Lecture Notes
September 18, 2018

First individual/pair assignment due before class today. If you had difficulty completing it, please see one of the IAs during office hours. Everyone needs to be able to use github (or equivalent - but you should only use an alternative if you know what you are doing!).

This assignment introduced the simplest form of version control - each team member's own local copy of codebase vs. shared codebase

Version control is employed by both waterfall and agile, and by nearly all software organizations (aka configuration management, source code control)

"Weak" version control does not require anything special, just regular file system. I have the patterns:

foo.bar	foo.bar.old
foo.old.bar	foo.bar.older
foo.bar.original	foo.bar.latest
foo.bar.keepme	foo.18Aug18.bar
foo.bar_v2	foo.18Aug18a.bar
foo.v2.bar	foo.gek.bar
foo.gek_abc.bar	foo.gek_abc_def.bar

People create (or rename) files with names like these because they expect they may later need to retrieve a file as it was at a certain point in time, or as it was before they, or a collaborator, changed it - this often happens when shared via email rather than a repository

- ➢Keeps your work safe, can undo mistakes
- ➢I have seen numerous demos where the demo didn't run at all, or started up and quickly crashed; the demo'ers *always* say "it worked yesterday" - but that was (usually) before something changed...

"Weak" version control ok for a small number of files, small numbers of versions, small number of collaborators - quickly becomes unmanageable with larger numbers of files/versions and more collaborators

Imagine Microsoft or Apple with thousands of employees sharing network folders named Windows 10 or Mac OS X that everyone updates with their new code for the operating system, utilities, apps, etc. - or sending changed files to their collaborators via email

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

- Mac Time Machine
- Windows Backup and Restore
- Linux backup commands
- 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 provide cloud backup of full file systems

Why aren't backup storage tools sufficient for large development teams?

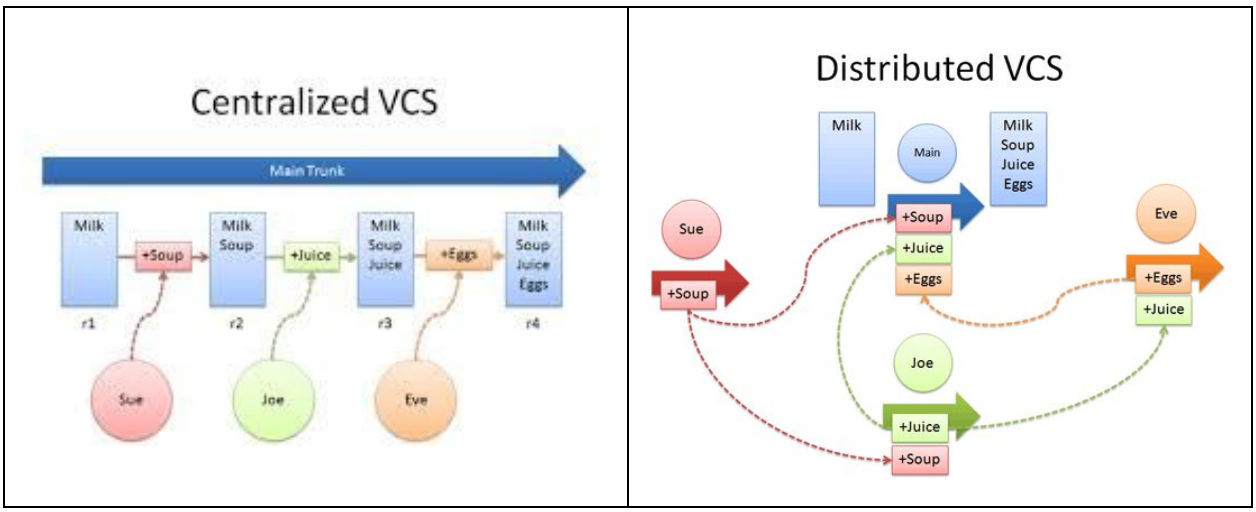
Today's lecture will, hopefully, be review for most of you. Team software development functions very poorly without a shared VCS - but, shockingly, version control is often omitted from undergraduate computer science education even though >40 years old (SCCS).

"Strong" version control instead 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

- You need to know exactly what you shipped in order to reproduce and fix bugs
- Users report bugs in the deployed version they are using, **not** the new version you are working on, not necessarily even the most recently shipped version (people were still using Windows XP in 2017 - released in 2001 and not supported since 2014, but it **was** supported for 13 years!)

File database is logically centralized storage independent from developers' machines - may be physically centralized (svn) or distributed (git)



Basic functionality of most version control systems:

Backup and restore applied to coordinated collections of files, not just individual files (all or nothing)

Term "revision" sometimes used instead of "version"

- ➢A revision usually refers to a changed individual file whereas version may refer to an individual file, a collection of files, or full application or system

Synchronization among multiple users

- LOCO model - lock on checkout, unlock on checkin
- MOM model - merge on modify

Why pick LOCO vs. MOM?

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 automatically record who and when, but human has to explain *why*

I have sometimes written commit comments with the content ".", "another change", "fixed", "need to fix later", etc. Why are these **bad** commit comments?	

Basic Terminology:

Repository (repo) = the file database

What is kept in a repo?

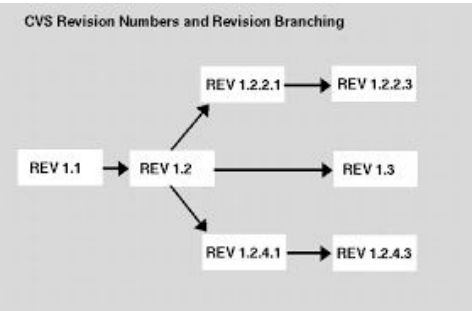
- Source, tests, scripts, resources, configuration
- Usually **not** executables or other files generated from the stored files (these are what is "built" during build and continuous integration)

Server = where the repo lives

Client = developer machine, via command line shell, GUI client, file system snapin, IDE 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 tree with branches

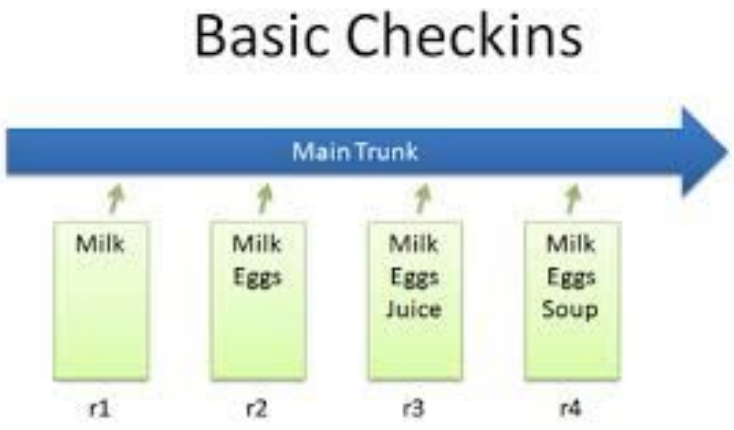


Basic actions applied to a repository:

Add file to local working set, add new or changed file to change-set (part or all of working set) intended to eventually be committed

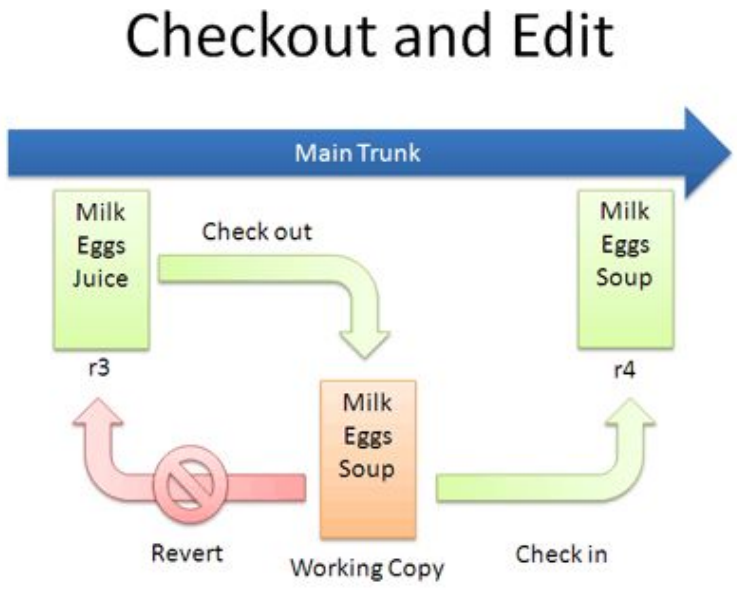
Checkin/Commit/Push - upload or copy change-set from local working set to shared repo

- Updates revision number of each file
- Records commit message and who made the change when in changelog/history
- Often integrated with issue tracker, e.g., associate bugfix commit with bug report
- Accidental (or intentional) overwrites of other changes, possibly by other developers if VCS doesn't automatically merge or detect conflicts



Checkout/Pull - download from repo to local working set

- Clone = initial copy
- LOCO (pessimistic) vs. MOM (optimistic)
- Some VCSs distinguish read-only vs. read-write checkout (allows some parallel work with LOCO)



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

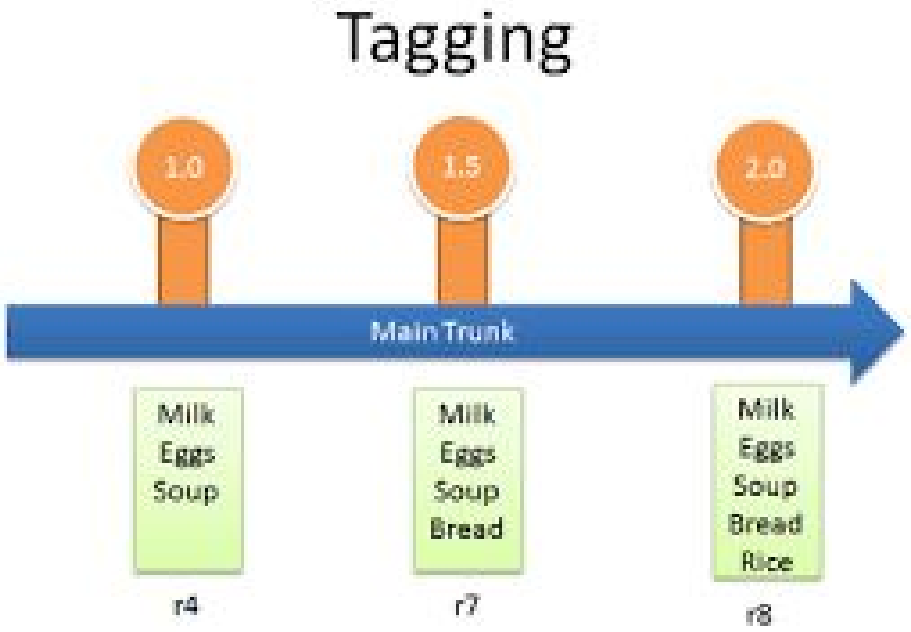
Update/sync

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

Some of the (many!) advanced actions:

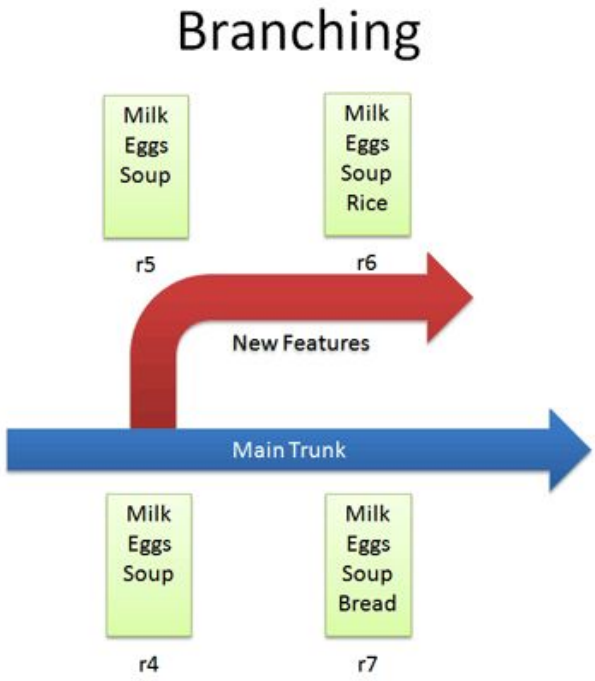
Tagging

- Not the same as revision numbering
- Different files may be revised at different paces, e.g., one file changes rarely while another changes 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 - branch is both a noun and a verb (the branch, to branch)

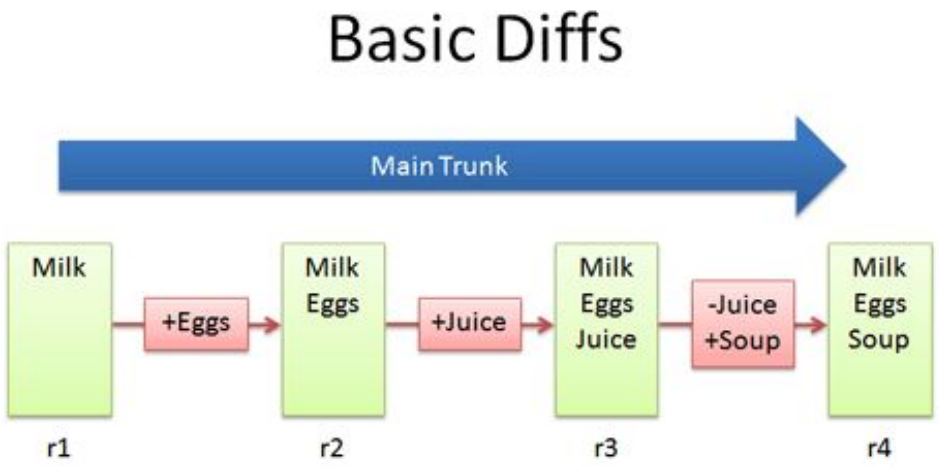
Branch - Copy (possibly copy on write) codebase and track changes to the copy separately, may or may not later merge back to trunk or with another branch



Fork - might be synonymous with copy, might mean something else … what?

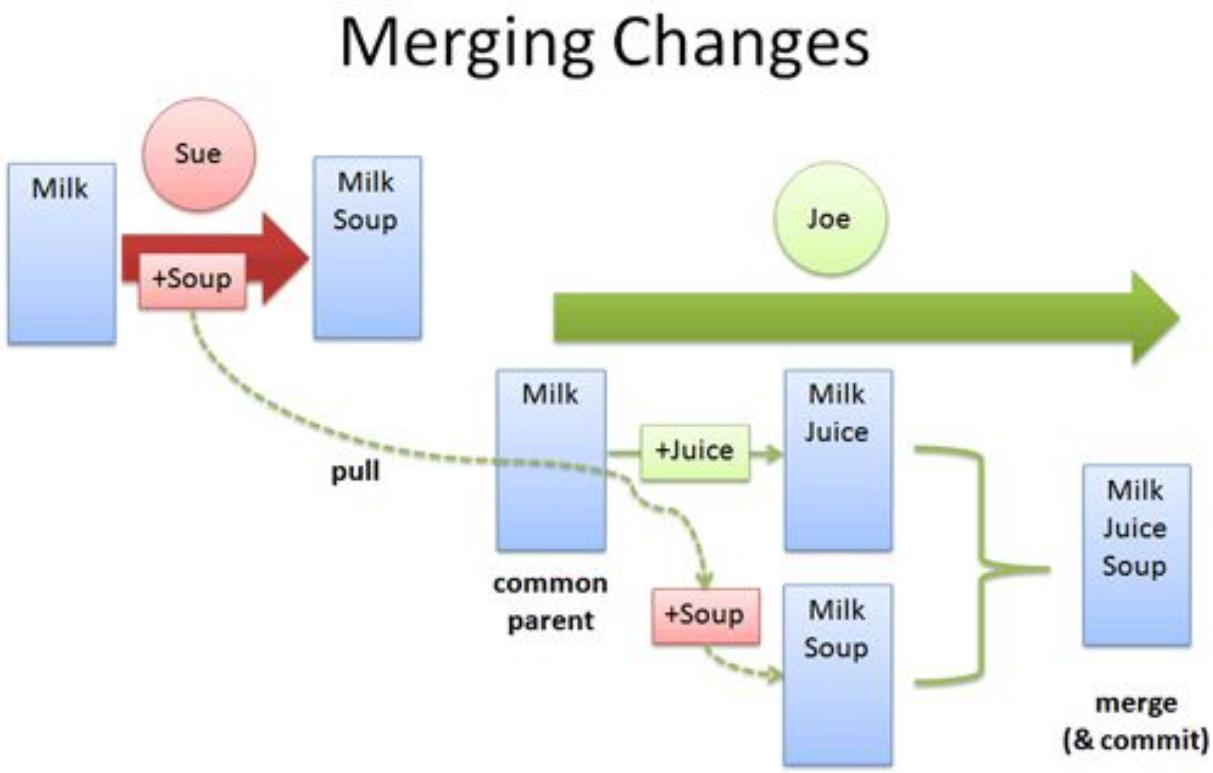
Diff (or delta) - automatically find differences between two revisions of a file, usually successive

- Many "diff" algorithms used by different tools, helpful for merging
- Can also reduce storage required for saving many revisions
- Keep original file, and then store only diffs from that revision to the next - to get latest version need to reapply sequence of diffs
- Keep latest file, and then store only reverse-diffs to the previous revision - easy to get latest version, need to apply reverse-diffs to reconstruct older versions



Merge

- Integrate changed files from working set with previous file revisions (usually most recent)
- Also applies to full branches
- May be automatic when there are no conflicts



Conflicts detected automatically

- Usually in context of simultaneous changes by different developers, when pending changes contradict each other
- Usually purely lexical, same line or same method is modified - in worst case same file

Original:

```
int main(){
    char* ptr;
    FILE* data;
    char buf[100];
    ...some code...
    return 0;
}
```

Two concurrent changes, how should they be merged?
No changed lines, only added lines

```
int main(){
    char* ptr;
    FILE* data;
    char buf[100];
    ….some code….
    data = fopen("data.txt", "r");
    if(fgets(buf,100,data) == NULL)
        return 1;
    printf("Read: %s", buf);
    fclose(data);
    data = fopen("data.txt", "w");
    fwrite("juice\n", 6, 1, data);
    fclose(data);
    return 0;
    }
```

```
int main(){
    char* ptr;
    FILE* data;
    char buf[100];
    ….some code...
    data = fopen("data.txt", "r");
    if(fgets(buf,100,data) == NULL)
        return 1;
    printf("Read: %s", buf);
    fclose(data);
    data = fopen("data.txt", "w");
    fwrite("soup\n", 7, 1, data);
    fclose(data);
    return 0;
    }
```

Original:

```
void display(char* str){
   write(1, str, strlen(str));
}
int main(){
   char* msg_ok = "Ok!\n";
   char* user = calloc(256, 1);
   char* pass = calloc(256, 1);
   strncpy(user, "default", 8);
   strncpy(pass, "default", 8);
   ...some code...
   write(2, msg_ok, strlen(msg_ok));
   display(user);
   return 0;
}
```

Two concurrent changes, how should they be merged? Both changed and added lines

```
void display(char* str){
   write(1, str, strlen(str));
}
int main(){
   char* msg_ok = "Ok!\n";
   char* user = calloc(256, 1);
   char* pass = calloc(256, 1);
   strncpy(user, "default", 8);
   strncpy(pass, "default", 8);
   …. some code...
   read(0, user, 255);
   read(0, pass, 255);
   write(2, msg_ok, strlen(msg_ok));
   display(user);
   return 0;
}
```

```
void display(char* str){
   write(1, str, strlen(str));
}
int main(){
   char* msg_ok = "Ok!\n";
   char* user = calloc(256, 1);
   char* passwd = calloc(256, 1);
   strncpy(user, "default", 8);
   strncpy(passwd, "default", 8);
   ...some code...
   read(0, passwd, 255);
   write(2, msg_ok, strlen(msg_ok));
   display(user);
   return 0;
}
```

Resolution usually manual

- Someone needs to integrate two or more sets of changes - good time to use pair programming even if not normally used!
- Then commit merged version

Show some other github features using
https://github.com/randoop/randoop

Reading for Thursday:
User Stories: An Agile Introduction

First team assignment: Everyone will need a team (of 4 students = 2 pairs) to develop your team project, so find teammates - talk to other students after class, post advertisement on piazza, etc. The team formation assignment is due Thursday, 11:59pm.

Important team formation considerations: Agree on same platform (Linux, Mac, Windows) and same programming language (Java, C/C++, Python). Make sure all prospective teammates have compatible schedules and locations for frequent team meetings (and even more frequent pair meetings).

You preliminary project proposal will be due Thursday, September 27, so it would be wise to brainstorm with prospective teammates about potential project ideas.

Your team project does **not** need to be a web or mobile app, which was required in recent previous years, and instead I discourage web and mobile apps.

Your project **does** need to be something that you will be able to demo - there needs to be a command line console or GUI.

Your project should also use some kind of persistent and structured data store (e.g., SQL, NoSQL, key-value store) and use some third-party library, framework or API (not just the built-in libraries that come with the programming language).

Some example APIs can be found here

If you do not have a full team before class on Thursday, talk to me at *beginning* of class, do not wait until the end of class

In-class exercise if time permits: "Mingle" to try to find a compatible team if you do not already have a full team of exactly 4 students