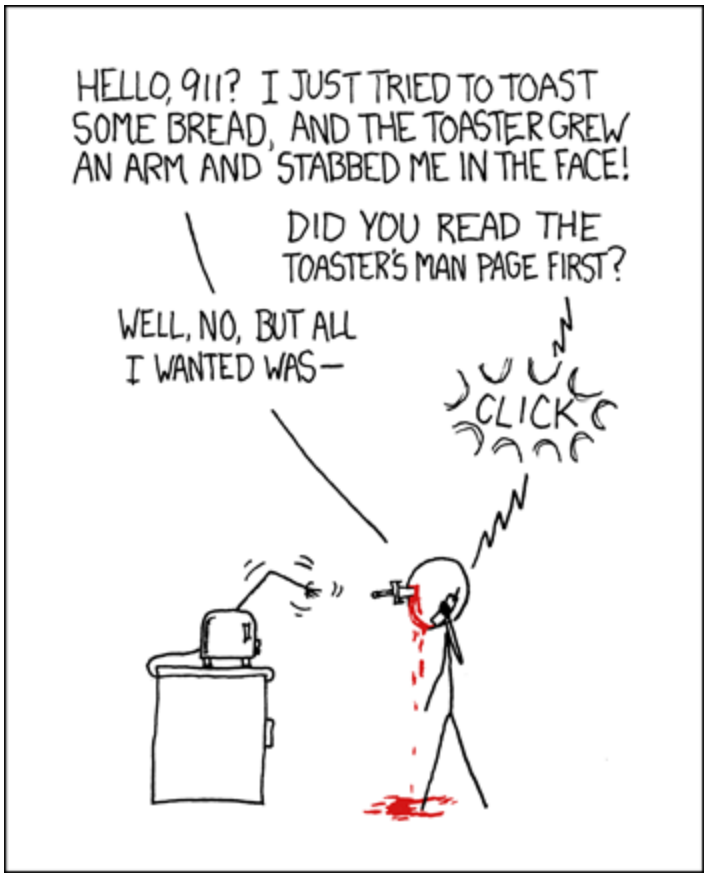
So you want a code review…

A Google-style code review is most likely unlike anything you've ever experienced for. Here is how to make it go as well as possible for everyone involved.

Read the style guide

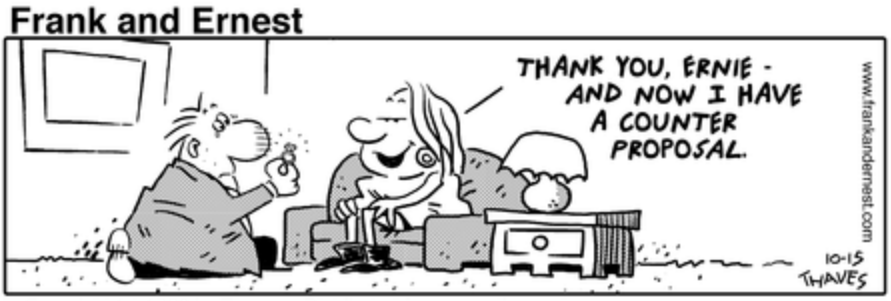


We generally follow these style guides:

- Android Code Style Rules
- Sun/Oracle Code Conventions [partly superseded by the former]
- Javadoc

If you want to ease your way through code reviews and delight your professors and professional colleagues with your code and documentation, read them. Skipping them basically says that you consider your time more valuable than the time and good will of your reviewers. Do you really want to insult your reviewers?

Start with a proposal



Before making major changes to a project, you should write a proposal or design document and get it approved by interested parties. After all, you wouldn't want to spend weeks coding up a feature, only to discover when you submit the code for review that there was a simple way to do it or that nobody wants that feature after all.

Pay attention to the little things

Here is a summary of the rules most often violated by new programmers:

- Line length is limited to 100 characters (except for package names, URLs, and Javadoc references).
- There should be a space on either side of a binary operator.
- There should be a space after a comma.
- Javadoc:
    - Javadoc should be created for new packages (package-info.java) and non-private classes and methods.
    - Return tags are optional on getters and other simple methods where the description for the return tag would be the same as the method summary.
    - Type information should not be included in descriptions of parameters or return values; it is generated automatically.
- Comments should be proof-read for grammar, spelling, punctuation, and clarity. If your English is not expert, ask someone local for help.
- The `@Override` annotation should be used when overriding a method or implementing an interface. (It is generally not necessary to write Javadoc for such methods.)

Before sending your code for its official review, view it using the same tools as the reviewer will use. This will help you find things you meant to take out, such as debugging code, as well as things you never meant to add, such as changes to spacing.

Keep it simple

Keep commits small. It's much easier to get a prompt review of a 40-line change than a 1000-line one, and you're likelier to save yourself work by catching mistakes sooner. You should rarely fix more than one issue in a single commit.

Also, keep your code simple. As Kernighan and Plaugh wrote in *The Elements of Programming Style* back in 1974:

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Get it in writing

Commit descriptions

Include a good commit description. The description is for the benefit of the reviewer, teammates, code archaeologists, and your future self. Here is some advice from StackOverflow by Jay Bazuzi:

- **First line** of the description is a short summary of the change. Many source control systems let you see a list of changes that show this line, so it gives you a rough summary.
- Include bug ID **and** bug title. Don't make people look it up! Make the Bug ID be a **URL** to open the bug, if your bug tracking supports it.
- Say **what you changed.**
- Say **why** you made this change, instead of taking some other approach.
- Making **each change small** makes it easier to follow the history + easier to write & read a good commit description.

Comments

In addition to the Javadoc requirements, you should also comment your code, although not excessively. The canonical bad comment is:

```
i = i + 1;   // Add one to i
```

Here are some good reasons for commenting:

Describing a block of code

```
// If only one account matches, use it and remember it for next time.
if (accounts.length == 1) {
```

```
  persistAccountName(accounts[0].name);
  return accounts[0];
}
```

Explaining a tricky or special case

```
final Object value = (new JSONTokener(jsonString)).nextValue();
// Note that the JSONTokener may return a value equals() to null.
if (value == null || value.equals(null)) {
  return null;
}
```

Providing a reference

```
// http://tools.ietf.org/html/rfc4180 suggests that CSV lines should be
// terminated by CRLF, hence the \r\n.
csvStringBuilder.append("\r\n");
```

Noting an imperfect or incomplete solution

```
// TODO(halabelson): The Java documentation warns that formatters are
// not thread-safe.  Is there any way that this can bite us?
```

Making invariants explicit, including in what cases code will be reached

```
// If we get here, rect1 has been mutated to hold the intersection of
// the two bounding boxes.
```

Explaining why a different approach was not used

```
// NOTE(lizlooney) - I tried just doing canvas.setBitmap(bitmap), but
// after that the canvas.drawCircle() method did not work correctly. So,
// we need to create a whole new canvas.
canvas = new android.graphics.Canvas(bitmap);
```

Providing historical information

```
/**
 * If False, don't show this component on the palette.  This was added
 * to support the Form/Screen component.
 */
@boolean showOnPalette() default true;
```

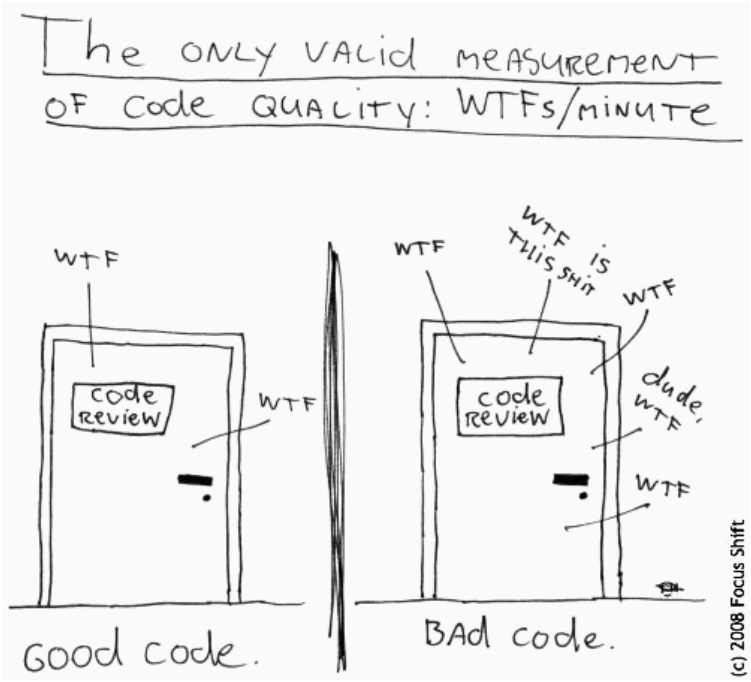
Test, test, test

Write unit tests. Unit tests are a great way to catch bugs early and to increase confidence in your code, and they reduce the likelihood of other people's making changes that break your code.

Before every iteration of submitting code for review, you should:

- Build the entire system.
- Run the tests and view the results.
- Build the Javadoc and view any changed portions in the browser. If possible, include the URL of the Javadoc in your review request.

Acronyms

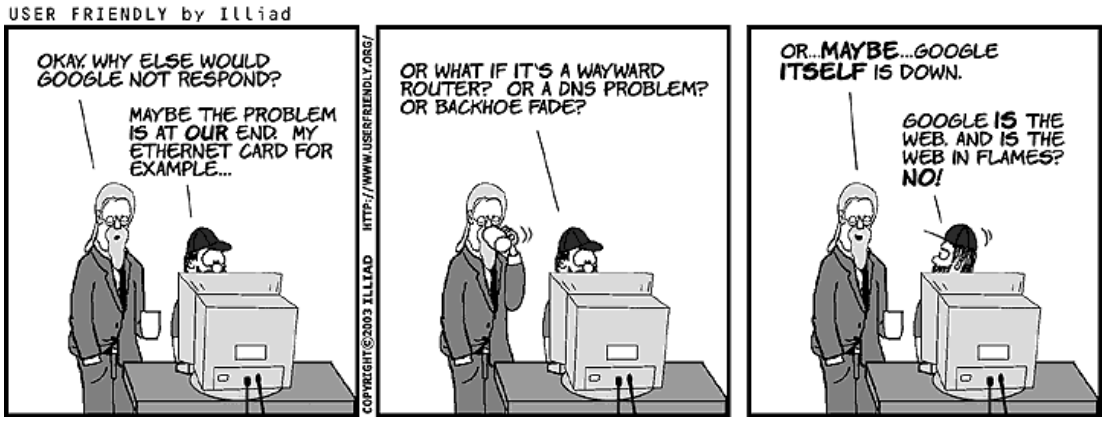


Some acronyms we use are:

- LGTM [looks good to me], which indicates approval.
- PTAL [please take another look], a message you might send to a reviewer after uploading improved code.

We do not actually use RTFM or WTF.

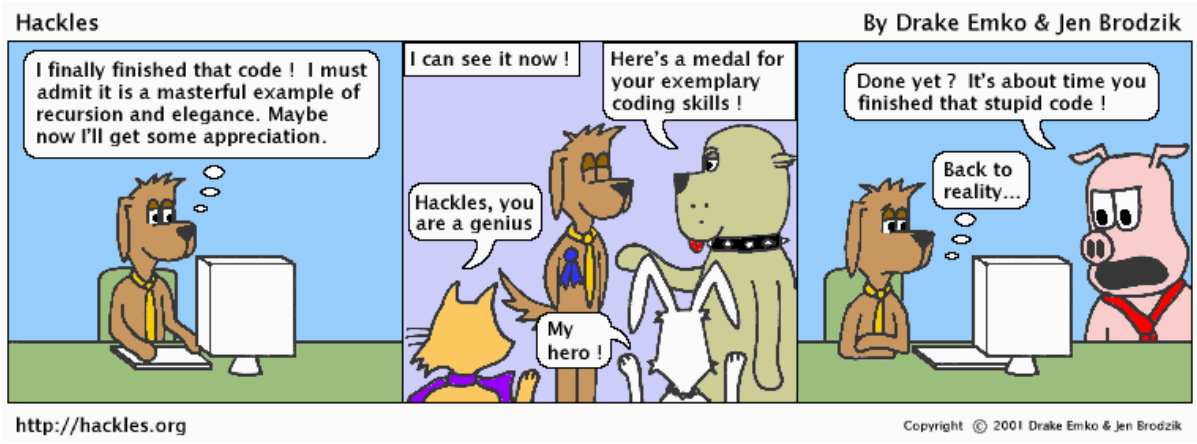
What's at stake



All programmers make mistakes. We can't prevent this. What we can influence is when these mistakes get discovered. Ideally they're discovered before being going into the repository. Worst case, they're pushed into production and bring down a Google data center (yes, this

happened to me) or make school children cry (yes, I've done this too, if you count college students).

Keeping perspective



I began working at Google when on sabbatical after having been granted tenure early at Mills College, where I had been working since becoming MIT[3] (SB, SM, and PhD). As a professor, I had been the sole judge of others' code. I expected to sail through my first code review and hear that they had never seen such excellent code before.

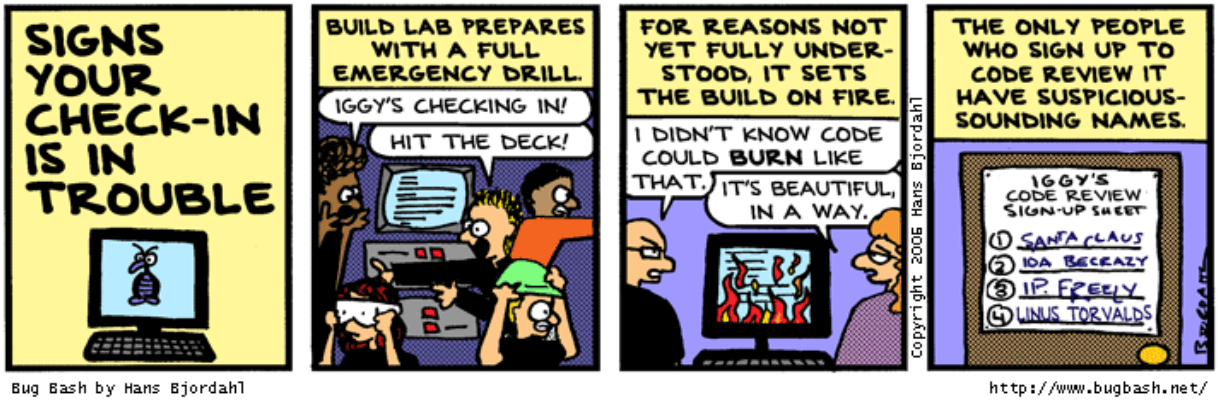
That's not what happened. To give you an idea of what did, here is the art that was created from my suffering, since reshared by many other Googlers:

The Five Stages of Dealing with a Code Review

1. Disbelief: "How could someone find so much to complain about in my code?!"
2. Anger: "How dare someone suggest that a hash-map of hash-maps isn't an efficient way of storing data?"
3. Bargaining: "Maybe if I add the comments the reviewer requested in one file, I won't have to fix the tricky Unicode issues in another."
4. Depression: "There was so much wrong with my code. How can I show my face to the person who reviewed my code?"
5. Acceptance: "Wow, my code is a lot better due to the feedback. This is a useful process, although I bet it's occasionally painful for everyone."

(For another take on the five stages, see this video.)

It's much better not to take suggestions personally.



Ellen Spertus
Last modified: February 1, 2014