Engineering Development Through Changes

Engineering teams develop code describing systems from both small, self-contained modules to large systems built from many sub-components. This document describes the perspectives and prominent details of collaboratively engineering such code via a process of incremental changes.

Pull Request Processes

As with any form of engineering, it is standard practise (and indeed essential) for all produced works go through a review process before being accepted by the wider team, business group, and company. A pull request is simply the term for asking one or more people to check that some changes you've made to something are acceptable, i.e. review, before those changes are made permanently.

Pull requests are the standard, and most intuitive, method of working collaboratively with git, whether the requests are made formally or informally. A pull request can be raised in several ways, some of which have a more formal and trackable process than others:

- In person, e.g. Alice meets her colleagues in a meeting room and says "Bob, would you look at my work in `/path/to/repo` on branch `foo` and merge it into your branch `bar`, please." No history of this request is kept automatically.
- By email or instant message containing much the same comment, but some history is recorded.
- In project meetings, like how team leads will discuss which feature branches to merge into BigProject/master. The request is made during meetings, progress is tracked through a project-management tool, e.g. Atlassian's Jira Software), and history is kept through meeting minutes and/or internal boards, e.g. Wiki or Confluence pages.
- Using a specialised tool with a well-established formal process (BitBucket, GitHub, etc.) which allows a single user to propose a changeset, and multiple reviewers to comment, approve, or deny the request. These tools allow commenting to track progress on minor issues which are specific to a piece of work, e.g. "Need a better comment here." or "This constant should be 123.". A pull request made via these specialised tools is often referred to as a PR to distinguish it from less formal requests.

The term "pull request" is usually used in the context of developing text-based code using the git version control system. Other areas of engineering use different terms like "Engineering Change Order" with different tools and processes, but the concepts are equivalent.

Different Methods of Communication, Same Problems

Difficulties in these processes is the same whether a pull request is made via informal project planning meetings or formally through a specialised tool like BitBucket. The developer proposing the changes must make their work understandable by the reviewers who eventually merge the changes into a more authoritative branch. The only difference is the communication method by which developers and reviewers reach an agreement.

Developer vs Reviewer Perspectives

Reviewers are not expected to be deeply involved in the development process, and this is *a good thing*. If a design is only reviewed by its designers, that's simply a continuation of design work. Instead, reviewers should provide a fresh set of eyes, concerned only with the end result, and undistracted by intermediate decisions or politics amongst competing designers/testers.

As reviewers are not supposed to be too involved in the development process, it is essential that developers ensure their request is easily understandable to a fresh set of eyes. Therefore, developers must keep in mind how their work (in the form of a PR) will be seen by others and follow best practice guidelines. When a reviewer chooses to accept a PR and have it merged into something they have responsibility for, they are putting their seal of approval on the work. This is effectively making a statement like "I accept some responsibility if this changeset breaks something important." - which should not be done lightly. If reviewers cannot easily understand the changes in a PR, they will (should) rightly refuse to approve it until their concerns have been satisfied.

Guidelines and Other Resources on Pull Requests

Tutorials on all aspects of git are widely available online, but these are some particularly useful pages, written by Atlassian (parent of BitBucket and Jira):

- https://www.atlassian.com/git/tutorials/making-a-pull-request
- https://www.atlassian.com/git/tutorials/using-branches
- https://www.atlassian.com/git/tutorials/comparing-workflows

This is a summary of the most important guidelines:

- Write well-formed commit messages and provide a well-considered summary with your PR. Leaving technical reasons aside, reviewers are human and first impressions do count - so giving your reviewers a bad impression is not a good idea. Let them see that you've paid attention to detail and that your changes are going to have a positive effect.

- Use proper formal English (capitalisation, punctuation, spelling, grammar, and all) in all comments and discussions to reduce the risk of miscommunication. This is especially important when others working on the project have different backgrounds and might interpret idioms, slang, and vague language in unexpected ways.
- Stick to conventional terminology and avoid inventing unnecessary acronyms or abbreviations. For example, introducing "ADV" and "DIS" (for advantages and disadvantages) is unnecessarily confusing where the de-facto terms "pro" and "con" are readily understood. Similarly, use well-established placeholder names (if you need them) like foo, bar, Hello world!, Alice, Bob, Charlie, Eve, etc. Readers should not be distracted by unusual terms, or worse, waste time deciphering if names like zEbra and Consuela are somehow special.
- Use capitilisation carefully everywhere, particularly when referring to technical items (`foo` is quite different from `fOO`). You (and others) may want to search for historical discussions referring to that important signal name, or automated tools may scan for key phrases.
- A PR should only contain the minimum required changes. Mixing in unrelated changes makes it difficult for a reviewer to understand why those changes are necessary, and causes frustration when they realise the futile waste of effort. If a PR has a name like "Improve code comments", then there should be no changes to logic or other things like PDFs or anything else that isn't specifically improving code comments.

Additionally, there are some guidelines on the format of a commit message. These are discussed by two prominent contributors to git (https://git-scm.com/docs/git-commit#_discussion), and have even been used in a popular online comic (https://xkcd.com/1296/). Several sets of guidelines are available online, but this is a sample set:

- Limit the subject line to 50 characters.
- If 50 characters is not enough to explain the changes, then write a full explanation in the body.
- Separate the subject line from the body with a blank line, i.e. the 2nd line should always be empty.
- Hard wrap the body at 72 characters.
- Explain why changes are needed.
- Explain what changes have been made.

How a Reviewer Works

There are 3 main ways that a reviewer can gain an understanding of the changes in a PR, listed in order of preference:

1. Review the overall differences. This is the simplest and least-effort method and relies on the changeset being minimal.

2. Review the commit history. This is substantially more effort, but is sometimes necessary when the overall differences are too large/messy to make sense of. A reviewer will first scan the titles of commit messages in the history, then explore the body and diff of commits which look relevant. Commit messages must be properly formed with a title and body, otherwise this is a tedious and frustrating process. If the titles of your commits are like "fixes", "works better", or something equally uninformative, then a reviewer may decline the PR and request that you create another with a usable difference and history.
3. Review the overall result. Again, this is substatially more effort as the reviewer is required to setup a full development environment and painstakingly inspect every detail. While an involved designer might think it's okay to have a quick scan through the relevant files, the position of a reviewer is quite different. A reviewer may be opening an unseen piece of work, without involved knowledge of the design process, and being asked to verify that it is free of errors - all without any real base of comparison. A reviewer should *rarely* be expected to use this method to approve a PR.

Working with Git Branches

Git is a tool specifically designed to facilitate distributed collaboration on multiple parallel pieces of work. Branches allow a developer to work on one set of changes, and quickly switch back and forth between other sets of changes. Branches are also a convenient way to work with pull requests. A PR is raised when work on a branch is near completion and the developer wants reviewers to look at their work before merging it into a more authoritive branch, usually called `master`.

Single Branch

When there is only a single set of changes being developed at once, the PR process is the most simple: The developer raises their PR, the reviewer can easily see the changes, and they can work together until the work is in an acceptable state to be merged.

Independent Branches

When there are multiple branches of work on independent parts of a repository, the developer may need to periodically merge changes from the base branch (usually `master`) into the development branch. When the changes between PRs are independent, this is an easy process using the `git merge` command. This ensures that development branches, and the eventual PRs, do not diverge too much from their base branches.

Dependent Branches

A developer should not raise parallel PRs for dependent branches and expect reviewers to work out the differences and/or ordering requirements. Given that a PR is supposed to be a self-contained piece of work, a reviewer should be able to select any PR on their to-review list and expect it to be readable. Reviewers may have 10s of PRs to review at any one time, so it is not acceptable for a developer to demand that open PRs are reviewed in a specific order. Instead, when wishing to have work merged in through serial stages, i.e. dependent branches, the developer should simply wait for their one outstanding PR to be accepted before raising the next. Note that this workflow only applies when the eventual piece of work (result of the sequence of dependent PRs) is so vastly different from the base branch that it cannot be sensibly reviewed as one piece. In the development of large complex systems, this may be unavoidable, hence the existence of branch planning meetings. However, simple IPs can (and should) avoid this type of situation through proper planning of the design and code structure.

Uninteresting Branches

A developer should not raise a PR for changes between branches that reviewers have no interest in. For example, Alice may be developing on her own branch `playground/alice_foo`, but Bob has no interest in reviewing those changes. However, Bob will certainly be interested in reviewing any proposed changes to `master` or `feature/bob_EssentialFeature`. If Alice really wants Bob to be aware of her progress, she may raise a PR clearly marked as a draft, but this should be in agreement with Bob and any other relevant reviewers.