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****

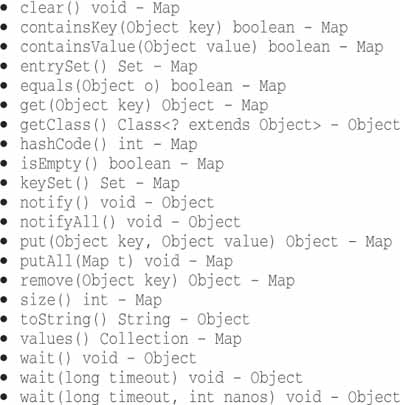
We seldom control all the software in our systems. Sometimes we buy third-party packages or use open source. Other times we depend on teams in our own company to produce components or subsystems for us. Somehow we must cleanly integrate this foreign code with our own. In this chapter we look at practices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

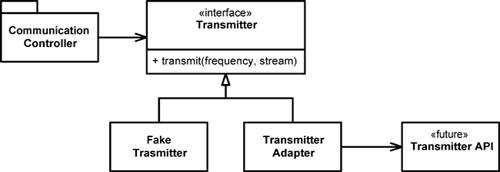
To keep from being blocked, we defined our own interface. We called it something catchy, like Transmitter. We gave it a method called transmit that took a frequency and a data stream. This was the interface we wished we had.

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****