-keeping system. Change histories, for example, just clutter up source files with volumes of historical and uninteresting text. In general, meta-data such as authors, last-modified-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.

#### **C2:** Obsolete Comment

A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. If you find an obsolete comment, it is best to update it or get rid of it as quickly as possible. Obsolete comments tend to migrate away from the code they once described. They become floating islands of irrelevance and misdirection in the code.

#### **C3:** Redundant Comment

A comment is redundant if it describes something that adequately describes itself. For example:

   i++; // increment i

Another example is a Javadoc that says nothing more than (or even less than) the function signature:

   /\*\*  
    \* @param sellRequest  
    \* @return  
    \* @throws ManagedComponentException  
    \*/  
   public SellResponse beginSellItem(SellRequest sellRequest)  
     throws ManagedComponentException

Comments should say things that the code cannot say for itself.

#### **C4:** Poorly Written Comment

A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write. Choose your words carefully. Use correct grammar and punctuation. Don’t ramble. Don’t state the obvious. Be brief.

#### **C5:** Commented-Out Code

It makes me crazy to see stretches of code that are commented out. Who knows how old it is? Who knows whether or not it’s meaningful? Yet no one will delete it because everyone assumes someone else needs it or has plans for it.

That code sits there and rots, getting less and less relevant with every passing day. It calls functions that no longer exist. It uses variables whose names have changed. It follows conventions that are long obsolete. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code is an abomination.

When you see commented-out code, delete it! Don’t worry, the source code control system still remembers it. If anyone really needs it, he or she can go back and check out a previous version. Don’t suffer commented-out code to survive.

### **ENVIRONMENT**

#### **E1:** Build Requires More Than One Step

Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control. You should not need a sequence of arcane commands or context dependent scripts in order to build the individual elements. You should not have to search near and far for all the various little extra JARs, XML files, and other artifacts that the system requires. You should be able to check out the system with one simple command and then issue one other simple command to build it.

   svn get mySystem  
   cd mySystem  
   ant all

#### **E2:** Tests Require More Than One Step

You should be able to run all the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE. In the worst case you should be able to issue a single simple command in a shell. Being able to run all the tests is so fundamental and so important that it should be quick, easy, and obvious to do.

### **FUNCTIONS**

#### **F1:** Too Many Arguments

Functions should have a small number of arguments. No argument is best, followed by one, two, and three. More than three is very questionable and should be avoided with prejudice. (See “[Function Arguments](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lev1sec6)” on page [40](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#page_40).)

#### **F2:** Output Arguments

Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on. (See “[Output Arguments](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lev2sec11)” on page [45](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#page_45).)

#### **F3:** Flag Arguments

Boolean arguments loudly declare that the function does more than one thing. They are confusing and should be eliminated. (See “[Flag Arguments](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lev2sec5)” on page [41](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#page_41).)

#### **F4:** Dead Function

Methods that are never called should be discarded. Keeping dead code around is wasteful. Don’t be afraid to delete the function. Remember, your source code control system still remembers it.

### **GENERAL**

#### **G1:** Multiple Languages in One Source File

Today’s modern programming environments make it possible to put many different languages into a single source file. For example, a Java source file might contain snippets of XML, HTML, YAML, JavaDoc, English, JavaScript, and so on. For another example, in addition to HTML a JSP file might contain Java, a tag library syntax, English comments, Javadocs, XML, JavaScript, and so forth. This is confusing at best and carelessly sloppy at worst.

The ideal is for a source file to contain one, and only one, language. Realistically, we will probably have to use more than one. But we should take pains to minimize both the number and extent of extra languages in our source files.

#### **G2:** Obvious Behavior Is Unimplemented

Following “The Principle of Least Surprise,”2 any function or class should implement the behaviors that another programmer could reasonably expect. For example, consider a function that translates the name of a day to an enum that represents the day.

2. Or “The Principle of Least Astonishment”: http://en.wikipedia.org/wiki/  
         Principle\_of\_least\_astonishment

   Day day = DayDate.StringToDay(String dayName);

We would expect the string “Monday” to be translated to Day.MONDAY. We would also expect the common abbreviations to be translated, and we would expect the function to ignore case.

When an obvious behavior is not implemented, readers and users of the code can no longer depend on their intuition about function names. They lose their trust in the original author and must fall back on reading the details of the code.

#### **G3:** Incorrect Behavior at the Boundaries

It seems obvious to say that code should behave correctly. The problem is that we seldom realize just how complicated correct behavior is. Developers often write functions that they think will work, and then trust their intuition rather than going to the effort to prove that their code works in all the corner and boundary cases.

There is no replacement for due diligence. Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. Don’t rely on your intuition. Look for every boundary condition and write a test for it.

#### **G4:** Overridden Safeties

Chernobyl melted down because the plant manager overrode each of the safety mechanisms one by one. The safeties were making it inconvenient to run an experiment. The result was that the experiment did not get run, and the world saw it’s first major civilian nuclear catastrophe.

It is risky to override safeties. Exerting manual control over serialVersionUID may be necessary, but it is always risky. Turning off certain compiler warnings (or all warnings!) may help you get the build to succeed, but at the risk of endless debugging sessions. Turning off failing tests and telling yourself you’ll get them to pass later is as bad as pretending your credit cards are free money.

#### **G5:** Duplication

This is one of the most important rules in this book, and you should take it very seriously. Virtually every author who writes about software design mentions this rule. Dave Thomas and Andy Hunt called it the DRY3 principle (Don’t Repeat Yourself). Kent Beck made it one of the core principles of Extreme Programming and called it: “Once, and only once.” Ron Jeffries ranks this rule second, just below getting all the tests to pass.

3. [[PRAG](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#PRAG)].

Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design. Other programmers can use the abstract facilities you create. Coding becomes faster and less error prone because you have raised the abstraction level.

The most obvious form of duplication is when you have clumps of identical code that look like some programmers went wild with the mouse, pasting the same code over and over again. These should be replaced with simple methods.

A more subtle form is the switch/case or if/else chain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.

Still more subtle are the modules that have similar algorithms, but that don’t share similar lines of code. This is still duplication and should be addressed by using the TEMPLATE METHOD,4 or STRATEGY5pattern.

4. [[GOF](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#GOF)].

5. [[GOF](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#GOF)].

Indeed, most of the design patterns that have appeared in the last fifteen years are simply well-known ways to eliminate duplication. So too the Codd Normal Forms are a strategy for eliminating duplication in database schemae. OO itself is a strategy for organizing modules and eliminating duplication. Not surprisingly, so is structured programming.

I think the point has been made. Find and eliminate duplication wherever you can.

#### **G6:** Code at Wrong Level of Abstraction

It is important to create abstractions that separate higher level general concepts from lower level detailed concepts. Sometimes we do this by creating abstract classes to hold the higher level concepts and derivatives to hold the lower level concepts. When we do this, we need to make sure that the separation is complete. We want all the lower level concepts to be in the derivatives and all the higher level concepts to be in the base class.

For example, constants, variables, or utility functions that pertain only to the detailed implementation should not be present in the base class. The base class should know nothing about them.

This rule also pertains to source files, components, and modules. Good software design requires that we separate concepts at different levels and place them in different containers. Sometimes these containers are base classes or derivatives and sometimes they are source files, modules, or components. Whatever the case may be, the separation needs to be complete. We don’t want lower and higher level concepts mixed together.

Consider the following code:

   public interface Stack {  
     Object pop() throws EmptyException;  
     void push(Object o) throws FullException;  
     double percentFull();  
  
     class EmptyException extends Exception {}  
     class FullException extends Exception {}  
   }

The percentFull function is at the wrong level of abstraction. Although there are many implementations of Stack where the concept of fullness is reasonable, there are other implementations that simply could not know how full they are. So the function would be better placed in a derivative interface such as BoundedStack.

Perhaps you are thinking that the implementation could just return zero if the stack were boundless. The problem with that is that no stack is truly boundless. You cannot really prevent an OutOfMemoryException by checking for

   stack.percentFull() < 50.0.

Implementing the function to return 0 would be telling a lie.

The point is that you cannot lie or fake your way out of a misplaced abstraction. Isolating abstractions is one of the hardest things that software developers do, and there is no quick fix when you get it wrong.

#### **G7:** Base Classes Depending on Their Derivatives

The most common reason for partitioning concepts into base and derivative classes is so that the higher level base class concepts can be independent of the lower level derivative class concepts. Therefore, when we see base classes mentioning the names of their derivatives, we suspect a problem. In general, base classes should know nothing about their derivatives.

There are exceptions to this rule, of course. Sometimes the number of derivatives is strictly fixed, and the base class has code that selects between the derivatives. We see this a lot in finite state machine implementations. However, in that case the derivatives and base class are strongly coupled and always deploy together in the same jar file. In the general case we want to be able to deploy derivatives and bases in different jar files.

Deploying derivatives and bases in different jar files and making sure the base jar files know nothing about the contents of the derivative jar files allow us to deploy our systems in discrete and independent components. When such components are modified, they can be redeployed without having to redeploy the base components. This means that the impact of a change is greatly lessened, and maintaining systems in the field is made much simpler.

#### **G8:** Too Much Information

Well-defined modules have very small interfaces that allow you to do a lot with a little. Poorly defined modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high.

Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better.

Hide your data. Hide your utility functions. Hide your constants and your temporaries. Don’t create classes with lots of methods or lots of instance variables. Don’t create lots of protected variables and functions for your subclasses. Concentrate on keeping interfaces very tight and very small. Help keep coupling low by limiting information.

#### **G9:** Dead Code

Dead code is code that isn’t executed. You find it in the body of an if statement that checks for a condition that can’t happen. You find it in the catch block of a try that never throws. You find it in little utility methods that are never called or switch/case conditions that never occur.

The problem with dead code is that after awhile it starts to smell. The older it is, the stronger and sourer the odor becomes. This is because dead code is not completely updated when designs change. It still compiles, but it does not follow newer conventions or rules. It was written at a time when the system was different. When you find dead code, do the right thing. Give it a decent burial. Delete it from the system.

#### **G10:** Vertical Separation

Variables and function should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope. We don’t want local variables declared hundreds of lines distant from their usages.

Private functions should be defined just below their first usage. Private functions belong to the scope of the whole class, but we’d still like to limit the vertical distance between the invocations and definitions. Finding a private function should just be a matter of scanning downward from the first usage.

#### **G11:** Inconsistency

If you do something a certain way, do all similar things in the same way. This goes back to the principle of least surprise. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.

If within a particular function you use a variable named response to hold an HttpServletResponse, then use the same variable name consistently in the other functions that use HttpServletResponse objects. If you name a method processVerificationRequest, then use a similar name, such as processDeletionRequest, for the methods that process other kinds of requests.

Simple consistency like this, when reliably applied, can make code much easier to read and modify.

#### **G12:** Clutter

Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Variables that aren’t used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well organized, and free of clutter.

#### **G13:** Artificial Coupling

Things that don’t depend upon each other should not be artificially coupled. For example, general enums should not be contained within more specific classes because this forces the whole application to know about these more specific classes. The same goes for general purpose static functions being declared in specific classes.

In general an artificial coupling is a coupling between two modules that serves no direct purpose. It is a result of putting a variable, constant, or function in a temporarily convenient, though inappropriate, location. This is lazy and careless.

Take the time to figure out where functions, constants, and variables ought to be declared. Don’t just toss them in the most convenient place at hand and then leave them there.

#### **G14:** Feature Envy

This is one of Martin Fowler’s code smells.6 The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it envies the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating. For example:

6. [[Refactoring](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#Refactoring)].

   public class HourlyPayCalculator {  
     public Money calculateWeeklyPay(HourlyEmployee e) {  
       int tenthRate = e.getTenthRate().getPennies();  
       int tenthsWorked = e.getTenthsWorked();  
       int straightTime = Math.min(400, tenthsWorked);  
       int overTime = Math.max(0, tenthsWorked - straightTime);  
       int straightPay = straightTime \* tenthRate;  
       int overtimePay = (int)Math.round(overTime\*tenthRate\*1.5);  
       return new Money(straightPay + overtimePay);  
     }  
   }

The calculateWeeklyPay method reaches into the HourlyEmployee object to get the data on which it operates. The calculateWeeklyPay method envies the scope of HourlyEmployee. It “wishes” that it could be inside HourlyEmployee.

All else being equal, we want to eliminate Feature Envy because it exposes the internals of one class to another. Sometimes, however, Feature Envy is a necessary evil. Consider the following:

   public class HourlyEmployeeReport {  
     private HourlyEmployee employee ;  
  
     public HourlyEmployeeReport(HourlyEmployee e) {  
       this.employee = e;  
     }  
  
     String reportHours() {  
       return String.format(  
         “Name: %s\tHours:%d.%1d\n”,  
         employee.getName(),  
         employee.getTenthsWorked()/10,  
         employee.getTenthsWorked()%10);  
     }  
   }

Clearly, the reportHours method envies the HourlyEmployee class. On the other hand, we don’t want HourlyEmployee to have to know about the format of the report. Moving that format string into the HourlyEmployee class would violate several principles of object oriented design.7 It would couple HourlyEmployee to the format of the report, exposing it to changes in that format.

7. Specifically, the Single Responsibility Principle, the Open Closed Principle, and the Common Closure Principle. See [[PPP](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#PPP)].

#### **G15:** Selector Arguments

There is hardly anything more abominable than a dangling false argument at the end of a function call. What does it mean? What would it change if it were true? Not only is the purpose of a selector argument difficult to remember, each selector argument combines many functions into one. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions. Consider:

   public int calculateWeeklyPay(boolean overtime) {  
     int tenthRate = getTenthRate();  
     int tenthsWorked = getTenthsWorked();  
     int straightTime = Math.min(400, tenthsWorked);  
     int overTime = Math.max(0, tenthsWorked - straightTime);  
     int straightPay = straightTime \* tenthRate;  
     double overtimeRate = overtime ? 1.5 : 1.0 \* tenthRate;  
     int overtimePay = (int)Math.round(overTime\*overtimeRate);  
     return straightPay + overtimePay;  
   }

You call this function with a true if overtime is paid as time and a half, and with a false if overtime is paid as straight time. It’s bad enough that you must remember what calculateWeeklyPay(false)means whenever you happen to stumble across it. But the real shame of a function like this is that the author missed the opportunity to write the following:

   public int straightPay() {  
     return getTenthsWorked() \* getTenthRate();  
   }  
  
   public int overTimePay() {  
     int overTimeTenths = Math.max(0, getTenthsWorked() - 400);  
     int overTimePay = overTimeBonus(overTimeTenths);  
     return straightPay() + overTimePay;  
   }  
  
   private int overTimeBonus(int overTimeTenths) {  
     double bonus = 0.5 \* getTenthRate() \* overTimeTenths;  
     return (int) Math.round(bonus);  
   }

Of course, selectors need not be boolean. They can be enums, integers, or any other type of argument that is used to select the behavior of the function. In general it is better to have many functions than to pass some code into a function to select the behavior.

#### **G16:** Obscured Intent

We want code to be as expressive as possible. Run-on expressions, Hungarian notation, and magic numbers all obscure the author’s intent. For example, here is the overTimePay function as it might have appeared:

   public int m\_otCalc() {  
   return iThsWkd \* iThsRte +  
     (int) Math.round(0.5 \* iThsRte \*  
       Math.max(0, iThsWkd - 400)  
     );  
   }

Small and dense as this might appear, it’s also virtually impenetrable. It is worth taking the time to make the intent of our code visible to our readers.

#### **G17:** Misplaced Responsibility

One of the most important decisions a software developer can make is where to put code. For example, where should the PI constant go? Should it be in the Math class? Perhaps it belongs in the Trigonometry class? Or maybe in the Circle class?

The principle of least surprise comes into play here. Code should be placed where a reader would naturally expect it to be. The PI constant should go where the trig functions are declared. The OVERTIME\_RATE constant should be declared in the HourlyPay-Calculator class.

Sometimes we get “clever” about where to put certain functionality. We’ll put it in a function that’s convenient for us, but not necessarily intuitive to the reader. For example, perhaps we need to print a report with the total of hours that an employee worked. We could sum up those hours in the code that prints the report, or we could try to keep a running total in the code that accepts time cards.

One way to make this decision is to look at the names of the functions. Let’s say that our report module has a function named getTotalHours. Let’s also say that the module that accepts time cards has a saveTimeCard function. Which of these two functions, by it’s name, implies that it calculates the total? The answer should be obvious.

Clearly, there are sometimes performance reasons why the total should be calculated as time cards are accepted rather than when the report is printed. That’s fine, but the names of the functions ought to reflect this. For example, there should be a computeRunning-TotalOfHours function in the timecard module.

#### **G18:** Inappropriate Static

Math.max(double a, double b) is a good static method. It does not operate on a single instance; indeed, it would be silly to have to say new Math().max(a,b) or even a.max(b). All the data that max uses comes from its two arguments, and not from any “owning” object. More to the point, there is almost no chance that we’d want Math.max to be polymorphic.

Sometimes, however, we write static functions that should not be static. For example, consider:

   HourlyPayCalculator.calculatePay(employee, overtimeRate).

Again, this seems like a reasonable static function. It doesn’t operate on any particular object and gets all it’s data from it’s arguments. However, there is a reasonable chance that we’ll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay, for example, OvertimeHourlyPayCalculator and StraightTimeHourlyPayCalculator. So in this case the function should not be static. It should be a nonstatic member function of Employee.

In general you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you’ll want it to behave polymorphically.

#### **G19:** Use Explanatory Variables

Kent Beck wrote about this in his great book Smalltalk Best Practice Patterns8 and again more recently in his equally great book Implementation Patterns.9 One of the more powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names.

8. [[Beck97](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#Beck97)], p. [108](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter07.html#page_108).

9. [[Beck07](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#Beck07)].

Consider this example from FitNesse:

   Matcher match = headerPattern.matcher(line);  
   if(match.find())  
   {  
     String key = match.group(1);  
     String value = match.group(2);  
     headers.put(key.toLowerCase(), value);  
   }

The simple use of explanatory variables makes it clear that the first matched group is the key, and the second matched group is the value.

It is hard to overdo this. More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by breaking the calculations up into well-named intermediate values.

#### **G20:** Function Names Should Say What They Do

[Image](https://ssl.safaribooksonline.com/9780134661742/CODE_01_02_00)

Look at this code:

   Date newDate = date.add(5);

Would you expect this to add five days to the date? Or is it weeks, or hours? Is the date instance changed or does the function just return a new Date without changing the old one? You can’t tell from the call what the function does.

If the function adds five days to the date and changes the date, then it should be called addDaysTo or increaseByDays. If, on the other hand, the function returns a new date that is five days later but does not change the date instance, it should be called daysLater or daysSince.

If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

#### **G21:** Understand the Algorithm

Lots of very funny code is written because people don’t take the time to understand the algorithm. They get something to work by plugging in enough if statements and flags, without really stopping to consider what is really going on.

Programming is often an exploration. You think you know the right algorithm for something, but then you wind up fiddling with it, prodding and poking at it, until you get it to “work.” How do you know it “works”? Because it passes the test cases you can think of.

There is nothing wrong with this approach. Indeed, often it is the only way to get a function to do what you think it should. However, it is not sufficient to leave the quotation marks around the word “work.”

Before you consider yourself to be done with a function, make sure you understand how it works. It is not good enough that it passes all the tests. You must know10 that the solution is correct.

10. There is a difference between knowing how the code works and knowing whether the algorithm will do the job required of it. Being unsure that an algorithm is appropriate is often a fact of life. Being unsure what your code does is just laziness.

Often the best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is obvious how it works.

#### **G22:** Make Logical Dependencies Physical

If one module depends upon another, that dependency should be physical, not just logical. The dependent module should not make assumptions (in other words, logical dependencies) about the module it depends upon. Rather it should explicitly ask that module for all the information it depends upon.

For example, imagine that you are writing a function that prints a plain text report of hours worked by employees. One class named HourlyReporter gathers all the data into a convenient form and then passes it to HourlyReportFormatter to print it. (See [Listing 17-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#ch17lt1).)

**Listing 17-1 HourlyReporter.java**

   public class HourlyReporter {  
     private HourlyReportFormatter formatter;  
     private List<LineItem> page;  
     private final int PAGE\_SIZE = 55;  
  
     public HourlyReporter(HourlyReportFormatter formatter) {  
       this.formatter = formatter;  
       page = new ArrayList<LineItem>();  
     }  
  
     public void generateReport(List<HourlyEmployee> employees) {  
       for (HourlyEmployee e : employees) {  
         addLineItemToPage(e);  
         if (page.size() == PAGE\_SIZE)  
           printAndClearItemList();  
       }  
       if (page.size() > 0)  
         printAndClearItemList();  
     }  
  
     private void printAndClearItemList() {  
       formatter.format(page);  
       page.clear();  
     }  
  
     private void addLineItemToPage(HourlyEmployee e) {  
       LineItem item = new LineItem();  
       item.name = e.getName();  
       item.hours = e.getTenthsWorked() / 10;  
  
       item.tenths = e.getTenthsWorked() % 10;  
       page.add(item);  
     }  
  
     public class LineItem {  
       public String name;  
       public int hours;  
       public int tenths;  
     }  
   }

This code has a logical dependency that has not been physicalized. Can you spot it? It is the constant PAGE\_SIZE. Why should the HourlyReporter know the size of the page? Page size should be the responsibility of the HourlyReportFormatter.

The fact that PAGE\_SIZE is declared in HourlyReporter represents a misplaced responsibility [G17] that causes HourlyReporter to assume that it knows what the page size ought to be. Such an assumption is a logical dependency. HourlyReporter depends on the fact that HourlyReportFormatter can deal with page sizes of 55. If some implementation of HourlyReportFormatter could not deal with such sizes, then there would be an error.

We can physicalize this dependency by creating a new method in HourlyReport-Formatter named getMaxPageSize(). HourlyReporter will then call that function rather than using the PAGE\_SIZE constant.

#### **G23:** Prefer Polymorphism to If/Else or Switch/Case

This might seem a strange suggestion given the topic of [Chapter 6](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter06.html#ch6). After all, in that chapter I make the point that switch statements are probably appropriate in the parts of the system where adding new functions is more likely than adding new types.

First, most people use switch statements because it’s the obvious brute force solution, not because it’s the right solution for the situation. So this heuristic is here to remind us to consider polymorphism before using a switch.

Second, the cases where functions are more volatile than types are relatively rare. So every switch statement should be suspect.

I use the following “ONE SWITCH” rule: There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.

#### **G24:** Follow Standard Conventions

Every team should follow a coding standard based on common industry norms. This coding standard should specify things like where to declare instance variables; how to name classes, methods, and variables; where to put braces; and so on. The team should not need a document to describe these conventions because their code provides the examples.

Everyone on the team should follow these conventions. This means that each team member must be mature enough to realize that it doesn’t matter a whit where you put your braces so long as you all agree on where to put them.

If you would like to know what conventions I follow, you’ll see them in the refactored code in [Listing B-7](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt7) on page [394](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#page_394), through [Listing B-14](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app02.html#app2lt14).

#### **G25:** Replace Magic Numbers with Named Constants

[Image](https://ssl.safaribooksonline.com/9780134661742/CODE_01_02_00)

This is probably one of the oldest rules in software development. I remember reading it in the late sixties in introductory COBOL, FORTRAN, and PL/1 manuals. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.

For example, the number 86,400 should be hidden behind the constant SECONDS\_PER\_DAY. If you are printing 55 lines per page, then the constant 55 should be hidden behind the constant LINES\_PER\_PAGE.

Some constants are so easy to recognize that they don’t always need a named constant to hide behind so long as they are used in conjunction with very self-explanatory code. For example:

   double milesWalked = feetWalked/5280.0;  
   int dailyPay = hourlyRate \* 8;  
   double circumference = radius \* Math.PI \* 2;

Do we really need the constants FEET\_PER\_MILE, WORK\_HOURS\_PER\_DAY, and TWO in the above examples? Clearly, the last case is absurd. There are some formulae in which constants are simply better written as raw numbers. You might quibble about the WORK\_HOURS\_PER\_DAY case because the laws or conventions might change. On the other hand, that formula reads so nicely with the 8 in it that I would be reluctant to add 17 extra characters to the readers’ burden. And in the FEET\_PER\_MILEcase, the number 5280 is so very well known and so unique a constant that readers would recognize it even if it stood alone on a page with no context surrounding it.

Constants like 3.141592653589793 are also very well known and easily recognizable. However, the chance for error is too great to leave them raw. Every time someone sees 3.1415927535890793, they know that it is π, and so they fail to scrutinize it. (Did you catch the single-digit error?) We also don’t want people using 3.14, 3.14159, 3.142, and so forth. Therefore, it is a good thing that Math.PI has already been defined for us.

The term “Magic Number” does not apply only to numbers. It applies to any token that has a value that is not self-describing. For example:

   assertEquals(7777, Employee.find(“John Doe”).employeeNumber());

There are two magic numbers in this assertion. The first is obviously 7777, though what it might mean is not obvious. The second magic number is “John Doe,” and again the intent is not clear.

It turns out that “John Doe” is the name of employee #7777 in a well-known test database created by our team. Everyone in the team knows that when you connect to this database, it will have several employees already cooked into it with well-known values and attributes. It also turns out that “John Doe” represents the sole hourly employee in that test database. So this test should really read:

   assertEquals(  
     HOURLY\_EMPLOYEE\_ID,  
     Employee.find(HOURLY\_EMPLOYEE\_NAME).employeeNumber());

#### **G26:** Be Precise

Expecting the first match to be the only match to a query is probably naive. Using floating point numbers to represent currency is almost criminal. Avoiding locks and/or transaction management because you don’t think concurrent update is likely is lazy at best. Declaring a variable to be an ArrayList when a List will due is overly constraining. Making all variables protected by default is not constraining enough.

When you make a decision in your code, make sure you make it precisely. Know why you have made it and how you will deal with any exceptions. Don’t be lazy about the precision of your decisions. If you decide to call a function that might return null, make sure you check for null. If you query for what you think is the only record in the database, make sure your code checks to be sure there aren’t others. If you need to deal with currency, use integers11 and deal with rounding appropriately. If there is the possibility of concurrent update, make sure you implement some kind of locking mechanism.

11. Or better yet, a Money class that uses integers.

Ambiguities and imprecision in code are either a result of disagreements or laziness. In either case they should be eliminated.

#### **G27:** Structure over Convention

[Image](https://ssl.safaribooksonline.com/9780134661742/CODE_01_03_00)

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance. For example, switch/cases with nicely named enumerations are inferior to base classes with abstract methods. No one is forced to implement the switch/case statement the same way each time; but the base classes do enforce that concrete classes have all abstract methods implemented.

#### **G28:** Encapsulate Conditionals

Boolean logic is hard enough to understand without having to see it in the context of an if or whilestatement. Extract functions that explain the intent of the conditional.

For example:

   if (shouldBeDeleted(timer))

is preferable to

   if (timer.hasExpired() && !timer.isRecurrent())

#### **G29:** Avoid Negative Conditionals

Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives. For example:

   if (buffer.shouldCompact())

is preferable to

   if (!buffer.shouldNotCompact())

#### **G30:** Functions Should Do One Thing

It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than one thing, and should be converted into many smaller functions, each of which does one thing.

For example:

   public void pay() {  
     for (Employee e : employees) {  
       if (e.isPayday()) {  
         Money pay = e.calculatePay();  
         e.deliverPay(pay);  
       }  
     }  
   }

This bit of code does three things. It loops over all the employees, checks to see whether each employee ought to be paid, and then pays the employee. This code would be better written as:

   public void pay() {  
     for (Employee e : employees)  
       payIfNecessary(e);  
   }  
  
   private void payIfNecessary(Employee e) {  
     if (e.isPayday())  
       calculateAndDeliverPay(e);  
   }  
  
   private void calculateAndDeliverPay(Employee e) {  
     Money pay = e.calculatePay();  
     e.deliverPay(pay);  
   }

Each of these functions does one thing. (See “[Do One Thing](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#ch3lev1sec2)” on page [35](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter03.html#page_35).)

#### **G31:** Hidden Temporal Couplings

Temporal couplings are often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvious. Consider the following:

   public class MoogDiver {  
     Gradient gradient;  
     List<Spline> splines;  
  
     public void dive(String reason) {  
       saturateGradient();  
       reticulateSplines();  
       diveForMoog(reason);  
     }  
     …  
   }

The order of the three functions is important. You must saturate the gradient before you can reticulate the splines, and only then can you dive for the moog. Unfortunately, the code does not enforce this temporal coupling. Another programmer could call reticulate-Splines before saturateGradient was called, leading to an UnsaturatedGradientException. A better solution is:

   public class MoogDiver {  
     Gradient gradient;  
     List<Spline> splines;  
  
     public void dive(String reason) {  
       Gradient gradient = saturateGradient();  
       List<Spline> splines = reticulateSplines(gradient);  
       diveForMoog(splines, reason);  
     }  
     …  
   }

This exposes the temporal coupling by creating a bucket brigade. Each function produces a result that the next function needs, so there is no reasonable way to call them out of order.

You might complain that this increases the complexity of the functions, and you’d be right. But that extra syntactic complexity exposes the true temporal complexity of the situation.

Note that I left the instance variables in place. I presume that they are needed by private methods in the class. Even so, I want the arguments in place to make the temporal coupling explicit.

#### **G32:** Don’t Be Arbitrary

Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it. If a structure appears consistently throughout the system, others will use it and preserve the convention. For example, I was recently merging changes to FitNesse and discovered that one of our committers had done this:

   public class AliasLinkWidget extends ParentWidget  
   {  
     public static class VariableExpandingWidgetRoot {  
       …  
  
     …  
   }

The problem with this was that VariableExpandingWidgetRoot had no need to be inside the scope of AliasLinkWidget. Moreover, other unrelated classes made use of AliasLinkWidget.VariableExpandingWidgetRoot. These classes had no need to know about AliasLinkWidget.

Perhaps the programmer had plopped the VariableExpandingWidgetRoot into AliasWidget as a matter of convenience, or perhaps he thought it really needed to be scoped inside AliasWidget. Whatever the reason, the result wound up being arbitrary. Public classes that are not utilities of some other class should not be scoped inside another class. The convention is to make them public at the top level of their package.

#### **G33:** Encapsulate Boundary Conditions

Boundary conditions are hard to keep track of. Put the processing for them in one place. Don’t let them leak all over the code. We don’t want swarms of +1s and -1s scattered hither and yon. Consider this simple example from FIT:

   if(level + 1 < tags.length)  
   {  
     parts = new Parse(body, tags, level + 1, offset + endTag);  
     body = null;  
   }

Notice that level+1 appears twice. This is a boundary condition that should be encapsulated within a variable named something like nextLevel.

   int nextLevel = level + 1;  
   if(nextLevel < tags.length)  
   {  
     parts = new Parse(body, tags, nextLevel, offset + endTag);  
     body = null;  
   }

#### **G34:** Functions Should Descend Only One Level of Abstraction

The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function. This may be the hardest of these heuristics to interpret and follow. Though the idea is plain enough, humans are just far too good at seamlessly mixing levels of abstraction. Consider, for example, the following code taken from FitNesse:

   public String render() throws Exception  
   {  
     StringBuffer html = new StringBuffer(“<hr”);  
     if(size > 0)  
       html.append(” size=\“”).append(size + 1).append(”\“”);  
     html.append(“>”);  
  
     return html.toString();  
   }

A moment’s study and you can see what’s going on. This function constructs the HTML tag that draws a horizontal rule across the page. The height of that rule is specified in the size variable.

Now look again. This method is mixing at least two levels of abstraction. The first is the notion that a horizontal rule has a size. The second is the syntax of the HR tag itself. This code comes from the HruleWidget module in FitNesse. This module detects a row of four or more dashes and converts it into the appropriate HR tag. The more dashes, the larger the size.

I refactored this bit of code as follows. Note that I changed the name of the size field to reflect its true purpose. It held the number of extra dashes.

   public String render() throws Exception  
   {  
     HtmlTag hr = new HtmlTag(“hr”);  
     if (extraDashes > 0)  
       hr.addAttribute(“size”, hrSize(extraDashes));  
     return hr.html();  
   }  
  
   private String hrSize(int height)  
   {  
     int hrSize = height + 1;  
     return String.format(“%d”, hrSize);  
   }

This change separates the two levels of abstraction nicely. The render function simply constructs an HR tag, without having to know anything about the HTML syntax of that tag. The HtmlTag module takes care of all the nasty syntax issues.

Indeed, by making this change I caught a subtle error. The original code did not put the closing slash on the HR tag, as the XHTML standard would have it. (In other words, it emitted <hr> instead of <hr/>.) The HtmlTag module had been changed to conform to XHTML long ago.

Separating levels of abstraction is one of the most important functions of refactoring, and it’s one of the hardest to do well. As an example, look at the code below. This was my first attempt at separating the abstraction levels in the HruleWidget.render method.

   public String render() throws Exception  
   {  
     HtmlTag hr = new HtmlTag(“hr”);  
     if (size > 0) {  
       hr.addAttribute(“size”, “”+(size+1));  
     }  
     return hr.html();  
   }

My goal, at this point, was to create the necessary separation and get the tests to pass. I accomplished that goal easily, but the result was a function that still had mixed levels of abstraction. In this case the mixed levels were the construction of the HR tag and the interpretation and formatting of the sizevariable. This points out that when you break a function along lines of abstraction, you often uncover new lines of abstraction that were obscured by the previous structure.

#### **G35:** Keep Configurable Data at High Levels

If you have a constant such as a default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to that low-level function called from the high-level function. Consider the following code from FitNesse:

     public static void main(String[] args) throws Exception  
     {  
       Arguments arguments = parseCommandLine(args);  
       …  
     }  
  
   public class Arguments  
   {  
     public static final String DEFAULT\_PATH = “.”;  
     public static final String DEFAULT\_ROOT = “FitNesseRoot”;  
     public static final int DEFAULT\_PORT = 80;  
     public static final int DEFAULT\_VERSION\_DAYS = 14;  
     …  
   }

The command-line arguments are parsed in the very first executable line of FitNesse. The default values of those arguments are specified at the top of the Argument class. You don’t have to go looking in low levels of the system for statements like this one:

   if (arguments.port == 0) // use 80 by default

The configuration constants reside at a very high level and are easy to change. They get passed down to the rest of the application. The lower levels of the application do not own the values of these constants.

#### **G36:** Avoid Transitive Navigation

In general we don’t want a single module to know much about its collaborators. More specifically, if Acollaborates with B, and B collaborates with C, we don’t want modules that use A to know about C. (For example, we don’t want a.getB().getC().doSomething();.)

This is sometimes called the Law of Demeter. The Pragmatic Programmers call it “Writing Shy Code.”12In either case it comes down to making sure that modules know only about their immediate collaborators and do not know the navigation map of the whole system.

12. [[PRAG](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#PRAG)], p. [138](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter10.html#page_138).

If many modules used some form of the statement a.getB().getC(), then it would be difficult to change the design and architecture to interpose a Q between B and C. You’d have to find every instance of a.getB().getC() and convert it to a.getB().getQ().getC(). This is how architectures become rigid. Too many modules know too much about the architecture.

Rather we want our immediate collaborators to offer all the services we need. We should not have to roam through the object graph of the system, hunting for the method we want to call. Rather we should simply be able to say:

   myCollaborator.doSomething().

### **JAVA**

#### **J1:** Avoid Long Import Lists by Using Wildcards

If you use two or more classes from a package, then import the whole package with

   import package.\*;

Long lists of imports are daunting to the reader. We don’t want to clutter up the tops of our modules with 80 lines of imports. Rather we want the imports to be a concise statement about which packages we collaborate with.

Specific imports are hard dependencies, whereas wildcard imports are not. If you specifically import a class, then that class must exist. But if you import a package with a wildcard, no particular classes need to exist. The import statement simply adds the package to the search path when hunting for names. So no true dependency is created by such imports, and they therefore serve to keep our modules less coupled.

There are times when the long list of specific imports can be useful. For example, if you are dealing with legacy code and you want to find out what classes you need to build mocks and stubs for, you can walk down the list of specific imports to find out the true qualified names of all those classes and then put the appropriate stubs in place. However, this use for specific imports is very rare. Furthermore, most modern IDEs will allow you to convert the wildcarded imports to a list of specific imports with a single command. So even in the legacy case it’s better to import wildcards.

Wildcard imports can sometimes cause name conflicts and ambiguities. Two classes with the same name, but in different packages, will need to be specifically imported, or at least specifically qualified when used. This can be a nuisance but is rare enough that using wildcard imports is still generally better than specific imports.

#### **J2:** Don’t Inherit Constants

I have seen this several times and it always makes me grimace. A programmer puts some constants in an interface and then gains access to those constants by inheriting that interface. Take a look at the following code:

   public class HourlyEmployee extends Employee {  
     private int tenthsWorked;  
     private double hourlyRate;  
  
     public Money calculatePay() {  
       int straightTime = Math.min(tenthsWorked, TENTHS\_PER\_WEEK);  
       int overTime = tenthsWorked - straightTime;  
       return new Money(  
         hourlyRate \* (tenthsWorked + OVERTIME\_RATE \* overTime)  
       );  
     }  
     …  
   }

Where did the constants TENTHS\_PER\_WEEK and OVERTIME\_RATE come from? They might have come from class Employee; so let’s take a look at that:

   public abstract class Employee implements PayrollConstants {  
     public abstract boolean isPayday();  
     public abstract Money calculatePay();  
     public abstract void deliverPay(Money pay);  
   }

Nope, not there. But then where? Look closely at class Employee. It implements PayrollConstants.

   public interface PayrollConstants {  
     public static final int TENTHS\_PER\_WEEK = 400;  
     public static final double OVERTIME\_RATE = 1.5;  
   }

This is a hideous practice! The constants are hidden at the top of the inheritance hierarchy. Ick! Don’t use inheritance as a way to cheat the scoping rules of the language. Use a static import instead.

**import static PayrollConstants.\*;**  
  
   public class HourlyEmployee extends Employee {  
     private int tenthsWorked;  
     private double hourlyRate;  
  
     public Money calculatePay() {  
       int straightTime = Math.min(tenthsWorked, TENTHS\_PER\_WEEK);  
       int overTime = tenthsWorked - straightTime;  
       return new Money(  
         hourlyRate \* (tenthsWorked + OVERTIME\_RATE \* overTime)  
       );  
     }  
     …  
   }

#### **J3:** Constants versus Enums

Now that enums have been added to the language (Java 5), use them! Don’t keep using the old trick of public static final ints. The meaning of ints can get lost. The meaning of enums cannot, because they belong to an enumeration that is named.

What’s more, study the syntax for enums carefully. They can have methods and fields. This makes them very powerful tools that allow much more expression and flexibility than ints. Consider this variation on the payroll code:

   public class HourlyEmployee extends Employee {  
     private int tenthsWorked;  
     **HourlyPayGrade grade;**  
  
     public Money calculatePay() {  
       int straightTime = Math.min(tenthsWorked, TENTHS\_PER\_WEEK);  
       int overTime = tenthsWorked - straightTime;  
       return new Money(  
         **grade.rate()** \* (tenthsWorked + OVERTIME\_RATE \* overTime)  
       );  
     }  
     …  
   }  
   public enum HourlyPayGrade {  
     APPRENTICE {  
       public double rate() {  
         return 1.0;  
       }  
     },  
     LEUTENANT\_JOURNEYMAN {  
       public double rate() {  
         return 1.2;  
       }  
     },  
     JOURNEYMAN {  
       public double rate() {  
         return 1.5;  
       }  
     },  
     MASTER {  
       public double rate() {  
         return 2.0;  
       }  
       };  
  
       public abstract double rate();  
   }

### **NAMES**

#### **N1:** Choose Descriptive Names

Don’t be too quick to choose a name. Make sure the name is descriptive. Remember that meanings tend to drift as software evolves, so frequently reevaluate the appropriateness of the names you choose.

This is not just a “feel-good” recommendation. Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant. Names are too important to treat carelessly.

Consider the code below. What does it do? If I show you the code with well-chosen names, it will make perfect sense to you, but like this it’s just a hodge-podge of symbols and magic numbers.

   public int x() {  
       int q = 0;  
       int z = 0;  
       for (int kk = 0; kk < 10; kk++) {  
         if (l[z] == 10)  
         {  
           q += 10 + (l[z + 1] + l[z + 2]);  
           z += 1;  
         }  
         else if (l[z] + l[z + 1] == 10)  
         {  
           q += 10 + l[z + 2];  
           z += 2;  
         } else {  
           q += l[z] + l[z + 1];  
           z += 2;  
         }  
       }  
       return q;  
     }

Here is the code the way it should be written. This snippet is actually less complete than the one above. Yet you can infer immediately what it is trying to do, and you could very likely write the missing functions based on that inferred meaning. The magic numbers are no longer magic, and the structure of the algorithm is compellingly descriptive.

   public int score() {  
     int score = 0;  
     int frame = 0;  
     for (int frameNumber = 0; frameNumber < 10; frameNumber++) {  
       if (isStrike(frame)) {  
         score += 10 + nextTwoBallsForStrike(frame);  
         frame += 1;  
       } else if (isSpare(frame)) {  
         score += 10 + nextBallForSpare(frame);  
         frame += 2;  
       } else {  
         score += twoBallsInFrame(frame);  
         frame += 2;  
       }  
     }  
     return score;  
   }

The power of carefully chosen names is that they overload the structure of the code with description. That overloading sets the readers’ expectations about what the other functions in the module do. You can infer the implementation of isStrike() by looking at the code above. When you read the isStrike method, it will be “pretty much what you expected.”13

13. See Ward Cunningham’s quote on page [11](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter01.html#page_11).

   private boolean isStrike(int frame) {  
     return rolls[frame] == 10;  
   }

#### **N2:** Choose Names at the Appropriate Level of Abstraction

Don’t pick names that communicate implementation; choose names the reflect the level of abstraction of the class or function you are working in. This is hard to do. Again, people are just too good at mixing levels of abstractions. Each time you make a pass over your code, you will likely find some variable that is named at too low a level. You should take the opportunity to change those names when you find them. Making code readable requires a dedication to continuous improvement. Consider the Modeminterface below:

   public interface Modem {  
     boolean dial(String phoneNumber);  
     boolean disconnect();  
     boolean send(char c);  
     char recv();  
     String getConnectedPhoneNumber();  
   }

At first this looks fine. The functions all seem appropriate. Indeed, for many applications they are. But now consider an application in which some modems aren’t connected by dialling. Rather they are connected permanently by hard wiring them together (think of the cable modems that provide Internet access to most homes nowadays). Perhaps some are connected by sending a port number to a switch over a USB connection. Clearly the notion of phone numbers is at the wrong level of abstraction. A better naming strategy for this scenario might be:

   public interface Modem {  
     boolean connect(String connectionLocator);  
     boolean disconnect();  
     boolean send(char c);  
     char recv();  
     String getConnectedLocator();  
   }

Now the names don’t make any commitments about phone numbers. They can still be used for phone numbers, or they could be used for any other kind of connection strategy.

#### **N3:** Use Standard Nomenclature Where Possible

Names are easier to understand if they are based on existing convention or usage. For example, if you are using the DECORATOR pattern, you should use the word Decorator in the names of the decorating classes. For example, AutoHangupModemDecorator might be the name of a class that decorates a Modem with the ability to automatically hang up at the end of a session.

Patterns are just one kind of standard. In Java, for example, functions that convert objects to string representations are often named toString. It is better to follow conventions like these than to invent your own.

Teams will often invent their own standard system of names for a particular project. Eric Evans refers to this as a ubiquitous language for the project.14 Your code should use the terms from this language extensively. In short, the more you can use names that are overloaded with special meanings that are relevant to your project, the easier it will be for readers to know what your code is talking about.

14. [[DDD](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter17.html#DDD)].

#### **N4:** Unambiguous Names

Choose names that make the workings of a function or variable unambiguous. Consider this example from FitNesse:

   private String doRename() throws Exception  
   {  
     if(refactorReferences)  
       renameReferences();  
     renamePage();  
  
     pathToRename.removeNameFromEnd();  
     pathToRename.addNameToEnd(newName);  
     return PathParser.render(pathToRename);  
   }

The name of this function does not say what the function does except in broad and vague terms. This is emphasized by the fact that there is a function named renamePage inside the function named doRename! What do the names tell you about the difference between the two functions? Nothing.

A better name for that function is renamePageAndOptionallyAllReferences. This may seem long, and it is, but it’s only called from one place in the module, so it’s explanatory value outweighs the length.

#### **N5:** Use Long Names for Long Scopes

The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.

Variable names like i and j are just fine if their scope is five lines long. Consider this snippet from the old standard “Bowling Game”:

   private void rollMany(int n, int pins)  
   {  
     for (int i=0; i<n; i++)  
       g.roll(pins);  
   }

This is perfectly clear and would be obfuscated if the variable i were replaced with something annoying like rollCount. On the other hand, variables and functions with short names lose their meaning over long distances. So the longer the scope of the name, the longer and more precise the name should be.

#### **N6:** Avoid Encodings

Names should not be encoded with type or scope information. Prefixes such as m\_ or f are useless in today’s environments. Also project and/or subsystem encodings such as vis\_ (for visual imaging system) are distracting and redundant. Again, today’s environments provide all that information without having to mangle the names. Keep your names free of Hungarian pollution.

#### **N7:** Names Should Describe Side-Effects

Names should describe everything that a function, variable, or class is or does. Don’t hide side effects with a name. Don’t use a simple verb to describe a function that does more than just that simple action. For example, consider this code from TestNG:

   public ObjectOutputStream getOos() throws IOException {  
     if (m\_oos == null) {  
       m\_oos = new ObjectOutputStream(m\_socket.getOutputStream());  
     }  
     return m\_oos;  
   }

This function does a bit more than get an “oos”; it creates the “oos” if it hasn’t been created already. Thus, a better name might be createOrReturnOos.

### **TESTS**

#### **T1:** Insufficient Tests

How many tests should be in a test suite? Unfortunately, the metric many programmers use is “That seems like enough.” A test suite should test everything that could possibly break. The tests are insufficient so long as there are conditions that have not been explored by the tests or calculations that have not been validated.

#### **T2:** Use a Coverage Tool!

Coverage tools reports gaps in your testing strategy. They make it easy to find modules, classes, and functions that are insufficiently tested. Most IDEs give you a visual indication, marking lines that are covered in green and those that are uncovered in red. This makes it quick and easy to find if or catchstatements whose bodies haven’t been checked.

#### **T3:** Don’t Skip Trivial Tests

They are easy to write and their documentary value is higher than the cost to produce them.

#### **T4:** An Ignored Test Is a Question about an Ambiguity

Sometimes we are uncertain about a behavioral detail because the requirements are unclear. We can express our question about the requirements as a test that is commented out, or as a test that annotated with @Ignore. Which you choose depends upon whether the ambiguity is about something that would compile or not.

#### **T5:** Test Boundary Conditions

Take special care to test boundary conditions. We often get the middle of an algorithm right but misjudge the boundaries.

#### **T6:** Exhaustively Test Near Bugs

Bugs tend to congregate. When you find a bug in a function, it is wise to do an exhaustive test of that function. You’ll probably find that the bug was not alone.

#### **T7:** Patterns of Failure Are Revealing

Sometimes you can diagnose a problem by finding patterns in the way the test cases fail. This is another argument for making the test cases as complete as possible. Complete test cases, ordered in a reasonable way, expose patterns.

As a simple example, suppose you noticed that all tests with an input larger than five characters failed? Or what if any test that passed a negative number into the second argument of a function failed? Sometimes just seeing the pattern of red and green on the test report is enough to spark the “Aha!” that leads to the solution. Look back at page [267](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter16.html#page_267) to see an interesting example of this in the SerialDateexample.

#### **T8:** Test Coverage Patterns Can Be Revealing

Looking at the code that is or is not executed by the passing tests gives clues to why the failing tests fail.

#### **T9:** Tests Should Be Fast

A slow test is a test that won’t get run. When things get tight, it’s the slow tests that will be dropped from the suite. So do what you must to keep your tests fast.

### **CONCLUSION**

This list of heuristics and smells could hardly be said to be complete. Indeed, I’m not sure that such a list can ever be complete. But perhaps completeness should not be the goal, because what this list does do is imply a value system.

Indeed, that value system has been the goal, and the topic, of this book. Clean code is not written by following a set of rules. You don’t become a software craftsman by learning a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines.

## ****Appendix A**** ****Concurrency II****

This appendix supports and amplifies the Concurrency chapter on page [177](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter13.html#page_177). It is written as a series of independent topics and you can generally read them in any order. There is some duplication between sections to allow for such reading.

### **CLIENT/SERVER EXAMPLE**

Imagine a simple client/server application. A server sits and waits listening on a socket for a client to connect. A client connects and sends a request.

#### **The Server**

Here is a simplified version of a server application. Full source for this example is available starting on page [343](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_343), Client/Server Nonthreaded.

   ServerSocket serverSocket = new ServerSocket(8009);  
  
   while (keepProcessing) {  
       try {  
           Socket socket = serverSocket.accept();  
           process(socket);  
       } catch (Exception e) {  
           handle(e);  
       }  
   }

This simple application waits for a connection, processes an incoming message, and then again waits for the next client request to come in. Here’s client code that connects to this server:

   private void connectSendReceive(int i) {  
       try {  
           Socket socket = new Socket(“localhost”, PORT);  
           MessageUtils.sendMessage(socket, Integer.toString(i));  
           MessageUtils.getMessage(socket);  
           socket.close();  
       } catch (Exception e) {  
           e.printStackTrace();  
       }  
   }

How well does this client/server pair perform? How can we formally describe that performance? Here’s a test that asserts that the performance is “acceptable”:

   @Test(timeout = 10000)  
   public void shouldRunInUnder10Seconds() throws Exception {  
        Thread[] threads = createThreads();  
        startAllThreadsw(threads);  
        waitForAllThreadsToFinish(threads);  
   }

The setup is left out to keep the example simple (see “[ClientTest.java](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lt4)” on page [344](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_344)). This test asserts that it should complete within 10,000 milliseconds.

This is a classic example of validating the throughput of a system. This system should complete a series of client requests in ten seconds. So long as the server can process each individual client request in time, the test will pass.

What happens if the test fails? Short of developing some kind of event polling loop, there is not much to do within a single thread that will make this code any faster. Will using multiple threads solve the problem? It might, but we need to know where the time is being spent. There are two possibilities:

• I/O—using a socket, connecting to a database, waiting for virtual memory swapping, and so on.

• Processor—numerical calculations, regular expression processing, garbage collection, and so on.

Systems typically have some of each, but for a given operation one tends to dominate. If the code is processor bound, more processing hardware can improve throughput, making our test pass. But there are only so many CPU cycles available, so adding threads to a processor-bound problem will not make it go faster.

On the other hand, if the process is I/O bound, then concurrency can increase efficiency. When one part of the system is waiting for I/O, another part can use that wait time to process something else, making more effective use of the available CPU.

#### **Adding Threading**

Assume for the moment that the performance test fails. How can we improve the throughput so that the performance test passes? If the process method of the server is I/O bound, then here is one way to make the server use threads (just change the processMessage):

   void process(final Socket socket) {  
       if (socket == null)  
           return;  
  
       Runnable clientHandler = new Runnable() {  
           public void run() {  
               try {  
                   String message = MessageUtils.getMessage(socket);  
                   MessageUtils.sendMessage(socket, “Processed: ” + message);  
                   closeIgnoringException(socket);  
               } catch (Exception e) {  
                   e.printStackTrace();  
               }  
           }  
       };  
  
       Thread clientConnection = new Thread(clientHandler);  
       clientConnection.start();  
   }

Assume that this change causes the test to pass;1 the code is complete, correct?

1. You can verify that for yourself by trying out the before and after code. Review the nonthreaded code starting on page [343](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_343). Review the threaded code starting on page [346](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_346).

#### **Server Observations**

The updated server completes the test successfully in just over one second. Unfortunately, this solution is a bit naive and introduces some new problems.

How many threads might our server create? The code sets no limit, so the we could feasibly hit the limit imposed by the Java Virtual Machine (JVM). For many simple systems this may suffice. But what if the system is meant to support many users on the public net? If too many users connect at the same time, the system might grind to a halt.

But set the behavioral problem aside for the moment. The solution shown has problems of cleanliness and structure. How many responsibilities does the server code have?

• Socket connection management

• Client processing

• Threading policy

• Server shutdown policy

Unfortunately, all these responsibilities live in the process function. In addition, the code crosses many different levels of abstraction. So, small as the process function is, it needs to be repartitioned.

The server has several reasons to change; therefore it violates the Single Responsibility Principle. To keep concurrent systems clean, thread management should be kept to a few, well-controlled places. What’s more, any code that manages threads should do nothing other than thread management. Why? If for no other reason than that tracking down concurrency issues is hard enough without having to unwind other nonconcurrency issues at the same time.

If we create a separate class for each of the responsibilities listed above, including the thread management responsibility, then when we change the thread management strategy, the change will impact less overall code and will not pollute the other responsibilities. This also makes it much easier to test all the other responsibilities without having to worry about threading. Here is an updated version that does just that:

   public void run() {  
     while (keepProcessing) {  
      try {  
       ClientConnection clientConnection = connectionManager.awaitClient();  
       ClientRequestProcessor requestProcessor  
         = new ClientRequestProcessor(clientConnection);  
       clientScheduler.schedule(requestProcessor);  
       } catch (Exception e) {  
         e.printStackTrace();  
       }  
     }  
     connectionManager.shutdown();  
   }

This now focuses all things thread-related into one place, clientScheduler. If there are concurrency problems, there is just one place to look:

   public interface ClientScheduler {  
       void schedule(ClientRequestProcessor requestProcessor);  
   }

The current policy is easy to implement:

   public class ThreadPerRequestScheduler implements ClientScheduler {  
       public void schedule(final ClientRequestProcessor requestProcessor) {  
           Runnable runnable = new Runnable() {  
               public void run() {  
                   requestProcessor.process();  
               }  
           };  
  
          Thread thread = new Thread(runnable);  
          thread.start();  
       }  
   }

Having isolated all the thread management into a single place, it is much easier to change the way we control threads. For example, moving to the Java 5 Executor framework involves writing a new class and plugging it in ([Listing A-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lt1)).

**Listing A-1 ExecutorClientScheduler.java**

   import java.util.concurrent.Executor;  
   import java.util.concurrent.Executors;  
  
   public class ExecutorClientScheduler implements ClientScheduler {  
       Executor executor;  
  
       public ExecutorClientScheduler(int availableThreads) {  
           executor = Executors.newFixedThreadPool(availableThreads);  
        }  
  
       public void schedule(final ClientRequestProcessor requestProcessor) {  
           Runnable runnable = new Runnable() {  
               public void run() {  
                   requestProcessor.process();  
               }  
           };  
           executor.execute(runnable);  
        }  
   }

#### **Conclusion**

Introducing concurrency in this particular example demonstrates a way to improve the throughput of a system and one way of validating that throughput through a testing framework. Focusing all concurrency code into a small number of classes is an example of applying the Single Responsibility Principle. In the case of concurrent programming, this becomes especially important because of its complexity.

### **POSSIBLE PATHS OF EXECUTION**

Review the method incrementValue, a one-line Java method with no looping or branching:

   public class IdGenerator {  
     int lastIdUsed;  
  
     public int incrementValue() {  
       return ++lastIdUsed;  
     }  
   }

Ignore integer overflow and assume that only one thread has access to a single instance of IdGenerator. In this case there is a single path of execution and a single guaranteed result:

• The value returned is equal to the value of lastIdUsed, both of which are one greater than just before calling the method.

What happens if we use two threads and leave the method unchanged? What are the possible outcomes if each thread calls incrementValue once? How many possible paths of execution are there? First, the outcomes (assume lastIdUsed starts with a value of 93):

• Thread 1 gets the value of 94, thread 2 gets the value of 95, and lastIdUsed is now 95.

• Thread 1 gets the value of 95, thread 2 gets the value of 94, and lastIdUsed is now 95.

• Thread 1 gets the value of 94, thread 2 gets the value of 94, and lastIdUsed is now 94.

The final result, while surprising, is possible. To see how these different results are possible, we need to understand the number of possible paths of execution and how the Java Virtual Machine executes them.

#### **Number of Paths**

To calculate the number of possible execution paths, we’ll start with the generated byte-code. The one line of java (return ++lastIdUsed;) becomes eight byte-code instructions. It is possible for the two threads to interleave the execution of these eight instructions the way a card dealer interleaves cards as he shuffles a deck.2 Even with only eight cards in each hand, there are a remarkable number of shuffled outcomes.

2. This is a bit of a simplification. However, for the purpose of this discussion, we can use this simplifying model.

For this simple case of N instructions in a sequence, no looping or conditionals, and T threads, the total number of possible execution paths is equal to

Image

**Calculating the Possible Orderings**

This comes from an email from Uncle Bob to Brett:

With N steps and T threads there are T\* N total steps. Prior to each step there is a context switch that chooses between the T threads. Each path can thus be represented as a string of digits denoting the context switches. Given steps A and B and threads 1 and 2, the six possible paths are 1122, 1212, 1221, 2112, 2121, and 2211. Or, in terms of steps it is A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1, and A2B2A1B1. For three threads the sequence is 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, ….

One characteristic of these strings is that there must always be N instances of each T. So the string 111111 is invalid because it has six instances of 1 and zero instances of 2 and 3.

So we want the permutations of N 1’s, N 2’s, … and N T’s. This is really just the permutations of N\* T things taken N\* T at a time, which is (N\* T)!, but with all the duplicates removed. So the trick is to count the duplicates and subtract that from (N\* T)!.

Given two steps and two threads, how many duplicates are there? Each four-digit string has two 1s and two 2s. Each of those pairs could be swapped without changing the sense of the string. You could swap the 1s or the 2s both, or neither. So there are four isomorphs for each string, which means that there are three duplicates. So three out of four of the options are duplicates; alternatively one of four of the permutations are NOT duplicates. 4! \* .25 = 6. So this reasoning seems to work.

How many duplicates are there? In the case where N = 2 and T = 2, I could swap the 1s, the 2s, or both. In the case where N = 2 and T = 3, I could swap the 1s, the 2s, the 3s, 1s and 2s, 1s and 3s, or 2s and 3s. Swapping is just the permutations of N. Let’s say there are P permutations of N. The number of different ways to arrange those permutations are P\*\*T.

So the number of possible isomorphs is N!\*\*T. And so the number of paths is (T\*N)!/(N!\*\*T). Again, in our T = 2, N = 2 case we get 6 (24/4).

For N = 2 and T = 3 we get 720/8 = 90.

For N = 3 and T = 3 we get 9!/6^3 = 1680.

For our simple case of one line of Java code, which equates to eight lines of byte-code and two threads, the total number of possible paths of execution is 12,870. If the type of lastIdUsed is a long, then every read/write becomes two operations instead of one, and the number of possible orderings becomes 2,704,156.

What happens if we make one change to this method?

   public **synchronized** void incrementValue() {  
       ++lastIdUsed;  
   }

The number of possible execution pathways becomes two for two threads and N! in the general case.

#### **Digging Deeper**

What about the surprising result that two threads could both call the method once (before we added synchronized) and get the same numeric result? How is that possible? First things first.

What is an atomic operation? We can define an atomic operation as any operation that is uninterruptable. For example, in the following code, line 5, where 0 is assigned to lastid, is atomic because according to the Java Memory model, assignment to a 32-bit value is uninterruptable.

   01: public class Example {  
   02:    int lastId;  
   03:  
   04:    public void resetId() {  
   05:        value = 0;  
   06:    }  
   07:  
   08:    public int getNextId() {  
   09:        ++value;  
   10:    }  
   11:}

What happens if we change type of lastId from int to long? Is line 5 still atomic? Not according to the JVM specification. It could be atomic on a particular processor, but according to the JVM specification, assignment to any 64-bit value requires two 32-bit assignments. This means that between the first 32-bit assignment and the second 32-bit assignment, some other thread could sneak in and change one of the values.

What about the pre-increment operator, ++, on line 9? The pre-increment operator can be interrupted, so it is not atomic. To understand, let’s review the byte-code of both of these methods in detail.

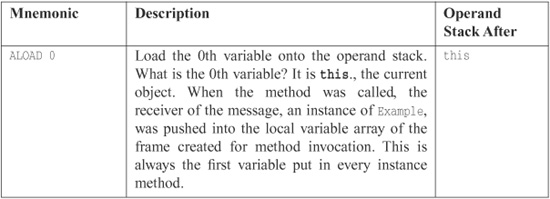
Before we go any further, here are three definitions that will be important:

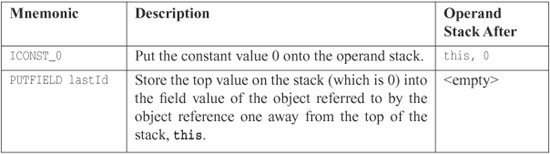
• Frame—Every method invocation requires a frame. The frame includes the return address, any parameters passed into the method and the local variables defined in the method. This is a standard technique used to define a call stack, which is used by modern languages to allow for basic function/method invocation and to allow for recursive invocation.

• Local variable—Any variables defined in the scope of the method. All nonstatic methods have at least one variable, **this**, which represents the current object, the object that received the most recent message (in the current thread), which caused the method invocation.

• Operand stack—Many of the instructions in the Java Virtual Machine take parameters. The operand stack is where those parameters are put. The stack is a standard last-in, first-out (LIFO) data structure.

Here is the byte-code generated for resetId():

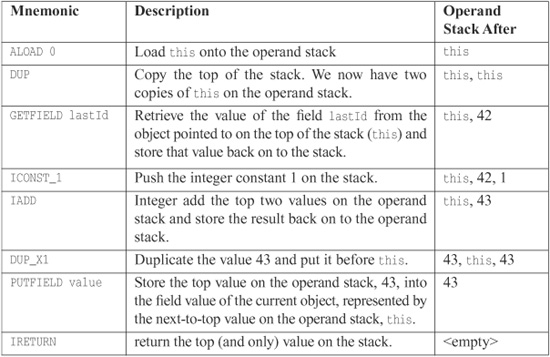




These three instructions are guaranteed to be atomic because, although the thread executing them could be interrupted after any one of them, the information for the PUTFIELD instruction (the constant value 0 on the top of the stack and the reference to this one below the top, along with the field value) cannot be touched by another thread. So when the assignment occurs, we are guaranteed that the value 0 will be stored in the field value. The operation is atomic. The operands all deal with information local to the method, so there is no interference between multiple threads.

So if these three instructions are executed by ten threads, there are 4.38679733629e+24 possible orderings. However, there is only one possible outcome, so the different orderings are irrelevant. It just so happens that the same outcome is guaranteed for longs in this case as well. Why? All ten threads are assigning a constant value. Even if they interleave with each other, the end result is the same.

With the ++ operation in the getNextId method, there are going to be problems. Assume that lastId holds 42 at the beginning of this method. Here is the byte-code for this new method:



Imagine the case where the first thread completes the first three instructions, up to and including GETFIELD, and then it is interrupted. A second thread takes over and performs the entire method, incrementing lastId by one; it gets 43 back. Then the first thread picks up where it left off; 42 is still on the operand stack because that was the value of lastId when it executed GETFIELD. It adds one to get 43 again and stores the result. The value 43 is returned to the first thread as well. The result is that one of the increments is lost because the first thread stepped on the second thread after the second thread interrupted the first thread.

Making the getNexId() method synchronized fixes this problem.

#### **Conclusion**

An intimate understanding of byte-code is not necessary to understand how threads can step on each other. If you can understand this one example, it should demonstrate the possibility of multiple threads stepping on each other, which is enough knowledge.

That being said, what this trivial example demonstrates is a need to understand the memory model enough to know what is and is not safe. It is a common misconception that the ++ (pre- or post-increment) operator is atomic, and it clearly is not. This means you need to know:

• Where there are shared objects/values

• The code that can cause concurrent read/update issues

• How to guard such concurrent issues from happening

### **KNOWING YOUR LIBRARY**

#### **Executor Framework**

As demonstrated in the ExecutorClientScheduler.java on page [321](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#page_321), the Executor framework introduced in Java 5 allows for sophisticated execution using thread pools. This is a class in the java.util.concurrent package.

If you are creating threads and are not using a thread pool or are using a hand-written one, you should consider using the Executor. It will make your code cleaner, easier to follow, and smaller.

The Executor framework will pool threads, resize automatically, and recreate threads if necessary. It also supports futures, a common concurrent programming construct. The Executor framework works with classes that implement Runnable and also works with classes that implement the Callableinterface. A Callable looks like a Runnable, but it can return a result, which is a common need in multithreaded solutions.

A future is handy when code needs to execute multiple, independent operations and wait for both to finish:

   public String processRequest(String message) throws Exception {  
       Callable<String> makeExternalCall = new Callable<String>() {  
  
           public String call() throws Exception {  
               String result = “”;  
               // make external request  
               return result;  
           }  
       };  
  
       Future<String> result = executorService.submit(makeExternalCall);  
       String partialResult = doSomeLocalProcessing();  
       return result.get() + partialResult;  
   }

In this example, the method starts executing the makeExternalCall object. The method continues other processing. The final line calls result.get(), which blocks until the future completes.

#### **Nonblocking Solutions**

The Java 5 VM takes advantage of modern processor design, which supports reliable, nonblocking updates. Consider, for example, a class that uses synchronization (and therefore blocking) to provide a thread-safe update of a value:

   public class ObjectWithValue {  
       private int value;  
       public void synchronized incrementValue() { ++value; }  
       public int getValue() { return value; }  
   }

Java 5 has a series of new classes for situations like this: AtomicBoolean, AtomicInteger, and AtomicReference are three examples; there are several more. We can rewrite the above code to use a nonblocking approach as follows:

   public class ObjectWithValue {  
       private AtomicInteger value = new AtomicInteger(0);  
  
       public void incrementValue() {  
       value.incrementAndGet();  
       }  
       public int getValue() {  
           return value.get();  
       }  
   }

Even though this uses an object instead of a primitive and sends messages like incrementAndGet()instead of ++, the performance of this class will nearly always beat the previous version. In some cases it will only be slightly faster, but the cases where it will be slower are virtually nonexistent.

How is this possible? Modern processors have an operation typically called Compare and Swap (CAS). This operation is analogous to optimistic locking in databases, whereas the synchronized version is analogous to pessimistic locking.

The synchronized keyword always acquires a lock, even when a second thread is not trying to update the same value. Even though the performance of intrinsic locks has improved from version to version, they are still costly.

The nonblocking version starts with the assumption that multiple threads generally do not modify the same value often enough that a problem will arise. Instead, it efficiently detects whether such a situation has occurred and retries until the update happens successfully. This detection is almost always less costly than acquiring a lock, even in moderate to high contention situations.

How does the Virtual Machine accomplish this? The CAS operation is atomic. Logically, the CAS operation looks something like the following:

   int variableBeingSet;  
  
   void simulateNonBlockingSet(int newValue) {  
       int currentValue;  
       do {  
          currentValue = variableBeingSet  
       } while(currentValue != compareAndSwap(currentValue, newValue));  
   }  
  
   int synchronized compareAndSwap(int currentValue, int newValue) {  
       if(variableBeingSet == currentValue) {  
           variableBeingSet = newValue;  
           return currentValue;  
       }  
       return variableBeingSet;   
   }

When a method attempts to update a shared variable, the CAS operation verifies that the variable getting set still has the last known value. If so, then the variable is changed. If not, then the variable is not set because another thread managed to get in the way. The method making the attempt (using the CAS operation) sees that the change was not made and retries.

#### **Nonthread-Safe Classes**

There are some classes that are inherently not thread safe. Here are a few examples:

• SimpleDateFormat

• Database Connections

• Containers in java.util

• Servlets

Note that some collection classes have individual methods that are thread-safe. However, any operation that involves calling more than one method is not. For example, if you do not want to replace something in a HashTable because it is already there, you might write the following code:

   if(!hashTable.containsKey(someKey)) {  
       hashTable.put(someKey, new SomeValue());  
   }

Each individual method is thread-safe. However, another thread might add a value in between the containsKey and put calls. There are several options to fix this problem.

• Lock the HashTable first, and make sure all other users of the HashTable do the same—client-based locking:

   synchronized(map) {  
   if(!map.conainsKey(key))  
       map.put(key, value);  
   }

• Wrap the HashTable in its own object and use a different API—server-based locking using an ADAPTER:

   public class WrappedHashtable<K, V> {  
       private Map<K, V> map = new Hashtable<K, V>();  
  
       public synchronized void putIfAbsent(K key, V value) {  
           if (map.containsKey(key))  
               map.put(key, value);  
       }  
   }

• Use the thread-safe collections:

   ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer,  
   String>();  
   map.putIfAbsent(key, value);

The collections in java.util.concurrent have operations like putIfAbsent() to accommodate such operations.

### **DEPENDENCIES BETWEEN METHODS CAN BREAK CONCURRENT CODE**

Here is a trivial example of a way to introduce dependencies between methods:

   public class IntegerIterator implements Iterator<Integer>  
       private Integer nextValue = 0;  
  
       public synchronized boolean hasNext() {  
           return nextValue < 100000;  
       }  
       public synchronized Integer next() {  
           if (nextValue == 100000)  
               throw new IteratorPastEndException();  
           return nextValue++;  
       }  
       public synchronized Integer getNextValue() {  
           return nextValue;  
       }  
   }

Here is some code to use this IntegerIterator:

   IntegerIterator iterator = new IntegerIterator();  
   while(iterator.hasNext()) {  
      int nextValue = iterator.next();  
      // do something with nextValue  
   }

If one thread executes this code, there will be no problem. But what happens if two threads attempt to share a single instance of IngeterIterator with the intent that each thread will process the values it gets, but that each element of the list is processed only once? Most of the time, nothing bad happens; the threads happily share the list, processing the elements they are given by the iterator and stopping when the iterator is complete. However, there is a small chance that, at the end of the iteration, the two threads will interfere with each other and cause one thread to go beyond the end of the iterator and throw an exception.

Here’s the problem: Thread 1 asks the question hasNext(), which returns true. Thread 1 gets preempted and then Thread 2 asks the same question, which is still true. Thread 2 then calls next(), which returns a value as expected but has a side effect of making hasNext() return false. Thread 1 starts up again, thinking hasNext() is still true, and then calls next(). Even though the individual methods are synchronized, the client uses **two** methods.

This is a real problem and an example of the kinds of problems that crop up in concurrent code. In this particular situation this problem is especially subtle because the only time where this causes a fault is when it happens during the final iteration of the iterator. If the threads happen to break just right, then one of the threads could go beyond the end of the iterator. This is the kind of bug that happens long after a system has been in production, and it is hard to track down.

You have three options:

• Tolerate the failure.

• Solve the problem by changing the client: client-based locking

• Solve the problem by changing the server, which additionally changes the client: server-based locking

#### **Tolerate the Failure**

Sometimes you can set things up such that the failure causes no harm. For example, the above client could catch the exception and clean up. Frankly, this is a bit sloppy. It’s rather like cleaning up memory leaks by rebooting at midnight.

#### **Client-Based Locking**

To make IntegerIterator work correctly with multiple threads, change this client (and every other client) as follows:

   IntegerIterator iterator = new IntegerIterator();  
  
       while (true) {  
         int nextValue;  
     synchronized (iterator) {  
       if (!iterator.hasNext())  
         break;  
       nextValue = iterator.next();  
     }  
     doSometingWith(nextValue);  
   }

Each client introduces a lock via the synchronized keyword. This duplication violates the DRY principle, but it might be necessary if the code uses non-thread-safe third-party tools.

This strategy is risky because all programmers who use the server must remember to lock it before using it and unlock it when done. Many (many!) years ago I worked on a system that employed client-based locking on a shared resource. The resource was used in hundreds of different places throughout the code. One poor programmer forgot to lock the resource in one of those places.

The system was a multi-terminal time-sharing system running accounting software for Local 705 of the trucker’s union. The computer was in a raised-floor, environment-controlled room 50 miles north of the Local 705 headquarters. At the headquarters they had dozens of data entry clerks typing union dues postings into the terminals. The terminals were connected to the computer using dedicated phone lines and 600bps half-duplex modems. (This was a very, very long time ago.)

About once per day, one of the terminals would “lock up.” There was no rhyme or reason to it. The lock up showed no preference for particular terminals or particular times. It was as though there were someone rolling dice choosing the time and terminal to lock up. Sometimes more than one terminal would lock up. Sometimes days would go by without any lock-ups.

At first the only solution was a reboot. But reboots were tough to coordinate. We had to call the headquarters and get everyone to finish what they were doing on all the terminals. Then we could shut down and restart. If someone was doing something important that took an hour or two, the locked up terminal simply had to stay locked up.

After a few weeks of debugging we found that the cause was a ring-buffer counter that had gotten out of sync with its pointer. This buffer controlled output to the terminal. The pointer value indicated that the buffer was empty, but the counter said it was full. Because it was empty, there was nothing to display; but because it was also full, nothing could be added to the buffer to be displayed on the screen.

So we knew why the terminals were locking, but we didn’t know why the ring buffer was getting out of sync. So we added a hack to work around the problem. It was possible to read the front panel switches on the computer. (This was a very, very, very long time ago.) We wrote a little trap function that detected when one of these switches was thrown and then looked for a ring buffer that was both empty and full. If one was found, it reset that buffer to empty. Voila! The locked-up terminal(s) started displaying again.

So now we didn’t have to reboot the system when a terminal locked up. The Local would simply call us and tell us we had a lock-up, and then we just walked into the computer room and flicked a switch.

Of course sometimes they worked on the weekends, and we didn’t. So we added a function to the scheduler that checked all the ring buffers once per minute and reset any that were both empty and full. This caused the displays to unclog before the Local could even get on the phone.

It was several more weeks of poring over page after page of monolithic assembly language code before we found the culprit. We had done the math and calculated that the frequency of the lock-ups was consistent with a single unprotected use of the ring buffer. So all we had to do was find that one faulty usage. Unfortunately, this was so very long ago that we didn’t have search tools or cross references or any other kind of automated help. We simply had to pore over listings.

I learned an important lesson that cold Chicago winter of 1971. Client-based locking really blows.

#### **Server-Based Locking**

The duplication can be removed by making the following changes to IntegerIterator:

   public class IntegerIteratorServerLocked {  
       private Integer nextValue = 0;  
       public synchronized Integer getNextOrNull() {  
           if (nextValue < 100000)  
               return nextValue++;  
           else  
              return null;  
       }  
   }

And the client code changes as well:

   while (true) {  
       Integer nextValue = iterator.getNextOrNull();  
       if (next == null)  
           break;  
       // do something with nextValue  
   }

In this case we actually change the API of our class to be multithread aware.3 The client needs to perform a null check instead of checking hasNext().

3. In fact, the Iterator interface is inherently not thread-safe. It was never designed to be used by multiple threads, so this should come as no surprise.

In general you should prefer server-based locking for these reasons:

• It reduces repeated code—Client-based locking forces each client to lock the server properly. By putting the locking code into the server, clients are free to use the object and not worry about writing additional locking code.

• It allows for better performance—You can swap out a thread-safe server for a non-thread safe one in the case of single-threaded deployment, thereby avoiding all overhead.

• It reduces the possibility of error—All it takes is for one programmer to forget to lock properly.

• It enforces a single policy—The policy is in one place, the server, rather than many places, each client.

• It reduces the scope of the shared variables—The client is not aware of them or how they are locked. All of that is hidden in the server. When things break, the number of places to look is smaller.

What if you do not own the server code?

• Use an ADAPTER to change the API and add locking

   public class ThreadSafeIntegerIterator {  
       private IntegerIterator iterator = new IntegerIterator();  
  
       public synchronized Integer getNextOrNull() {  
           if(iterator.hasNext())  
               return iterator.next();  
           return null;  
       }  
   }

• OR better yet, use the thread-safe collections with extended interfaces

### **INCREASING THROUGHPUT**

Let’s assume that we want to go out on the net and read the contents of a set of pages from a list of URLs. As each page is read, we will parse it to accumulate some statistics. Once all the pages are read, we will print a summary report.

The following class returns the contents of one page, given a URL.

   public class PageReader {  
     //…  
     public String getPageFor(String url) {  
       HttpMethod method = new GetMethod(url);  
  
       try {  
         httpClient.executeMethod(method);  
         String response = method.getResponseBodyAsString();  
         return response;  
       } catch (Exception e) {  
         handle(e);  
       } finally {  
         method.releaseConnection();  
       }  
     }  
   }

The next class is the iterator that provides the contents of the pages based on an iterator of URLs:

   public class PageIterator {  
     private PageReader reader;  
     private URLIterator urls;  
  
     public PageIterator(PageReader reader, URLIterator urls) {  
       this.urls = urls;  
       this.reader = reader;  
     }  
  
     public synchronized String getNextPageOrNull() {  
       if (urls.hasNext())  
         getPageFor(urls.next());  
       else  
         return null;  
     }  
  
     public String getPageFor(String url) {  
       return reader.getPageFor(url);  
     }  
   }

An instance of the PageIterator can be shared between many different threads, each one using it’s own instance of the PageReader to read and parse the pages it gets from the iterator.

Notice that we’ve kept the synchronized block very small. It contains just the critical section deep inside the PageIterator. It is always better to synchronize as little as possible as opposed to synchronizing as much as possible.

#### **Single-Thread Calculation of Throughput**

Now lets do some simple calculations. For the purpose of argument, assume the following:

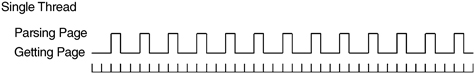
• I/O time to retrieve a page (average): 1 second

• Processing time to parse page (average): .5 seconds

• I/O requires 0 percent of the CPU while processing requires 100 percent.

For N pages being processed by a single thread, the total execution time is 1.5 seconds \* N. [Figure A-1](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1fig1)shows a snapshot of 13 pages or about 19.5 seconds.

**Figure A-1** Single thread

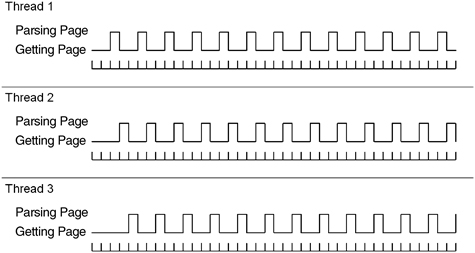


#### **Multithread Calculation of Throughput**

If it is possible to retrieve pages in any order and process the pages independently, then it is possible to use multiple threads to increase throughput. What happens if we use three threads? How many pages can we acquire in the same time?

As you can see in [Figure A-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1fig2), the multithreaded solution allows the process-bound parsing of the pages to overlap with the I/O-bound reading of the pages. In an idealized world this means that the processor is fully utilized. Each one-second page read is overlapped with two parses. Thus, we can process two pages per second, which is three times the throughput of the single-threaded solution.

**Figure A-2** Three concurrent threads



### **DEADLOCK**

Imagine a Web application with two shared resource pools of some finite size:

• A pool of database connections for local work in process storage

• A pool of MQ connections to a master repository

Assume there are two operations in this application, create and update:

• Create—Acquire connection to master repository and database. Talk to service master repository and then store work in local work in process database.

• Update—Acquire connection to database and then master repository. Read from work in process database and then send to the master repository

What happens when there are more users than the pool sizes? Consider each pool has a size of ten.

• Ten users attempt to use create, so all ten database connections are acquired, and each thread is interrupted after acquiring a database connection but before acquiring a connection to the master repository.

• Ten users attempt to use update, so all ten master repository connections are acquired, and each thread is interrupted after acquiring the master repository but before acquiring a database connection.

• Now the ten “create” threads must wait to acquire a master repository connection, but the ten “update” threads must wait to acquire a database connection.

• Deadlock. The system never recovers.

This might sound like an unlikely situation, but who wants a system that freezes solid every other week? Who wants to debug a system with symptoms that are so difficult to reproduce? This is the kind of problem that happens in the field, then takes weeks to solve.

A typical “solution” is to introduce debugging statements to find out what is happening. Of course, the debug statements change the code enough so that the deadlock happens in a different situation and takes months to again occur.4

4. For example, someone adds some debugging output and the problem “disappears.” The debugging code “fixes” the problem so it remains in the system.

To really solve the problem of deadlock, we need to understand what causes it. There are four conditions required for deadlock to occur:

• Mutual exclusion

• Lock & wait

• No preemption

• Circular wait

#### **Mutual Exclusion**

Mutual exclusion occurs when multiple threads need to use the same resources and those resources

• Cannot be used by multiple threads at the same time.

• Are limited in number.

A common example of such a resource is a database connection, a file open for write, a record lock, or a semaphore.

#### **Lock & Wait**

Once a thread acquires a resource, it will not release the resource until it has acquired all of the other resources it requires and has completed its work.

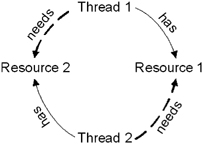
#### **No Preemption**

One thread cannot take resources away from another thread. Once a thread holds a resource, the only way for another thread to get it is for the holding thread to release it.

#### **Circular Wait**

This is also referred to as the deadly embrace. Imagine two threads, T1 and T2, and two resources, R1 and R2. T1 has R1, T2 has R2. T1 also requires R2, and T2 also requires R1. This gives something like [Figure A-3](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1fig3):

**Figure A-3**



All four of these conditions must hold for deadlock to be possible. Break any one of these conditions and deadlock is not possible.

#### **Breaking Mutual Exclusion**

One strategy for avoiding deadlock is to sidestep the mutual exclusion condition. You might be able to do this by

• Using resources that allow simultaneous use, for example, AtomicInteger.

• Increasing the number of resources such that it equals or exceeds the number of competing threads.

• Checking that all your resources are free before seizing any.

Unfortunately, most resources are limited in number and don’t allow simultaneous use. And it’s not uncommon for the identity of the second resource to be predicated on the results of operating on the first. But don’t be discouraged; there are three conditions left.

#### **Breaking Lock & Wait**

You can also eliminate deadlock if you refuse to wait. Check each resource before you seize it, and release all resources and start over if you run into one that’s busy.

This approach introduces several potential problems:

• Starvation—One thread keeps being unable to acquire the resources it needs (maybe it has a unique combination of resources that seldom all become available).

• Livelock—Several threads might get into lockstep and all acquire one resource and then release one resource, over and over again. This is especially likely with simplistic CPU scheduling algorithms (think embedded devices or simplistic hand-written thread balancing algorithms).

Both of these can cause poor throughput. The first results in low CPU utilization, whereas the second results in high and useless CPU utilization.

As inefficient as this strategy sounds, it’s better than nothing. It has the benefit that it can almost always be implemented if all else fails.

#### **Breaking Preemption**

Another strategy for avoiding deadlock is to allow threads to take resources away from other threads. This is usually done through a simple request mechanism. When a thread discovers that a resource is busy, it asks the owner to release it. If the owner is also waiting for some other resource, it releases them all and starts over.

This is similar to the previous approach but has the benefit that a thread is allowed to wait for a resource. This decreases the number of startovers. Be warned, however, that managing all those requests can be tricky.

#### **Breaking Circular Wait**

This is the most common approach to preventing deadlock. For most systems it requires no more than a simple convention agreed to by all parties.

In the example above with Thread 1 wanting both Resource 1 and Resource 2 and Thread 2 wanting both Resource 2 and then Resource 1, simply forcing both Thread 1 and Thread 2 to allocate resources in the same order makes circular wait impossible.

More generally, if all threads can agree on a global ordering of resources and if they all allocate resources in that order, then deadlock is impossible. Like all the other strategies, this can cause problems:

• The order of acquisition might not correspond to the order of use; thus a resource acquired at the start might not be used until the end. This can cause resources to be locked longer than strictly necessary.

• Sometimes you cannot impose an order on the acquisition of resources. If the ID of the second resource comes from an operation performed on the first, then ordering is not feasible.

So there are many ways to avoid deadlock. Some lead to starvation, whereas others make heavy use of the CPU and reduce responsiveness. TANSTAAFL!5

5. There ain’t no such thing as a free lunch.

Isolating the thread-related part of your solution to allow for tuning and experimentation is a powerful way to gain the insights needed to determine the best strategies.

### **TESTING MULTITHREADED CODE**

How can we write a test to demonstrate the following code is broken?

   01: public class ClassWithThreadingProblem {  
   02:    int nextId;  
   03:  
   04:    public int takeNextId() {  
   05:        return nextId++;  
   06:    }  
   07:}

Here’s a description of a test that will prove the code is broken:

• Remember the current value of nextId.

• Create two threads, both of which call takeNextId() once.

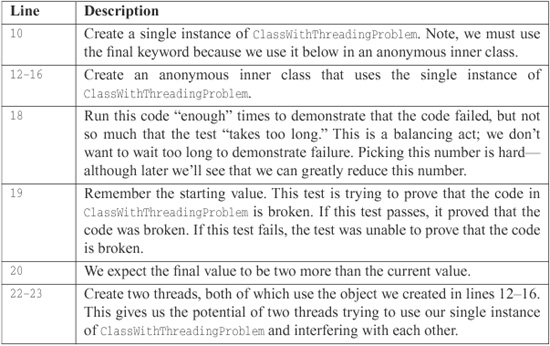
• Verify that nextId is two more than what we started with.

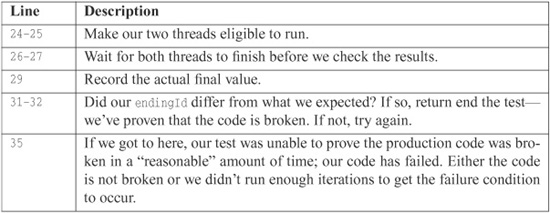
• Run this until we demonstrate that nextId was only incremented by one instead of two.

[Listing A-2](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/app01.html#app1lt2) shows such a test:

**Listing A-2 ClassWithThreadingProblemTest.java**

   01: package example;  
   02:  
   03: import static org.junit.Assert.fail;  
   04:  
   05: import org.junit.Test;  
   06:  
   07: public class ClassWithThreadingProblemTest {  
   08:     @Test  
   09:     public void twoThreadsShouldFailEventually() throws Exception {  
   10:         final ClassWithThreadingProblem classWithThreadingProblem  
                   = new ClassWithThreadingProblem();  
   11:  
   12:         Runnable runnable = new Runnable() {  
   13:             public void run() {  
   14:                 classWithThreadingProblem.takeNextId();  
   15:             }  
   16:         };  
   17:  
   18:         for (int i = 0; i < 50000; ++i) {  
   19:             int startingId = classWithThreadingProblem.lastId;  
   20:             int expectedResult = 2 + startingId;  
   21:  
   22:             Thread t1 = new Thread(runnable);  
   23:             Thread t2 = new Thread(runnable);  
   24:             t1.start();  
   25:             t2.start();  
   26:             t1.join();  
   27:             t2.join();  
   28:  
   29:             int endingId = classWithThreadingProblem.lastId;  
   30:  
   31:             if (endingId != expectedResult)  
   32:                 return;  
   33:         }  
   34:  
   35:         fail(“Should have exposed a threading issue but it did not.”);  
   36:     }  
   37: }





This test certainly sets up the conditions for a concurrent update problem. However, the problem occurs so infrequently that the vast majority of times this test won’t detect it.

Indeed, to truly detect the problem we need to set the number of iterations to over one million. Even then, in ten executions with a loop count of 1,000,000, the problem occurred only once. That means we probably ought to set the iteration count to well over one hundred million to get reliable failures. How long are we prepared to wait?

Even if we tuned the test to get reliable failures on one machine, we’ll probably have to retune the test with different values to demonstrate the failure on another machine, operating system, or version of the JVM.

And this is a simple problem. If we cannot demonstrate broken code easily with this problem, how will we ever detect truly complex problems?

So what approaches can we take to demonstrate this simple failure? And, more importantly, how can we write tests that will demonstrate failures in more complex code? How will we be able to discover if our code has failures when we do not know where to look?

Here are a few ideas:

• **Monte Carlo Testing.** Make tests flexible, so they can be tuned. Then run the test over and over—say on a test server—randomly changing the tuning values. If the tests ever fail, the code is broken. Make sure to start writing those tests early so a continuous integration server starts running them soon. By the way, make sure you carefully log the conditions under which the test failed.

• Run the test on every one of the target deployment platforms. Repeatedly. Continuously. The longer the tests run without failure, the more likely that

– The production code is correct or

– The tests aren’t adequate to expose problems.

• Run the tests on a machine with varying loads. If you can simulate loads close to a production environment, do so.

Yet, even if you do all of these things, you still don’t stand a very good chance of finding threading problems with your code. The most insidious problems are the ones that have such a small cross section that they only occur once in a billion opportunities. Such problems are the terror of complex systems.

### **TOOL SUPPORT FOR TESTING THREAD-BASED CODE**

IBM has created a tool called ConTest.6 It instruments classes to make it more likely that non-thread-safe code fails.

6. <http://www.haifa.ibm.com/projects/verification/contest/index.html>

We do not have any direct relationship with IBM or the team that developed ConTest. A colleague of ours pointed us to it. We noticed vast improvement in our ability to find threading issues after a few minutes of using it.

Here’s an outline of how to use ConTest:

• Write tests and production code, making sure there are tests specifically designed to simulate multiple users under varying loads, as mentioned above.

• Instrument test and production code with ConTest.

• Run the tests.

When we instrumented code with ConTest, our success rate went from roughly one failure in ten million iterations to roughly one failure in thirty iterations. Here are the loop values for several runs of the test after instrumentation: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. So clearly the instrumented classes failed much earlier and with much greater reliability.

### **CONCLUSION**

This chapter has been a very brief sojourn through the large and treacherous territory of concurrent programming. We barely scratched the surface. Our emphasis here was on disciplines to help keep concurrent code clean, but there is much more you should learn if you are going to be writing concurrent systems. We recommend you start with Doug Lea’s wonderful book Concurrent Programming in Java: Design Principles and Patterns.7

7. See [[Lea99](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter13.html#Lea99)] p. [191](https://www.safaribooksonline.com/library/view/clean-code/9780136083238/chapter13.html#page_191).

In this chapter we talked about concurrent update, and the disciplines of clean synchronization and locking that can prevent it. We talked about how threads can enhance the throughput of an I/O-bound system and showed the clean techniques for achieving such improvements. We talked about deadlock and the disciplines for preventing it in a clean way. Finally, we talked about strategies for exposing concurrent problems by instrumenting your code.

### **TUTORIAL: FULL CODE EXAMPLES**

#### **Client/Server Nonthreaded**

**Listing A-3 Server.java**

   package com.objectmentor.clientserver.nonthreaded;  
  
   import java.io.IOException;  
   import java.net.ServerSocket;  
   import java.net.Socket;  
   import java.net.SocketException;  
  
   import common.MessageUtils;  
  
   public class Server implements Runnable {  
       ServerSocket serverSocket;  
       volatile boolean keepProcessing = true;  
  
       public Server(int port, int millisecondsTimeout) throws IOException {  
           serverSocket = new ServerSocket(port);  
           serverSocket.setSoTimeout(millisecondsTimeout);  
       }  
  
       public void run() {  
           System.out.printf(“Server Starting\n”);  
  
           while (keepProcessing) {  
               try {  
                   System.out.printf(“accepting client\n”);  
                   Socket socket = serverSocket.accept();  
                   System.out.printf(“got client\n”);  
                   process(socket);  
               } catch (Exception e) {  
                   handle(e);  
               }  
           }  
       }  
  
       private void handle(Exception e) {  
           if (!(e instanceof SocketException)) {  
               e.printStackTrace();  
           }  
       }  
  
       public void stopProcessing() {  
           keepProcessing = false;  
           closeIgnoringException(serverSocket);  
       }  
       void process(Socket socket) {  
           if (socket == null)  
               return;  
  
           try {  
               System.out.printf(“Server: getting message\n”);  
               String message = MessageUtils.getMessage(socket);  
               System.out.printf(“Server: got message: %s\n”, message);  
               Thread.sleep(1000);  
               System.out.printf(“Server: sending reply: %s\n”, message);  
               MessageUtils.sendMessage(socket, “Processed: ” + message);  
               System.out.printf(“Server: sent\n”);  
               closeIgnoringException(socket);  
           } catch (Exception e) {  
               e.printStackTrace();  
           }  
  
       }  
  
       private void closeIgnoringException(Socket socket) {  
           if (socket != null)  
               try {  
                   socket.close();  
               } catch (IOException ignore) {  
               }  
       }  
  
       private void closeIgnoringException(ServerSocket serverSocket) {  
           if (serverSocket != null)  
               try {  
                   serverSocket.close();  
               } catch (IOException ignore) {  
               }  
       }  
   }

**Listing A-4 ClientTest.java**

package com.objectmentor.clientserver.nonthreaded;  
  
import java.io.IOException;  
import java.net.Socket;  
  
import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
  
import common.MessageUtils;  
  
  
public class ClientTest {  
    private static final int PORT = 8009;  
    private static final int TIMEOUT = 2000;  
  
    Server server;  
    Thread serverThread;  
  
    @Before  
    public void createServer() throws Exception {  
        try {  
            server = new Server(PORT, TIMEOUT);  
            serverThread = new Thread(server);  
            serverThread.start();  
        } catch (Exception e) {  
            e.printStackTrace(System.err);  
            throw e;  
        }  
    }  
  
    @After  
    public void shutdownServer() throws InterruptedException {  
        if (server != null) {  
            server.stopProcessing();  
            serverThread.join();  
        }  
    }  
  
    class TrivialClient implements Runnable {  
        int clientNumber;  
  
        TrivialClient(int clientNumber) {  
            this.clientNumber = clientNumber;  
        }  
  
        public void run() {  
            try {  
                connectSendReceive(clientNumber);  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
    @Test(timeout = 10000)  
    public void shouldRunInUnder10Seconds() throws Exception {  
        Thread[] threads = new Thread[10];  
  
  
        for (int i = 0; i < threads.length; ++i) {  
            threads[i] = new Thread(new TrivialClient(i));  
            threads[i].start();  
        }  
  
        for (int i = 0; i < threads.length; ++i) {  
            threads[i].join();  
        }  
    }  
  
    private void connectSendReceive(int i) throws IOException {  
        System.out.printf("Client %2d: connecting\n", i);  
        Socket socket = new Socket("localhost", PORT);  
        System.out.printf("Client %2d: sending message\n", i);  
        MessageUtils.sendMessage(socket, Integer.toString(i));  
        System.out.printf("Client %2d: getting reply\n", i);  
        MessageUtils.getMessage(socket);  
        System.out.printf("Client %2d: finished\n", i);  
        socket.close();  
    }  
}

**Listing A-5 MessageUtils.java**

   package common;  
  
   import java.io.IOException;  
   import java.io.InputStream;  
   import java.io.ObjectInputStream;  
   import java.io.ObjectOutputStream;  
   import java.io.OutputStream;  
   import java.net.Socket;  
  
   public class MessageUtils {  
       public static void sendMessage(Socket socket, String message)  
               throws IOException {  
          OutputStream stream = socket.getOutputStream();  
          ObjectOutputStream oos = new ObjectOutputStream(stream);  
          oos.writeUTF(message);  
          oos.flush();  
       }  
  
       public static String getMessage(Socket socket) throws IOException {  
           InputStream stream = socket.getInputStream();  
           ObjectInputStream ois = new ObjectInputStream(stream);  
           return ois.readUTF();  
       }  
   }

#### **Client/Server Using Threads**

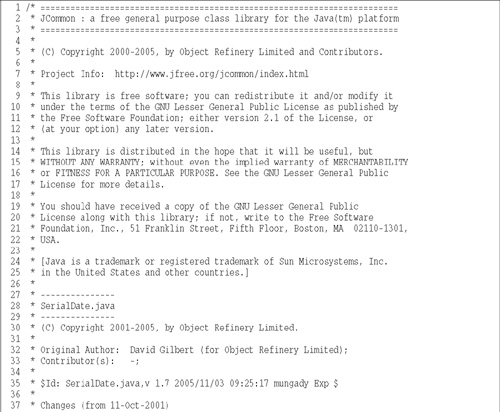
Changing the server to use threads simply requires a change to the process message (new lines are emphasized to stand out):

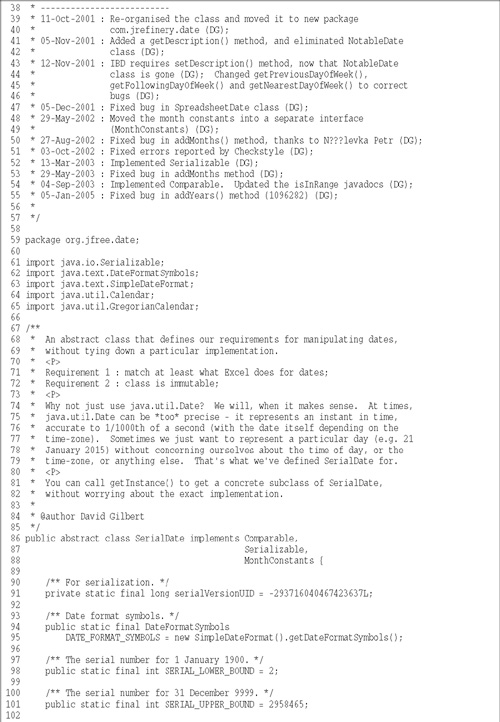
   void process(final Socket socket) {  
       if (socket == null)  
           return;  
  
       **Runnable clientHandler = new Runnable() {**  
           public void run() {  
  
               try {  
                   System.out.printf("Server: getting message\n");  
                   String message = MessageUtils.getMessage(socket);  
                   System.out.printf("Server: got message: %s\n", message);  
                    Thread.sleep(1000);  
                    System.out.printf("Server: sending reply: %s\n", message);  
                    MessageUtils.sendMessage(socket, "Processed: " + message);  
                   System.out.printf("Server: sent\n");  
                   closeIgnoringException(socket);  
                } catch (Exception e) {  
                   e.printStackTrace();  
               }  
           }  
       };  
  
       **Thread clientConnection = new Thread(clientHandler);  
       clientConnection.start();**  
   }

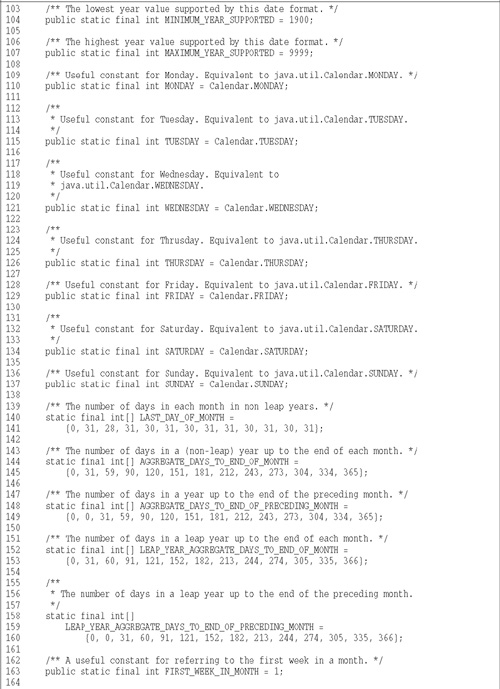
## ****Appendix B**** org.jfree.date.SerialDate

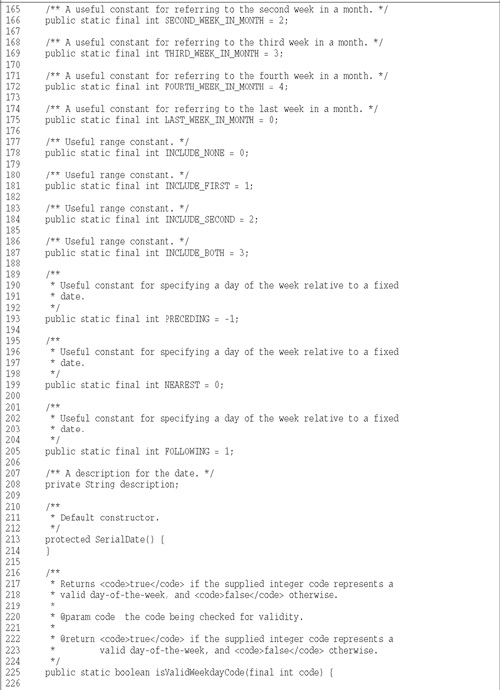
**Listing B-1**

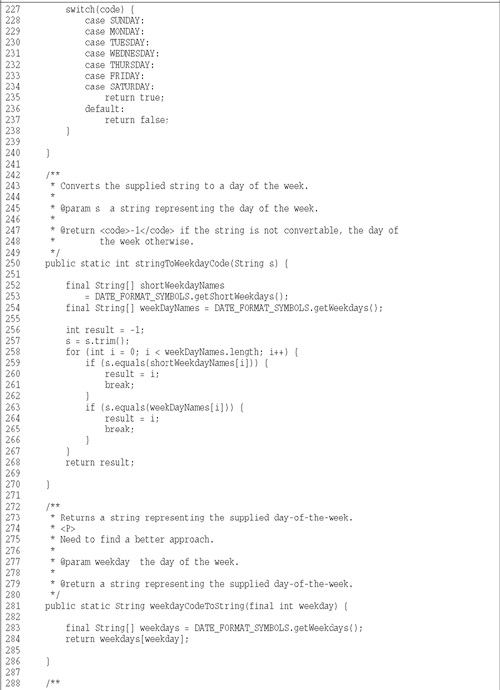
**SerialDate.Java**

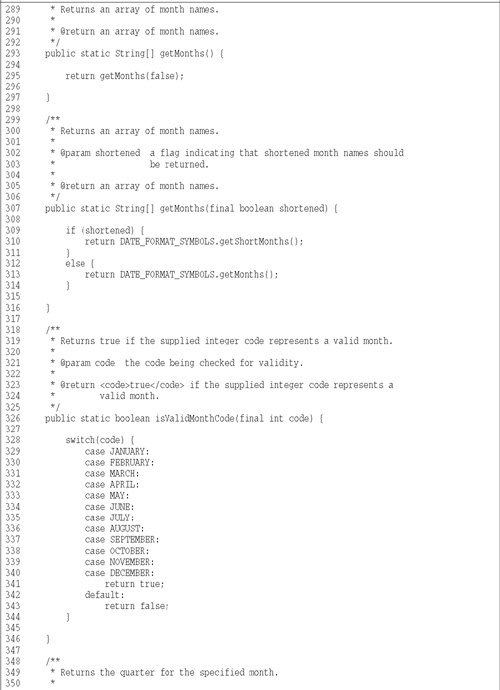


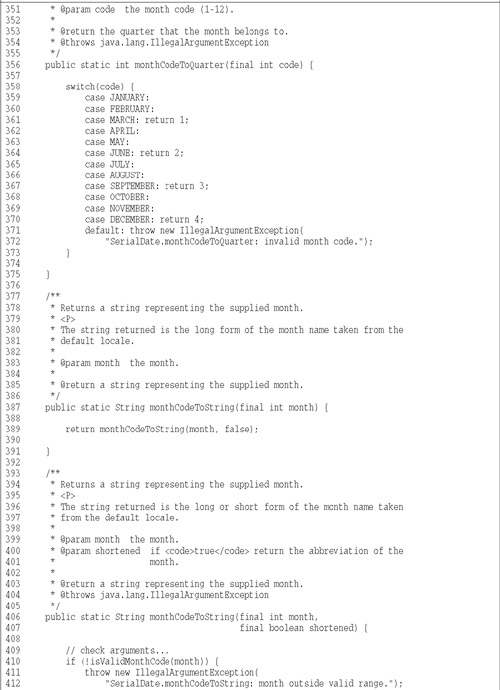


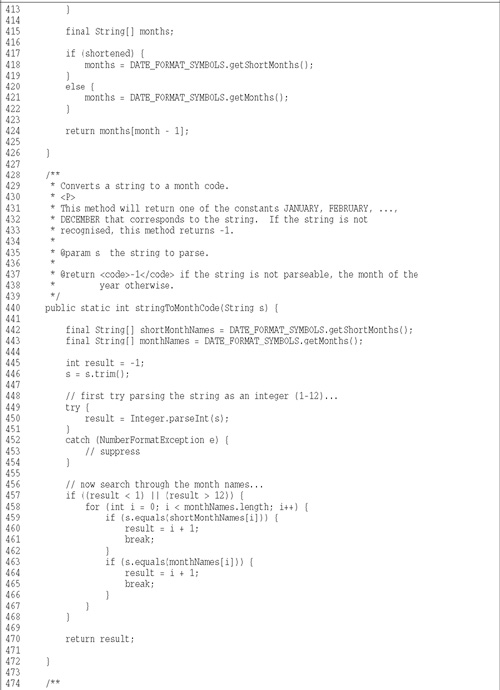


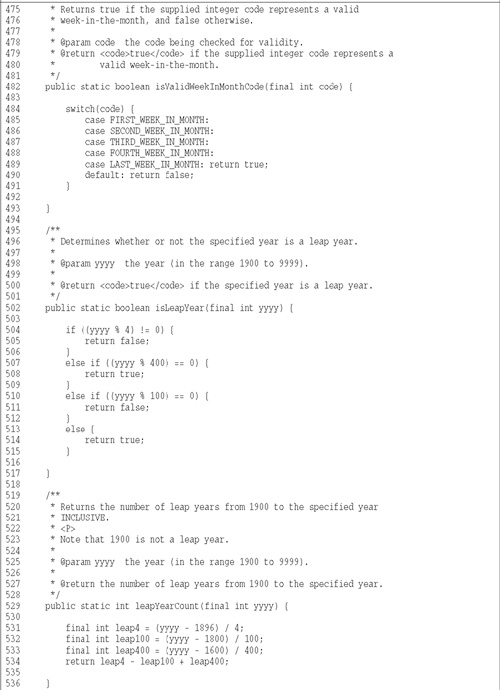






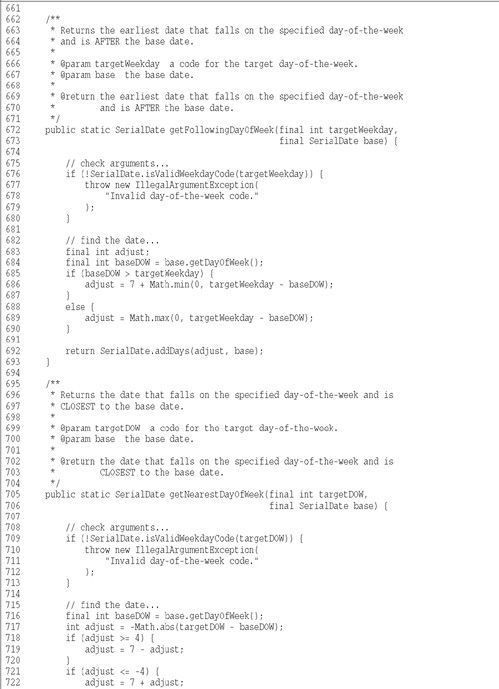


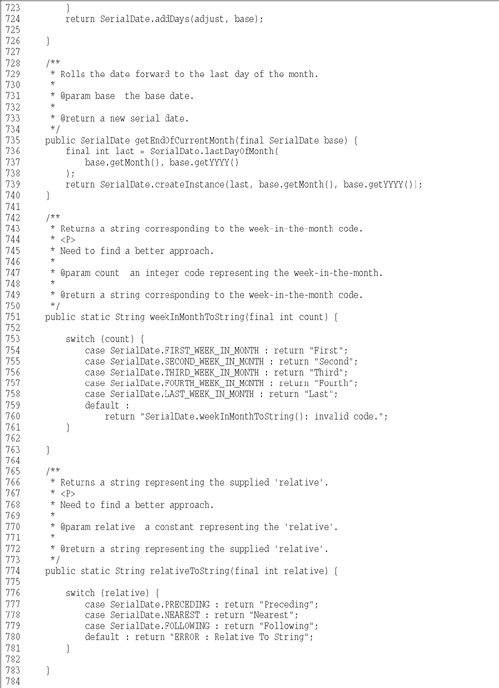


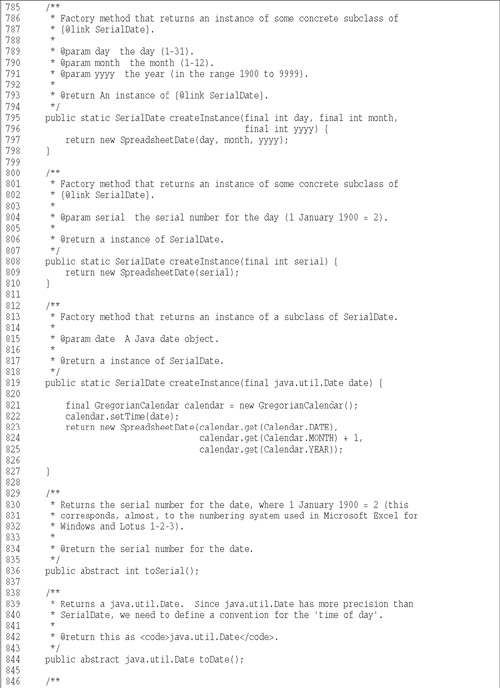


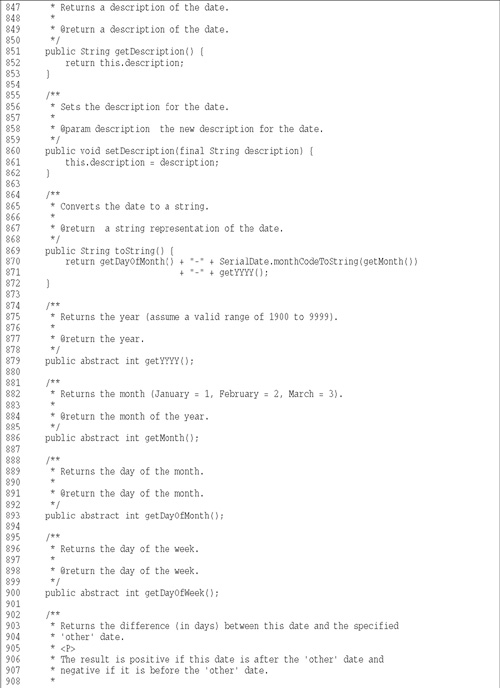


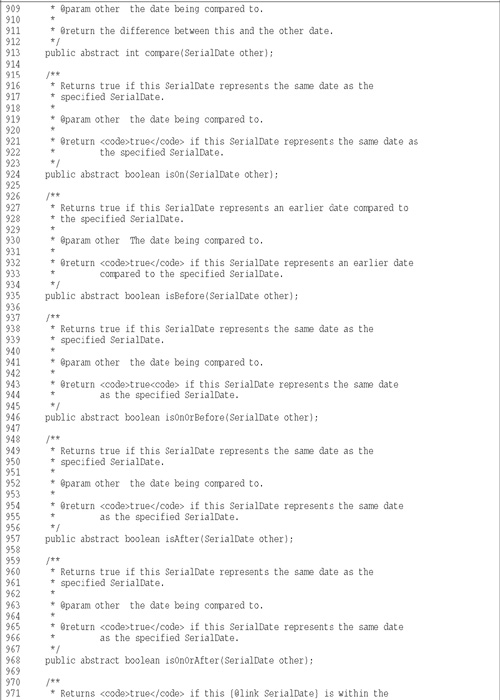


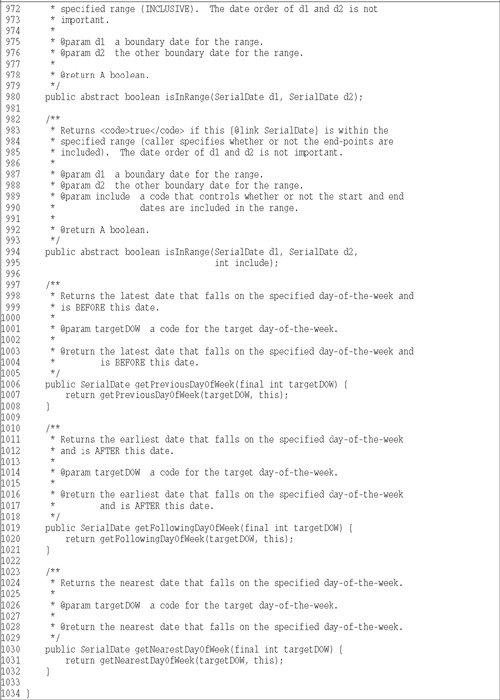






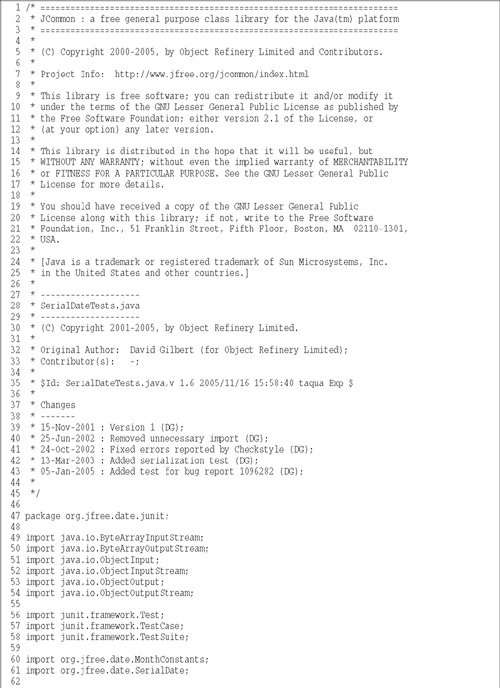


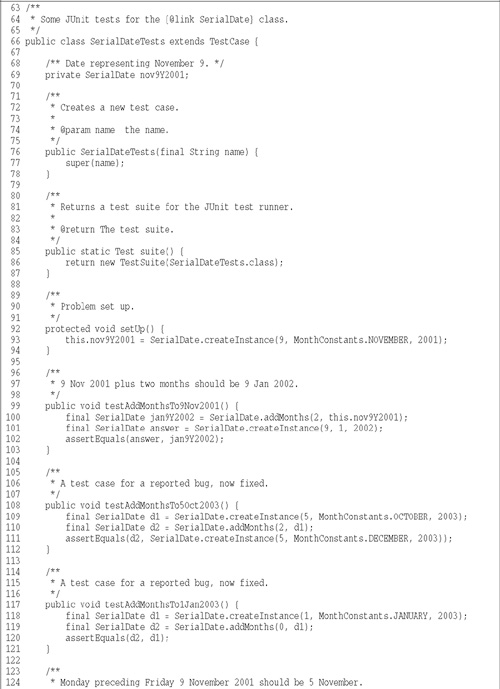


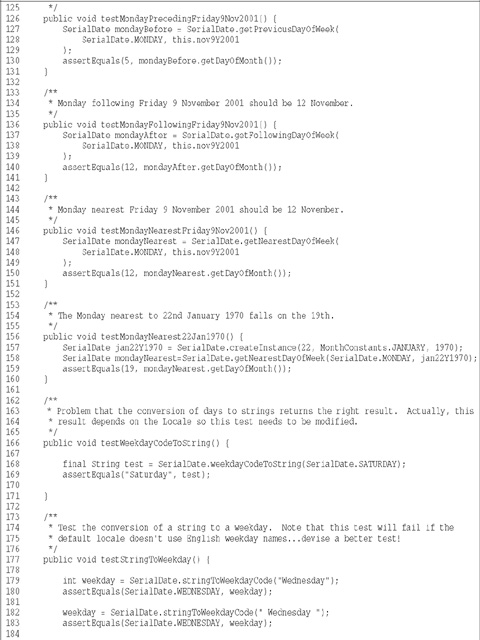


**Listing B-2**

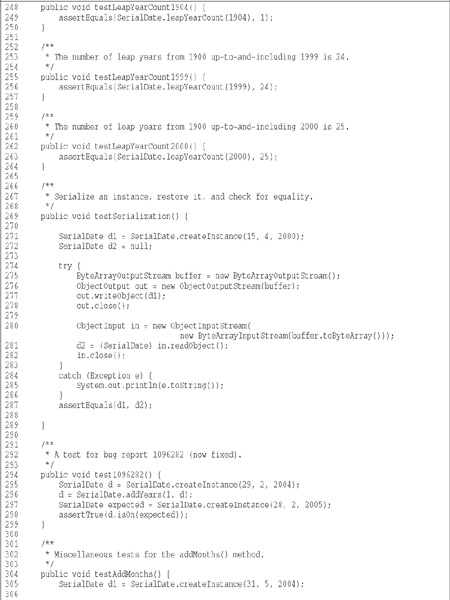
**SerialDateTest.java**

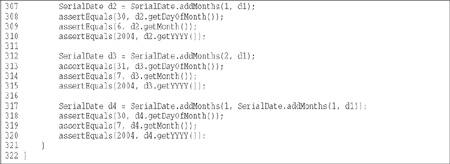






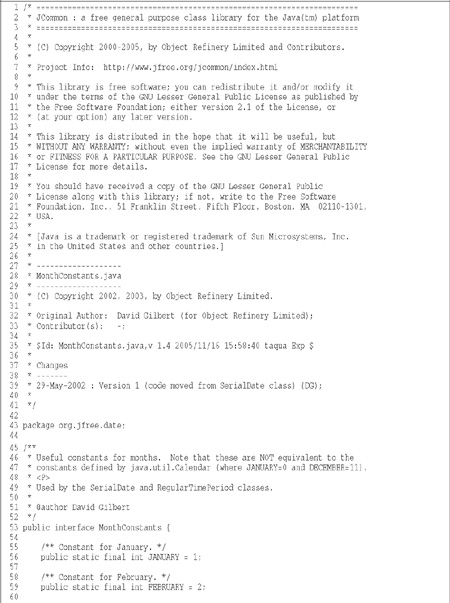


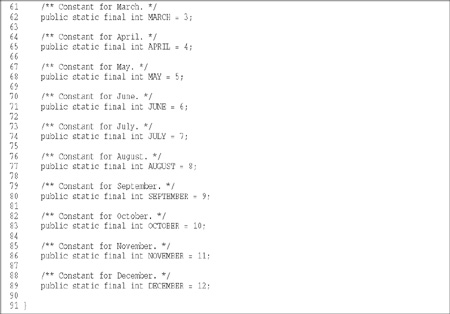




**Listing B-3**

**MonthConstants.java**



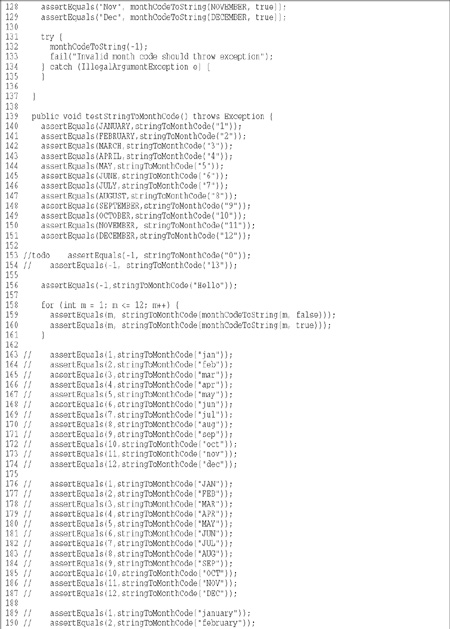


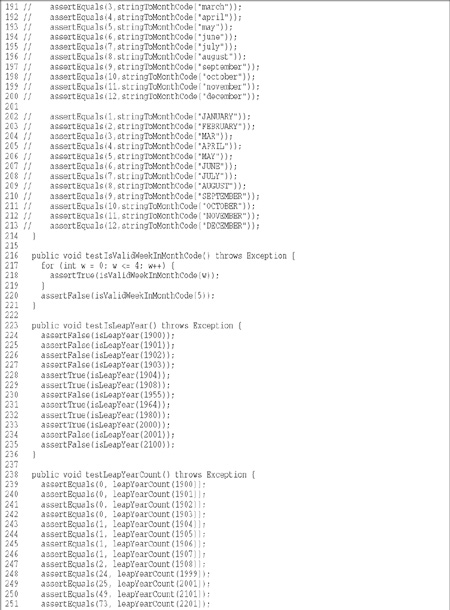
**Listing B-4**

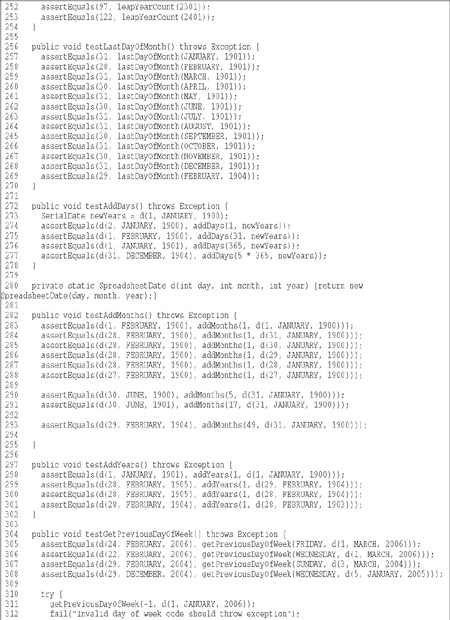
**BobsSerialDateTest.java**











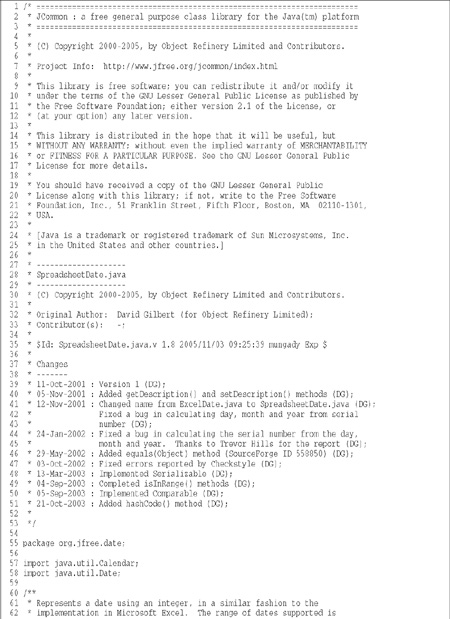


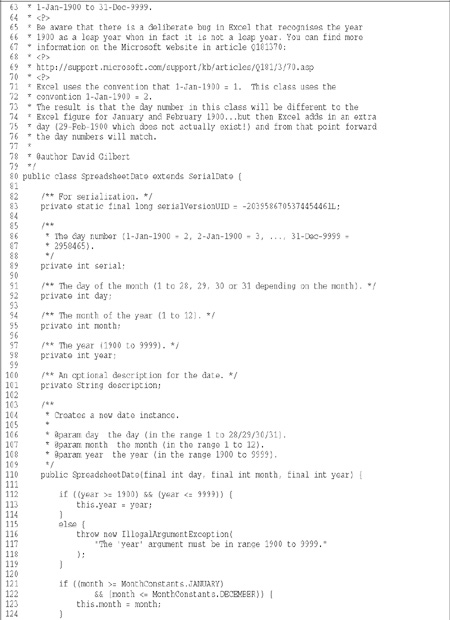


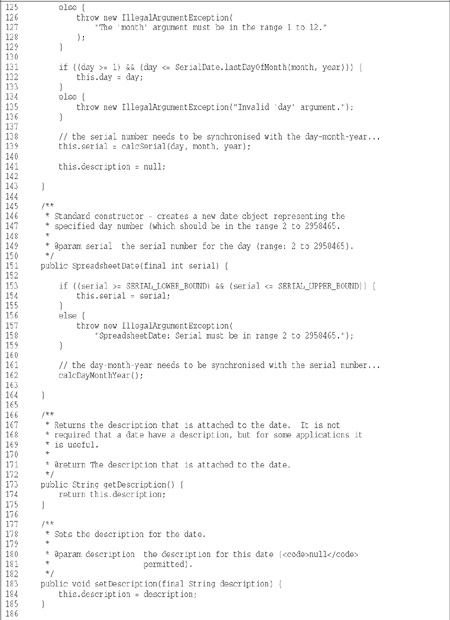


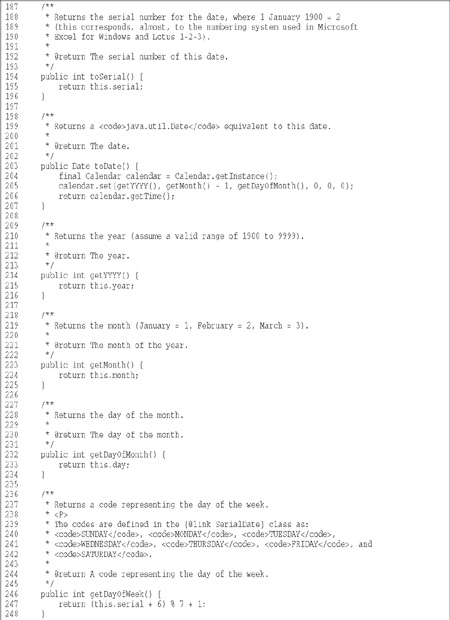
**Listing B-5**

**SpreadsheetDate.java**

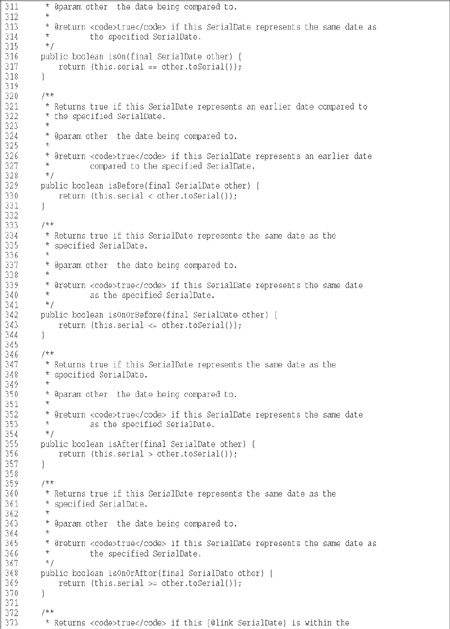


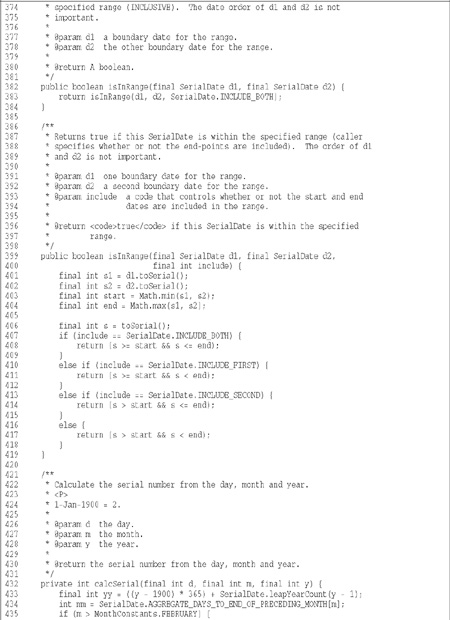


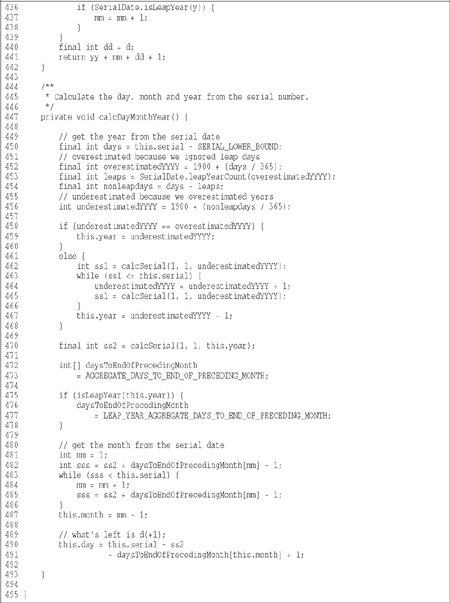








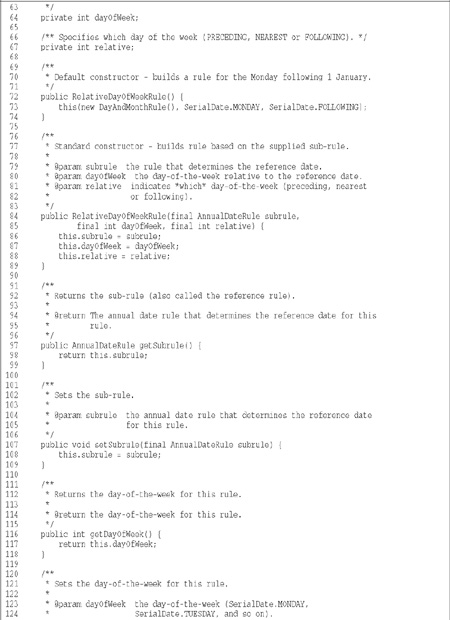


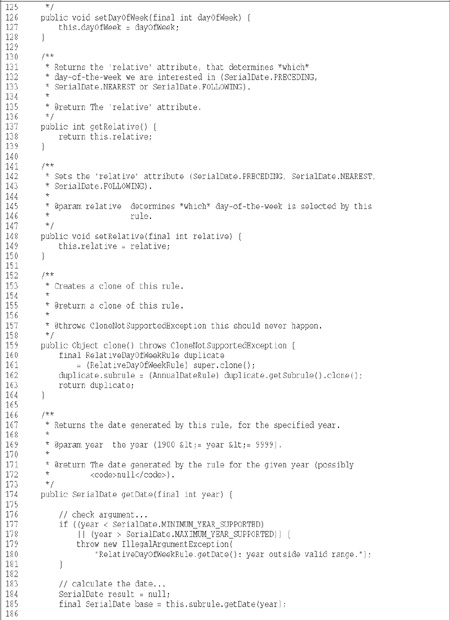


**Listing B-6**

**RelativeDayOfWeekRule.java**









**Listing B-7**

**DayDate.java (Final)**

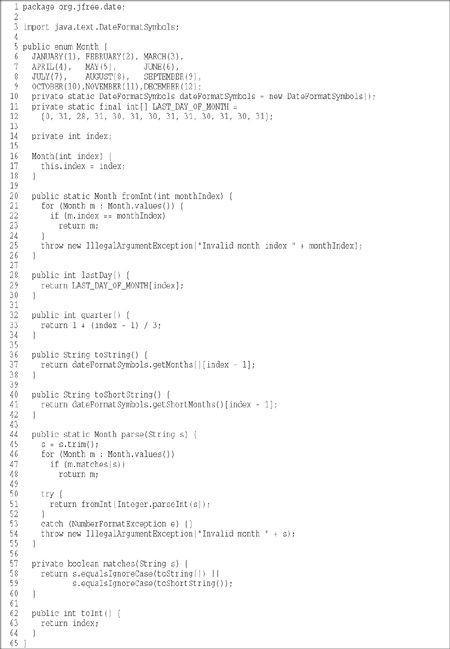






**Listing B-8**

**Month.java (Final)**



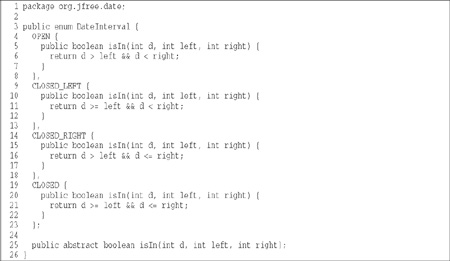
**Listing B-9**

**Day.java (Final)**



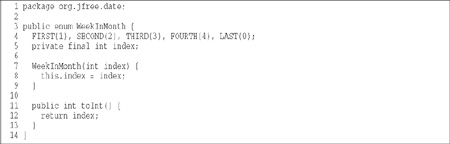
**Listing B-10**

**DateInterval.java (Final)**



**Listing B-11**

**WeekInMonth.java (Final)**



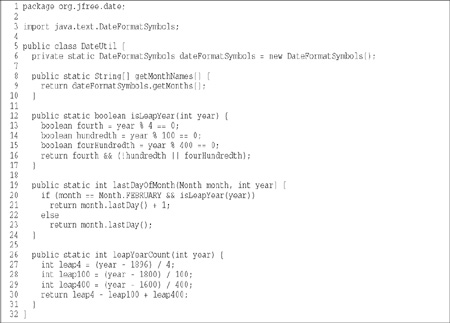
**Listing B-12**

**WeekdayRange.java (Final)**

Image

**Listing B-13**

**DateUtil.java (Final)**



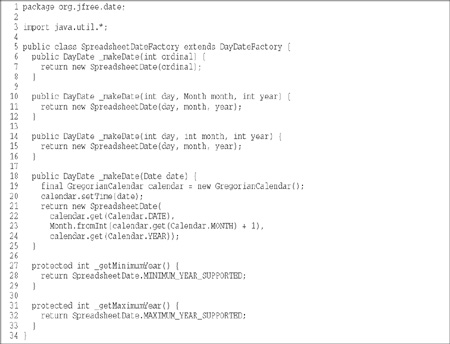
**Listing B-14**

**DayDateFactory.java (Final)**



**Listing B-15**

**SpreadsheetDateFactory.java (Final)**



**Listing B-16**

**SpreadsheetDate.java (Final)**



