EECE 310 Version Control in Software Engineering

Presentation Slides

- Many requests for the slides
- Share on Piazza, attach (File Insert)

Learning objectives

- Explain why VC is an essential tool for SW development teams.
- Describe common VC features.
- Understand the differences between Distributed VC and Centralized VC, explaing advantages / disadvantages.

What is Version Control?

- The process of keeping track of different versions of software components and the systems in which these components are used. [Sommerville 9]

Why do we need VC?

Versioning

- Labeling changes made to software
- ‘Dependency hell’
  - The larger your software, the more pain
- Systematic? How?

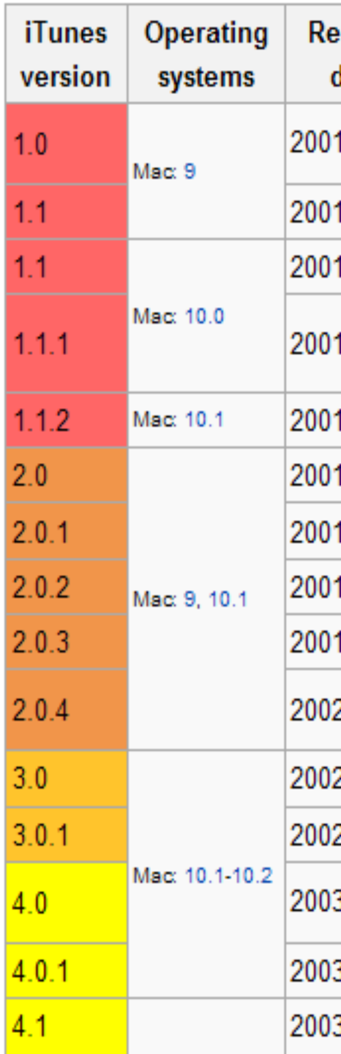
Semantic Versioning

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make **incompatible** API changes,
- MINOR version when you **add functionality** in a **backwards-compatible** manner, and
- PATCH version when you make **backwards-compatible bug fixes**.

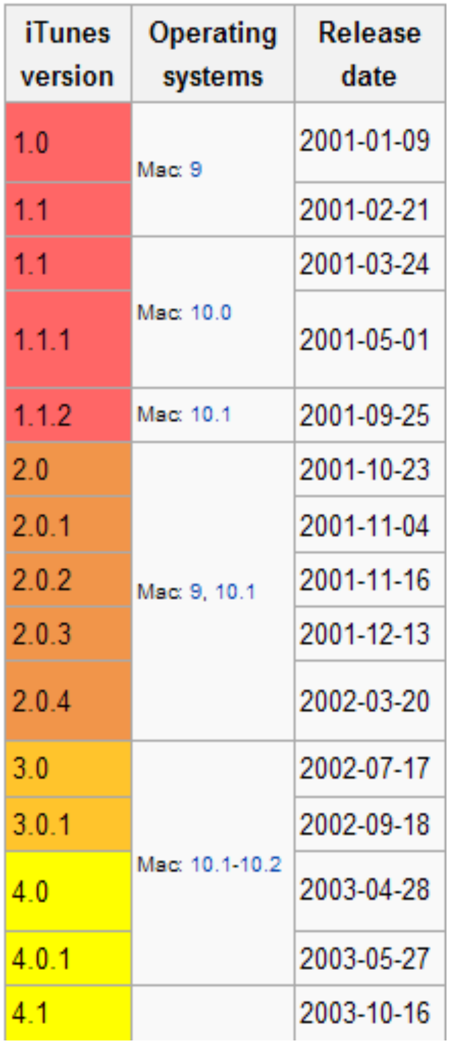
Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

http://semver.org/



iTunes version	Operating systems	Re d
1.0	Mac: 9	2001
1.1		2001
1.1	Mac: 10.0	2001
1.1.1		2001
1.1.2	Mac: 10.1	2001
2.0	Mac: 9, 10.1	2001
2.0.1		2001
2.0.2		2001
2.0.3		2001
2.0.4		2002
3.0	Mac: 10.1-10.2	2002
3.0.1		2002
4.0		2003
4.0.1		2003
4.1		2003

Motivation for Version Control



iTunes version	Operating systems	Release date
1.0	Mac: 9	2001-01-09
1.1		2001-02-21
1.1	Mac: 10.0	2001-03-24
1.1.1		2001-05-01
1.1.2	Mac: 10.1	2001-09-25
2.0	Mac: 9, 10.1	2001-10-23
2.0.1		2001-11-04
2.0.2		2001-11-16
2.0.3		2001-12-13
2.0.4		2002-03-20
3.0	Mac: 10.1-10.2	2002-07-17
3.0.1		2002-09-18
4.0		2003-04-28
4.0.1		2003-05-27
4.1		2003-10-16

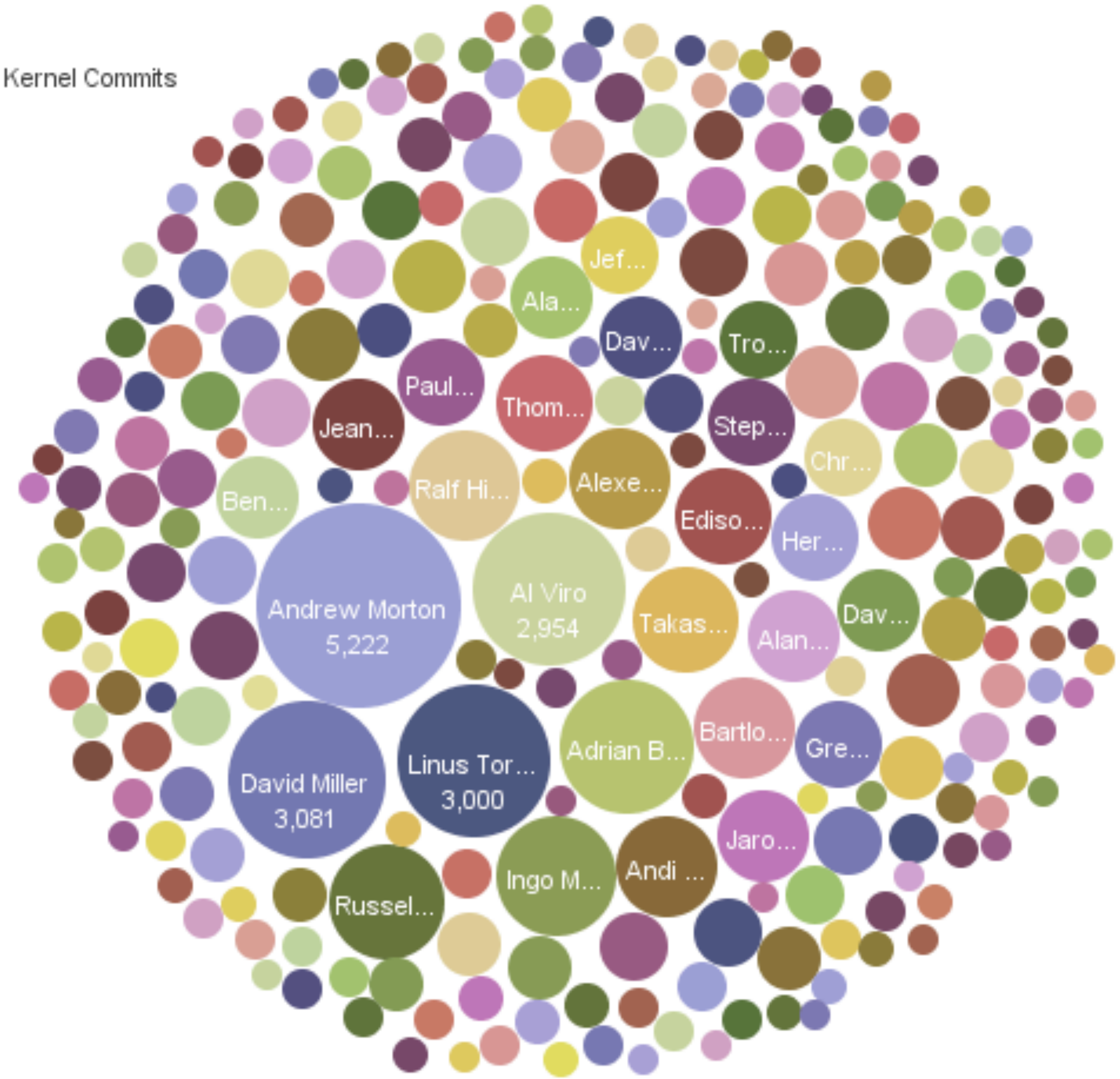
Software systems typically have many **public releases** and may have even more **internal versions**. These versions need to be managed.

Imagine how difficult it would be to manage these versions by simply storing multiple copies of the software (like Word).

http://en.wikipedia.org/wiki/ITunes_version_history

99621

Disks colored by Linux 2.6 Kernel Commits



- 5,000
- 3,000
- 1,000
- 0

To highlight or find totals

Bug localization

Software developers need to know exactly which revision of each file or component was released with each version so that they can find and fix bugs

Same code, multiple people

Multiple developers need to work on the same files at the same time. This could cause

- Overwriting of important changes
- Conflicts

when they are ready to share their changes.

Coordination Issues

- The situation:
  - One code base, Many coders
  - Everybody works in parallel
  - Therefore:
    - **Lines of code touched by different people**
    - **Individuals working on outdated code**
- The problems:
  - This worked yesterday, what happened?
  - I cannot reproduce the bug on my copy!
  - The system is broken! (a.k.a. Broken Build or Test Failures!)

Version Control

(a.k.a. Source Control, Revision Control)

- Manages multiple versions of the same unit of code
- Makes it possible to identify
  - **Which parts have changes**
  - **Who changed it**, and **when**
  - How does it **compare** or **conflict with my own changes**
- Can often automatically merge non-conflicting changes
  - Usually when changes occur in different parts of the file

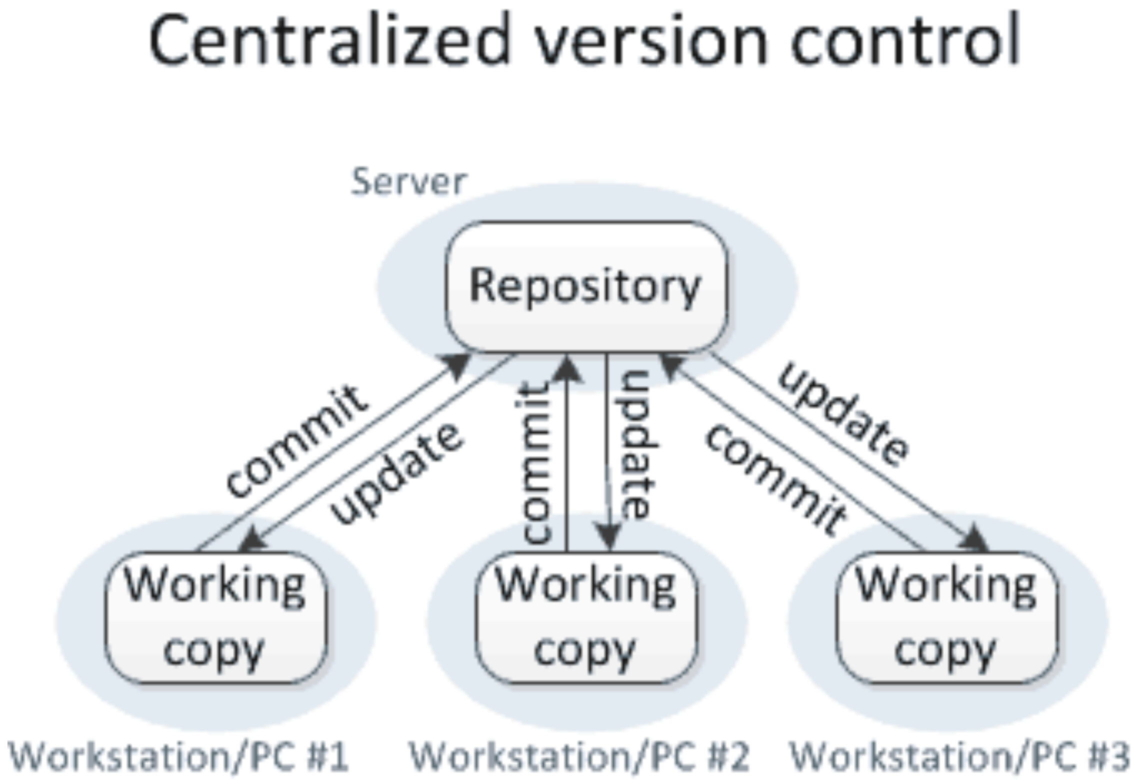
Version Control Tools

- Tons of them!
  - CVS
  - Subversion (SVN)
  - Visual Source Safe (VSS)
  - IBM Jazz / RTC
  - Mercurial
  - **Git**

Centralized Version Control

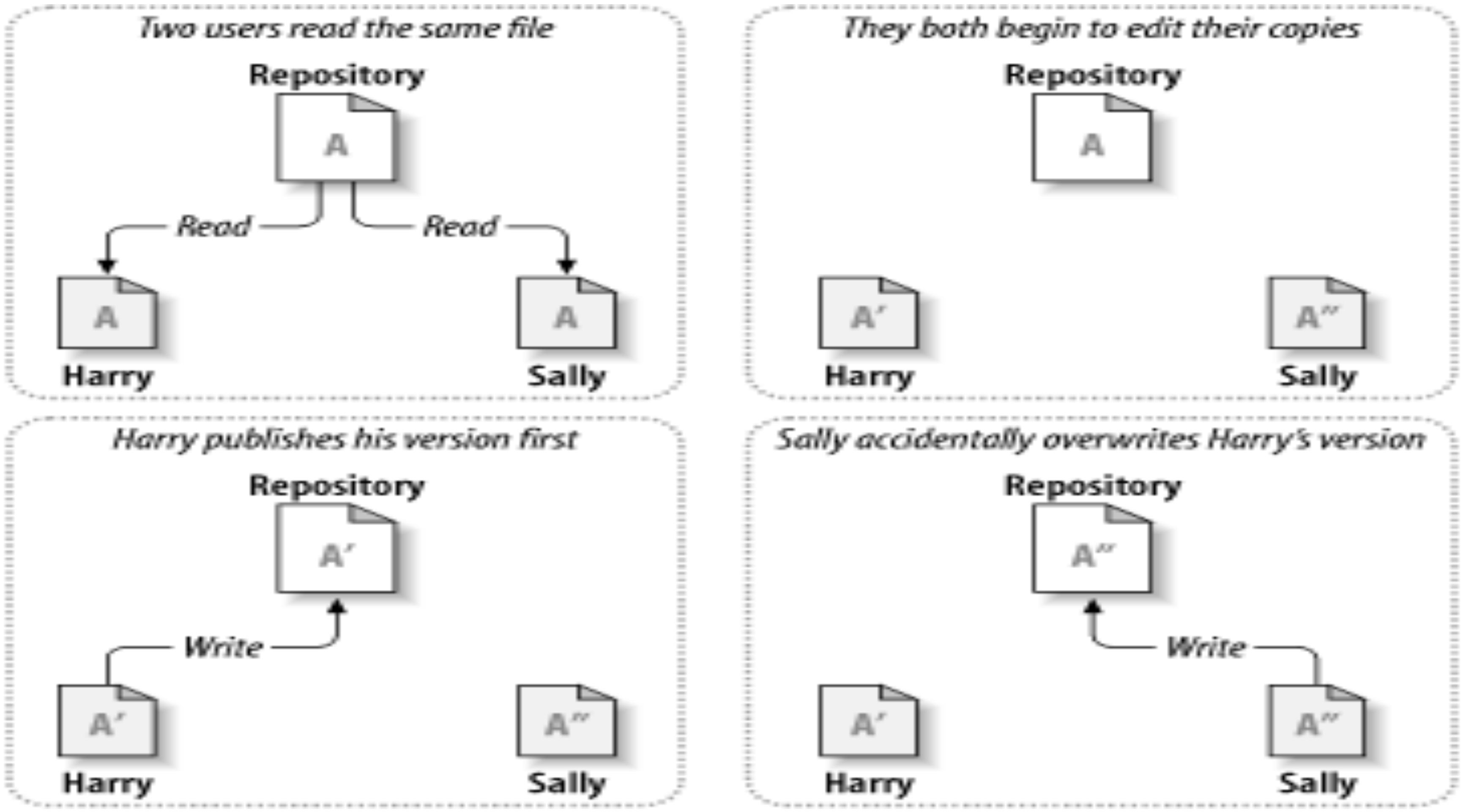
- Two basic operations
  - Check-out (e.g. svn chekout, svn update)
    - Take the latest code from the shared repository and move it to your own workspace
  - Check-in (e.g. svn commit)
    - Move the code from your own workspace to the shared repository
- Old systems were locking
  - Only one user can check-out a given file
- Today, multiple users can check-out the same file
  - Problem: Conflicts

Typical client/server system

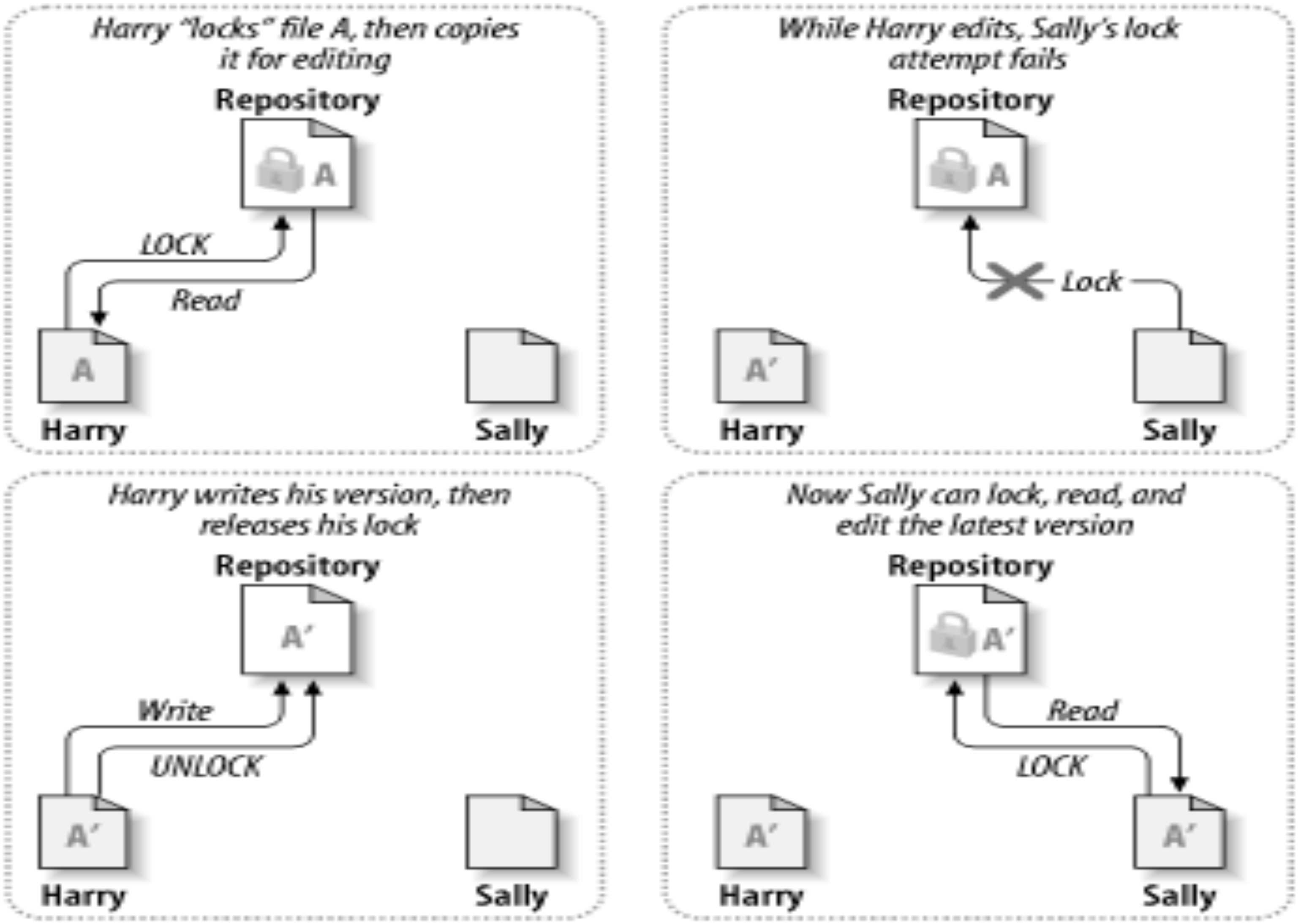


Source: http://homes.cs.washington.edu/~mernst/advice/version-control.html

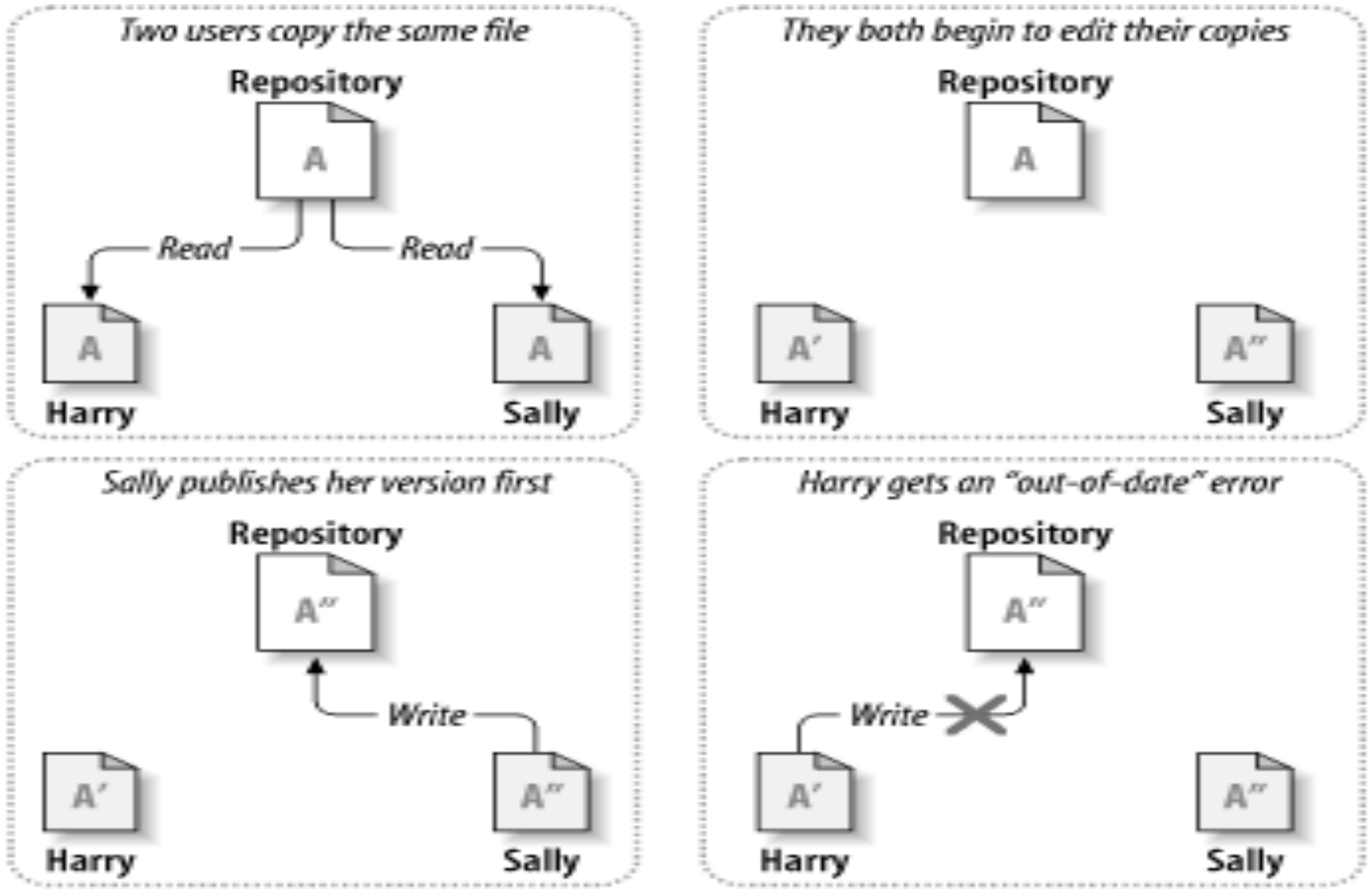
The problem to avoid: Overwriting


Two users read the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Harry publishes his version first
Repository
A′
Write
A′
A″
Harry
Sally
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
A″
Harry
Sally

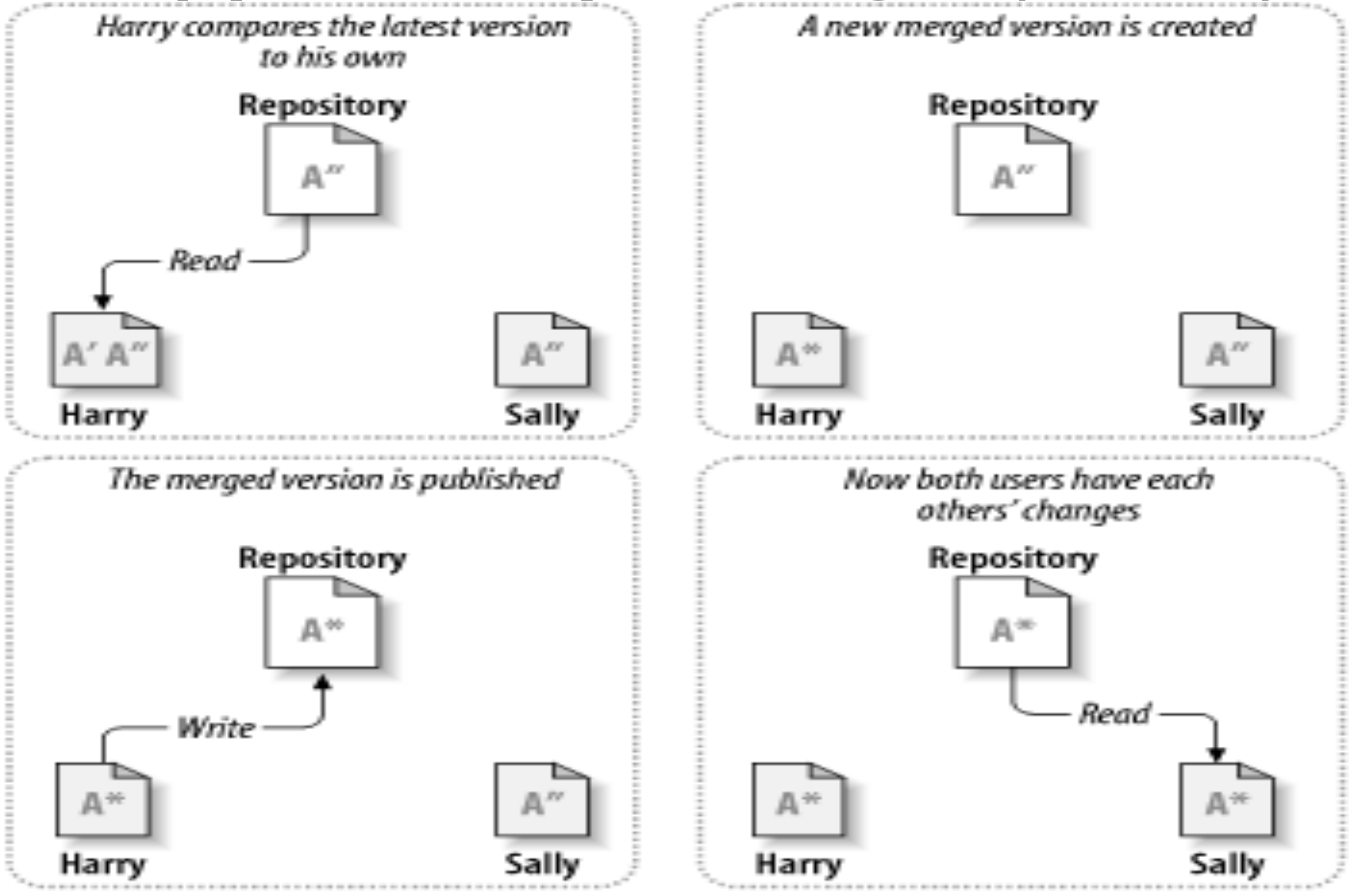
Solution: Lock-modify-unlock


Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A′
Harry
Sally
Harry writes his version, then releases his lock
Repository
A′
Write
UNLOCK
A′
Harry
Sally
Now Sally can lock, read, and edit the latest version
Repository
A′
Read
LOCK
A′
A′
Harry
Sally

Copy-modify-merge


Two users copy the same file
Repository
A
Read
Read
A
Harry
A
Sally
They both begin to edit their copies
Repository
A
A′
Harry
A″
Sally
Sally publishes her version first
Repository
A″
Write
A′
Harry
A″
Sally
Harry gets an "out-of-date" error
Repository
A″
Write
A′
Harry
A″
Sally

Copy-modify-merge (cont.)


Harry compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Harry
Sally
A new merged version is created
Repository
A″
A*
A″
Harry
Sally
The merged version is published
Repository
A*
Write
A*
A″
Harry
Sally
Now both users have each others' changes
Repository
A*
Read
A*
A*
Harry
Sally

Standard Cycle

- Checkout/update
- Edit
- Update (merge)
- Commit

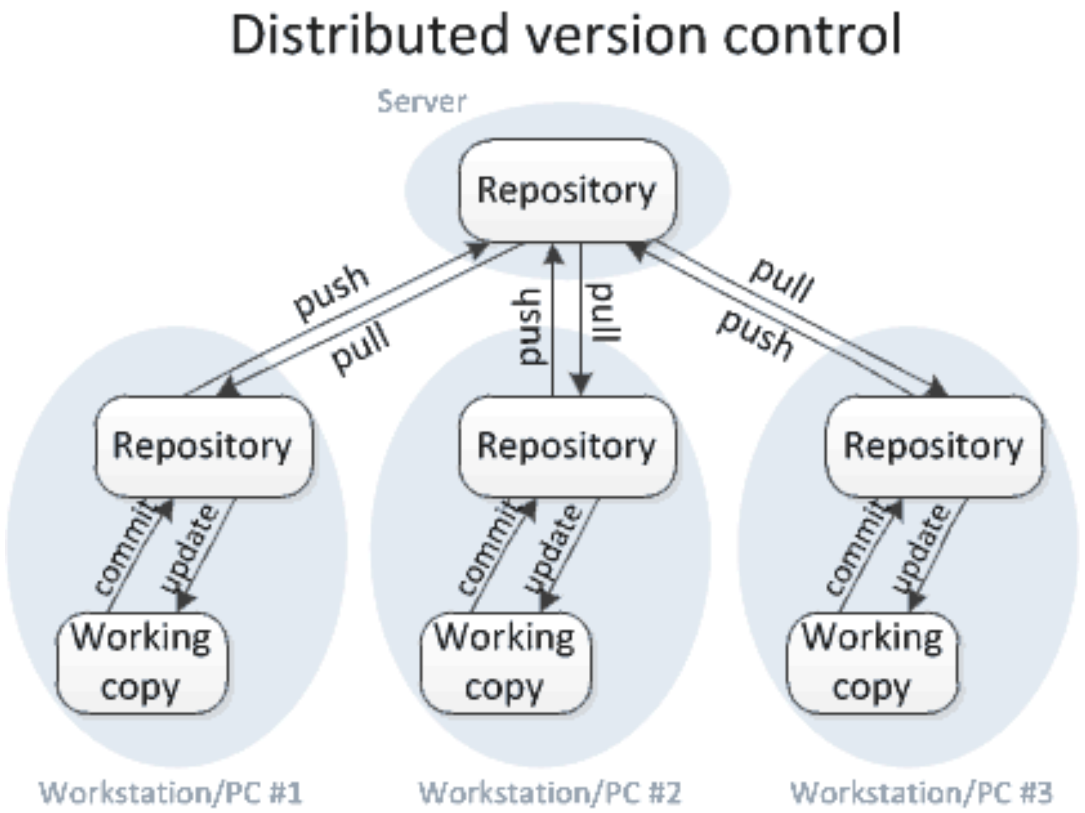
Standard Cycle - Ideal

- Checkout/update and run tests
- Edit and run tests
- Update (merge) and run tests
- Commit
    - Automatically run continuous integration tests

Centralized vs. Distributed (CVCS vs. DVCS)

- A centralized version control system
  - e.g. CVS, Subversion
  - A single, centralized, authoritative repository.
  - Devs make local "working copies".
  - When a dev is ready to share his/her changes, he/she sends the changes to the centralized repository.
  - Devs can also update their local copy to reflect the current state of the centralized repository.

Distributed version control (DVC)

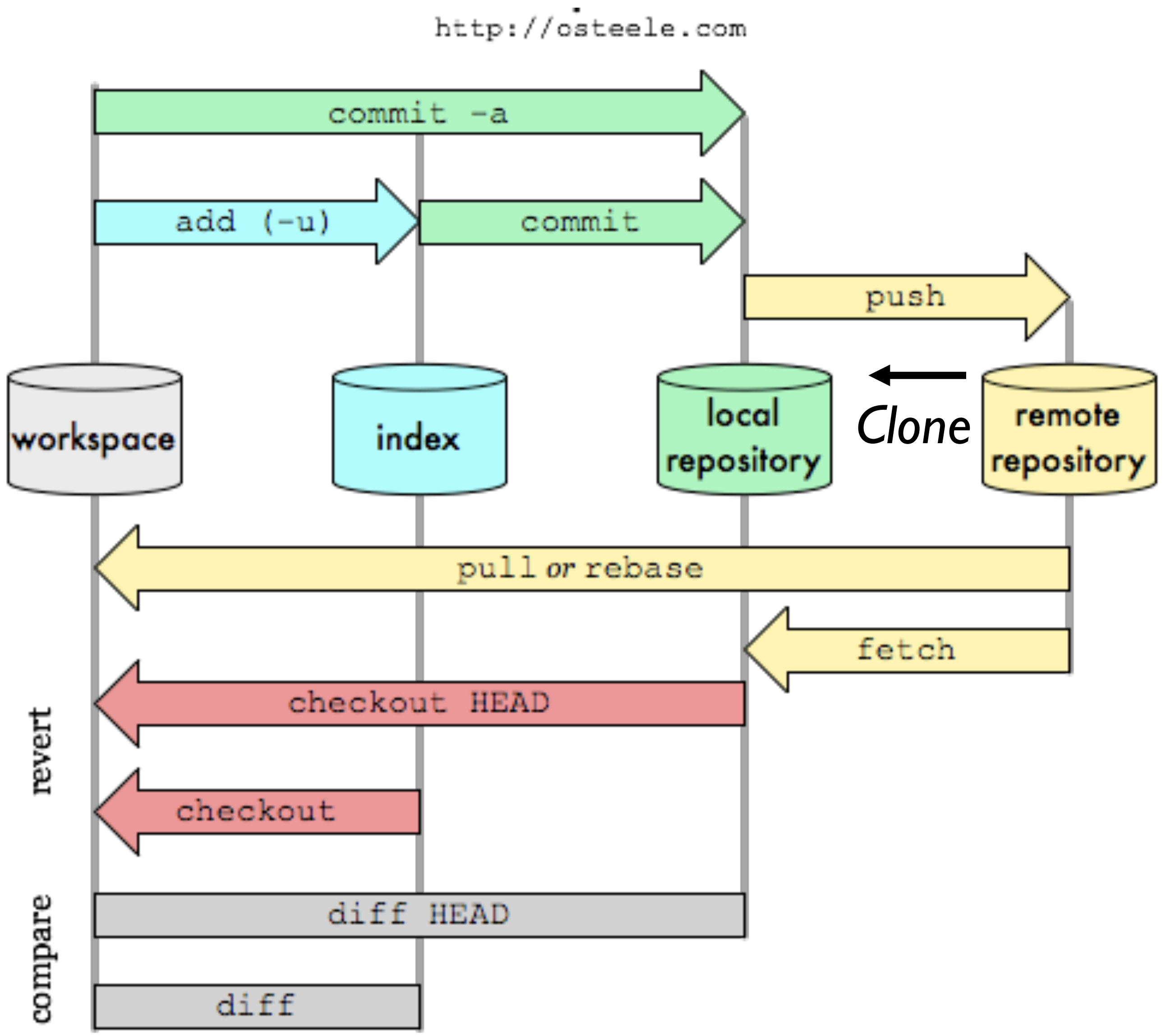


Source: http://homes.cs.washington.edu/~mernst/advice/version-control.html

CVCS vs. DVCS (Centralized vs. Distributed)

- Distributed version control system
  - e.g. Mercurial, Git
  - No single, authoritative repository.
  - Any repository can be cloned and devs can share their changes with each other.
  - Offline repository use supported.

DVCS Workflow (git)



DVCS (Git) Terminology

- clone - copy a repository locally
- ***pull*** - bring in changes from a **remote** repo and merge.
- merge - apply those changes to local repo/branch
- add - take the changes and stick them in index
- commit - commit index changes to the (local) repo
- status –s - show change status wrt remote
- stash -- don't add changes to the index, but save them for later (**ignore** local changes for now)
- ***push*** – send local changes to a **remote** repo

see: http://book.git-scm.com/index.html

CVCS vs. DVCS Tradeoffs (Centralized vs. Distributed)

- Commit often
  - Since changes don't affect anyone else until they are pushed, local code can be committed more often and thus better managed. Only before a push must it be free of errors that might break the rest of the project.
- History (of changes)
  - DVCS store the entire history with the repository; in CVCS many operations require access to the server because that's where the history is kept.

Merging is less problematic

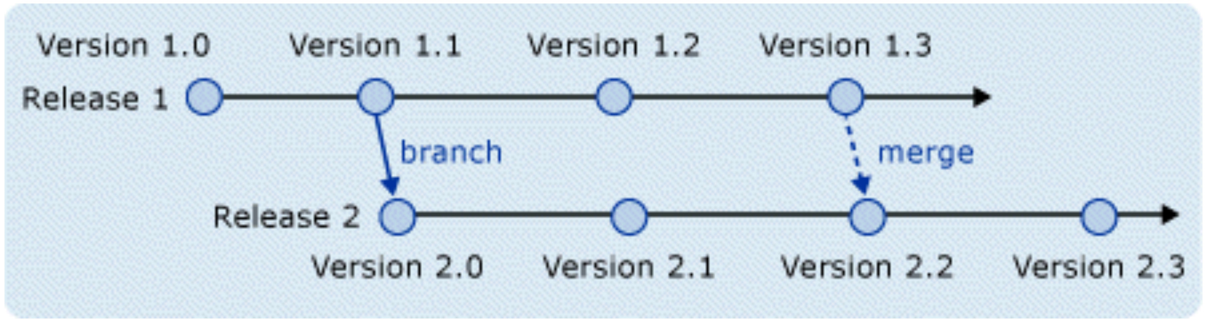
- Merging
  - Since DVCS **encourage frequent commits**, they track more information about changes than a CVCS. When it's time to merge, the system can automate most of the work, and often there are no conflicts for the developer to address.
  - CVCS often **compare only the latest versions** of a file and require the developer to work through (many) merge conflicts.

Branching made easy

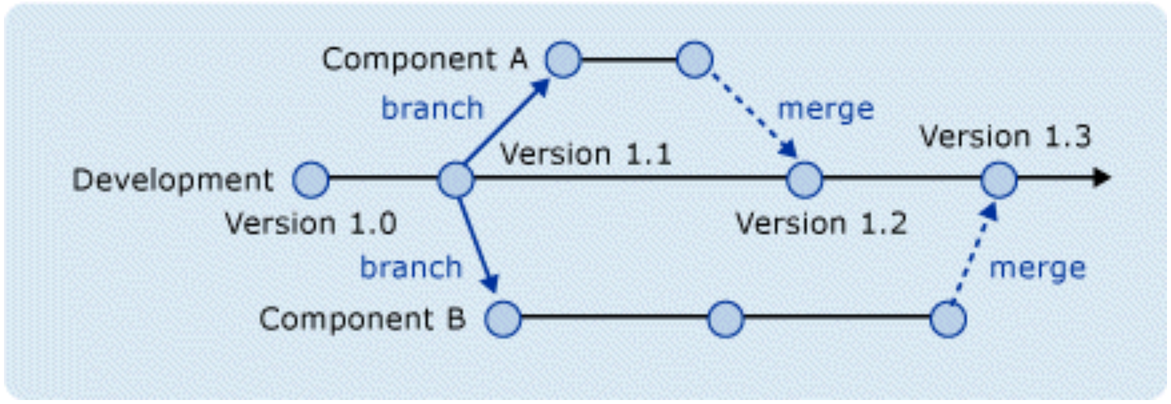
- Ease of experimentation
  - Since merging is much easier and less demanding on the developer than in a CVCS, one can **branch** and **experiment** easily in a DVCS.
- Central server
  - DVCS can use a central server to make it easier for developers to exchange files, but the server has no special status beyond acting as hub (example is GitHub).

Branching and Release

- Branch by Release



- Branch by Feature



- **Continuous** integration/delivery

Class exercise

- Readme.txt: (on github/example)

  A, B, C

- Suzie: **git pull**
- Joe: **git pull**
- Suzie: edits Readme.txt and changes it to

  A, B, C, D

- Suzie: **git commit –am “added D to readme”**
- Joe: **git pull**

- **What does Joe’s version of Readme.txt contain?**

Class exercise

- Readme.txt: (on github/example)

  A, B, C

- Suzie: **git pull** (from github/example)
- Joe: **git pull**
- Suzie: edits Readme.txt and changes it to

  A, B, C, D

- Suzie: **git commit –am “added D to readme”**
- Joe: **git pull**

- **What does Joe’s version of Readme.txt contain?**

  A, B, C

- Readme.txt: (on github/example)
  - A, B, C
- Suzie: git pull (from github/example)
- Joe: git pull
- Suzie: edits Readme.txt and changes it to
  - A, B, C, D
- Suzie: git commit –am “added D to readme”
- **Suzie: git push origin master**
- Joe: git pull

- **What does Joe’s version of Readme.txt contain?**
  - **Answer: A, B, C, D**

Branches

- Git:
    - origin master   (github/example)
        - bug-24
    - master (local machine)
        - **new-cool-ui**
        - bug-24

Git Branching

- git checkout –b test-ui
    - Now we are on a new branch
- git push –u origin test-ui
    - Our local branch is pushed to GitHub repos as a branch

Branching

- Merging branches can be a big check-in
  - Usually generate many conflicts (in CVCS)
  - Referred to as “integration” for hierarchical branches
  - Remember check-in is:
    - CVCS: Check-out → Solve Conflict → Check-in
    - DVCS: Pull from main → Solve Conflict → Push to main

Conflicts and merging

- Merging means resolving conflicts between two branches.
- Conflicts: same line of code is changed by different devs.
- Diff tool compares (lines of) files. (git mergetool)
- Conflicts when the same physical line is different.
- VCS will leave markers in file to indicate conflicts to be resolved.

- Readme.txt: (on github/example)

  This is a readme file

- Mark: **git pull** (from github/example)
- Joe: **git pull**
- Mark: **changes Readme.txt and commits:**

  This is a readme file for git.

- Mark: **git push origin master**
- Joe: **changes Readme.txt and commits**

  This is a readme file

  It includes an example too.

- Joe**: git pull**

What does Joe's version of Readme.txt contain?

Changes merged automagically

What does Joe's version of Readme.txt contain?

This is a readme file for git

It includes an example too.

Controlling Change

- Different branches/versions have different impact
- Example
  - Module branch affects 4-7 persons
  - Subsystem branch affects 30 persons
  - System branch affects 200 persons
  - Released beta version affects 2000 users
  - Released version affects 10000 customers
- Policies describe what is required so that code can be promoted from one level to the next

Maturity Level

- The stricter the policy, the more mature the code
- Examples
  - Only one person depends on it
    - Code must compile
  - Teams depend on it
    - Must pass unit tests, must have good comments
  - Many projects depend on it
    - How-to-use documentation, system tests
  - Released to the outside world
    - Acceptance tests, rigorous testing, extensive documentation, extensive comments, incubation period, etc.

Assignment 2 released

- See Connect -> Assignments

Best Practices (1)

- Update/pull every morning
- Commit/push early and often
- Commit logical units
- Don't break the build (unit tests must run)

Best Practices (2)

- Always run the tests before committing
- Provide clear comments in your commit messages
  - Good: “fix issue #23 on concurrency”
  - Bad: “more updates”
  - Ugly: “ ”
- Communicate with team members to avoid conflicts
  - JIRA/Bugzilla/GitHub Issues

What to store in VC?

- Source code:
  - Java, C, Python, Ruby
  - HTML, CSS, JavaScript
- Images
- Configuration files
- Unit and other automated tests
- Everything required for your app to run

What to store (cont.)?

- Requirements documents
- Documentation – user manual
- UML design documents
- Most documents/files related to your project

What NOT to store?

- Compiled code: .class
- Generated code: .project, .class, target/
- Generated documentation: JavaDoc API
- Secured information:
  - Id/password files
  - License keys

.gitignore

```
# Emacs
*~

# maven
target/

# Eclipse
.classpath
.project
.settings/
```

Never commit to “master”

When using DVCS (aka Git):

- Always make a **branch** for the new feature or bug fix you want to work on
- Once done, send a **pull request (GitHub)**
  - To be merged into “master”
- This allows **code reviews** by peers
  - Can detect inconsistencies in your code
  - Feedback on code quality, proper use of APIs/libraries, etc

How *not* to use Version Control

- Check-out once, Work, Check-in on due date
- Add tons of dependencies, don't tell anybody, don't check-in until late
- Radically restructure a crucial file, check-in
- Routinely overwrite another group's changes without discussing it with them

Lab Work

- Collaboration, collaboration, collaboration!
- Branching, Committing, Merging
  - A: makes a branch, sends a pull request
  - B: Reviews, accepts the pull request, merges into Master
- Social coding is important!
- Not the only measure of participation
  - There will be peer-evaluation as well.

Questions?