

Write Sweet-Smelling Comments

by Mike Clark

Ever since I was a wee programmer, I've been reminded that good code has a lot of comments. After all, code will be read many more times than it will be written. Thus, good programmers should take the time to write comments because someday, some lucky person will need to understand what our programs do in order to change them.

The trouble is, I can't ever remember being taught this important lesson: Learning to *remove* a comment can lead to improvements in the code. This was a craft technique that I didn't learn until I had tried to maintain code. When you're under that yoke, you live for the next comment. But too many comments can be worse than too few.

It turns out that good code actually needs fewer comments than does bad code. That may run counter to what you've been taught and may even sound a bit like heresy. So this month we'll build on the good naming skills we acquired last month by practicing the craft of commenting.

Keep It DRY

One sure-fire way to make a software program difficult to change is to use similar chunks of code in more than one place. If you've ever been the recipient of duplicated code, you won't soon forget it. It slows down even the best programmer because every change—fixing a bug, adding a feature, improving the design, or optimizing—means changing all the occurrences. Duplicated code is evil because when it becomes out of date, it drives up the cost of change.

The same holds true for duplication of knowledge between code and its comments. When they represent the same information, you can't change one without changing the other. And once you've changed the code, it seems like everything in the universe begins to work against your updating the comments. Just as you're about to rewrite the



Getty Images

comment, the guy down the hall drops by to recap how his favorite team disgraced yours. If that weren't bad enough, when his monologue ends you've forgotten what you were doing and end up with the following code:

```
/**
 * Returns the start date.
 */
public Date getEndDate()
```

This is a minor copy/paste slip-up, really. With any luck, someone will see the error of your ways and correct the comment. In the meantime, imagine you're the one interested in this particular method. Now, you're pretty sure the code is telling the truth, but how confident are you that the next comment you read won't be lying?

In general, if a comment already describes what the code does, then the comment is redundant and should be removed. Otherwise, you risk creating a misleading comment, which is worse than no comment at all. As recommended in *The Pragmatic Programmer* (Addison-

Wesley, 2000), strive to keep each piece of knowledge in one piece, as prescribed by the DRY (Don't Repeat Yourself) principle:

Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system.

Make the Code Speak for Itself

Comments are often good indications that the code can be improved. Martin Fowler refers to these types of comments as “deodorant” because they attempt to compensate for bad smells in the code. Remove the smell, and there's no longer any need for the deodorant.

Last month we practiced choosing good names that communicate the intent of code. Choosing meaningful names for classes, methods, and variables is one of the key craft techniques for making comments unnecessary. Consider, for example, the following comment/code pair:

```
/**
 * Authenticates the user.
 */
public void auth()
```

The `auth()` method certainly benefits from the comment. Without the comment, the method is meaningless. But rather than investing time and energy to write a good comment for the method, your time is better spent focusing on the real problem: a poorly named method. By renaming the method to `authenticateUser()`, the comment becomes superfluous. That's not to say you shouldn't add comments if the `authenticateUser()` method requires additional explanation. However, first try to put the knowledge into the code, then comment as necessary.

Comments tend to mask another code smell: long methods. How many lines of code does it take before a method is too long? It depends on how much work you're willing to put into understanding what the method does. One telltale sign of a long method is comments sprinkled throughout to help you find your way around. Take, for example, the following method:

```
public void authenticateUser() {
    if (!validPassword) {
        while (someCondition) {
            (inlined code here)
        } // end while
    } // end if
}
```

The inlined code has been left out to save your hairline, but you see the evidence of a long method in the use of the `end while` and `end if` comments. When control structures are so long or deeply nested that at the end they need a comment that refers back to the beginning, it's not the comment you smell—it's the code that needs to be improved. Extract the code inside the control structures into a well-named method so that understanding the enclosing method doesn't require navigational aids.

Long methods are easy to spot because they typically have a common form: a series of comments describing the purpose of a block of code that follows each comment. Here's an example:

```
public void authenticateUser() {
    // Find out which authentication method to use
    // If it's a network connection, authenticate the host
    // Prompt the user for a password
    // Check if the supplied password is valid
    // If the password is invalid, then sound the alarm
}
```

Again, assume that between each comment are multiple lines of code that expose more details of the program than you care about right now. Consider that authenticating a user is a fairly high-level concept. If you're reading this method, you abruptly transition from a high-level abstraction directly into low-level details. The comments break the details into logical sections, but you still have to parse through the minutia in order to understand how a user is authenticated.

That said, the comments document logical chunks of code, and the knowledge conveyed by those comments is good stuff. Writing comments like this can be a valuable exercise in that the process reveals good verb phrases to be used in method names. To help preserve the information so that it doesn't fall out of date with the code, simply extract each block of code into its own method—a method whose name documents what the code does:

```
public void authenticateUser() {
    findAuthenticationMethod();
    if (isNetworkConnection()) authenticateHost();
    String password = getPasswordFromUser();
    if (invalidPassword(password)) soundAlarm();
}
```

Extracting methods makes the `authenticateUser()` method shorter than the original version, but it's not necessarily the number of lines of code in the method that make it better. Notice how in the refactored `authenticateUser()` method the authentication process is expressed in code that reads like a domain-specific language. Each step in the process is documented in its own method. If the person trying to understand the code wants to know how a password is obtained

from the user, for example, she can peek inside the `getPasswordFromUser()` method. In the meantime, she isn't distracted by those details. Another benefit of extracting smaller methods from comments is that the methods can be reused across the system.

Any Integrated Development Environment (IDE) worth its salt supports this type of refactoring and makes it extremely safe. You highlight a block of code prefaced by comments, navigate a menu to the *Extract Method* refactoring, and type in the name of the new method. The new method is created, and the inlined code is automatically replaced with a call to the new method.

Write a Sweet-Smelling Comment

We've looked at instances where removing bad comments made the code more sweet smelling. Now let's turn our attention to cases where adding good

comments can make the code more maintainable.

Comments themselves are a sweet smell when they tell you why something was done. When you come across a comment while you're reading code, it should stop you dead in your tracks. The programmer before you felt so compelled to tell you something important—something that couldn't be documented anywhere else—that she left you a comment. Ignore it at your own peril.

The list of “eye-popping comments” is by no means an exhaustive list, but it gives a sense for valid reasons to write comments.

Good comments help you make the next change quickly; bad comments slow you down. Before writing a comment, try to make the code easy to understand so that you need fewer comments. Simple, safe refactorings—such as renaming classes, methods, and variables, and extracting well-named methods—go a long way toward removing duplication. Then, when all else fails, please leave us a good comment. Until next time . . . **{end}**

Mike Clark is a consultant, author, speaker, and programmer. He is the author of Pragmatic Project Automation (The Pragmatic Bookshelf, 2004), a frequent speaker at software development conferences, and the creator of several popular open source tools. Mike helps teams build better software faster through his company, Clarkware Consulting, Inc. (<http://clarkware.com>). Contact him at mike@clarkware.com.

Don't Stop Now!

Log on to **StickyMinds.com** and join Mike Clark and your peers in a conversation about this issue's topic. At the end of the digital column, add your views or just read what others have to say.

Want searchable access to every article ever published in **Better Software?**

The StickyMinds.com **PowerPass** gets you there.

Visit **StickyMinds.com/powerpass** to learn more.

A few eye-popping comments:

A comment at the top of every class tells you what purpose the class serves in the system:

```
# This class represents a single day. We could use
# the Data class, but it's fairly heavyweight and
# we'd need to add extra stuff anyway.
```

Gotchas, assumptions, limitations, or explanations of decisions that aren't obvious:

```
// You may be tempted to remove the read lock
// here, but don't because...
```

Police tape around code crime scenes lets others know we care about keeping code clean:

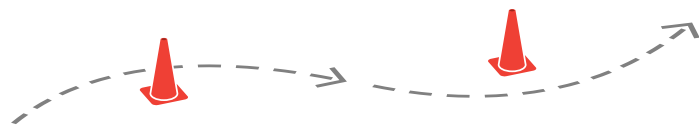
```
/*
 * TODO:
 * The following code needs to be refactored, and I'm
 * embarrassed to call it my own. However, we must
 * ship this code now! I promise to clean it up before
 * anyone starts thinking smelly code is acceptable.
 */
```

Use of a published algorithm:

```
/**
 * This cycle detection algorithm is documented in
 * 'Introduction to Algorithms' by Cormen, Leiserson,
 * and Rivest, MIT Press, Chapter 23.
 */
```

Adopting Agile Development? Steer Clear of the Roadblocks:

- **Poor visibility** slows your response to fast-changing customer needs
- **Can't easily synchronize** the day-to-day efforts of your distributed team
- Costly **IT headaches** with multiple systems tracking requests, requirements, tests, defects & tasks



Roll out Agile with Rally™ Software Development Management On Demand

Effective Agile project management combines high communication with low bureaucracy so your entire organization can embrace change and speed delivery of customer value.

Learn! Attend “The Road to Agile” webinar series and register for your FREE “instant-on” test drive at rallydev.com/bsm



RALLY
software development

deliver early, deliver often

www.rallydev.com/bsm