The answers to these questions are not present in the code sample, *but they could have been*. Say that we're working in a mine sweeper game. We find that the board is a list of cells called `theList`. Let's rename that to `gameBoard`.

Each cell on the board is represented by a simple array. We further find that the zeroth subscript is the location of a status value and that a status value of 4 means "flagged." Just by giving these concepts names we can improve the code considerably:

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Notice that the simplicity of the code has not changed. It still has exactly the same number of operators and constants, with exactly the same number of nesting levels. But the code has become much more explicit.

We can go further and write a simple class for cells instead of using an array of `int`s. It can include an intention-revealing function (call it `isFlagged`) to hide the magic numbers. It results in a new version of the function:

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

With these simple name changes, it's not difficult to understand what's going on. This is the power of choosing good names.

Avoid Disinformation

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning. For example, `hp`, `aix`, and `sco` would be poor variable names because they are the names of Unix platforms or variants. Even if you are coding a hypotenuse and `hp` looks like a good abbreviation, it could be disinformative.

Do not refer to a grouping of accounts as an `accountList` unless it's actually a `List`. The word list means something specific to programmers. If the container holding the accounts is not actually a `List`, it may lead to false conclusions.[1] So `accountGroup` or `bunchOfAccounts` or just plain `accounts` would be better.

1. As we'll see later on, even if the container *is* a `List`, it's probably better not to encode the container type into the name.

Beware of using names which vary in small ways. How long does it take to spot the subtle difference between a `XYZControllerForEfficientHandlingOfStrings` in one module and, somewhere a little more distant, `XYZControllerForEfficientStorageOfStrings`? The words have frightfully similar shapes.

Spelling similar concepts similarly is *information*. Using inconsistent spellings is *disinformation*. With modern Java environments we enjoy automatic code completion. We write a few characters of a name and press some hotkey combination (if that) and are rewarded with a list of possible completions for that name. It is very helpful if names for very similar things sort together alphabetically and if the differences are very obvious, because the developer is likely to pick an object by name without seeing your copious comments or even the list of methods supplied by that class.

A truly awful example of disinformative names would be the use of lower-case `L` or uppercase `O` as variable names, especially in combination. The problem, of course, is that they look almost entirely like the constants one and zero, respectively.

```
int a = l;
if ( O == l )
  a = O1;
else
  l = 01;
```

The reader may think this a contrivance, but we have examined code where such things were abundant. In one case the author of the code suggested using a different font so that the differences were more obvious, a solution that would have to be passed down to all future developers as oral tradition or in a written document. The problem is conquered with finality and without creating new work products by a simple renaming.

Make Meaningful Distinctions

Programmers create problems for themselves when they write code solely to satisfy a compiler or interpreter. For 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.[2]

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.

---

2. Consider, for example, the truly hideous practice of creating a variable named `klass` just because the name `class` was used for something else.

Number-series naming (`a1, a2, .. aN`) is the opposite of intentional naming. Such names are not disinformative—they are noninformative; they provide no clue to the author's intention. Consider:

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

This function reads much better when `source` and `destination` are used for the argument names.

Noise words are another meaningless distinction. Imagine that you have a `Product` class. If you have another called `ProductInfo` or `ProductData`, you have made the names different without making them mean anything different. `Info` and `Data` are indistinct noise words like `a`, `an`, and `the`.

Note that there is nothing wrong with using prefix conventions like `a` and `the` so long as they make a meaningful distinction. For example you might use `a` for all local variables and `the` for all function arguments.[3] The problem comes in when you decide to call a variable `theZork` because you already have another variable named `zork`.

Noise words are redundant. The word `variable` should never appear in a variable name. The word `table` should never appear in a table name. How is `NameString` better than `Name`? Would a `Name` ever be a floating point number? If so, it breaks an earlier rule about disinformation. Imagine finding one class named `Customer` and another named `CustomerObject`. What should you understand as the distinction? Which one will represent the best path to a customer's payment history?

There is an application we know of where this is illustrated. we've changed the names to protect the guilty, but here's the exact form of the error:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

How are the programmers in this project supposed to know which of these functions to call?

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

Humans are good at words. A significant part of our brains is dedicated to the concept of words. And words are, by definition, pronounceable. It would be a shame not to take

3. Uncle Bob used to do this in C++ but has given up the practice because modern IDEs make it unnecessary.

advantage of that huge portion of our brains that has evolved to deal with spoken language. So make your names pronounceable.

If you can't pronounce it, you can't discuss it without sounding like an idiot. "Well, over here on the bee cee arr three cee enn tee we have a pee ess zee kyew int, see?" This matters because programming is a social activity.

A company I know has `genymdhms` (generation date, year, month, day, hour, minute, and second) so they walked around saying "gen why emm dee aich emm ess". I have an annoying habit of pronouncing everything as written, so I started saying "gen-yah-mudda-hims." It later was being called this by a host of designers and analysts, and we still sounded silly. But we were in on the joke, so it was fun. Fun or not, we were tolerating poor naming. New developers had to have the variables explained to them, and then they spoke about it in silly made-up words instead of using proper English terms. Compare

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

to

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

Intelligent conversation is now possible: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

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.

One might easily grep for `MAX_CLASSES_PER_STUDENT`, but the number 7 could be more troublesome. Searches may turn up the digit as part of file names, other constant definitions, and in various expressions where the value is used with different intent. It is even worse when a constant is a long number and someone might have transposed digits, thereby creating a bug while simultaneously evading the programmer's search.

Likewise, the name `e` is a poor choice for any variable for which a programmer might need to search. It is the most common letter in the English language and likely to show up in every passage of text in every program. In this regard, longer names trump shorter names, and any searchable name trumps a constant in code.

My personal preference is that single-letter names can ONLY be used as local variables inside short methods. *The length of a name should correspond to the size of its scope*

[N5]. If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name. Once again compare

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

to

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Note that `sum`, above, is not a particularly useful name but at least is searchable. The intentionally named code makes for a longer function, but consider how much easier it will be to find `WORK_DAYS_PER_WEEK` than to find all the places where 5 was used and filter the list down to just the instances with the intended meaning.

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.

HN was considered to be pretty important back in the Windows C API, when everything was an integer handle or a long pointer or a `void` pointer, or one of several implementations of "string" (with different uses and attributes). The compiler did not check types in those days, so the programmers needed a crutch to help them remember the types.

In modern languages we have much richer type systems, and the compilers remember and enforce the types. What's more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they're using.

Java programmers don't need type encoding. Objects are strongly typed, and editing environments have advanced such that they detect a type error long before you can run a compile! So nowadays HN and other forms of type encoding are simply impediments. They make it harder to change the name or type of a variable, function, or class. They make it harder to read the code. And they create the possibility that the encoding system will mislead the reader.

```
PhoneNumber phoneString;
// name not changed when type changed!
```

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.

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Besides, people quickly learn to ignore the prefix (or suffix) to see the meaningful part of the name. The more we read the code, the less we see the prefixes. Eventually the prefixes become unseen clutter and a marker of older code.

Interfaces and Implementations

These are sometimes a special case for encodings. For example, say you are building an ABSTRACT FACTORY 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

Readers shouldn't have to mentally translate your names into other names they already know. This problem generally arises from a choice to use neither problem domain terms nor solution domain terms.

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. This is because those single-letter names for loop counters are traditional. However, in most other contexts a single-letter name is a poor choice; it's just a place holder that the reader must mentally map to the actual concept. There can be no worse reason for using the name `c` than because `a` and `b` were already taken.

In general programmers are pretty smart people. Smart people sometimes like to show off their smarts by demonstrating their mental juggling abilities. After all, if you can reliably remember that `r` is the lower-cased version of the url with the host and scheme removed, then you must clearly be very smart.

One difference between a smart programmer and a professional programmer is that the professional understands that *clarity is king*. Professionals use their powers for good and write code that others can understand.

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.[4]

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

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.

---

4. http://java.sun.com/products/javabeans/docs/spec.html

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. Will they know what the function named `HolyHandGrenade` is supposed to do? Sure, it's cute, but maybe in this case `DeleteItems` might be a better name. Choose clarity over entertainment value.

Cuteness in code often appears in the form of colloquialisms or slang. For example, don't use the name `whack()` to mean `kill()`. Don't tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Say what you mean. Mean what you say.

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.

Modern editing environments like Eclipse and IntelliJ-provide context-sensitive clues, such as the list of methods you can call on a given object. But note that the list doesn't usually give you the comments you wrote around your function names and parameter lists. You are lucky if it gives the parameter *names* from function declarations. The function names have to stand alone, and they have to be consistent in order for you to pick the correct method without any additional exploration.

Likewise, it's confusing to have a `controller` and a `manager` and a `driver` in the same code base. What is the essential difference between a `DeviceManager` and a `ProtocolController`? Why are both not `controller`s or both not `manager`s? Are they both Drivers really? The name leads you to expect two objects that have very different type as well as having different classes.

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.

If you follow the "one word per concept" rule, you could end up with many classes that have, for example, an `add` method. As long as the parameter lists and return values of the various `add` methods are semantically equivalent, all is well.

However one might decide to use the word `add` for "consistency" when he or she is not in fact adding in the same sense. Let's say we have many classes where `add` will create a new value by adding or concatenating two existing values. Now let's say we are writing a new class that has a method that puts its single parameter into a collection. Should we call this method `add`? It might seem consistent because we have so many other `add` methods, but in this case, the semantics are different, so we should use a name like `insert` or `append` instead. To call the new method `add` would be a pun.

Our goal, as authors, is to make our code as easy as possible to understand. We want our code to be a quick skim, not an intense study. We want to use the popular paperback model whereby the author is responsible for making himself clear and not the academic model where it is the scholar's job to dig the meaning out of the paper.

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.

The name `AccountVisitor` means a great deal to a programmer who is familiar with the VISITOR pattern. What programmer would not know what a `JobQueue` was? There are lots of very technical things that programmers have to do. Choosing technical names for those things is usually the most appropriate course.

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. Instead, you need to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces. When all else fails, then prefixing the name may be necessary as a last resort.

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.

Consider the method in Listing 2-1. Do the variables need a more meaningful context? The function name provides only part of the context; the algorithm provides the rest. Once you read through the function, you see that the three variables, `number`, `verb`, and `pluralModifier`, are part of the "guess statistics" message. Unfortunately, the context must be inferred. When you first look at the method, the meanings of the variables are opaque.

Listing 2-1
`Variables with unclear context.`

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
  }
```

The function is a bit too long and the variables are used throughout. To split the function into smaller pieces we need to create a `GuessStatisticsMessage` class and make the three variables fields of this class. This provides a clear context for the three variables. They are *definitively* part of the `GuessStatisticsMessage`. The improvement of context also allows the algorithm to be made much cleaner by breaking it into many smaller functions. (See Listing 2-2.)

Listing 2-2
Variables have a context.

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
       verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Don't Add Gratuitous Context

In an imaginary application called "Gas Station Deluxe," it is a bad idea to prefix every class with `GSD`. Frankly, you are working against your tools. You type `G` and press the completion key and are rewarded with a mile-long list of every class in the system. Is that wise? Why make it hard for the IDE to help you?

Likewise, say you invented a `MailingAddress` class in `GSD`'s accounting module, and you named it `GSDAccountAddress`. Later, you need a mailing address for your customer contact application. Do you use `GSDAccountAddress`? Does it sound like the right name? Ten of 17 characters are redundant or irrelevant.

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

The names `accountAddress` and `customerAddress` are fine names for instances of the class `Address` but could be poor names for classes. `Address` is a fine name for a class. If I need to differentiate between MAC addresses, port addresses, and Web addresses, I might consider `PostalAddress`, `MAC`, and `URI`. The resulting names are more precise, which is the point of all naming.

Final Words

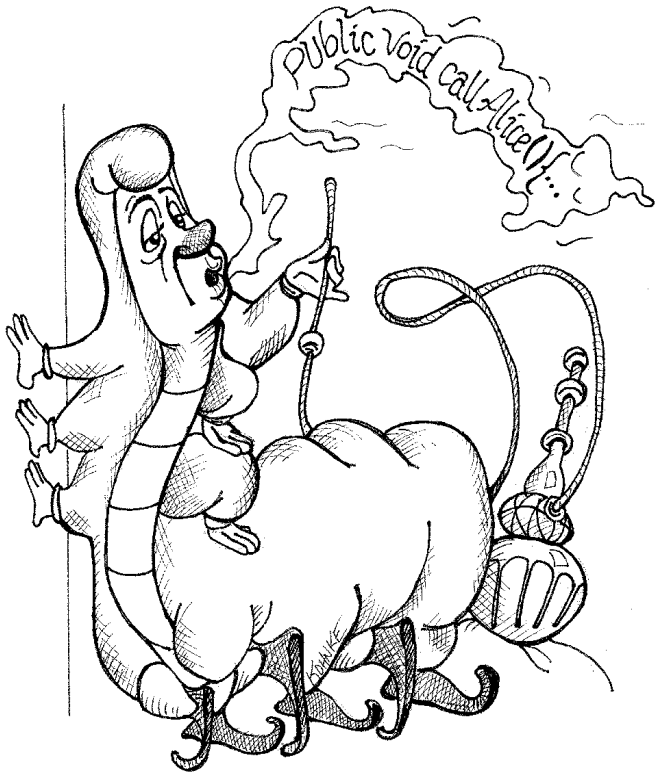
The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don't learn to do it very well.

People are also afraid of renaming things for fear that some other developers will object. We do not share that fear and find that we are actually grateful when names change (for the better). Most of the time we don't really memorize the names of classes and methods. We use the modern tools to deal with details like that so we can focus on whether the code reads like paragraphs and sentences, or at least like tables and data structure (a sentence isn't always the best way to display data). You will probably end up surprising someone when you rename, just like you might with any other code improvement. Don't let it stop you in your tracks.

Follow some of these rules and see whether you don't improve the readability of your code. If you are maintaining someone else's code, use refactoring tools to help resolve these problems. It will pay off in the short term and continue to pay in the long run.

3

Functions



In the early days of programming we composed our systems of routines and subroutines. Then, in the era of Fortran and PL/1 we composed our systems of programs, subprograms, and functions. Nowadays only the function survives from those early days. Functions are the first line of organization in any program. Writing them well is the topic of this chapter.

Consider the code in Listing 3-1. It's hard to find a long function in FitNesse,[1] but after a bit of searching I came across this one. Not only is it long, but it's got duplicated code, lots of odd strings, and many strange and inobvious data types and APIs. See how much sense you can make of it in the next three minutes.

Listing 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

1. An open-source testing tool. www.fitnese.org

Listing 3-1 (continued)
`HtmlUtil.java (FitNesse 20070619)`

```
      if (includeSuiteSetup) {
        WikiPage suiteTeardown =
          PageCrawlerImpl.getInheritedPage(
                  SuiteResponder.SUITE_TEARDOWN_NAME,
                  wikiPage
          );
        if (suiteTeardown != null) {
          WikiPagePath pagePath =
            suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
          String pagePathName = PathParser.render(pagePath);
          buffer.append("!include -teardown .")
                .append(pagePathName)
                .append("\n");
        }
      }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
  }
```

Do you understand the function after three minutes of study? Probably not. There's too much going on in there at too many different levels of abstraction. There are strange strings and odd function calls mixed in with doubly nested `if` statements controlled by flags.

However, with just a few simple method extractions, some renaming, and a little restructuring, I was able to capture the intent of the function in the nine lines of Listing 3-2. See whether you can understand *that* in the next 3 minutes.

Listing 3-2
`HtmlUtil.java (refactored)`

```
  public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
  ) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
      WikiPage testPage = pageData.getWikiPage();
      StringBuffer newPageContent = new StringBuffer();
      includeSetupPages(testPage, newPageContent, isSuite);
      newPageContent.append(pageData.getContent());
      includeTeardownPages(testPage, newPageContent, isSuite);
      pageData.setContent(newPageContent.toString());
    }

    return pageData.getHtml();
  }
```

Unless you are a student of FitNesse, you probably don't understand all the details. Still, you probably understand that this function performs the inclusion of some setup and teardown pages into a test page and then renders that page into HTML. If you are familiar with JUnit,[2] you probably realize that this function belongs to some kind of Web-based testing framework. And, of course, that is correct. Divining that information from Listing 3-2 is pretty easy, but it's pretty well obscured by Listing 3-1.

So what is it that makes a function like Listing 3-2 easy to read and understand? How can we make a function communicate its intent? What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?

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*. This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.

In the eighties we used to say that a function should be no bigger than a screen-full. Of course we said that at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. Nowadays with a cranked-down font and a nice big monitor, you can fit 150 characters on a line and a 100 lines or more on a screen. Lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.

How short should a function be? In 1999 I went to visit Kent Beck at his home in Oregon. We sat down and did some programming together. At one point he showed me a cute little Java/Swing program that he called *Sparkle*. It produced a visual effect on the screen very similar to the magic wand of the fairy godmother in the movie Cinderella. As you moved the mouse, the sparkles would drip from the cursor with a satisfying scintillation, falling to the bottom of the window through a simulated gravitational field. When Kent showed me the code, I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in *this* program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. *That's* how short your functions should be![3]

---

2. An open-source unit-testing tool for Java. www.junit.org
3. I asked Kent whether he still had a copy, but he was unable to find one. I searched all my old computers too, but to no avail. All that is left now is my memory of that program.

How short should your functions be? They should usually be shorter than Listing 3-2! Indeed, Listing 3-2 should really be shortened to Listing 3-3.

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

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

It should be very clear that Listing 3-1 is doing lots more than one thing. It's creating buffers, fetching pages, searching for inherited pages, rendering paths, appending arcane strings, and generating HTML, among other things. Listing 3-1 is very busy doing lots of different things. On the other hand, Listing 3-3 is doing one simple thing. It's including setups and teardowns into test pages.

The following advice has appeared in one form or another for 30 years or more.

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

The problem with this statement is that it is hard to know what "one thing" is. Does Listing 3-3 do one thing? It's easy to make the case that it's doing three things:

1. Determining whether the page is a test page.
2. If so, including setups and teardowns.
3. Rendering the page in HTML.

So which is it? Is the function doing one thing or three things? Notice that the three steps of the function are one level of abstraction below the stated name of the function. We can describe the function by describing it as a brief *TO*[4] paragraph:

> *TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.*

If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. After all, the reason we write functions is to decompose a larger concept (in other words, the name of the function) into a set of steps at the next level of abstraction.

It should be very clear that Listing 3-1 contains steps at many different levels of abstraction. So it is clearly doing more than one thing. Even Listing 3-2 has two levels of abstraction, as proved by our ability to shrink it down. But it would be very hard to meaningfully shrink Listing 3-3. We could extract the `if` statement into a function named `includeSetupsAndTeardownsIfTestPage`, but that simply restates the code without changing the level of abstraction.

So, another way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation [G34].

Sections within Functions

Look at Listing 4-7 on 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 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")`.

Mixing levels of abstraction within a function is always confusing. Readers may not be able to tell whether a particular expression is an essential concept or a detail. Worse,

4. The LOGO language used the keyword "TO" in the same way that Ruby and Python use "def." So every function began with the word "TO." This had an interesting effect on the way functions were designed.

like broken windows, once details are mixed with essential concepts, more and more details tend to accrete within the function.

Reading Code from Top to Bottom: *The Stepdown Rule*

We want the code to read like a top-down narrative.[5] We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions. I call this *The Stepdown Rule*.

To say this differently, we want to be able to read the program as though it were a set of *TO* paragraphs, each of which is describing the current level of abstraction and referencing subsequent *TO* paragraphs at the next level down.

> *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
>
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
>
> *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
>
> *To search the parent. . .*

It turns out to be very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing." Making the code read like a top-down set of *TO* paragraphs is an effective technique for keeping the abstraction level consistent.

Take a look at Listing 3-7 at the end of this chapter. It shows the whole `testableHtml` function refactored according to the principles described here. Notice how each function introduces the next, and each function remains at a consistent level of abstraction.

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.

---

5. [KP78], p. 37.
6. And, of course, I include if/else chains in this.

Consider Listing 3-4. It shows just one of the operations that might depend on the type of employee.

Listing 3-4
`Payroll.java`

```
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);
    }
  }
```

There are several problems with this function. First, it's large, and when new employee types are added, it will grow. Second, it very clearly does more than one thing. Third, it violates the Single Responsibility Principle[7] (SRP) because there is more than one reason for it to change. Fourth, it violates the Open Closed Principle[8] (OCP) because it must change whenever new types are added. But possibly the worst problem with this function is that there are an unlimited number of other functions that will have the same structure. For example we could have

```
isPayday(Employee e, Date date),
```

or

```
deliverPay(Employee e, Money pay),
```

or a host of others. All of which would have the same deleterious structure.

The solution to this problem (see Listing 3-5) is to bury the `switch` statement in the basement of an ABSTRACT FACTORY,[9] and never let anyone see it. The factory will use the `switch` statement to create appropriate instances of the derivatives of `Employee`, and the various functions, such as `calculatePay`, `isPayday`, and `deliverPay`, will be dispatched polymorphically through the `Employee` interface.

My general rule for `switch` statements is that they can be tolerated if they appear only once, are used to create polymorphic objects, and are hidden behind an inheritance

7. a. http://en.wikipedia.org/wiki/Single_responsibility_principle
   b. http://www.objectmentor.com/resources/articles/srp.pdf
8. a. http://en.wikipedia.org/wiki/Open/closed_principle
   b. http://www.objectmentor.com/resources/articles/ocp.pdf
9. [GOF].

Listing 3-5
Employee and Factory

```
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);
    }
  }
}
```

relationship so that the rest of the system can't see them [G23]. Of course every circumstance is unique, and there are times when I violate one or more parts of that rule.

Use Descriptive Names

In Listing 3-7 I changed the name of our example function from `testableHtml` to `SetupTeardownIncluder.render`. This is a far better name because it better describes what the function does. I also gave each of the private methods an equally descriptive name such as `isTestable` or `includeSetupAndTeardownPages`. It is hard to overestimate the value of good names. Remember Ward's principle: "*You know you are working on clean code when each routine turns out to be pretty much what you expected*." Half the battle to achieving that principle is choosing good names for small functions that do one thing. The smaller and more focused a function is, the easier it is to choose a descriptive name.

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. Modern IDEs like Eclipse or IntelliJ make it trivial to change names. Use one of those IDEs and experiment with different names until you find one that is as descriptive as you can make it.

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 `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage`, and `includeSetupPage`. The similar phraseology in those names allows the sequence to tell a story. Indeed, if I showed you just the sequence above, you'd ask yourself: "What happened to `includeTeardownPages`, `includeSuiteTeardownPage`, and `includeTeardownPage`?" How's that for being ". . . *pretty much what you expected*."

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.

Arguments are hard. They take a lot of conceptual power. That's why I got rid of almost all of them from the example. Consider, for instance, the `StringBuffer` in the example. We could have passed it around as an argument rather than making it an instance variable, but then our readers would have had to interpret it each time they saw it. When you are reading the story told by the module, `includeSetupPage()` is easier to understand than `includeSetupPageInto(newPageContent)`. The argument is at a different level of abstraction than the function name and forces you to know a detail (in other words, `StringBuffer`) that isn't particularly important at that point.

Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial. If there's one argument, it's not too hard. With two arguments the problem gets a bit more challenging. With more than two arguments, testing every combination of appropriate values can be daunting.

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. `SetupTeardownIncluder.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, `InputStream fileOpen("MyFile")` transforms a file name `String` into an `InputStream` return value. These two uses are what readers expect when they see a function. You should choose names that make the distinction clear, and always use the two forms in a consistent context. (See Command Query Separation below.)

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. The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system, 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.

Try to avoid any monadic functions that don't follow these forms, for example, `void includeSetupPageInto(StringBuffer pageText)`. 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!

In Listing 3-7 we had no choice because the callers were already passing that flag in, and I wanted to limit the scope of refactoring to the function and below. Still, the method call `render(true)` is just plain confusing to a poor reader. Mousing over the call and seeing `render(boolean isSuite)` helps a little, but not that much. We should have split the function into two: `renderForSuite()` and `renderForSingleTest()`.

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)`.[10] Though the meaning of both is clear, the first glides past the eye, easily depositing its meaning. The second requires a short pause until we learn to ignore the first parameter. And *that*, of course, eventually results in problems because we should never ignore any part of code. The parts we ignore are where the bugs will hide.

There are times, of course, where two arguments are appropriate. For example, `Point p = new Point(0,0);` is perfectly reasonable. Cartesian points naturally take two arguments. Indeed, we'd be very surprised to see `new Point(0)`. However, the two arguments in this case *are ordered components of a single value!* Whereas `output-Stream` and `name` have neither a natural cohesion, nor a natural ordering.

Even obvious dyadic functions like `assertEquals(expected, actual)` are problematic. How many times have you put the `actual` where the `expected` should be? The two arguments have no natural ordering. The `expected, actual` ordering is a convention that requires practice to learn.

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 mechanims may be available to you to convert them into monads. For example, you might make the `writeField` method a member of `outputStream` so that you can say `outputStream.writeField(name)`. Or you might make the `outputStream` a member variable of the current class so that you don't have to pass it. Or you might extract a new class like `FieldWriter` that takes the `outputStream` in its constructor and has a `write` method.

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.

For example, consider the common overload of `assertEquals` that takes three arguments: `assertEquals(message, expected, actual)`. How many times have you read the `message` and thought it was the `expected`? I have stumbled and paused over that particular triad many times. In fact, *every time I see it,* I do a double-take and then learn to ignore the message.

On the other hand, here is a triad that is not quite so insidious: `assertEquals(1.0, amount, .001)`. Although this still requires a double-take, it's one that's worth taking. It's always good to be reminded that equality of floating point values is a relative thing.

---

10. I just finished refactoring a module that used the dyadic form. I was able to make the `outputStream` a field of the class and convert all the `writeField` calls to the monadic form. The result was much cleaner.

Argument Objects

When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own. Consider, for example, the difference between the two following declarations:

```
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)
```

So all the same rules apply. Functions that take variable arguments can be monads, dyads, or even triads. But it would be a mistake to give them more arguments than that.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... 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."

This last is an example of the *keyword* form of a function name. Using this form we encode the names of the arguments into the function name. For example, `assertEquals` might be better written as `assertExpectedEqualsActual(expected, actual)`. This strongly mitigates the problem of having to remember the ordering of the arguments.

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. 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

Arguments are most naturally interpreted as *inputs* to a function. If you have been programming for more than a few years, I'm sure you've done a double-take on an argument that was actually an *output* rather than an input. 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.

In the days before object oriented programming it was sometimes necessary to have output arguments. However, much of the need for output arguments disappears in OO languages because `this` is *intended* to act as an output argument. In other words, it would be better for `appendFooter` to be invoked as

```
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

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion. Consider, for example, the following function:

```
public boolean set(String attribute, String value);
```

This function sets the value of a named attribute and returns `true` if it is successful and `false` if no such attribute exists. 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 author intended `set` to be a verb, but in the context of the `if` statement it *feels* like an adjective. So the statement reads as "If the `username` attribute was previously set to `unclebob`" and not "set the `username` attribute to `unclebob` and if that worked then. . . ." We

could try to resolve this by renaming the `set` function to `setAndCheckIfExists`, but that doesn't much help the readability of the `if` statement. 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.

```
if (deletePage(page) == E_OK)
```

This does not suffer from verb/adjective confusion but does lead to deeply nested structures. 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());
}
```

In the above, the `delete` function is all about error processing. It is easy to understand and then ignore. The `deletePageAndAllReferences` function is all about the processes of fully deleting a `page`. Error handling can be ignored. This provides a nice separation that makes the code easier to understand and modify.

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. This implies (as in the example above) that if the keyword `try` exists in a function, it should be the very first word in the function and that there should be nothing after the `catch/finally` blocks.

The `Error.java` Dependency Magnet

Returning error codes usually implies that there is some class or enum in which all the error codes are defined.

```
public enum Error {
  OK,
  INVALID,
  NO_SUCH,
  LOCKED,
  OUT_OF_RESOURCES,
  WAITING_FOR_EVENT;
}
```

Classes like this are a *dependency magnet;* many other classes must import and use them. Thus, when the `Error enum` changes, all those other classes need to be recompiled and redeployed.[11] This puts a negative pressure on the `Error` class. Programmers don't want

11. Those who felt that they could get away without recompiling and redeploying have been found—and dealt with.

to add new errors because then they have to rebuild and redeploy everything. So they reuse old error codes instead of adding new ones.

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.[12]

Don't Repeat Yourself[13]

Look back at Listing 3-1 carefully and you will notice that there is an algorithm that gets repeated four times, once for each of the `SetUp`, `SuiteSetUp`, `TearDown`, and `SuiteTearDown` cases. It's not easy to spot this duplication because the four instances are intermixed with other code and aren't uniformly duplicated. Still, the duplication is a problem because it bloats the code and will require four-fold modification should the algorithm ever have to change. It is also a four-fold opportunity for an error of omission.

This duplication was remedied by the `include` method in Listing 3-7. Read through that code again and notice how the readability of the whole module is enhanced by the reduction of that duplication.

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. Consider, for example, that all of Codd's database normal forms serve to eliminate duplication in data. Consider also how object-oriented programming serves to concentrate code into base classes that would otherwise be redundant. Structured programming, Aspect Oriented Programming, Component Oriented Programming, are all, in part, strategies for eliminating duplication. It would appear that since the invention of the subroutine, innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

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 `continue` statements in a loop, and never, *ever,* any `goto` statements.

---

12. This is an example of the Open Closed Principle (OCP) [PPP02].
13. The DRY principle. [PRAG].
14. [SP72].

While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit.

So if you keep your functions small, then the occasional multiple `return`, `break`, or `continue` statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule. On the other hand, `goto` only makes sense in large functions, so it should be avoided.

How Do You Write Functions Like This?

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code.

So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing.

In the end, I wind up with functions that follow the rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could.

Conclusion

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. This is not some throwback to the hideous old notion that the nouns and verbs in a requirements document are the first guess of the classes and functions of a system. Rather, this is a much older truth. The art of programming is, and has always been, the art of language design.

Master programmers think of systems as stories to be told rather than programs to be written. They use the facilities of their chosen programming language to construct a much richer and more expressive language that can be used to tell that story. Part of that domain-specific language is the hierarchy of functions that describe all the actions that take place within that system. In an artful act of recursion those actions are written to use the very domain-specific language they define to tell their own small part of the story.

This chapter has been about the mechanics of writing functions well. If you follow the rules herein, your functions will be short, well named, and nicely organized. But

never forget that your real goal is to tell the story of the system, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.

SetupTeardownIncluder

Listing 3-7
SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }
```

Listing 3-7 (continued)
SetupTeardownIncluder.java

```
  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }

  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }

  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }

  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }

  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }

  private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }

  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
      .append("\n!include ")
```

Listing 3-7 (continued)
`SetupTeardownIncluder.java`

```
        .append(arg)
        .append(" .")
        .append(pagePathName)
        .append("\n");
    }
}
```

Bibliography

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

[PPP02]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

4

Comments

"Don't comment bad code—rewrite it."

—Brian W. Kernighan and P. J. Plaugher[1]

Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module more than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.

Comments are not like Schindler's List. They are not "pure good." Indeed, comments are, at best, a necessary evil. If our programming languages were expressive enough, or if

1. [KP78], p. 144.

we had the talent to subtly wield those languages to express our intent, we would not need comments very much—perhaps not at all.

The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word *failure*. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration.

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.

Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often. The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. The reason is simple. Programmers can't realistically maintain them.

Code changes and evolves. Chunks of it move from here to there. Those chunks bifurcate and reproduce and come together again to form chimeras. Unfortunately the comments don't always follow them—*can't* always follow them. And all too often the comments get separated from the code they describe and become orphaned blurbs of ever-decreasing accuracy. For example, look what has happened to this comment and the line it was intended to describe:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
  "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
  "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Other instance variables that were probably added later were interposed between the `HTTP_DATE_REGEXP` constant and it's explanatory comment.

It is possible to make the point that programmers should be disciplined enough to keep the comments in a high state of repair, relevance, and accuracy. I agree, they should. But I would rather that energy go toward making the code so clear and expressive that it does not need the comments in the first place.

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.

Truth can only be found in one place: the code. Only the code can truly tell you what it does. It is the only source of truly accurate information. Therefore, though comments are sometimes necessary, we will expend significant energy to minimize them.

Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code. We write a module and we know it is confusing and disorganized. We know it's a mess. So we say to ourselves, "Ooh, I'd better comment that!" No! You'd better clean it!

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

There are certainly times when code makes a poor vehicle for explanation. Unfortunately, many programmers have taken this to mean that code is seldom, if ever, a good means for explanation. This is patently false. 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

Some comments are necessary or beneficial. We'll look at a few that I consider worthy of the bits they consume. Keep in mind, however, that the only truly good comment is the comment you found a way not to write.

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.

Here, for example, is the standard comment header that we put at the beginning of every source file in FitNesse. I am happy to say that our IDE hides this comment from acting as clutter by automatically collapsing it.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Comments like this should not be contracts or legal tomes. Where possible, refer to a standard license or other external document rather than putting all the terms and conditions into the comment.

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:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

A comment like this can sometimes be useful, but it is better to use the name of the function to convey the information where possible. For example, in this case the comment could be made redundant by renaming the function: `responderBeingTested`.

Here's a case that's a bit better:

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

In this case the comment lets us know that the regular expression is intended to match a time and date that were formatted with the `SimpleDateFormat.format` function using the specified format string. Still, it might have been better, and clearer, if this code had been moved to a special class that converted the formats of dates and times. Then the comment would likely have been superfluous.

Explanation of Intent

Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. In the following case we see an interesting decision documented by a comment. When comparing two objects, the author decided that he wanted to sort objects of his class higher than objects of any other.

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // we are greater because we are the right type.
}
```

Here's an even better example. 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. In general it is better to find a way to make that argument or return value clear in its own right; but when its part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.

```
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
}
```

There is a substantial risk, of course, that a clarifying comment is incorrect. Go through the previous example and see how difficult it is to verify that they are correct. This explains both why the clarification is necessary and why it's risky. So before writing comments like this, take care that there is no better way, and then take even more care that they are accurate.

Warning of Consequences

Sometimes it is useful to warn other programmers about certain consequences. For example, here is a comment that explains why a particular test case is turned off:

```
// 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);
}
```

Nowadays, of course, we'd turn off the test case by using the `@Ignore` attribute with an appropriate explanatory string. `@Ignore("Takes too long to run")`. But back in the days before JUnit 4, putting an underscore in front of the method name was a common convention. The comment, while flippant, makes the point pretty well.

Here's another, more poignant example:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
  //SimpleDateFormat is not thread safe,
  //so we need to create each instance independently.
  SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM  yyyy HH:mm:ss z");
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

You might complain that there are better ways to solve this problem. I might agree with you. But the comment, as given here, is perfectly reasonable. It will prevent some overly eager programmer from using a static initializer in the name of efficiency.

TODO Comments

It is sometimes reasonable to leave "To do" notes in the form of `//TODO` comments. In the following case, the `TODO` comment explains why the function has a degenerate implementation and what that function's future should be.

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
  return null;
}
```

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.

Nowadays, most good IDEs provide special gestures and features to locate all the TODO comments, so it's not likely that they will get lost. Still, you don't want your code to be littered with TODOs. So scan through them regularly and eliminate the ones you can.

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 javadocs for the standard Java library are a case in point. It would be difficult, at best, to write Java programs without them.

If you are writing a public API, then you should certainly write good javadocs for it. But keep in mind the rest of the advice in this chapter. Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment.

Bad Comments

Most comments fall into this category. Usually they are crutches or excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to himself.

Mumbling

Plopping in a comment just because you feel you should or because the process requires it, is a hack. If you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write.

Here, for example, is a case I found in FitNesse, where a comment might indeed have been useful. But the author was in a hurry or just not paying much attention. His mumbling left behind an enigma:

```
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
  }
}
```

What does that comment in the `catch` block mean? Clearly it meant something to the author, but the meaning does not come through all that well. Apparently, if we get an `IOException`, it means that there was no properties file; and in that case all the defaults are loaded. But who loads all the defaults? Were they loaded before the call to `loadProperties.load`? Or did `loadProperties.load` catch the exception, load the defaults, and then pass the exception on for us to ignore? Or did `loadProperties.load` load all the defaults before attempting to load the file? Was the author trying to comfort himself about the fact that he was leaving the `catch` block empty? Or—and this is the scary possibility—was the author trying to tell himself to come back here later and write the code that would load the defaults?

Our only recourse is to examine the code in other parts of the system to find out what's going on. Any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you and is not worth the bits it consumes.

Redundant Comments

Listing 4-1 shows a simple function with a header comment that is completely redundant. The comment probably takes longer to read than the code itself.

Listing 4-1
`waitForClose`

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

What purpose does this comment serve? It's certainly not more informative than the code. It does not justify the code, or provide intent or rationale. It is not easier to read than the code. Indeed, it is less precise than the code and entices the reader to accept that lack of precision in lieu of true understanding. It is rather like a gladhanding used-car salesman assuring you that you don't need to look under the hood.

Now consider the legion of useless and redundant javadocs in Listing 4-2 taken from Tomcat. These comments serve only to clutter and obscure the code. They serve no documentary purpose at all. To make matters worse, I only showed you the first few. There are many more in this module.

Listing 4-2
`ContainerBase.java (Tomcat)`

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * The processor delay for this component.
   */
  protected int backgroundProcessorDelay = -1;


  /**
   * The lifecycle event support for this component.
   */
  protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);


  /**
   * The container event listeners for this Container.
   */
  protected ArrayList listeners = new ArrayList();


  /**
   * The Loader implementation with which this Container is
   * associated.
   */
  protected Loader loader = null;


  /**
   * The Logger implementation with which this Container is
   * associated.
   */
  protected Log logger = null;


  /**
   * Associated logger name.
   */
  protected String logName = null;
```

Listing 4-2 (continued)
ContainerBase.java (Tomcat)

```
    /**
     * The Manager implementation with which this Container is
     * associated.
     */
    protected Manager manager = null;


    /**
     * The cluster with which this Container is associated.
     */
    protected Cluster cluster = null;


    /**
     * The human-readable name of this Container.
     */
    protected String name = null;


    /**
     * The parent Container to which this Container is a child.
     */
    protected Container parent = null;


    /**
     * The parent class loader to be configured when we install a
     * Loader.
     */
    protected ClassLoader parentClassLoader = null;


    /**
     * The Pipeline object with which this Container is
     * associated.
     */
    protected Pipeline pipeline = new StandardPipeline(this);


    /**
     * The Realm with which this Container is associated.
     */
    protected Realm realm = null;


    /**
     * The resources DirContext object with which this Container
     * is associated.
     */
    protected DirContext resources = null;
```

Misleading Comments

Sometimes, with all the best intentions, a programmer makes a statement in his comments that isn't precise enough to be accurate. Consider for another moment the badly redundant but also subtly misleading comment we saw in Listing 4-1.

Did you discover how the comment was misleading? The method does not return *when* `this.closed` becomes `true`. It returns *if* `this.closed` is `true`; otherwise, it waits for a blind time-out and then throws an exception *if* `this.closed` is still not `true`.

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. Comments like this just clutter up the code, propagate lies, and lend to general confusion and disorganization.

For example, required javadocs for every function lead to abominations such as Listing 4-3. This clutter adds nothing and serves only to obfuscate the code and create the potential for lies and misdirection.

Listing 4-3

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

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.

```
 * Changes (from 11-Oct-2001)
 * --------------------------
 * 11-Oct-2001 : Re-organised the class and moved it to new package
 *               com.jrefinery.date (DG);
 * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
 *               class (DG);
 * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
 *               class is gone (DG);  Changed getPreviousDayOfWeek(),
 *               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
 *               bugs (DG);
 * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
 * 29-May-2002 : Moved the month constants into a separate interface
 *               (MonthConstants) (DG);
 * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 29-May-2003 : Fixed bug in addMonths method (DG);
 * 04-Sep-2003 : Implemented Comparable.  Updated the isInRange javadocs (DG);
 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Long ago there was a good reason to create and maintain these log entries at the start of every module. We didn't have source code control systems that did it for us. Nowadays, however, these long journals are just more clutter to obfuscate the module. They should be completely removed.

Noise Comments

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

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

No, *really?* Or how about this:

```
/** The day of the month. */
    private int dayOfMonth;
```

And then there's this paragon of redundancy:

```
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

These comments are so noisy that we learn to ignore them. As we read through code, our eyes simply skip over them. Eventually the comments begin to lie as the code around them changes.

The first comment in Listing 4-4 seems appropriate.[2] It explains why the `catch` block is being ignored. But the second comment is pure noise. Apparently the programmer was just so frustrated with writing `try`/`catch` blocks in this function that he needed to vent.

Listing 4-4
`startSending`

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // normal. someone stopped the request.
  }
  catch(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
      //Give me a break!
    }
  }
}
```

Rather than venting in a worthless and noisy comment, the programmer should have recognized that his frustration could be resolved by improving the structure of his code. He should have redirected his energy to extracting that last `try`/`catch` block into a separate function, as shown in Listing 4-5.

Listing 4-5
`startSending (refactored)`

```
private void startSending()
{
  try
  {
    doSending();
  }
```

2. The current trend for IDEs to check spelling in comments will be a balm for those of us who read a lot of code.

Listing 4-5 (continued)
`startSending (refactored)`

```
      catch(SocketException e)
      {
        // normal. someone stopped the request.
      }
      catch(Exception e)
      {
        addExceptionAndCloseResponse(e);
      }
    }

    private void addExceptionAndCloseResponse(Exception e)
    {
      try
      {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
      }
      catch(Exception e1)
      {
      }
    }
```

Replace the temptation to create noise with the determination to clean your code. You'll find it makes you a better and happier programmer.

Scary Noise

Javadocs can also be noisy. What purpose do the following Javadocs (from a well-known open-source library) serve? Answer: nothing. They are just redundant noisy comments written out of some misplaced desire to provide documentation.

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

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;

/** The version. */
private String info;
```

Read these comments again more carefully. Do you see the cut-paste error? If authors aren't paying attention when comments are written (or pasted), why should readers be expected to profit from them?

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))
```

The author of the original code may have written the comment first (unlikely) and then written the code to fulfill the comment. However, the author should then have refactored the code, as I did, so that the comment could be removed.

Position Markers

Sometimes programmers like to mark a particular position in a source file. For example, I recently found this in a program I was looking through:

```
// Actions ///////////////////////////////
```

There are rare times when it makes sense to gather certain functions together beneath a banner like this. But in general they are clutter that should be eliminated—especially the noisy train of slashes at the end.

Think of it this way. 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

Sometimes programmers will put special comments on closing braces, as in Listing 4-6. Although this might make sense for long functions with deeply nested structures, it serves only to clutter the kind of small and encapsulated functions that we prefer. So if you find yourself wanting to mark your closing braces, try to shorten your functions instead.

Listing 4-6
`wc.java`

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
```

Listing 4-6 (continued)
wc.java

```
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
      } //while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
      System.err.println("Error:" + e.getMessage());
    } //catch
  } //main
}
```

Attributions and Bylines

```
/* Added by Rick */
```

Source code control systems are very good at remembering who added what, when. There is no need to pollute the code with little bylines. 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.

Again, the source code control system is a better place for this kind of information.

Commented-Out Code

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

```
    InputStreamResponse response = new InputStreamResponse();
    response.setBody(formatter.getResultStream(), formatter.getByteCount());
//  InputStream resultsStream = formatter.getResultStream();
//  StreamReader reader = new StreamReader(resultsStream);
//  response.setContent(reader.read(formatter.getByteCount()));
```

Others who see that commented-out code won't have the courage to delete it. They'll think it is there for a reason and is too important to delete. So commented-out code gathers like dregs at the bottom of a bad bottle of wine.

Consider this from apache commons:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
```

```
    else {
        this.pngBytes = null;
    }
    return this.pngBytes;
```

Why are those two lines of code commented? Are they important? Were they left as reminders for some imminent change? Or are they just cruft that someone commented-out years ago and has simply not bothered to clean up.

There was a time, back in the sixties, when commenting-out code might have been useful. But we've had good source code control systems for a very long time now. Those systems will remember the code for us. We don't have to comment it out any more. Just delete the code. We won't lose it. Promise.

HTML Comments

HTML in source code comments is an abomination, as you can tell by reading the code below. It makes the comments hard to read in the one place where they should be easy to read—the editor/IDE. If comments are going to be extracted by some tool (like Javadoc) to appear in a Web page, then it should be the responsibility of that tool, and not the programmer, to adorn the comments with appropriate HTML.

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 *     classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 *             resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 *     suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 *     fitnesseport=&quot;8082&quot;
 *     resultsdir=&quot;${results.dir}&quot;
 *     resultshtmlpage=&quot;fit-results.html&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

Nonlocal Information

If you must write a comment, then make sure it describes the code it appears near. Don't offer systemwide information in the context of a local comment. Consider, for example, the javadoc comment below. Aside from the fact that it is horribly redundant, it also offers information about the default port. And yet the function has absolutely no control over what that default is. The comment is not describing the function, but some other, far distant part of the system. Of course there is no guarantee that this comment will be changed when the code containing the default is changed.

```
/**
 * Port on which fitnesse would run. Defaults to <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
  this.fitnessePort = fitnessePort;
}
```

Too Much Information

Don't put interesting historical discussions or irrelevant descriptions of details into your comments. The comment below was extracted from a module designed to test that a function could encode and decode base64. Other than the RFC number, someone reading this code has no need for the arcane information contained in the comment.

```
/*
   RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
   Part One: Format of Internet Message Bodies
   section 6.8.  Base64 Content-Transfer-Encoding
   The encoding process represents 24-bit groups of input bits as output
   strings of 4 encoded characters. Proceeding from left to right, a
   24-bit input group is formed by concatenating 3 8-bit input groups.
   These 24 bits are then treated as 4 concatenated 6-bit groups, each
   of which is translated into a single digit in the base64 alphabet.
   When encoding a bit stream via the base64 encoding, the bit stream
   must be presumed to be ordered with the most-significant-bit first.
   That is, the first bit in the stream will be the high-order bit in
   the first 8-bit byte, and the eighth bit will be the low-order bit in
   the first 8-bit byte, and so on.
 */
```

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.

Consider, for example, this comment drawn from apache commons:

```
/*
 * start with an array that is big enough to hold all the pixels
 * (plus filter bytes), and an extra 200 bytes for header info
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

What is a filter byte? Does it relate to the +1? Or to the *3? Both? Is a pixel a byte? Why 200? The purpose of a comment is to explain code that does not explain itself. It is a pity when a comment needs its own explanation.

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

As useful as javadocs are for public APIs, they are anathema to code that is not intended for public consumption. 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.

Example

I wrote the module in Listing 4-7 for the first *XP Immersion*. It was intended to be an example of bad coding and commenting style. Kent Beck then refactored this code into a much more pleasant form in front of several dozen enthusiastic students. Later I adapted the example for my book *Agile Software Development, Principles, Patterns, and Practices* and the first of my *Craftsman* articles published in *Software Development* magazine.

What I find fascinating about this module is that there was a time when many of us would have considered it "well documented." Now we see it as a small mess. See how many different comment problems you can find.

Listing 4-7
`GeneratePrimes.java`

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria.  The first man to calculate the
 * circumference of the Earth.  Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple.  Given an array of integers
 * starting at 2.  Cross out all multiples of 2.  Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;

public class GeneratePrimes
{
  /**
   * @param maxValue is the generation limit.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;
```

Listing 4-7 (continued)
GeneratePrimes.java

```
      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // get rid of known non-primes
      f[0] = f[1] = false;

      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
        }
      }

      // how many primes are there?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // bump count.
      }

      int[] primes = new int[count];

      // move the primes into the result
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // if prime
          primes[j++] = i;
      }

      return primes;  // return the primes
    }
    else // maxValue < 2
      return new int[0]; // return null array if bad input.
  }
}
```

In Listing 4-8 you can see a refactored version of the same module. Note that the use of comments is significantly restrained. There are just two comments in the whole module. Both comments are explanatory in nature.

Listing 4-8
PrimeGenerator.java (refactored)

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
```

Listing 4-8 (continued)
PrimeGenerator.java (refactored)

```
 * multiples.  Repeat until there are no more multiples
 * in the array.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size,
    // so we don't have to cross out multiples of numbers
    // larger than that root.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
  }
```

Listing 4-8 (continued)
PrimeGenerator.java (refactored)

```
  private static boolean notCrossed(int i)
  {
    return crossedOut[i] == false;
  }

  private static void putUncrossedIntegersIntoResult()
  {
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        result[j++] = i;
  }

  private static int numberOfUncrossedIntegers()
  {
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;

    return count;
  }
}
```

It is easy to argue that the first comment is redundant because it reads very much like the `generatePrimes` function itself. Still, I think the comment serves to ease the reader into the algorithm, so I'm inclined to leave it.

The second argument is almost certainly necessary. It explains the rationale behind the use of the square root as the loop limit. I could find no simple variable name, nor any different coding structure that made this point clear. On the other hand, the use of the square root might be a conceit. Am I really saving that much time by limiting the iteration to the square root? Could the calculation of the square root take more time than I'm saving?

It's worth thinking about. Using the square root as the iteration limit satisfies the old C and assembly language hacker in me, but I'm not convinced it's worth the time and effort that everyone else will expend to understand it.

Bibliography

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw-Hill, 1978.

5

Formatting

When people look under the hood, we want them to be impressed with the neatness, consistency, and attention to detail that they perceive. We want them to be struck by the orderliness. We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project.

You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you.

The Purpose of Formatting

First of all, let's be clear. Code formatting is *important*. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer's first order of business.

Perhaps you thought that "getting it working" was the first order of business for a professional developer. I hope by now, however, that this book has disabused you of that idea. The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made. The coding style and readability set precedents that continue to affect maintainability and extensibility long after the original code has been changed beyond recognition. Your style and discipline survives, even though your code does not.

So what are the formatting issues that help us to communicate best?

Vertical Formatting

Let's start with vertical size. How big should a source file be? In Java, file size is closely related to class size. We'll talk about class size when we talk about classes. For the moment let's just consider file size.

How big are most Java source files? It turns out that there is a huge range of sizes and some remarkable differences in style. Figure 5-1 shows some of those differences.

Seven different projects are depicted. Junit, FitNesse, testNG, Time and Money, JDepend, Ant, and Tomcat. The lines through the boxes show the minimum and maximum file lengths in each project. The box shows approximately one-third (one standard deviation[1]) of the files. The middle of the box is the mean. So the average file size in the FitNesse project is about 65 lines, and about one-third of the files are between 40 and 100+ lines. The largest file in FitNesse is about 400 lines and the smallest is 6 lines. Note that this is a log scale, so the small difference in vertical position implies a very large difference in absolute size.

1. The box shows sigma/2 above and below the mean. Yes, I know that the file length distribution is not normal, and so the standard deviation is not mathematically precise. But we're not trying for precision here. We're just trying to get a feel.

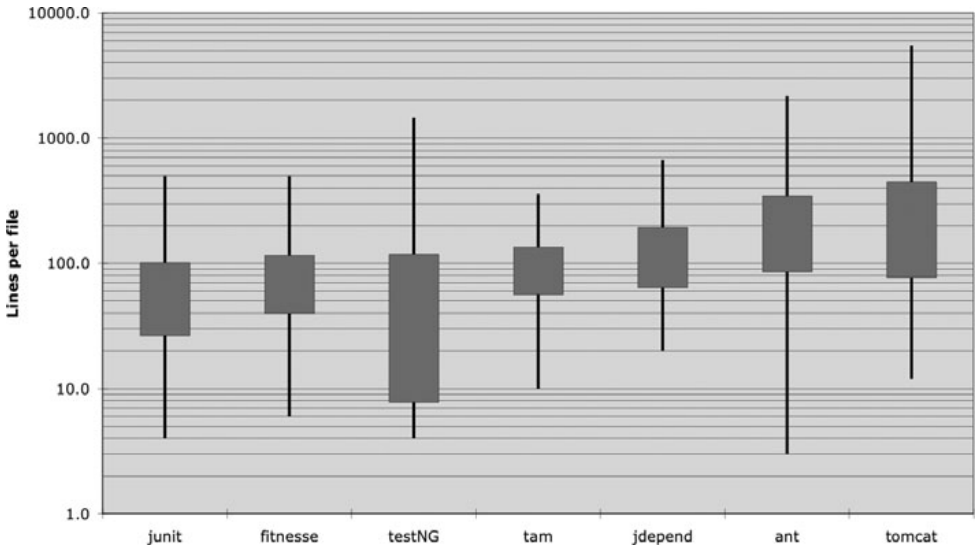


Figure 5-1
File length distributions LOG scale (box height = sigma)

Junit, FitNesse, and Time and Money are composed of relatively small files. None are over 500 lines and most of those files are less than 200 lines. Tomcat and Ant, on the other hand, have some files that are several thousand lines long and close to half are over 200 lines.

What does that mean to us? 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 are.

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.

A newspaper is composed of many articles; most are very small. Some are a bit larger. Very few contain as much text as a page can hold. This makes the newspaper *usable*. If the newspaper were just one long story containing a disorganized agglomeration of facts, dates, and names, then we simply would not read it.

Vertical Openness Between Concepts

Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.

Consider, for example, Listing 5-1. 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.

Listing 5-1
`BoldWidget.java`

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Taking those blank lines out, as in Listing 5-2, has a remarkably obscuring effect on the readability of the code.

Listing 5-2
`BoldWidget.java`

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

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

If openness separates concepts, then vertical density implies close association. So lines of code that are tightly related should appear vertically dense. Notice how the useless comments in Listing 5-3 break the close association of the two instance variables.

Listing 5-3

```
public class ReporterConfig {

  /**
   * The class name of the reporter listener
   */
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Listing 5-4 is much easier to read. It fits in an “eye-full,” or at least it does for me. I can look at it and see that this is a class with two variables and a method, without having to move my head or eyes much. The previous listing forces me to use much more eye and head motion to achieve the same level of comprehension.

Listing 5-4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Vertical Distance

Have you ever chased your tail through a class, hopping from one function to the next, scrolling up and down the source file, trying to divine how the functions relate and operate, only to get lost in a rat's nest of confusion? Have you ever hunted up the chain of inheritance for the definition of a variable or function? This is frustrating because you are trying to understand *what* the system does, but you are spending your time and mental energy on trying to locate and remember *where* the pieces are.

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.

For those concepts that are so closely related that they belong in the same source file, their vertical separation should be a measure of how important each is to the understandability of the other. We want to avoid forcing our readers to hop around through our source files and classes.

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.

```
private static void readPreferences() {
  InputStream is= null;
  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

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

```
  public int countTestCases() {
    int count= 0;
    for (Test each : tests)
      count += each.countTestCases();
    return count;
  }
```

In rare cases a variable might be declared at the top of a block or just before a loop in a long-ish function. You can see such a variable in this snippet from the midst of a very long function in TestNG.

```
...
for (XmlTest test : m_suite.getTests()) {
     TestRunner tr = m_runnerFactory.newTestRunner(this, test);
     tr.addListener(m_textReporter);
     m_testRunners.add(tr);

     invoker = tr.getInvoker();

     for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
       beforeSuiteMethods.put(m.getMethod(), m);
     }

     for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
       afterSuiteMethods.put(m.getMethod(), m);
     }
   }
...
```

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.

There have been many debates over where instance variables should go. In C++ we commonly practiced the so-called *scissors rule*, which put all the instance variables at the bottom. The common convention in Java, however, is to put them all at the top of the class. I see no reason to follow any other convention. The important thing is for the instance variables to be declared in one well-known place. Everybody should know where to go to see the declarations.

Consider, for example, the strange case of the `TestSuite` class in JUnit 4.3.1. I have greatly attenuated this class to make the point. If you look about halfway down the listing, you will see two instance variables declared there. It would be hard to hide them in a better place. Someone reading this code would have to stumble across the declarations by accident (as I did).

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
    ...
  }
```

```
  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
     ...
  }

  public static Test warning(final String message) {
     ...
  }

  private static String exceptionToString(Throwable t) {
     ...
  }

  private String fName;

  private Vector<Test> fTests= new Vector<Test>(10);

  public TestSuite() {
  }

   public TestSuite(final Class<? extends TestCase> theClass) {
     ...
  }

  public TestSuite(Class<? extends TestCase>  theClass, String name) {
     ...
  }
  ... ... ... ... ...
}
```

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. Consider, for example, the snippet from FitNesse in Listing 5-5. Notice how the topmost function calls those below it and how they in turn call those below them. This makes it easy to find the called functions and greatly enhances the readability of the whole module.

Listing 5-5
`WikiPageResponder.java`

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
```

Listing 5-5 (continued)
`WikiPageResponder.java`

```
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
...
```

As an aside, this snippet provides a nice example of keeping constants at the appropriate level [G35]. The `"FrontPage"` constant could have been buried in the `getPageNameOrDefault` function, but that would have hidden a well-known and expected constant in an inappropriately low-level function. It was better to pass that constant down from the place where it makes sense to know it to the place that actually uses it.

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. Consider this snippet of code from Junit 4.3.1:

```
public class Assert {
  static public void assertTrue(String message, boolean condition) {
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

  static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
...
```

These functions have a strong conceptual affinity because they share a common naming scheme and perform variations of the same basic task. The fact that they call each other is secondary. Even if they didn't, they would still want to be close together.

Vertical Ordering

In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling.[2] This creates a nice flow down the source code module from high level to low level.

As in newspaper articles, we expect the most important concepts to come first, and we expect them to be expressed with the least amount of polluting detail. We expect the low-level details to come last. This allows us to skim source files, getting the gist from the

2. This is the exact opposite of languages like Pascal, C, and C++ that enforce functions to be defined, or at least declared, *before* they are used.

first few functions, without having to immerse ourselves in the details. Listing 5-5 is organized this way. Perhaps even better examples are Listing 15-5 on page 263, and Listing 3-7 on page 50.

Horizontal Formatting

How wide should a line be? To answer that, let's look at how wide lines are in typical programs. Again, we examine the seven different projects. Figure 5-2 shows the distribution of line lengths of all seven projects. The regularity is impressive, especially right around 45 characters. Indeed, every size from 20 to 60 represents about 1 percent of the total number of lines. That's 40 percent! Perhaps another 30 percent are less than 10 characters wide. Remember this is a log scale, so the linear appearance of the drop-off above 80 characters is really very significant. Programmers clearly prefer short lines.

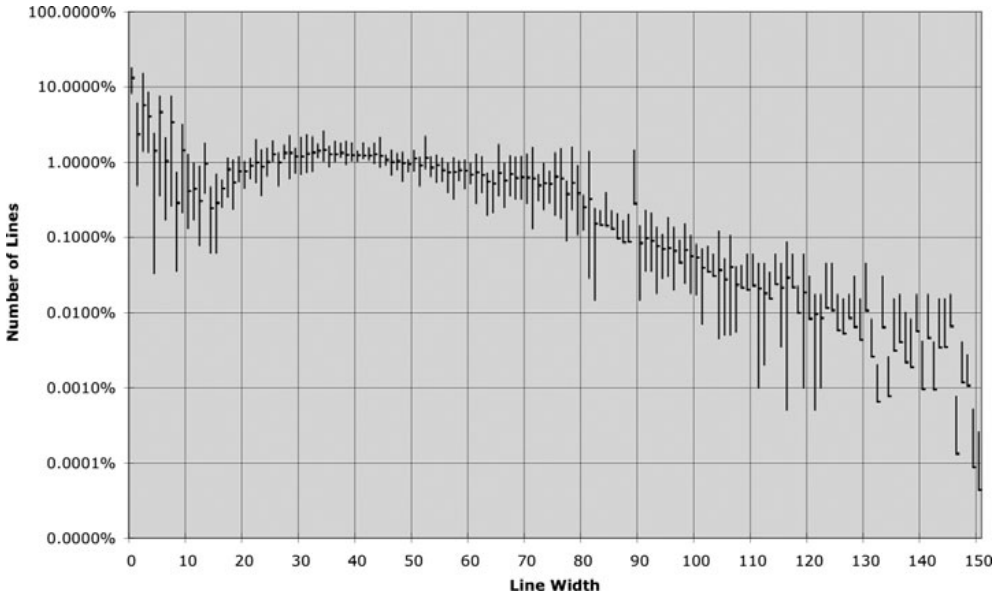


Figure 5-2
Java line width distribution

This suggests that we should strive to keep our lines short. The old Hollerith limit of 80 is a bit arbitrary, and I'm not opposed to lines edging out to 100 or even 120. But beyond that is probably just careless.

I used to follow the rule that you should never have to scroll to the right. But monitors are too wide for that nowadays, and younger programmers can shrink the font so small

that they can get 200 characters across the screen. Don't do that. I personally set my limit at 120.

Horizontal Openness and Density

We use horizontal white space to associate things that are strongly related and disassociate things that are more weakly related. Consider the following function:

```
  private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
  }
```

I surrounded the assignment operators with white space to accentuate them. Assignment statements have two distinct and major elements: the left side and the right side. The spaces make that separation obvious.

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.

Another use for white space is to accentuate the precedence of operators.

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Notice how nicely the equations read. The factors have no white space between them because they are high precedence. The terms are separated by white space because addition and subtraction are lower precedence.

Unfortunately, most tools for reformatting code are blind to the precedence of operators and impose the same spacing throughout. So subtle spacings like those shown above tend to get lost after you reformat the code.

Horizontal Alignment

When I was an assembly language programmer,[3] I used horizontal alignment to accentuate certain structures. When I started coding in C, C++, and eventually Java, I continued to try to line up all the variable names in a set of declarations, or all the rvalues in a set of assignment statements. My code might have looked like this:

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket          socket;
  private   InputStream     input;
  private   OutputStream    output;
  private   Request         request;
  private   Response        response;
  private   FitNesseContext context;
  protected long            requestParsingTimeLimit;
  private   long            requestProgress;
  private   long            requestParsingDeadline;
  private   boolean         hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

I have found, however, that this kind of alignment is not useful. The alignment seems to emphasize the wrong things and leads my eye away from the true intent. For example, in the list of declarations above you are tempted to read down the list of variable names without looking at their types. Likewise, in the list of assignment statements you are tempted to look down the list of rvalues without ever seeing the assignment operator. To make matters worse, automatic reformatting tools usually eliminate this kind of alignment.

So, in the end, I don't do this kind of thing anymore. 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. The length of the list of declarations in `FitNesseExpediter` below suggests that this class should be split up.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
```

3. Who am I kidding? I still am an assembly language programmer. You can take the boy away from the metal, but you can't take the metal out of the boy!

```
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;

  public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
  {
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Indentation

A source file is a hierarchy rather like an outline. There is information that pertains to the file as a whole, to the individual classes within the file, to the methods within the classes, to the blocks within the methods, and recursively to the blocks within the blocks. Each level of this hierarchy is a scope into which names can be declared and in which declarations and executable statements are interpreted.

To make this hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hiearchy. Statements at the level of the file, such as most class declarations, are not indented at all. Methods within a class are indented one level to the right of the class. Implementations of those methods are implemented one level to the right of the method declaration. Block implementations are implemented one level to the right of their containing block, and so on.

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.

Consider the following programs that are syntactically and semantically identical:

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }

-----


public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
```

```
  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Your eye can rapidly discern the structure of the indented file. You can almost instantly spot the variables, constructors, accessors, and methods. It takes just a few seconds to realize that this is some kind of simple front end to a socket, with a time-out. The unindented version, however, is virtually impenetrable without intense study.

Breaking Indentation. It is sometimes tempting to break the indentation rule for short `if` statements, 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. So I avoid collapsing scopes down to one line like this:

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

I prefer to expand and indent the scopes instead, like this:

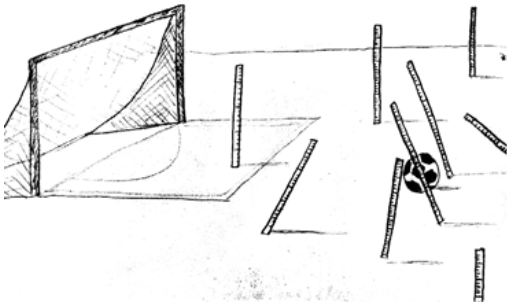
```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```

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



The title of this section is a play on words. Every programmer has his own favorite formatting rules, but if he works in a team, then the team rules.

A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a consistent style. We don't want it to appear to have been written by a bunch of disagreeing individuals.

When I started the FitNesse project back in 2002, I sat down with the team to work out a coding style. This took about 10 minutes. We decided where we'd put our braces, what our indent size would be, how we would name classes, variables, and methods, and so forth. Then we encoded those rules into the code formatter of our IDE and have stuck with them ever since. These were not the rules that I prefer; they were rules decided by the team. As a member of that team I followed them when writing code in the FitNesse project.

Remember, a good software system is composed of a set of documents that read nicely. They need to have a consistent and smooth style. The reader needs to be able to trust that the formatting gestures he or she has seen in one source file will mean the same thing in others. The last thing we want to do is add more complexity to the source code by writing it in a jumble of different individual styles.

Uncle Bob's Formatting Rules

The rules I use personally are very simple and are illustrated by the code in Listing 5-6. Consider this an example of how code makes the best coding standard document.

Listing 5-6
`CodeAnalyzer.java`

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;

  public CodeAnalyzer() {
    lineWidthHistogram = new LineWidthHistogram();
  }

  public static List<File> findJavaFiles(File parentDirectory) {
    List<File> files = new ArrayList<File>();
    findJavaFiles(parentDirectory, files);
    return files;
  }

  private static void findJavaFiles(File parentDirectory, List<File> files) {
    for (File file : parentDirectory.listFiles()) {
      if (file.getName().endsWith(".java"))
        files.add(file);
      else if (file.isDirectory())
        findJavaFiles(file, files);
    }
  }

  public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader(javaFile));
    String line;
    while ((line = br.readLine()) != null)
      measureLine(line);
  }

  private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
  }

  private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
      maxLineWidth = lineSize;
      widestLineNumber = lineCount;
    }
  }

  public int getLineCount() {
    return lineCount;
  }

  public int getMaxLineWidth() {
    return maxLineWidth;
  }
```

Listing 5-6 (continued)
CodeAnalyzer.java

```
  public int getWidestLineNumber() {
    return widestLineNumber;
  }

  public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
  }

  public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
  }

  public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
      cumulativeLineCount += lineCountForWidth(width);
      if (cumulativeLineCount > lineCount/2)
        return width;
    }
    throw new Error("Cannot get here");
  }

  private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
  }

  private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
  }
}
```

6

Objects and Data Structures

There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

Data Abstraction

Consider the difference between Listing 6-1 and Listing 6-2. Both represent the data of a point on the Cartesian plane. And yet one exposes its implementation and the other completely hides it.

Listing 6-1
Concrete Point

```
public class Point {
  public double x;
  public double y;
}
```

Listing 6-2
Abstract Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

The beautiful thing about Listing 6-2 is that there is no way you can tell whether the implementation is in rectangular or polar coordinates. It might be neither! And yet the interface still unmistakably represents a data structure.

But it represents more than just a data structure. The methods enforce an access policy. You can read the individual coordinates independently, but you must set the coordinates together as an atomic operation.

Listing 6-1, on the other hand, is very clearly implemented in rectangular coordinates, and it forces us to manipulate those coordinates independently. This exposes implementation. Indeed, it would expose implementation even if the variables were private and we were using single variable getters and setters.

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.

Consider Listing 6-3 and Listing 6-4. The first uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage. In the concrete case you can be pretty sure that these are just accessors of variables. In the abstract case you have no clue at all about the form of the data.

Listing 6-3
Concrete Vehicle

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Listing 6-4
Abstract Vehicle

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

In both of the above cases the second option is preferable. We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.

Data/Object Anti-Symmetry

These two examples show the difference between objects and data structures. Objects hide their data behind abstractions and expose functions that operate on that data. Data structure expose their data and have no meaningful functions. Go back and read that again. Notice the complimentary nature of the two definitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications.

Consider, for example, the procedural shape example in Listing 6-5. The `Geometry` class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the `Geometry` class.

Listing 6-5
Procedural Shape

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
```

Listing 6-5 (continued)
Procedural Shape

```
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Object-oriented programmers might wrinkle their noses at this and complain that it is procedural—and they'd be right. But the sneer may not be warranted. Consider what would happen if a `perimeter()` function were added to `Geometry`. The shape classes would be unaffected! Any other classes that depended upon the shapes would also be unaffected! On the other hand, if I add a new shape, I must change all the functions in `Geometry` to deal with it. Again, read that over. Notice that the two conditions are diametrically opposed.

Now consider the object-oriented solution in Listing 6-6. Here the `area()` method is polymorphic. No `Geometry` class is necessary. So if I add a new shape, none of the existing *functions* are affected, but if I add a new function all of the *shapes* must be changed![1]

Listing 6-6
Polymorphic Shapes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

1. There are ways around this that are well known to experienced object-oriented designers: VISITOR, or dual-dispatch, for example. But these techniques carry costs of their own and generally return the structure to that of a procedural program.

Listing 6-6 (continued)
Polymorphic Shapes

```
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Again, we see the complimentary nature of these two definitions; they are virtual opposites! 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.*

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO!

In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.

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. As we saw in the last section, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.

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*

2. http://en.wikipedia.org/wiki/Law_of_Demeter

- An object passed as an argument to *f*
- An object held in an instance variable of *C*

The method should *not* invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.

The following code[3] appears to violate the Law of Demeter (among other things) because it calls the `getScratchDir()` function on the return value of `getOptions()` and then calls `getAbsolutePath()` on the return value of `getScratchDir()`.

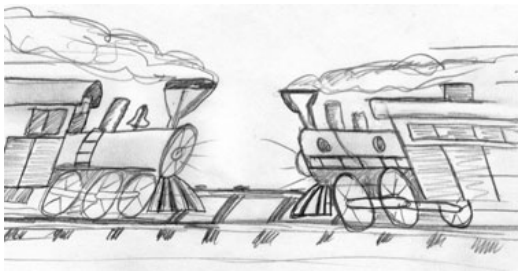
```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

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();
```

Are these two snippets of code violations of the Law of Demeter? Certainly the containing module knows that the `ctxt` object contains options, which contain a scratch directory, which has an absolute path. That's a lot of knowledge for one function to know. The calling function knows how to navigate through a lot of different objects.



Whether this is a violation of Demeter depends on whether or not `ctxt`, `Options`, and `ScratchDir` are 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;
```

This issue would be a lot less confusing if data structures simply had public variables and no functions, whereas objects had private variables and public functions. However,

3. Found somewhere in the apache framework.

there are frameworks and standards (e.g., “beans”) that demand that even simple data structures have accessors and mutators.

Hybrids

This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure. They have functions that do significant things, and they also have either public variables or public accessors and mutators that, for all intents and purposes, make the private variables public, tempting other external functions to use those variables the way a procedural program would use a data structure.[4]

Such hybrids make it hard to add new functions but also make it hard to add new data structures. They are the worst of both worlds. Avoid creating them. They are indicative of a muddled design whose authors are unsure of—or worse, ignorant of—whether they need protection from functions or types.

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. How then would we get the absolute path of the scratch directory?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

or

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

The first option could lead to an explosion of methods in the ctxt object. The second presumes that `getScratchDirectoryOption()` returns a data structure, not an object. Neither option feels good.

If `ctxt` is an object, we should be telling it to *do something;* we should not be asking it about its internals. So why did we want the absolute path of the scratch directory? What were we going to do with it? Consider this code from (many lines farther down in) the same module:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

The admixture of different levels of detail [G34][G6] is a bit troubling. Dots, slashes, file extensions, and `File` objects should not be so carelessly mixed together, and mixed with the enclosing code. Ignoring that, however, we see that the intent of getting the absolute path of the scratch directory was to create a scratch file of a given name.

---

4. This is sometimes called Feature Envy from [Refactoring].

So, what if we told the `ctxt` object to do this?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

That seems like a reasonable thing for an object to do! This allows `ctxt` to hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn't know about.

Data Transfer Objects

The quintessential form of a data structure is a class with public variables and no functions. This is sometimes called a data transfer object, or DTO. DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, and so on. They often become the first in a series of translation stages that convert raw data in a database into objects in the application code.

Somewhat more common is the "bean" form shown in Listing 6-7. Beans have private variables manipulated by getters and setters. The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other benefit.

Listing 6-7
`address.java`

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }
```

Listing 6-7 (continued)
address.java

```
  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

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`. Typically these Active Records are direct translations from database tables, or other data sources.

Unfortunately we often find that developers try to treat these data structures as though they were objects by putting business rule methods in them. This is awkward because it creates a hybrid between a data structure and an object.

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).

Conclusion

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects. Data structures expose data and have no significant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.

In any given system we will sometimes want the flexibility to add new data types, and so we prefer objects for that part of the system. Other times we will want the flexibility to add new behaviors, and so in that part of the system we prefer data types and procedures. Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.

Bibliography

[Refactoring]: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

This page intentionally left blank

7

Error Handling

by Michael Feathers

It might seem odd to have a section about error handling in a book about clean code. Error handling is just one of those things that we all have to do when we program. Input can be abnormal and devices can fail. In short, things can go wrong, and when they do, we as programmers are responsible for making sure that our code does what it needs to do.

The connection to clean code, however, should be clear. Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, *but if it obscures logic, it's wrong*.

In this chapter I'll outline a number of techniques and considerations that you can use to write code that is both clean and robust—code that handles errors with grace and style.

Use Exceptions Rather Than Return Codes

Back in the distant past there were many languages that didn't have exceptions. In those languages the techniques for handling and reporting errors were limited. You either set an error flag or returned an error code that the caller could check. The code in Listing 7-1 illustrates these approaches.

Listing 7-1
DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
      // Save the device status to the record field
      retrieveDeviceRecord(handle);
      // If not suspended, shut down
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended.  Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

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.

Listing 7-2 shows the code after we've chosen to throw exceptions in methods that can detect errors.

Listing 7-2
DeviceController.java (with exceptions)

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }
```

Listing 7-2 (continued)
`DeviceController.java (with exceptions)`

```
  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }

  ...
}
```

Notice how much cleaner it is. This isn't just a matter of aesthetics. The code is better because two concerns that were tangled, the algorithm for device shutdown and error handling, are now separated. You can look at each of those concerns and understand them independently.

Write Your `Try-Catch-Finally` Statement First

One of the most interesting things about exceptions is that they *define a scope* within your program. When you execute code in the `try` portion of a `try-catch-finally` statement, you are stating that execution can abort at any point and then resume at the `catch`.

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`.

Let's look at an example. We need to write some code that accesses a file and reads some serialized objects.

We start with a unit test that shows that we'll get an exception when the file doesn't exist:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

The test drives us to create this stub:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // dummy return until we have a real implementation
  return new ArrayList<RecordedGrip>();
}
```

Our test fails because it doesn't throw an exception. Next, we change our implementation so that it attempts to access an invalid file. This operation throws an exception:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Our test passes now because we've caught the exception. At this point, we can refactor. We can narrow the type of the exception we catch to match the type that is actually thrown from the `FileInputStream` constructor: `FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Now that we've defined the scope with a `try-catch` structure, we can use TDD to build up the rest of the logic that we need. That logic will be added between the creation of the `FileInputStream` and the `close`, and can pretend that nothing goes wrong.

Try to write tests that force exceptions, and then add behavior to your handler to satisfy your tests. This will cause you to build the transaction scope of the `try` block first and will help you maintain the transaction nature of that scope.

Use Unchecked Exceptions

The debate is over. For years Java programmers have debated over the benefits and liabilities of checked exceptions. When checked exceptions were introduced in the first version of Java, they seemed like a great idea. The signature of every method would list all of the exceptions that it could pass to its caller. Moreover, these exceptions were part of the type of the method. Your code literally wouldn't compile if the signature didn't match what your code could do.

At the time, we thought that checked exceptions were a great idea; and yes, they can yield *some* benefit. However, it is clear now that they aren't necessary for the production of robust software. C# doesn't have checked exceptions, and despite valiant attempts, C++ doesn't either. Neither do Python or Ruby. Yet it is possible to write robust software in all of these languages. Because that is the case, we have to decide—really—whether checked exceptions are worth their price.

What price? The price of checked exceptions is an Open/Closed Principle[1] violation. If you throw a checked exception from a method in your code and the `catch` is three levels above, *you must declare that exception in the signature of each method between you and the* `catch`. This means that a change at a low level of the software can force signature changes on many higher levels. The changed modules must be rebuilt and redeployed, even though nothing they care about changed.

Consider the calling hierarchy of a large system. Functions at the top call functions below them, which call more functions below them, ad infinitum. Now let's say one of the lowest level functions is modified in such a way that it must throw an exception. If that exception is checked, then the function signature must add a `throws` clause. But this means that every function that calls our modified function must also be modified either to catch the new exception or to append the appropriate `throws` clause to its signature. Ad infinitum. The net result is a cascade of changes that work their way from the lowest levels of the software to the highest! Encapsulation is broken because all functions in the path of a throw must know about details of that low-level exception. Given that the purpose of exceptions is to allow you to handle errors at a distance, it is a shame that checked exceptions break encapsulation in this way.

Checked exceptions can sometimes be useful if you are writing a critical library: You must catch them. But in general application development the dependency costs outweigh the benefits.

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*.

---

1. [Martin].

Let's look at an example of poor exception classification. Here is a `try-catch-finally` statement for a third-party library call. It covers all of the exceptions that the calls can throw:

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

That statement contains a lot of duplication, and we shouldn't be surprised. In most exception handling situations, the work that we do is relatively standard regardless of the actual cause. We have to record an error and make sure that we can proceed.

In this case, because we know that the work that we are doing is roughly the same regardless of the exception, we can simplify our code considerably by wrapping the API that we are calling and making sure that it returns a common exception type:

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

Our `LocalPort` class is just a simple wrapper that catches and translates exceptions thrown by the `ACMEPort` class:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
```

```
        throw new PortDeviceFailure(e);
      }
    }
    …
}
```

Wrappers like the one we defined for `ACMEPort` can be very useful. In fact, wrapping third-party APIs is a best practice. When you wrap a third-party API, you minimize your dependencies upon it: You can choose to move to a different library in the future without much penalty. Wrapping also makes it easier to mock out third-party calls when you are testing your own code.

One final advantage of wrapping is that you aren't tied to a particular vendor's API design choices. You can define an API that you feel comfortable with. In the preceding example, we defined a single exception type for `port` device failure and found that we could write much cleaner code.

Often a single exception class is fine for a particular area of code. The information sent with the exception can distinguish the errors. Use different classes only if there are times when you want to catch one exception and allow the other one to pass through.

Define the Normal Flow

If you follow the advice in the preceding sections, you'll end up with a good amount of separation between your business logic and your error handling. The bulk of your code will start to look like a clean unadorned algorithm. However, the process of doing this pushes error detection to the edges of your program. 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 `ExpenseReportDAO` so 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

I think that any discussion about error handling should include mention of the things we do that invite errors. The first on the list is returning `null`. I can't begin to count the number of applications I've seen in which nearly every other line was a check for `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);
      }
    }
  }
}
```

If you work in a code base with code like this, it might not look all that bad to you, but it is bad! When we return `null`, we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing `null` check to send an application spinning out of control.

Did you notice the fact that there wasn't a `null` check in the second line of that nested `if` statement? What would have happened at runtime if `persistentStore` were `null`? We would have had a `NullPointerException` at runtime, and either someone is catching `NullPointerException` at the top level or they are not. Either way it's *bad*. What exactly should you do in response to a `NullPointerException` thrown from the depths of your application?

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.

In many cases, special case objects are an easy remedy. Imagine that you have code like this:

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Right now, `getEmployees` can return `null`, but does it have to? If we change `getEmployee` so that it returns an empty list, we can clean up the code:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

Fortunately, Java has `Collections.emptyList()`, and it returns a predefined immutable list that we can use for this purpose:

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

If you code this way, you will minimize the chance of `NullPointerExceptions` and your code will be cleaner.

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.

Let's look at an example to see why. Here is a simple method which calculates a metric for two points:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

What happens when someone passes `null` as an argument?

```
calculator.xProjection(null, new Point(12, 13));
```

We'll get a `NullPointerException`, of course.

How can we fix it? We could create a new exception type and throw it:

```
public class MetricsCalculator
{
```

```
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x - p1.x) * 1.5;
  }
}
```

Is this better? It might be a little better than a `null` pointer exception, but remember, we have to define a handler for `InvalidArgumentException`. What should the handler do? Is there any good course of action?

There is another alternative. We could use a set of assertions:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x - p1.x) * 1.5;
  }
}
```

It's good documentation, but it doesn't solve the problem. If someone passes `null`, we'll still have a runtime error.

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.

Conclusion

Clean code is readable, but it must also be robust. These are not conflicting goals. We can write robust clean code if we see error handling as a separate concern, something that is viewable independently of our main logic. To the degree that we are able to do that, we can reason about it independently, and we can make great strides in the maintainability of our code.

Bibliography

[Martin]: *Agile Software Development: Principles, Patterns, and Practices,* Robert C. Martin, Prentice Hall, 2002.

8

Boundaries

by James Grenning

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, `Map`s 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 `Map`s do not reliably constrain the types of objects placed within them. Any determined user can add items of any type to any `Map`.

- `clear() void - Map`
- `containsKey(Object key) boolean - Map`
- `containsValue(Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`
- `notifyAll() void - Object`
- `put(Object key, Object value) Object - Map`
- `putAll(Map t) void - Map`
- `remove(Object key) Object - Map`
- `size() int - Map`
- `toString() String - Object`
- `values() Collection - Map`
- `wait() void - Object`
- `wait(long timeout) void - Object`
- `wait(long timeout, int nanos) void - Object`

Figure 8-1
The methods of `Map`

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

```
Map sensors = new HashMap();
```