Software Engineering

Software Configuration Management

何明昕 HE Mingxin, Max

Send your email to c.max@yeah.net with
a subject like: SE-*id-Andy: On What …*

Download from c.program@yeah.net
/文件中心/网盘/SoftwareEngineering2024

Topics

- Definitions
- Working Scenarios
- Revision Control Functions
  - Git and GitHub

Why Software Config Mgmt?

Software configuration management (SCM) is the process of controlling the evolution of a software system

- Simplifies sharing code and other documents
- The ability to revert to an older version ("undo")
- Coherently integrate contributions from team members ("merge")
- Notify interested parties about new modifications ("reporting")
- Track software issues (e.g., bugs)
- Create an auditing trail ("archiving")
- SCM system allows us to answer questions:
  - When was this software module last changed?
  - Who made changes in this line of code?
  - What is the difference between the current version and last week's?
  - How many lines of code did we change in this release?
  - Which files are changed most frequently?

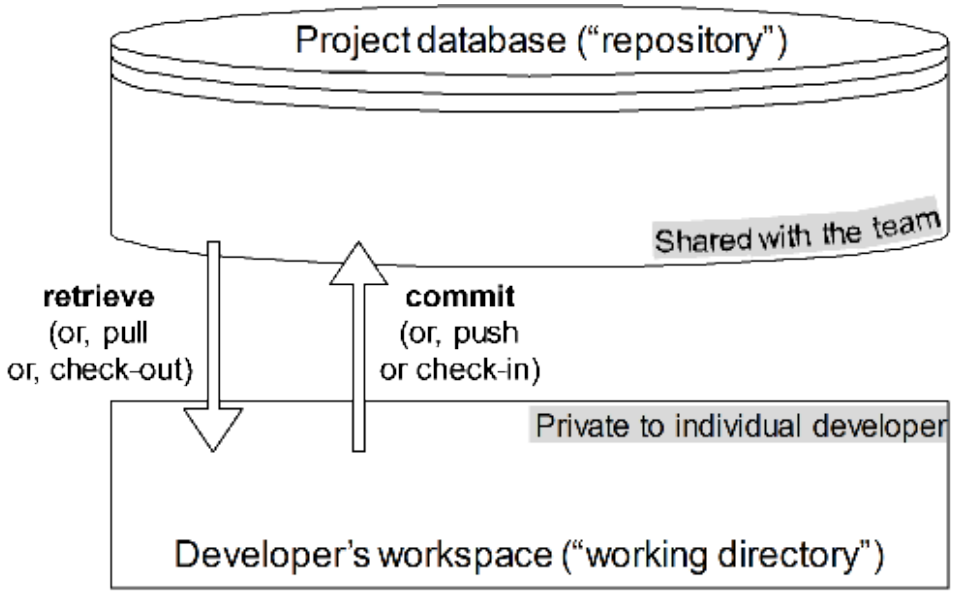
Why Software Config Mgmt?

- SCM simplifies collaboration and increases productivity
- But it all can be done with old-fashioned email, file system, etc.
  - Naming convention?
  - How to reconcile your version #1 with my version #1?

Definitions

- A **software configuration** is a collection of work products ("artifacts") created during a project lifetime
- **Software configuration item** (**SCI**) is any piece of knowledge (a.k.a. "work product" or "artifact") created by a person (developer or customer)
- A snapshot of a software configuration or a configuration item represents the **version** of that item at a specific time
- A **commit** (or "check-in") refers to submitting a software configuration to the project database ("repository")
- A **build** is a version of a program code (part of a software configuration), and a process by which program code is converted into a stand-alone form that can be run on a computer ("executable code")

Repository vs. Workspace

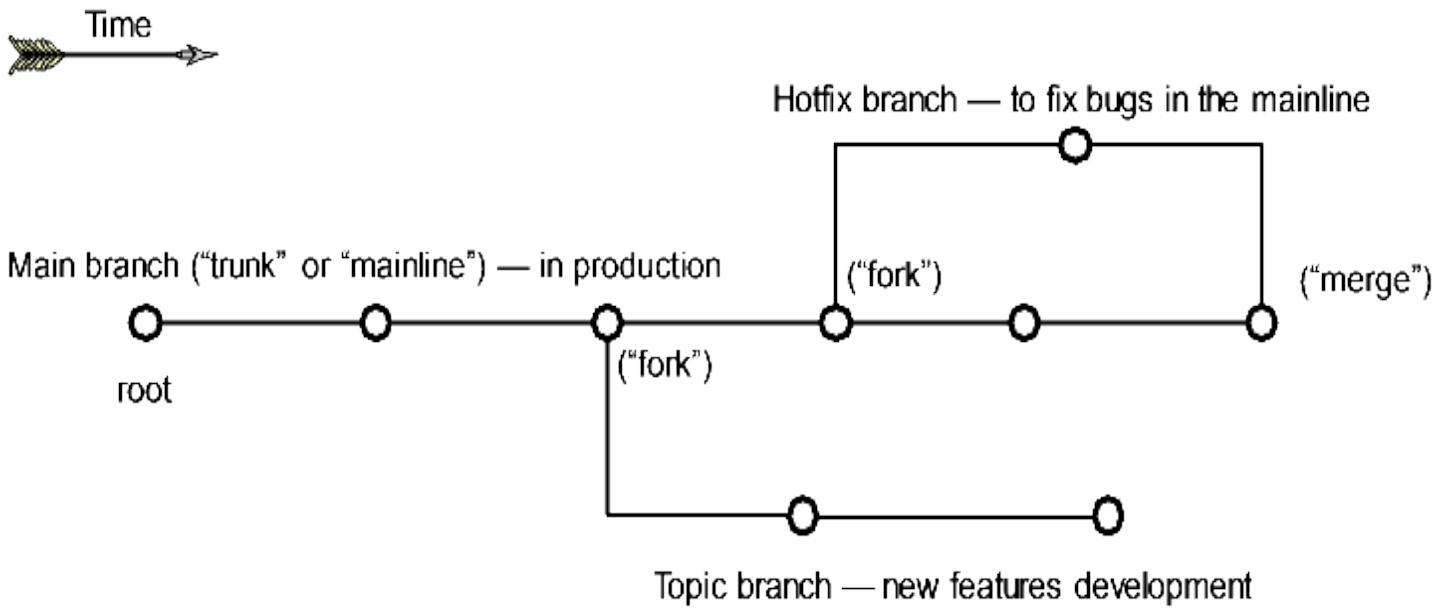


Committed configurations
(at discrete times during project life)

Working copies
(piecewise continuous development)

- Project database is usually on a remote server, or "mirrored" on several servers
  - This is the "*team memory*," unlike individual workspaces that are private
- SCM provides a set of tools to interact with project database

Version Graph and Branching

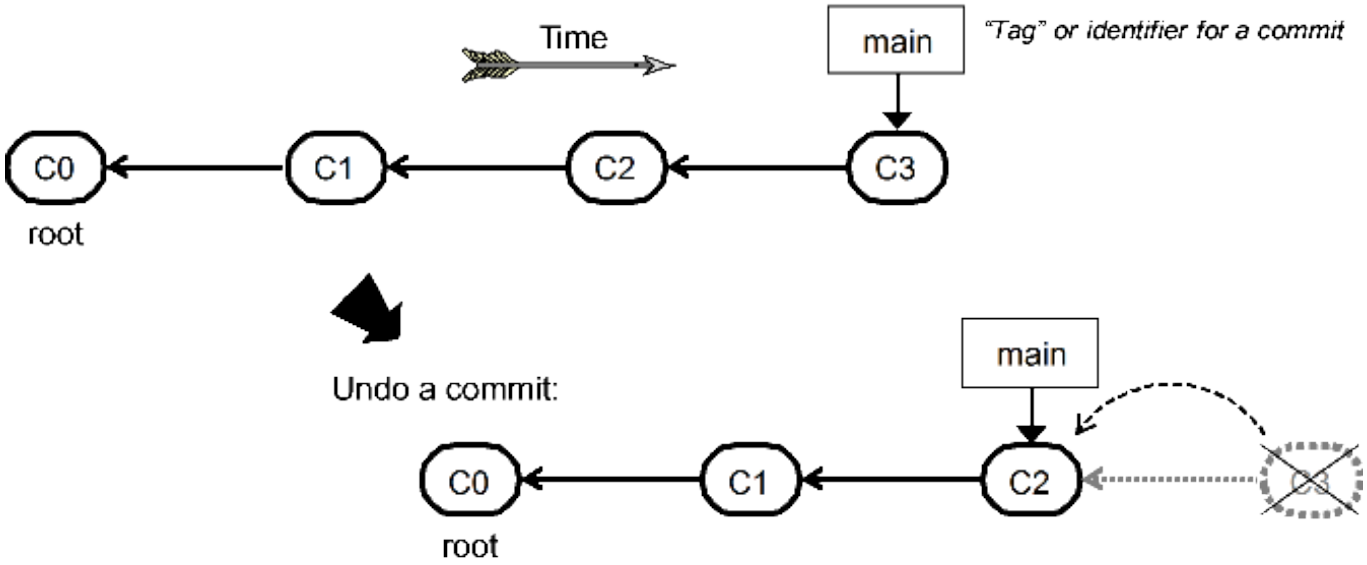


- Each graph node (a "commit") represents a "version" of the software configuration shared with the team
  - The private workspace of one team member was publicized in the shared repository and becomes a part of "team memory"
- Think of *branches* as separate folders, each with its own content and history
- The project snapshot at the tip of a branch represents the *latest version*

Example Working Scenarios

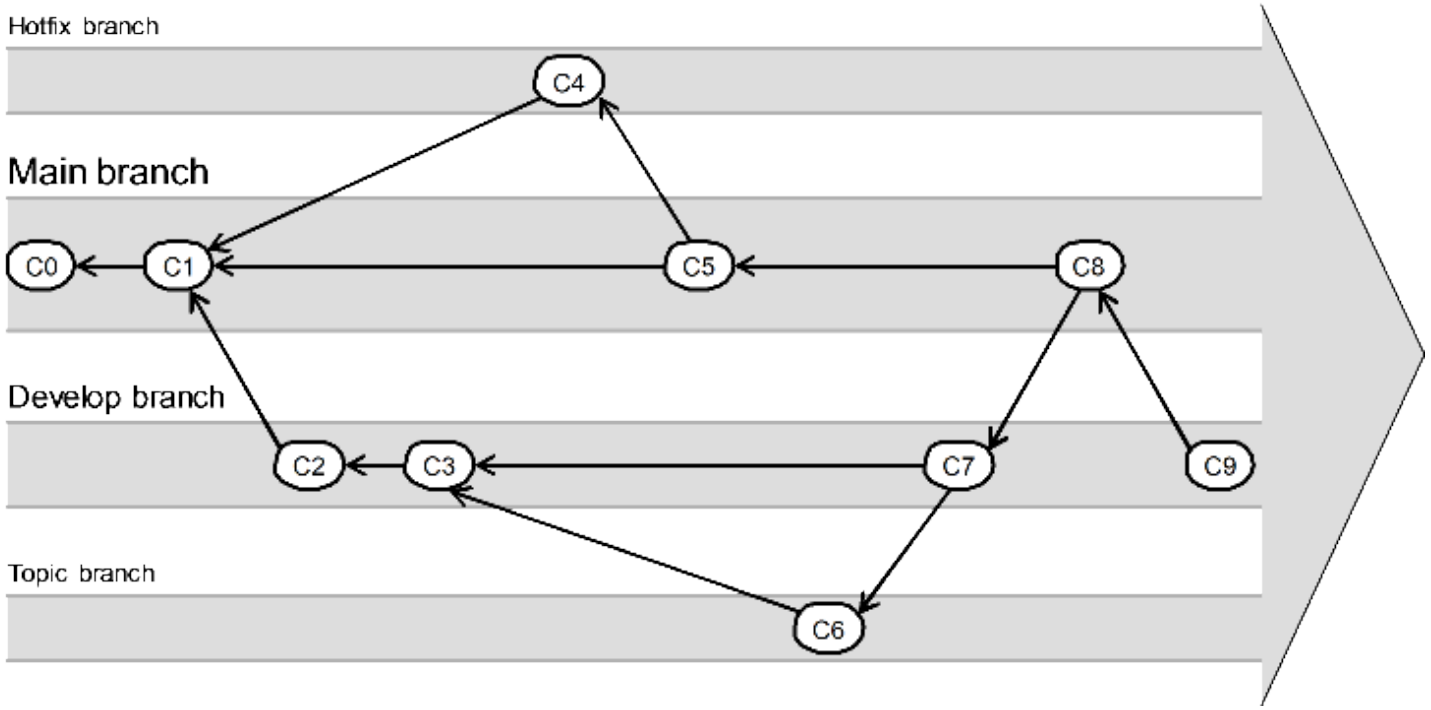
- Undoing mistakes in new work
- Supporting multi-pronged product evolution
  - Parallel versions coexist at times, but will eventually merge (single trunk—"Main")
- Developing product lines
  - Coexisting parallel versions (same product for different markets) that will never merge (multiple trunks)
    - but they still need synchronization so that versions for different markets are at the same "level"
- Working with a small team of peers
  - All developers can write to the project database
- Working with a managed team
  - Only configuration manager can write to the project database

Undoing Commits



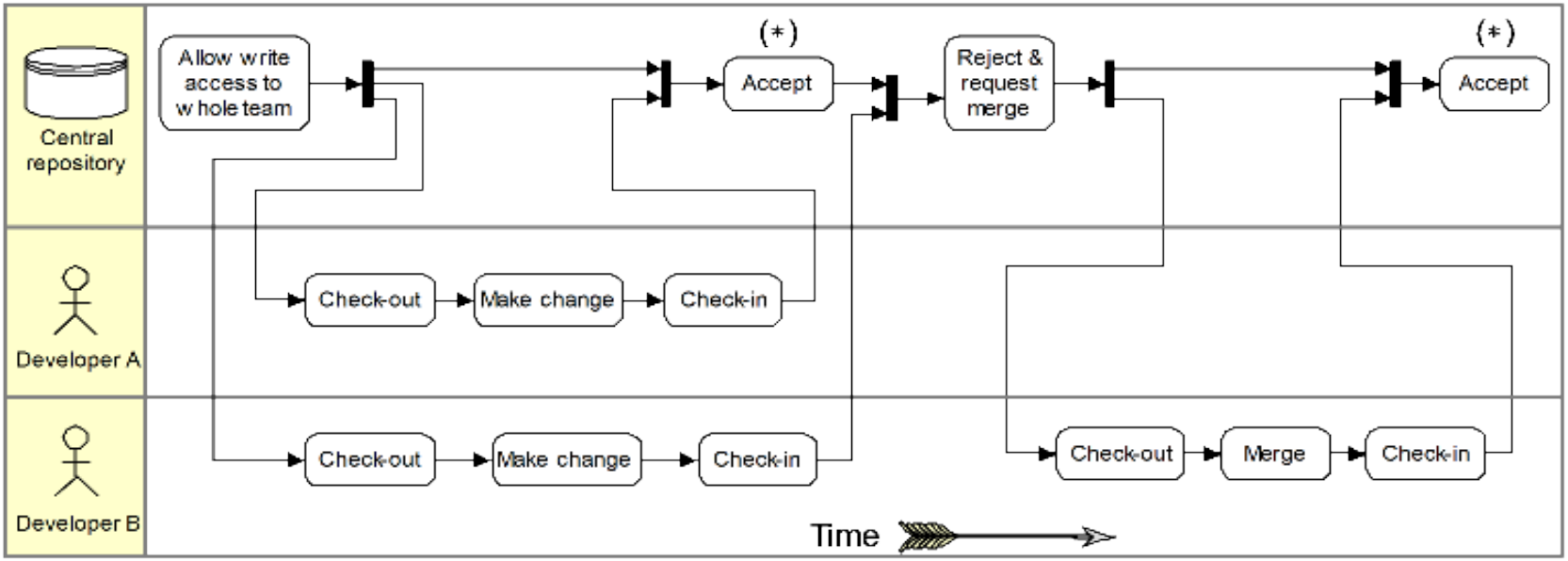
- We can undo a commit; undo a merge; or undo a checkout
- (See later how undo can be done without deleting the history)

Multi-pronged Product Evolution



- Parallel versions coexist at times, but will eventually merge
  (to a single trunk—Main branch)

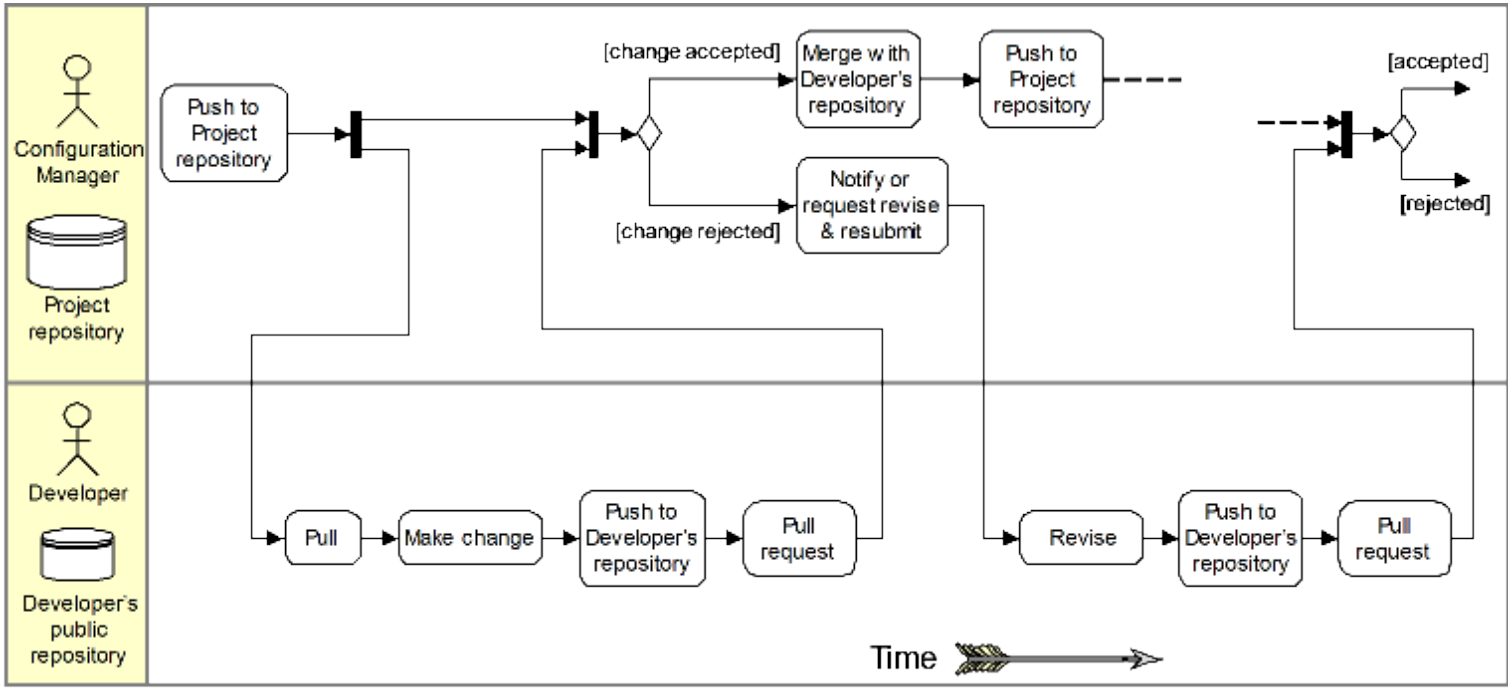
Working with Peers: Centralized Workflow



(*) Assuming no other commits in the meantime; otherwise need to merge

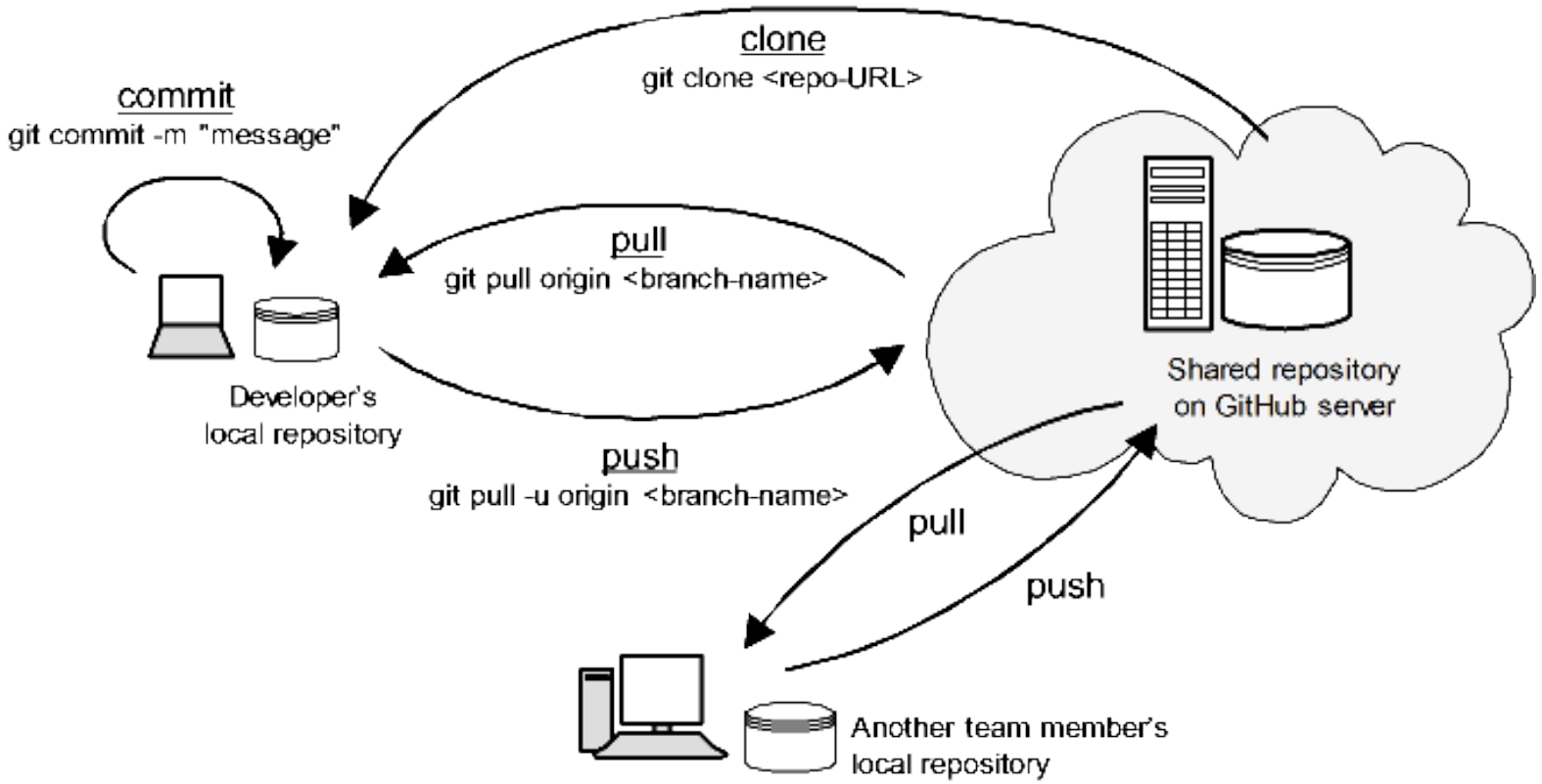
- Example scenario: Two developers clone from the hub and both make changes
- The first developer to push his changes back up can do so with no problems
- The second developer must merge in the first one's work before pushing changes up, so as not to overwrite the first developer's changes

Working with a Managed Team

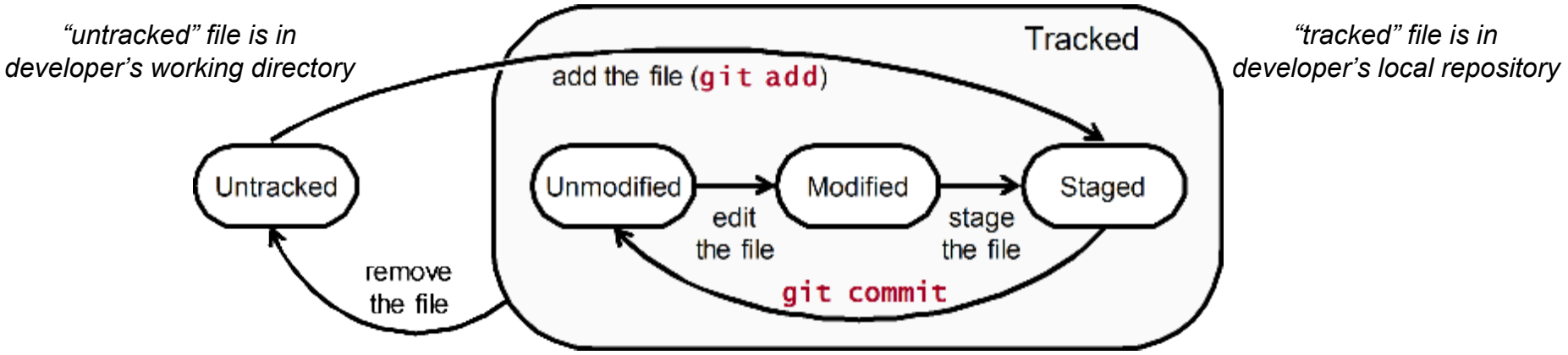


- Example scenario:
- 1. The configuration manager pushes the current version to the central project repository
- 2. A contributor clones that repository and makes changes
- 3. The contributor pushes the changed version to his own public repository
- 4. The contributor notifies the configuration manager requesting to pull changes
- 5. The configuration manager adds the contributor's repository as a remote and merges locally
- 6. The configuration manager pushes merged changes to the central project repository

Git Commands Summary

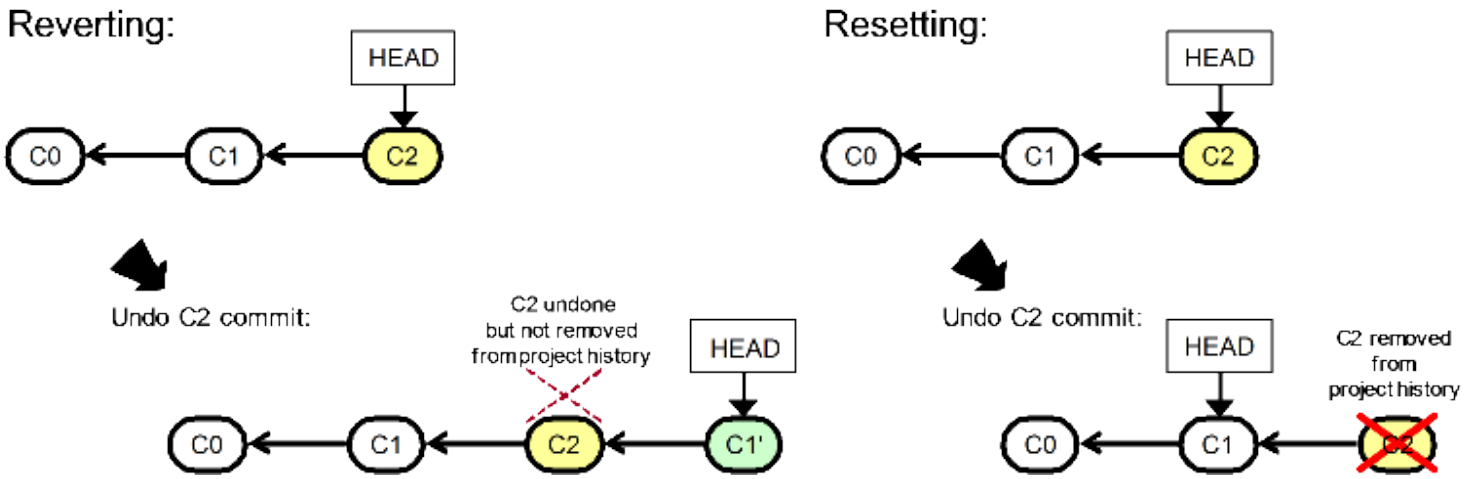


Git: States of Software Configuration Items



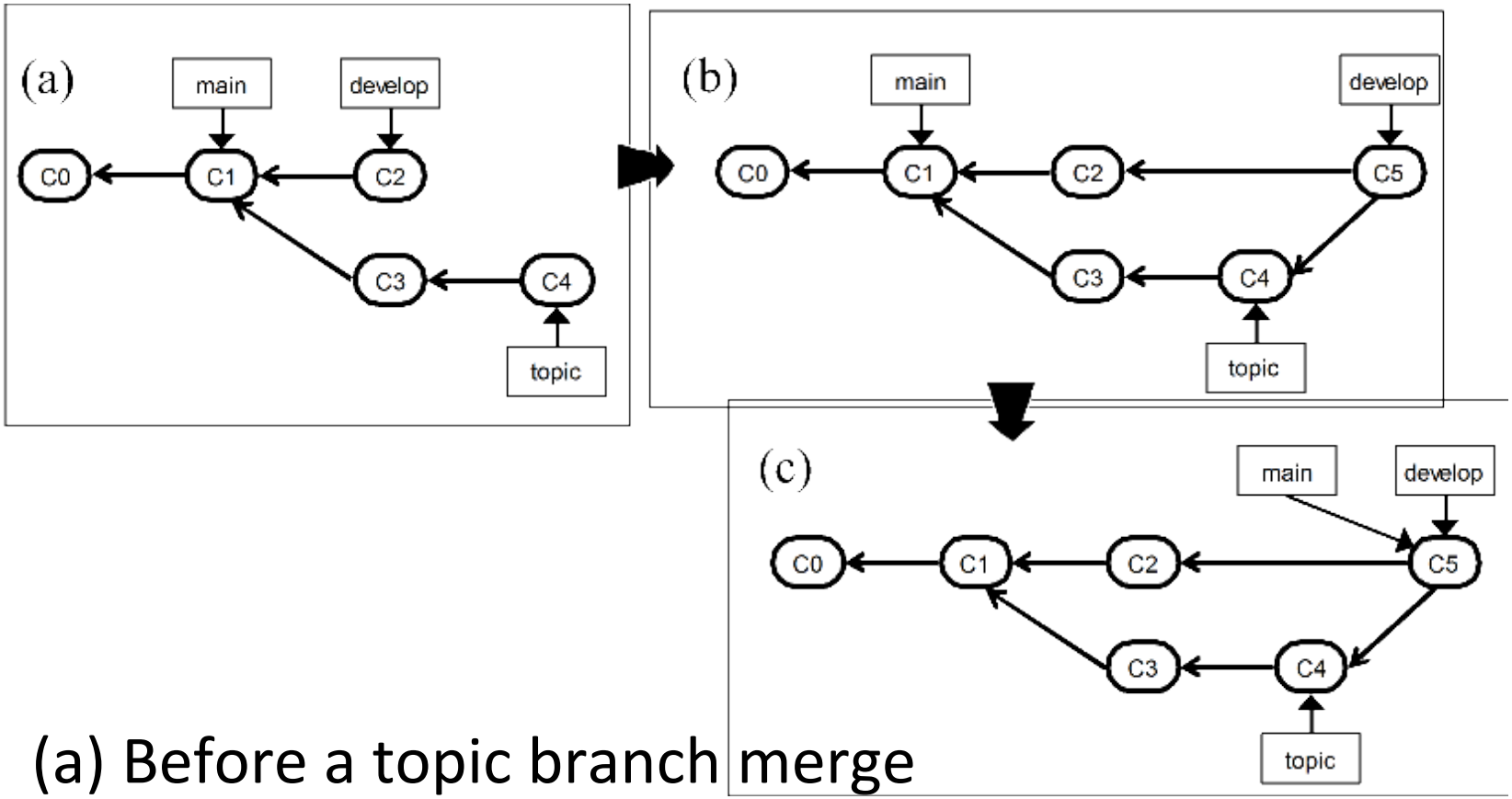
- **git add** is a multipurpose command (to begin tracking new files, to stage files, etc., e.g., marking merge-conflicted files as resolved
- Using git add we signal to Git that we wish to "add this content to the next commit" (place it to the "staging area")
- Next time we execute git commit,
  Git will "add this file to the project"

Undo: Reverting vs. Resetting



- git revert undoes a committed snapshot C2 by undoing the changes and appending a new commit C1' (instead of removing the commit C2 from the project history) ← "safe"
  - Historic record of abandoned versions exists
- git reset works backwards from the current commit by removing all of the commits after the target commit ← "unsafe"
  - No historic record of abandoned versions

Merge on Large/Important Projects



- (a) Before a topic branch merge
- (b) After a topic branch merge
- (c) After a project release ("main" tag is moved)

Rebasing Existing Branches

- Rebasing = moving a branch to a different baseline commit
  - Appears as if it forked off from a different version
  - All commits in the branch are *replayed* on new base