Contents

1 Introduction

Responsible code development is not a byproduct of good programmers. It is a paradigm of development in and of itself. It is this paradigm that shapes the modern programmer, the weight on which it lays can be seen in the software issues of the past. The Therac-25, was a machine with software that had not been scrutinized, leading to deaths. The 2012 Knight Capital incident led to losses of millions because of issues stemming from integrating changes into an old system. The Y2K incident spread panic among civilians based on the idea that the internal clocks of old systems would not be able to represent the year 2000 causing system failures. All of these, are grounded on the limited role reliability played in past system development. To meet these issues, we have shaped various procedures to underline our development. One of these is continuous integration.

When developers make rapid changes as they have to do during team development they inadvertently introduce many issues to the stability and quality of the shared code base.

Continuous integration is the process of enabling developers to make rapid changes to an existing code base while sustaining code reliability. This is done through two main strategies, the first is running tests on changes and the second is to, depending on the results of these tests control the actual changes made to the shared code base.

2 The Tests

Poking holes in programs, observing if they fail allows us to find errors that otherwise would have flown by our radar. Testing in software is a crucial part and one principle that is of especial importance is brought up in an article on software testing, A test cannot prove the reliability of a program, it can only prove that lack of it.[2]

We must be wary, if we do not heed this, we may end up building and building without having ever broken the code down, leading to bugs that were once easy to catch, now, layered deep beneath mazes of functionality.

2.1 Manual Tests A Review Approach

Manual tests is a valuable step of this process and, thus, a big focus of CI is enabling the correct peer review system so that the quality of the code does not falter.

2.1.1 The Power of the Mind

The mind of humans can do amazing things, a computer is good at running enormous amounts of specific tasks but the mind is good at making connections. This is the power of the mind, our ability to abstract and reason about code changes is invaluable and is something a computer cannot yet accomplish to the same proficiency.

2.1.2 The Right Environment

However, the mind needs the right environment to do what we want properly, to avoid falling into lazy habits such as just pressing 'okay' for an upcoming change. We have to shape the right kind of environment that values feedback and discussion over conflict avoidance.

2.1.3 How to Foster the Right Environment

To make the environment correct, we have to enable our creativity and thinking part, we have to make code reviewing into a problem-solving activity instead of a defect-finding venture. This implies that changes are to be discussed, and we need tools that allow effective discussion of code. Thus, we are searching for tools for code referencing, response referencing, and easy access to coworking sessions of visual demonstrations of issues. This allows the reviewer to easily reference and exemplify visual problems in the code and the author to effectively reference the reviewers' comments and edit the visual diagram if necessary for clarification. (cite VPS). Moreover, we are making the information into a much more compact form by diagramming instead of having a bunch of text describing the situation. This helps people hop on the review process that has already started.

2.1.4 Selecting the Right Process

- The old code review process.

  The older process consists of a methodical line-by-line inspection of code [1]. This approach was very time-consuming and, thus, was not practical enough to apply at companies.

- The Modern Code Review.

  The modern process of code review is focused on small changes and fast interactions. [1] This brings up a very important connection to what the human mind is built for, we are good at reasoning and abstracting. Thus, if a change is made in the name of a specific process, instead of reading line by line we can focus on the abstract of how the new change is made possible by code. This means that for effective code reviews, we should make sure that our changes stand out. For example, using code-diff-tools to highlight changed code but we should also make sure that the idea that the change addresses is discussed in the imported change. If the author tried to translate an idea into code, this should be relatively simple to address since the idea is what the author began with. [3]. Moreover, we need to realize that we are emotional creatures and studies have shown that negative feedback provides worse results. This means we have to focus on discussing the problems rather than the humans and if we decide to discuss the humans involved we should be focusing on positive aspects [3].

2.2 Automated Tests

However, if testing is only manual, then we are not leveraging the full capacity of the modern computer, a modern computer can run millions of tests and always remembers to rerun tests that we have written. Thus, we have the frameworks for breadth and the assurance of memory that we do not have with manual testing. It is essential to utilize these aspects of testing to catch bugs that are reflected in coding mistakes rather than in ideas and how they are supposed to be implemented and catch bugs that break parts that humans simply could not have noticed during a short inspection.

2.2.1 Test Tools

There is a distinction between tools used to analyze programs and tests but there does not need to be. What automated tests do is produce a report based on a computer analysis of a program. That is the essence of tests. Thus, tools that produce a more valuable report such as

- Profiling tools
- Coverage tools
- Complexity evaluation tools

Is essential for an automated testing procedure. Therefore, any CI should generate a report based on the tools available to the language that is being developed in.

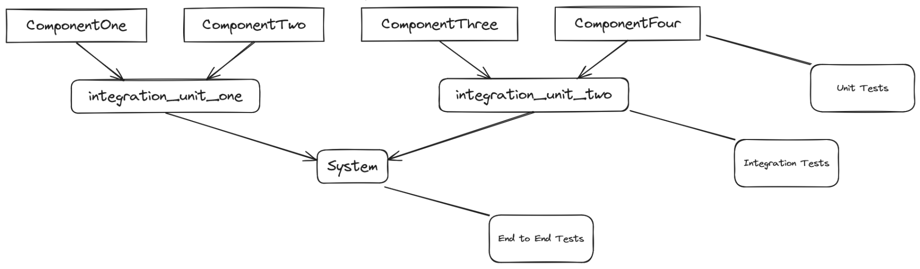
2.2.2 The Smart Input Selection

Considering automatic vs manual test, we humans have the capacity to reason about input to choose based on the code itself. Thus, we can choose good input to try. However, even though this is true for manual as can be seen by the sources supporting manual testing. How does this hold up to trying to be nifty about future failures?

The research shows that our attempts to predict good input for automatic tests fair worse than just random input [2]. Thus, when creating tests we should leverage the power of a computer, we should try to randomize our tests and bring them as far as possible away from human error. However, what good is errors if we cannot pinpoint what they are caused by. Therefore, it is also important to have some manual tests that tests very specific input to output relationships.

2.2.3 A Modular Approach

That last sentiment is also reflected in how modern testing is structured. We try to pin point the different intersection of components and their inner functioning to create tests that helps us quickly find the source of the test failure.



2.3 The Manual and Automatic Synergy

The full value of tests is only achieved once the two parts are brought together and that is the essence of continuous integration. First the computer runs the tests for requested code changes then a reviewer respond with their decision based on the information gathered by the tests. Thus, the reviewer gets the most amount of information possible to abstract from, code coverage, profiling, test failures and many more code test reports. Thus, a more thorough analysis than just "this should work because the code looks good" is achieved.

2.3.1 Regression Tests

Moreover, the manual review sessions form a feedback loop with the tests. Any bug caught by a review is made into a regression test. A test that aptly names prevents regresses, that is diving into functionality and bugs that should already be working.

Another importance of regression tests is the mentality that, new changes are tested against old system requirements. This means that tests for old systems are not just discarded since they can hold important bugs that the new developer did not think of in the replacement system.

3 The Control

Now given a thorough testing groundwork, we should ideally be able to construct one big report from the analysis of the code and based on this analysis, a decision must be made. Do we allow the code to integrate into the shared code base?

3.1 Version Control Systems

First, we need to mention how this integration decision is controlled and that in a modern environment through version control systems.

This is essential, we want to be able to separate shared code bases that have lesser restrictions on them and are there for different reasons. Moreover, we must be able to do a rollback in case of mistakes or change of direction and we want to be able to track who is an appropriate person to ask for a manual review. For example, Google chooses someone who made changes recently to review a suggested code change [3].

All this is provided by a modern VCS. [6]

3.2 The Feedback Control

There is no one size fits all for what decision to make based on the analysis. However, we do have some software development standards and when deciding what standards we should have we have to look at the best practices and to what degree we can follow these best practices without sacrificing too much practicality. Thus, a healthy discussion before the project must be met on the standards of the code base. Once this rule is set, the team should adhere to these rules when judging test results.

Some best practices are derived from this article [5]:

- Consistent code formatting and style:
  - Which industry standard should we follow?
- Code readability and documentation:
  - Is the code too complex to analyze and debug?
  - What level is the team on?

4 A Worked Example: Our Java Calculator

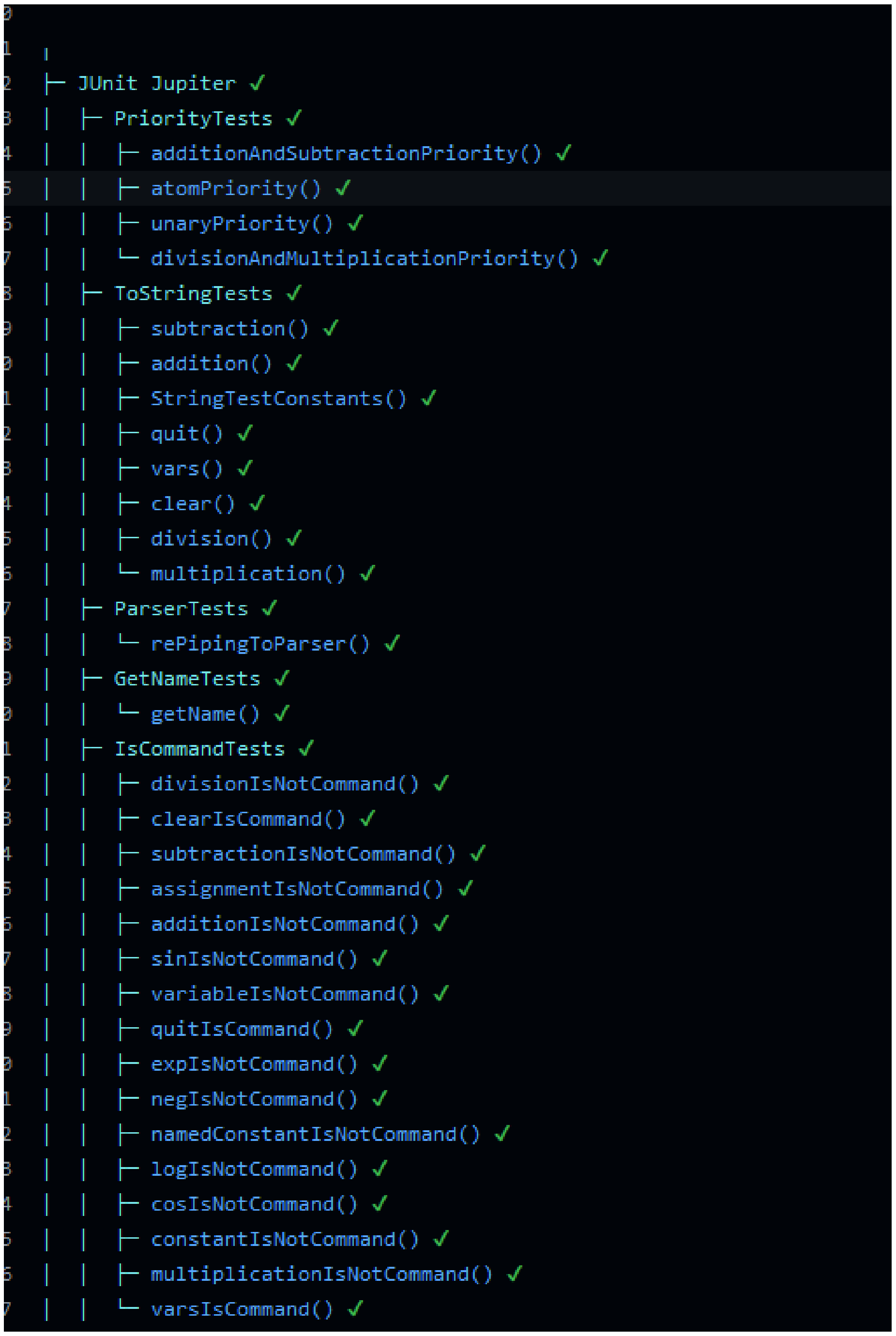
Now after having discussed the theory let's finish with a practical example, where I and my team used continuous integration for our product.

4.1 The CI file

To do the automated side of the CI process we use git actions. It runs the test automatically and it also generates a report on the tests, this report contains test coverage and results.

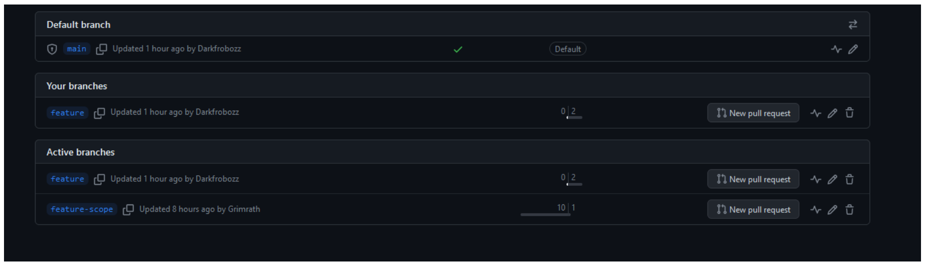
https://github.com/Darkfrobozz/inlupp4/blob/main/.github/workflows/makefile.yml

4.2 A Photo of Test Runs



```
╷
├─ JUnit Jupiter ✓
│  ├─ PriorityTests ✓
│  │  ├─ additionAndSubtractionPriority() ✓
│  │  ├─ atomPriority() ✓
│  │  ├─ unaryPriority() ✓
│  │  └─ divisionAndMultiplicationPriority() ✓
│  ├─ ToStringTests ✓
│  │  ├─ subtraction() ✓
│  │  ├─ addition() ✓
│  │  ├─ StringTestConstants() ✓
│  │  ├─ quit() ✓
│  │  ├─ vars() ✓
│  │  ├─ clear() ✓
│  │  ├─ division() ✓
│  │  └─ multiplication() ✓
│  ├─ ParserTests ✓
│  │  └─ rePipingToParser() ✓
│  ├─ GetNameTests ✓
│  │  └─ getName() ✓
│  ├─ IsCommandTests ✓
│  │  ├─ divisionIsNotCommand() ✓
│  │  ├─ clearIsCommand() ✓
│  │  ├─ subtractionIsNotCommand() ✓
│  │  ├─ assignmentIsNotCommand() ✓
│  │  ├─ additionIsNotCommand() ✓
│  │  ├─ sinIsNotCommand() ✓
│  │  ├─ variableIsNotCommand() ✓
│  │  ├─ quitIsCommand() ✓
│  │  ├─ expIsNotCommand() ✓
│  │  ├─ negIsNotCommand() ✓
│  │  ├─ namedConstantIsNotCommand() ✓
│  │  ├─ logIsNotCommand() ✓
│  │  ├─ cosIsNotCommand() ✓
│  │  ├─ constantIsNotCommand() ✓
│  │  ├─ multiplicationIsNotCommand() ✓
│  │  └─ varsIsCommand() ✓
```

4.3 Protected Branch Control System



Default branch

main Updated 1 hour ago by Darkfrobozz ✓ Default

Your branches

feature Updated 1 hour ago by Darkfrobozz 0 | 2

Active branches

feature Updated 1 hour ago by Darkfrobozz 0 | 2

feature-scope Updated 8 hours ago by Grimrath 10 | 1

Our main branch in this example is protected, so pushing is not allowed to main, one must make a pull request and the pull request will not go through if no one has reviewed it and if the tests fail.

5 Conclusion

This essay is to serve as an introduction to CI, providing the theoretical narrative of what CI is made of and what is necessary for CI to work correctly.

The essay has highlighted the building blocks of CI that is testing and control. It has explored what kind of testing is relevant in modern development and how control is decided, of special interest has been the exploration of the often undervalued role of manual reviews and we can create the correct environment to foster this aspect of CI. This venture is undertaken because we need to discuss the human side of computer science more. [4] Moreover, the interplay between automated and manual tests has been explored to establish that effective CI is not one or the other but both since they complement each other not only by filling different roles but also by filling new roles together.

References

[1] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. “The Silent Helper: The Impact of Continuous Integration on Code Reviews”. en. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb. 2020, pp. 423–434. ISBN: 978-1-72815-143-4. DOI: 10.1109/SANER48275.2020.9054818. URL: https://ieeexplore.ieee.org/document/9054818/ (visited on 11/29/2023).

[2] Bertrand Meyer. “Seven Principles of Software Testing”. In: *Computer* 41.8 (Aug. 2008). Conference Name: Computer, pp. 99–101. ISSN: 1558-0814. DOI: 10.1109/MC.2008.306. URL: https://ieeexplore.ieee.org/abstract/document/4597151#:~:text=The%20principles%20that%20follow%20emerged,0814 (visited on 11/25/2023).

[3] Caitlin Sadowski et al. “Modern code review: a case study at google”. en. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. Gothenburg Sweden: ACM, May 2018, pp. 181–190. ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183525. URL: https://dl.acm.org/doi/10.1145/3183519.3183525 (visited on 11/29/2023).

[4] Kishore Senthil. *Behind the Screen: The Human Side of Software Development*. en-US. Aug. 2023. URL: https://www.kovair.com/blog/behind-the-screen-the-human-side-of-software-development/ (visited on 11/29/2023).

[5] *Standards in software development and 9 best practices*. en. URL: https://www.opslevel.com/resources/standards-in-software-development-and-9-best-practices (visited on 11/29/2023).

[6] “Version Control System: A Review”. en-US. In: *Procedia Computer Science* 135 (Jan. 2018). Publisher: Elsevier, pp. 408–415. ISSN: 1877-0509. DOI: 10.1016/j.procs.2018.08.191. URL: https://www.sciencedirect.com/science/article/pii/S1877050918314819 (visited on 11/29/2023).