ECE155: Engineering Design with Embedded Systems Winter 2016

Lecture 23 — Code Reviews

Patrick Lam & Jeff Zarnett

Reviews

A *review* is any activity where reviewers examine a work product to provide feedback. Reviews can reveal defects early, when the defects are less costly to fix. You can review anything, including: software requirements specifications, schedules, design documents, code, test plans, test cases, and defect reports.

Reviews can be either informal or formal.

- An *informal review* is a written or verbal review requested by a developer of a work product to improve that product.
- A *formal review* is a written review conducted by a team leader or a moderator for the purpose of identifying, documenting, and fixing defects in a work product.

Types of Reviews. Here are some types of reviews.

- A *desk check* is an informal review where the author distributes a work product to peers for reviews and comments.
- A *walkthrough* is an informal review meeting, moderated by the author of a work product.
- An *inspection* is a formal review meeting, guided by a moderator. The meeting produces a log of identified defects in a work product.
- A *code review* is a software inspection where defects in source code are identified, logged, and perhaps corrected.

It is useful to combine informal and formal reviews. Informal reviews help identify simple defects. Formal reviews can identify complex defects that might be missed by simple desk checks.

Desk Checks. A *desk check* is the first line of defence against defects. You can speed up formal inspections by taking care of simple defects in desk checks first. For many work products, desk checks suffice, and you might not need to go to a formal inspection.

However, desk checks are only effective if taken seriously. It's easy to just say "LGTM" (Looks Good To Me) without actually checking the product. It's important to spend enough time on desk checks, and managers must allocate time for them.

Walkthroughs. A *walkthrough* guides reviewers through the review of a work product. The author of the work product presents the design and ensures that the attendees understand its design.

Walkthroughs allow people with less expertise to review a work product, and users of the work product are often invited to walkthroughs. New points-of-view often help identify defects.

- Prior to the walkthrough, the author should distribute presentation materials to attendees.

- During the walkthrough, the author should solicit feedback from the audience.
- After the walkthrough, the author should follow up with attendees who have helped out by giving comments.

Formal Inspections. An *inspection* is a formal review meeting where participants identify and document defects or possible improvements in a work product. At an inspection, participants aim to identify and propose ideas for solving defects. Having a good mix of participants can help find previously-overlooked defects (perhaps subtle or complex ones).

Here are some steps for a formal inspection:

- *Preparation*: Before the meeting, distribute a printout of the work project to each member of the inspection team, along with a checklist indicating what to review.
- *Overview*: At the inspection meeting, the moderator provides an overview of the work product.
- *Page-by-page Review*: The moderator walks the inspection team through the work product page-by-page and logs defects.
- *Rework*: After the meeting, the author goes through the list of defects and fixes them.
- *Follow-up*: The inspection team members verify that the author has fixed the defects.
- *Approval*: The inspection team approves the work.

We do something similar for master's and PhD theses.

Code Reviews. A *code review* examines source code (or, more commonly, a patch) to identify defects or possible improvements.

Different projects have different review coverage; some projects (Mozilla) review everything, whereas companies might review only a representative sample. One can argue for sampling because developers tend to repeat the same mistakes, so that finding one issue will prompt developers to look for the same issue elsewhere.

When sampling, here are some places to look:

- source code that only one person has the expertise to maintain;
- tricky algorithms that are susceptible to defects;
- source code that calls difficult-to-use libraries;
- code written by inexperienced developers; and
- functions that could fail catastrophically if a defect is present.

In industry, you might have code review by a team; I don't know. However, in the open-source world (more below), you typically have 1 experienced developer doing a review. Generally, code reviews look for: clarity, maintainability, accuracy, reliability and robustness, security, scalability, reusability, and efficiency.

Pair Programming. We've talked about pair programming in the context of agile programming. Ideally, it can serve as instantaneous code review.

Code Reviews and University

One problem we clearly have at Waterloo, and other universities, is that we have the students do a lot of programming assignments but we do not review student code. We never sit down with students and take a look at the code and give feedback on variable names, on comments, on documentation (if appropriate). None of those items matter to the compiler or to the execution of the program, but they're important six months or a year down the road when someone else will need to read and understand that code. In the context of the university-assigned programming you will do, possibly the only case where a piece of code you write will be seen again more than four months later is in your Fourth-Year Design Project. The result is that students write throwaway code (not commented, not documented, hard to understand or debug) because there are no consequences for doing so and doing it the "right" way seems more difficult. This will fall flat in the real world [wP13]. Perhaps students get some code reviews done on co-op terms, but there is no guarantee of that.

This is something we are trying to address in the ECE 155 labs, by asking the TAs to review your code and give you feedback about things like your choice of variable name, comments, et cetera. Still, the most important issue is not really how the variables are named, but how the problem is decomposed.

Problem Decomposition

Problem decomposition is one of the most critical problem-solving skills. It means taking a big, complex problem, and breaking it down into a number of smaller problems that are easier to solve. Each of the subproblems can be broken down further as necessary until the subproblems are of small enough size that they can be easily implemented. If your starting problem is "write ATM software", you can break this down into a few subproblems: withdraw cash, deposit cash, and check balance. Each of those will need to be broken down into some series of other subproblems (like verify card and PIN).

When doing a code review of something larger than a minor patch, such as when a new feature is being added, it is important to consider how well the problem has been decomposed.

Good programmers decompose the problems well into subproblems that can be clearly described, independently implemented, and easily tested. Some documentation about what the parts do is often written in advance, but writing the rest of the documentation is simple because of the good structure [wP13].

Adequate programmers decompose problems reasonably, but they tend to have some awkward data structures which result in a lot of special-case code. The code seems to be "debugged" into existence and the documentation is written all at the end once things are more or less finished [wP13].

Poor programmers decompose the problem seemingly randomly, with unhelpful variable and method names like `x`, `foo`, or `doIt()`. Their code is often poorly tested and fails on boundary conditions. It sometimes appears that the code is "evolved" into existence: make random changes and see if that improves the output. Documentation, if it exists, is difficult to read or misleading [wP13].

Problem decomposition is a skill and the best way to improve is to practice it. Make the effort to define your subproblems well, choose appropriate variable names, and write (at least an outline of the) documentation early on. Proper decomposing of the problems might be valuable at university, in that you can complete assignments quicker and with fewer errors. Things like variable names might not make much of a difference in assignments, but it will get you into the correct habits for later – and just might impress your employers!

Reviews for Open-Source Software Projects

We've mentioned that open-source projects have an official repository. These projects also have a set of committers, who may commit changes to the repository. Outside contributors sometimes send patches

to a project, adding new features or fixing bugs; in that case, a committer will typically review the patch before committing it. Committers may also seek review for their patches if they want someone else to take a look at it.

Case Study: Mozilla Review Policy. The Mozilla Project develops the Firefox web browser (and others projects). Due to the size of the codebase, they have an elaborate reviewing policy [Moz12, Net13]: they require at least one review ("owner/peer review") for all patches, and a second review ("super-review") for many of the patches as well.

The owner/peer review is by a domain expert who understands the code being modified and the implications of the change. In particular, "A review is focused on a patch's design, implementation, usefulness in fixing a stated problem, and fit within its module." From the Code Review FAQ, reviews check for: whether the patch fixes a problem; the API/design; maintainability; security; integration; testing; and license compliance.

Super-reviews are by "strong hackers", who understand the way Mozilla code is supposed to look, but need not have domain expertise. In particular (per the Code Review FAQ), they are supposed to look out for: proper use of APIs; adherence to Mozilla's portability guidelines; cross-module effects; and respect of Mozilla coding practices.

Other organizations. Google and the Linux kernel both review code extensively. You can also read about Gerrit [Mar09], a tool out of Google for code reviews.

Fog Creek Code Review Checklist

Here is a Code Review Checklist from Fog Creek, which can serve as the basis of your code review checklist. This is taken directly from [Wil15].

General

- Does the code work? Does it perform its intended function, the logic is correct etc.
- Is all the code easily understood?
- Does it conform to your agreed coding conventions? These will usually cover location of braces, variable and function names, line length, indentations, formatting, and comments.
- Is there any redundant or duplicate code?
- Is the code as modular as possible?
- Can any global variables be replaced?
- Is there any commented out code?
- Do loops have a set length and correct termination conditions?
- Can any of the code be replaced with library functions?
- Can any logging or debugging code be removed?

Security

- Are all data inputs checked (for the correct type, length, format, and range) and encoded?
- Where third-party utilities are used, are returning errors being caught?

- Are output values checked and encoded?
- Are invalid parameter values handled?

Documentation

- Do comments exist and describe the intent of the code?
- Are all functions commented?
- Is any unusual behavior or edge-case handling described?
- Is the use and function of third-party libraries documented?
- Are data structures and units of measurement explained?
- Is there any incomplete code? If so, should it be removed or flagged with a suitable marker like `TODO`?

Testing

- Is the code testable? i.e. don't add too many or hide dependencies, unable to initialize objects, test frameworks can use methods etc.
- Do tests exist and are they comprehensive? i.e. has at least your agreed on code coverage.
- Do unit tests actually test that the code is performing the intended functionality?
- Are arrays checked for "out-of-bound" errors?
- Could any test code be replaced with the use of an existing API?

References

[Mar09] Don Marti. Gerrit: Google-style code review meets git, 2009. Online; accessed 21-December-2013. URL: `http://lwn.net/Articles/359489/`.

[Moz12] Mozilla.org. Super-Review Policy, 2012. Online; accessed 21-December-2013. URL: `http://www.mozilla.org/hacking/reviewers.html`.

[Net13] Mozilla Developer Network. Code Review FAQ, 2013. Online; accessed 21-December-2013. URL: `https://developer.mozilla.org/en/Code_Review_FAQ`.

[Wil15] Gareth Wilson. Stop more bugs with our code review checklist. `http://blog.fogcreek.com/increase-defect-detection-with-our-code-review-checklist-example/`, 2015. Online; accessed 09-January-2015.

[wP13] Gas Station without Pumps. Critiquing code, 2013. Online; accessed 12-December-2013. URL: `http://gasstationwithoutpumps.wordpress.com/2013/11/08/critiquing-code/`.