S06 Code Review

Code Review
Comments and Documentation

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

What is Software Complexity?

- ***Accidental* Complexity**
  - Software that is designed or presented in a way that is more difficult for a **human to understand, use and modify** *than it needs to be*.
  - It is difficult to write elegant, clear, reusable code.
- ***Essential* Complexity**
  - Inherent to the problem being solved. Irreducible.
- **Not to be confused with** computational complexity (about performance).

Code Review

- One or more people review code who are removed from the implementation.
- Commonly done for a specific change (e.g., set of git commits) but can also be done for a complete project / implementation.
  - **Fix** a specific bug
  - **Implement** a new feature
  - **Refactor** part of the code
- Gitlab offers a “merge request” workflow (“pull request” on github) where reviewers / maintainers review the changes **before** they are merged into the mainline branch.

Code Review Motivations

- Barrier to ensure project remains **maintainable**.
  - Improve implementation / quality.
  - Clarify code, double-check edge cases.
  - On-balance rejection of a feature (accidental or essential complexity).
- Second pair of eyes: potentially less biased, can consider bigger picture, can bring new insight.
- Effective way to **learn** a new code-base and a team's processes / conventions. Highlights interrelated parts.
- Can catch some bugs before reaching production… but implementer really should have adequate tests developed and passing.

Doing a Code Review

- **Objective**: is it in scope of this project
- **Functionality** (for end-users and developers):
  - does it do what is intended
  - edge cases / bugs
  - might have to run code for UI changes etc
- **Tests**: present, appropriate
- **Complexity**: design minimises / encapsulates complexity
- **Good names**: convey information and not too long
- **Comments**: help to understand decisions and the why, not repeating code, appropriately documenting interfaces
- **Conformance** to project style guide / conventions.
  - Formatting at a later stage can destroy attribution information, i.e. git blame become useless

Further Tips

- Be considerate.
- Point out things that are good!
- Clearly label *nitpicks* as such.
  - (Or don’t nitpick at all – next slide.)
- No code is ever perfect. Tailor to circumstances:
  - flight control software
  - a game

“Stop Nitpicking in Code Reviews” Dan Lew

- “**First, we saw a vastly improved signal-to-noise ratio.** Imagine a code review that results in five nits and one critical issue to address. In the hullabaloo of fixing those nits, the critical comment can seem less important or even get overlooked.”
- “**Second, it improved everyone’s relationships with code reviews and those who conduct them.** I’ve seen plenty of people talk online about how you shouldn’t take code reviews personally, that it’s the code being critiqued not the person, blah blah blah, it’s a bunch of bullshit. I’ve been going through code reviews for a decade and it still stings when someone points out my mistakes or pushes back on my code designs.”

Code Comments / Documentation

- **Class or method comments – always for** `public`
  - How to use, edge cases, side-effects, pre/post-conditions, invariants, explain abstraction, examples.
  - Should not leak the implementation details.
- **Implementation comments – as required**
  - Give intuition where implementation is non-obvious to a likely contributor / your future self
  - Highlight where edge cases are handled if hidden
  - Rationale for the design if not the obvious choice
  - Should not just repeat code

Additional pointers

- Examples
  - https://github.com/junit-team/junit5/pull/3477
  - https://github.com/junit-team/junit5/pull/3206
- Guide
  - https://google.github.io/eng-practices/review/