Version Control

Let there be collaboration
The Challenge

Modern systems are too large for one person to build, but collaboration is difficult!

- Coordinating tasks
- Communicating quickly and efficiently
- Keeping everybody up to date, all the time
- As the team grows, so does the organizational overhead
Collaboration in Software

Collaboration is a general term, and can mean many different things:

- Share codebase
- Manage tasks & schedules
  - Keep track of what needs to get done, by who and when.
- Coordinate with “outsiders”
  - Other departments, such as marketing, design or business
  - Business partners’ software team
- Spread knowledge
- And more ...
Today’s Focus

Sharing a code base

○ How does a team of software professionals work efficiently on a single codebase?
○ What problems do they face?
○ Which tools and processes do they use?
○ How do these problems and tools change over time?
What do we mean by “sharing a codebase”?

Let’s develop our requirements gradually ...
Fundamental requirement

Multiple coders should be able to edit a shared codebase.
Very Simple Solution

● Shared folder + some conventions:
  ○ Read/write source files directly
  ○ Communicate via email (or instant messaging) to avoid stepping on each other’s toes
Limitations

- **No backtracking**
  - What if someone accidently breaks the code?

- **No traceability**
  - Changes since you last looked at the code?

- **Reliability**
  - Miss an email and you might accidently overwrite your colleague’s work.
New Requirements

1. Shared codebase

2. History log, with the option to rollback to previous versions.
Simple Solution

- Use Google Docs (or any of its equivalents)
- Collaboration is built-in

Question: Where does this simple solution fail for software engineers?
Limitations

● Revision history of individual files
  ○ Cannot (easily) rollback the whole codebase to a point in time
  ○ Ex: foo.h and foo.c often change together

● No control over revision granularity
  ○ Auto-save is usually not ideal for coding

● No commit messages to go along with revisions

● Editor not suitable for coding
New Requirements

1. Shared codebase

2. History

3. Coding-specific requirements
   ○ Rollback the full codebase (not just individual files)
   ○ Commit when ready
   ○ Edit locally (using an IDE and/or text editor)
   ○ Define a snapshot of repo, namely a way of remembering version of every file at a point in time.
You get the point ... Requirements evolve gradually over time
History Lesson

- **sccs** built in 1972

- Focus on revision control for individual files
  - Main revision history for each file
  - Ability to restore file(s) to a given revision
  - Ability to define snapshots using labels
  - Coordinate via lock ➔ check-in ➔ unlock cycles
History Lesson Continues

- sccs has some serious limitations
  - Runs on a single machine
  - As teams get larger, blocking others is impractical

- Version Control Systems have evolved, in order to solve these problems...
Client-Server VCS

- VCS software is installed on a server.
- Code in an "official repository" on the server.
- Common workflow:
  - Checkout a copy of the code to your machine
  - Make changes locally
  - When the changes are ready (and tested), commit (i.e. save) changes back to the server
- VCS requires conflicts to be resolved, before they can be committed to the repository.
Client-Server VCS

- Use *branches* to keep things manageable
  - Branch out to create multiple versions of the codebase
  - Compare differences between different branches
  - Merge changes from one branch to another

- Examples:
  - Different branches for different versions of the product
  - Feature branches
  - Experimental branches
The Linux project quickly evolved into a very demanding project, in terms of version control.

- Many remote teams working on many different versions of the same codebase
- Different independent teams working (competing, even) to build same component. Conflict galore!
- Different time zones, languages and cultures
- All levels of competence and trustworthiness
Linux Project (before 2002)

- Different teams work independently on different versions, all based on some official repository.
  - Periodically, “catch up” with the official repo.

- When ready to contribute the code:
  - Send a patch file to Linus Torvalds for review
  - Linus (and other maintainers) fix problems and merge the patch at their discretion
What Do Patches Look Like?

diff -Naur base/foo.c changed/foo.c
--- base/foo.c 2015-09-13 10:51:35.000000000 -0400
+++ changed/foo.c 2015-09-13 10:51:17.000000000 -0400
@@ -1 +1,2 @@
void f(){ /*implement f*/}
+void f2(){ /*implement new function*/ }

diff -Naur base/foo.h changed/foo.h
--- base/foo.h 2015-09-13 10:51:54.000000000 -0400
+++ changed/foo.h 2015-09-13 10:51:48.000000000 -0400
@@ -1,2 +1,3 @@
//interface file
void f();
+void f2(); //add this definition

Do you think staring at diff output is a lot of fun?
Applying patches

- Nasty work, but someone has to do it.
  - Applying patches is tedious.
- Nevertheless, the Linux team was onto something important
  - Workflow for many autonomous teams.
  - Contributors package their changes, and send them as a single unit to the project maintainer(s).
  - Goal: Distribute responsibility/work among contributors, and prevent maintainers from becoming a bottleneck.
- This workflow evolved into what we know as *pull request*
End Of History Lesson

- In 2002, the Linux project started using **BitKeeper**
  - Distributed Version Control System - A new type of VCS that tackled the collaborations issues that the Linux team (among others) was experiencing.
  - Proprietary (and *not* open-source) technology

- In 2005, the Linux project switched to **Git**
  - Git was built by Linus Torvalds as a replacement for BitKeeper
  - The reason for the switch ... licensing disagreement
New Requirements

1. Shared codebase

2. History

3. Coders’ requirements

4. Better support for Linux style collaboration:
   - Independent versions (i.e. different repos), based on an official repo
   - Easily catch up with the official repo
   - Pull request
Solution - Distributed VCS

- No single “official” repository.
- Repositories (with their complete history) can be cloned at any point in time.
- Changes (aka commits) can be pushed/pulled between repositories.
- Every commit is a snapshot of the filesystem.
- A repo is graph of commits.
Advantages Of Distributed VCS

- Allow for Pull Requests
  - A better way to request someone to pull a given set of changes into their codebase
  - Easier for project maintainers to distribute duties (e.g. resolving conflicts) to team members
  - Essentially, collaboration is built into the version control system
Other Advantages Of DVCS

● Better *branching* and *merging*
  ○ Allow you to maintain multiple versions of the same codebase
  ○ Much easier when you think of snapshots, instead of changes to individual files.

● Work offline and push changes later
  ○ Ex: Working on an airplane.

● Control over commit history, which documents the evolution of your system.
Open Source Software

- DVCS has had a significant positive impact on Open-Source Software.
- In many OSS projects
  - Team members are remote
  - Different time-zones and languages
  - May be complete strangers (cannot assume trust)
- DVCS allowed us to build better tools for open-source software development.
History Lessons - Summary

- The landscape is constantly changing.
  - More demanding requirements lead to more powerful tools, which lead to even more demanding requirements ...
  - Ex: SCCS → CVS → SVN → Git → ?

- But one concept always stays the same - Professionals use tools to become more productive.
Version Control in CSC301

- **Git**
  - Becoming the *industry standard* DVCS
  - *Open-source*

- **GitHub**
  - Hosting service for Git repositories
  - Web-based toolset for code/project management.
  - Free for public projects
  - Industry standard for OSS development
Enough motivation, let’s see the tools ...