Implementation

Alex Sverdlov
alex@theparticle.com

1 Introduction

The ultimate goal of a software development project is to have written code to solve a problem. Obviously there are a lot of important peripheral tasks such as requirements, design, architecture that must be done in order to ensure that the code correctly solves the correct problem.

2 Source Control

Source control refers to a set of tools/procedures that keeps track of source code changes. It is a good idea to use source control on all projects, of any size.

One example is `git`. It is an open source distributed version control system. The distributed part implies there is no central repository—and everyone has their own copy of the entire repository. Often folks still setup a central *hub* to act as a central repository—often these *hub*s are managed by other corporations, such as Microsoft, hosting *github*.

The idea of source control is that every change to the code creates a new version, which you can examine or revert to at a later stage. Changes are commited—expand the history of changes. Commits often have comments explaining the relevant details about the change.

Most tools (including *git*) support branches. The idea is that developers write/commit code in changes, and later merge those commits into the master branch.

Releases are often branched out from the master branch—such that a release can be subsequently itself branched out for any fixes/changes to the release branch (and subsequently merged back into the release branch, to potentially create another release branch).

2.1 Comments

Make commit comments meaningful: for example, you can tag the issue-id (jira link) so you can quickly figure out which branch/commit has a fix for a particular issue.

3 Configuration Management

Source control is just one part of keeping-track-of-things. There are various platforms that are much broader in managing what's built and running where on which platform, etc.

4 Code Review

It is a good idea to have regular code/approach reviews (with active participants). These do not have to be formal—find a coworker and ask them to review your code/approach.

5 Pair Programming

Pair programming works similar to online-code-review... where two minds are developing the same code (with 1 person doing the typing, but both people doing the thinking).

6 Modularize

Keep things small and interchangeable. Follow unix philosophy: small tools with a well-defined bullet-proof function—that can be combined in infinite ways to solve all sorts of problems.

This makes it easier to test and use your code, and lets you get confidence in its correctness: less chance of unexpected outcomes.

6.1 Plan for distributed environment

It is a good idea to plan for a distributed environment, even if your immediate project is centralized. For example, if you have a choice of two algorithms to solve a problem, try to select one that can be parallelized.

6.2 Cloud

More of an architecture decision, but try to build code that can easily shift to the cloud. That means building components that are losely coupled without using enviroment assumptions... for example, avoid using files directly: use some wrapper to store and retrieve information.