UECS2363 SOFTWARE CONSTRUCTION AND CONFIGURATION
CHAPTER 3 : SCM

LOO YIM LING
ylloo@utar.edu.my

UECS2363 SOFTWARE CONSTRUCTION AND CONFIGURATION
CHAPTER 3 : SCM (PART I)

SCM?

- Software Configuration Management
- Software Change Management
- Source Code Management
- Revision control system
- Version control system

What’s the problem?

- More work you do, the more you can lose
- The more iterations you make, the harder it is to remember what was in each one, harder to retrace
- More people involved, more likely they are to conflict with each other
- Undo and Redo need to be macro-scale … “Big Smart Backup”

SCM Benefits

- Collaboration
  - SCM tools prevent one user from accidentally overwriting the changes of another, allowing many developers to work on the same code without stepping on each other's toes.
- History
  - SCM tools track the complete development history of the software, including the exact changes which have occured between releases and made those changes.

SCM Benefits

- Release notes generation
  - Given the tracking of each change, the SCM can be used to generate notes for their software releases which accurately capture all of the changes included in the new release.
- Documentation and test management
  - SCM tools can be used to manage not just software source code, but also test suites and documentation for their software.

SCM Benefits

- **Change notifications**
  - **To keep interested members of the team informed when changes occur to the source code.**

SCM Users

- **Project Developers**
  - **writing source code, individual or team work**
- **Open Source Communities**
  - **project developers, to the nth degree**
- **Advanced Users / Education**
  - **wishing to examine source code**

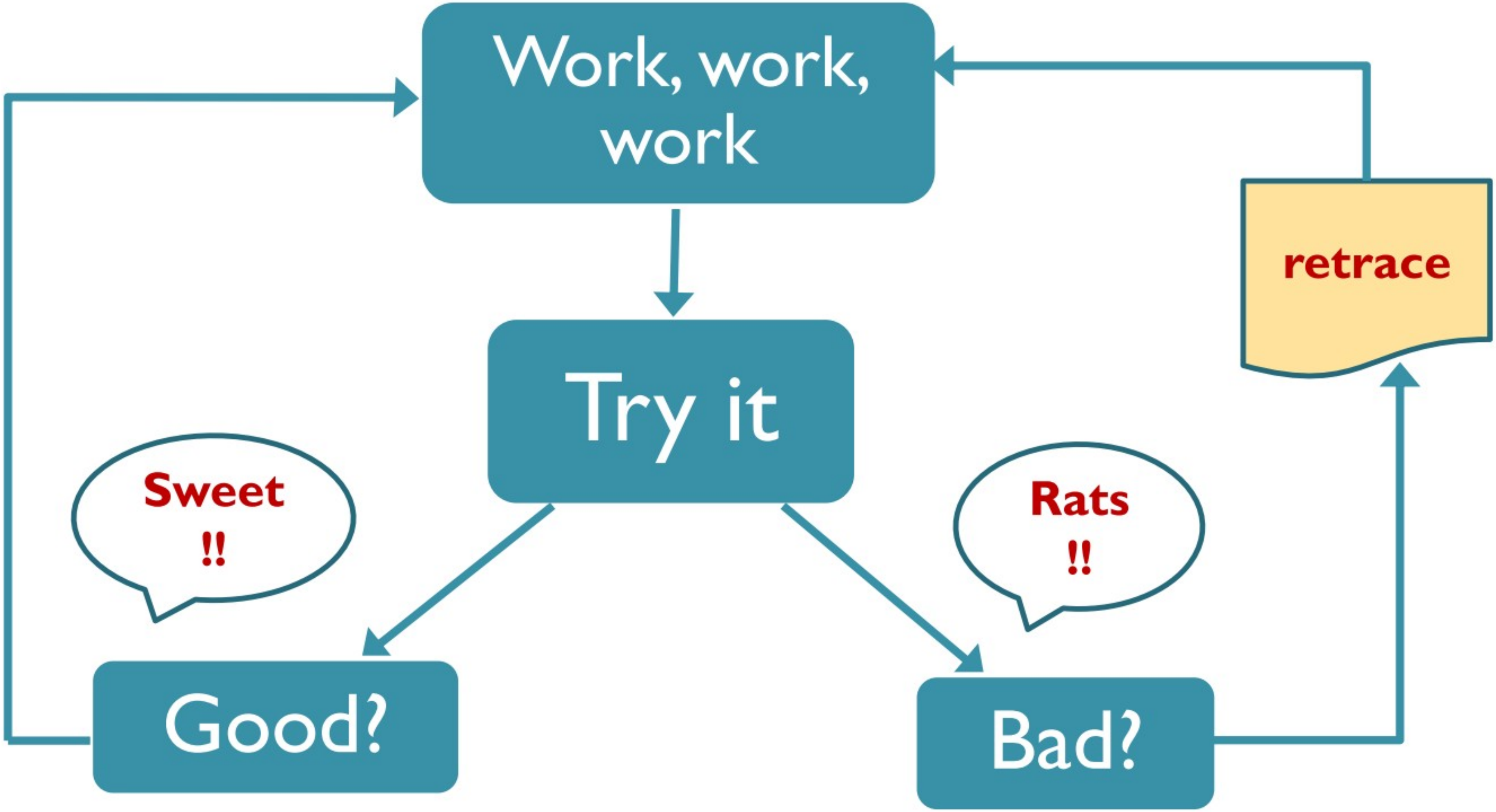
SCM Users

- **Testers**
  - **needing to download the latest release/version**
- **Archives / History**
  - **FreeBSD CVS tree goes back to mid ’80s and more**

SCM Objects

- Project
- Source code
- Tests (code)
- Document / file / binary
- Build scripts
- Reporting / Notification scripts
- Version tree
- Log / History

Common Work Cycle



Manual Retrace/Versioning

- **Keep backup folders (v1, v2-stable, etc.)**
- **Tarballs (v1.tar.gz, v2.tar.gz, etc.)**
- **Comment out large chunks of code**
- **Write down notes (ReadMe, help.txt, code comment blocks?)**
- **Maybe save it on a file share if you are thinking backup safety**

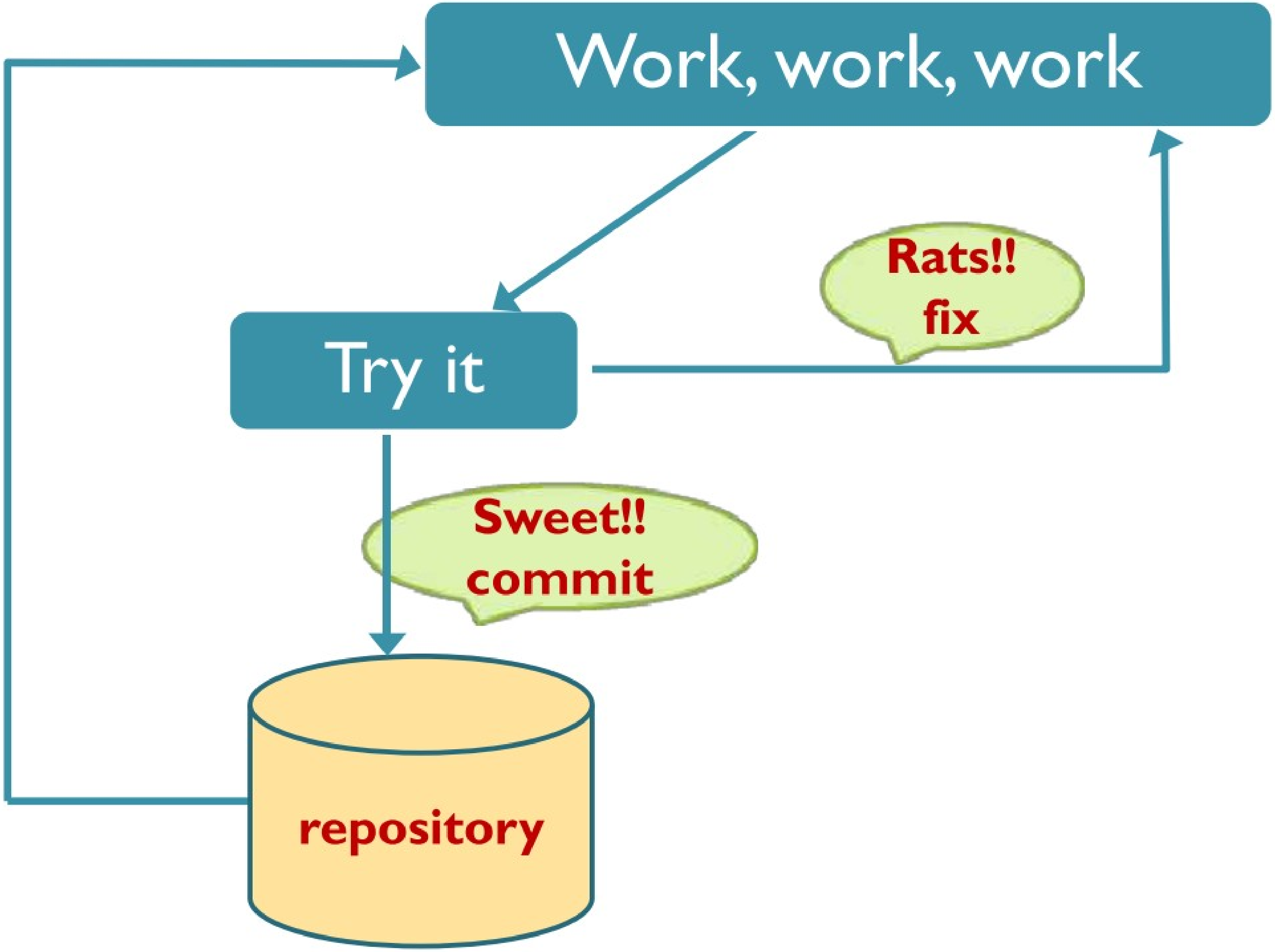
SCM Vocabulary

- Repository
  - This is the (central) copy of the source code with all the history and archive information
- Working copy
  - What Jane and Joe use to actually work on, each has their own one

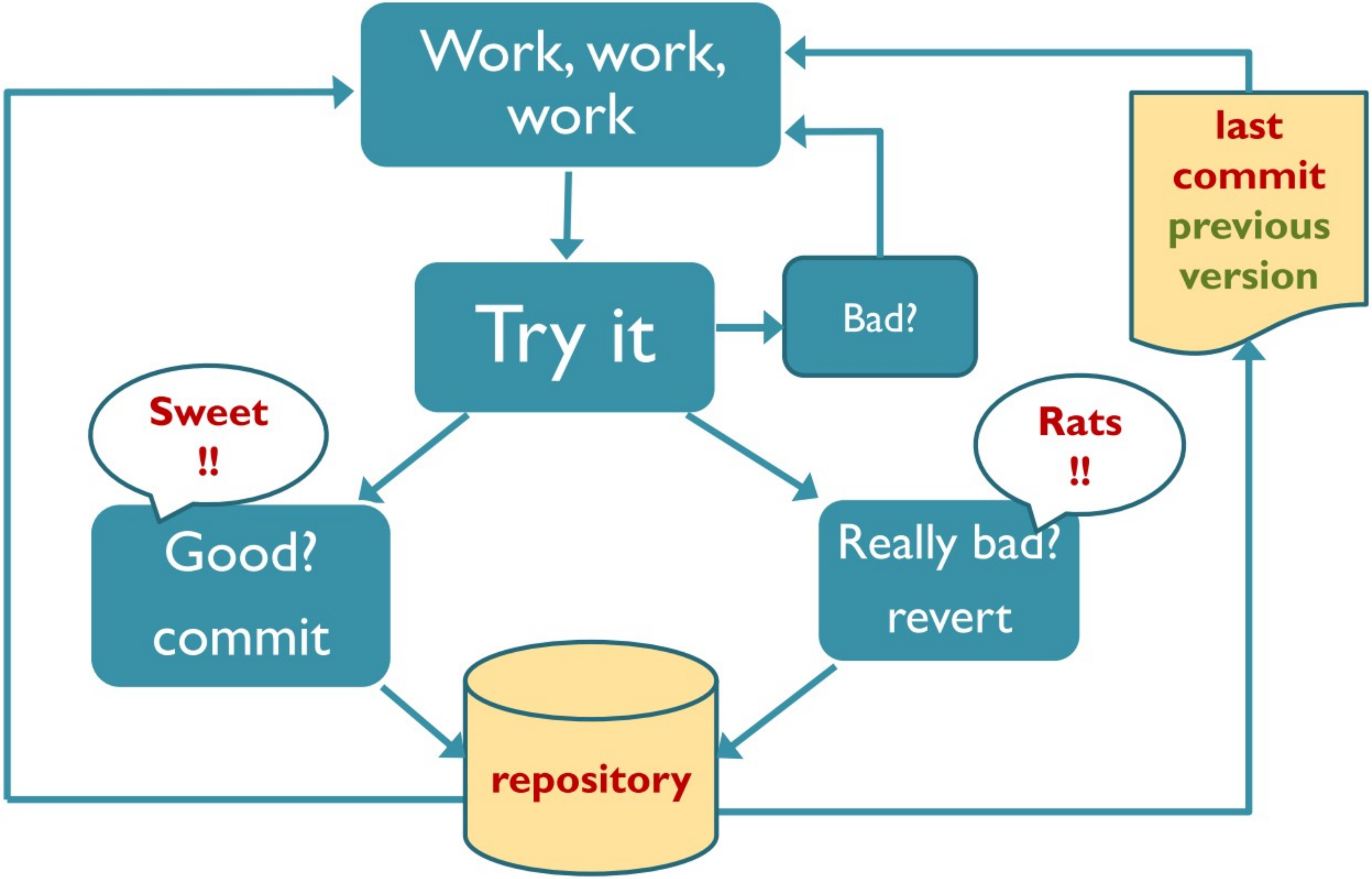
SCM Vocabulary

- **Checkout**
  - The process of fetching the repository content you need to your machine
- **Commit**
  - Write changes (save) to the repository
- **Update**
  - Get latest files (with committed changes) from the repository

SCM Work Capture



SCM Work Capture



SCM Work Cycle

- So typical SCM work cycle looks like this:
  - Update your working copy
    - gets others committed changes
  - Make your own changes
  - Examine your changes
  - Commit your changes
    - resolve edit conflicts on fail (occasionally)
  - -- and again …

Collaboration Issues

- Jane and Joe work on the same project
- They share the source code through a network drive
- Works as long as they do never edit the same file at the same time
- If one introduces a bug, the code for both is broken
- This is obviously a BAD THING!

Collaboration, Better

- Jane and Joe work on the same project
- They each use a private copy of the code, which allows editing the same files
- If one breaks the code, the other still can work on it
- Extreme care is needed when merging their changes back into one source
- This is obviously still a Not Great Thing

Collaboration, SCM-style

- Jane and Joe use some SCM software
- They can now both edit the same files – the SCM takes care of protecting their changes
- They still have their private copies and can work unencumbered even if one breaks it
- Merging they changes together is easier through the assistance of the SCM software

Change Tracking Issues

- Everyone has already done a change that broke something – and couldn’t remember exactly what it was
- If working together, it is crucial to know what the others on the team changed!
- This can all be achieved by creating backup and using tools like diff
- But is this the easy way?

Change Tracking via SCM

- **SCM software keeps track of changes**
- **You can always view the history of a file**
- **You can see who changed what**
- **You can even undo changes or get back a older version of a file**
- **Specific “points in time” can be marked for later reference**

SCM Model

- **Centralized**
  - **Concurrent Versions System /CVS**
    - **Oldie but goodie, 1986, orgiginally from Unix, central server manages projects on top of RCS (files only)**
  - **Subversion /SVN**
    - **2000, Apache Software Foundation, better CVS**
- **Distributed**
  - **Git**
    - **2005, Linus Torvalds (Linux), fast updates/merges**

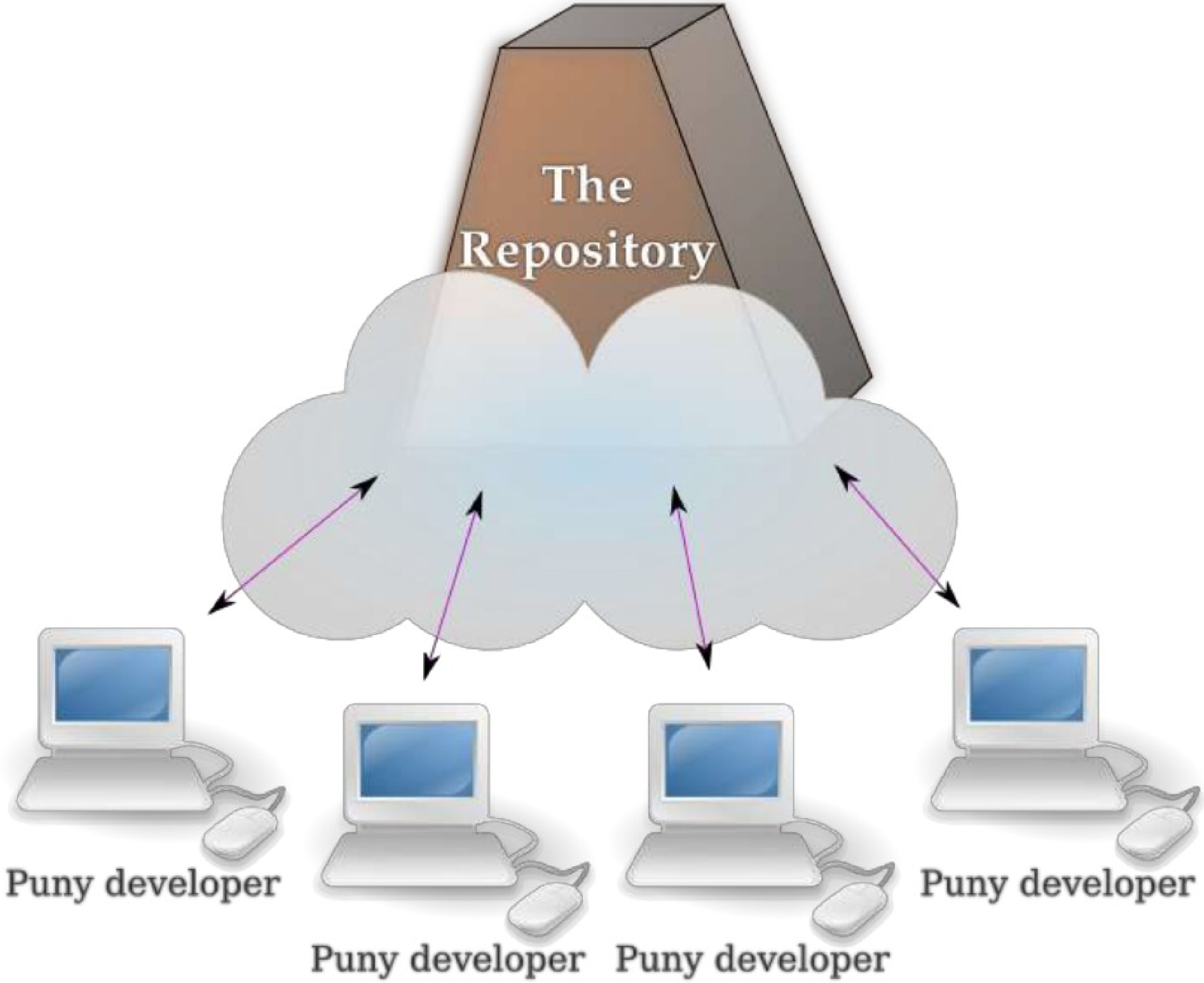
Centralized SCM: CVS

- Concurrent Versions System
- 1986, Dick Grune
- Extension of the Unix RCS
- Widely used, was Unix standard
- Handles collections of files as projects
  - RCS was individual files only

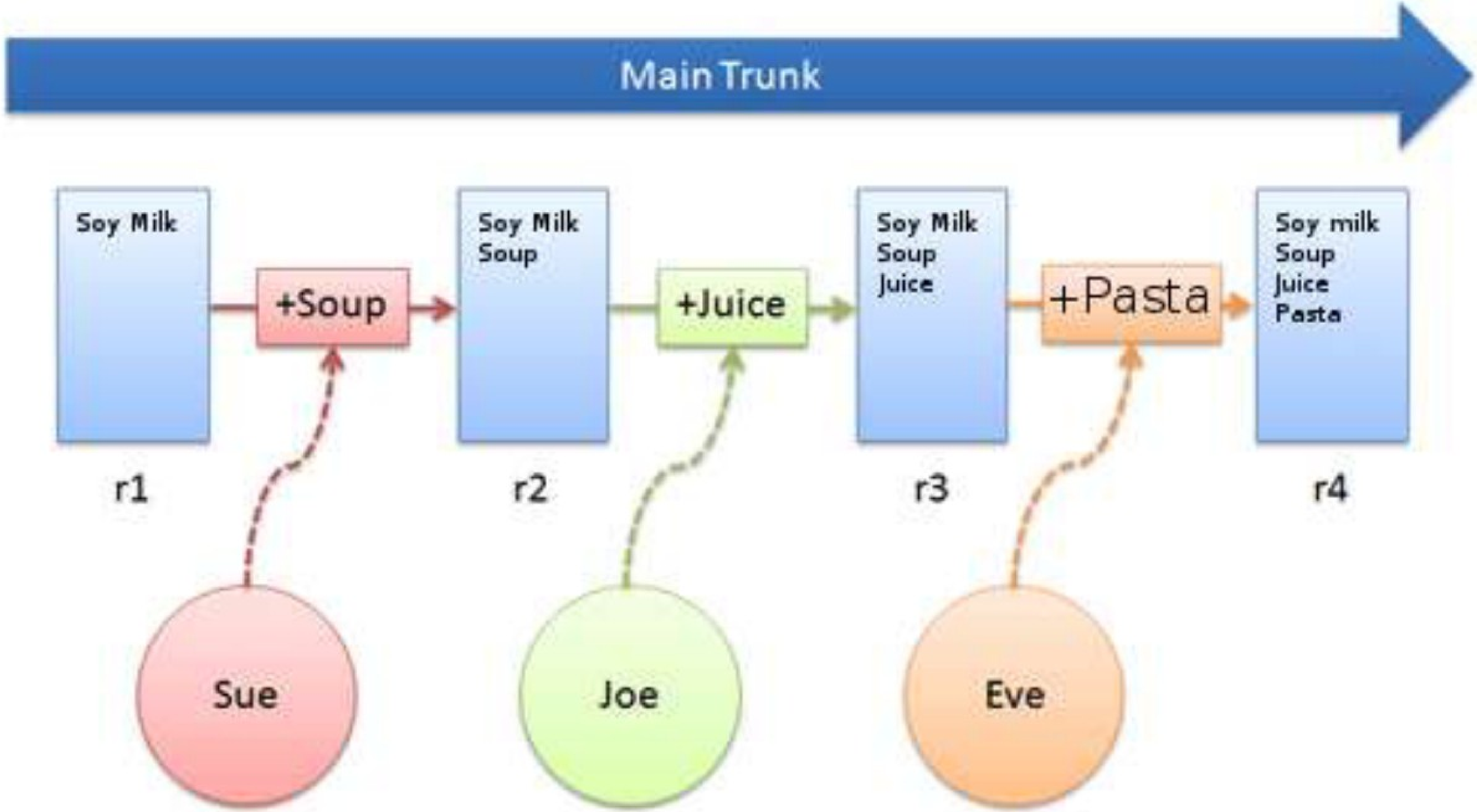
CVS: Centralized Model

- Client / server architecture
- Central server is THE code repository
- Developers run CVS clients, transfer files across network to local store for work
- Commit copies files back to central repository
- Detect conflicts for resolution

Centralized Server



Centralized Version Files



CVS: terminology

- CVS labels a single project (set of related files) that it manages as a module
- Server stores modules in its repository
- Check out gives a user a copy of a module: the files are the working copy, sandbox, or workspace
- Changes in the working copy are reflected in the repository by committing them
- To update is to acquire or merge the changes in the repository with the working copy.

CVS: update working copy

- Just do these:
  - `cd my/working/copy`
  - `cvs update`
- This will update your working copy to the current state of the repository
- Your local changes if any are preserved
- Conflicts may arise (more later)

CVS: make changes

- Edit your files to your liking
- If you add files, do:
  - `cvs add myfile.php`
- If you remove files, do:
  - `cvs remove myfile.php`

CVS: examine your changes

- There are three ways to check what you did:
  - `cvs status`
  - `cvs diff`
  - `cvs update`
- Using `cvs update` may not be very wise
  - It shows you changed files, but may pull in changes from the repository
- `cvs status` may produce a lot of output

CVS: examine change history

- To check what has been done to a file in the past, you can view the history
  - `cvs log`
- This is only really useful if ever change has a meaning ful commit message
- CVS can display the current revision and modification information for each line
  - `cvs annotate`

CVS: commit your changes

- If you are happy with your changes, share them with your fellow developers
- To do this, commit them with
  - `cvs –m ‘some descriptions’ commit`
- This will write your changes to the repository
- The description explains your change, so make use of it
- Commits may fail...

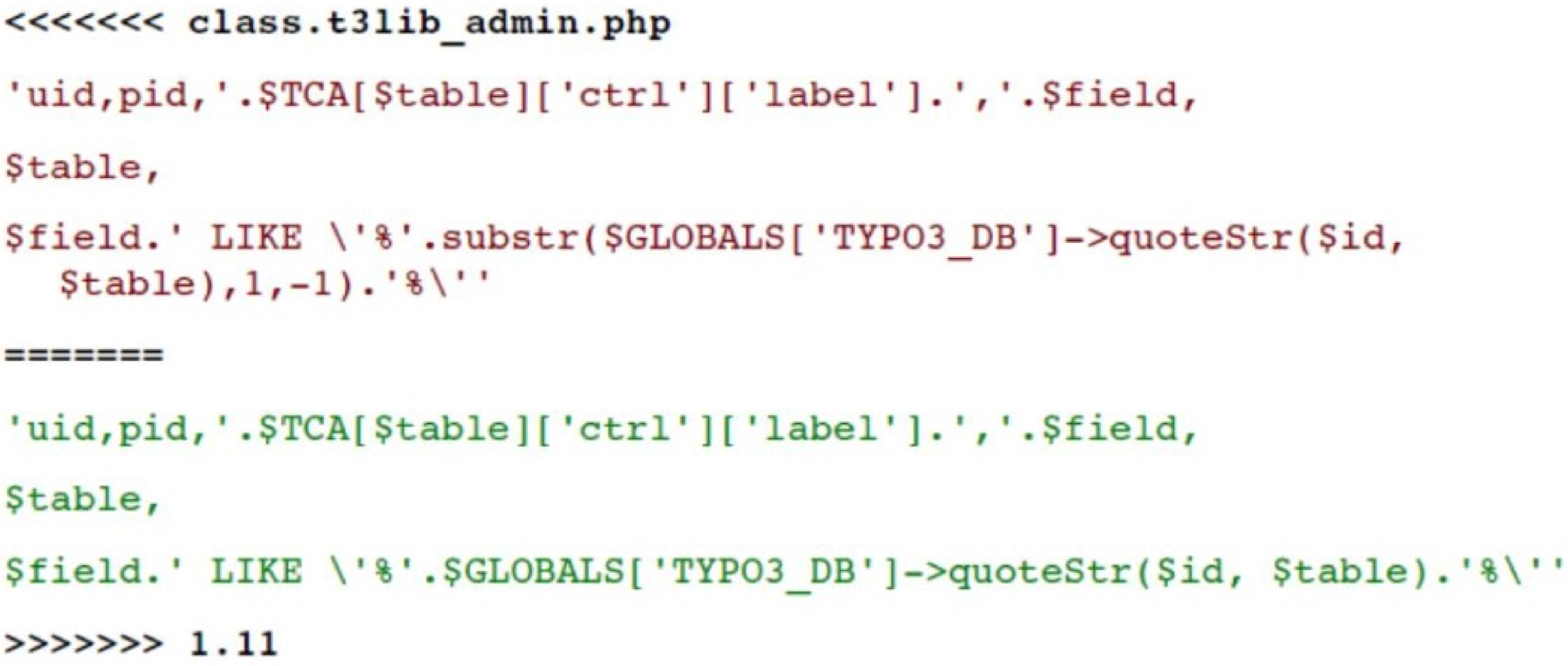
CVS: if a commit fails

- ... this may have one simple reason: your changed copy is not up to date
- This happens if someone committed another change to the file after you got your working copy of the file
- To prevent you from overwriting those changes, you can only commit after updating, thus merging the changes into your copy

CVS: conflicts

- **A conflict occurs if the remote changes and the local changes cannot be merged automatically**
- **Conflicts need to be resolved manually**
- **Usually this is easier than it sounds**

CVS: how conflicts look



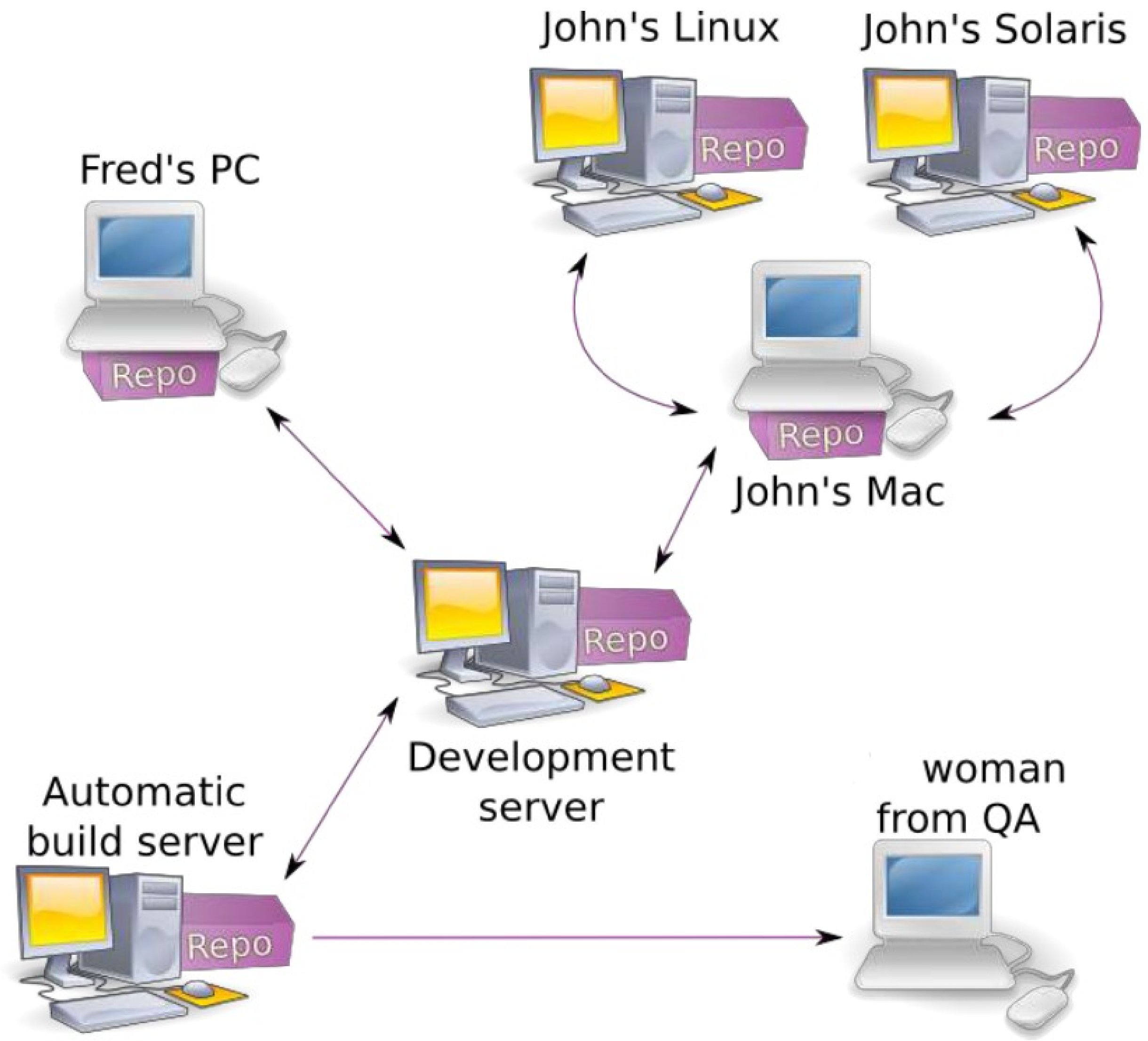
```
<<<<<<< class.t3lib_admin.php

'uid,pid,'.$TCA[$table]['ctrl']['label'].','.$field,

$table,

$field.' LIKE \'%'.substr($GLOBALS['TYPO3_DB']->quoteStr($id,
   $table),1,-1).'%\''

=======

'uid,pid,'.$TCA[$table]['ctrl']['label'].','.$field,

$table,

$field.' LIKE \'%'.$GLOBALS['TYPO3_DB']->quoteStr($id, $table).'%\''

>>>>>>> 1.11
```

UECS2363 SOFTWARE CONSTRUCTION AND CONFIGURATION
CHAPTER 3 : SCM (PART II)

Distributed SCM

- Becoming the de facto way
- Better supports large open source efforts
- No central server/repository
- All developers have all code locally
- Consistency is maintained via network transfers
- Example: Git (Github)

Distributed Servers



Git: A Conceptual Overview

- Data Model: the Repository
- Operations to manipulate the repository
  - Adding files and data
  - Branching
  - Merging
- How to use git to collaborate
- Rebase: an alternative to merging

Data Model: the Repository

- Repository contains
  - Set of commit objects
  - Set of heads – references to commit objects
- Repository stored in same directory as the project itself, in a subdirectory named .git
- .git is at the project root, and there is one .git for the whole project
- Repos. stored in files along with project files
- No central server

Data Model: Commit Object

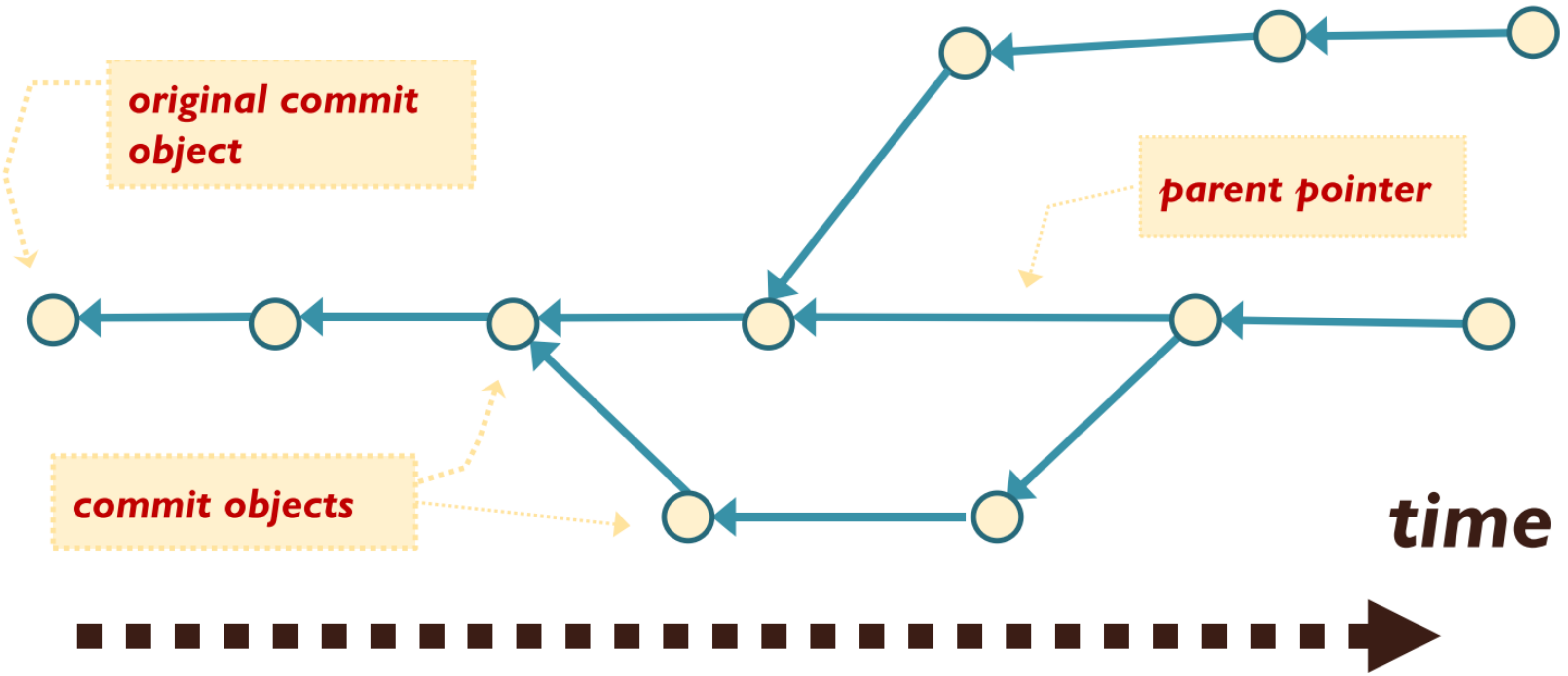
- Commit Object contains 3 things
  - set of files, reflecting the state of a project at a given point in time.
  - parent commit objects (references to them)
  - SHA1 name, a 40-character hash string that uniquely identifies the commit object.

Data Model: Commit Object

- **Parent commit objs are those commits that were edited to make the subsequent project state**
  - **Generally a commit object will have one parent commit (one generally takes a project in a given state, makes a few changes, and saves the new project state)**

History structure... think DAG

- There is always one commit object with no parent (the original one used to create the project)
- Parent pointers point from commit object back in time to previous commit object, back to the original



HEAD and head

- Pointer to a commit object is called a head
- Every head has a name
- Default: every repository has one head named master
- A repository can have any number of heads
- At any given time, one head is selected as the current head
- HEAD is as alias that always refers to the current head

Make a Project

- Let's call the project '*myBigIdea*'
- Create a directory called '*myBigIdea*' (or use one that already exists… it need not be empty)
  - `mkdir myBigIdea`
  - `cd myBigIdea`
  - `git init`
- This makes the .git subdirectory in the directory *myBigIdea*

To create a commit …

- Tell Git which files to include in the commit, using git add
- If a file has not changed since the previous commit (the "parent" commit), git will include it automatically in the commit object you are constructing
- Thus, you only need to add files that you have created or modified

To create a commit …

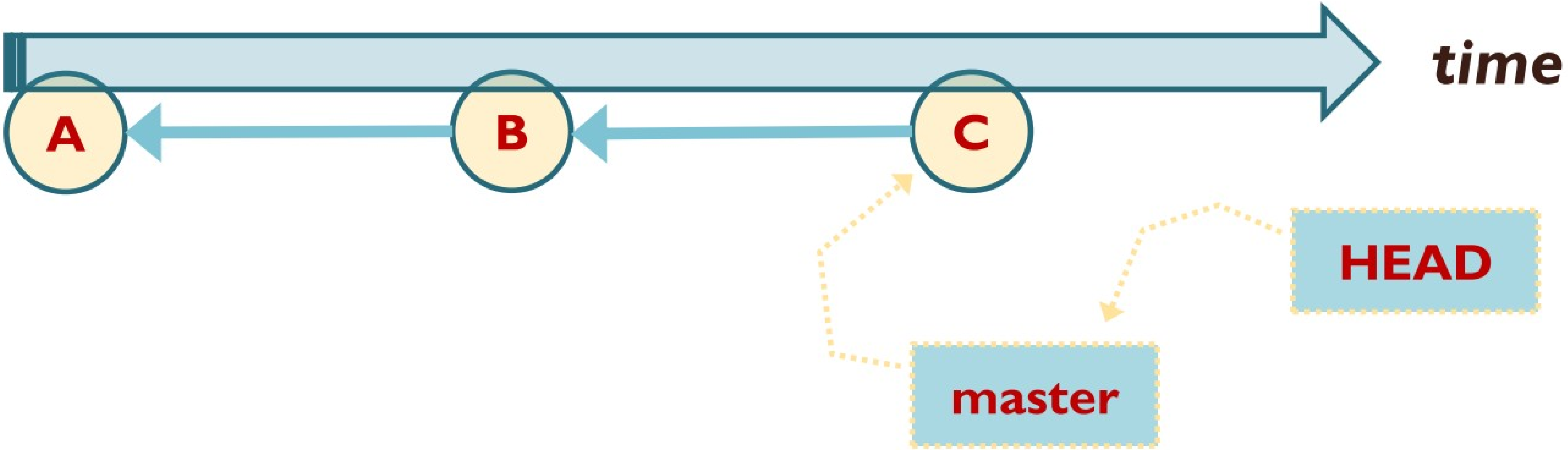
- Note that add will act recursively down into directories, so the command
  - `git add .`
- will add everything that has changed
- Call git commit to create the commit object
- The new commit object will have the current HEAD as its parent
- After the commit is complete, HEAD will point to the new commit object

Shortcut

- Here is a shortcut
  - `git commit -a`
- Automatically does an add on all modified files
- Does not include newly created ones

Work, work, work …

- **Let’s say you do 3 commits as just described**
- **Project repository history looks like this:**



A is the original commit

B is parent of ***C***

A is parent of ***B***

More Commands

- `git log`
- shows a log of all commits starting from HEAD back to the initial commit (can do more too)

More Commands

- `git status`
- shows which files have changed between the current project state and HEAD files are put in one of three categories:
  - new files that haven’t been added (with git add)
  - modified files that haven’t been added
  - files that have been added

More Commands

- `git diff`
- shows the diff between HEAD and the current project state. With the --cached option it compares added files against HEAD; otherwise it compares files not yet added
- `git rm` and `git mv`
- mark files to be removed and moved (renamed), respectively, much like `git add`

Common workflow

- Do some programming
- `git status` to see what files you changed
- `git diff` [file] to see exactly what you modified
- `git commit -a –m` “some log message” to make a new commit object

Referring to a Commit

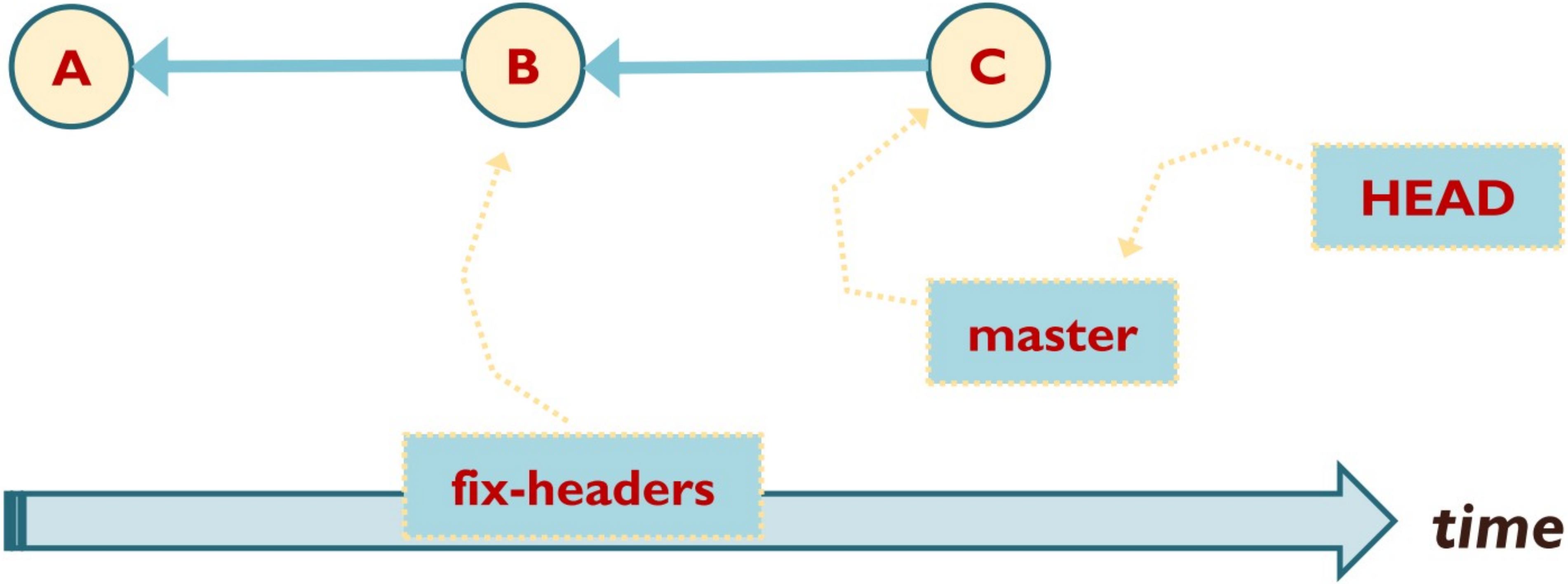
- **Now that we have made some commits, how can we refer to a specific commit object ?**
  - **By its SHA1 hash (shown in the log)**
  - **By the first few chars of the SHA1 hash**
  - **By a head, such as HEAD or master**

Referring to a Commit

- Relative to another commit
  - A caret (^) after a commit name refers to its parent.
  - HEAD^ denotes the parent of the current head
  - master^ refers to the second most recent commit

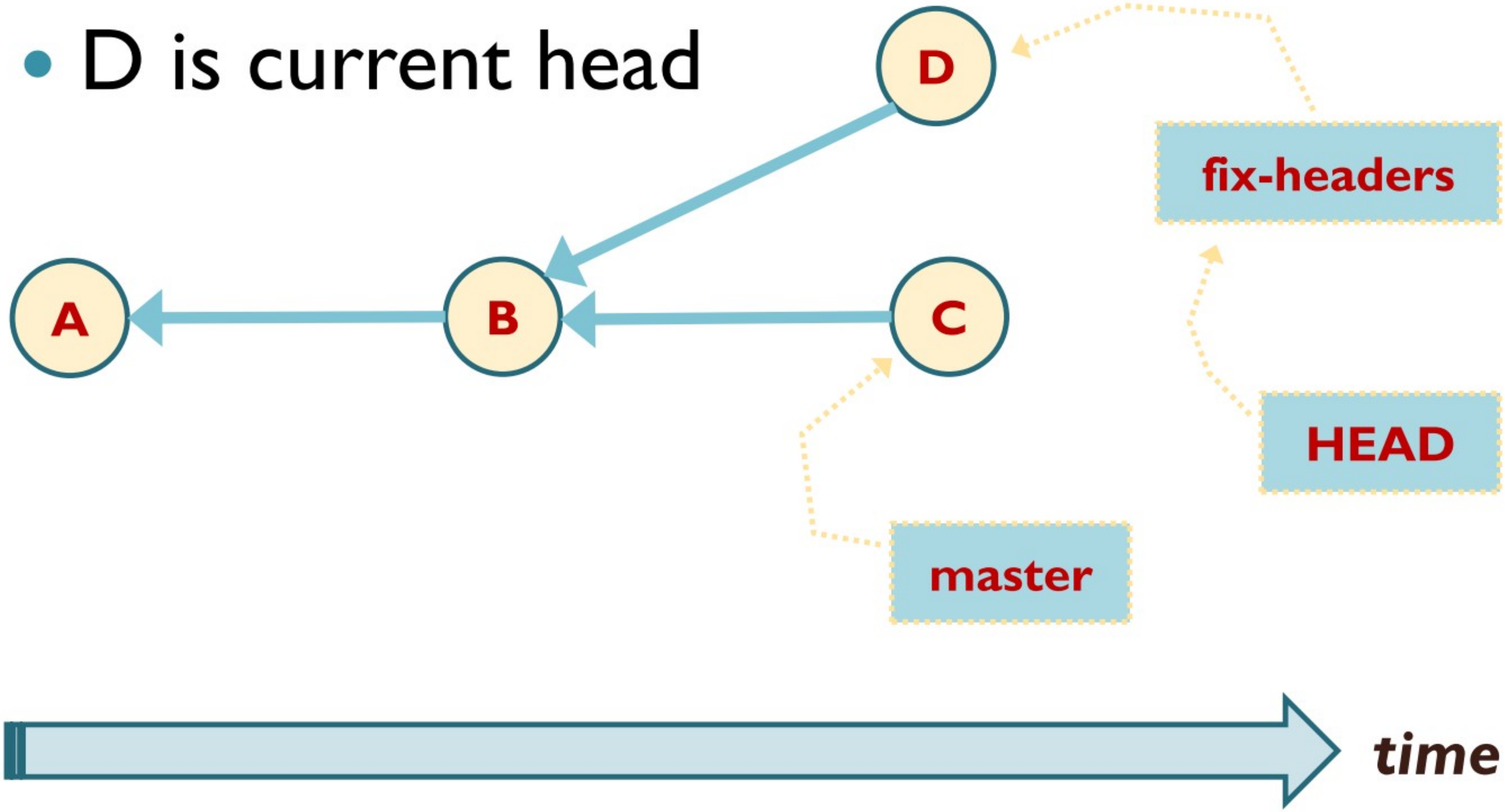
Making a Branch

- **After**
- `git branch fix-headers HEAD^`



Making a Branch

- **After**
- `git commit`



- D is current head

END OF LECTURE 04