Apps Factory

Development Operations – Code Review

Motivation

This proposal discusses the merits of the peer review system and the value it provides to rapid-development and startup teams. Not only is the intended purpose to suggest best practices, but more importantly, help establish a framework to analyze the various solutions that are available and help to support those recommendations. The goal of peer review is to improve the quality of the code being developed which increases the value proposition of the overall product.

While the focus of this document is to look at code review systems, such solutions may not always fit with the intended audience so a methodology known as “Pair Programming” is also examined to provide a more robust analysis to aid in the decision making process. Once the case for a code review system has been established, we’ll suggest various value-driven recommendations when developing in Git.

Why Peer Review?

Given the various set of standards (ie IEEE Std 1028) and certifications that are available to which large enterprises aspire to achieve when it comes to code quality, peer review is an important part of the software development life cycle in such environments. In a startup environment, there lacks pressure for companies to adhere to such standards, and with time pressures being even more obvious, there is more bias against peer review. In my experience, while development wants code review, management thinks its too expensive and not worth the effort.

One argument that counters this train of thought is the concept of Technical Debt[i]. One can view Technical Debt as work that needs to be done before a particular job can be considered complete. As the product evolves and progresses through the lifecycle (and incurs “interest”), it increases in complexity and if not properly maintained, the technical debt rises. Here is an interesting analogy that ties in technical complexity and debt:

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation" [ii]

The next two sections go into detail on the various forms of peer review. While pair programming focuses on the way the code is written, there is opportunity to integrate the processes of code review with this methodology. But before you begin, the basic pre-conditions are needed if a successful peer review system is to be implemented:

- At least one developer that adheres to principles of good architecture design[iii]
- 1 coffee break worth of time a week

Pair Programming – Is it enough?

The concept of pair programming implies that you have two developers work on the same machine. A driving analogy can be applied to help explain this concept: The driver codes, the navigator is reading, checking, and sanity testing the code while deciding how to tackle the next problem.

Various advantages stem from this way of development:

- ✓ Mix and match different specialists which allows for skill transferring
- ✓ Better awareness of code architecture which reduces code maintenance downstream and allows team to focus on product develop rather then fixing bugs
- ✓ Bigger the team, the more opportunities to encourage communication and sharing of ideas which results in more innovative products
- ✓ Quality of Code – Writing code beside someone is a perfect motivator not to be lazy and write bad code.

The following needs to be considered when looking at peer programming:

- ✓ Do you have enough resources for your project to implement this methodology?

- ✓ What is the personality mix of your team? Are your developers able to effectively work with others?
- ✓ What is the technical mix of your team? Do you have developers who are specialists in their own area? Or do you have a team of developers who are broad in their knowledge and could utilize the economies of knowledge from working with other developers?
- ✓ How stable is your team's structure? Is the retention rate low enough you can utilize the knowledge sharing that comes from peer programming?

Git Best Practices – Foundations for tooling

- ✓ Use Feature Branches
    - One of Git's main advantages is cheap, fast, local branches. If you're working on implementing a new feature or fixing a bug, you can separate that work into its own branch until you're ready to merge it back into your developer branch
- ✓ Post-Commit Reviews
    - The flexibility of feature branches, allows developers to work on other code while the changes are being reviewed. while post-commit reviews provide metadata in Git to allow for tracking and to revert work when changes are needed to be done.
- ✓ Commit rework in your feature branch
    - If there are defects in your review, you'll need to rework your code to fix the defects before the review can finish. Perform these reworks in the feature branch you used to create the original review. This ensures that your changes are always based on the code that is under review
- ✓ Prefer mergers over rebasing
    - When your review has been accepted, it's time to include your changes in your master branch. This can be accomplished in two ways: Rebase or Merge. When you rebase, Git creates new commits that have the same code changes and metadata except that the commit IDs are different. This means that, when you've committed a change for review, the rebased commit will not

match this and it could make it difficult to tie code reviews to Git commit

Disclaimer: While some of these points are often the topic of heated conversation, some of these practices don't apply to all startup teams as it largely depends on context!

Code Review – Available Open Source Tools

ReviewBoard (http://www.reviewboard.org/)

Supporting both concepts of pre/post-commit, ReviewBoard works well when a team has a central "official" git repository (ie GitHub). ReviewBoard requires a team to download and set a local instance of the solution. The basic workflow that it supports is the following:

- ✓ Clone the central repo
- ✓ Create a local branch and make a change you want reviewed
- ✓ Run post-review(a ReviewBoard tool), comparing the branch to master
- ✓ Get reviews, make additional changes on the branch as needed
- ✓ When the change is marked to ship, merge it into master and push it to origin

What makes ReviewBoard different is that it supports the concept of pre-committing changes to be reviewed before they are committed to a branch. A more strict way of enforcing code quality, it can be used as a means to enforce code stability in a repo. By using pre-commit git hooks, here's the workflow it supports:

- ✓ Clone the central repo
- ✓ Make a change you want reviewed, but do not commit it yet
- ✓ Run post-review
- ✓ Get reviews, update your change as needed

- ✓ When the change is marked to ship, commit it to master and push it to origin

While the flexibility of ReviewBoard allows it to work with any workflow, it does provide two constraints:

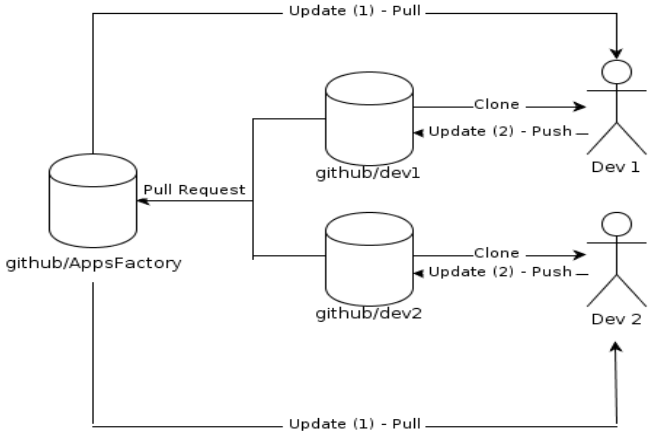
- It doesn't support any plug-ins for continuous integration tools such as Jenkins and Hudson which help to verify code quality(see verification step for Gerrit).
- Complex customization is required to support forked repos as suggested by GitHub

GitHub (https://github.com/)

A web-based hosting solution for Git-based projects, GitHub offers features found in most code-review tools:

- ✓ Easy-to-use dashboard
- ✓ Detailed review request
- ✓ Powerful diff viewer with inline commenting
- ✓ Contextual discussions and reviews which can be attached to commits

It promotes the forking mechanism (a local copy in a user's own github account) as both a lightweight workflow and a way for teams to stage their code. In other words, to review and assemble code before it's ready to be in production. One example can be described using this model:

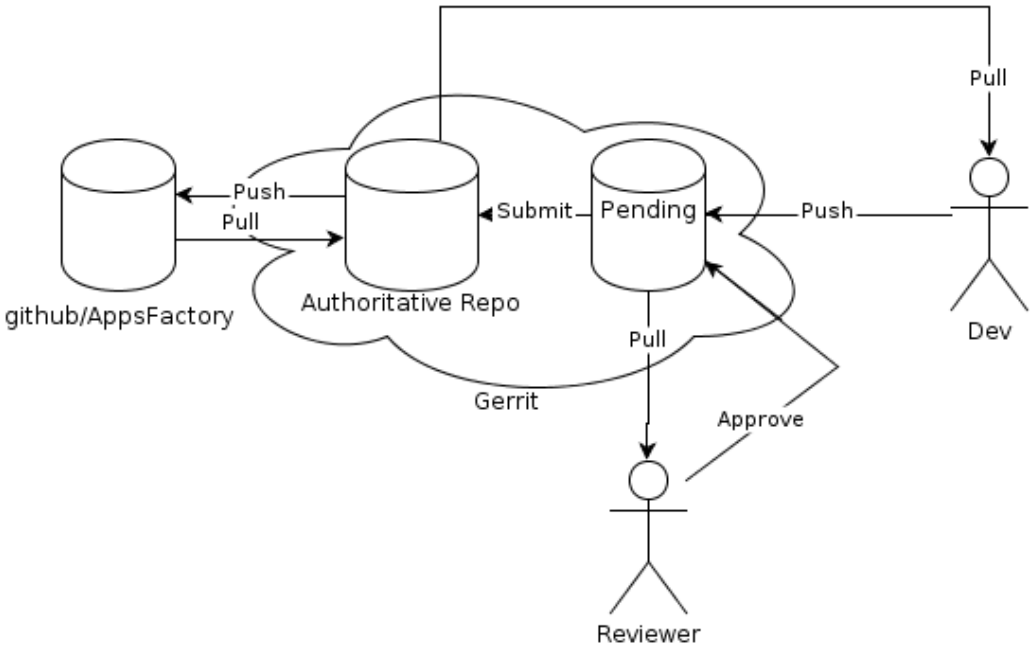


github/dev1 and github/dev2 are forked repos of github/AppsFactory. A forked repo usually is not expected to be merged back upstream

Within this context, GitHub also advocates the use of the "Pull Request" feature which can be seen as a light implementation of the final step of code review while providing some administrative control at the same time. A drawback to GitHub is the cost. While free to use, the free service offering only applies to publicly accessed repositories, whereas, if there is a need for private repos this comes at a cost. Given this, however, its offerings are quite comprehensive and a worthwhile investment for any team.

Gerrit (https://code.google.com/p/gerrit/)

Originally designed for the Android project, Gerrit is a neatly packaged solution that offers much more then just a code-review dashboard. Similar to GitHub where it offers an intermediary staging step, it requires a team to have it's own internal server (like ReviewBoard) to run this tool. Here is an example workflow:



Similar to GitHub, Gerrit has the ability to use various plug-ins that support continuous integration solutions such as Jenkins (http://jenkins-ci.org/) for

improved testing performance. Here is the following two-step process that occurs in the "Pending" state within Gerrit:

1. Review: By design, this step would require a gatekeeper looking at the code and ensuring it meets the project guidelines. As an example, a product owner/manager to sign off on the patch.
2. Verification: This step ensures tests have passed and that the code actually compiles. This step can be done manually by the gatekeeper or automatically through continuous integration tools such as Jenkins or Hudson.

As with all tools presented in this article, there isn't a "best tool": Trade-offs need to be considered and often times just trying out a tool and see how it works with your team is the best way to determine which is the most appropriate!

---

[i] Technical Debt Primer - http://en.wikipedia.org/wiki/Technical_debt
[ii] Ward Cunningham (1992-03-26). "The WyCash Portfolio Management System"

[iii] Really good summary of badly designed software: http://devlog.info/2009/09/16/8-features-of-badly-designed-software/