THE UNIVERSITY OF MELBOURNE

IT Capstone Project

COMP 30022 2025

Week 4 All things coding

THE UNIVERSITY OF MELBOURNE

Overview of Lecture

- Coding – Web languages are a big learning curve, but competent skill level achievable
- Pair programming
- Repository configuration
- Web Frameworks
- Code reviews

THE UNIVERSITY OF MELBOURNE

Coding

- Use coding standards – plenty of standards around
- Linters are useful
- Be aware of open source licensing agreements
  - https://opensource.org/licenses
- Help your teammates
- Does everyone need to code?

THE UNIVERSITY OF MELBOURNE

Configuring your Repository

- Best practices for initial setup:
  - README file
  - Licensing
  - Branch protection
  - Integrations and Notifications
- Development via pull requests (don't push to Master)
- Meaningful pull requests (templates can help)
- Manage releases: https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository

A simple workflow

Image: buddy.works

Feature Branch Workflow

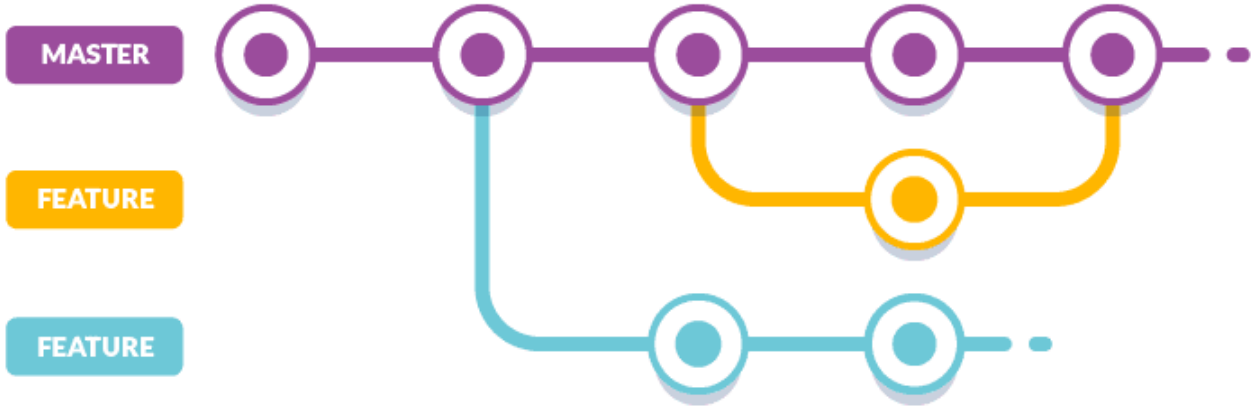


Image: buddy.works

Gitflow

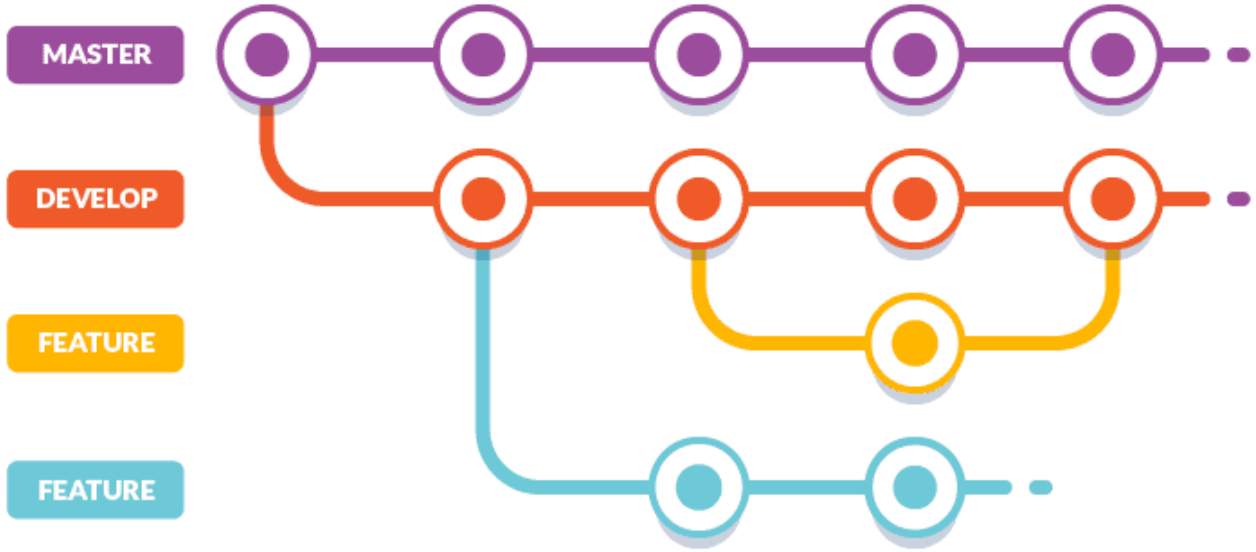


Image: buddy.works

Trunk-based development

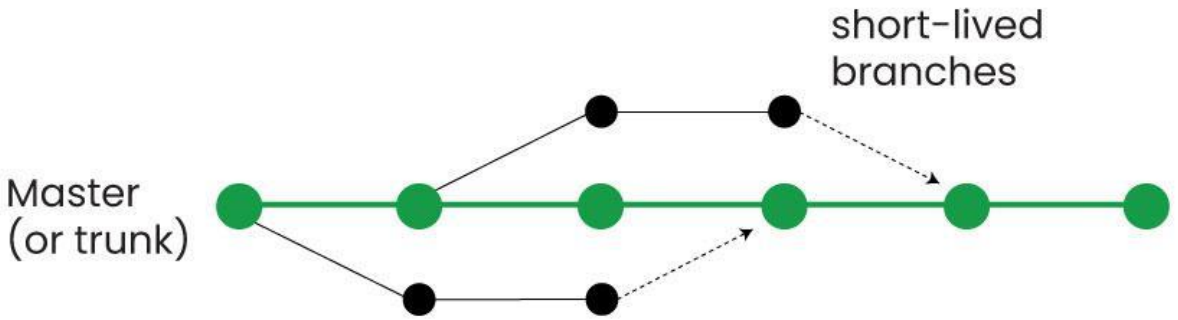
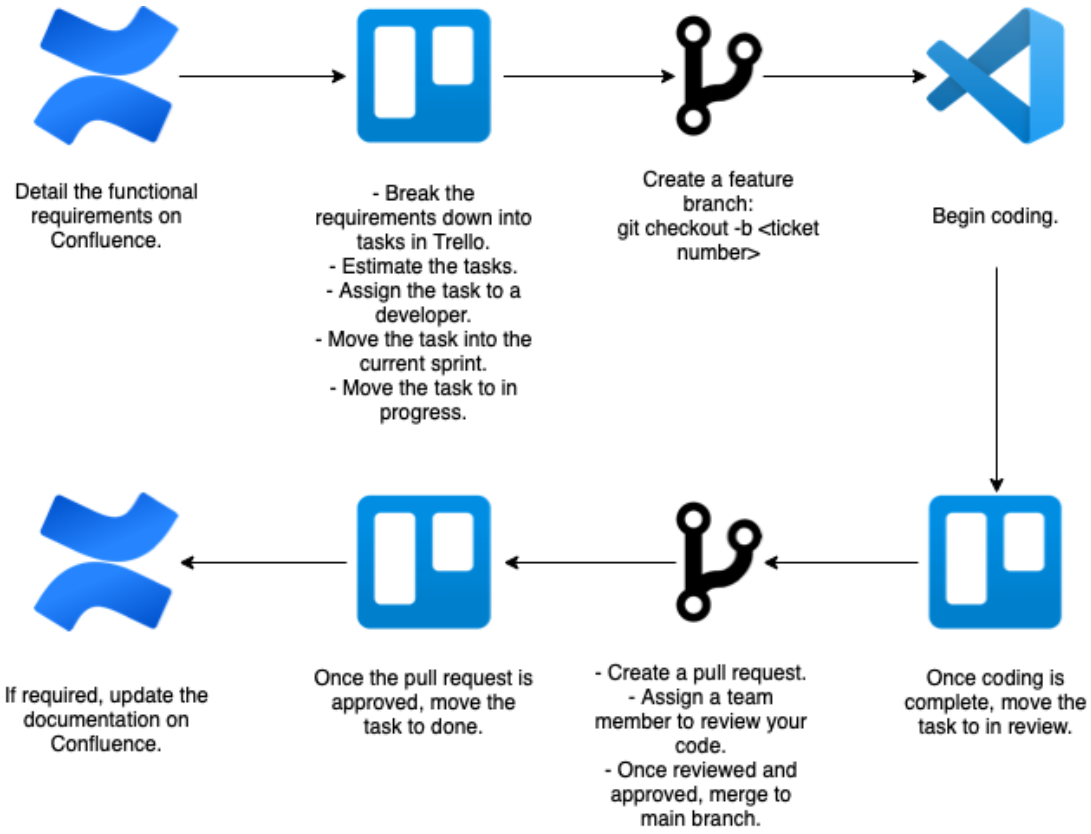


Image: geeksforgeeks.org

One possible workflow



THE UNIVERSITY OF MELBOURNE

Technology Stack

- You have freedom to choose a stack that suits your project
- Key considerations:
  - Client Preference
  - Team Expertise
  - Learning Opportunities
  - Resource Constraints
- Web Apps are a common project

Components of a Web Framework

- Key components:
  - Frontend Framework
  - Backend Framework
  - Web Server
  - Data Persistence

Popular Frontend Frameworks

- **HTML & CSS with Bootstrap**: Bootstrap simplifies layout and design.
- **ReactJS**: A JavaScript library for building user interfaces with reusable components.
- **Angular**: A comprehensive framework for dynamic web apps.
- **Vue.js**: A progressive framework for building UIs and single-page applications.

Popular Backend Frameworks

Language	Frameworks
Python	Django, Flask, Pylons
Java	Spring Boot
Ruby	Ruby on Rails
PHP	Laravel
JavaScript	Express.js, Koa

Popular Database Frameworks

- **MySQL**: A widely-used relational database.
- **MongoDB**: A document database suited for hierarchical data storage.
- **PostgreSQL**: An advanced open-source relational database.

Popular Web Servers

- **Nginx**: Known for its high performance, stability, and rich feature set.
- **Apache**: A robust, feature-rich web server with extensive support.

Pair Programming

- Pair Programming is a technique where two developers work together on the same task. In this one person writes the code (driver) and the other person reviews each line and provides feedback (navigator). This approach differs from traditional solo programming and offers various advantages.

from https://dev.to/documatic/pair-programming-best-practices-and-tools-154j#:~:text=Pair%20Programming%20is%20a%20technique,programming%20and%20offers%20various%20advantages – Jatin Sharma

Benefits of pair programming

- Improved code quality
- Increased collaboration and communication
- Increased productivity
- Myths
  - Pair programming is expensive
  - Pair programming slows things down
  - Pair programming causes tensions

Best practices for pair programming

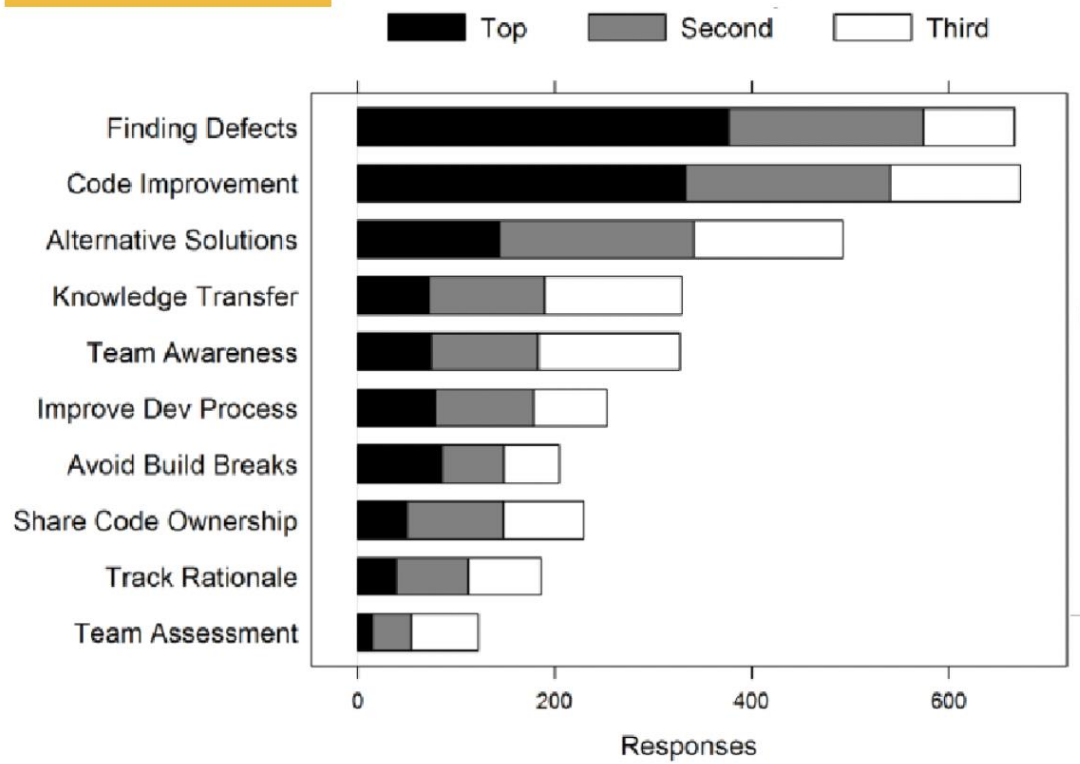
- Clearly define roles (*but not too rigidly, and both need to be involved*)
- Effective communication (👍👍)
- Respect and empathy (*always*)
- Take breaks
- Embrace learning opportunities
- Practice active code review (*next topic*)
- Plan and set goals
- Document and share knowledge (*Very important*)
- Reflect and improve

What is Code Review?

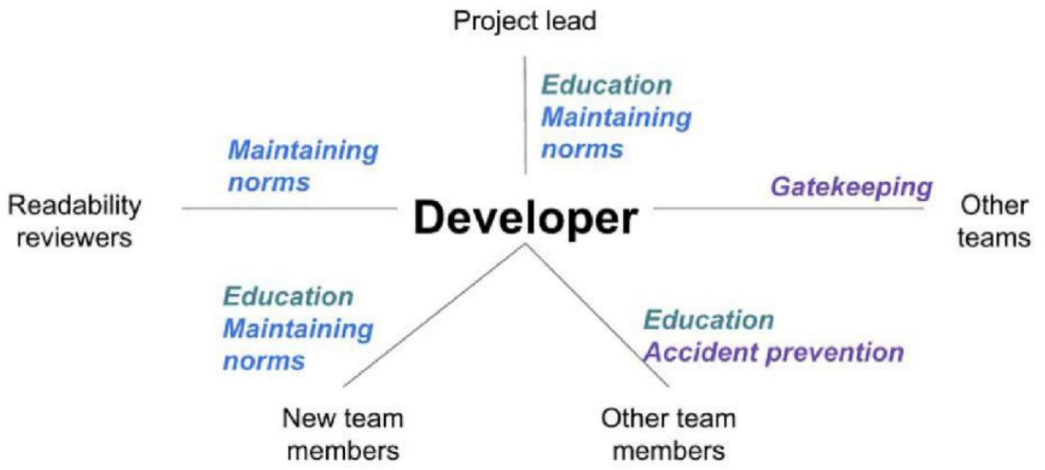
- Based on a lecture from Dr Patanamon Thongtanunam in 2020 to Masters of Software Engineering students
- Lecture available at https://cis-projects.github.io/project_based_course_notes/topics/code_review.html
- A Software Quality Assurance (QA) practice
- (Peer) Code Review - A practice of manually reviewing a software artefact by team members other than the owner.
- The code review can be used for checking all artefacts including test code, architecture design, etc.
- Code Review is a common practice in industrial and open source software projects.

Why we should do a code review?

At Microsoft



At Google



A. Bachelli and C. Bird, "Expectations, Outcomes, and Challenges Of Modern Code Review," in Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 712–721
C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern Code Review: A Case Study at Google," in Companion Proceedings of the 40th International Conference on Software Engineering (ICSE), 2018, pp. 181–190.

Why we should do a code review?

Finding Defects, Code Improvement, & Alternative solutions

"Given enough eyeballs, all bugs are shallow"

— Linus Torvalds

- An effective code review can reduce more than half of defects, which is more than the number of defects found during the test phase.
- The more code that undergo reviews (a high review coverage), the less number of defects and design anti-patterns.
- Code reviews identify weakness about the functionality/correctness (25%), and the evolution/maintainability of the code (75%).
- Code reviews nowadays become more ***collaborative*** *as* a reviewer aims to provide improvement suggestions rather than listing defects.

Why we should do a code review?

Knowledge Transfer, Team Awareness, & Share Code Ownership

- Novices can learn from experienced developers through code reviews
- Allow you to keep sharing and updating the knowledge within the team
- Enable you to be open for criticism!

"In the past people were very reluctant to put themselves in positions where they were having other people critiquing their code. The fact that code reviews are considered as a normal thing helps immensely with making people less protective about their code."

A senior developer at Microsoft

How should we do a code review?: Process

- The code review process (a.k.a Software Inspection) was invented in 1976 by Fagan
  - Formal, in-person meetings
- Code review process nowadays is lightweight, and tightly-integrated with the Version Control System (e.g., Git)
  - Code review tools: Gerrit, ReviewBoard, Phabricator
  - Pull-based development models

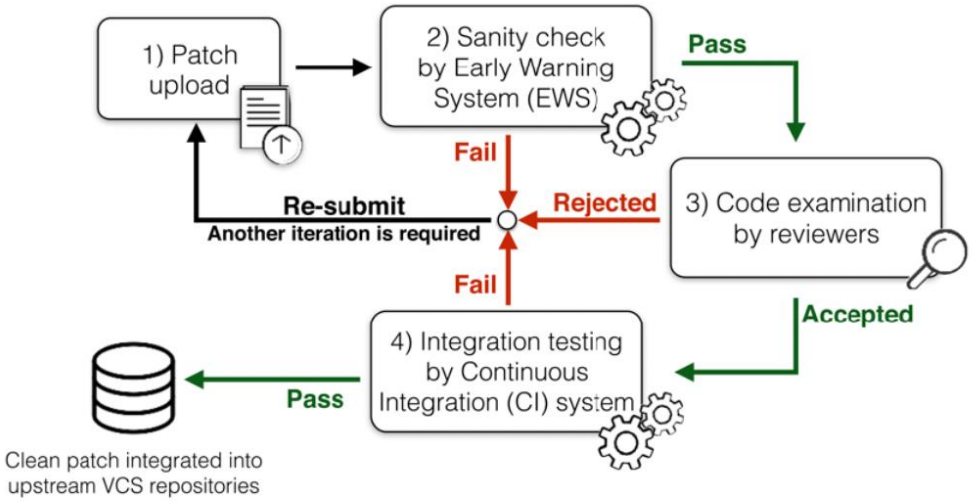
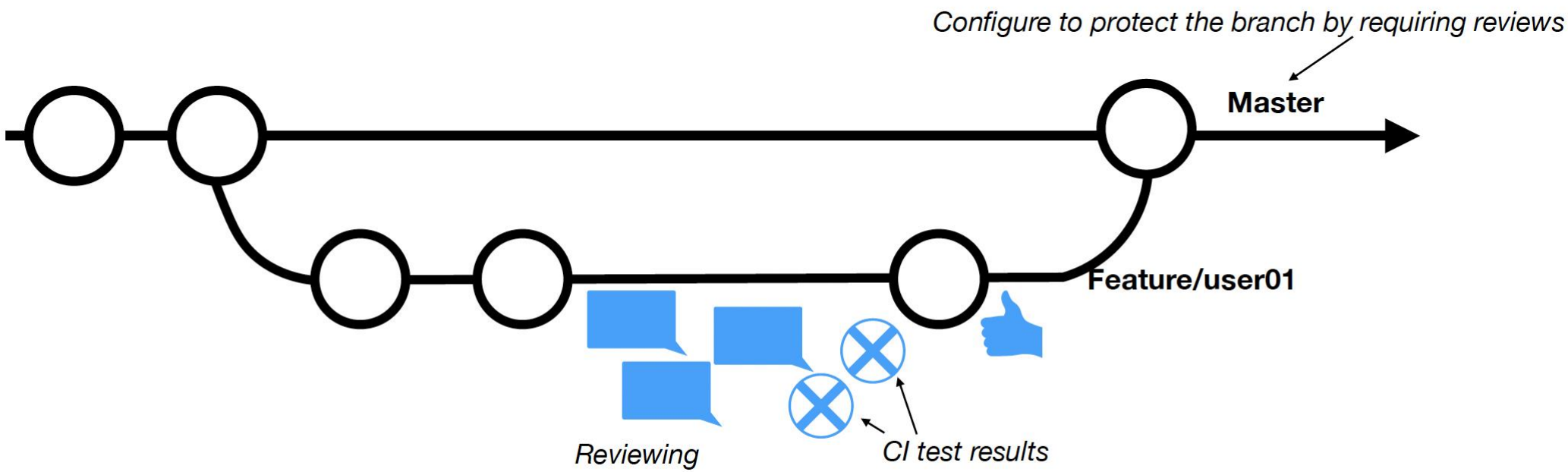


Fig. 1: Gerrit-based code review process of the Qt system

P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System," in Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR), 2015, pp. 168–179.

How should we do a code review?: Process

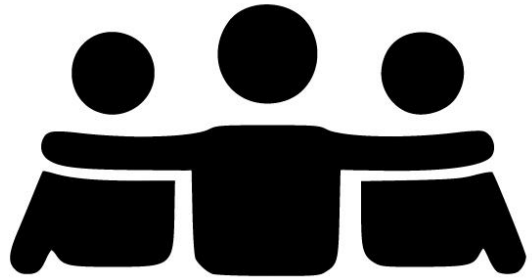
- An example of a pull-based development model



How should we do a code review?:Best Practices

As a team, you should

- Build and maintain ***a positive review culture***.
- Develop, reflect on, and revise ***code-reviewing policies***.
- Ensure that ***time spent is counted*** and expected, but watch for negative impacts of assessments.
- Ensure that the appropriate tools are available and used.
- Promote the development of appropriate review checklists.
- Have sufficient training in place for code review activities.
- ***Develop a mechanism to watch for bottlenecks*** in the process.



.. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches," IEEE Software., vol. 35, pp. 34–42, 2018.

How should we do a code review?:Best Practices

Code Review Policy

- How many reviewers per a review?
- Who should be a reviewer?
- How large of the code and how often?
- How long should it take for a review?
- How should an author prepare for a review (providing all the details)?
- What are the major concerns should reviewers looking for?
- When should be reviewed? And how soon should reviewers respond?

How should we do a code review?:Best Practices

Code Review Checklists

- Plenty examples of code review checklists available online

Google's Engineering Practices documentation

- [Complexity] Are individual lines too complex?
- [Complexity] Are functions too complex?
- [Complexity] Are classes too complex?
- [Naming] Did the developer pick good names for everything?
- [Comments] Did the developer write clear comments in understandable English?
- [Comments] Are all of the comments actually necessary?

Google's Engineering Practices documentation: What to look for in a code review https://google.github.io/eng-practices/review/reviewer/looking-for.html

How should we do a code review?:Best Practices

Code Review Checklists

- Consider all aspects of quality
  - Does the code follow documented architecture?
  - Does the code satisfy the acceptance criteria?
  - Is the code maintainable easily?
  - Does the code follow design principles, code convention?
  - Are the unit tests correct and sufficient?
  - Is the code comment informative and easy to understand?
  - etc.

THE UNIVERSITY OF MELBOURNE

How should we do a code review?:Best Practices

Code Review Checklists

Other software artefacts (e.g., requirements, architecture, documentation) should also have a quality checklist

1. **Actors**
   - 1.1. Are there any actors that are not defined in the use case model, that is, will the system communicate with any other systems, hardware or human users that have not been described?
   - 1.2. Are there any superfluous actors in the use case model, that is, human users or other systems that will not provide input to or receive output from the system?
   - 1.3. Are all the actors clearly described, and do you agree with the descriptions?
   - 1.4. Is it clear which actors are involved in which use cases, and can this be clearly seen from the use case diagram and textual descriptions? Are all the actors connected to the right use cases?
2. **The use cases**
   - 2.1. Is there any missing functionality, that is, do the actors have goals that must be fulfilled, but that have not been described in use cases?
   - 2.2. Are there any superfluous use cases, that is, use cases that are outside the boundary of the system, do not lead to the fulfilment of a goal for an actor or duplicate functionality described in other use cases?
   - 2.3. Do all the use cases lead to the fulfilment of exactly one goal for an actor, and is it clear from the use case name what is the goal?
   - 2.4. Are the descriptions of how the actor interacts with the system in the use cases consistent with the description of the actor?
   - 2.5. Is it clear from the descriptions of the use cases how the goals are reached and do you agree with the descriptions?
3. **The description of each use case**
   - 3.1. Is expected input and output correctly defined in each use case; is the output from the system defined for every input from the actor, both for normal flow of events and variations?
   - 3.2. Does each event in the normal flow of events relate to the goal of its use case?
   - 3.3. Is the flow of events described with concrete terms and measurable concepts and is it described at a suitable level of detail without details that restrict the user interface or the design of the system?
   - 3.4. Are there any variants to the normal flow of events that have not been identified in the use cases, that is, are there any missing variations?
   - 3.5. Are the triggers, starting conditions, for each use case described at the correct level of detail?
   - 3.6. Are the pre- and post-conditions correctly described for all use cases, that is, are they described with the correct level of detail, do the pre- and post conditions match for each of the use cases and are they testable?
4. **Relation between the use cases:**
   - 4.1. Do the use case diagram and the textual descriptions match?
   - 4.2. Has the include-relation been used to factor out common behaviour?
   - 4.3. Does the behaviour of a use case conflict with the behaviour of other use cases?
   - 4.4. Are all the use cases described at the same level of detail?

Figure 1. Checklist for inspections of use case model

Anda, Bente, and Dag IK Sjøberg. "Towards an inspection technique for use case models." *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. 2002.

How should we do a code review?:Best Practices

As a code author, you should

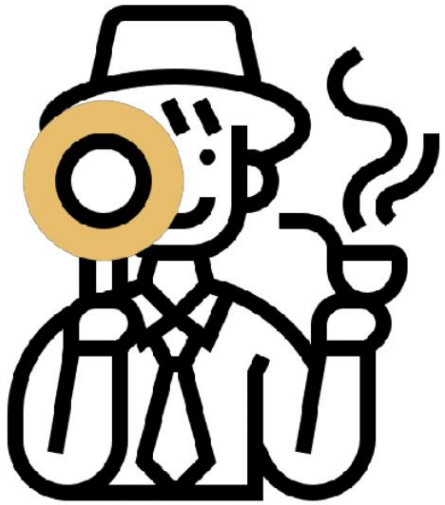
- Carefully check the code changes (including a sanity check) for a review
- Cluster only related changes
- Describe your changes and the motivation for them
- Notify reviewers as early as possible
- Promote an ongoing dialogue with reviewers
- Track the suggested changes and confirm that they're fixed
- Confirm that the decisions are documented

L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches," IEEE Software., vol. 35, pp. 34–42, 2018.

How should we do a code review?:Best Practices

As a reviewer, you should

- Set aside dedicated, bounded time for reviews
- Review frequently, doing fewer changes at a time
- Provide feedback to authors as soon as possible
- Focus on core issues first; avoid nitpicking
- Give constructive, respectful feedback



- Choose communication channels carefully; talk face-to-face for contentious issues (Don't forget to document the conclusion!)
- Be prepared to iterate and review again

L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, "Code Reviewing in the Trenches," IEEE Software., vol. 35, pp. 34–42, 2018.

How should we do a code review?: Common practices

- A review typically takes 1 day
- Waiting time is about 1 hour to 1 day
- Small code changes (11-44 lines were changed)
- Typically two reviewers per a review

- A review typically takes 4 hours
- Waiting time is about 1 hour for a small change and 5 hours for a large change
- Small code changes (typically 24 lines were changed)
- The number of reviewers depends on the change size (typically one)

P. C. Rigby and C. Bird, "Convergent Contemporary Software Peer Review Practices," in Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE), 2013, pp. 202–212.
C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern Code Review: A Case Study at Google," in Companion Proceedings of the 40th International Conference on Software Engineering (ICSE), 2018, pp. 181–190.

Remarks

- Finding defects is not the sole goal for reviewing
- Improving not assessing
- Not only code should be reviewed
  - All the deliverables (e.g., unit test, documentation) need a review as well
- Communicating, communicating, and communicating!
- Open for criticism
- Be ware of some subconscious biases
  - Experienced developers are not always right
  - Don’t follow the crowd decision. Do raise a concern if you have ones.

References

- A. Bacchelli and C. Bird, “Expectations, Outcomes, and Challenges Of Modern Code Review,” in Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013, pp. 712–721
- C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern Code Review: A Case Study at Google,” in Companion Proceedings of the 40th International Conference on Software Engineering (ICSE), 2018, pp. 181–190.
- K. Burke, “Why code review beats testing: evidence from decades of programming research,” [Online]. Available: https://kevin.burke.dev/kevin/the-best-ways-to-find-bugs-in-your-code/
  P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System,” in Proceedings of the 12th International Working Conference on Mining Software Repositories (MSR), 2015, pp. 168–179.
- S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The Impact of Code Review Coverage and Code Review Participation on Software Quality,” in Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR), 2014, pp. 192–201.
- R. Morales, S. McIntosh, and F. Khomh, “Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects,” in Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SAnER), 2015, pp. 171–180.
- P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, “Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review,” in Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016, pp. 1039–1050.
- Example of Buggy code: http://www.mit.edu/~6.005/fa14/classes/03-testing-and-code-review/#smelly_example_2, http://courses.cs.vt.edu/~cs1206/Fall00/bugs_CAS.html
- L. MacLeod, M. Greiler, M.-A. Storey, C. Bird, and J. Czerwonka, “Code Reviewing in the Trenches,” IEEE Software., vol. 35, pp. 34–42, 2018.
- P. C. Rigby and C. Bird, “Convergent Contemporary Software Peer Review Practices,” in Proceedings of the 9th joint meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE), 2013, pp. 202–212.