Lecture Notes
September 5, 2017

Introduction

Who am I? Prof. Gail Kaiser, home page, lab page

How to contact me about this course: use piazza, do **not** send email
(assuming you want me to read and respond)

Piazza is available from the web, from inside canvas (courseworks2), as a mobile app

Piazza supports:

- Announcements
- Private notes just to “kaiser”
- Private notes just to “instructors” (includes IAs)
- Other notes seen by whole class (preferred!)
- You can post as <your name> or anonymously for whole-class posts

What is this course? 4156 Advanced Software Engineering

This is an *introductory* course in software engineering, does not assume a previous course in software engineering - so why is it called “Advanced”?

- Old 3156 Introduction to Software Engineering targeted sophomores, replaced by 3157 Advanced Programming
- 4156 targets MS students and upper-level undergraduates who have completed 3157 or equivalent for *programming maturity*
- We don’t build on 3157 directly, but we do assume two or more years programming experience and knowledge of two or more mainstream programming languages
- If you want to take a truly “Advanced” course in Software Engineering, take 6156 Topics in Software Engineering, which will next be offered in Spring 2018
- Prof. Yang’s 6121 Reliable Software and Prof. Jana’s 4995 Secure Software Development are also advanced courses in software engineering

Canvas = courseworks2 (web and also mobile app)

- Lecture schedule, assignment deadlines, etc. posted in canvas
- Post all assignment submissions to canvas unless told otherwise
- Check canvas calendar, not just within course

Go through course home page in canvas

- ❖ Ewan Lowe and J.P. Morgan “coaches”

Textbook Head First Software Development

- *Other readings outside the book will also be posted, from “free” internet sources*
- This book was written for (novice) professionals, not as a textbook, it’s very easy reader
- Specific chapters/sections to read posted as canvas assignments
- This book is mostly about the process of developing and maintaining software
- It will not tell you how to use specific tools and technologies

Use google or your favorite search engine to find videos, tutorials, manuals, knowledge bases, help forums, etc. to learn about tools and technologies

- Stackoverflow is your friend
- In most classes, posting questions on the internet about how to do your homework is considered academic dishonesty, and punished
- But such questions are **allowed** and **encouraged** for this class, assuming you’re asking for help with a tool/technology
    - - you cannot ask someone to “do” your assignment, but you can ask “how to do”
- No internet queries (no books, no notes, no electronics) during quizzes and exams
- Otherwise, the department’s academic honesty policy applies as written, see http://www.cs.columbia.edu/education/honesty/

What is Software Engineering?

Software Engineering is **not** Programming

Many people consider themselves IT, programmers or web/mobile developers, but that doesn't make them *software engineers*

Programmers **Program**	**Software Engineers** **Build Software Products**
Well defined problem	Ill-defined problem
Alone	Teams
Hundreds/thousands LOC	Tens-thousands->X millions LOC
Submitted for grading	Runs in production
Fresh codebase	Existing codebase
No 3rd party code	Use anything that saves time
Stop and Restart to Change	Continual Availability
Low cost of errors	Potentially high cost of errors

Note there's nothing explicit above about testing (or static analysis, code review, alpha/beta users, or any other means of finding bugs) and then fixing those bugs, but testing (and subsequent debugging) as the hallmark of software engineers is implied by:

- Ill-defined problem - you won't get it right the first time
- Teams - even if everyone's code works perfectly in isolation, it won't work perfectly when initially integrated
- Ten-thousands->X millions LOC - there will be some bugs lurking in there somewhere
- Runs in production - users will do things never imagined by the developers and consider it buggy if the software doesn't work the way the users expected

- Existing codebase - some code may have worked once upon a time with earlier versions of the language, compiler, operating system, network protocol, database, hardware, etc. but cannot magically continue to work “as is” when the platform changes
- Use anything that saves time - third party libraries are always suspect
- Continual availability - for conventional business/consumer software, users have very limited patience in waiting for critical errors to be fixed, they will quickly switch to a competitor’s product that works right now
- Potentially high cost of errors - while this course does not address safety-critical software, where bugs can lead to human deaths and other disasters, bugs in conventional business/consumer software can lead to substantial financial, security and privacy losses

The one thing about software engineering courses that students everywhere hate most is testing (or any other means of finding bugs) - but the first thing almost every new software engineer does on his/her first job is test/debug someone else’s code.

If you do not want to learn how to rigorously test your own code and other people’s code, do not take this course … but do not try to get a job as a software engineer.

Top 15 Worst Computer Software Blunders

Pair Programming

Should be called “pair software engineering”, but “pair programming” sounds better
- applies to all phases of software products, not just coding
- requirements, design, testing, debugging, operations

What is *pair programming*?

Two people sit side by side at same computer (or remote desktop sharing)

- Take turns “driving” (typing) vs. “navigating” (continuous code review)
- Slide keyboard and mouse back and forth
- Switch roles at designated intervals such as 25 minutes
- The pair does **not** divide up the work, they do everything **together**
- If necessary for some reason to work separately on something, they review together

Who has done pair programming - In school? Outside school?

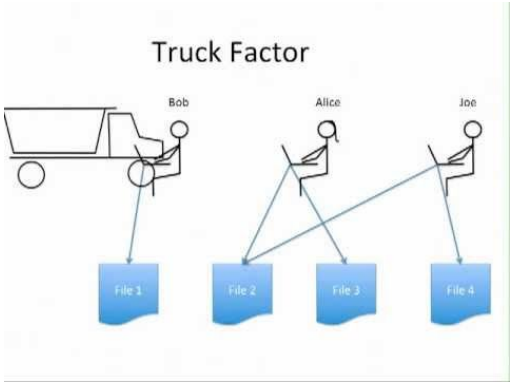
Many companies employ pair programming for interviews (auditions) even if they do not use for regular development

Workspace layout is important - see slideshare

Studies show higher quality, but mixed results on productivity
- Does it take twice as many person hours to do the same work?
- Hard to convince non-technical management

Organizations that use pair programming typically switch pairs often to limit risk

- *Collective code ownership*
- Maximize "**truck factor**" = number of engineers that would have to disappear before project would be in serious jeopardy



Other benefits of pair programming

- Shared responsibility to complete tasks on time
- Stay focused and on task - shared time treated as more valuable
  Less likely to read email, surf web, etc.
  Less likely to be interrupted
- Each member of pair expects other to follow “best practices”
- Two people can solve problems that one couldn’t do alone and/or produce better quality solutions
- Pool knowledge resources, improve each other’s skills

The dark side of pair programming.

Version Control

Also called configuration management, source code control

Who has never used a version control system?

“Weak” version control does not require any special system, besides regular file system

foo.bar
foo.old.bar
foo.bar.original
foo.bar.keepme
foo.bar_v2
foo.gek.bar
foo.gek_jsb_fhs.bar
foo.bar.old
foo.bar.older
foo.bar.latest
foo.23May17.bar
foo.v2.bar
foo.gek_jsb.bar

People create (or rename) files like these because they may need to later retrieve a file as it was at a certain point in time, or as it was before they (or someone else) changed it

- Keeps your work safe, can undo mistakes
- Ok for a small number of files and small numbers of versions
- Quickly becomes unmanageable with larger numbers of files/versions

Some operating systems and third-party tools (besides true VCSs) provide better file versioning

- Mac Time Machine
- Windows Backup and Restore
- Dropbox, google drive, etc. support access to older versions and history of changes, and enable access from multiple machines/devices, but you have to add your files explicitly
- Backblaze and other tools support internet backup of full file systems

When *multiple* people collaborate together on a file, or set of files, versioning becomes much more complicated

The tools above do not address coordinated *collections* of files

Imagine Microsoft or Apple employees sharing folders named Windows 10 or Mac OS X that everyone updates with their new code for the operating system, utilities, applications, etc.

- or sending changed files to their supervisors via email

Who has never used git? (Or svn, team foundation server, etc.)
Who has never used github? (Or bitbucket, sourceforge, etc.)

"Strong" version control provides a *file database* to coordinate source code and other project resources among teams of developers

"System of record" for code that goes into production (deployed to customers/users)

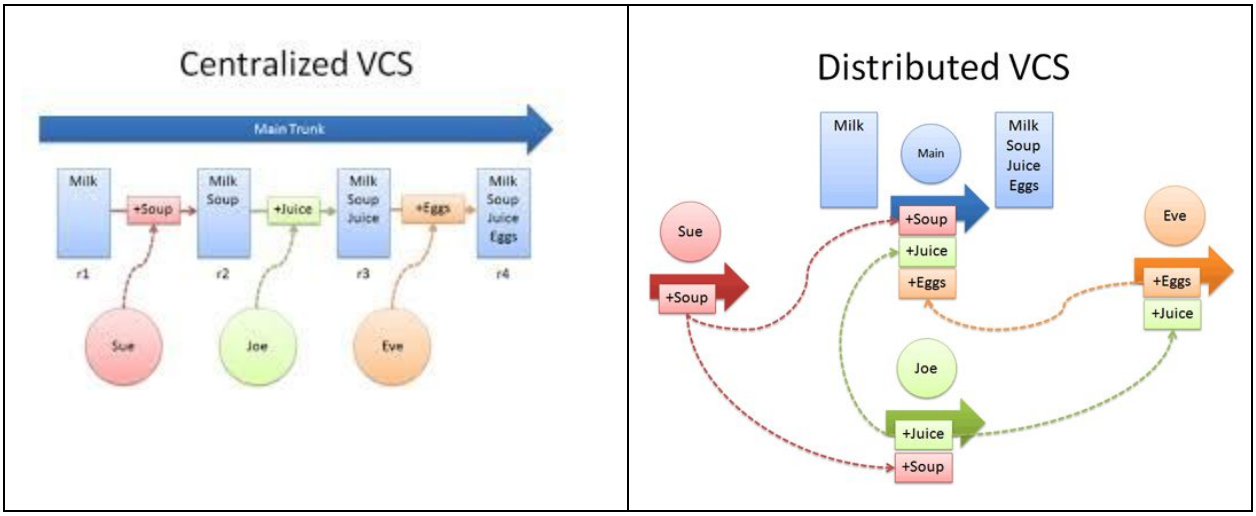
You need to know exactly what you shipped in order to reproduce and fix bugs

Users will report bugs in the deployed version that they are using, not the new version you are working on, not necessarily even the most recently shipped version

(recent news reports about hacker attacks made clear people are still using Windows XP, released in 2001 and not supported since 2014, but it **was** supported for 13 years!)

Logically centralized storage independent from developers' machines

May be physically centralized (svn) or distributed (git), or both (git server, github)



VCS supports automated builds (= *continuous integration*)

- External tools hooked to VCS through tool like Travis CI or Jenkins
- Includes at least compile/package of full application
  Developers do not need to remember detailed commands, options, etc.
- Often includes running full test suite
  (tests relevant to changed files should always pass locally before commit)
- Possibly static analyzers
  (always run locally before commit)
- Best to set up *pre-commit* hooks as well as post-commit
  (changed files should always be successfully built locally before commit)
- On every commit or nightly (some test suites take hours to run!)
- **"Don't break the build!"** - team alerted to any errors

Sometimes automated deployment (= *continuous delivery*) on demand or at preset times, to install within company (devops) or ship (external customer)

We will cover CI/CD in more detail later

Continue here

Basic functionality of most version control systems:

Backup and restore apply to coordinated collections of files, not just individual files

Synchronization among multiple users

- LOCO model - lock on checkout, unlock on checkin (need way to break lock)
- MOM model - merge on modify

Short term undo - return to last known good version

Long term undo - return to old version as of specific date, specific release, etc.

Track changes - commit messages with who, when, (hopefully) why

Repository (repo) = the file database

What is kept in a repo?

- Source, tests, scripts, resources, configuration
- Usually **not** executables or other generated files built from the stored files (these are what is "built" during continuous integration)

Server = where the repo lives

Client = developer machine

command line shell, special GUI client, file system snapin, IDE or code editor plugin

Working set/working copy = local file directory on client where developer makes changes

Main = master = trunk

- Primary set of versions in the repository
- Think of a tree (with branches)

Head = latest revision

Basic actions applied to a repository:

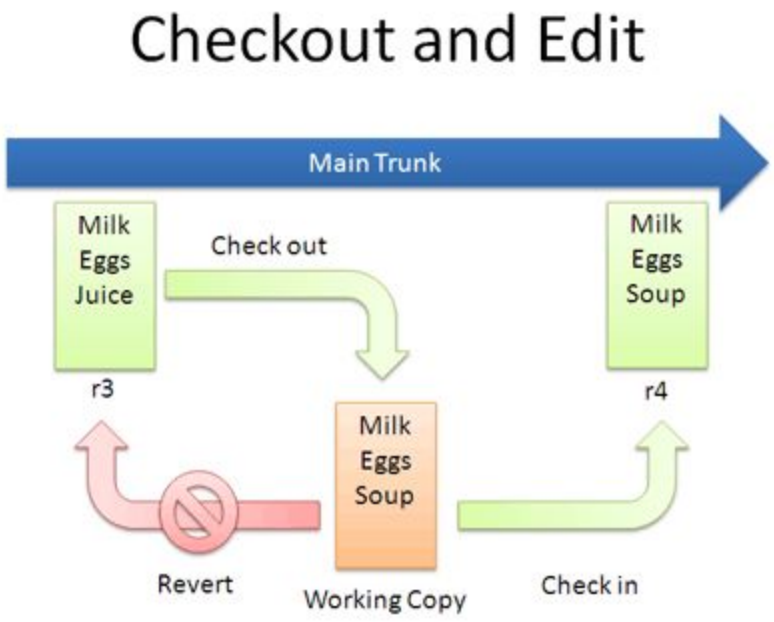
Add file to repo - doesn't work to just add to local copy, need to tell the VCS to track

Checkin/Commit/push - upload or copy from local working set to repo

- Updates revision number
- Records commit message and who made the change in changelog/history
- Might be integrated with issue tracker, e.g., associating a bugfix commit with the original bug report
- Here is where most of the problems arise!
- - Accidental or intentional overwrites of other changes

Checkout/pull - download or copy from repo to local working set

- Clone = initial copy
- LOCO (pessimistic) vs. MOM (optimistic)
- Limitations/problems with both, no perfect solution
- Some VCSs distinguish read-only vs. read-write checkout



Revert - throw away local changes and reload latest revisions from repo

Update/sync

- Get latest revisions of files from repo, may have changed since checkout
- Be careful not to overwrite any local changes (stash)

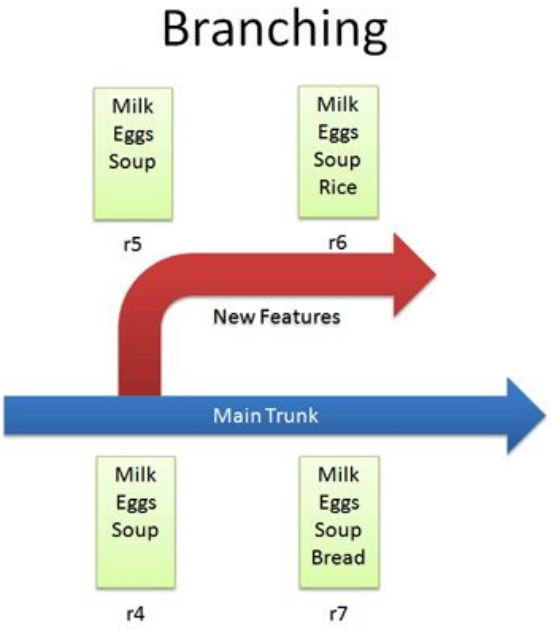
Advanced actions:

Tagging

- Not the same as revision numbering
- Different files may be revised at different paces, one file rarely changes while another is changed several times per day
- Tagging marks the set of all file revisions contributing to some distinguished milestone, e.g., demo or release
- May keep checkpoint snapshot

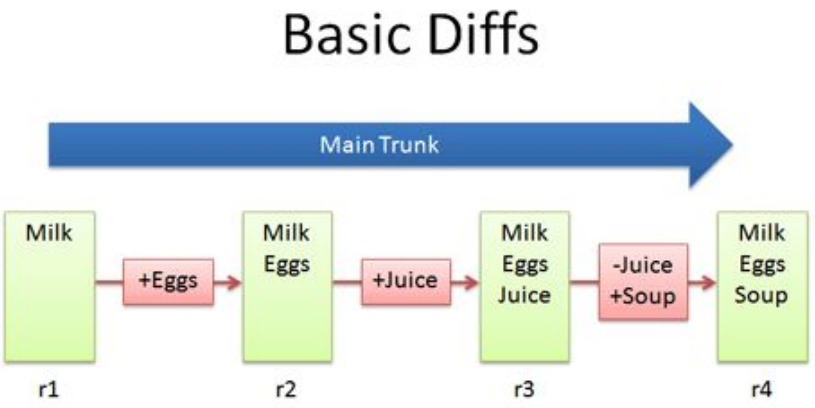
Branching and merging - branch is both a noun and a verb (the branch, to branch)

Fork copy of code base and track changes separately, may or may not later merge back to main line or with another branch



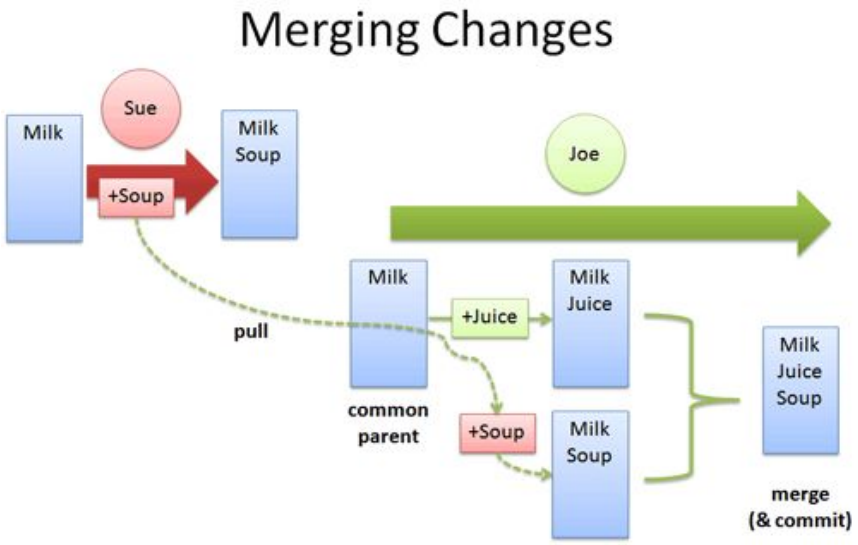
Diff/delta - find differences between two revisions of a file, usually sequential revisions

- Many “diff” algorithms used by different tools, helpful for merging
- Can also reduce storage required for saving many revisions
- Maintain original file, and then store only diffs from that revision to the next
  - to get latest version need to reapply all those diffs
- Maintain latest file, and then store only reverse-diffs to the previous revision
  - to get latest version just grab it, only need to reapply diffs to get older versions



Merge

- Integrate changes from working set into previous revision
- Also applies to full branches
- May be automatic when there are no conflicts



Conflicts may be detected automatically

- Usually in context of simultaneous changes by different developers
- When pending changes contradict each other
- Usually purely lexical (dumb), same line or same method is modified
- Ideally semantic (smart), requires sophisticated static analysis such as slicing

Resolution usually manual

- One developer integrates two or more sets of changes (good time to use pair programming even if not normally used!)
- Then commits corrected version

Team software development cannot function without a shared VCS

SCCS 1972 … RCS 1982 first generation (probably others used internally even earlier)
There is nothing novel or researchy about VCS, just use it!