

# Configuration Management

## Object Oriented Programming

<https://softeng.polito.it/courses/09CBI>



**SoftEng**  
<http://softeng.polito.it>

Version 1.3.1 - May 2021

© Marco Torchiano, 2021



This work is licensed under the Creative Commons Attribution–NonCommercial–NoDerivatives 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-nd/4.0/>.

You are free: to copy, distribute, display, and perform the work

Under the following conditions:



**Attribution.** You must attribute the work in the manner specified by the author or licensor.



**Non-commercial.** You may not use this work for commercial purposes.



**No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

# Learning objectives

---

- Understand what is configuration management
  - ♦ What is Version Control
  - ♦ What are the main concepts of VC
- Know the main tools for version control
- Learn how SVN can be used for CM

---

3

## Configuration Management

---

- The discipline that applies technical and administrative direction and surveillance in order to:
  - ♦ identify and document the functional and physical characteristics of a configuration item,
  - ♦ control changes to those characteristics,
  - ♦ record and report change processing and implementation status, and
  - ♦ verify compliance with specified requirements

[IEEE Std 828–2012]

---

4

# Issues

---

- What is the history of a document?
    - ♦ Versioning
  - Who can change what and how?
    - ♦ Change control
  - What is the correct set of documents for a specific need?
    - ♦ Configuration
  - How is the final system obtained?
    - ♦ Build management
- 

5

## Goals of CM

---

- Identify and manage parts of software
  - Control access and changes to parts
  - Allow to rebuild previous version of software
- 

6

---

# VERSIONING

---

7

## Versioning

---



Thesis.docx



ThesisFinal.docx



ThesisFinal  
Final.docx



ThesisFinalest  
Final.docx



ThesisFinalest  
FinalForsure.docx



ThesisFinalestF\*\*k  
FinalForsure.docx

---

8

# Terms

---

- Configuration item (CI)
- Configuration Management aggregate
- Configuration
- Version
- Baseline

---

9

## Configuration Item (CI)

---

- *Aggregation of work products that is treated as a single entity in the configuration management process*
- CI (typically a file):
  - ♦ Has a name
  - ♦ All its versions are numbered and kept
  - ♦ User decides to change version number with specific operation (commit)
  - ♦ It is possible to retrieve any previous version

---

10

# Version

---

- The initial release or a re-release of a configuration item
- Instance of CI, e.g.
  - ♦ Req document 1.0
  - ♦ Req document 1.1

---

11

## Version identification

---

- Procedures for version identification should define an unambiguous way of identifying component versions
- Basic techniques for component identification
  - ♦ Version numbering
  - ♦ Attribute-based identification

---

12

# Version numbering

---

- Simple naming scheme uses a linear derivation  
e.g. V1, V1.1, V1.2, V2.1, V2.2 etc.
- Actual derivation structure is a tree or a network rather than a sequence
- Names are not meaningful.
- Hierarchical naming scheme may be better

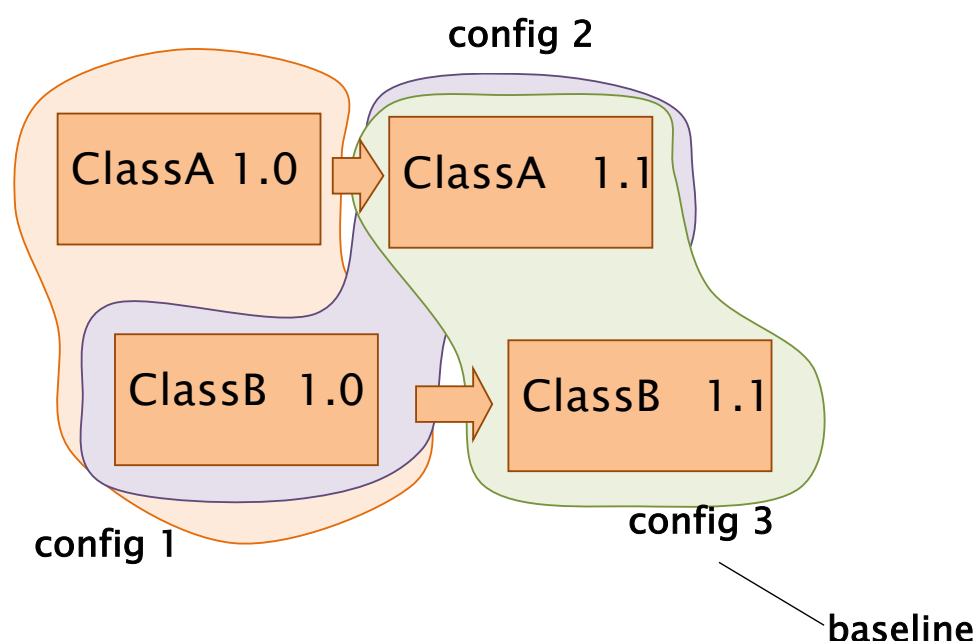
---

13

# Configuration

---

- Set of CIs, each in a specific version



---

14

# Configuration

---

- Snapshot of software at certain time
  - ♦ Various CIs, each in a specific version
  - ♦ Same CI may appear in different configurations
  - ♦ Configuration has version too

---

15

## Baseline

---

- Configuration in stable, frozen form
  - ♦ Not all configurations are baselines
  - ♦ Any further change / development will produce new version(s) of CI(s), will not modify baseline
- Types of baselines
  - ♦ Development – for internal use
  - ♦ Product – for delivery

---

16



# Semantic Versioning

---

- Product numbering based on  
*MAJOR.MINOR.PATCH*
- Increment:
  - ♦ MAJOR: when you make large (possibly incompatible) API changes,
  - ♦ MINOR: when you add functionality in a backwards-compatible manner, and
  - ♦ PATCH: when you make backwards-compatible bug fixes.

<http://semver.org>

---

17

---

## CHANGE CONTROL

---

18

# Repository

---

- A collection of all software–related artifacts belonging to a system
- The location/format in which such a collection is stored

---

19

## Typical case

---

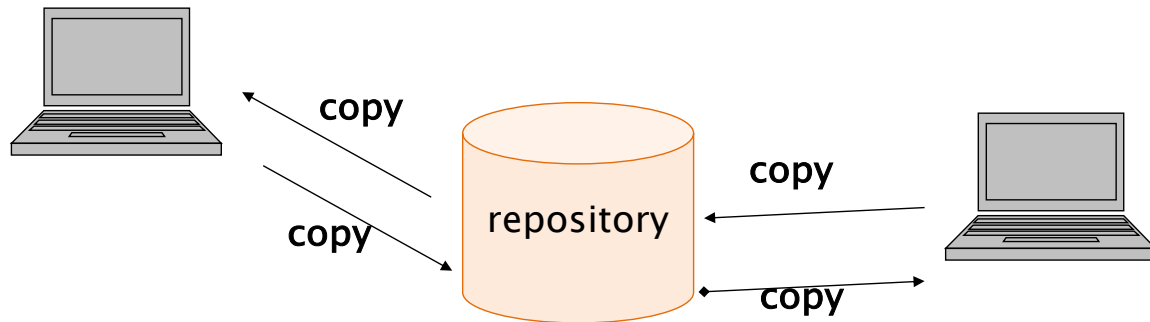
- Team develops software
- Many people need to access different parts of software
  - ♦ Common repository (shared folder),
  - ♦ Everybody can read/write documents/files

---

20

# Change control – repository

---

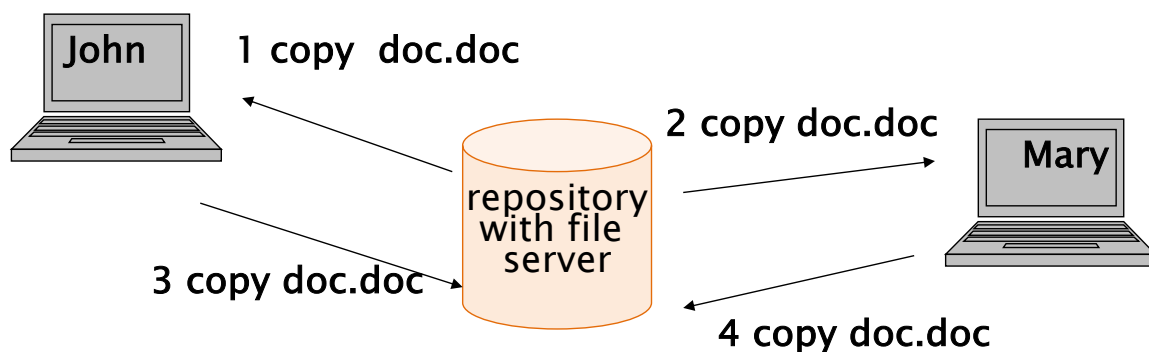


---

21

## Repository – file server

---



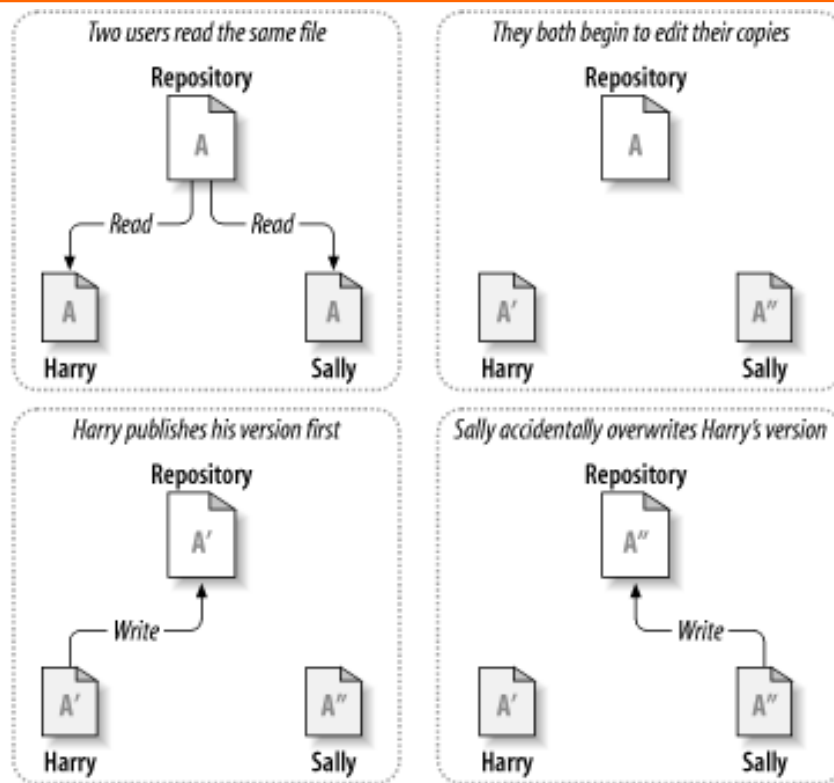
Changes by John are lost

---

22

# File system

---



© B.Collins-Sussman, B.W.Fitzpatrick C.M.Pilato

23

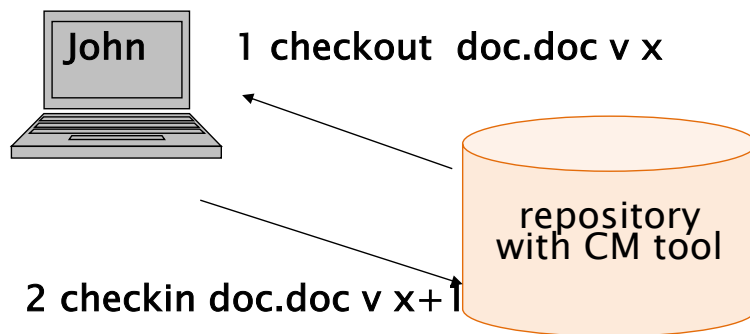
## Check-in / check-out

---

- Check-out
  - ♦ Extraction of CI from repository
    - with goal of either changing it or not
- Check-in (or commit)
  - ♦ Insertion of CI under control

# Repository – check in checkout

---



---

25

## Check-in / check-out – scenarios

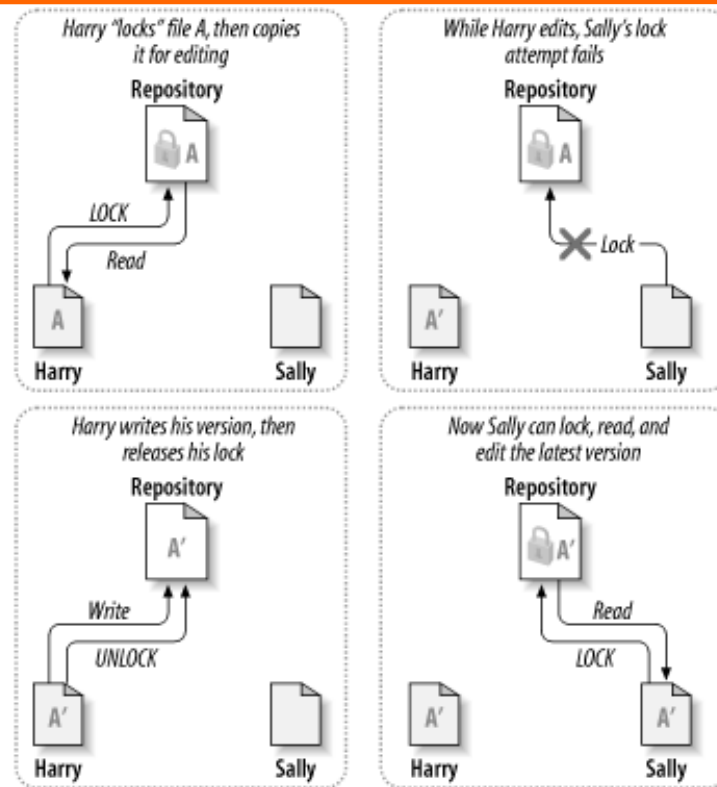
---

- Lock–modify–unlock (or serialization)
  - ♦ Only one developer can change at a time
- Copy–modify–merge
  - ♦ Many change in parallel, then merge

---

26

# Lock-Modify-Unlock



© B.Collins-Sussman, B.W.Fitzpatrick C.M.Pilato

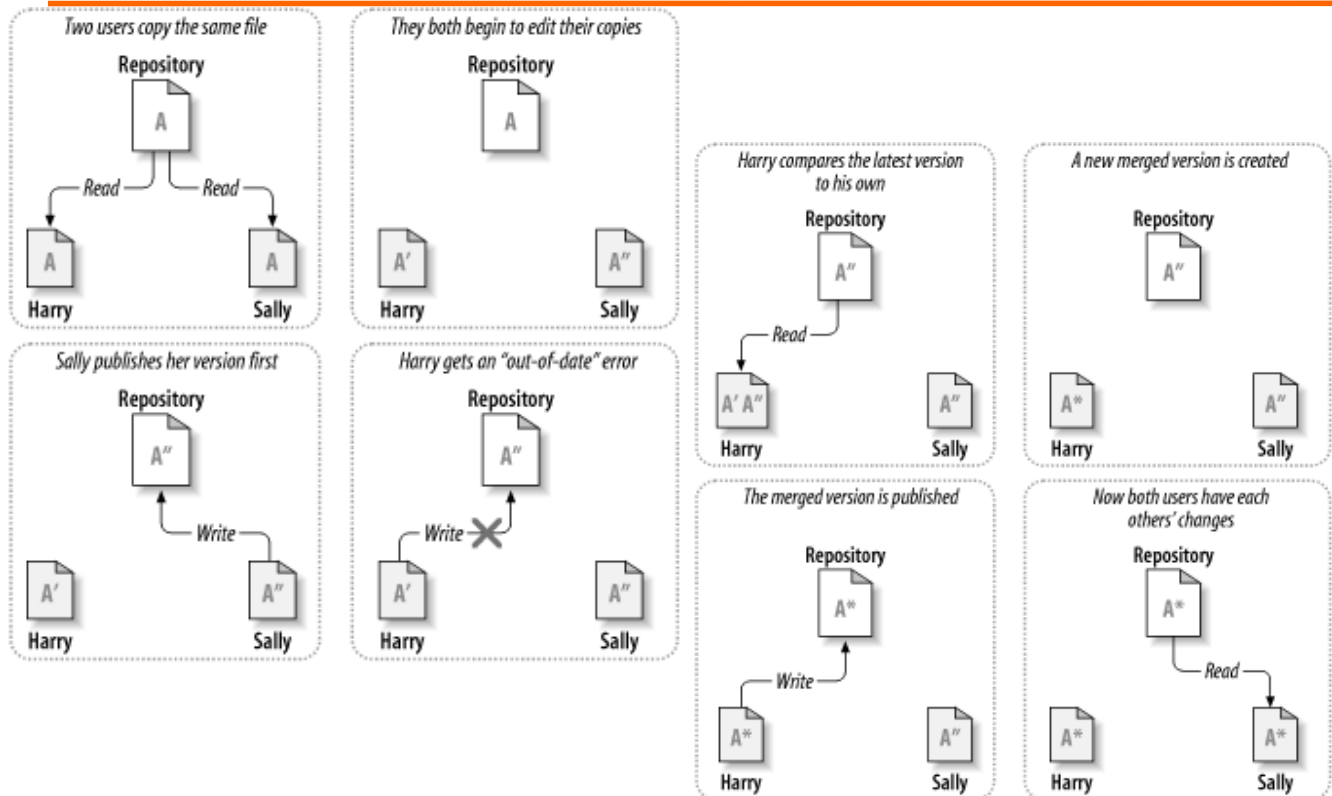
27

# Lock-Modify-Unlock

- Pro
  - ◆ Conflicts are impossible
- Cons
  - ◆ No parallel work is possible, large delays can be induced
  - ◆ Developers can possibly forget to unlock so blocking the whole team

28

# Copy-Modify-Merge



© B.Collins-Sussman, B.W.Fitzpatrick C.M.Pilato

29

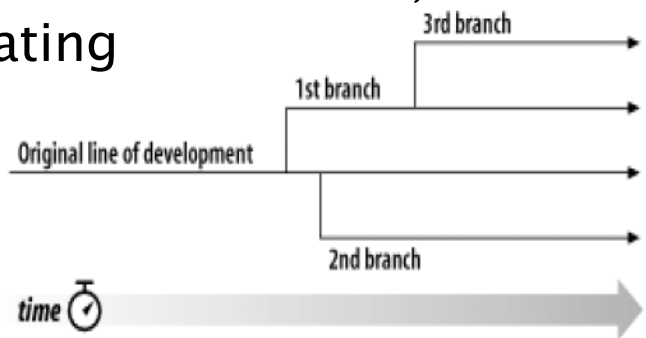
# Copy-Modify-Merge

- Pros
  - ◆ More flexible
  - ◆ Several developers can work in parallel
  - ◆ No developer can block others
- Con
  - ◆ Requires care to resolve the conflicts

# Branches: general concept

---

- Line of development that exists independently of another line, yet still shares a common history when looking far enough back in time.
- A branch always takes life as a copy of something, and moves on from there, independently generating its own history



31

## Branches: motivation

---

- Branches allow working in isolation from the main branch
  - ♦ Several new features or fixes can be developed independently and concurrently
  - ♦ When work is complete it can be merged into the main branch
- Branches may also represent different configurations, e.g. by platform

32



# Tools

---

## ■ Change Control+Versioning+Configuration

- ◆ RCS
- ◆ CVS
- ◆ SCCS
- ◆ PCVS
- ◆ Subversion
- ◆ BitKeeper
- ◆ Mercurial
- ◆ Git



---

33

## BUILD MANAGEMENT

---

34

# Build management

---

- Prepare the environment
- Gather third party components
- Gather source code
- Compile
- Create packages
- Run tests
- Deploy

---

35

## Tools

---

- Build management

- ♦ Make
- ♦ Ant
- ♦ Maven
- ♦ Gradle



**maven**



---

36

# Continuous Integration

---

- Maintain a single source repository
- Automate the build
- Make your build self-testing
- Any commit build on integration machine
  - ♦ Keep the build fast
- Test in a clone of the production environment
- Automate deployment

---

37

# Continuous integration

---

- Commit frequently
- Don't commit broken code
- Don't commit untested code
- Don't commit when the build is broken
- Don't go home after committing until the system builds

---

38

# Tool CI

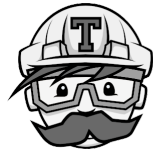
---

- Continuous Integration

- ♦ Travis CI

- ♦ Jenkins

- ♦ Cruise Control



**Travis CI**



**Jenkins**



---

39



## VERSION CONTROL WITH SUBVERSION

---

40

# What is Subversion

---

- Open-source version control system:
  - ♦ it manages any collection of files and directories over time in a central repository;
  - ♦ it remembers every change ever made to your files and directories;
  - ♦ it can access its repository across networks

---

41

## Features

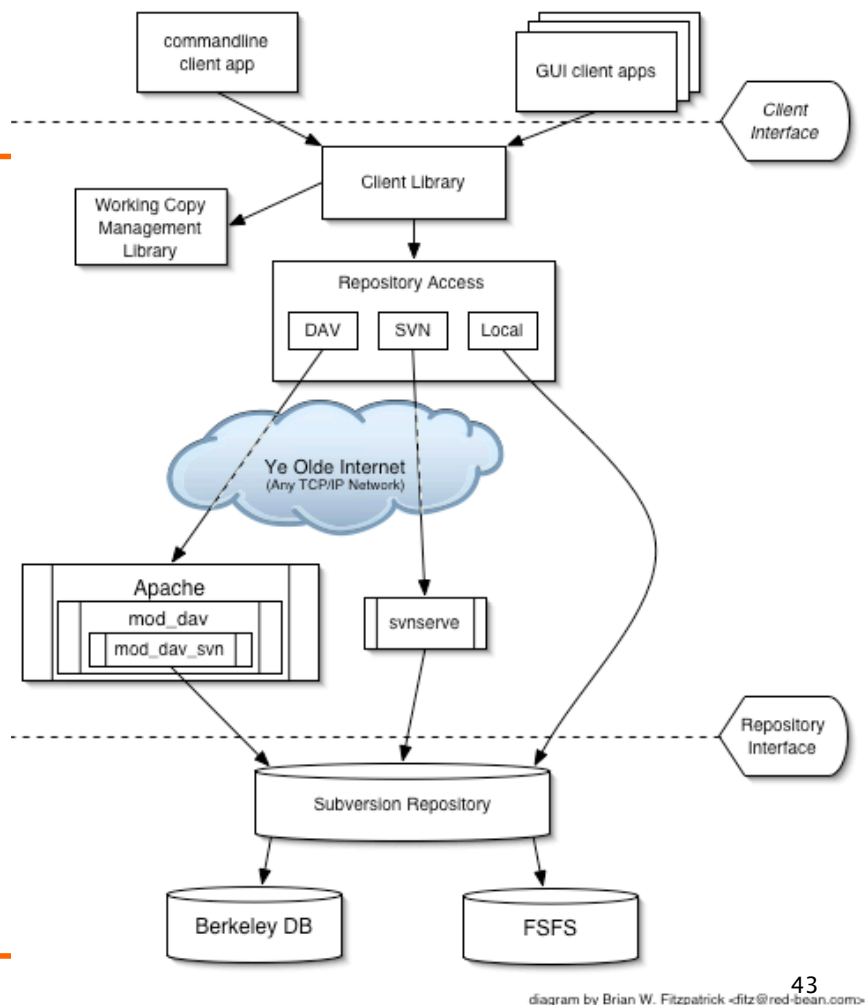
---

- Directory versioning and true version history
- Atomic commits
- Metadata versioning
- Several topologies of network access
- Consistent data handling
- Branching and tagging
- Usable by other applications and languages

---

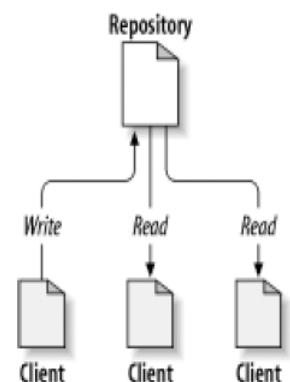
42

# Architecture



## The SVN repository

- Central store of data
- It stores information in the form of a file system
- Any number of clients connect to the repository, and then
  - ♦ read (**update**) or
  - ♦ write (**commit**) to these files.



# The working copy (WC)

---

- Ordinary directory tree on your local system, containing a copy of the repository files (**checkout**)
- Subversion will never incorporate other people's changes (**update**), nor make your own changes available to others (**commit**), until you explicitly tell it to do so.

---

45

## Revisions

---

- Each time the repository accepts a commit, it creates a new state of the file system tree, called a revision.
- Global revision numbers: each revision is assigned a progressive unique natural number (previous revision + 1)
  - ♦ A freshly created repository has revision 0 (zero)
- All files in the repo get a new revision number
  - ♦ Revision  $N$  represents the state of the repository after the  $N$ th commit.

---

46

# Base and head revisions

---

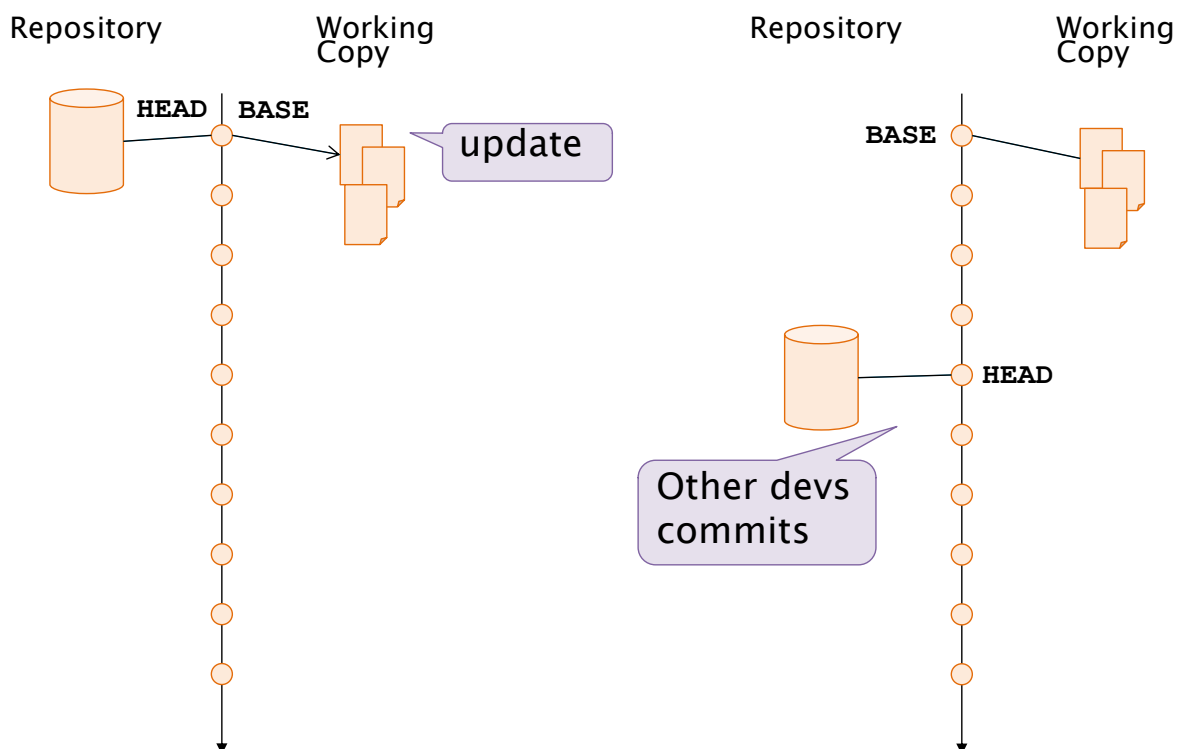
- **HEAD** is the latest revision in the repo
- **BASE** is the latest revision in WC
- When an **update** is performed on a WC
  - ♦  $\text{BASE} = \text{HEAD}$
  - ♦ Contents of files in WC are updated to those in repo

---

47

# Head and base revisions

---

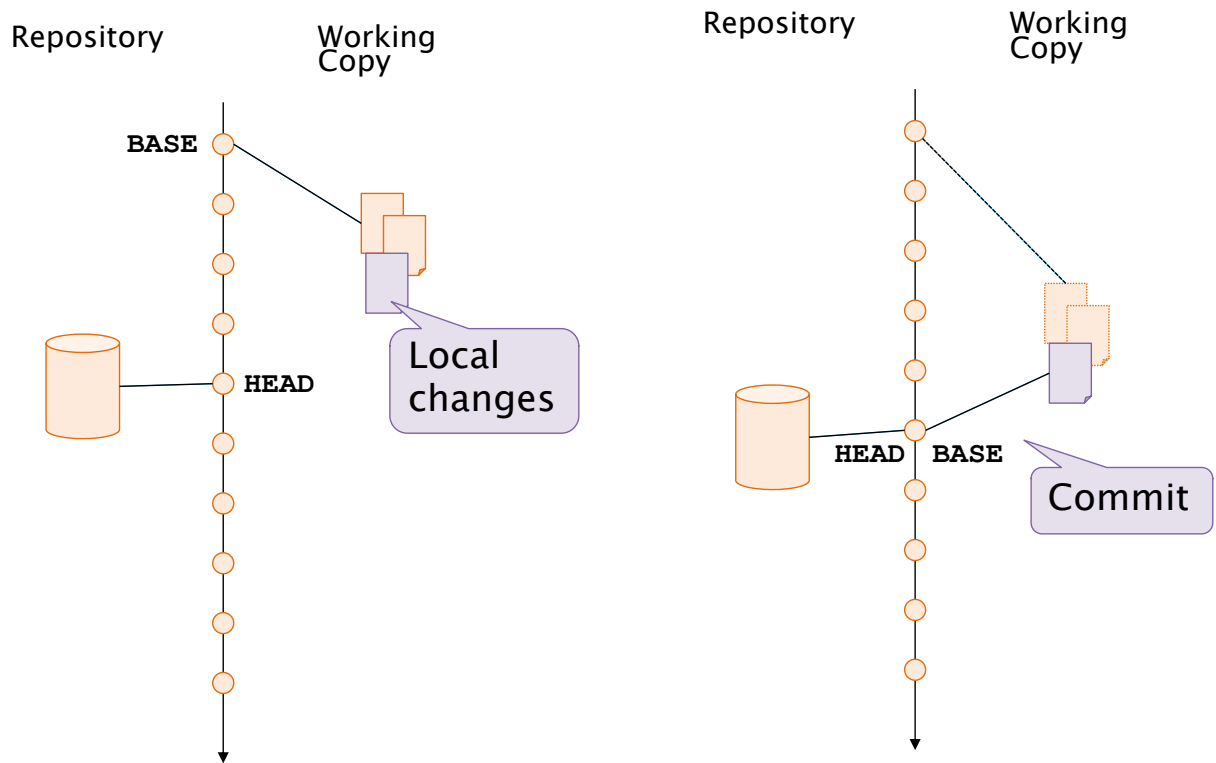


---

48

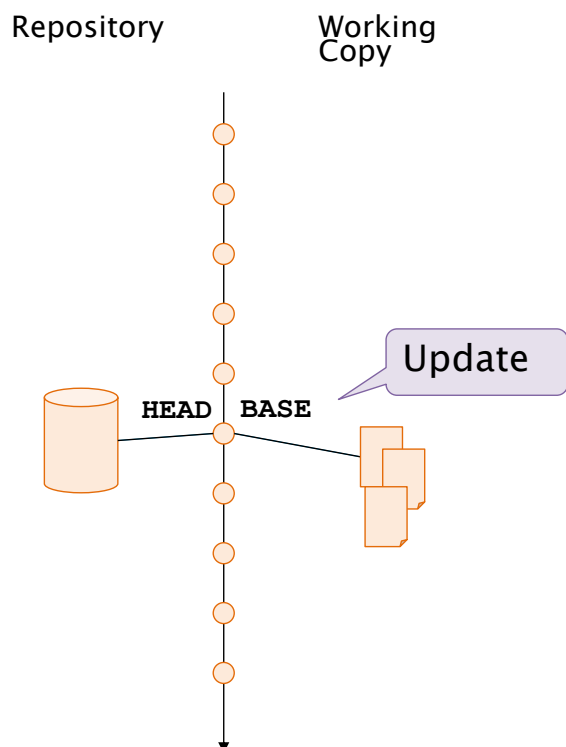


# Mixed revisions



49

# Mixed revisions



50

# Mixed revisions

---

- Suppose you have a working copy entirely at revision 10. You edit the file `foo.html` and then perform an `svn commit`, which creates revision 15 in the repository.
- Therefore the only safe thing the Subversion client can do is mark the one file—`foo.html`—as being at revision 15. The rest of the working copy remains at revision 10. This is a mixed revision.
- Only by running `svn update` can the latest changes be downloaded, and the whole working copy be marked as revision 15.
- Memento:
  - ♦ Every time you run `svn commit`, your working copy ends up with some mixture of revisions: the things you just committed are marked as having larger working revisions than everything else.

---

51

## Basic Work Procedure

---

- Create working copy from a repository
  - ♦ `svn checkout <repository>`*When ready...*
- Synchronize contents of WC with repo
  - ♦ `svn update`
- Work locally on WC files
- Possibly put new files under control
  - ♦ `svn add <file list>`
- Push work to repository
  - ♦ `svn commit -m "<Log message>"`

---

52

# Commit Log Message

---

- Structure of the message

`<type>(<scope>) : <subject>`

`<body>`

`<footer>`

- Example

`fix(middleware): ensure Range headers  
adhere more closely to RFC 2616`

`Added one new dependency, use `range-  
parser` (Express dependency) to compute  
range. It is more well-tested in the  
wild.`

`Fixes #2310`

<http://karma-runner.github.io/1.0/dev/git-commit-msg.html>

---

53

## Conflicts

---

- A conflict arise, upon commit, when
  - ♦ A file was changed locally
  - ♦ Meanwhile someone else changed it, and
  - ♦ She committed the change to the repo
- A conflict occurs if:
  - ♦ HEAD > BASE and
  - ♦ Contents of revisions HEAD and BASE of the file being committed differ

---

54

# Conflicts

---

- After a conflict is detected an update must be performed, then Subversion add three extra files to WC:
  - ♦ `filename.mine` : the local file as it existed in the working copy before the update
  - ♦ `filename.rOLDREV` : the BASE revision before the update.
  - ♦ `filename.rNEWREV` : the HEAD revision of file
- The original file is changed to a mix version of HEAD (`.rNEW`) and local (`.mine`) with change markers

---

55

## Conflict example

---

- **You** and **Sally** both edit file `sandwich.txt` at the same time. Sally commits her changes, and when you go to update your working copy, you get a conflict

```
$ svn update
```

```
Conflict discovered in 'sandwich.txt'.
```

```
Select: (p)postpone, (df)diff-full, (e)edit,  
        (h)elp for more options : p
```

```
C  sandwich.txt
```

```
Updated to revision 2.
```

---

56

# Conflict example

---

- In your working copy you get

```
$ ls
```

```
sandwich.txt
```

```
sandwich.txt.mine
```

```
sandwich.txt.r1
```

```
sandwich.txt.r2
```

- You're going to have to edit `sandwich.txt` to resolve the conflicts

---

57

# Conflict example

---

- The contents of the file `sandwich.txt` is

```
Top piece of bread
```

```
Mayonnaise
```

```
Lettuce
```

```
<<<<<< .mine
```

```
Salami
```

```
Mortadella
```

```
Prosciutto
```

```
=====
```

```
Sauerkraut
```

```
Grilled Chicken
```

```
>>>>>> .r2
```

```
Creole Mustard
```

```
Bottom piece of bread
```

Changes your made in  
the conflicting area

Changes Sally previously  
committed in the area

---

58

# Conflict example

---

- The updated file `sandwich.txt` you create and saved is

Top piece of bread

Mayonnaise

Lettuce


Mortadella

Prosciutto

Grilled Chicken

Creole Mustard

Bottom piece of bread



Pick and choose  
"by hand"

---

59

# Conflict example

---

- Once the conflict has been composed you ought to signal it has been resolved

```
$ svn resolve --accept working sandwich.txt
```

```
Resolved conflicted state of 'sandwich.txt'
```

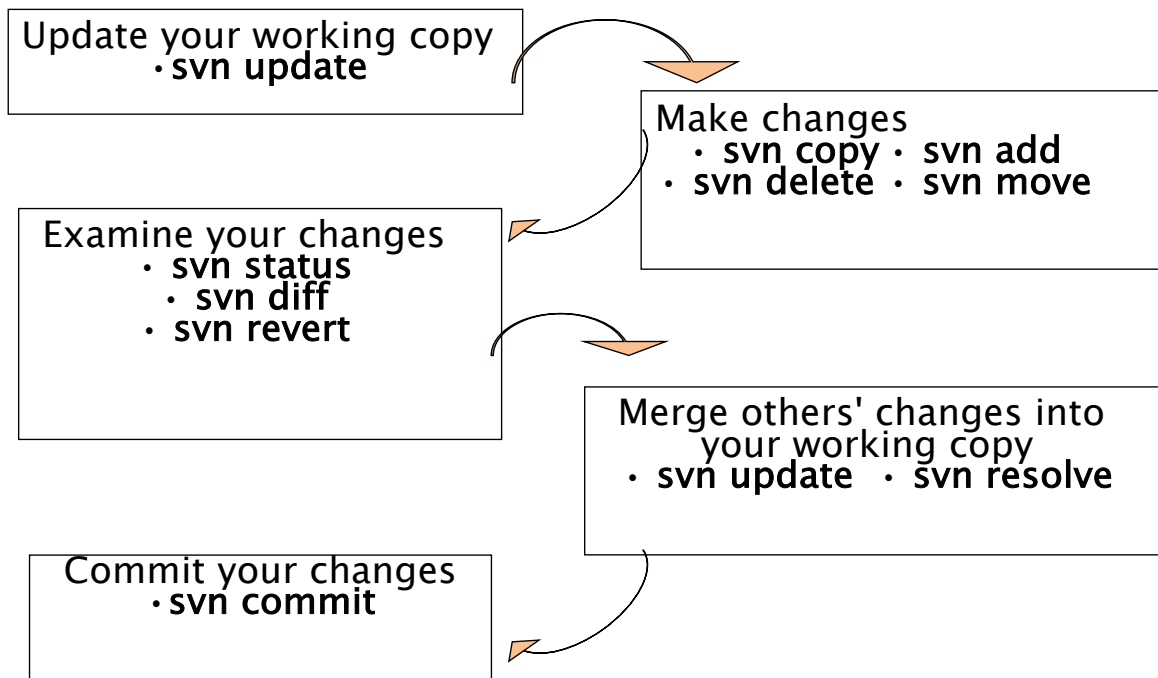
```
$ svn commit -m "Picked and chosen."
```

---

60

# Typical work cycle

---



---

61

## Branches in Subversion

---

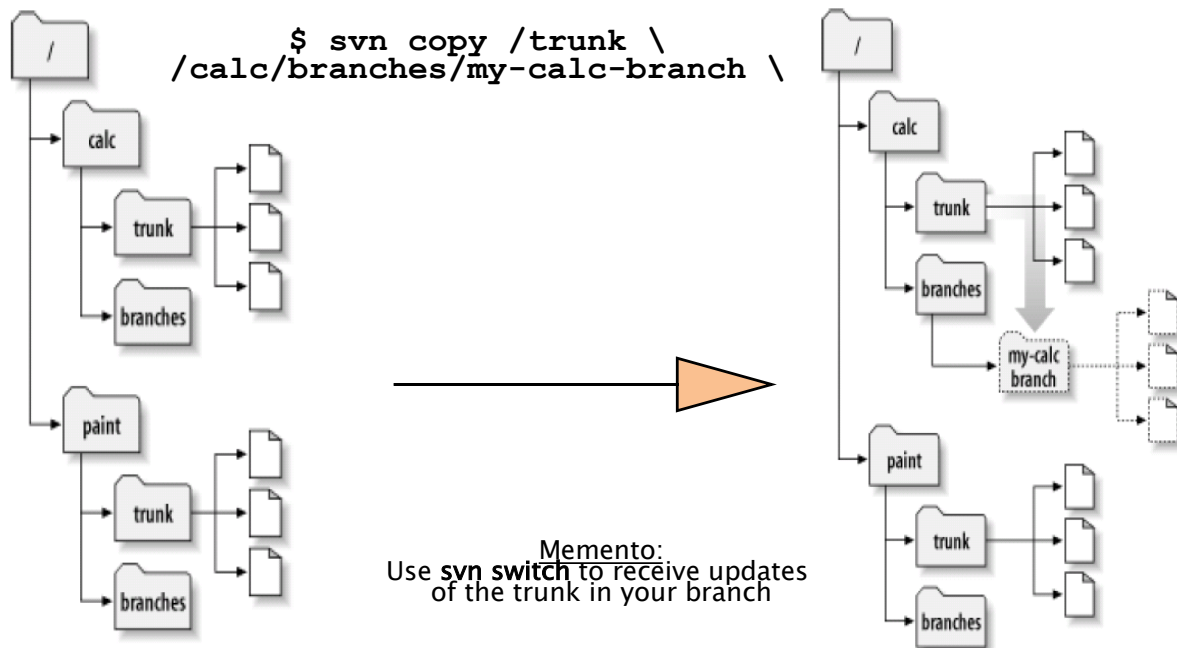
- Branches in subversion
  - ♦ exist as normal file system directories in the repository
    - carry some extra historical information
    - Do not exist in some “extra dimension”
- Subversion has no internal concept of a branch; it only deal with copies.
  - ♦ A directory becomes a branch because that is how we interpret it
  - ♦ Any copy brings also the previous history

---

62

# Branches in Subversion

You create a branch with `svn copy`:



63

## Subversion repo structure

- To use branches a repository by convention has three top-level folders:
  - ♦ **trunk**: contains the main branch
  - ♦ **branches**: contain the branches
    - one sub-folder for each branch
  - ♦ **tags**: contains snapshots of branches
    - One sub-folder per tag (version/release)
    - Copies created represent frozen baseline

64



# Merge

---

- When work is done in a branch, it must be brought back into the *trunk*.
- This is done by `svn merge` command.
  - ♦ Similar to `svn diff` command, instead of printing the differences to your terminal, it applies them directly to the local working copy. Svn diff command ignores ancestry, svn merge does not.
  - ♦ Two repository trees are compared, and the differences are applied to a working copy.
- Conflicts may be produced by `svn merge`:
  - ♦ They are solved in the usual way

---

65

## Wrap-up session

---

- Configuration management deals with several issues:
  1. Versioning
  2. Configuration
  3. Change control
  4. Build management
- Subversion is an open-source platform supporting 1, 2, 3

---

66

# References and Further Readings

---

- IEEE STD 1042 – 1987 IEEE guide to software configuration management
- IEEE STD 828–2012: IEEE Standard for Configuration Management in Systems and Software Engineering
- B.Collins–Sussman, B.W.Fitzpatrick C.M.Pilato. Version Control with Subversion: For Subversion 1.7, 2011
- Semantic Versioning. <http://semver.org>
- M.Fowler. Continuous Integration. <https://martinfowler.com/articles/continuousIntegration.html>