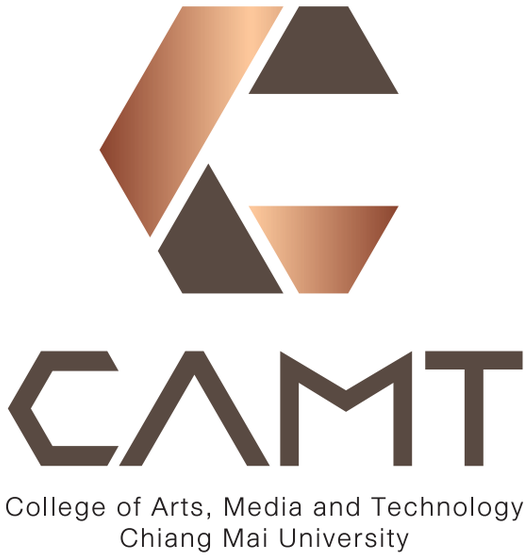
SE 234 Basic Development and Operations

#2 Version Control



Lect Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

Version Control — In Short

- A system or toolset that keeps tracking history of all the changes made in software projects.
  - Provide monitor access to the files
    - Every change made to the source is tracked
      - What is changed?
      - Who made it?
      - Why they made it?
      - References to the problem fixed

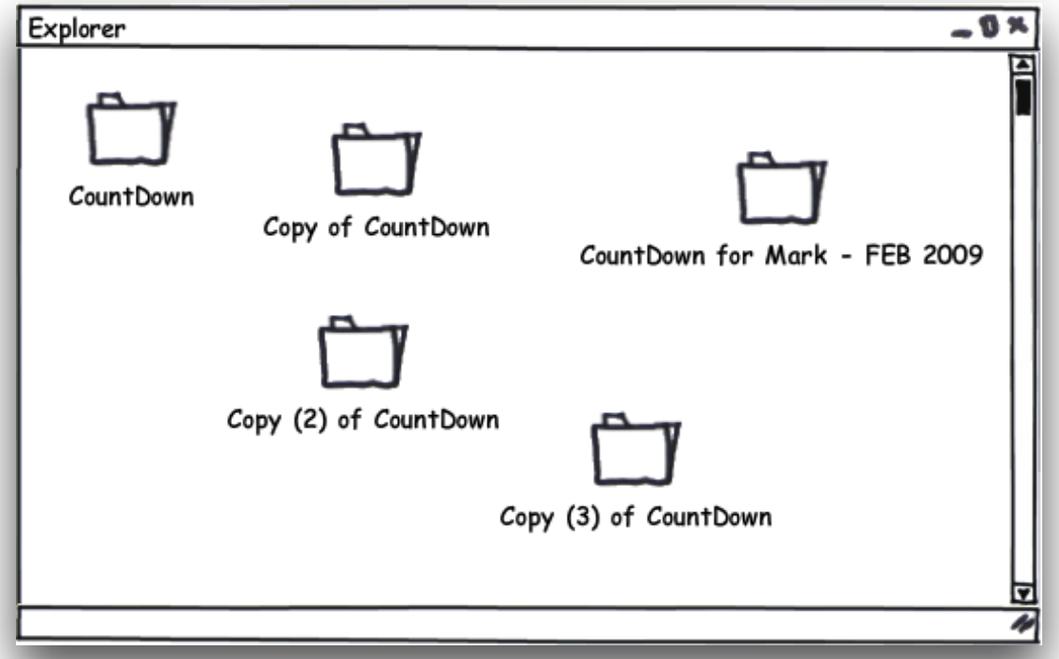
Version Control — In Short

Revision	Action	User	Date/Time	Description
3	Update Minerals.txt	Fred	March 22, 2014 10:18:39 AM	Delete "Potash" Add "Pyrite" and "Silica"
3	Update Vegetables.txt	Fred	March 22, 2014 10:18:39 AM	Delete "Sprouts" Add "Carrots"
2	Add Minerals.txt	Barb	March 21, 2014 12:40:22 PM	Add the Minerals.txt file
2	Update Animals.txt	Barb	March 21, 2014 12:40:22 PM	Delete "Skunk" Add "Elk"
1	Add Vegetables.txt	Fred	March 20, 2014 6:20:40 PM	Add the Vegetables.txt file
1	Add Animals.txt	Fred	March 20, 2014 6:20:40 PM	Add the Animals.txt file

Ref: https://www.red-gate.com/simple-talk/sql/sql-development/core-database-source-control-concepts/

Before Version Control

- File renaming
  - e.g., Draft1-Dec20.doc, Draft2-Dec28.doc
- New directory
  - /Version1, /Version1-1
- Zip file after a version
  - Jan10-Release1.0.zip



Difficult to track and review the changes when looking back what have been done

Basic Features

- A place to store your source code.
- A historical record of what have been done over time.
- A way for developers to work on separate tasks in parallel, merging their efforts later.
- A way for developers to work together without getting in each others' way.

Goals of a Version Control System

- We want people to be able to work **simultaneously**, not **serially**.
- When people are working at the same time, we want their changes not to conflict with each other.
- We want to archive every version of everything that has ever existed — ever.

Why we should use a VCS — scenarios

- We made change to several code-based files, and realized that there were mistakes requiring us to reverse them back.
- Some codes are lost due to mistaken removal.
- We have to seriously maintain multiple versions of a product, and there are overlapping contents between versions.
- We need to share our code or let other people work on the codes.
- We want to experiment with some fancy features, but we do not want them interfered the currently working codes.

Brief History

- 40+ years
- Generally divided into 3 generations
- First generation tools make utilize locks to provide concurrency — Only one person can work on the file at a time
- In the second generation, users are allow to work on the same file and then the tool will merge each users' copy and make a version.
- In the third generation, merging and making a version are independent.

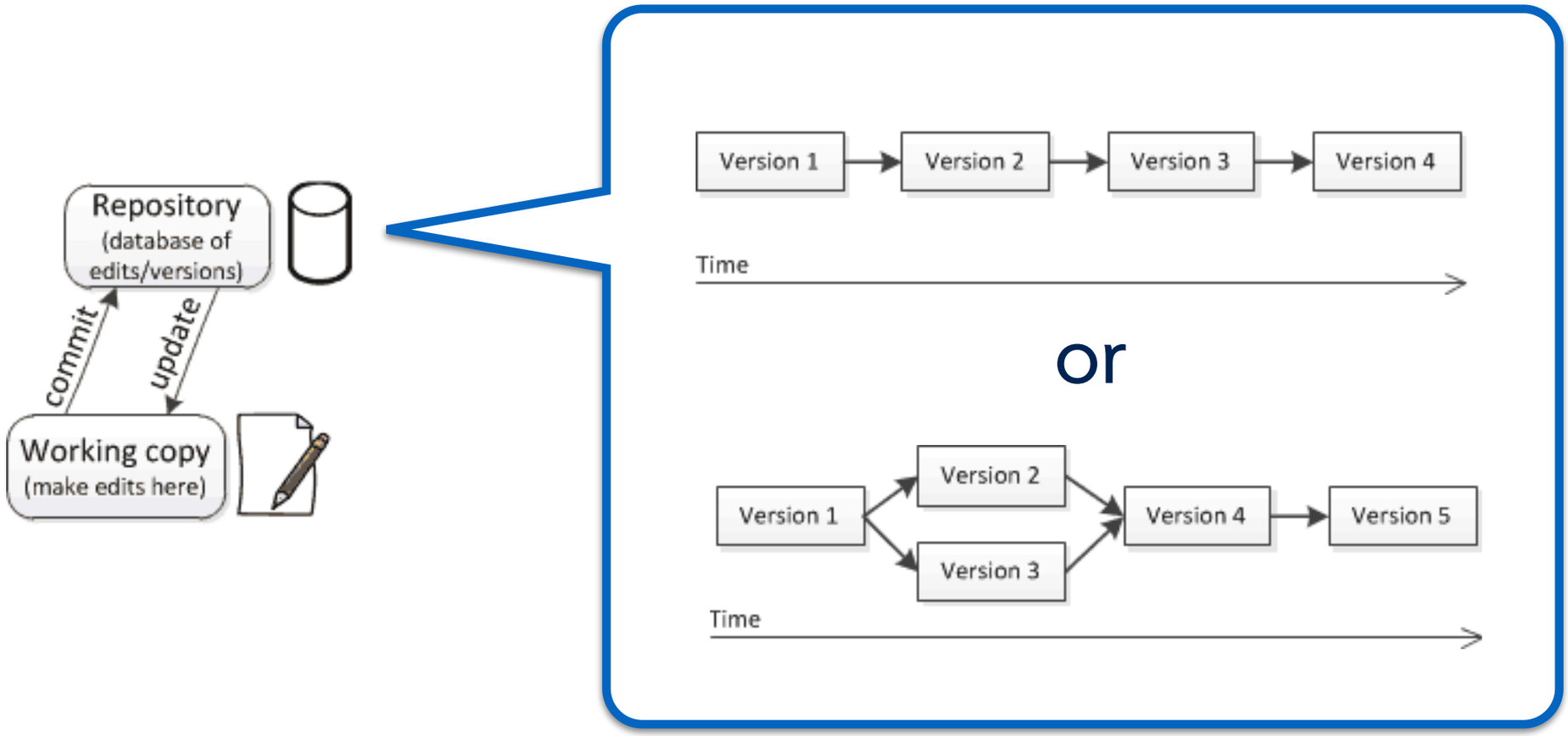
Terminology — Commonly Used Words

- **Repository** — The official place where you store all your work
- **Working copy** — A snapshot of the repository used by a developer as a place to make changes
- **Revision/Version** — A number refers a snapshot
- **Trunk/Master** — The main branch of a revision history
- **Branch —** to fork off the copy into two different directions
- **Head** — The latest version in a branch

Terminology — Commonly Used Words

- **Check-in/Push** — The act of sending a working copy to your repository to become a permanent part of its history
- **Check-out/Pull** — An operation used when you need to make a new working copy for a repository that already exists
- **Commit** — A variation for check-in
- **Delta/Diff** — A description of changes between two versions

Repo and Working Copies



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

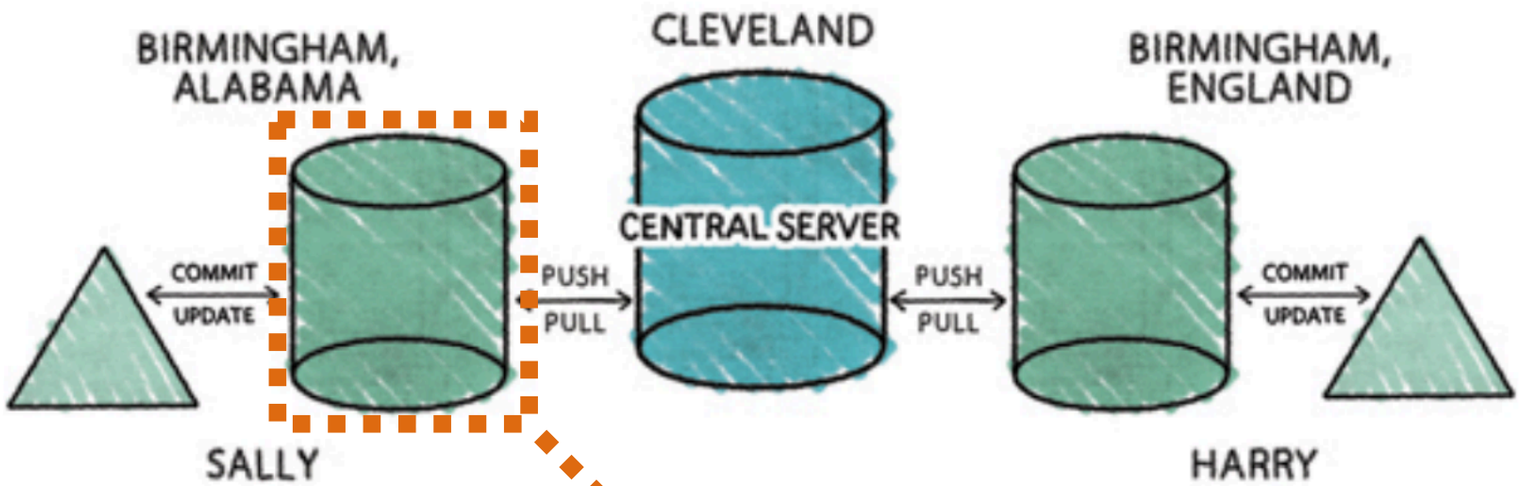
Type of Version Control System (VCS)

- Centralized
  - Early VCS
  - One project tends to have one repository
  - Single point of failure
    - Need to stay online all the time
- Decentralized (A.k.a. Distributed)
  - A project may have several different repositories
  - Support a sort of super-merge between repositories = Attempt to reconcile their change histories.

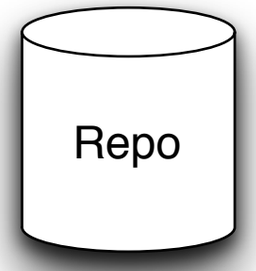
Operations

- Centralized VCS

- Decentralized VCS



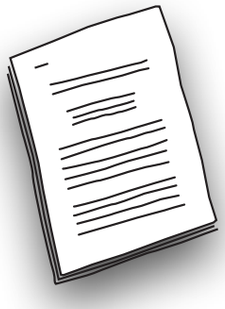
Operations

- **Create** 

  - Create a new, empty repository.
  - Repository has 3 dimensions
    - Directories — Tree structure which is the same as that of any file system
    - File Storage — Can be viewed as a simple network file system
    - Time — Every single version of the file being store in the repository

Operations

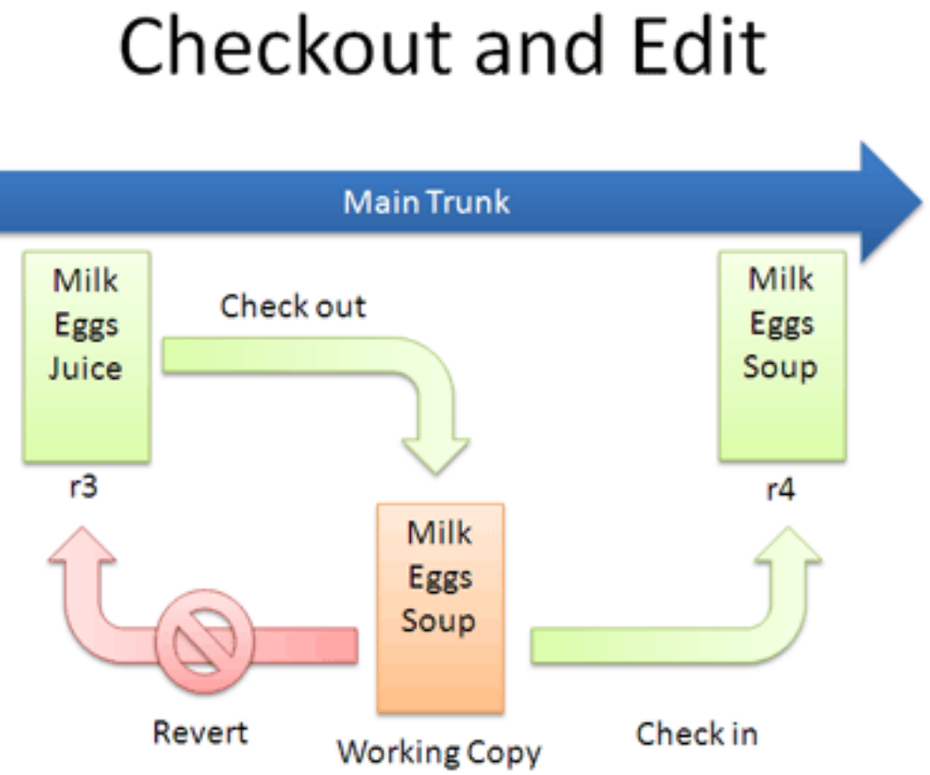
- **Checkout / clone**



  - Create a working copy.
  - Generally the repository is shared by the whole team.
  - But people do not modify it directly.
  - Rather, each individual developer works by using a working copy.
  - Allow one to work locally in their administrative area

Operations

- Check out / clone

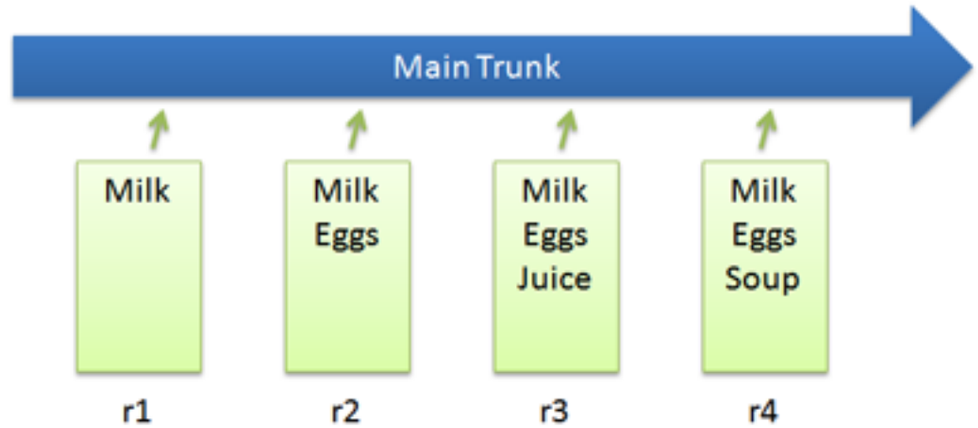


Operations

- **Commit**
  - Apply the modifications in the working copy to the repository.
  - Files to be uploaded to the repo is not the working copy version but what are changed from the master version being stored in the repo.
  - Comment messages should be sent along with the commit.

Operations

- Commit



Operations

- **Update/pull**
  - Update the working copy with respect to the repository.
  - Make the working copy up-to-date.
    - At the time of check-out, working copy is the same as what is being stored in the repo.
      - If anyone makes a commit to the repo, then the master version is changed.
      - Update operation helps synchronize it.

Operations

- **Add**
  - Add means add a file that is not under version control to the repository
- **Edit**
  - = Modify
- **Delete**
  - Remove file or directory from the repository
  - As all the history is kept, file is not really deleted.

Operations

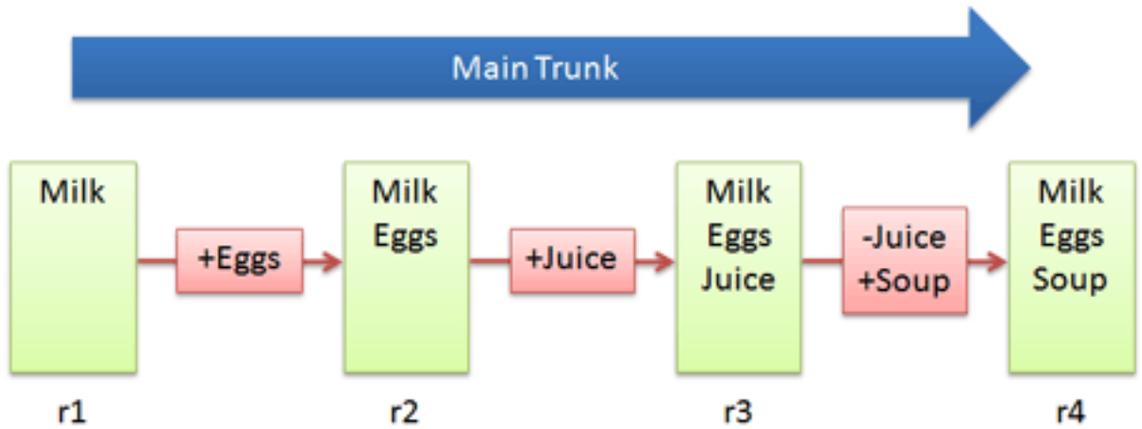
- **Rename**
  - Rename is quite tricky because history tracking is involved.
  - Different tools implement rename differently.
- **Move**
  - Is used when you want to move a file or directory from one place in the tree to another
  - Some tools treat rename and move as the same operation.

Operations

- **Diff**
  - Show the details of the modifications that have been made to the working copy.

Operations

- Diff



Note diff(r1,r4) = +Eggs +Soup

Operations

- Status
  - List the modifications that have been made to the working copy.
  - i.e., it shows you what changes would be applied to the repository if you were to commit.
  - = Show all Diffs

Operations

- **Revert**
  - Undo modifications that have been made to the working copy to a version.

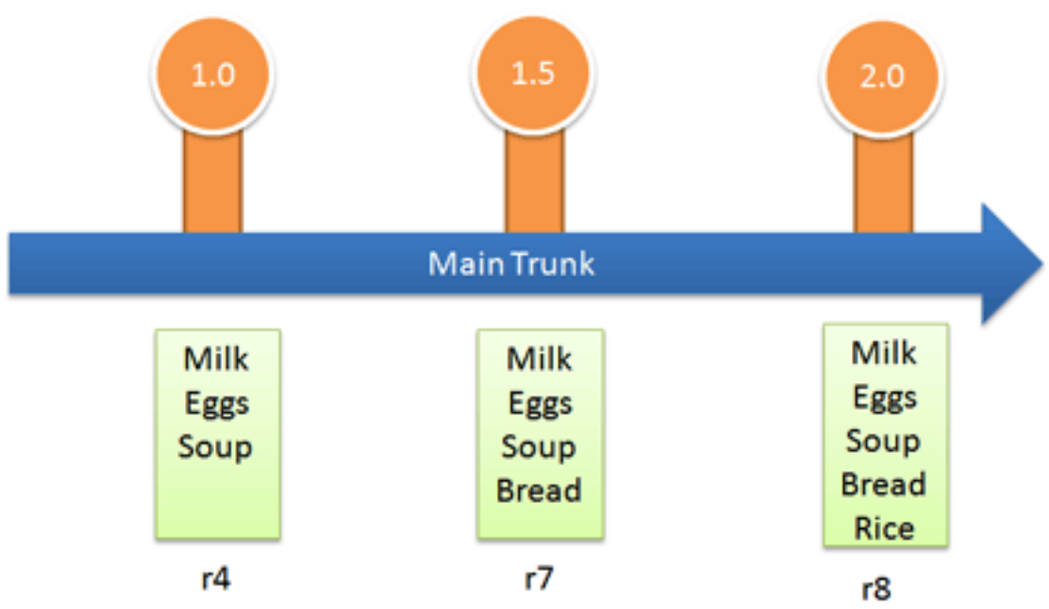
revert

Operations

- Log
  - Your repository keeps track of every version that has ever existed.
  - The log operation is the way to see those records.
  - If all the members properly provide comments whenever they make a commit operation, information that can be fully tracked will be
    - Who made the change?
    - When was the change made?
    - What was the explanation to the change?

Operations

- Tag
  - Associate a meaningful name with a specific version in the repository.

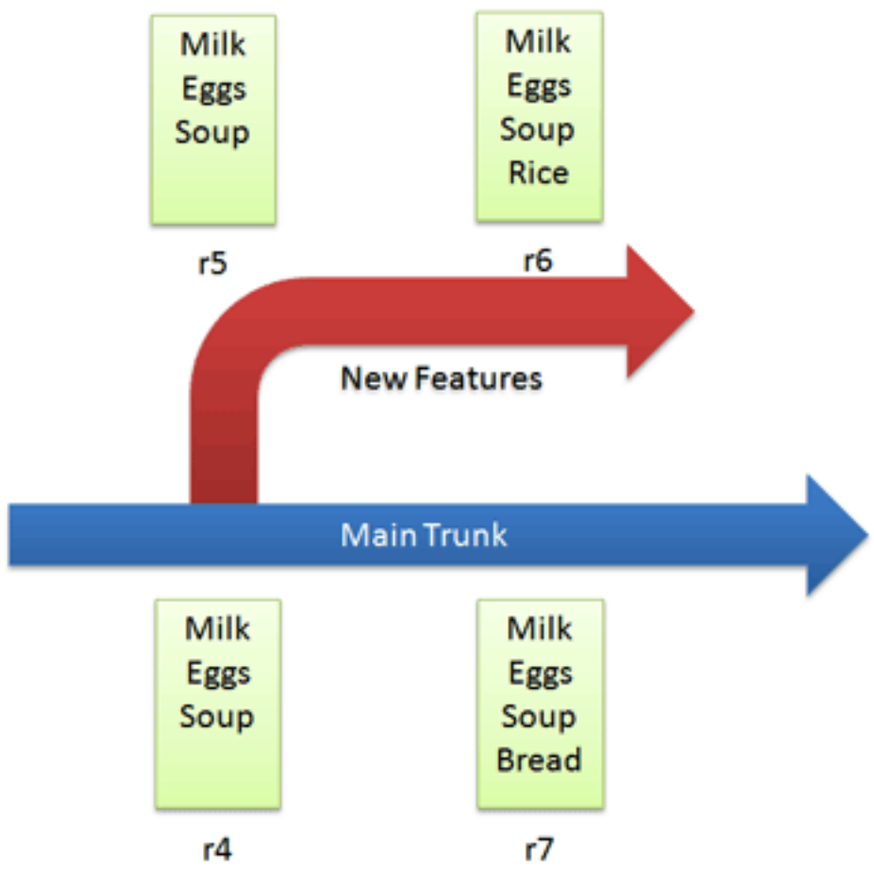


Operations

- **Branch**
  - The branch operation is what you use when you want your development process to fork off into two different directions.
  - For example, when you release version 3.0, you might want to create a branch so that development of 4.0 features can be kept separate from 3.0.x bug-fixes.

Operations

- Branch

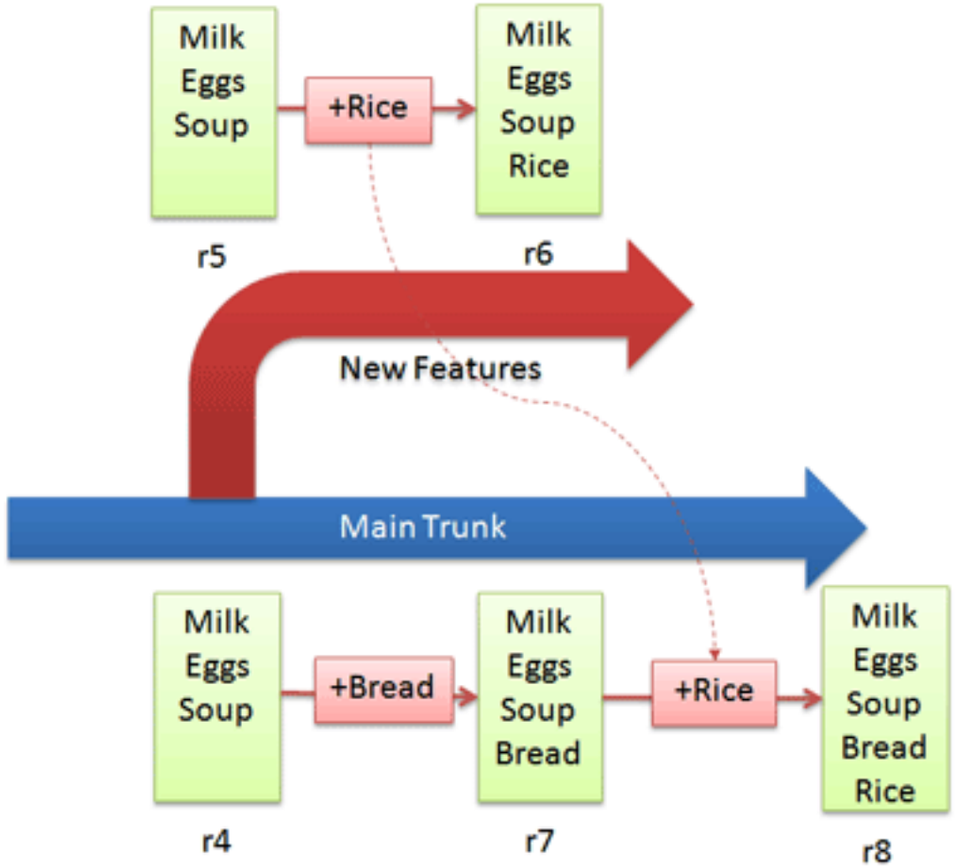


Operations

- **Merge**
  - Typically when you have used branch to enable your development to diverge, you later want it to converge again, at least partially.
  - For example, if you created a branch for 2.0.x bug-fixes, you probably want those bug fixes to happen in the main line of development as well.
  - Modern tools aim at making this merge feature as much automatically as possible.

Operations

- Merge



Operations

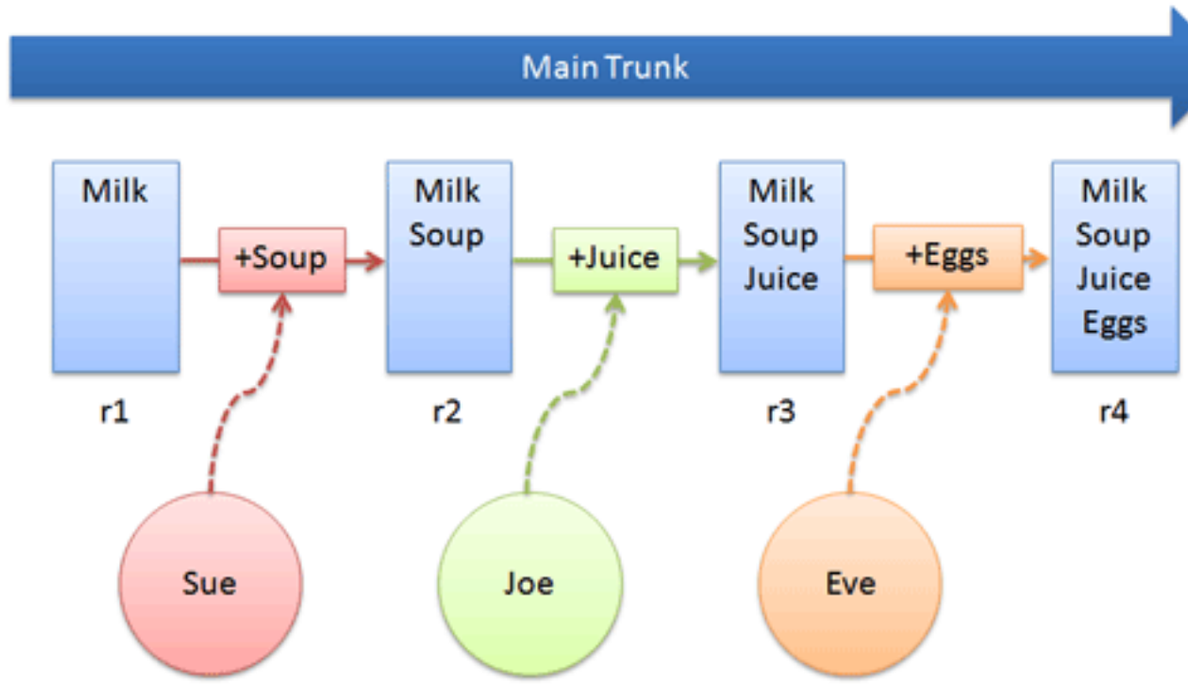
- **Resolve**
  - Handle conflicts resulting from a merge using human intervention.

Conflict

Operations

- **Resolve**
  - Merge automatically deals with everything that can be done safely.
  - Everything else is considered a conflict.
  - The resolve operation is used to help the user figure things out and to inform the VCS how the conflict should be handled.

Centralized VCS — Multi Users



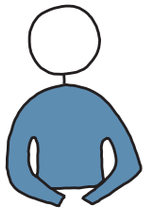
Basic Conflict Resolution in VCS

- The simplest way — locking

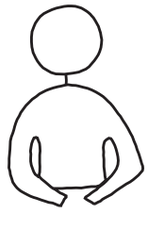
Alice locks the file and begins modify it.

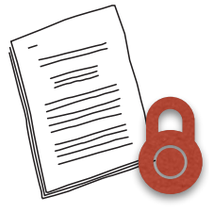
Bob attempts to modify the file but it is locked by Alice, so the file cannot be touched.

Bob is blocked, so he may get a cup of coffee or starts playing a game.



Alice finishes her changes and commits them.





Bob finishes his business, returns, and checkout the file.

Workflow too often looks more like this:

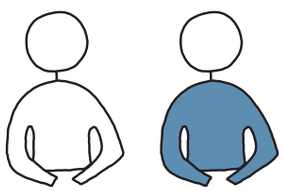
Alice locks the file and begins modify it.

Bob attempts to modify the file but it is locked by Alice,

Alice rushes off to an urgent meeting.

After a few games, Bob come back. However he is still unable to work on the file.

Bob feels frustrated and tells the VCS that he wants to steal the lock.



Both of Alice and Bob modify the file at the same time.

Changes are now in conflict, thus, locking is proven useless

Locking — Note

- If the VCS has no facility for stealing locks, the change conflicts can be prevented but Bob may be blocked forever.
- If Alice is not notified that her lock has been stolen, she may continue working on the file and later receive a surprise when she attempts to commit it.
- Even if conflicts can be resolved by merging working copies by hand, this scales poorly and tends to frustrate all the party members.

Locking — Note

- Historically, locking was the first invented conflict-resolution method.
- It is also associated with first-generation centralized VCS.
- Locking is still used in modern toolsets, but only for non-textual files, such as image and video.

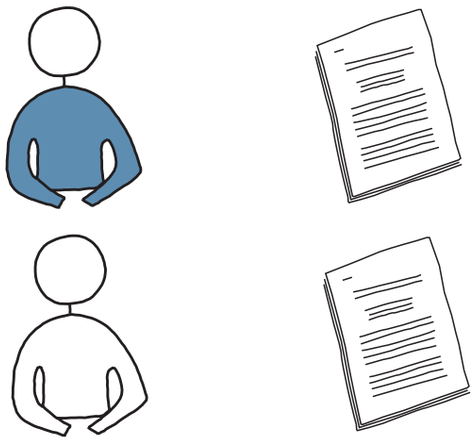
Conflict Resolution in Centralized VCS

- Conflict Resolution in Centralized VCS usually uses a method named merge-before-commit
- In short, Centralized VCS will notify you if you are attempting a commit against a file that has been changed since you started editing, and requires you to resolve the conflict before you can commit it.

Ask you to MERGE before letting you COMMIT it

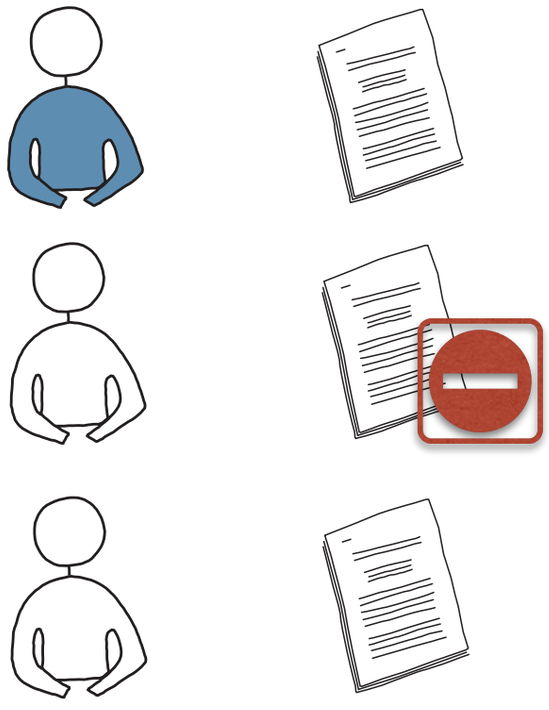
Conflict Resolution in Centralized VCS

- The workflow



Alice checks out the file and begins modifying it.

Bob checks out the same file and begins modifying it.



Alice finishes her changes and commits the file.

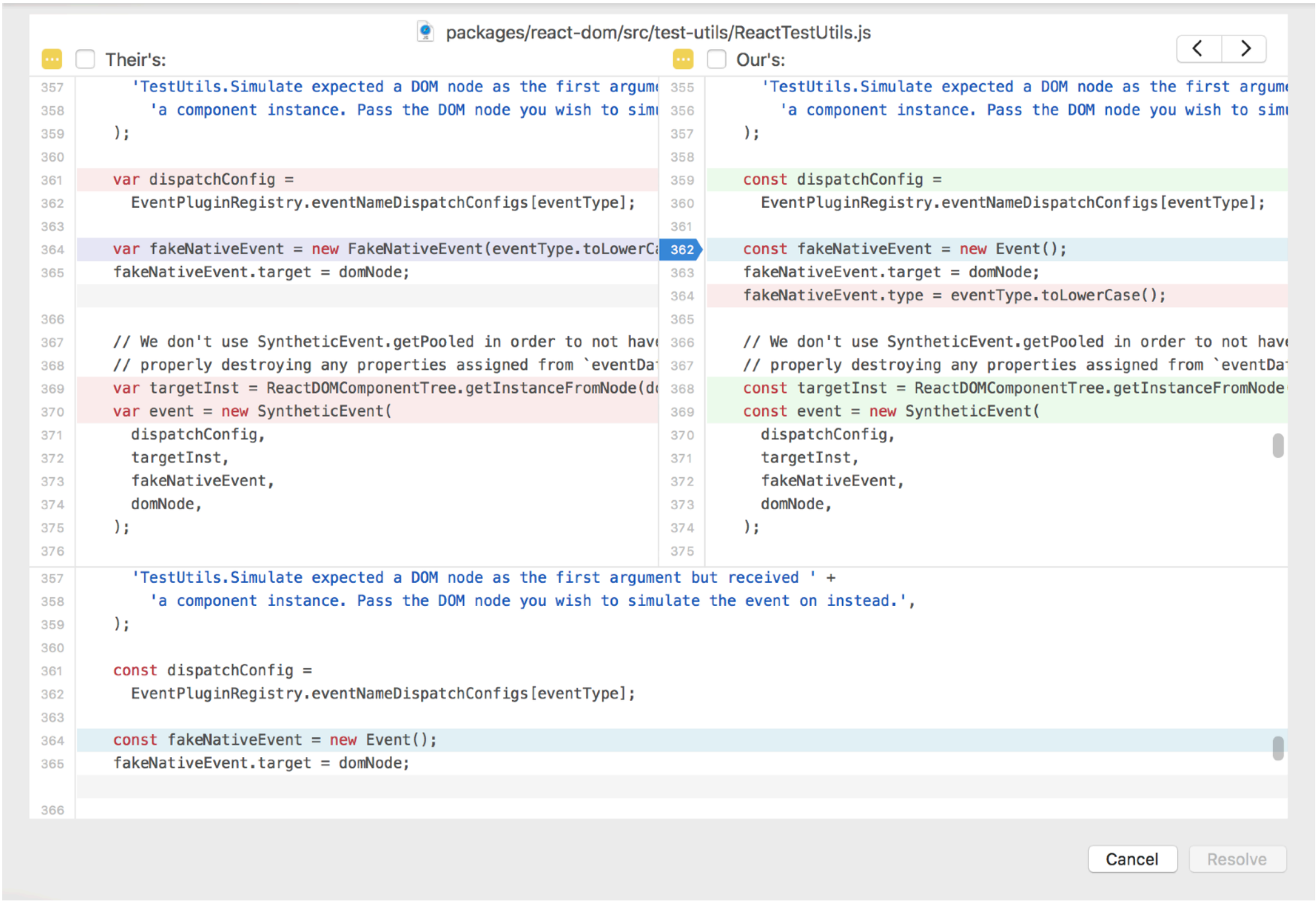
Bob attempts to commit but the VCS tells him that the version has been changed after he checked out, and he has to resolve the conflict before committing it.

Bob runs a merge command that applies Alice's changes to his working copy. Now he can commit.

Conflict Resolution in Centralized VCS

- With this workflow, there will be no conflict presenting in the repository
- The merge command shall return success, and the VCS will allow Bob to commit the merged version.

Conflict Resolution in Centralized VCS



Conflict Resolution in Centralized VCS

- In practice, merging based on the **merge-before-commit** do not generally produce any conflicts.
- Once they do occur, a **merge-before-commit** VCS will typically put conflicting lines of both working copies and marking them with some kind of conspicuous marker.
- VCS will accept the commit once Bob has edited out the marked contents.

Problems with Centralized VCS

- It does not scale well when there are many collaborators
- Cannot check-in half-finished work to the master
- Cannot keep track of multiple branches for every collaborators

As the project grows, you may want to split features into chunks, developing and testing in isolation and slowly merging changes into the main line.

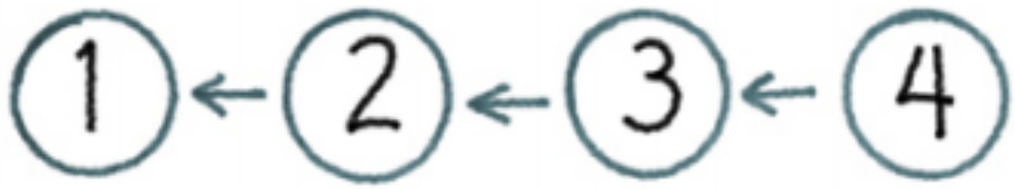
Problems with Centralized VCS

- Even if merging is always **possible** in a centralized VCS, it seems to be cumbersome.

Anyway, compared with not using a VCS at all,
using any VCS is a positive step forward for any project.

Algorithmically Explanation

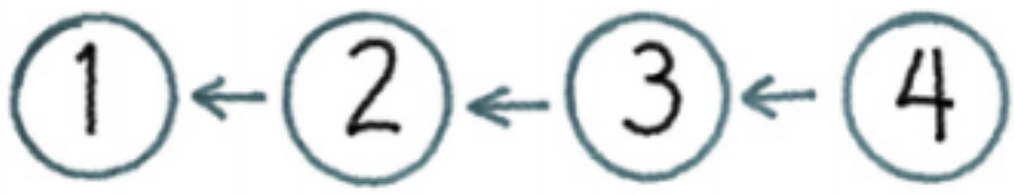
- In a centralized VCS, the history of everything in the repository is modeled as a linear model



- To create a new version, one grabs a version, make some changes, and check the changes back in.
- This is very simple as it provides an unambiguous answer to the question of which version is latest.
- Linear — need to merge before commit

Algorithmically Explanation

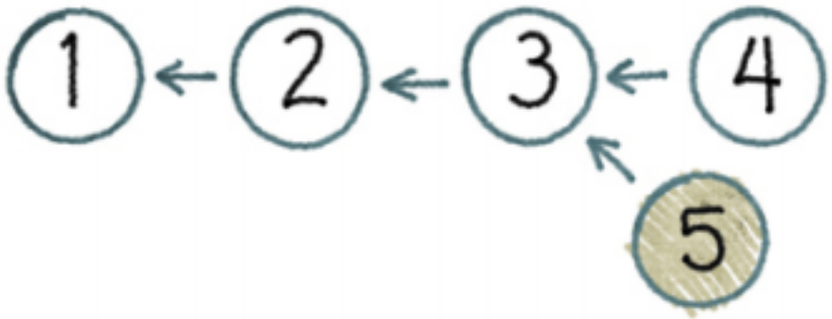
- But the linear model has one big problem: You can only commit a new version if it was based on the latest version.



- Bob grabs the latest version, and at the time, it was version 3.
- Bob makes some changes to it.
- While Bob is doing this, somebody else commits version 4.
- When Bob goes to commit his changes, he cannot, because they are not based on the repository's current version.
- The parent for Bob's changes was version 3.

Algorithmically Explanation

- With linear model, we are not able to crate the following version 5:

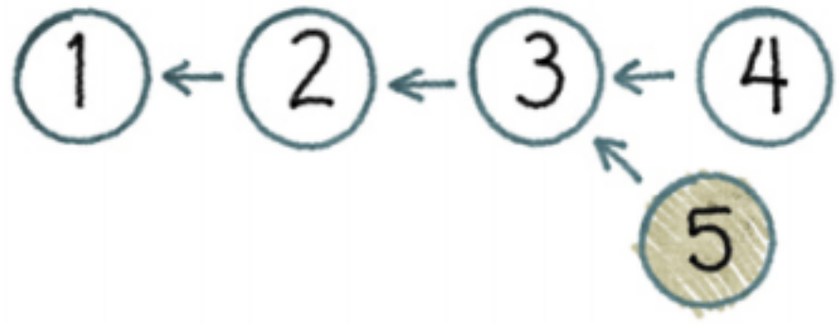


- Instead, in a linear VCS, it is required to take the changes which were made between versions 3 and 4 and apply them to the latest modified version.
- This is unfortunate. Changes were intended to be made against version 3, but now they are getting blended with the changes from version 4.

What if they do not blend well?

Designed Structure for Representing Versions of Things

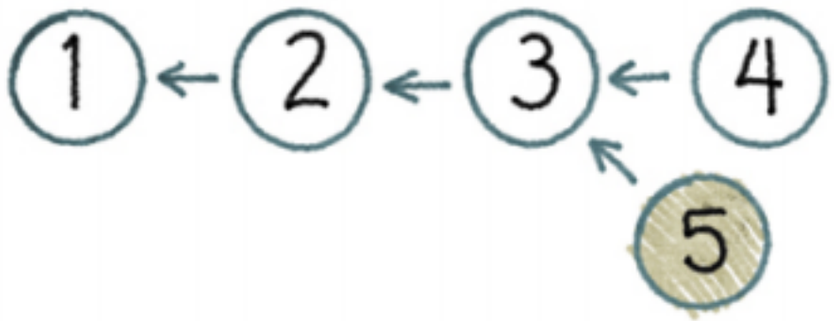
- If we allow version 3 to be the parent of the version 5, then our history would no longer be a line.
- Instead it would be a **Directed Acyclic Graph (DAG).**



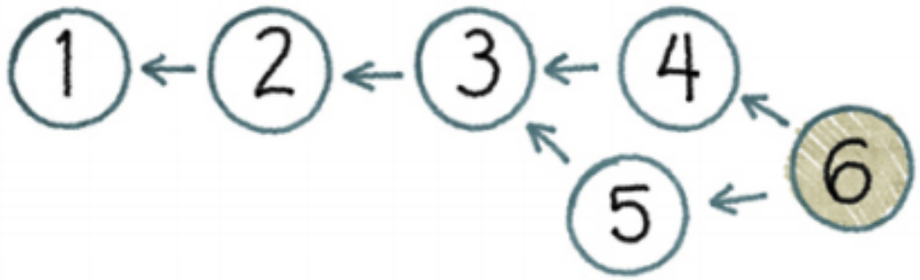
- The DAG is probably a more pure representation of what happens in a team practicing concurrent development
  - DAG does not interrupt the developer at the moment he/she is trying to commit his/her work.

Designed Structure for Representing Versions of Things

- However, making a DAG, we may not know which is the actual latest version.

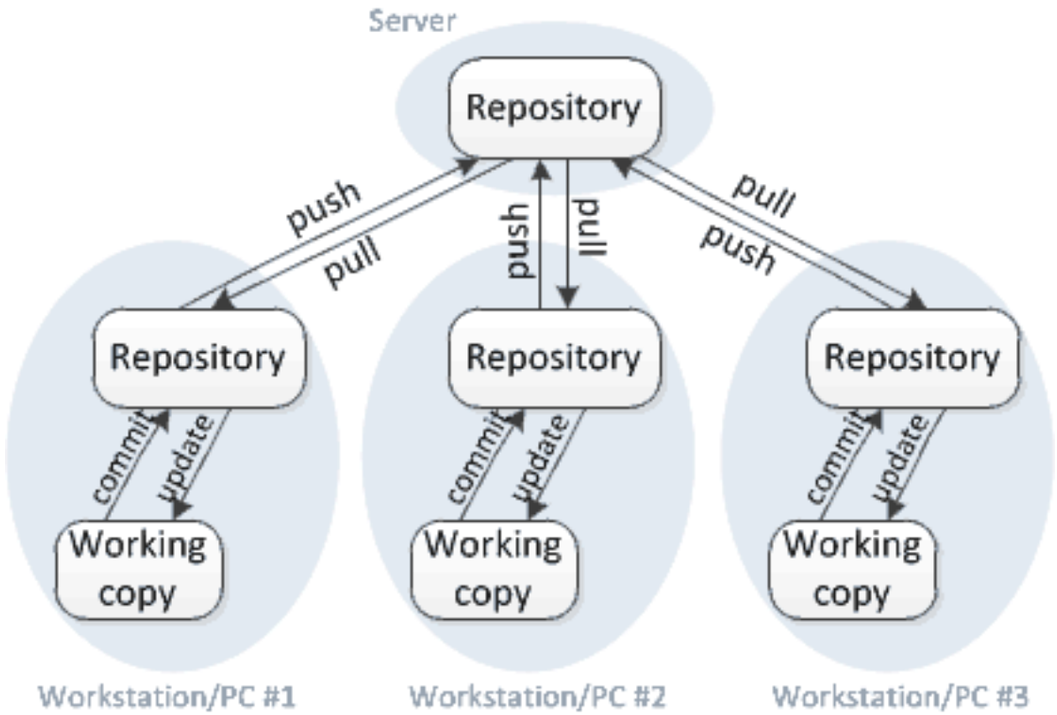


- This kind of problem can be simply solved by applying a merge.



- Therefore, in a **Distributed VCS**, committing and merging become independent.

Distributed VCS



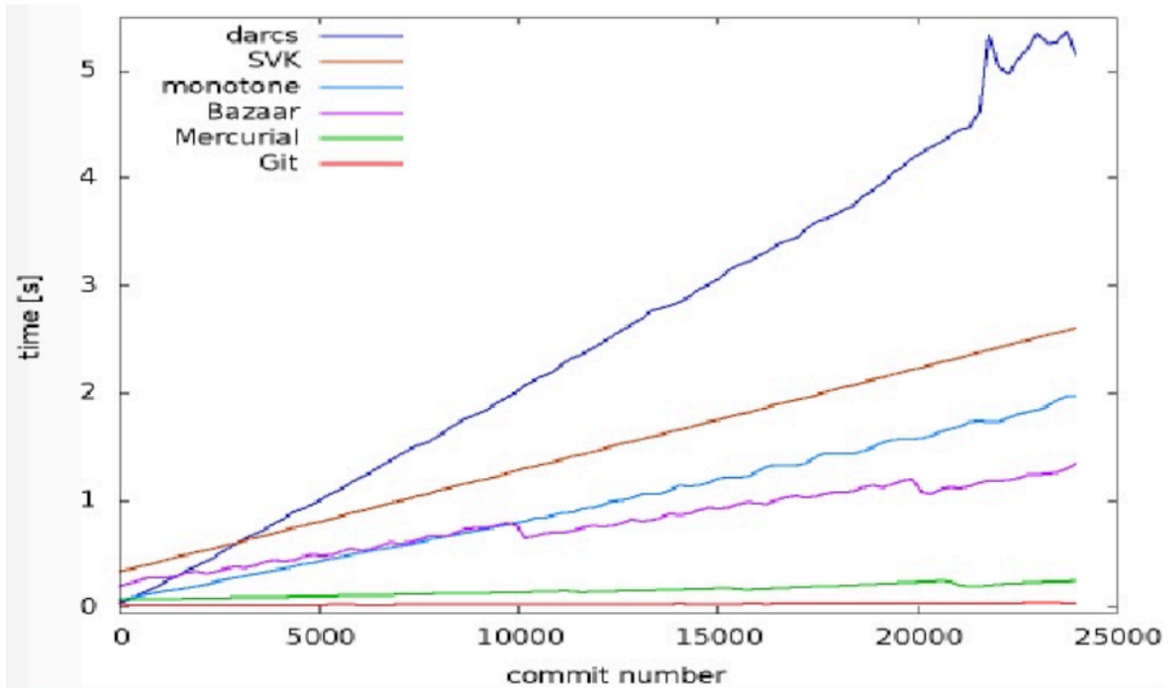
- Everyone has their own repository, or a clone of a repository.
- Commits are executed locally, but changes are distributed via pushes and pulls.

Distributed VCS — Benefits

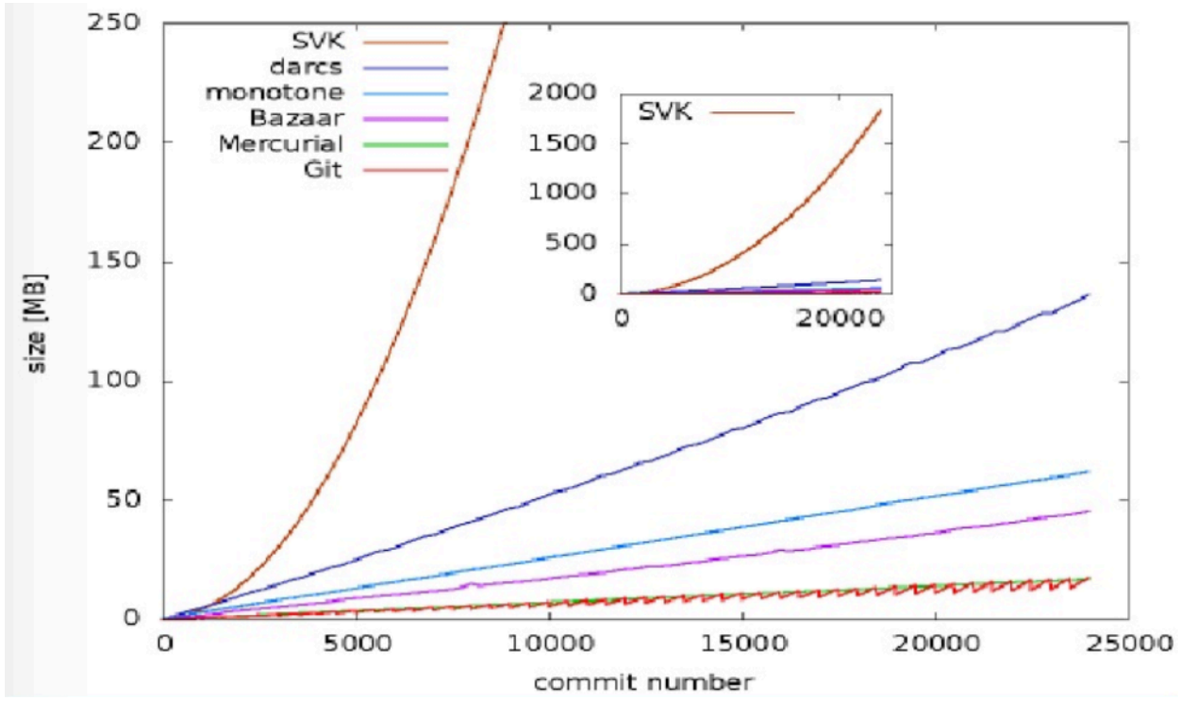
- More flexible — It allows different workflows and collaborative behavior.
  - Private workspace — private copy of the entire repository.
  - Similar to the multithread paradigm, we get the maximum performance when we are able to avoid thread synchronization as much as possible.
  - The act of committing a change to a repository instance is distinct from the act of publishing that change to the rest of the team. — E.g., file staging in Git

Distributed VCS — Benefits

- It is faster



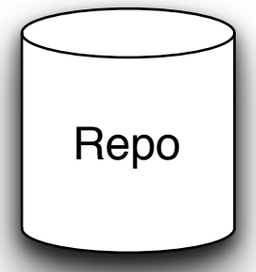
- It is more efficient



Benefits Explained in Short

- Commit can be performed offline, then, sync later.
- Cheap local branching helps facilitating easy experiments.
- Each working copy is a complete back up of the repository.

Operations — Distributed VCS

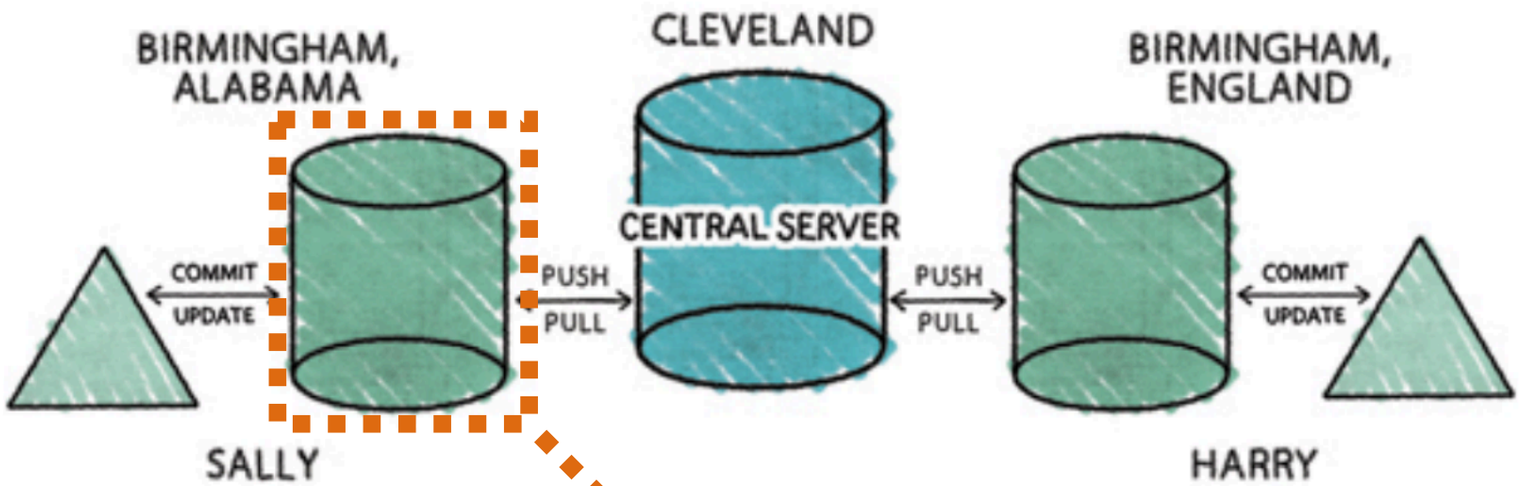
- Clone a 

  - **Create** a new repository instance that is a copy of another.
  - The notion of a repository instance is one of the main differences between Centralized VCS and Distributed VCS
    - In a Centralized VCS, the repository exists in one place on a central server. Every collaborator accesses it for all the operation.
    - In contrast, a Distributed VCS allows the repository to exist in more than one place. Therefore, we will need ways of keeping them synchronized.

Operations

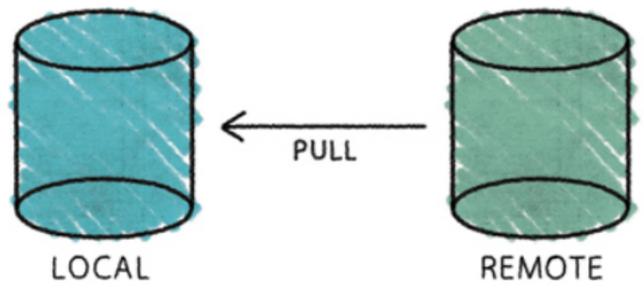
- Centralized VCS

- Distributed VCS



Operations — DVCS

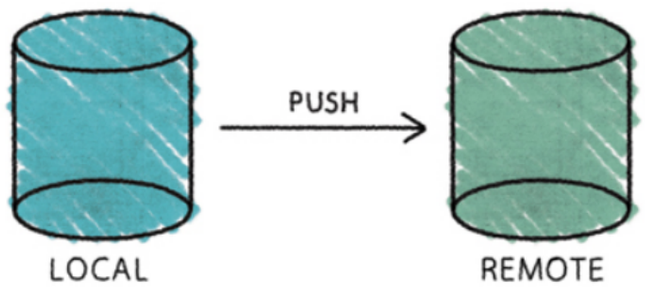
- **Pull**



  - Copy changes from a remote repository instance to a local one.
    - To synchronize between **two** repository instances.
      - Can be one to any other instances
    - Usually, the remote instance is the one from which the local was cloned.

Operations — DVCS

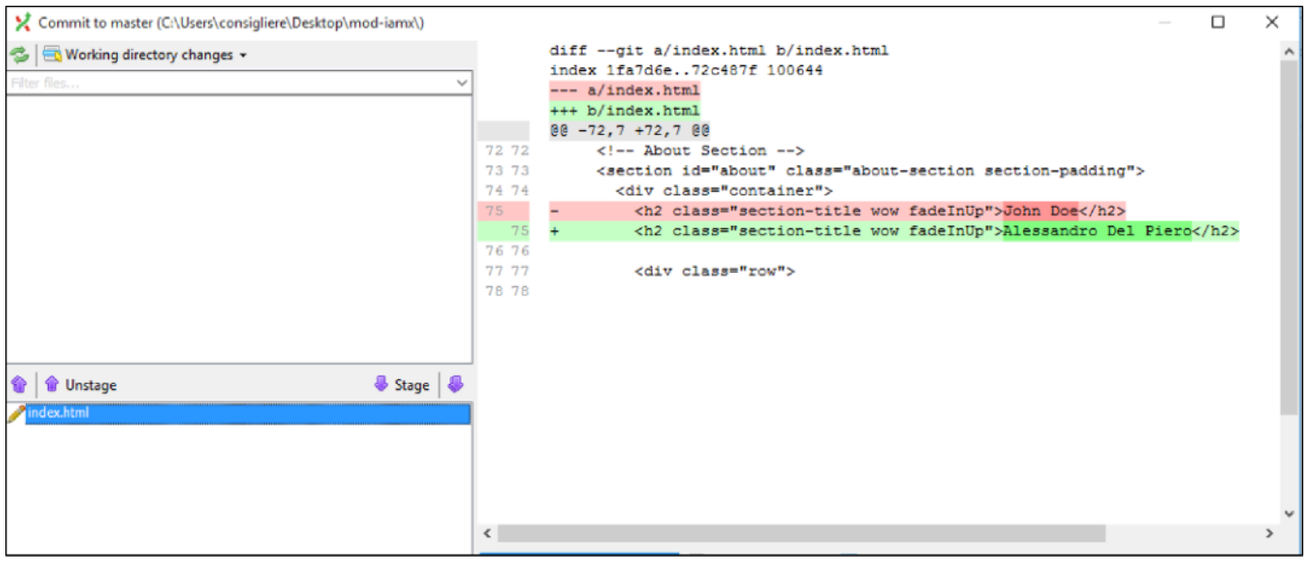
- **Push**



  - Copy changes from a local repository instance to a remote one.
    - To synchronize between **two** repository instances.
      - Can be one to any other instances
    - Usually, the remote instance is the one from which the local was cloned.

Some Insightful Notes

- Git has a staging area.
  - This makes a two-step checkin
    - Stage then commit

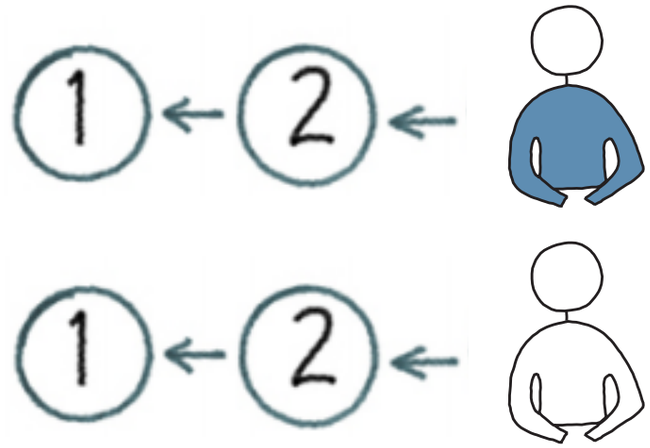


Some Insightful Notes

- Branches are like how we used **save as** in the past.
  - Easily merge changes with the original
  - No wasted space — the same files are entirely stored once
  - Not only files, but also directories

Conflict Resolution in Distributed VCS

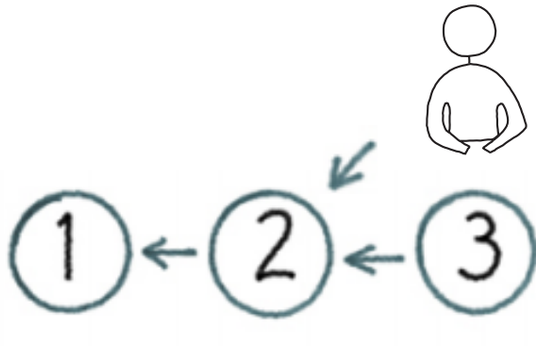
- Commit before merge



Alice checks out a copy at version 2

Bob also checks out a copy at version 2

Alice finishes her changes before Bob and commits her version.

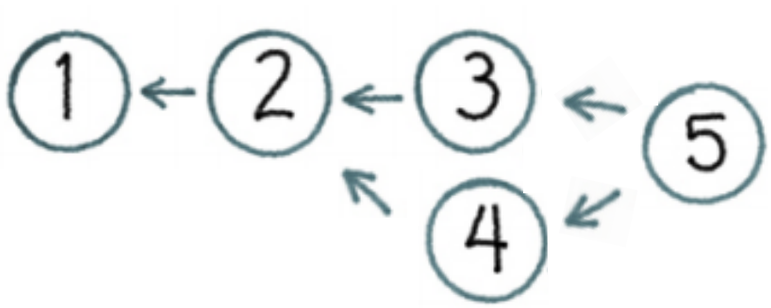


At the time Bob attempts to commit, the VCS informs Bob about the changes in the main version.

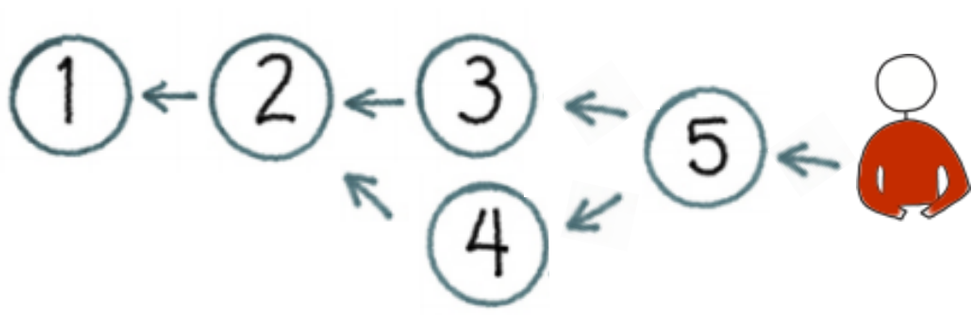
- His commit will begin a new branch

Conflict Resolution in Distributed VCS

- Commit before merge



If Bob wants to merge his version with Alice's, he will run a merge tool. The merge will be the descendant of Bob's and Alice's.



If another developer checkouts the file after the Bob's merge, she will retrieve the merged version.

Compared to Locking

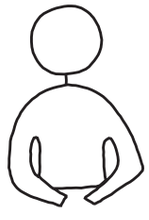
- The simplest way — locking

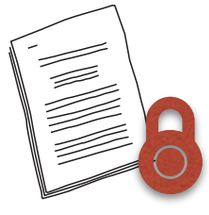
Alice locks the file and begins modify it.

Bob attempts to modify the file but it is locked by Alice, so the file cannot be touched.

Bob is blocked, so he may get a cup of coffee or starts playing a game.

Alice finishes her changes and commits them.

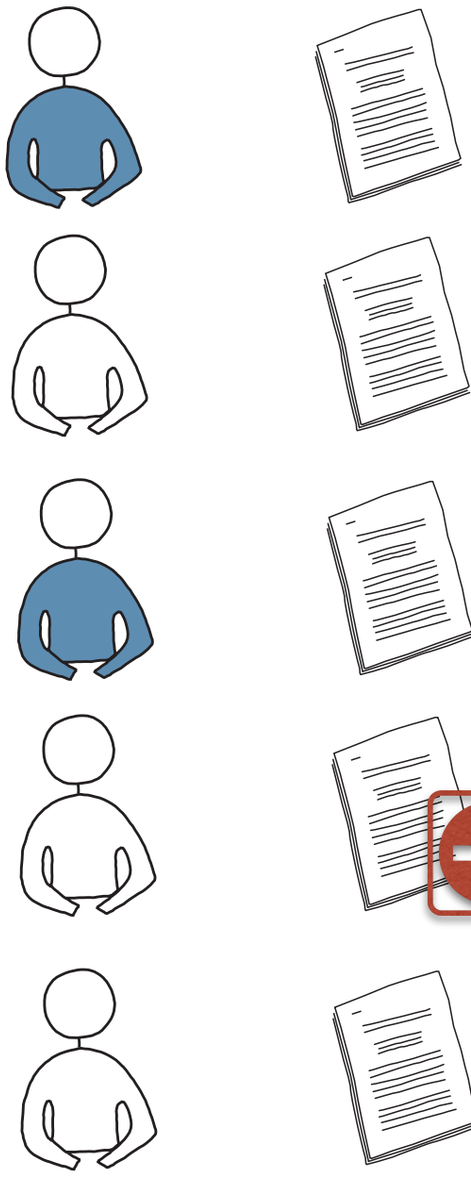




Bob finishes his business, returns, and checkout the file.

Compared to Merging before Commit

- The workflow



Alice checks out the file and begins modifying it.

Bob checks out the same file and begins modifying it.

Alice finishes her changes and commits the file.

Bob attempts to commit but the VCS tells him that the version has been changed after he checked out, and he has to resolve the conflict before committing it.

Bob runs a merge command that applies Alice's changes to his working copy. Now he can commit.

Summary — Version Control Enables

- Backup and restore
- Synchornization between multiple machines and/or between multiple corroborators
- Massive undo and redo
- Comparing arbitrary historical version — tracking changes
- Branching and merging back when needed
- Maintaining multiple live versions

Question Times