The Art of Readable Code

Dustin Boswell and Trevor Foucher

The Art of Readable Code
by Dustin Boswell and Trevor Foucher

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Code Should Be Easy to Understand

Over the past five years, we have collected hundreds of examples of "bad code" (much of it our own), and analyzed what made it bad, and what principles/techniques were used to make it better. What we noticed is that all of the principles stem from a single theme.

KEY IDEA

Code should be easy to understand.

We believe this is the most important guiding principle you can use when deciding how to write your code. Throughout the book, we'll show how to apply this principle to different aspects of your day-to-day coding. But before we begin, we'll elaborate on this principle and justify why it's so important.

What Makes Code "Better"?

Most programmers (including the authors) make programming decisions based on gut feel and intuition. We all know that code like this:

```
for (Node* node = list->head; node != NULL; node = node->next)
    Print(node->data);
```

is better than code like this:

```
Node* node = list->head;
if (node == NULL) return;

while (node->next != NULL) {
    Print(node->data);
    node = node->next;
}
if (node != NULL) Print(node->data);
```

(even though both examples behave exactly the same).

But a lot of times, it's a tougher choice. For example, is this code:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

better or worse than:

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

The first version is more compact, but the second version is less intimidating. Which criterion is more important? In general, how do you decide which way to code something?

The Fundamental Theorem of Readability

After studying many code examples like this, we came to the conclusion that there is one metric for readability that is more important than any other. It's so important that we call it "The Fundamental Theorem of Readability."

> KEY IDEA
> **Code should be written to minimize the time it would take for someone else to understand it.**

What do we mean by this? Quite literally, if you were to take a typical colleague of yours, and measure how much time it took him to read through your code and understand it, this "time-till-understanding" is the theoretical metric you want to minimize.

And when we say "understand," we have a very high bar for this word. For someone to *fully understand* your code, they should be able to make changes to it, spot bugs, and understand how it interacts with the rest of your code.

Now, you might be thinking, *Who cares if someone else can understand it? I'm the only one using the code!* Even if you're on a one-man project, it's worth pursuing this goal. That "someone else" might be *you* six months later, when your own code looks unfamiliar to you. And you never know—someone might join your project, or your "throwaway code" might get reused for another project.

Is Smaller Always Better?

Generally speaking, the less code you write to solve a problem, the better (see Chapter 13, *Writing Less Code*). It probably takes less time to understand a 2000-line class than a 5000-line class.

But fewer lines isn't always better! There are plenty of times when a one-line expression like:

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

takes more time to understand than if it were two lines:

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

Similarly, a comment can make you understand the code more quickly, even though it "adds code" to the file:

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```

So even though having fewer lines of code is a good goal, minimizing the time-till-understanding is an even better goal.

Does Time-Till-Understanding Conflict with Other Goals?

You might be thinking, *What about other constraints, like making code efficient, or well-architected, or easy to test, and so on? Don't these sometimes conflict with wanting to make code easy to understand?*

We've found that these other goals don't interfere much at all. Even in the realm of highly optimized code, there are still ways to make it highly readable as well. And making your code easy to understand often leads to code that is well architected and easy to test.

The rest of the book discusses how to apply "easy to read" in different circumstances. But remember, when in doubt, the Fundamental Theorem of Readability trumps any other rule or principle in this book. Also, some programmers have a compulsive need to fix any code that isn't perfectly factored. It's always important to step back and ask, *Is this code easy to understand?* If so, it's probably fine to move on to other code.

The Hard Part

Yes, it requires extra work to constantly think about whether an imaginary outsider would find your code easy to understand. Doing so requires turning on a part of your brain that might not have been on while coding before.

But if you adopt this goal (as we have), we're certain you will become a better coder, have fewer bugs, take more pride in your work, and produce code that everyone around you will love to use. So let's get started!

What NOT to Comment

Reading a comment takes time away from reading the actual code, and each comment takes up space on the screen. That is, it better be worth it. So where do you draw the line between a worthless comment and a good one?

All of the comments in this code are worthless:

```
// The class definition for Account
class Account {
  public:
    // Constructor
    Account();

    // Set the profit member to a new value
    void SetProfit(double profit);

    // Return the profit from this Account
    double GetProfit();
};
```

These comments are worthless because they don't provide any new information or help the reader understand the code better.

KEY IDEA

Don't comment on facts that can be derived quickly from the code itself.

The word "quickly" is an important distinction, though. Consider the comment for this Python code:

```
# remove everything after the second '*'
name = '*'.join(line.split('*')[:2])
```

Technically, this comment doesn't present any "new information" either. If you look at the code itself, you'll eventually figure out what it's doing. But for most programmers, reading the commented code is much faster than understanding the code without it.

Don't Comment Just for the Sake of Commenting

Some professors require their students to have a comment for each function in their homework code. As a result, some programmers feel guilty about leaving a function naked without comments and end up rewriting the function's name and arguments in sentence form:

```
// Find the Node in the given subtree, with the given name, using the given depth.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

This one falls into the "worthless comments" category—the function's declaration and the comment are virtually the same. This comment should be either removed or improved.

If you want to have a comment here, it might as well elaborate on more important details:

```
// Find a Node with the given 'name' or return NULL.
// If depth <= 0, only 'subtree' is inspected.
// If depth == N, only 'subtree' and N levels below are inspected.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

Don't Comment Bad Names—Fix the Names Instead

A comment shouldn't have to make up for a bad name. For example, here's an innocent-looking comment for a function named `CleanReply()`:

```
// Enforce limits on the Reply as stated in the Request,
// such as the number of items returned, or total byte size, etc.
void CleanReply(Request request, Reply reply);
```

Most of the comment is simply explaining what "clean" means. Instead, the phrase "enforce limits" should be moved into the function name:

```
// Make sure 'reply' meets the count/byte/etc. limits from the 'request'
void EnforceLimitsFromRequest(Request request, Reply reply);
```

This function name is more "self-documenting." A good name is better than a good comment because it will be seen everywhere the function is used.

Here is another example of a comment for a poorly named function:

```
// Releases the handle for this key. This doesn't modify the actual registry.
void DeleteRegistry(RegistryKey* key);
```

The name `DeleteRegistry()` sounds like a dangerous function (it *deletes* the registry?!). The comment "This doesn't modify the actual registry" is trying to clear up the confusion.

Instead, we could use a more self-documenting name like:

```
void ReleaseRegistryHandle(RegistryKey* key);
```

In general, you don't want "crutch comments"—comments that are trying to make up for the unreadability of the code. Coders often state this rule as **good code > bad code + good comments.**

Recording Your Thoughts

Now that you know what *not* to comment, let's discuss what *should* be commented (but often isn't).

A lot of good comments can come out of simply "recording your thoughts"—that is, the important thoughts you had as you were writing the code.

Include "Director Commentary"

Movies often have a "director commentary" track where the filmmakers give their insights and tell stories to help you understand how the film was made. Similarly, you should include comments to record valuable insights about the code.

Here's an example:

```
// Surprisingly, a binary tree was 40% faster than a hash table for this data.
// The cost of computing a hash was more than the left/right comparisons.
```

This comment teaches the reader something and stops any would-be optimizer from wasting their time.

Here's another example:

```
// This heuristic might miss a few words. That's OK; solving this 100% is hard.
```

Without this comment, the reader might think there's a bug and might waste time trying to come up with test cases that make it fail, or go off and try to fix the bug.

A comment can also explain why the code isn't in great shape:

```
// This class is getting messy. Maybe we should create a 'ResourceNode' subclass to
// help organize things.
```

This comment acknowledges that the code is messy but also encourages the next person to fix it (with specifics on how to get started). Without the comment, many readers would be intimidated by the messy code and afraid to touch it.

Comment the Flaws in Your Code

Code is constantly evolving and is bound to have flaws along the way. Don't be embarrassed to document those flaws. For example, noting when improvements should be made:

```
// TODO: use a faster algorithm
```

or when code is incomplete:

```
// TODO(dustin): handle other image formats besides JPEG
```

There are a number of markers that have become popular among programmers:

Marker	Typical meaning
`TODO:`	Stuff I haven't gotten around to yet
`FIXME:`	Known-broken code here
`HACK:`	Admittedly inelegant solution to a problem
`XXX:`	Danger! major problem here

Advertising Likely Pitfalls

When documenting a function or class, a good question to ask yourself is, *What is surprising about this code? How might it be misused?* Basically, you want to "think ahead" and anticipate the problems that people might run into when using your code.

For example, suppose you wrote a function that sends an email to a given user:

```
void SendEmail(string to, string subject, string body);
```

The implementation of this function involves connecting to an external email service, which might take up to a whole second, or possibly longer. Someone who is writing a web application might not realize this and mistakenly call this function while handling an HTTP request. (Doing this would cause their web application to "hang" if the email service is down.)

To prevent this likely mishap, you should comment on this "implementation detail":

```
// Calls an external service to deliver email.  (Times out after 1 minute.)
void SendEmail(string to, string subject, string body);
```

Here is another example: suppose you have a `FixBrokenHtml()` function that attempts to rewrite broken HTML by inserting missing closing tags and the like:

```
def FixBrokenHtml(html): ...
```

The function works great, except for the caveat that its running time blows up when there are deeply nested and unmatched tags. For nasty HTML inputs, this function could take *minutes* to execute.

Rather than let the user discover this later on his own, it's better to announce this upfront:

```
// Runtime is O(number_tags * average_tag_depth), so watch out for badly nested inputs.
def FixBrokenHtml(html): ...
```

"Big Picture" Comments

One of the hardest things for a new team member to understand is the "big picture"—how classes interact, how data flows through the whole system, and where the entry points are. The person who designed the system often forgets to comment about this stuff because he's so intimately involved with it.

Consider this thought experiment: **someone new just joined your team, she's sitting next to you, and you need to get her familiar with the codebase.**

As you're giving her a tour of the codebase, you might point out certain files or classes and say things like:

- "This is the glue code between our business logic and the database. None of the application code should use this directly."
- "This class looks complicated, but it's really just a smart cache. It doesn't know anything about the rest of the system."

After a minute of casual conversation, your new team member will know much more than she would from reading the source by herself.

This is exactly the type of information that should be included as high-level comments.

Here's a simple example of a file-level comment:

```
// This file contains helper functions that provide a more convenient interface to our
// file system. It handles file permissions and other nitty-gritty details.
```

Don't get overwhelmed by the thought that you have to write extensive, formal documentation. **A few well-chosen sentences are better than nothing at all.**

Summary Comments

Even deep inside a function, it's a good idea to comment on the "bigger picture." Here's an example of a comment that neatly summarizes the low-level code below it:

```
# Find all the items that customers purchased for themselves.
for customer_id in all_customers:
    for sale in all_sales[customer_id].sales:
        if sale.recipient == customer_id:
            ...
```

Without this comment, reading each line of code is a bit of a mystery. ("I see we're iterating through `all_customers` … but what for?")

It's especially helpful to have these summary comments in longer functions where there are a few large "chunks" inside:

```
def GenerateUserReport():
    # Acquire a lock for this user
    ...

    # Read user's info from the database
    ...

    # Write info to a file
    ...

    # Release the lock for this user
```

These comments also act as a bulleted summary of what the function does, so the reader can get the gist of what the function does before diving into details. (If these chunks are easily separable, you might just make them functions of their own. As we mentioned before, good code is better than bad code with good comments.)

SHOULD YOU COMMENT THE WHAT, THE WHY, OR THE HOW?

You may have heard advice like, "Comment the *why*, not the *what* (or the how)." Although catchy, we feel these statements are too simplistic and mean different things to different people.

Our advice is to do whatever helps the reader understand the code more easily. This may involve commenting the *what*, the *how*, or the *why* (or all three).

Final Thoughts—Getting Over Writer's Block

A lot of coders don't like to write comments because it feels like a lot of work to write a good one. When writers have this sort of "writer's block," the best solution is to just start writing. So the next time you're hesitating to write a comment, just go ahead and comment what you're thinking, however half-baked it may be.

For example, suppose you're working on a function and think to yourself, *Oh crap, this stuff will get tricky if there are ever duplicates in this list.* Just write that down:

```
// Oh crap, this stuff will get tricky if there are ever duplicates in this list.
```

See, was that so hard? It's actually not that bad of a comment—certainly better than nothing. The language is a little vague though. To fix it, we can just go through each phrase and replace it with something more specific:

- By "oh crap," you really mean "Careful: this is something to watch out for."
- By "this stuff," you mean "the code that's handling this input."
- By "will get tricky," you mean "will be hard to implement."

The new comment might be:

```
// Careful: this code doesn't handle duplicates in the list (because that's hard to do)
```

Notice that we've broken down the task of writing a comment into these simpler steps:

1. Write down whatever comment is on your mind.
2. Read the comment, and see what (if anything) needs to be improved.
3. Make improvements.

As you comment more often, you'll find that the quality of comments from step 1 gets better and better and eventually might not need fixing at all. And by commenting early and often, you avoid the unpleasant situation of needing to write a bunch of comments at the end.

Summary

The purpose of a comment is to help the reader know what the writer knew when writing the code. This whole chapter is about realizing all the not-so-obvious nuggets of information you have about the code and writing those down.

What *not* to comment:

- Facts that can be quickly derived from the code itself.
- "Crutch comments" that make up for bad code (such as a bad function name)—fix the code instead.

Thoughts you should be recording include:

- Insights about why code is one way and not another ("director commentary").
- Flaws in your code, by using markers like TODO: or XXX:.
- The "story" for how a constant got its value.

Put yourself in the reader's shoes:

- Anticipate which parts of your code will make readers say "Huh?" and comment those.
- Document any surprising behavior an average reader wouldn't expect.
- Use "big picture" comments at the file/class level to explain how all the pieces fit together.
- Summarize blocks of code with comments so that the reader doesn't get lost in the details.

Breaking Down Giant Expressions

The giant squid is an amazing and intelligent animal, but its near-perfect body design has one fatal flaw: it has a donut-shaped brain that wraps around its esophagus. So if it swallows too much food at once, it gets brain damage.

What does this have to do with code? Well, code that comes in "chunks" that are too big can have the same kind of effect. Recent research suggests that most of us can only think about three or four "things" at a time.* Simply put, the larger an expression of code is, the harder it will be to understand.

> KEY IDEA
>
> **Break down your giant expressions into more digestible pieces.**

In this chapter, we'll go through various ways you can manipulate and break down your code so that it's easier to swallow.

Explaining Variables

The simplest way to break down an expression is to introduce an extra variable that captures a smaller subexpression. This extra variable is sometimes called an "explaining variable" because it helps explain what the subexpression means.

Here is an example:

```
if line.split(':')[0].strip() == "root":
    ...
```

Here is the same code, now with an explaining variable:

```
username = line.split(':')[0].strip()
if username == "root":
    ...
```

Summary Variables

Even if an expression doesn't *need* explaining (because you can figure out what it means), it can still be useful to capture that expression in a new variable. We call this a *summary variable* if its purpose is simply to replace a larger chunk of code with a smaller name that can be managed and thought about more easily.

For example, consider the expressions in this code:

```
if (request.user.id == document.owner_id) {
    // user can edit this document...
}
```

* Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24, 97–185.

```
...

if (request.user.id != document.owner_id) {
    // document is read-only...
}
```

The expression `request.user.id == document.owner_id` may not seem that big, but it has five variables, so it takes a little extra time to think about.

The main concept in this code is, "Does the user own the document?" That concept can be stated more clearly by adding a summary variable:

```
final boolean user_owns_document = (request.user.id == document.owner_id);

if (user_owns_document) {
    // user can edit this document...
}

...

if (!user_owns_document) {
    // document is read-only...
}
```

It may not seem like much, but the statement `if (user_owns_document)` is a little easier to think about. Also, having `user_owns_document` defined at the top tells the reader upfront that "this is a concept we'll be referring to throughout this function."

Using De Morgan's Laws

If you ever took a course in circuits or logic, you might remember De Morgan's laws. They are two ways to rewrite a boolean expression into an equivalent one:

```
1)   not (a or b or c)    ⇔  (not a) and (not b) and (not c)
2)   not (a and b and c)  ⇔   (not a) or (not b) or (not c)
```

If you have trouble remembering these laws, a simple summary is "Distribute the `not` and switch `and`/`or`." (Or going the other way, you "factor out the `not`.")

You can sometimes use these laws to make a boolean expression more readable. For instance, if your code is:

```
if (!(file_exists && !is_protected)) Error("Sorry, could not read file.");
```

It can be rewritten to:

```
if (!file_exists || is_protected) Error("Sorry, could not read file.");
```