Version Control with Git

CSE 3311
Christoph Csallner
University of Texas at Arlington (UTA)

These slides contain material from the sources listed on the following slides titled “Sources”

Sources

- [CD] “What’s Wrong with Git? A Conceptual Design Analysis”
  - By Santiago Perez De Rosso and Daniel Jackson
  - http://dl.acm.org/citation.cfm?id=2509584
- [Mike] “Version control concepts and best practices”
  - By Mike Ernst
  - http://homes.cs.washington.edu/~mernst/advice/version-control.html#dvc-advice

Why Version Control?

- Manage multiple versions of a piece of information
- Many non-developers do this manually
  - Every time you modify a file: Save file under a new name
  - For example: Foo-1.txt → Foo-2.txt → Foo-3.txt

  [+] Easy initially

  [+] Do not need (to learn) any tools besides OS

  [-] Waste of disk space (redundant copies of unchanged parts)

  [-] Cumbersome if file name expected to remain constant (e.g., Java source files)

  [-] Sharing files across machines cumbersome (email?)

  [-] No support for team work (parallel edits & merge?)

Version Control via Cloud Storage

- Cloud storage mostly good for non-developers
- Every time you modify a file: Save (same name)
  - For example: Foo.txt → Foo.txt → Foo.txt
- Storage client stores new version in cloud
  - Some providers store last N versions of each file

  [+] Some clients allow reverting to older versions

  [+] Relatively simple clients + web-based administration, e.g., Microsoft OneDrive, Google Drive, DropBox, Box

  [+] Little disk space wasted locally (but maybe in cloud)

  [+] Sharing files across machines easy (auto-sync)

  [+] Some support for teams (parallel changes to same doc)

Is Cloud Storage Sufficient?

- Cloud Storage seems to do a lot of good things

- But limited support for things developers also do
  - Name and annotate each change
  - Refer to change from other tools (e.g., issue tracking)
  - Many parallel branches of same files
  - Fork and merge branches
  - Merge conflicting changes in same branch

Version Control

- Many alternative names
  - Revision control
  - Configuration management
  - Software configuration management
  - Source code management
  - Source code control
  - Source control

- Essentially all refer to very similar concepts
  - Other sources may provide contradicting definitions

Concrete Example Tools

- Following are a few representative example tools
  - There are many others
    http://en.wikipedia.org/wiki/Revision_control

- Proprietary
  - 1970s: client-server
  - 1990s: Bitkeeper (distributed)

- Open-source
  - 1990: CVS (client-server)
  - 2000: Subversion (client-server)
  - 2005: Git, Mercurial=Hg (distributed)

Version Control Benefits

- New version does not require new file name
- Keeps track of project history: For every change
  - What changed, who changed it when
  - Why it was made (if change is annotated)
  - Change can be referenced from external tools

- Helps recover from mistakes: Revert to old version
- Supports collaboration
  - Easy sharing files across machines
  - Create and merge many branches of same files
  - Identify and merge conflicting changes in same branch

Benefits Grow with Team Size

- Assume 500 software developers working on the same project ..
- .. without any kind of version control tools

→ The project is doomed

- Given the growing benefits, larger teams are more willing to use and invest in version control tools
- Even if these tools
  - Are hard to learn
  - Have known problems (scalability, etc.)

But Also Scales Down to Single User

- Very useful even for lone user

- Cloud Storage
  - Syncs files across different machines of single person
  - Creates backups of files (one per machine + cloud)
  - Allows reverting to old versions

- Version Control
  - Similar benefits
  - Can work on multiple branches of same file
  - Easy to compare and merge different versions

Three Tool Generations

1. Single computer
2. Client-server (e.g.: Subversion)
   - Typically server is only node storing full version history

   [+] Several people can work on same project

   [-] Single-server bottleneck

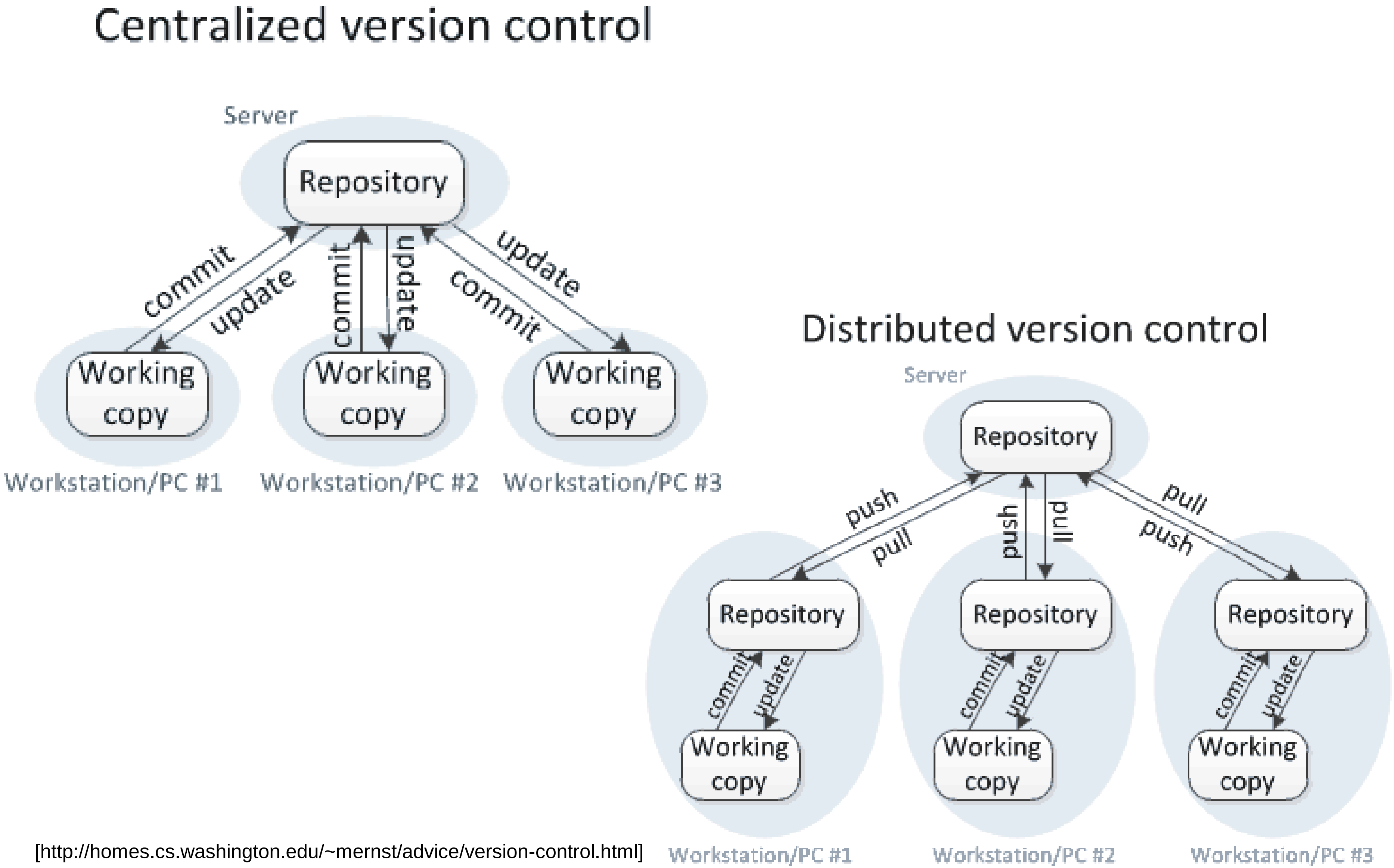
3. Peer-to-peer (“distributed”) (e.g.: Git, Mercurial)
   - Each peer has its own client and its own server with full version history → Enables many interaction styles

   [+] No single bottleneck if used peer-to-peer style

   [+] Can simulate client-server style (GitHub, BitBucket)

   [-] May use a lot of space on each peer (i.e., for binaries)

Comparison in Centralized Scenario



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

Following slides are based on [CD]

(except for domain model refresher slide)

GIT

In Git: “File” means “Path of the file”

- “File”
  - Git’s heuristic of the identity of an actual file in your OS’s file system (≠ file contents)
  - In Git: “File” means **path of the file**
  - “File path” would have been a less confusing name
  - Git does not keep track of the identity of an actual file independently of the file path

- Having “file” a/b/c in Git does not mean you have a/b/c in your machine’s file system
  - For example, in Git: “File” a/b/c may be marked as “staged for removal”

Rename / move an actual file → Git ?

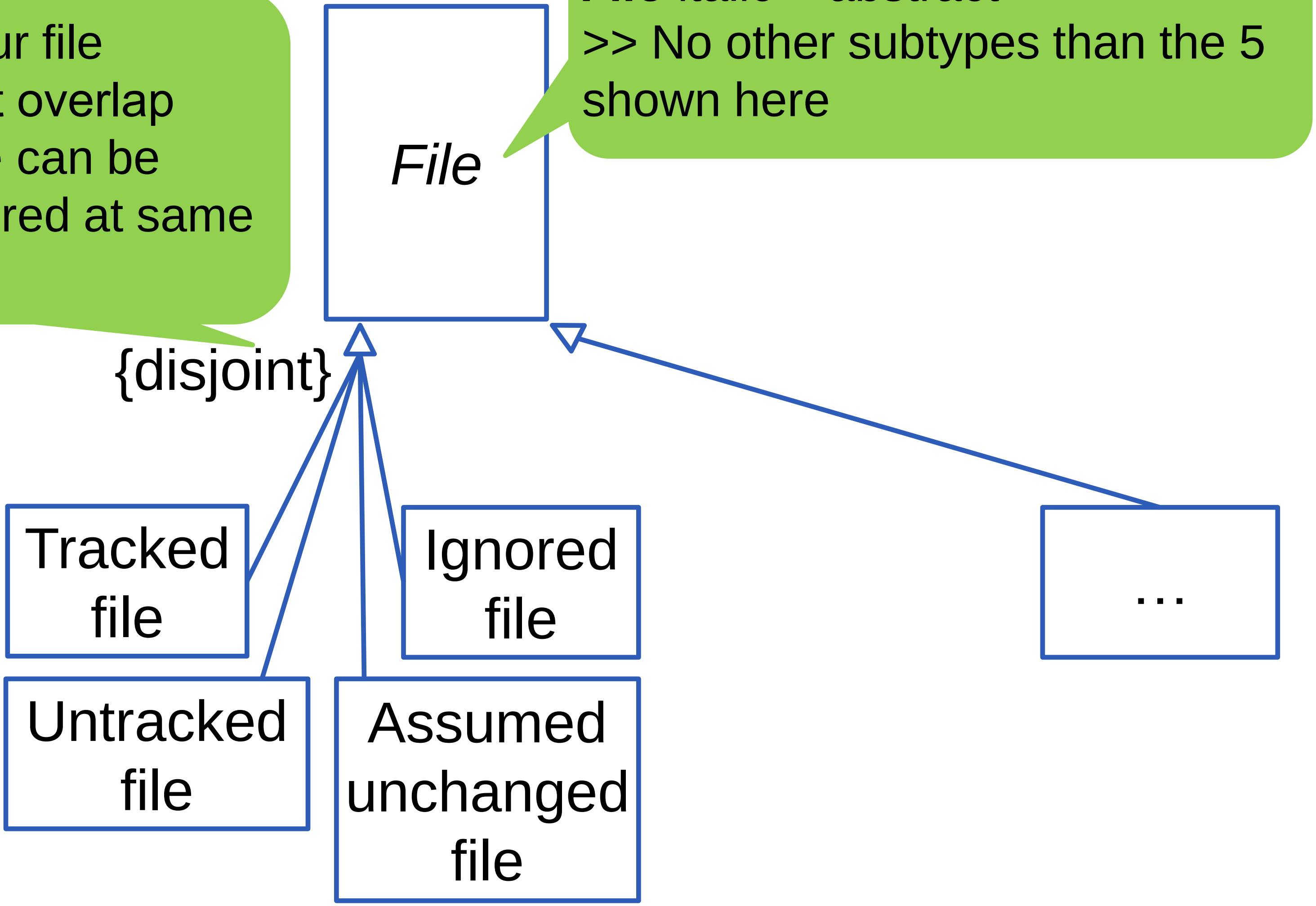
- Aside: Unlike Git, OS keeps track of actual file’s identity separately from actual file’s name and path
  - Rename actual file: OS only updates name
  - Move actual file: OS only updates path

- Breaks Git’s heuristic of the actual file’s identity
  - In Git: Delete a/b/c then add a/b/d (or add a/e/c)
  - “Git has a rename command git mv, but that is just for convenience. **The effect is indistinguishable from removing the file and adding another with different name and the same content**.”
    - [https://git.wiki.kernel.org/index.php/GitFaq#Why_does_Git_not_.22track.22_renames.3F]
    - Breaks commit history of actual file

Conceptual model: Core file subtypes

{disjoint} = Four file subtypes don’t overlap
>> E.g., no file can be tracked & ignored at same time

File italic = abstract
>> No other subtypes than the 5 shown here



[CD, Figure 1] (excerpt)

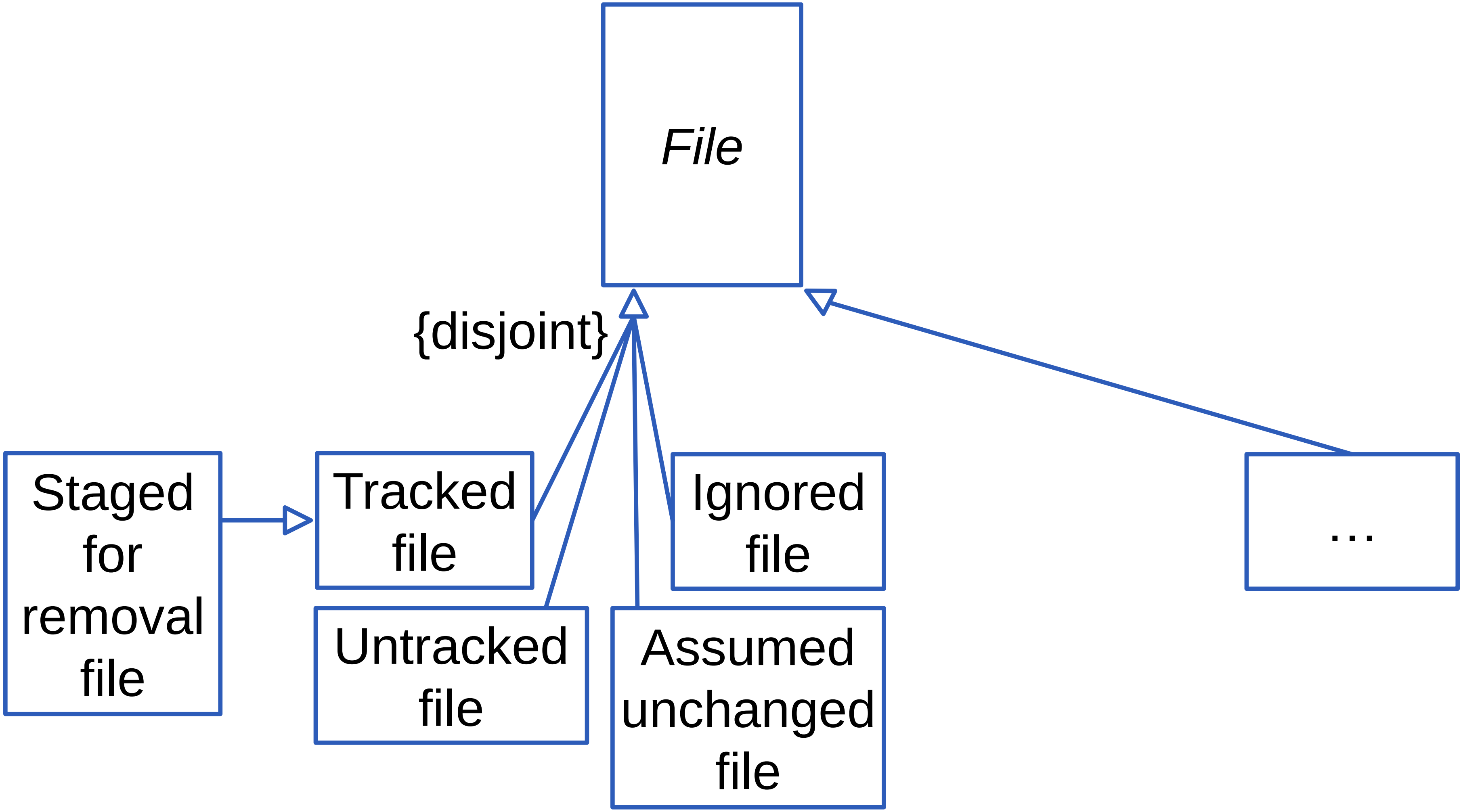
(Un)Tracked → Assumed unchanged

- Tracked file
  - "File" whose modifications Git will notice

- Untracked file
  - "File" in the working directory & has no committed version
  - Adding such a file to Git → Git will start tracking it

- Assumed unchanged file
  - "File" that was previously tracked
  - User has indicated Git should no longer track it
  - Adding such a file to Git → No effect
    - Instead: Unset file's "assume unchanged" bit

Conceptual model: Staged for removal



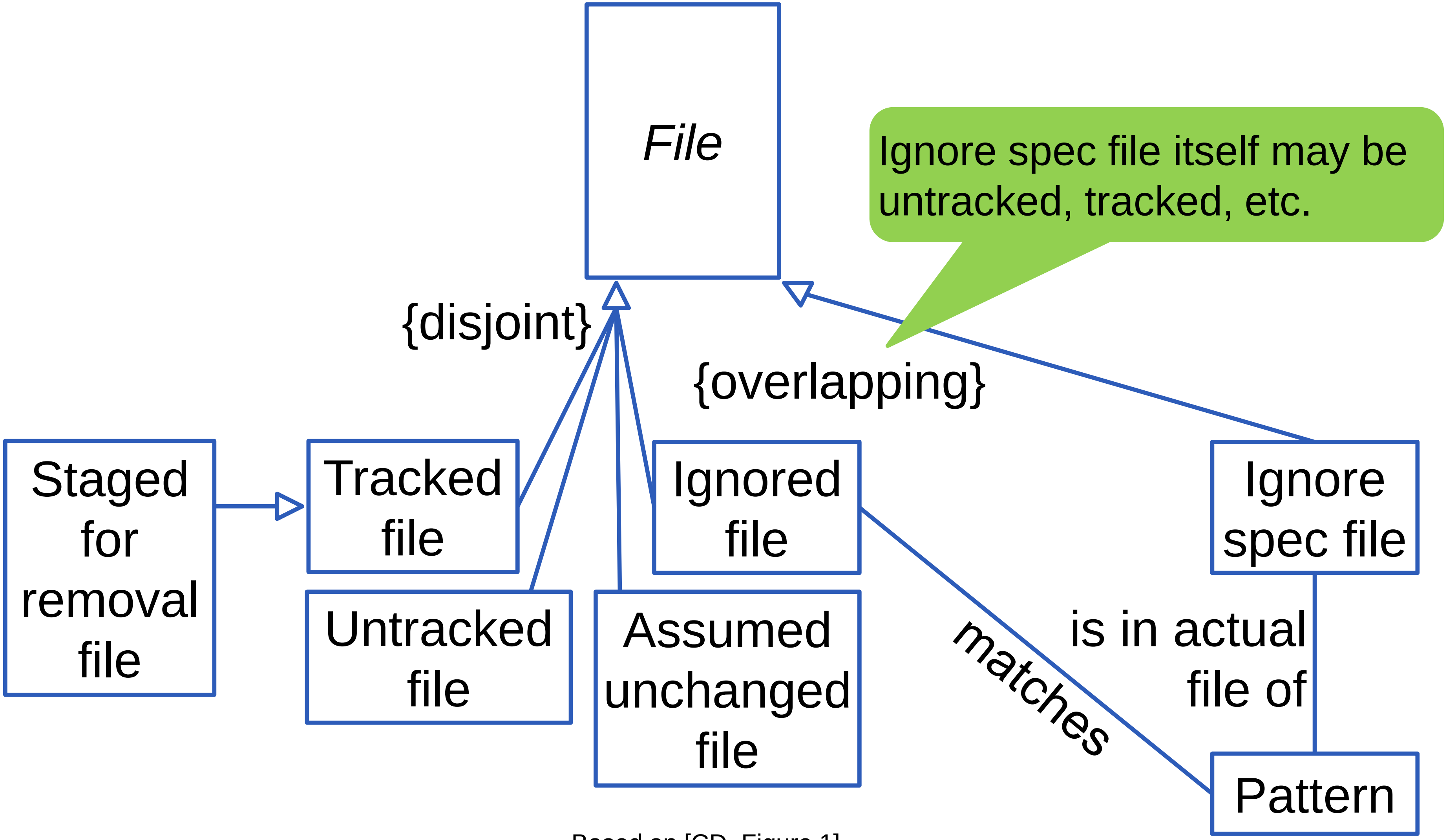
Based on [CD, Figure 1]

Staged for removal

- Staged for removal
  - File path of an actual file that no longer exists in working directory
  - Staging area records file’s absence
  - Will be removed from the repo on next commit

- Staged for removal vs. un-staging
  - Un-staged := Staged version is removed

Conceptual model: `.gitignore`



Ignore spec file itself may be untracked, tracked, etc.

Based on [CD, Figure 1]

Pattern

- String specifying a set of file paths
- Used in actual `.gitignore` files
- Example: `*.class` for Java project

Ignored

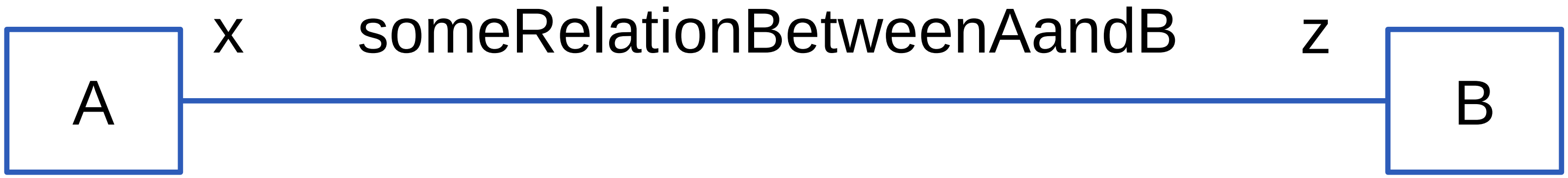
- Ignore spec file
  - File path of special kind of actual file (`.gitignore`)
  - Actual `.gitignore` file contains set of patterns of file paths Git should ignore
  - May be tracked, untracked, etc.

- Ignored file
  - File path ignored by Git
    - E.g.: Will not appear in output of “git status” command
  - Git will ignore “file” if it matches pattern in `.gitignore` in file’s directory or any of its (recursive) parent directories

Recall: Relation multiplicity constraint

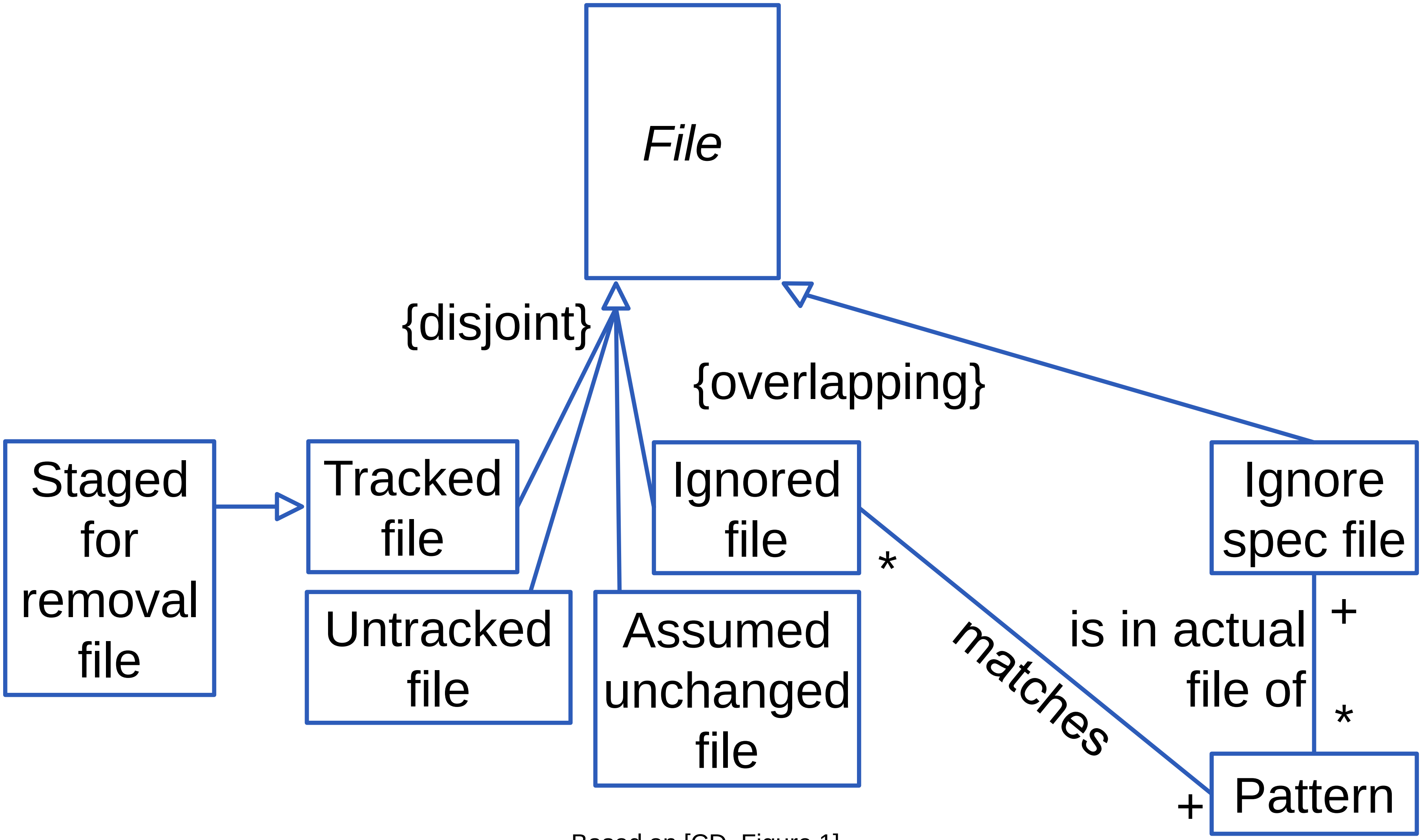
- $A = \{a_1, a_2, \ldots\}$
- $B = \{b_1, b_2, \ldots\}$
- someRelationBetweenAandB = $\{(a_2, b_1), \ldots\}$

Depending on the relation:
>> Some a_g, b_h may not appear in relation
>> Some a_i, b_j may appear several times



- Each item in A maps to z items in B
- Each item in B maps to x items in A
- Each annotation x, z may be a range or:
  - Star (*) = “≥0”, plus (+) = “≥1”
  - Question mark (?) = “0 or 1”

File (path) → Working, etc. version

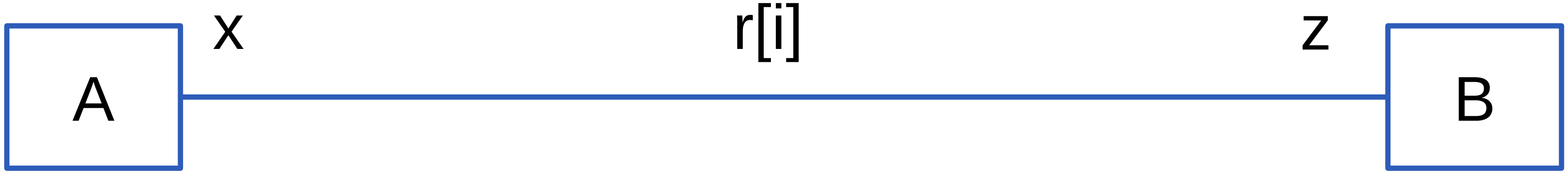


Based on [CD, Figure 1]

Extra notation: Set of similar relations

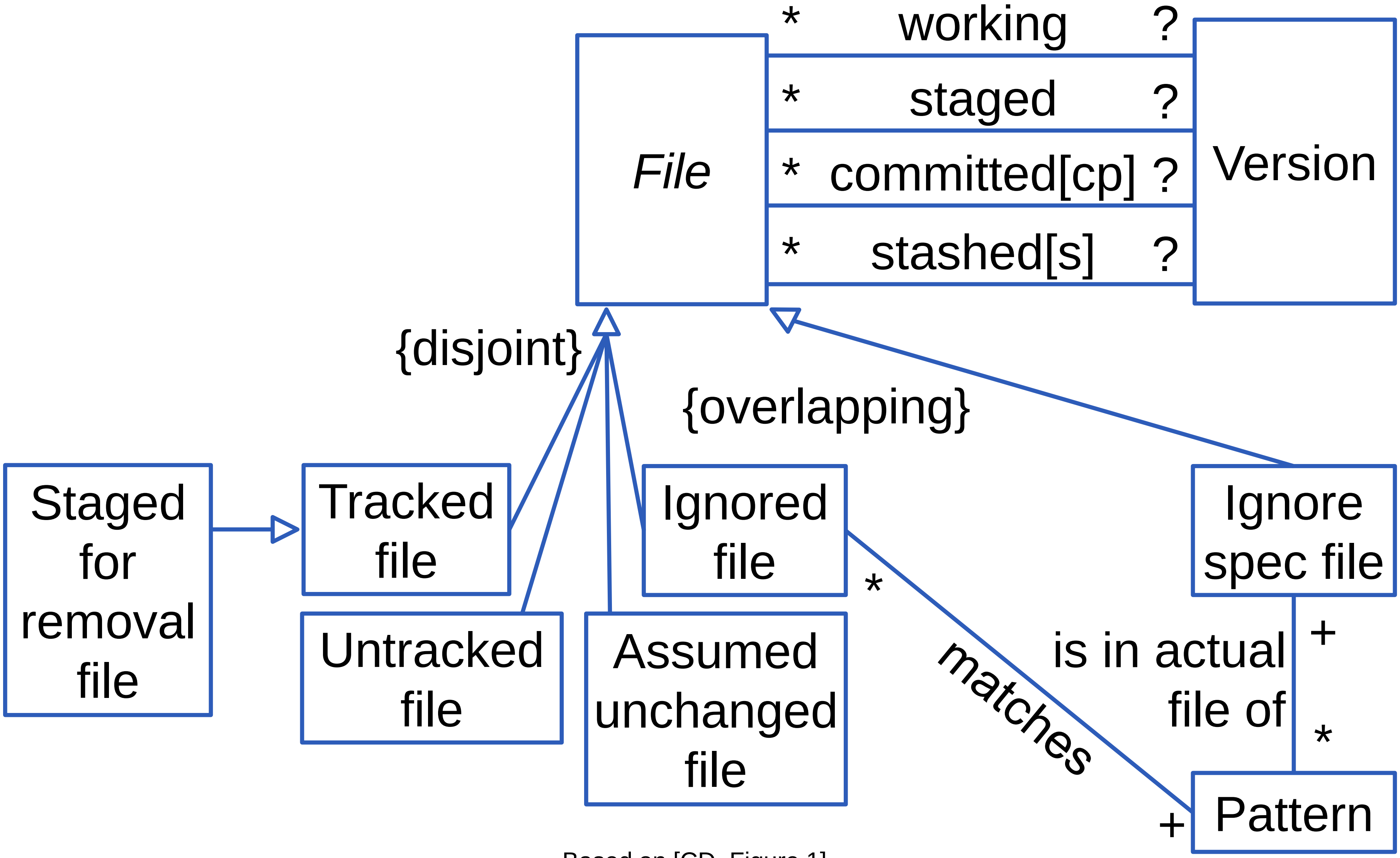
- $A = \{a_1, a_2, \ldots\}$
- $B = \{b_1, b_2, \ldots\}$
- $r[i] = \{(a_2, b_1), \ldots\}$

>> “r” is a mapping from some “index” set to a set of relations
>> “i” is an element of that “index” set
>> Each “r[i]” is a relation between A and B



- In each r[i]: Each item in A maps to z items in B
- In each r[i]: Each item in B maps to x items in A
- Each annotation x, z may be a range or:
  - Star (*) = “≥0”, plus (+) = “≥1”
  - Question mark (?) = “0 or 1”

File (path) → Working, etc. version



Based on [CD, Figure 1]

Version, Branch, Stash

- Version := Contents of a file at some point in time
  - Two different files with same content → same version
- Version ≠ “version number”
  - Version number would be a name for a version
  - Git doesn’t have concept of a version number

- Branch
  - Named collection of committed versions of files
  - Identifier, user can change name

Working version & Staged version

- Working
  - Working version of a file := stored in working directory
  - A file can have at most one working version
  - A file may be committed to repo but no longer be in working directory
- Staged
  - Staged version of a file := stored in staging area & saved to repo on commit
  - File can have at most one staged version
  - Untracked file cannot have a staged version
  - File can have a staged but no working version
    - If user deletes file from working directory after adding it to Git

Committed version & Stashed version

- Commit point (cp)
- Committed version
  - Committed version of a file := Stored in local repo at commit point cp

- Stash s := Collection of file versions saved together
  - Git maintains stack of all stashes
  - Stash name `stash{i}` assigned by Git (i = stack index)

- Stashed version
  - Stashed version of a file := Stored in stash s

(Implicit) git add → git commit

- `git add f`
  - Makes f’s working version also f’s staged version
  - If f is untracked → switch f to tracked
  - **Subsequent changes to actual file f will cause f’s working version to diverge from f’s staged version**
    - After changing contents of actual file f just run again: `git add f`
- `git commit`
  - Makes all **staged versions** also committed versions
- `git commit –a`
  - `git commit f1 f2`
  - Make **working versions** also committed versions
    - For all files or just for files f1, f2
    - Skips staging area (implicit git add)

Reverting

- `git reset HEAD f`
  - Disassociates f from staged version
  - If f was untracked before git add that staged it → switch f back to untracked

- `git checkout f`
  - Replace f’s working version with f’s staged version
    - No staged version → Replace working version with f’s committed version

Removing

- `rm f`
  - Remove f’s working version (standard OS command)
  - Doesn’t affect staged or committed versions

- `git rm f`
  - If f is tracked → stage f for removal, remove f from working directory
  - If f is assumed unchanged → same as above, but also f becomes tracked again

Following slides mostly based on Mike Ernst’s best practices:

http://homes.cs.washington.edu/~mernst/advice/version-control.html#best-practices

BEST PRACTICES

Write a descriptive commit message

- Briefly explain purpose of this commit
  - In present tense
  - Refer to a specific issue this commit addresses
    - Include (link to) issue number in issue tracking system

- [-] Initially takes a bit of effort to write
  - But after some practice only takes a few seconds

- [+] Valuable later on
  - When searching in the history of commits

Each commit should refer to an issue

- First create an issue in your issue tracking system

- Now each commit should refer to an issue
- Easy in BitBucket, e.g.:
  - “#13: Fix license file”
  - In web view, BitBucket will link “#13” to issue 13

- [+] Makes it easier to
  - Understand individual commit
  - Find all commits for one issue
    - Since developers may work on multiple issues at the same time

Make each commit a logical unit

- Each commit should have a single purpose
- A commit should not serve two or more purposes
  - Makes it hard to undo one of these purposes
- When changing many files in working directory that address multiple purposes
  - Package these changes into separate commits
  - Do not have to commit all changes in single commit

- Each purpose should be contained in a single commit
- Do not spread a purpose over multiple commits
  - Makes it harder to undo a purpose (multiple commits)

Git workflow for creating a commit

- `git status`
  - Lists all the modified files

- `git diff`
  - Shows specific differences
  - Helps you compose a commit message

- `git commit f1 f2 -m “My commit message“`
  - Commits just the files f1, f2 you want to commit for this particular commit

Incorporate others’ changes frequently

- Fetch others’ commits daily, via one of:
  - `git pull`
  - `git pull -r`

- Reduces risk of merge conflicts

Frequently share your commits

- `git push`

- After completing each small task
- But avoid breaking the overall system
  - Check, verify, test before sharing your commits

Coordinate conflicting edits

- Ideally do not work on same file as your co-worker at the same time
  - Avoid later merge conflicts

- Person B should only start editing file F **after** person A committed and shared changes to F

Remember tools are line-based

- Tools look for changed lines and let users merge conflicting lines

- Keep each line short, ideally to about 74 characters
  - Fits on screen
  - Longer lines are harder to merge, have to scroll

- Changes are harder to spot in longer lines
  - Tool marks the entire line as changed
  - Often does not point to location in line

Do not commit generated files

- Do not add or commit, e.g.:
  - .class generated from your .java files
  - Exclude such files explicitly or via pattern

- User can re-generate such files from the sources
- Unnecessarily increases size of repository
- May trigger merge conflicts
  - Wastes users' time dealing with such conflicts

- But do add important binaries that rarely change
  - .jar of libraries needed to run code

Do not force things

- Using the -f force option can get you into bigger trouble