Chapter 3. **Functions**

* Functions are the first line of organization in any program.
* 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");  } |

|  |
| --- |
| **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();  } |

**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.
* 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 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();  } |

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.

**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,

**Switch Statements**

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. 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.

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 Principle7 (SRP) because there is more than one reason for it to change. Fourth, it violates the Open Closed Principle8 (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.

**Use Descriptive**

|  |
| --- |
| **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. 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.

**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.

**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.

**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 46 Chapter 3: Functions 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");

...

}

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());

}

**Exact 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.

**Prefer Exceptions to Returning Error Codes**

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.

**Don’t Repeat Yourself**

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.

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.

**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.

Chapter 4 **Comments**

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 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 everdecreasing accuracy.

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

y 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});

**Good Comments**

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.

**Bad Comments**

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

**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.

**Journal Comments**

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

**Noise Comments**

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

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.