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 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;
}
```

`TODO`s 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 java-docs 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;

/**
  * 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();
 }
  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 {
      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;
      // 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
 * 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;
}

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.