5

Comments

“Don’t comment bad code—rewrite it.”

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

---

1. [KP78], p. 144.

“Comments are the distracting footnotes of code.”

—Jeff Langr

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.

Compensating for Failure

Comments are, at best, a necessary evil. If our programming languages were expressive enough, or if you and I 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 ourselves in code. Note that I used the word *failure*. I meant it. Comments are always failures of either our languages or our abilities.

This doesn't mean that we never write comments. We must sometimes write comments because we cannot always figure out how to express ourselves without them. However, their use is not a cause for celebration.

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 our languages and of our ability to express ourselves in those languages.

The comments you write may be necessary; but that necessity is also unfortunate.

It has been said that advice like this gives license to younger programmers to avoid writing comments. Shame on any programmers, young or old, who use these recommendations as an excuse for their laziness and/or unwillingness to add proper explanations and context. In the end, I cannot concern myself with how others might abuse my advice. My sole purpose here is help those who will not.

Hidden or Obscured Comments

Does your IDE hide the first comment in every source file? Mine does. That's because we seldom want to read that comment. We expect that it's some kind of boilerplate license/legalese that offers no insight into the actual intent of the module.

But collapsing the first comment is not the only way that our IDEs hide or obscure comments from us.

What color does your IDE use to display comments? For some of you, it will be a nice green—like the grass. And so, like the grass, you can ignore them. They fade into the background like the neighborhood lawns.

For others of you, it will be a nice soft gray; not invisible, but strongly de-emphasized.

Why? Because we don't want them in the way. We don't really want to see them. We like the fact that they are there; but we don't want them in our face.

I have my IDE paint comments in bright fire-engine red. I do this because I reckon that if someone wrote a comment, *I should read it.* And then, if that comment does not help me, I delete it on the spot. Or if that comment needs adjusting, I adjust it on the spot.

Comments, when they exist, should be visible!

Lying Comments

Why am I so down on comments? I'm really not. I'm only down on bad comments. I wholly support good comments. The problem, however, is that too many of the comments I come across fall into the former category.

First of all, they often lie. The lies aren't intentional, but they are also not infrequent. 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 will often change and evolve. Chunks of it will move from here to there. Those chunks bifurcate and reproduce and come together again to form hybrids and chimeras. Unfortunately, the comments don't, and can't, always follow them. 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}
  "[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.

You might complain that programmers should be disciplined enough to keep the comments in a high state of repair, relevance, and accuracy. However, I would rather that discipline and 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 certain in one place: the code. Only the code can be trusted to tell you what it does. It is the most reliable source of accurate information. Therefore, though comments are sometimes necessary, we ought to expend significant energy to minimize them.

Comments That Are Too Intimate

A significant problem with comments is that they are most often written by programmers who have an intimate understanding of the code, and are therefore written from the point of view of that intimacy. Unfortunately, the target reader of the comments does not share that intimacy, and therefore does not have the context to understand the comments.

It has happened to me, far more than once, that I could not understand a comment so helpfully provided by the author until I understood that author's code. The words in the comment simply didn't make sense to me,

nor help me in any way. Once I read through the code and understood it, then the intent of the comment became clear—but not before.

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 Your Intent 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
if ((employee.flags & HOURLY_FLAG) && (employee.age >
```

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. What follows are a few examples of comments that are worthy of the bits they consume—or are at

least not worth deleting.

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) 2003-2005 by Object Mentor, Inc. All
// Released under the GNU General Public License vers
```

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();
```

Why did the author resort to a comment instead of giving the method a better name like `responderBeingTested` ? And why would I allow this comment rather than changing the name of the method? That's because the author was using the well-known Singleton[2] design pattern. When you use a design pattern like that, you should follow the canonical form specified by that pattern. In the Singleton pattern, the name of the accessor function always ends in the word `Instance` . So the author decided to fall back on a comment, rather than abandon the canonical form of the pattern.

2. [GOF95] p. 127.

Here's another case where the comment adds helpful information.

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

I would not delete this comment, because regular expressions are always a nightmare to understand. This one is no different, and it was polite of the author to give us a hint with a comment that lets us know that the regular expression is intended to match a timestamp. Any Java programmer would easily infer this from the `kk:mm:ss EEE, MMM dd, yyyy` string that uses the codes from the `SimpleDataFormat` class.

However, the comment is a lie—or at least significantly misinformative. That's because the regular expression would also match the following string: `":: , , "` . And that is definitely not a timestamp. So I would fix the regular expression and let the comment remain.

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 the containing 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
    return compressedName.compareTo(compressedArgumer
  }
```

```
    return 1; // we are greater because we are the righ
  }
```

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 Excepti
    WidgetBuilder widgetBuilder = …

      //This is our best attempt to get a race conditic
      //by creating large number of threads.
      for (int i = 0; i < 25000; i++) {
        WidgetBuilderThread widgetBuilderThread =
          new WidgetBuilderThread(widgetBuilder, text,
        Thread thread = new Thread(widgetBuilderThread)
        thread.start();
      }
      assertEquals(false, failFlag.get());
    }
```

I would not delete that comment as part of a code review. However, I'd be tempted to get rid of it by extracting that `for` loop out into a function named `attemptRaceCondition` .

Future Bob:

As an aside, this is a terrible way to force a race condition. When you start up a bunch of threads in hopes that they will race, what they will do instead is line up single file and execute without racing. If you want to force a race condition, you either have to set the race up with semaphores, or introduce random delays and hope that the threads will slip past one another and accidentally enter a race configuration.

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 it's part of the standard library, or in code that you cannot alter, a helpful clarifying comment can be useful.

I wrote the following comments as part of a feature in FitNesse. I was concerned that the long list of `assertTrue` statements would make my readers' eyes cross. Optical impediments and illusions like this are particularly common in code. So I added the comments to be polite.

```
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, however, that a clarifying comment is incorrect. In fact, one of the comments in the preceding code is wrong. Go through the previous example and see if you can find it; and note how difficult it is to verify them. That difficulty explains why the clarification was necessary and also why it can be risky.

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", respc
  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 makeStandardHttpDateFc
{
  //SimpleDateFormat is not thread safe,
  //so we need to create each instance independently.
  SimpleDateFormat df = new SimpleDateFormat("EEE, dc
  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.[3] It will prevent some overly eager programmer from using a static initializer in the name of efficiency.

---

3. You've got to wonder how, in the language of the internet, an environment that is intrinsically asynchronous, this function has been allowed to remain thread unsafe for so many decades.

Amplification

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

A good example of that is the call to `trim()` below. The `trim` function is called so frequently and so frivolously that it's easy to ignore. The comment makes clear that you should not ignore this one.

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

Javadocs (and Their Ilk) in Public APIs

Nowadays, most language systems have some kind of documentation system that scrapes special document comments from your source files and arranges them in a nicely formatted document. In Java these are called *Javadocs*.

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 document comments for it. But keep in mind the rest of the advice in this chapter. Document comments can be just as misleading, nonlocal, and dishonest as any other kind of comment.

Bad Comments

Bad comments are usually crutches or excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to themselves.

When I see comments in this category, I delete them. What follows are some specific examples of such comments.

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. It appears to me that 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 + "/" +
   FileInputStream propertiesStream =
                  new FileInputStream(propertiesPath
   loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loa
  }
}
```

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 here. 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.

Wouldn't it have been better if the programmer had written the following?

```
catch(IOException e)
{
  LoadedProperties.loadDefaults();
}
```

Redundant Comments

The following code shows a simple function with a header comment that tells us nothing that the code does not. The comment probably takes longer to read than the code itself.

```
// Utility method that returns when this.closed is tr
// Throws an exception if the timeout is reached.
public synchronized void
waitForClose(long timeoutMillis) throws Exception
{
   if(!closed)
   {
      wait(timeoutMillis);
      if(!closed)
        throw new Exception("MockResponseSender could
   }
}
```

What purpose does this comment serve? It's certainly not more informative than the code. It does not justify the code, nor 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 glad-handing used-car salesman assuring you that you don't need to look under the hood.

Misleading Comments

Sometimes, with all the best intentions, a programmer makes a statement in their comments that allows misinterpretation. Consider, for another moment, the badly redundant but also subtly misleading comment we saw in the last example.

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 timeout 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 themselves in a debugging session trying to figure out why their code executed so slowly.

Redundancy and Imprecision

Now consider the legion of useless and redundant Javadocs in this code taken long ago from 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 Contai
 * associated.
 */
protected Loader loader = null;


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


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

/**
  * The Manager implementation with which this Cont
  * associated.
  */
protected Manager manager = null;


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


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


/**
```

```
     * The parent Container to which this Container is
     */
    protected Container parent = null;


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


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


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


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

These comments serve only to clutter and obscure the code. They serve no documentary purpose at all. Look at the first one. The variable is named `backgroundProcessorDelay`. The comment is `The processor delay for this component`.

The redundancy is painful. The variable name says more than the comment. And what in the world did the author mean by the word *component*?

Look further down. The same redundancy glares at us from every one of those Javadocs. On top of that, the author seems to use the words *component* and *container* interchangeably. Is that intentional? Is there a difference between `component` and `container`? And why are they

sometimes capitalized and sometimes not? And look at `parentClassLoader`. Is the lack of `component` or `container` significant?

Perhaps you think I'm being picky. I probably am. But code is *precise*, and imprecision in the comments can be both confusing and misleading. It also makes me wonder if that imprecision carries over into the code.

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 misinformation, and lend to general confusion and disorganization.

For example, required Javadocs for every function lead to abominations like this:

```
/**
 * @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 ir
 */
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);
}
```

If you ran the Javadoc tool over this code, and then deleted the comments and ran the tool again, the two HTML files created would be nearly identical. So the clutter of this Javadoc adds nothing and serves only to obfuscate the code and create the potential for misdirection.

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 t
*               package com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and
*               eliminated NotableDate class (DG);
* 12-Nov-2001 : IBD requires setDescription() method,
*               that NotableDate class is gone (DG);
*               Changed getPreviousDayOfWeek(),
*               getFollowingDayOfWeek() and
*               getNearestDayOfWeek() to correct
*               bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (D
* 29-May-2002 : Moved the month constants into a sepa
*               interface (MonthConstants) (DG);


* 27-Aug-2002 : Fixed bug in addMonths() method, thar
*               N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 0Sep-2003 : Implemented Comparable.  Updated the is
*             javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (10962
```

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. Let the source code control system keep the journal for you.

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 the code, our eyes simply skip over them. Eventually, as the code changes, we fail to notice that the comments have begun to lie.

The first comment below seems appropriate.[4] 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.

---

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

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // normal. someone stopped the request.
  }
  catch(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString
      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, like this:

```
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?

`TODO` Comments

In the first edition of this book, I was in favor of leaving `TODO` comments in the code as reminders of things that need to be taken care of. I was enamored of the features in my IDE that allowed me to quickly see all the `TODO` comments in my projects.

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

I have since changed my mind about this. After a few years of reading code written by others and by myself, I came to understand that `TODO` means *Don't Do*. So I changed my rule.

I still use `TODO` comments, but I will not check them in. I will either do the thing that needs to be done, or eliminate the reason that it needs to be done, or put it into the backlog of tasks that need to be performed.

Use a Function or a Variable Instead of a Comment

Consider the following stretch of code:

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

What does this comment mean? What is `<mod>`, and why is it in angle brackets? I put this code into my IDE and extracted out a few explanatory variables. Here is the result.

```
ArrayList moduleDependants = smodule.getDependSubsyst
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependants.contains(ourSubSystem))
```

It seems to me that this refactoring helps the final `if` statement read like well-written prose and is easier to understand than the comment (which I still don't understand).

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 avoided—especially the noisy train of slashes at the end.

Think of it this way. A banner is startling and obvious, especially 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—like the little boy who cried "Wolf!"

Attributions and Bylines

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

This is just graffiti. Please don't spray-paint your name all over the source code. 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. The same can be said for PR numbers and JIRA tags. Put those in commit comments, not in the source code.

Commented-Out Code

Few practices are as odious as commenting-out code. Commented-out code is an abomination before nature and nature's God. Don't check this stuff in!

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

Others who see that commented-out code will be unlikely to 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, th
}

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 anymore.

I still comment out code while debugging from time to time. It can be a useful technique. But my rule is to avoid checking in the commented-out

code.

HTML Comments

HTML in source code comments can be awful, as you can tell by reading the code below. It makes the comments hard to read in the one place where they should be easiest to read: the code. If comments are going to be extracted by some tool (like Javadoc) to appear in a nicely formatted document, then it should be the responsibility of that tool, and not the programmer, to adorn the comments with appropriate formatting.

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the re
 * <p/>
 * <pre>
 * Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot
 *     classname=&quot;fitnesse.ant.ExecuteFitnesseTe
 *     classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 *             resource=&quot;tasks.properties&quot;
 * <p/>
 * &lt;execute-fitnesse-tests
 *     suitepage=&quot;FitNesse.SuiteAcceptanceTests&
 *     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 8082
 *
 * @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
    Part One: Format of Internet Message Bodies
    section 6.8.  Base64 Content-Transfer-Encoding
    The encoding process represents 2bit groups of ir
    strings of 4 encoded characters. Proceeding from
    2bit input group is formed by concatenating 3 8-b
    These 24 bits are then treated as 4 concatenated
    of which is translated into a single digit in the
    When encoding a bit stream via the base64 encodir
    must be presumed to be ordered with the most-sign
    That is, the first bit in the stream will be the
    the first 8-bit byte, and the eighth bit will be
    in the first 8-bit byte, and so on.
  */
```

Unobvious 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
 * (plus filter bytes), and an extra 200 bytes for he
 */
this.pngBytes = new byte[((this.width + 1) * this.hei
```

What is a filter byte? Did the author mean filler byte? Do those bytes 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 and signature for a small function that does one thing is *usually* better than a comment header. For example, I would prefer:

```
public static int[] generateNPrimes(int n) {
```

over:

```
/**
 * Computes the first prime numbers; the return value
 * computed primes, in increasing order of size.
 * @param n
 *      How many prime numbers to compute.
 */
public static int[] generate(int n) {
```

Javadocs in Nonpublic Code

As useful as Javadocs are for public APIs, they are more of an impediment than a benefit to code that is maintained by a small team and is not intended for public consumption. The team already knows the code and is not likely to run the Javadoc tool to generate Javadoc pages, so the extra formality of the Javadoc comments amounts to little more than cruft and distraction.

Example

I wrote the following module 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.

```
/**
 * This class Generates prime numbers up to a user sp
 * maximum.  The algorithm used is the Sieve of Eratc
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Liby
 * d. c. 194, Alexandria.  The first man to calculate
 * circumference of the Earth.  Also known for workir
 * calendars with leap years and ran the library at A
 * <p>
 * The algorithm is quite simple.  Given an array of
 * starting at 2.  Cross out all multiples of 2.  Fir
 * uncrossed integer, and cross out all of its multip
 * Repeat until you have passed the square root of th
 * 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 mul
      {
        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
```

```
    }
  }
```

In the following, 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.

```
/**
 * This class Generates prime numbers up to a user sp
 * maximum.  The algorithm used is the Sieve of Eratc
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out al
 * multiples.  Repeat until there are no more multipl
 * 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 maxValu
  {
    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
      // is less than or equal to the root of the array
      // so we don't have to cross out multiples of num
      // larger than that root.
      double iterationLimit = Math.sqrt(crossedOut.leng
      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 comment is almost certainly helpful. 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.

Conclusion

Comments are not like Schindler's list—they are not "pure good." They are sometimes utterly essential and other times dangerous and damaging. The only language of "truth" at our disposal is our code. Unfortunately, that language is not always adequate to explain the intention behind our decisions. In those cases, the risk of using an ambiguous, amorphous, and contradictory language like English (or whatever your native language is) can sometimes be worth it. But tread carefully—there dwell dragons.