Collaborative Construction

INTERSECT - Bootcamp 2023

Dave Rumph

Collaborative Construction

- Testing and debugging are not the best way to find software defects
  - Primary benefit is to quickly detect and prevent regressions
- Design- and code-reviews are *much* better at finding defects
- These are called **collaborative construction techniques**
- Two widely-used techniques:
  - Code reviews (extremely widespread)
  - Pair-programming (less widespread)

Defect Detection / Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1000 sites)	60%	75%	85%

Code Complete 2ed., pg.470

Collaborative Construction

- Collaborative construction techniques are often presented as being focused primarily on finding defects
- This is definitely true:
  - Reviewers can identify bugs that the developer may have overlooked
  - Reviewers can gauge how well the code actually satisfies the requirements
  - Reviewers may be able to suggest better designs, better refactorings of the code being reviewed, other improvements
  - Since reviewers are often less familiar with the code, they can point out where coding style or commenting needs to be improved
  - Reviewers can verify that the code has sufficient test-automation, and can suggest additional tests to improve verification

Collaborative Construction

- Turns out to be many other benefits realized from these practices
- Positive peer-pressure to maintain a high standard of code quality
  - Knowing your code will be reviewed, increases motivation to do a good job
  - Motivates better commenting, code-cleanup tasks, writing good tests, etc.
- Facilitates knowledge-transfer among teammates
  - Helps more people to understand how the project's code works
  - Opportunity to mentor junior developers in best-practices
- Fosters healthy team dynamics
  - Teammates learn to give and receive thoughtful feedback with each other
  - Develops a sense of *community responsibility* for project code
  - Reduces feelings of "*my* code," and increases feelings of "*our* code"

Code Reviews

- **Code reviews** (a.k.a. **peer [code] reviews**) are a practice where someone other than the code's author reads and reviews the code
  - Author is often involved in the review, to answer reviewers' questions
- Sometimes these are *formal inspections* of a portion of the code
  - May be part of an acceptance process required e.g. for safety-critical software
  - Tends to be a more heavy-weight process with specific criteria to evaluate
- More commonly, these are *change-based* code reviews
  - e.g. "Every new feature or bug-fix must be reviewed before it is incorporated into the main-line of development."
  - This is the process adopted by most large software companies
  - Tends to be lighter-weight, akin to a "**code walk-through**" or "**read-through**"

Code Reviews – Benefits

- Benefits of formalized code reviews (inspections) is very well documented
- From Code Complete, 2ed. pp.480-481
  - IBM found that each hour of inspection prevented about 100 hours of related work (testing and defect correction) (Holland 1999)
  - Hewlett-Packard reported that its inspection program saved an estimated $21.5 million per year (Grady and Van Slack 1994)
  - A study of large programs found that each hour spent on inspections avoided an average of 33 hours of maintenance work and that inspections were up to 20 times more efficient than testing (Russell 1991)
  - A group of 11 programs were developed by the same group of people, and all were released to production. The first five were developed without reviews and averaged 4.5 errors per 100 lines of code. The other six were inspected and averaged only 0.82 errors per 100 lines of code. Reviews cut the errors by over 80% (Freedman and Weinberg 1990).
  - [*and many many more…*]

Code Reviews – Benefits (2)

- Less-formal code walk-throughs/read-throughs vary much more widely in their benefits
- They can still achieve many of the knowledge-sharing, team-building and defect-detection benefits, but not as reliably as formal reviews
- Main differences:
  - The code's author tends to lead the walk-through, as opposed to a separate moderator in an inspection. Read-throughs are self-driven by the reviewer.
  - Reviewers are varied in their commitment to providing a thorough and detailed assessment of the code.
  - Defects in the code may or may not be formally recorded as action-items to address, with a follow-up to see if they have been corrected
- As usual with life, you get out of it what you put into it
  - Taking care not to lose the rigor of the review process is essential to achieving all the benefits to be had

Code Review Best Practices: Scope

- It can be tempting to set lofty goals for code reviews…
- Reality: People just can't concentrate too deeply or for too long!

- **Widely-recommended practices:**
  - **Limit code reviews to 60 minutes or less**
  - **Limit the code being reviewed to 200-400 lines**

- Anything more will greatly reduce quality of feedback at some point…

Code Review Best Practices: Goals

- **Reviewers should have a clear sense of what feedback is being solicited**
  - Some companies have general guidelines for what code-reviewers should be looking for in the code they review
- If you are seeking specific kinds of feedback, communicate that to your reviewers!
  - In the absence of this, projects should include general code-review goals and guidelines for reviewers to reference
- If there is other background knowledge necessary for an effective code-review, provide it to the reviewers
  - Can be included/referenced in comments, provided in documentation, etc.

Example Code-Review Goals (Google)

Code reviews should look at:

- **Design:** Is the code well-designed and appropriate for your system?
- **Functionality:** Does the code behave as the author likely intended? Is the way the code behaves good for its users?
- **Complexity:** Could the code be made simpler? Would another developer be able to easily understand and use this code when they come across it in the future?
- **Tests:** Does the code have correct and well-designed automated tests?
- **Naming:** Did developer choose clear names for variables, classes, methods, etc.?
- **Comments:** Are the comments clear and useful?
- **Style:** Does the code follow our style guides?
- **Documentation:** Did the developer also update relevant documentation?

https://google.github.io/eng-practices/review/

Code Review Best Practices: Feedback

- **Providing courteous and respectful feedback on another person's work is a critical skill to develop**
  - Value this as a high priority! Never compromise on it, even in jest.
- Use objective, constructive, neutral language
  - Critique the code, not the programmer
  - Avoid theatrics and absolute judgments, like "This is the worst code I've ever seen!" "I have no idea how this ever worked in the first place."
- Ask questions and seek understanding, rather than rendering judgment
  - The author has thought about the problem a lot more than you have, and they may have reasons for what they did – even if it's still misguided in the end
- Explain your concerns clearly and completely
  - Try to explain your reasoning for issues you point out, e.g. inputs that would trigger undesirable behavior, alternate approaches that may be more efficient, etc.

Code Review Best Practices: Feedback

- You may have noticed that programmers are often highly opinionated…
- There are many good ways to solve any given problem – including ways that you personally wouldn't do things
  - Different ways of naming variables, writing loops, structuring functions
- **Indicate whether your feedback is about a serious issue (e.g. correctness, security, safety), vs. merely you expressing an opinion or suggestion**
  - Several companies specify that review feedback should be prefixed with "Nit: …" if you are merely being nitpicky
  - Reviewer opinions and nitpicks are free to be ignored by the author!
- A partial solution to this is a well-defined and complete coding standard
  - Google: "On matters of style, the style guide is the absolute authority. …"

Code Review Best Practices: Tools

- Some goals of code reviews are to check coding style, and to identify bugs, anti-patterns and language abuses
- *"Hey, guess what? Static code-analysis tools can do that!"*
- **Ideally, the code-review process is supplemented by static-analysis tools**
  - Verify (or apply) the team's coding standards
  - Flag common bugs and language anti-patterns
  - These tools can eliminate a whole range of issues that would otherwise need to be addressed in the code review
- Allows the review to focus on higher-level concerns – how the code works, how it benefits the project, higher-level structural/design issues, etc.
  - …exactly the kind of things that static code-analysis tools aren't capable of…

Code Review Best Practices: Tools (2)

- Static code-analysis tools for Python:
  - Flake8 – coding style and linting
  - Pylint – coding style and linting
  - Black – coding style enforcement
  - mypy – check Python code against type annotations

- In a future project you will need to incorporate Flake8 or Pylint to identify and fix bugs and language anti-patterns
- Of course, you are welcome to do this early too! ☺

Code Reviews: Resources

These three are incredibly valuable and should be read by everyone:

- Google – Code Review Developer Guide
  - Very detailed guidelines for all participants of code reviews
  - "How to do a code review document" is very good for mechanics of reviews
- Palantir – Code Review Best Practices
  - An excellent and detailed guide about code reviews
- StackOverflow – How to Make Good Code Reviews Better

Also excellent articles:

- Atlassian – Why code reviews matter (and actually save time!)
  - Discusses code reviews in the context of agile development methodologies
- Perforce – 9 Best Practices for Code Reviews

Pair Programming

- **Pair programming** (a.k.a. **pairing**) is an increasingly popular technique used in software development
  - Addresses the “high cost, low payback” feeling of being a code reviewer
- Two developers work together actively, to write a feature and its tests
- The **Driver** is the one who actually controls the keyboard and the mouse, and writes the code
  - This is “programming out loud” – the driver also talks, describing their approach, asking questions about potential issues, confusing details, etc.
- The **Navigator** observes what the driver is doing, asking/answering questions, evaluating the work, pointing out potential pitfalls, etc.
- Periodically, at reasonable points, Driver and Navigator switch roles

Pair Programming (2)

- Pair programming takes more time to write a given program…
  - “Two people do the work of one person” – a completely wrong assessment
  - Several studies show a ~15% increase to development time with pairing, after participants became comfortable with the approach
- But, if pairing saves sufficient costs in other parts of the development lifecycle (e.g. debugging, maintenance) then it’s worth it

- As with code reviews, the benefits of pair programming turn out to *far outweigh* this increase in development time
- Also, as with code reviews, this is only one of several significant benefits realized by pair programming

Pair Programming and Defects

- Pair programming does an excellent job of improving code quality!
  - One study showed that pair-programming produced code with 15% fewer defects than solo-programming
- Assuming that pair programming is generally this effective, the time-savings in future debugging and maintenance costs, more than compensates for the slight increase in development costs
- The longer a defect is undetected, the more costly it is to fix
  - Pair-programming tends to detect and resolve defects at the earliest possible moment, before code is even included in a commit to a repository

Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture		1	10	15	25-100
Construction			1	10	10-25

Pair Programming and Defects

- Pair programming does an excellent job of improving code quality!
- The reasons for this are what you would expect…
- Two people tend to do much better considering all possible execution scenarios than one person does
  - A single implementer can become hyper-focused on a specific part of the implementation, losing detail on the "big picture"
- Two people can validate an implementation against requirements more effectively
  - Motivating all parts of the implementation against all requirements
  - Variances in understanding of a requirement will also be identified through discussing the requirement
- "Rubber-duck debugging" is incorporated into the process

Pair Programming and Defects

- Pair programming also tends to produce higher-quality *designs*
  - In quantitative studies, pair-programmed implementations tended to be shorter than the corresponding solo-programmed implementations
  - (Assumption: shorter = better, which is usually the case)
- Two programmers working together will consider a larger design-space when solving a problem
- Strengths and weaknesses of each design are explored more deeply
- The process of negotiating to a consensus allows trade-offs to be balanced more carefully and effectively
  - (These trade-offs should also be evaluated against the actual requirements)

Improved Satisfaction and Team Dynamics

- Individual satisfaction increases dramatically from pair programming
  - Collaborating with someone on a shared task is far more enjoyable for most people
  - Celebrate successes, mourn setbacks together, etc.
  - One study consistently showed >90% of participants preferred pair programming over solo programming (Williams 2000)
- Team members learn to work with each other more effectively
  - Pair programming gives consistent practice in this area
  - Requiring pair-programming is especially effective, as people are forced to work through personality differences and different working styles
- Knowledge and skills are distributed more broadly throughout the team
  - Pairing junior and senior developers creates more opportunity for mentoring
  - Including teammates unfamiliar with a part of the code will teach them how it works, and broaden the shared knowledge throughout the team

Improved Productivity

- Teams that practice pair-programming frequently report increased productivity
  - Switching regularly between the Driver and Navigator roles keeps developers fresher and more engaged throughout the day
  - If one developer is distracted by some external interruption, the other person can get them back up to speed on the task more quickly

Pair Programming – Issues

- Pair programming requires active focus, communication and participation from both programmers
  - *Long periods of silence is a warning sign!*
- Pair programming simply won't work well if:
  - Either participant isn't focused on the task at hand
  - Either participant isn't making an effort to engage in ongoing communication about the task at hand
- Other practices to avoid (in a larger team setting):
  - The same pairs of developers always work together, or developers never work on areas they are unfamiliar with (inhibits knowledge-sharing benefits)
  - The participants have unresolvable personality conflicts (rare, but it happens)

Pair Programming – Issues

- Not every part of a program warrants pairing
- Example: Two experts working on simple, well-understood tasks will likely work faster individually than if they are pairing
- Pairing achieves the greatest benefits when:
  - The task is complex and/or difficult to implement correctly
  - The programmers are unfamiliar with the problem being solved
- Senior developers should also pair with junior developers in order to provide mentoring
  - Important for senior developer to be patient and work at the junior developer's pace
  - Also important for senior developer to solicit feedback from the junior developer on their code. *Everybody makes mistakes!*

Pair Programming – Guidance for Drivers

- Spend time up-front discussing possible design approaches with Navigator
  - Assuming you haven't already discussed various design approaches with your team
- Make sure you are "thinking out loud" as you program
  - Describe, at a reasonably coarse level of detail, how you are solving the problem you are trying to solve
  - After implementing anything significant (e.g. a function, a relatively complex loop or chunk of code), ask for input!
  - "Does this look right?" "Am I missing anything?"
- After writing code, always spend time brainstorming how to test it
  - Even if tests are not written while pairing, make notes about how to test
  - Recall: Test code should be as important as product code, and should be designed and maintained just as carefully
- "Give up the steering wheel" at suitable points! ☺

Pair Programming – Guidance for Navigators

- If you see a possible issue, mention it!
  - Don’t quietly assume that the Driver is aware of an issue you might see
  - They may be focused on some other part of the problem
- Consider whether the code’s control-flow handles all possible scenarios correctly
- Consider whether the code fully satisfies the requirements – and also whether you and the Driver fully understand the requirements
- Feel free to suggest structural/style improvements, as well as possible bugs
  - If part of the code is complex enough to warrant a comment, point it out
  - Don’t be too nit-picky about unimportant personal variations in coding style
- If any part of the code makes assumptions about inputs or internal state:
  - Make sure they are documented!
  - Make sure they are enforced, through assertions, sanity-checks, etc.

Pair Programming Activity — Setup

- Pair up with someone
- Clone the “Code-Review” repo
  - What you need for this activity is in the “activity” folder
- Make sure it runs on your system:
  - Run “python3 inklimit.py testimage.tiff outimage.tiff”
  - Install any missing modules using “pip3 install <module>”
- Make sure you can view the input and your output image files
  - How to do that depends on your system

Pair Programming Activity — Task

- Look at the task description (on right)
- Choose who will Drive, who will Navigate, and get to work
- Communicate
  - What possible approaches could you take?
- This activity is about the pair programming process, not the code!

- Complaint from users:
  - The inkjet printer leaves puddles of ink in the dark areas!
- Your task:
  - Change the code in inklimit.py to limit the total ink in the C, M, Y and K planes for each pixel to a given percentage (e.g. 240%)
  - Avoid changing the perceived color of the image
  - To limit the time needed for this task, I have provided you with the boilerplate code to open, read and write TIFF files
  - Your team needs only to implement the actual ink limiting algorithm

Code Review Activity

- Join together three (or two) pairs
- Choose which pair's code changes you will be reviewing
- Run flake8 or pylint on the inklimit.py file and interpret the output
  - "flake8 inklimit.py"
  - (pip3 install flake8 or pip3 install pylint if necessary)
- Choose someone to be "author"
- Determine what categories of "defect" should be identified
- Conduct a Code Walkthrough of the chosen code

Debrief of Collaborative Construction

- How did the Pair Programming go?
  - Did anything surprise you?
  - If you were Driver, did you feel that the Navigator's interactions were appropriate and helpful?
  - If you were Navigator, did you feel engaged in the development process?
- How did the Code Walkthrough go?
  - Did you find anything substantial?
  - If you were not the author, do you understand the code better?
  - Do you feel more part of the team?
- How can you make use of these practices in your own context?