CITS5501 Software Testing and Quality Assurance
Software reviews

Arran Stewart and Rachel Cardell-Oliver 2025

Outline

- Manual Code Reviews
- Automated Static Analysis
- Code Metrics

Manual Reviews

Motivation for Manual Reviews

Software reviews

- We use **review** as a catch-all term for manually conducted assessments that can be applied to any static software artifact – from requirements or specification documents, to source code, to use case descriptions, to test plans.
- If the review is not manual, but automated, we usually instead call that **static analysis**.
- Both these techniques are usually distinguished from testing
  - *Testing* involves actually running software, in order to observe its properties
  - It's therefore a form of *dynamic* analysis.

Types of reviews

Reviews vary in the amount of preparation, formality, and rigour applied to them.

- **Code review** on its own usually means a review by one other person – varying in the level of formality, and in whether a specified checklist/criteria are used
- **Code walkthroughs** are done synchronously by the developer and at least one reviewer;
  - Usually informal
  - The developer leads the review team through their code and the reviewers try to identify faults
- **Code inspections** are fairly formal – they are a detailed, step-by-step group review of a work product, with each step checked against predetermined criteria. They require preparation and follow-up.

Types of reviews, cont'd

- **Audits** are usually performed by an independent party, not the development team
  - This could be a QA or testing department, or could be an outside agency.
  - Although it can result in defects being identified, the main focus of an audit is on whether the system conforms to some standard.

Why do reviews?

- Because they're very effective, and much cheaper than finding defects via testing.

Comparative cost of reviews

- From one study:[1] correcting defects found by testing was 14.5 times the cost to find the problem in an inspection
  - This grew to 68 times the inspection cost if the defect was reported by a customer.
- From a study based on work at IBM: correcting defects found in a released product was 45 times the cost if the defect was fixed at design time.

[1] The figures cited here are from Jorgensen (2013), citing earlier work by Karl Weigers.

Effectiveness of reviews

- **Informal reviews, unit and regression tests** have a fairly low rate of detection ($\leq 35\%$)
- **High-volume beta testing** has a high rate of detection (around 75%) – but unfortunately, it occurs at the very end of the software development lifecycle, when defects are **most costly to remove**.
- **Formal inspections of design or code** have a detection rate of 55–60%.

https://kevin.burke.dev/kevin/the-best-ways-to-find-bugs-in-your-code/ quoting McConnell (2004), Code Complete, ch 20, table 20-2, citing earlier work by Capers Jones and Shull et al. Available online from UWA liibrary

Effectiveness of reviews (cont)

- Although many of these techniques have only a low rate of detection in isolation, McConnell (2004) points to research suggesting that using a wide variety of techniques in combination can result in detection rates of 95%.
- Many organizations today rely on only testing and informal code reviews – many defects are therefore being missed at early stages of development, and only corrected at late stages or after release (when the cost of doing so is much higher)

Benefits of reviews

Besides the fact that they can help detect defects, reviews have other benefits:

- Communication and knowledge transfer: reviews ensure that knowledge about code and design are shared amongst multiple members of a team
- Training: having code reviewed can be a useful part of training for new personnel
- Skill improvement: reviewees can benefit from others' suggestions; reviewers can benefit from techniques or approaches they may not have seen before.

Manual Review Techniques

Comparison of different review techniques

In ascending level of formality/preparation required:

- Code review
- Code walkthrough
- Code inspection

Code reviews

- Popular in many organizations
- Fairly cheap to do – just get another developer to look at code before it is merged into version control
- But if done without rigour, is also the least effective form of review
- Reviewers may have a checklist of things to look for.

Checklists

- A set of questions to stimulate critical appraisal of all aspects of the system
- Questions are usually general in nature and thus applicable to many types of system
  - But an organization may also have checklists/best practices that should be applied to a particular language or type of system

Code inspection

- Sometimes called a “Fagan inspection”; the term “code inspection” was introduced by Michael E Fagan.
- More formal version of a code walk-through
- Procedure:
  1. Overview
  2. Preparation
  3. Inspection
  4. Rework
  5. Follow up
- Meetings are chaired by a team moderator rather than the programmer

Code Review best practices

For an exhaustive discussion of best practices, the book by Cohen et al *Best Kept Secrets of Peer Code Review* is a good guide.

But we outline several best practices here.

Don't waste reviewers' time

- Don't waste reviewers' time doing things that could have been done by the original developer or automated software
- Original developer should have already ensured their code meets organizational standards, has been formatted for readability – reviewers shouldn't be doing the developers' job for them
- It's a waste of time for reveiwers to detect bugs or code formatting issues that could've been picked up automatically – code beautifiers/formatters and linters/static analyses should already have been run over the code

Review best practices

Do provide reviewer instructions and/or checklists

- Empirical research suggests[2] that reviews are more effective when reviewers are provided with checklists *or* other guides to what sort of problems they should be looking for or how the new code will be used.
- A checklist might feature such problems as e.g. code not organised logically, insufficient documentation, lack of tests, poor readability of code, repetitive code

Do ensure review requests include context

- Requests for review should clearly explain to the reviewer what has changed in the code, for what reason (e.g. provide a link to the relevant bug reports), and whether the new code poses increased risks.

[2] See e.g. Dunsmore et al (2000), cited in Cohen et al (2013).

Review best practices, cont'd

Do capture issues that can't be corrected immediately

- Reviewers may pick up issues or make suggestions that can't be fixed/implemented for the current release – but they should be captured for future use.
- (An easy way to do this is to add them to the organization's issue-tracking system.)

Do document the results of reviews

- All comments made, defects identified, etc should be recorded
- For instance via email (acceptable but not ideal for searching) or in an issue tracking system (much more useful)
- If it turns out reviewers are consistently identifying the same sorts of problems – can those problems be detected automatically?

Example – getNumOfDays

In labs, you will do code reviews of your own, for instance of a `getNumOfDays()` method (to calculate number of days in a given month of a given year).

```
if (year<1) {
  throw new YearOutOfBounds(year);
}

if (month==1 || month==3 || month==5 || month=7 || month==10
    || month==12) {
   numDays = 32;
} else if (month==4 || month==6 || month==9 || month==11) {
  numDays = 30;
} else if (month==2) {
  if (isLeapYear(year)) { numDays = 29;
  }
// ...
```

Static analysis

Static analysis

- **Static analysis** means the automated analysis of static software artifacts, in order to detect defects or identify other properties of the system.
  - e.g. "This program P never dereferences a null pointer"
- It runs the gamut from very very simple techniques (e.g. `grep`ping code for functions that are known to be unsafe or prone to misuse), to very complex.

Dynamic analysis

We contrast static analysis, which operates on static artifacts, with dynamic analysis, which runs actual (usually instrumented) code.

- Identifying branch coverage of tests is a dynamic analysis technique
- Other dynamic techniques include code sanitizers, which detect memory, concurrency and other issues at runtime.
- Advantages:
  - often more precise than static analysis
- Disadvantages:
  - need to ensure code where defects are located is actually run; whereas static analysis can have "perfect coverage" (since the whole of the source code is available)
  - may require code to be instrumented, and therefore recompiled
  - normally slower than static analysis, since code has to actually be run.

Static analysis limitations

We said that static analysis tools analyse source code to determine whether the program has some property P

- e.g. "Never results in a `ClassCastException`"

It is impossible to write a tool which detects any non-trivial property of a program perfectly (no false positives, no false negatives) – this is Rice's Theorem.

Therefore, all tools in practice are imperfect in some way.

They *approximate* the behaviour of the program: they provide either false positives or false negatives.

Terminology

False positive Reporting a program has some property when it does not

False negative Reporting a program does not have some property when it does

Static analysis limitations

If our focus is on identifying problematic properties, we will consider

- *false positives* to be cases where a problem is detected (but actually cannot occur)
- *false negatives* to be cases where a problem will occur, but is not detected.

Normally, we'd prefer to err on the side of having false positives.

Types of static analysis program

Compilers

Amongst other things, aim to detect violations of type safety rules

Style checkers/linters

Check conformance with style rules

Bug finders

Look for known bugs, and/or code practices that are known to be unsafe

Verifiers

Prove the absence of runtime errors of various sorts

Style Checkers

Style checking covers *good practice* for a language

Usually covers

- coding standards (layout, bracketing)
- naming conventions (e.g. `snake_case`, `camelCase`, `SCREAMING_SNAKE_CASE`)
- checking for dubious code constructs (e.g. in Python, use of `eval()`)
  - it therefore has some overlap with bug finders

Example style checking Tools

- Checkstyle (Java)
- ShellCheck (Bash)
- clang `clang-format`, `clang-tidy` (C and C++) (slow link)
- pylint, black (Python)

Bug Finders

Focus is on detecting code constructs known to be problematic.

Java examples:

- FindBugs
- PMD. Cross-language static code analyser

PMD has many capabilities, and can be augmented with custom rules.

Code Metrics

Code Metrics

- Measures of properties of code
  - Usually fairly "low level" properties, when compared with static analysis
  - But the boundary is blurry
- Examples:
  - graph theoretic complexity (of the program's control graph)
  - module accessibility (how many ways a module may be accessed)
  - number of entry and exit points per module
- Some of these metrics may correlate with the quality of the code, or how likely it is to contain errors

Code Smells

Code Smells are not bugs but are significant weaknesses in design that might increase the risk of failure in the future

PMD and Checkstyle (see above) can both check for common code smells

Recommendation: Run regular checks for code smells and refactor your code to avoid them

- Example of Code Smells
  - Comments (huh?? find out why - see link above)
  - Long Method / Long Parameter List
  - Dead Code
  - Data Classes (class stores only data, no methods)
  - Alternative Classes with Different Interfaces

Summary

Summary of Topics in this Lecture

- Manual Code Reviews
- Automated Static Analysis
- Code Metrics

References

- Cohen, J., Brown, E., DuRette, B., & Teleki, S. *Best Kept Secrets of Peer Code Review*. Austin, Tex.: Smart Bear, 2013.
- Best Practices for Code Review SmartBear website
- Dunsmore, A., Roper, M., & Wood, M. "Object-Oriented Inspection in the Face of Delocalisation." In *Proceedings of the 22nd International Conference on Software Engineering* (ICSE '00), 467–476. Limerick, Ireland: ACM Press, 2000. Available at https://doi.org/10.1145/337180.337343.
- Jorgensen, Paul C. *Software Testing: A Craftsman's Approach*. 4th edition. Boca Raton, Florida: Auerbach Publications, 2013.
- McConnell, Steve. *Code Complete*. 2nd edition. Redmond, Washington: Microsoft Press, 2004.