Notes of chapter 04- Comments

“Don’t comment bad code—rewrite it.”
—Brian W. Kernighan and P. J. Plaugher1

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.
The proper use of comments is to compensate for our failure to express ourself 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.

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. Programmers can’t realistically maintain them. Inaccurate comments are far worse than no comments at all.

Truth can only be found in one place: the code.

Comments Do Not Make Up for Bad Code

One of the more common motivations for writing comments is bad code. Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.

Explain Yourself in Code

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

Or this?

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

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. the only truly good comment is the comment you found a way not to write.

Legal Comments

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

```
// 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.
```

Informative Comments:

It is sometimes useful to provide basic information with a comment.

e.g:

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

but it is better to use the name of the function to convey the information where possible.

Explanation of Intent

Sometimes a comment goes beyond just useful information about the implementation and
provides the intent behind a decision.

```
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.
}
```

Clarification:

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

```
assertTrue(a.compareTo(a) == 0); // a == a
assertTrue(a.compareTo(b) != 0); // a != b
assertTrue(ab.compareTo(ab) == 0); // ab == ab
```

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.
To a certain code:

```
// Don't run unless you
// have some time to kill.
```

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").`

TODO Comments:

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

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()));
```

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.

Mmbling:

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.
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:

```
// 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 does not justify the code, or provide intent or rationale. It is not easier to read than the code.

Misleading Comments:

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

Journal 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);
```

Long ago there was a good reason to create and maintain these log entries at the star 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:

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

No, *really?* Or how about this:

```
/** The day of the month. */
private int dayOfMonth;
```

These comments are so noisy that we learn to ignore them.
frustration could be resolved by improving the structure of his code.
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.

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

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))
```

Position Markers:

e.g:

```
// 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.

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.
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);
```

So commented-out code gathers like dregs at the bottom of a bad bottle of wine.

HTML Comments

HTML in source code comments is an abomination
It makes the comments hard to read

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.

Too Much Information:

Don't put interesting historical discussions or irrelevant descriptions of details into your comments.

Inobvious Connection:
The connection between a comment and the code it describes should be obvious. The purpose of a comment is to explain code that does not explain itself.

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.