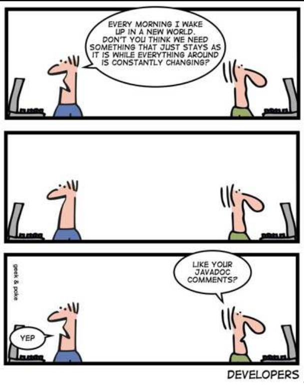

EVERY MORNING I WAKE UP IN A NEW WORLD. DON'T YOU THINK WE NEED SOMETHING THAT JUST STAYS AS IT IS WHILE EVERYTHING AROUND IS CONSTANTLY CHANGING?
LIKE YOUR JAVADOC COMMENTS?
YEP
geek & poke
DEVELOPERS

Clean Comments

www.cs.uoi.gr/~zarras/soft-devII.htm

from Clean Code by R. C. Martin, a.k.a "Uncle Bob"

Comments - A necessary evil !!

The proper use of comments is to **compensate** for our **failure** to **express** ourselves in code !!!

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….

Comments don't 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!"

Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.

Explain yourself in code !!

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
……
```

A comment over a complex expression ….

Can we do better than this ??

Explain yourself in code !!

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
……
```

A comment over a complex expression ….

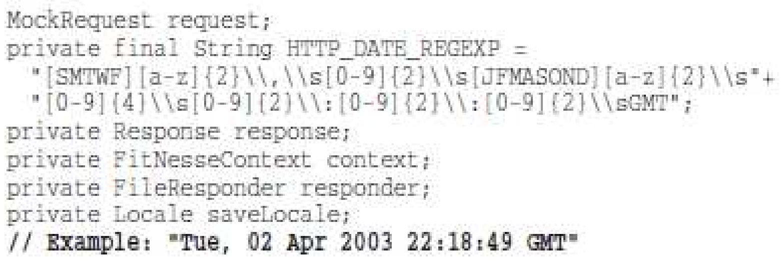
Can we do better that this ⁇

It takes only a few seconds of thought to explain most of our 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.

```
if (employee.isEligibleForFullBenefits())
```

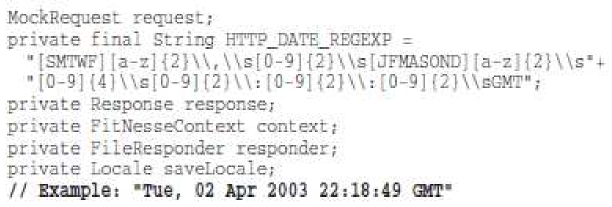
Comments lie !!



```
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"
```

Do you see anything strange ⁇

Comments lie !!



```
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"
```

Comments 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.*

Good Comments !!

Legal Comments

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

Copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file.

BUT, comments like this should not be contracts or legal tomes.

For example, 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 **that is not present in the implementation** with a comment.

For example, this comment that explains the return value of an abstract method; the method does not have an implementation so we cannot figure out the intent of the returned value:

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

Could we avoid it ??

Informative Comments

It is sometimes useful to provide basic information that is not present in the implementation with a comment.

For example, this comment that explains the return value of an abstract method; the method does not have an implementation so there is no way to figure out the intent of the returned value:

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

Could we avoid it ??

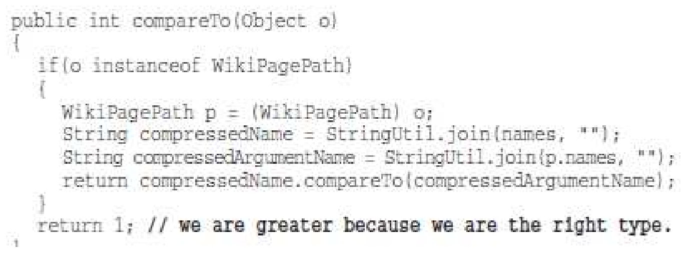
The method is not implemented but a better name would do

```
protected abstract Responder getResponderBeingTested();
```

Explain design decisions

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 any object that does not belong to the hosting class (WikiPagePath), with a WikiPagePath object, the latter is considered greater.

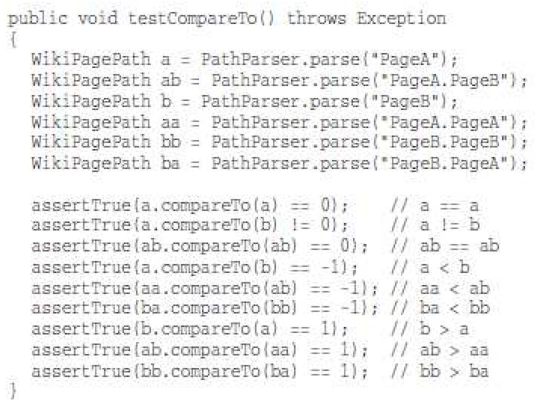


```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
 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

without the comment we could wonder why, with the comment its clearer that this was just a decision taken by the author

Clarification of arguments, ret values

Sometimes it is just helpful to clarify the meaning of some obscure argument or return value into something that's readable.



```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");

  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b > a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

Useful but hard to verify that comments are correct.
Are they ??

Warnings

Sometimes it is useful to warn other programmers about certain consequences.

```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile(){
  writeLinesToFile(10000000);
  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

TODO

It is sometimes reasonable to leave "To do" notes in the form of //TODO comments.

In the following case, the TODO comment explains why the function has a degenerate implementation and what that function's future should be.

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

TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment.

It might be a reminder to delete a deprecated feature or a plea for someone else to look at a problem.

It might be a request for someone else to think of a better name or a reminder to make a change.

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()));
```

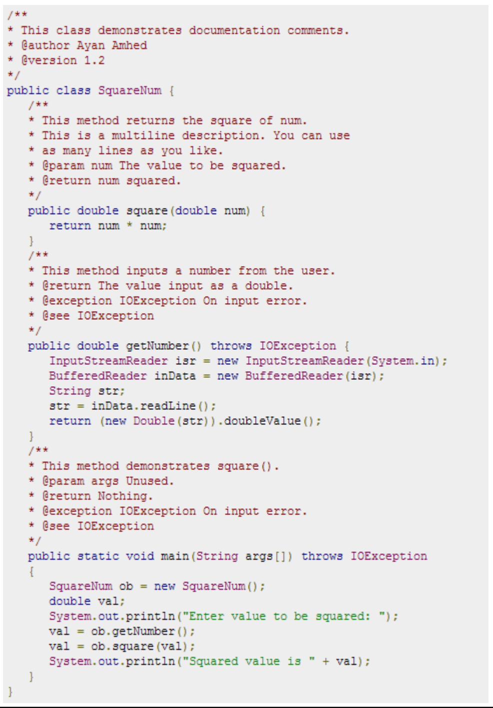
Javadocs

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 we are writing a public API, then we should certainly write good javadocs for it.

But keep that Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment.



```java
/**
* This class demonstrates documentation comments.
* @author Ayan Amhed
* @version 1.2
*/
public class SquareNum {
   /**
   * This method returns the square of num.
   * This is a multiline description. You can use
   * as many lines as you like.
   * @param num The value to be squared.
   * @return num squared.
   */
   public double square(double num) {
      return num * num;
   }
   /**
   * This method inputs a number from the user.
   * @return The value input as a double.
   * @exception IOException On input error.
   * @see IOException
   */
   public double getNumber() throws IOException {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader inData = new BufferedReader(isr);
      String str;
      str = inData.readLine();
      return (new Double(str)).doubleValue();
   }
   /**
   * This method demonstrates square().
   * @param args Unused.
   * @return Nothing.
   * @exception IOException On input error.
   * @see IOException
   */
   public static void main(String args[]) throws IOException
   {
      SquareNum ob = new SquareNum();
      double val;
      System.out.println("Enter value to be squared: ");
      val = ob.getNumber();
      val = ob.square(val);
      System.out.println("Squared value is " + val);
   }
}
```

Bad Comments !!

Mumbling

```
public void loadProperties()
{
  try
  {
   String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
   FileInputStream propertiesStream = new FileInputStream(propertiesPath);
   loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
  }
}
```

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.

Mumbling

```
public void loadProperties()
{
  try
  {
   String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
   FileInputStream propertiesStream = new FileInputStream(propertiesPath);
   loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
  }
}
```

MAYBE, if we get an IOException, it means that there was no properties file; and in that case all the defaults are loaded.

But are they loaded in a different place ? Where ? Who is responsible for that ?

Mumbling

```
public void loadProperties()
{
  try
  {
   String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
   FileInputStream propertiesStream = new FileInputStream(propertiesPath);
   loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
  }
}
```

Or — and this is the **scary possibility** —the author trying to tell himself to come back here later and write the code that would load the defaults?

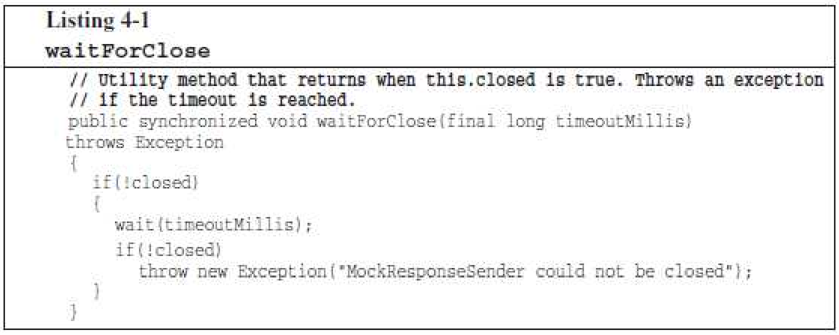
Mumbling

```
public void loadProperties()
{
  try
  {
   String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
   FileInputStream propertiesStream = new FileInputStream(propertiesPath);
   loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
  }
}
```

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

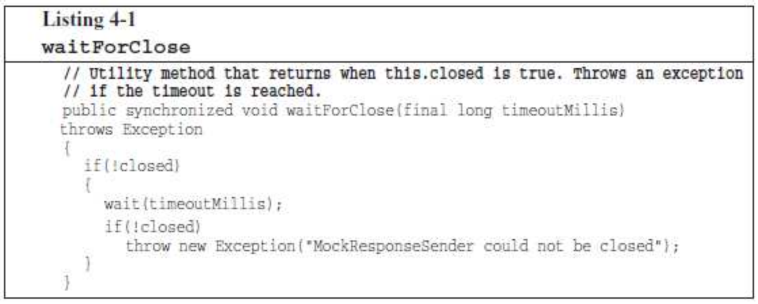


Listing 4-1
waitForClose

```
// 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?

Redundant Comments



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");
   }
}
```

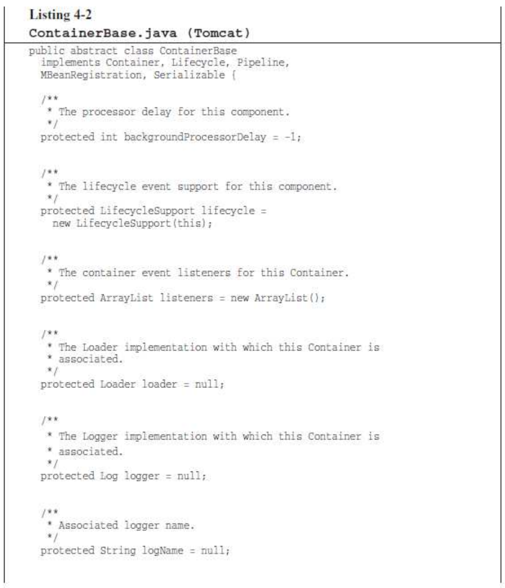
It's certainly not more informative than the code.

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.

Probably it **takes longer** to read than the code itself !!!

Redundant Comments



Listing 4-2

ContainerBase.java (Tomcat)

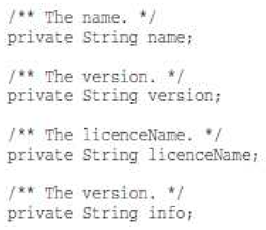
```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * 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;
```

More useless and redundant javadoc comments in a class taken from Tomcat.

These comments serve only to clutter and obscure the code.

Remember → vertical density !!

Redundant Comments



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

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

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

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

more redundant/noise comments !!

But there is something even more scary here !!

can you spot it ??

Redundant Comments

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

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

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

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

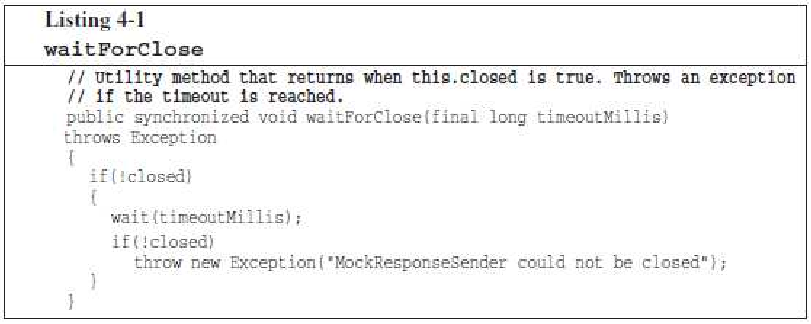
more redundant/noise comments !!

But there is something even more scary here !!

can you spot it ??

there is a copy paste error in the last comment !!

Misleading Comments



Listing 4-1

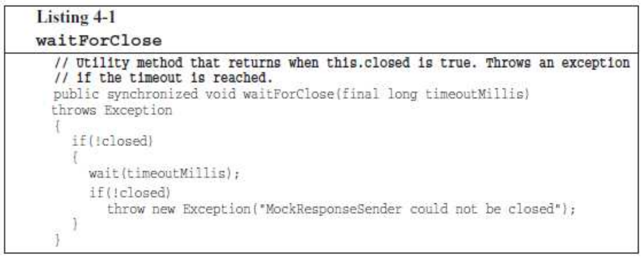
`waitForClose`

```
// 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");
  }
}
```

Sometimes, with all the best intentions, a programmer makes a statement in his comments that isn't precise enough to be accurate.

Do you see how the above comment is misleading?

Misleading Comments


Listing 4-1
waitForClose

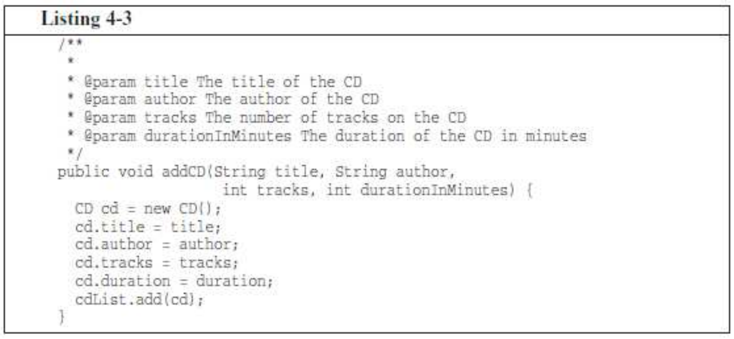
```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could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

Sometimes, with all the best intentions, a programmer makes a statement in his comments that **isn't precise** enough to be accurate.

Do you see how the above comment is misleading?

The method does not return when closed is true ; it returns if closed is true, else it waits for a time out and if closed is still false it throws an exception.

Mandated Comments

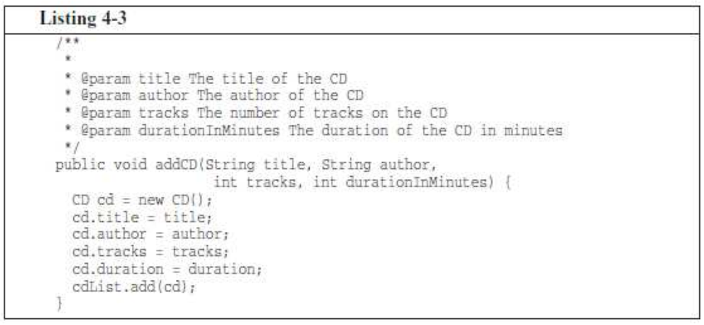

Listing 4-3

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

How about these comments ??

These comments are there because of a rule followed by the development team that requires each function to have a javadoc comment

Mandated Comments



Listing 4-3

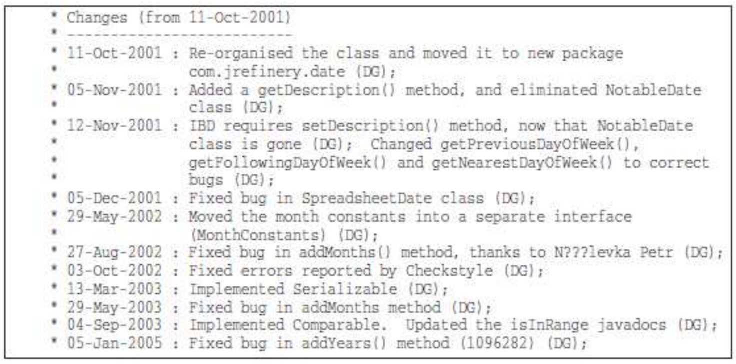
```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {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

How about these comments ⁇

These comments are there because of a rule followed by the development team that requires each function to have a javadoc comment

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 ‼

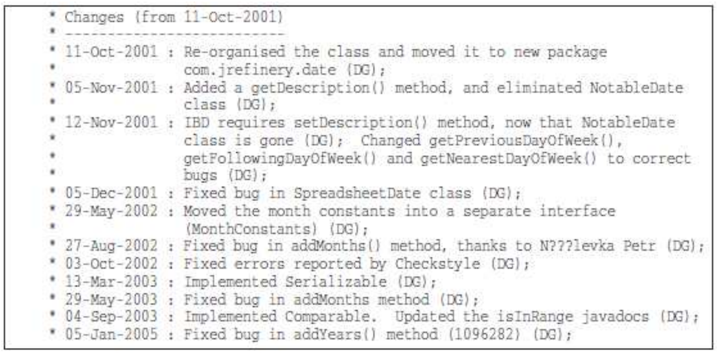
Log Comments



```
* 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);
```

How about these comments ⁇

Log Comments



```
* 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);
```

How about these comments ??

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 comments are just more clutter to obfuscate the module. They should be **completely removed**.

Position Markers

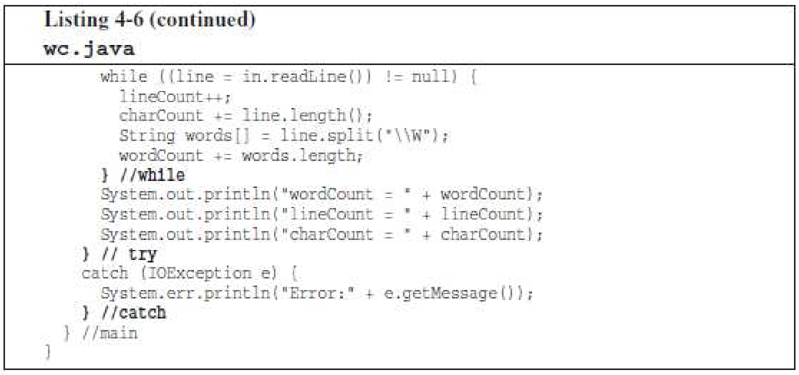
Sometimes programmers like to mark a particular position in a source file.

```
// 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.

So use them very sparingly, and only when the benefit is significant.

Closing Brace Comments



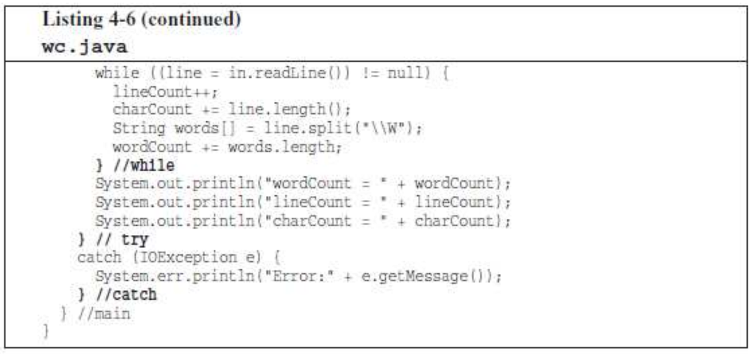
Listing 4-6 (continued)

wc.java

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

Closing brace comments make sense for long functions with deeply nested structures.

Closing Brace Comments



Listing 4-6 (continued)

wc.java

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

Closing brace comments make sense for long functions with deeply nested structures.
BUT, we don't like long functions !!

So if you find yourself wanting to mark your closing braces, try to shorten your functions instead !!

Attributions

```
/* 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**.

Commented out code

```
    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()));
```

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

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.

HTML in Comments

```
/**
 * 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>
 */
```

What does this comment say ?

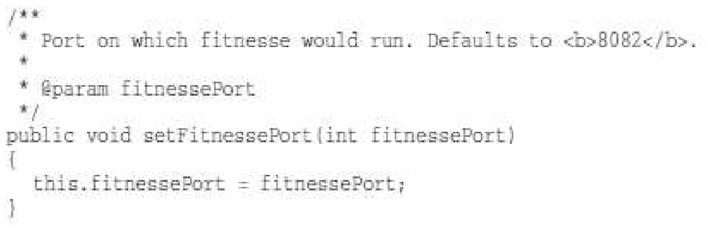
HTML in Comments

```
/**
 * 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>
 */
```

HTML in source code makes the comments **hard to read**.

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.

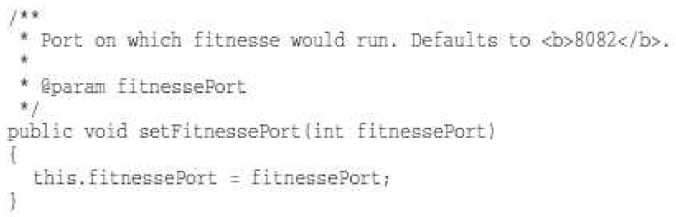
Non local information



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

How about this comment ??

Non local information

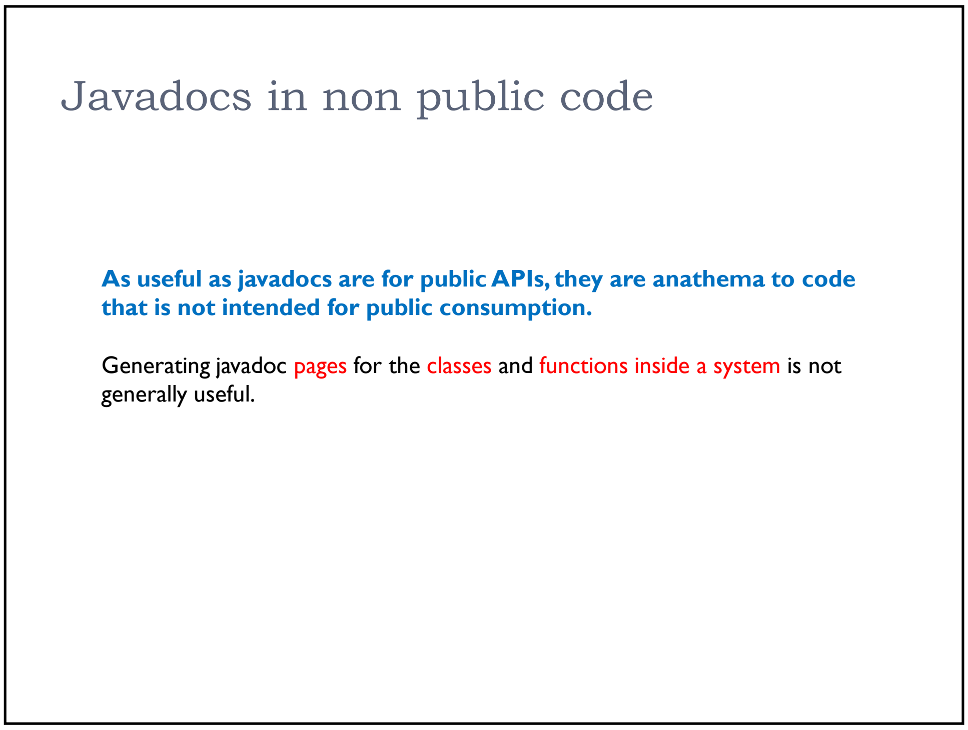


```
/**
 * Port on which fitnesse would run. Defaults to <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
  this.fitnessePort = fitnessePort;
}
```

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 port is changed !!!



Javadocs in non public 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.