CS 2340 Objects and Design

Clean Comments

Christopher Simpkins
chris.simpkins@gatech.edu

Clean Comments

Comments are (usually) evil.

- Most comments are compensation for failures to express ideas in code.
- Comments become baggage when chunkcs of code move.
- Comments become stale when code changes.
- Result: comments lie.

Comments don't make up for bad code. If you feel you need a comment to explain some code, put effort into improving the code, not authoring comments for it.

Good Names Can Obviate Comments

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

We're representing a business rule as a boolean expression and naming it in a comment. Use the language to express this idea:

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

Now if the business rule changes, we know exactly where to change the code that represents it, and the code can be reused. (What does "reused" mean?)
Another example:

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

Get rid of the comment and use the language:

```
protected abstract Responder responderBeingTested;
```

Good Comments

Sometimes comments are useful.

- Legal comments (copyright notices, licenses)
- Informative comments
- Explanation of intent
- Clarifications
- Warnings
- Todos
- Amplification
- Javadocs for public APIs

Informative Comments

Sometimes a comment can help the reader of code understand what code is supposed to do even if the code itself has a bug. Consider:

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

The regex format is hard to understand, but the comment makes it crystal clear. If the code has a bug, the programmer knows exactly how to go about fixing it. (How else might the programmer know?)

Explanation of Intent

```
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);
  }
  // we are greater because we are the right type.
  return 1;
}
```

This comment is acceptabe because it explains the programmer's intent: if the type of the other object is different, `this` object is greater

Explanation of Intent

Consider this test of some multithreaded code:

```
public void testConcurrentAddWidgets() throws Exception {
  // ...

  //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());
}
```

The comment quickly explains something that might be puzzling at first glance.

Clarification

If you're using code from the standard library or other code you can't change to express your idea in code, a clarifying comment can be helpful.

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

  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
  // ...
}
```

Be very careful though - it's easy to get the comments wrong, which substantially increases the cognitive burden on the reader trying to understand the code.

Warnings

Consider:

```
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;
}
```

On discovering this code you might be tempted to “optimize” it by making a single `SimpleDateFormat` to be shared. The comment would (hopefully) keep you from doing so.

Warnings

But don't use warning comments when you can express your idea in code. For example, instead of:

```
// 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);
}
```

Use Junit 4's annotations:

```
@Ignore("Takes too long to run").
public void _testWithReallyBigFile() {
// ...
```

Bad Comments

Most comments are bad. Most bad comments fall into these categories:

- Redundancies
- Misleading
- Noise

Redundant and Misleading Comments

Consider:

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

- Comment is redundant. Better to read the code or better yet a well-named method and parameter list.
- Comment is misleading. Method doesn't wait for `closed` to become `true` – it gives it `timeOUtMillis` to become `true` and throws an `Exception` if it doesn't.

Noise

- Journal comments added each time a file is modified.

```
* 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);
```

  Don't write journal comments. Use your VCS.
- Mandated comments, like Javadocs for code that's not part of a public API.
- Just plain dumb noise, like:

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

Position Markers and Brace Comments

Sometimes position markers can be useful, but be wary of commenting sections of code like this:

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

And never comment closing braces:

```
    public final E pop() {
        if (isEmpty()) {
            throw new java.util.EmptyStackException();
        } // end if
        return removeNext();
    } // end pop
```

Ending-brace comments are well-intentioned but redundant and risk becoming stale when method and class names change.

A few more commenting tips

- Don't put attributions in code, like `/* Added by Rick. */`. Your VCS handles this automatically. (Note, this is different from @author Javadoc tags.)
- Don't comment out code, delete it. Again, your VCS will remember old code for you.
- Don't put HTML markup in comments. If a tool like Javadoc turns comments into HTML, it's the tool's job to put in the HTML tags, not yours.
- Javadoc is usually overkill for code that's no part of a public API.
- Don't put nonlocal information in a comment. Everything you need to know to understand a comment should be within a couple of lines.
- Keep comments short.

Remember: comments make up for lack of expressivity in a programming language. You shouldn't need many, and you certainly don't need long comments.