Lecture 23 — Code Reviews

Patrick Lam & Jeff Zarnett
p.lam@ece.uwaterloo.ca & jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

January 9, 2015

Part I

Reviews

On Reviews

Review:
activity where reviewers examine a work product to provide feedback.

Advantage:
reveal defects early—defects less costly to fix.

What to review:
requirements specifications; schedules; bug reports; design documents; code; test plans; test cases.

Informal vs Formal

- *informal review*: written or verbal review requested by a developer of a work product.
- *formal review*: written review conducted by a team leader or a moderator to identify, document, and fix defects in a work product.

Types of Reviews

- desk check: informal review;
  author distributes work to peers for reviews and comments.
- walkthrough: informal review meeting;
  moderated by the author.
- inspection: formal review meeting;
  guided by a moderator.
  Produces a log of identified defects in a work product.
- code review: software inspection
  identifying, logging, and perhaps correcting bugs.

Desk Checks

Desk check: first line of defence against defects.

- can speed up formal inspections
  by taking care of simple defects in desk checks first.
- for many work products, desk checks suffice;
  often don't need a formal inspection.

But:

- only effective if taken seriously.
- 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

Walkthrough: guided review of a work product.

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

Author of the work product presents the design and ensures that the attendees understand its design.

How a walkthrough works:

- Before: distribute presentation materials.
- During: solicit feedback from the audience.
- After: follow up with attendees who have helped out by giving comments.

Inspection

Inspection: formal review meeting where participants identify and document defects or possible improvements.

Participants identify, and propose solutions to, defects.

A good mix of participants (but not too many!) helps find previously-overlooked defects (subtle, complex bugs).

Steps in a formal inspection

- Preparation: Before the meeting, distribute the work project to each member of the inspection team, plus a checklist indicating what to review.
- Overview: Moderator provides an overview of the item.
- Page-by-page Review: Moderator walks the inspection team through the work product and logs defects.
- Rework: Afterwards, the author goes through the list of defects and fixes them.
- Follow-up: Inspection team members verify that the author has fixed the defects.
- Approval: Inspection team approves the work.

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

Code Review

Code review: examines source code (usually a patch) to identify defects or possible improvements.

Coverage options:

- review everything (Mozilla); or,
- review a representative sample.

Representative sample:

developers tend to repeat the same mistakes.
If you find one bug, look for similar ones nearby.

Code Review (sampling)

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.

Code Review (who? what?)

In industry: by team?

Open-source world: 1 or 2 experienced developers, independently.

What to look for:

- clarity, maintainability, accuracy, reliability,
- robustness, security, scalability, reusability, efficiency.

Pair Programming

Part of Extreme Programming.

Can also serve as instant code review.

Part II

Reviews at University

Reviews at University

We have the students do a lot of programming assignments...
but we do not review student code.

We do not give feedback on variable names, comments, etc.

Irrelevant to the compiler & execution,
but important when someone (else) will need to read it.

Reviews at University

UW projects are, at most, 4 months long.
Possible exception: 4^{th} year design project.

There are no consequences for writing throwaway code.

Maybe on co-op terms, but how much does it happen?

Trying to ask TAs to do code reviews in ECE 155 labs.

Problem Decomposition

Key thing to look for in code reviews: problem decomposition.

Take a big, complex problem, break it down into a number of smaller problems that are easier to solve.

Each subproblem can be broken down further if necessary.

Problem Decomposition

If your starting problem is “write ATM software”:

Subproblems: Withdraw cash, deposit cash, check balance.

Each of those will need to be broken down into some series of other subproblems (like verify card and PIN).

Reviewing Problem Decomposition

Good programmers decompose the problems well into subproblems that can be:

- clearly described,
- independently implemented, and
- easily tested.

Documentation is often written in advance, but writing the documentation is simple because of the good structure.

Reviewing Problem Decomposition

Adequate programmers decompose problems reasonably.

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.

Documentation written all at the end once things are finished.

Reviewing Problem Decomposition

Poor programmers decompose problems seemingly randomly.

Unhelpful variable and method names like `x`, `foo`, or `doIt()`.

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.

Reviewing Problem Decomposition

Problem decomposition is a skill; improve by practicing.

Define your subproblems well, choose appropriate variable names, and write an outline of documentation early on.

Reviewing Problem Decomposition

Proper decomposing of the problems is valuable at UW:
you can complete assignments quicker and with fewer errors.

Variable names might not make a difference in assignments.

It will get you into the correct habits for later
and just might impress your employer!

Part III

Reviews for Open-Source Projects

How Open-Source Projects Work

Typically:

- an official repository. (SVN, Git, Mercurial)
- a set of committers, who may commit changes.

Outside contributors: may send patches
(bug fixes, new features).

- a committer reviews the patch before committing it.

Committers may/must also seek review for their patches.

Case Study: Reviews at Mozilla

Mozilla Foundation: develops the Firefox web browser (and other projects).

Case Study: Reviews at Mozilla

Huge codebase $\Rightarrow$ elaborate reviewing policy[1].

- require at least one review ("owner/peer review") for all patches, plus
- second review ("super-review") for many patches.

[1] `http://www.mozilla.org/hacking/reviewers.html`, `https://developer.mozilla.org/en/Code_Review_FAQ`

Case Study: Peer review at Mozilla

Owner/peer review: by a domain expert who understands the code being modified and the implications of the change.

> *A review is focused on a patch's design, implementation, usefulness in fixing a stated problem, and fit within its module.*

Reviews check for:

- whether the patch fixes a problem;
- API/design;
- maintainability;
- security;
- integration;
- testing; and
- license compliance.

Case Study: Super-reviews at Mozilla

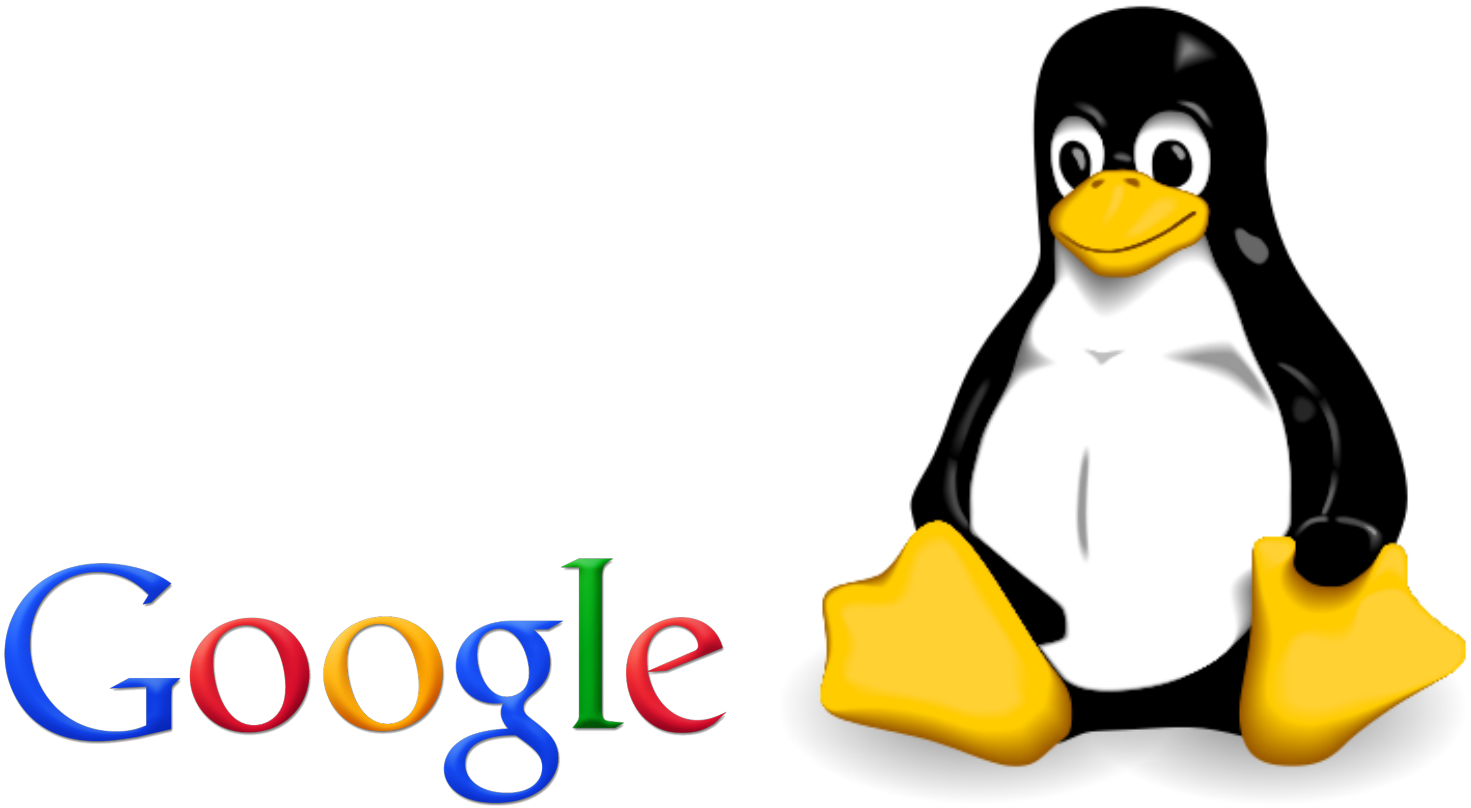
Super-reviews by “strong hackers”.

- understand the way Mozilla code is supposed to look,
- need not have domain expertise.

They look out for:

- proper use of APIs;
- adherence to Mozilla’s portability guidelines;
- cross-module effects; and
- respect of Mozilla coding practices.

Other organizations



Many organizations, including Google and Linux kernel hackers, review extensively.
Gerrit[2]: a tool out of Google for code reviews.

[2] `http://lwn.net/Articles/359489/`

Fog Creek Checklist

Here is a Code Review Checklist from Fog Creek.

This can serve as the basis of your code review checklist.

Fog Creek Checklist: General

- Does the code work?
- Is all the code easily understood?
- Does it conform to your agreed coding conventions?
- 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?

Fog Creek Checklist: Security

- Are all data inputs checked and encoded?
- Where third-party utilities are used, are returning errors being caught?
- Are output values checked and encoded?
- Are invalid parameter values handled?

Fog Creek Checklist: 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`?

Fog Creek Checklist: Testing

- Is the code testable?
- Do tests exist and are they comprehensive?
- 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?