MODULE

Commenting: how and why

FIT2099 staff
Updated 24 March 2017

Writing good comments

Why do we comment? Comments make it easier to fix the code, or to extend it by adding new functionality.

At the time you write your code, you may feel that it doesn't need any documentation at all — that it's self-documenting. Unfortunately, that feeling is seldom correct. The reason you find code easy to understand at the time you write it is that **you've already got a mental picture** of the way this part of the code works, how it fits together, and its relationship with other parts of the system. Other programmers won't necessarily have this inside knowledge about the code, and after you've moved on to other projects or other subsystems, you'll find your mental picture fades rapidly too.

Your code needs to help the reader to form that mental picture as efficiently and accurately as possible, so that they'll be able to fix or extend the code quickly and painlessly.

That brings us to our next question:

Who reads our comments?

It's always easiest to write well if you understand your audience: who will be reading your material? What do they already know, and what do you need to explain? This is true for any piece of technical writing, and commenting is no different.

The main audience for your code comments will always be other programmers. You can assume they are competent in the programming language, but you *can't* assume that they will be familiar with all the libraries that you're using. You definitely can't assume that they already understand the codebase you're documenting.

Even if you're completely sure that no other programmer will ever read the code, chances are that you will be an "other programmer" in six month's time.

Legacy code

Legacy code is code you inherit. Often, the original programmers have moved on and are not contactable. Often, the code is poorly documented. This kind of code will ruin your day.

Some day, if all goes well, your code will be legacy code.

Remember the software development lifecycle. Software doesn't cease to exist when you've finished writing it. A successful project will need to be maintained after implementation –to fix newly-discovered faults, to extend functionality, and to change the way it works.

Even if you did the best possible job of eliciting requirements, you're absolutely certain that you've understood and fulfilled them, and the users were thrilled with the product you handed over, software doesn't exist in a vacuum. Changing business conditions, regulatory environments, or even fashions in user interfaces may mean that your software needs to change.

That means that you need to document your code for the benefit of those developers who inherit it in the future.

What do your readers need?

In general, **code is written once but read many times**. That means that making your code easy to read and understand will save you and your colleagues time later on, helping keep your project on time and making budget overruns less likely.

The longer your code exists, the more likely it is that it will need to be changed some day. In the book *Code Simplicity*, author Max Kanat-Alexander presents some results he collected from his project's source control system. Some files had been modified *36 times* more frequently than had lines added!

When commenting, try to put yourselves in the shoes of a person who isn't familiar with the code, but who needs to extend it, fix it, or change the way it works. You can assume that this person is a competent programmer; otherwise they wouldn't be looking at the code in the first place. That means that, in principle, they *could* come to understand it well enough by just reading it. We write comments not to replace the reader's code-reading skills, but to **make this process faster**.

Commenting controversy

Commenting can be controversial. Some programmers, especially eXtreme Programming practitioners, have said that commenting is bad practice, because comments can easily become outdated. In his book *Refactoring: Improving the Design of Existing Code*, Martin Fowler describes excessive comments as a **code smell** – a sign that the code is a bit off and may need to be fixed.

Certainly, if you abide by the rule that inline comments should only be used if your code is complicated enough to need an explanation, there's some merit to this idea. If your code is more complicated than it ought to be, then arguably you should make it easier to read rather than spend time explaining it. Perhaps you can refactor the code to simplify it, or choose more descriptive names for variables and functions.

Limits of refactoring

Of course not all complexity can be refactored out of code. Perhaps the code is solving a really hard problem, such that using a complicated algorithm is unavoidable. Perhaps the code is complicated because it has had to be optimised for something other than readability in order to satisfy a nonfunctional requirement- for example, maybe it has been optimised to run fast, or to be secure. In those cases, refactoring to make the code easier to read would be a step backwards.

There's also the problem that not all complexity is worth refactoring out of the code. Your time isn't free, after all. If the job is likely to take a long time and the benefits to readability are likely to be marginal, it mightn't be the most effective use of your effort.

Good and bad comments

Good comments should:

- explain why the code is doing what it does
- give a brief outline of the purpose of each function or method, and how it should be called
- give a brief description of each module or class, and its role in the overall system
- point the reader to useful sources of information (e.g. references, language standards, etc)
- acknowledge any external external sources of code snippets, algorithms etc.
    - a nice thing to do from the point of view of intellectual property
    - also helps you keep your system up to date if the external source is updated
- be kept up to date

Bad commenting practice #1: stating the obvious

Here is an example of a *bad* inline comment:

```
1 | i++;     // add 1 to i
```

What's wrong with this piece of Java commenting? Well, it's certainly describing what the code does, but it doesn't tell you what `i` means or why it is being incremented. It tells you things that, if you're an even marginally competent Java programmer, you can see at a glance.

Novice programmers often write comments like this. When you are still learning the syntax of a programming language, and you're finding it confusing, it can be hard to remember that more experienced programmers *aren't* going to be confused by it.

Bad commenting practice #2: incorrect comments

Here's another bad practice that's unfortunately common:

```
1 | i += 5;  // add 1 to i
```

This kind of comment is not only unhelpful, but actively misleads the reader. Although it might have been correct in an earlier version of the code, it is no longer correct. If the reader tries to use it as a handy shortcut for understanding the code, they will end up thinking misunderstanding it – a sure-fire recipe for confusion and programmer error.

That's another reason it's a bad idea to overcomment. Your comments need to be maintained alongside the rest of the codebase, because even though they don't play a direct role in getting the system's functionality implemented, they (should) make life easier for the programmers whose responsibility it is to do this.

What to comment

Comments can appear anywhere in your code, but there are a few common kinds of comment that you are likely to be expected to use.

Block comments and inline comments

These kinds of comments appear *inside* a function or method and help explain what it's doing.

Inline comments are brief, and placed on the same line as the code they're documenting. They can be used to explain the purpose of lines of code:

```
if (s.startsWith("//"))
    continue;                // skip comment lines
```

They are also useful when explaining the purpose of variables or attributes:

```
private Control controller;   // Handles I/O and tape ops
```

Block comments appear above the code they're documenting. They're often used to explain a complex algorithm, but they can also be used to give a brief summary of what each "chunk" (or *stanza*) of code is for, to make it easier for the reader find specific parts of the code:

```
/* Check for duplicate transitions.
 * TODO: This should probably throw an Exception rather than
 * just printing to System.err.
 */
    for (Transition t: transitions)
        if (t.matches(s, i, w))
            System.err.println("Error: duplicate transition for " + state + " "
            + inputVal + " " + workVal);
```

Function and method comments

In a well-designed program, each function or method should do something that can be described fairly easily. Grouping lines of code together and then giving them a descriptive name makes it easier for programmers to use it, both you and others – you don't have to think about each line of code inside the function in order to be able to use it. That makes it easier.

Commenting your functions and methods makes using them easier still. Even somebody who is completely new to the code can easily see what they can do with it and how to use it. Function and method comments can also be used to explain how this fragment of code relates to the rest of the class or module.

As usual, you should consider what the reader of your comments needs to see. Imagine that *you* were handed a large chunk of unfamiliar source code and asked to find and fix some errors in it, and to extend its functionality in some way. Think about the information you'd want to see, and the use you would make of it.

As a rule of thumb, when commenting functions or methods, you should include *at least*:

- a brief description of what the function is for
- the parameters
- what the function returns
- any exceptions that the function throws
- any other side effects

Some companies or teams may have documentation standards that require more information than this.

Example:

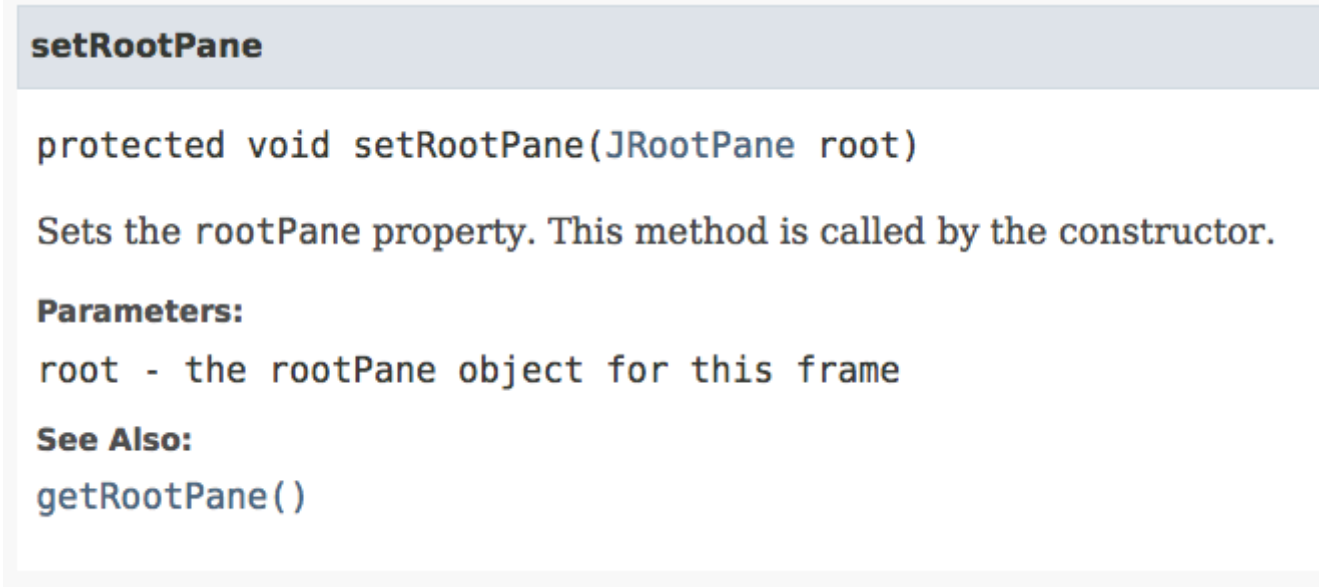
```
/** Write symbol to tape at head position and move the head right.

@param sym the symbol to write
@throws InvalidSymbolException if sym is null
@author ram
*/
public void write(Symbol sym) {
```

API documentation

A special case of method-level commenting is the documentation that you need to produce if you are writing an API (Application Programming Interface) that you are expecting other programmers to use. For example, you might be writing a library that allows third-party developers to write plugins for your system. Perhaps this library will be closed source, so other developers won't even have access to the source code – but even if they *can* read it, it doesn't follow that they'll *want to* read it.

Documenting an API can be tricky. At the time you're writing the documentation, you understand exactly how your library works, what each function or method does, both in the publicly-available interface and at the private level. It can be difficult to put yourself in the position of the third-party developer who does not have this information.

Here's an example of some API documentation that could probably be more useful:



What's wrong with this picture?

This is part of Oracle's online documentation for the `JFrame` class, which is part of the Swing GUI toolkit for Java. It was generated from specially-formatted comments inside the `JFrame` sourcecode. The comment is correct, properly formatted, and readable, but it opens up a big unanswered question: what's a `rootPane`? This comment is trivial. It doesn't help the reader understand how `JFrame` works.

It's not as bad as I'm making it appear, of course; there *is* documentation for the `rootPane` attribute. You just have to go digging for it. This API documentation would have been more helpful to the reader if the programmer had either put a brief explanation into the comment for this method, or put in a hyperlink to the `rootPane` documentation.

Class and module comments

If you're working on a complicated project, chances are that your code is broken up into chunks that contain related functionality and data. In an object-oriented language such as Java or C++, these will be classes, packages, or namespaces; in a hybrid language such as Python, they may be classes or modules; and in a relatively-unstructured language like as C they'll be files. Because these chunks of code contain (or *encapsulate*) functionality that's related, it makes sense to provide readers with documentation on what's available and how it all works.

Class or module comments go at the top of a class, module, or file, and tell you what it's for — what it contains, what it's used for, and how it relates to other structures within the codebase. Remember, the objective here is to *help other programmers understand how the code works as quickly as possible*, so that they can maintain or extend it. You don't need to duplicate the information found in a UML class diagram; if you feel that your project requires one of those, you should make one.

What to put into your class header comments

What you need to document depends to an extent upon what your toolchain is already doing for you. If you are using a clever development environment, such as Eclipse or Visual Studio, it'll be able to automate

some documentation tasks. It can show you a list of methods and attributes in a class, for example, with no need for you to

If you're *not* using such a development environment, it's a good idea to keep a list of components in the class header comments. This is tedious to maintain, though, so if your IDE can automate this task you should let it. Necessary comments are a lifesaver but superfluous comments are just clutter. Reserve your comments for the things that programmers can't easily find in their programming environment.

A good IDE can tell you the names of the components of your class, it can't tell you what they all do, how they're related, or how the class fits in with the rest of the system.

Your class, module, or file header comments should contain, at least:

- non-obvious relationships with other parts of the system
- explanations of design decisions that you might need to change later
- explanations of design decisions that readers might find questionable
- any information that a programmer needs to know in order to be able to use this code effectively

---