

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



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]

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

Goals of CM

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

VERSIONING

Versioning



Thesis.docx



ThesisFinal.docx



ThesisFinal
Final.docx



ThesisFinalest
Final.docx



ThesisFinalest
FinalForsure.docx



ThesisFinalestF**k
FinalForsure.docx

Terms

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

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

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

Version identification

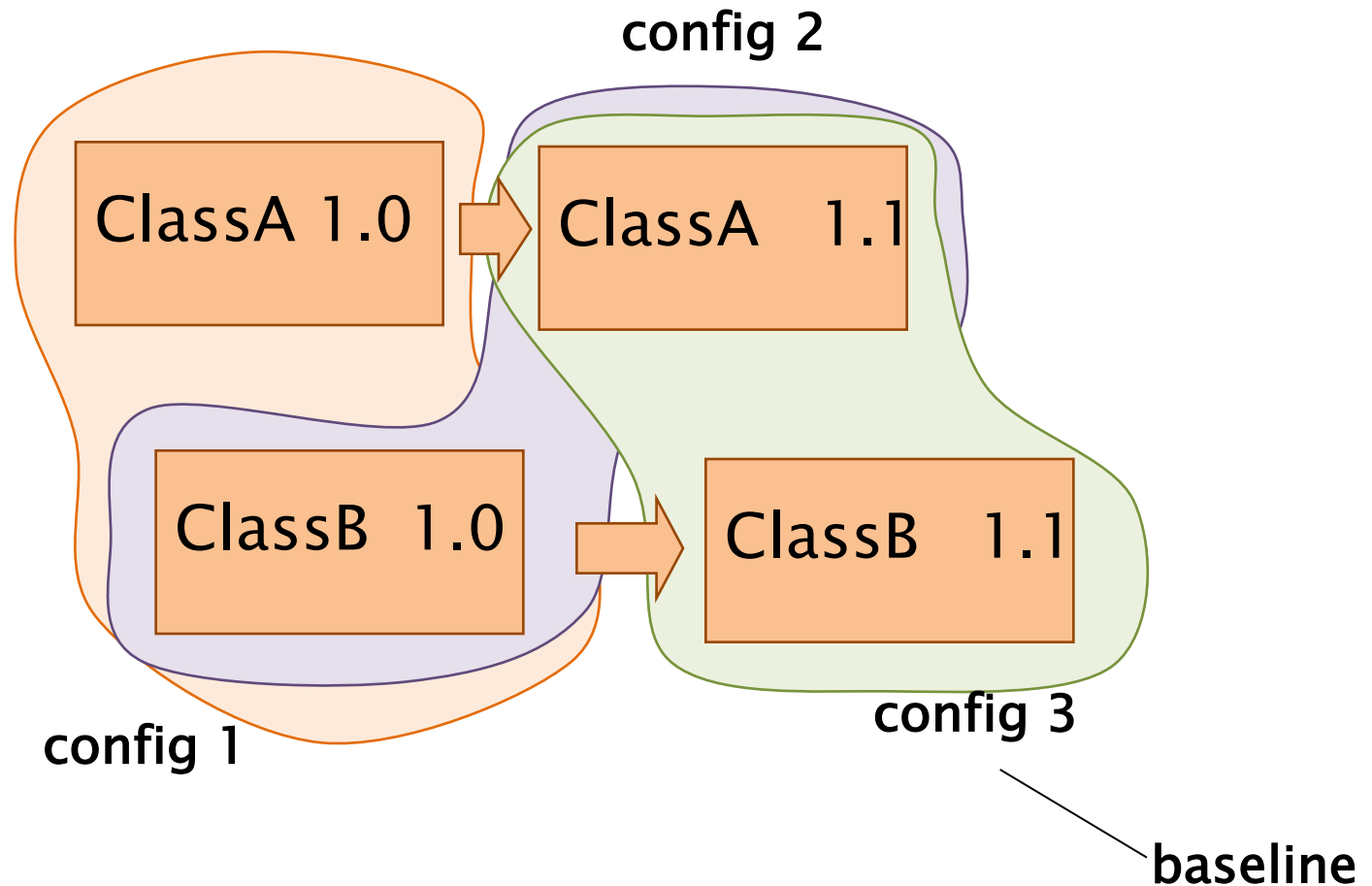
- Procedures for version identification should define an unambiguous way of identifying component versions
- Basic techniques for component identification
 - ◆ Version numbering
 - ◆ Attribute-based identification

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

Configuration

- Set of CIs, each in a specific version



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

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

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.

CHANGE CONTROL

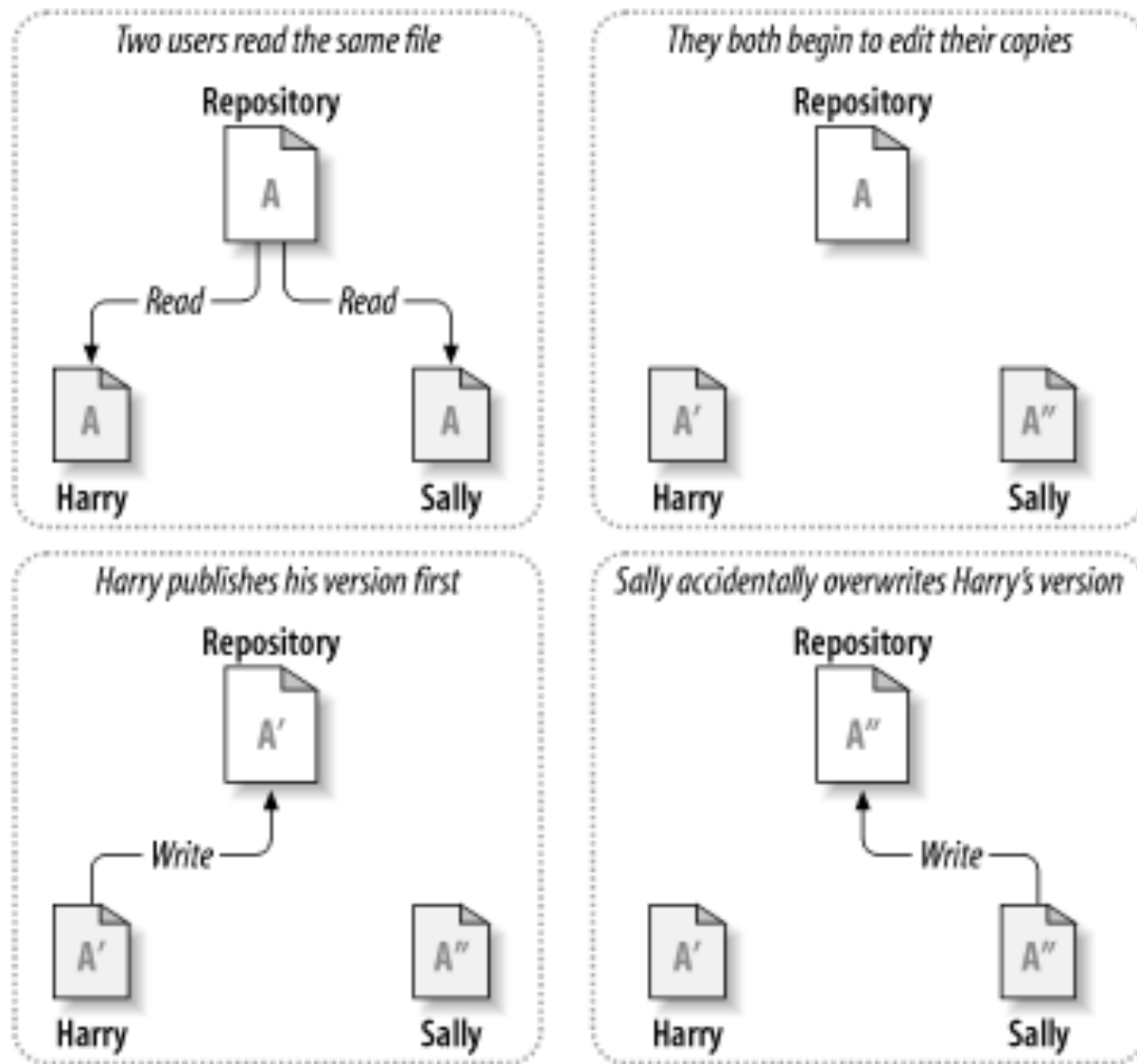
Repository

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

Typical case

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

File system



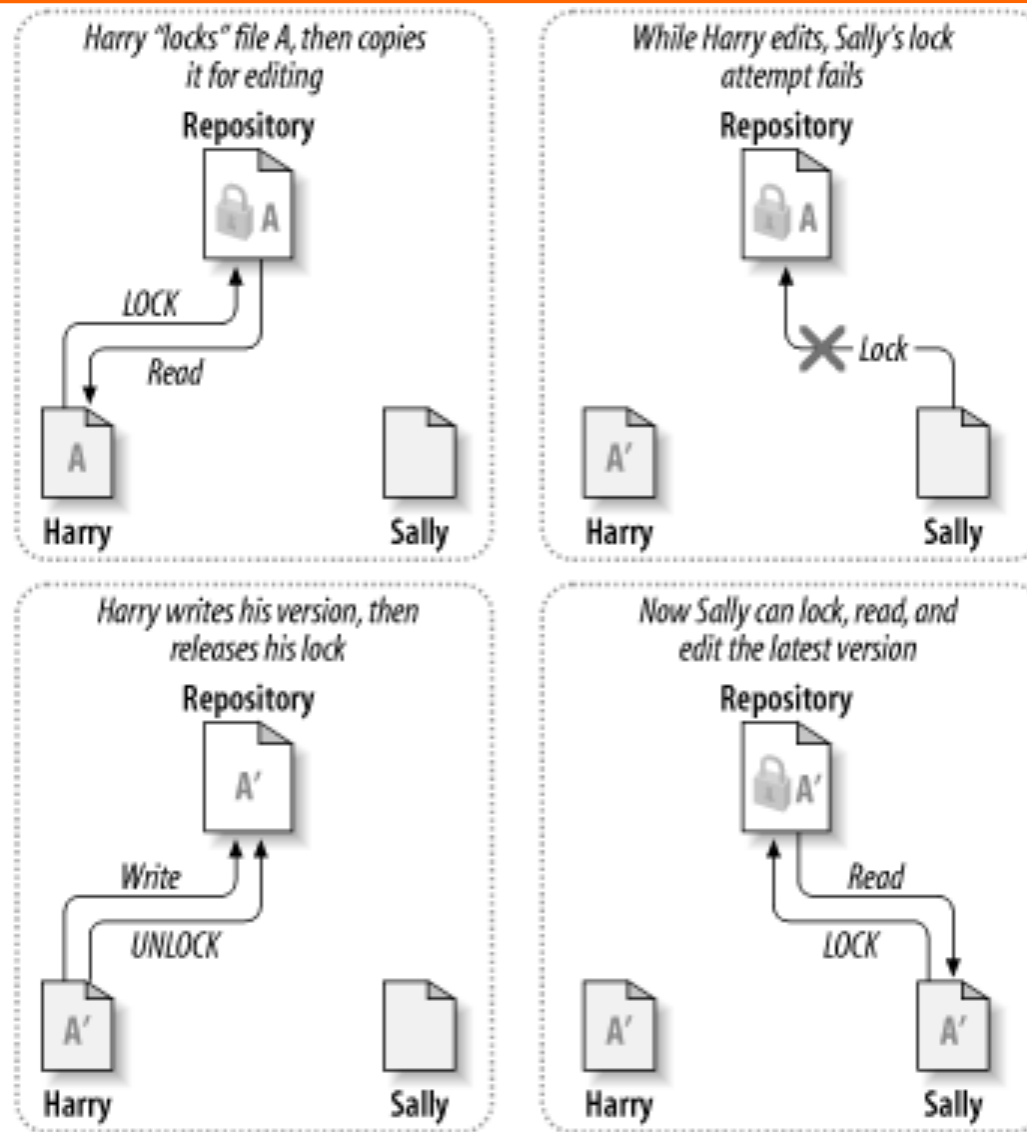
Check-in / check-out

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

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

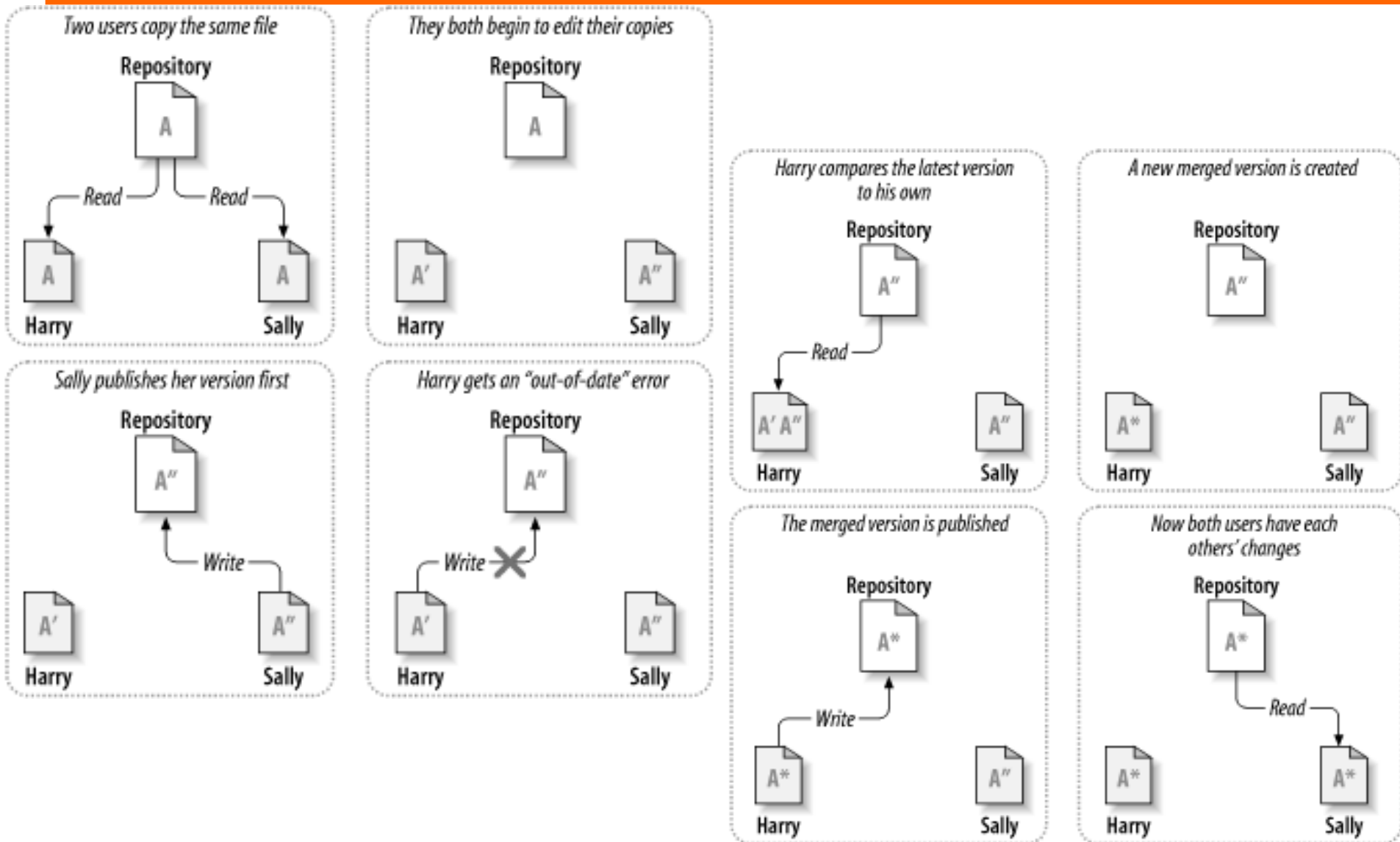
Lock-Modify-Unlock



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

Copy-Modify-Merge



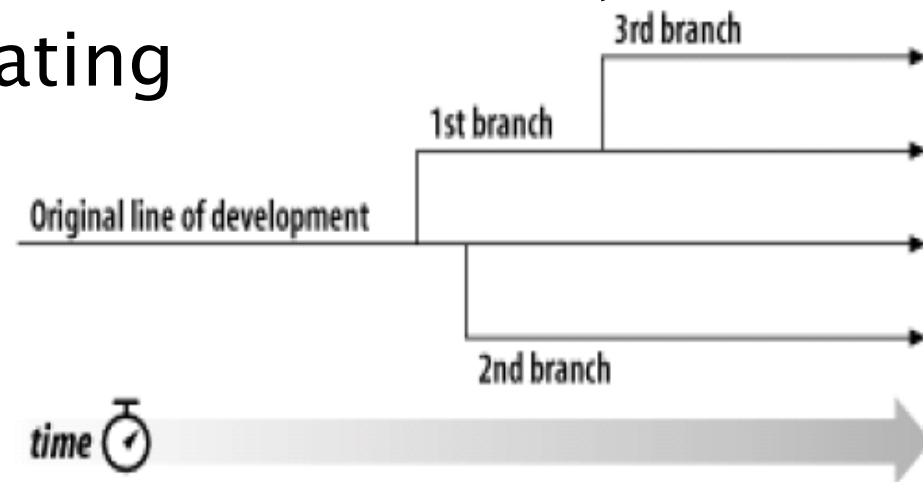
Copy–Modify–Merge

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

approach adopted by git

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



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

Tools

■ Change Control+Versioning+Configuration

- ◆ RCS LOCAL
- ◆ CVS CENTRALIZED
- ◆ SCCS
- ◆ PCVS
- ◆ Subversion
- ◆ BitKeeper
- ◆ Mercurial
- ◆ Git DISTRIBUTED



REPOSITORY:

- Local -> on a computer
- Centralized -> a central server to which more computers are connected
- Distributed -> a central server to which more computers are connected but each computer stores locally a copy of the repository
(the one we use: push, stores the new version in the local copy, commit, stores the new version present in the local copy but to the central server)

DATA MANAGEMENT MODELS:

- Save the differences (save space but you have to reconstruct the versions)
- Snapshots (occupy more space but easy to access to new versions)

Git

Github and Gitlab are different versions of Git

BUILD MANAGEMENT

Build management

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

Tools

- Build management

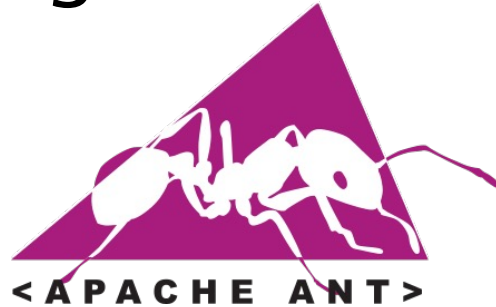
Take a configuration (set of files with their versions) and they set rules to take the files and compile them

- ◆ Make

- ◆ Ant

- ◆ Maven

- ◆ Gradle



maven



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

Continuous integration

- Commit frequently (In git you must push)
- Don't commit broken code (if not it will not compile)
- Don't commit untested code
- Don't commit when the build is broken
- Don't go home after committing until the system builds

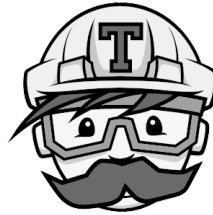
Tool CI

- Continuous Integration

- ◆ Travis CI

- ◆ Jenkins

- ◆ Cruise Control



Travis CI



Jenkins





VERSION CONTROL WITH SUBVERSION

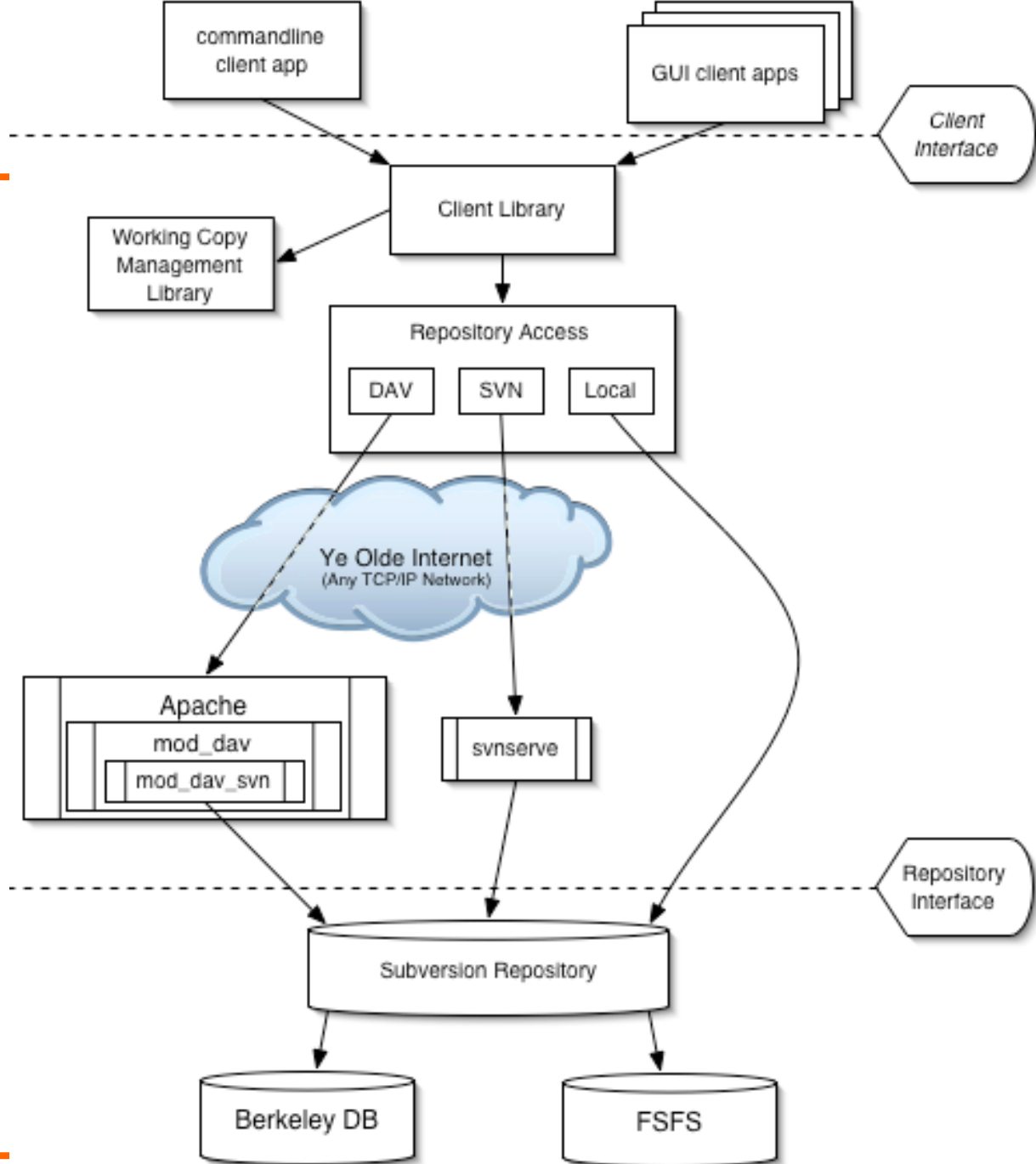
What is Subversion

- 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

Features

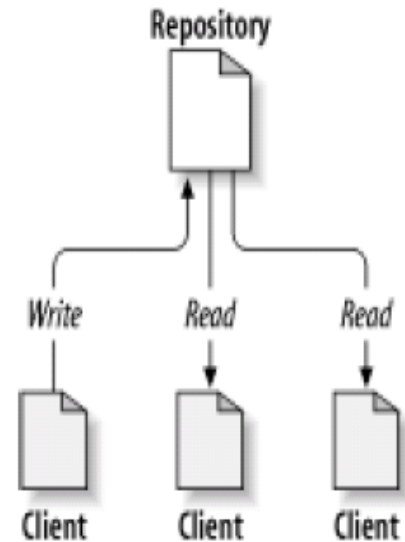
- Directory versioning and true version history
- Atomic commits
- Metadata versioning
- Several topologies of network access
- Consistent data handling
- Branching and tagging
- Usable by other applications and languages

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.

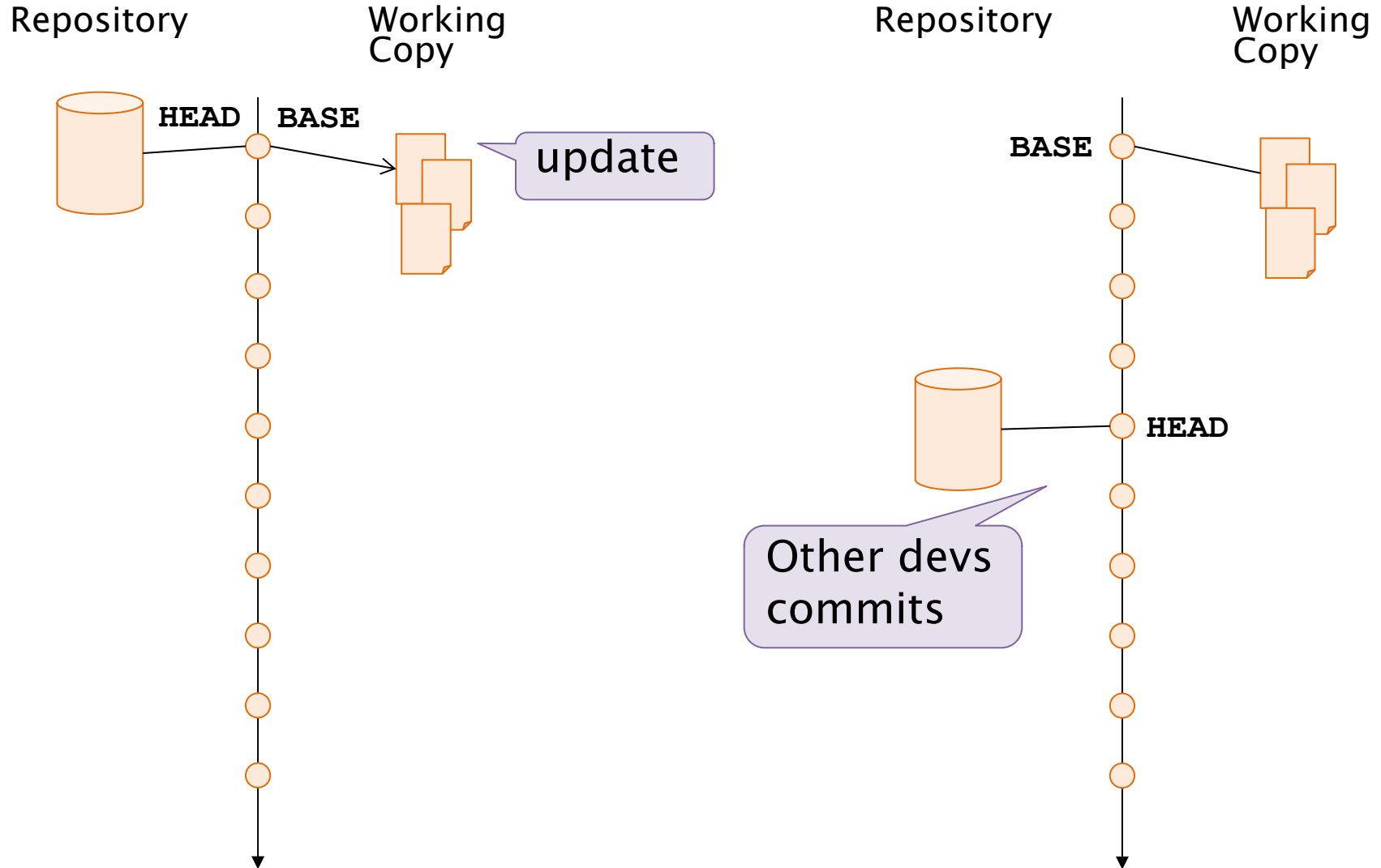
Revisions

- 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.

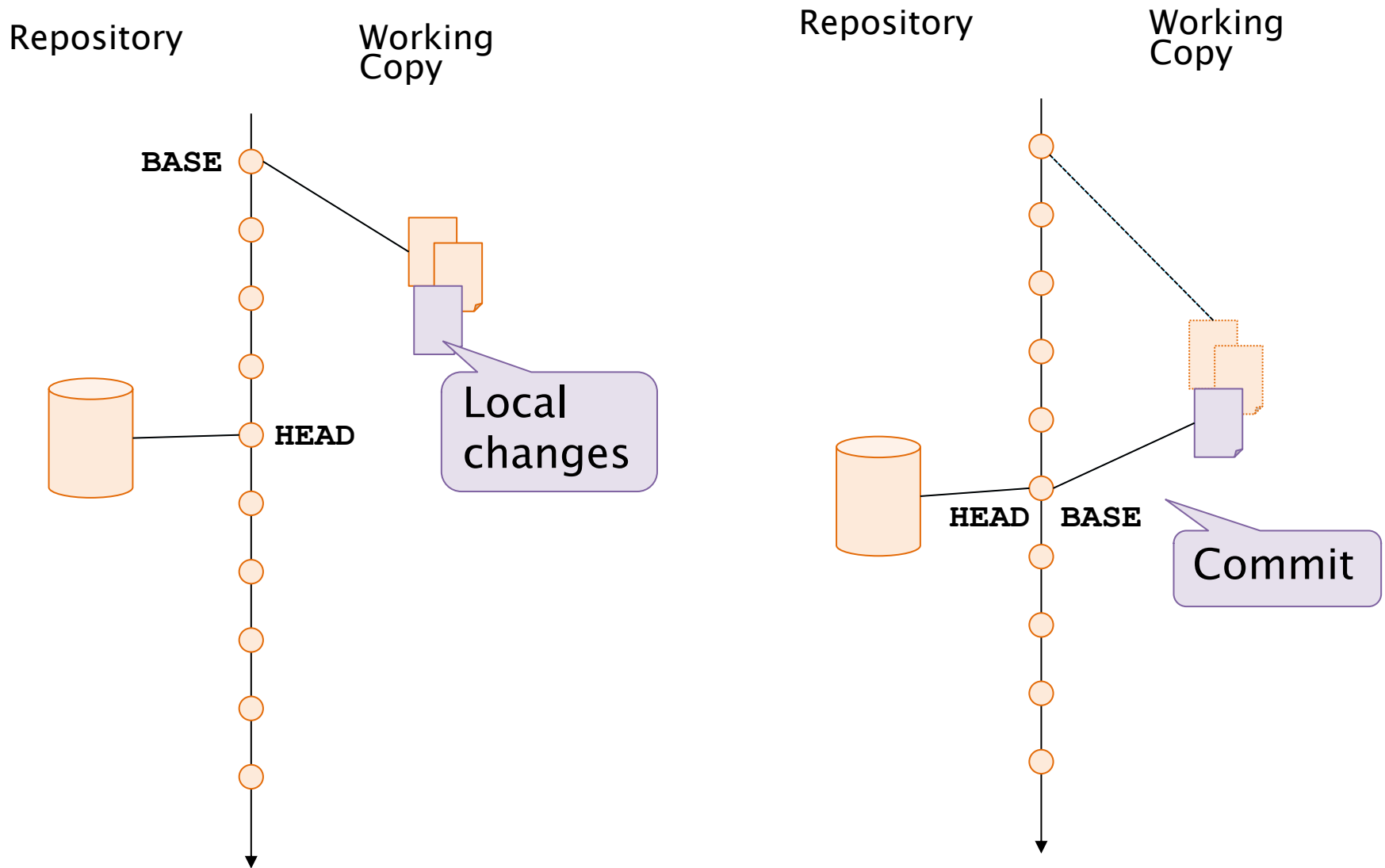
Base and head revisions

- **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

Head and base revisions



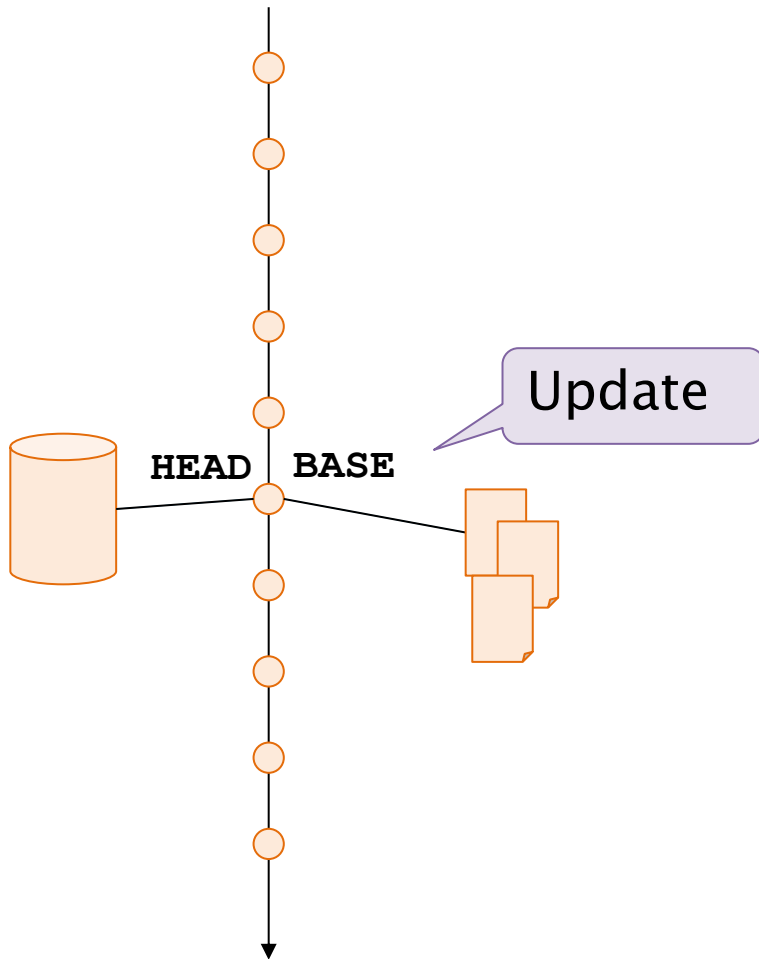
Mixed revisions



Mixed revisions

Repository

Working
Copy



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>*"

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`

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

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 a mix version of HEAD (**.rNEW**) and local (**.mine**) with change markers

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.
```

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

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

Conflict example

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

Top piece of bread

Mayonnaise

Lettuce

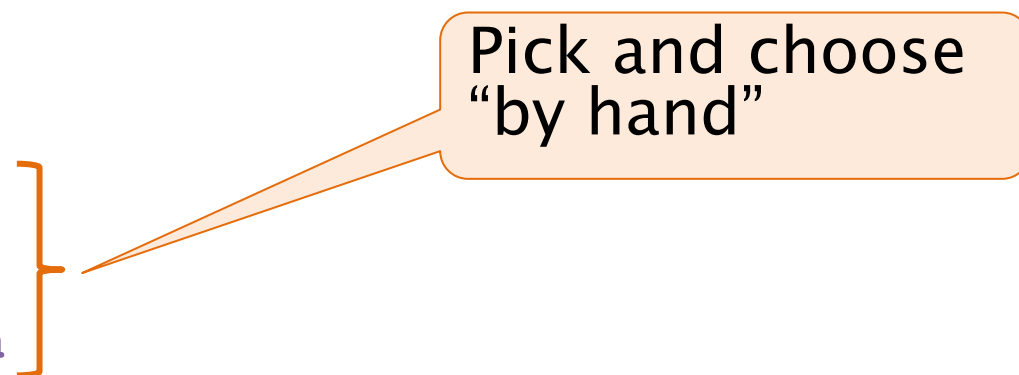
Mortadella

Prosciutto

Grilled Chicken

Creole Mustard

Bottom piece of bread



Pick and choose
“by hand”

Conflict example

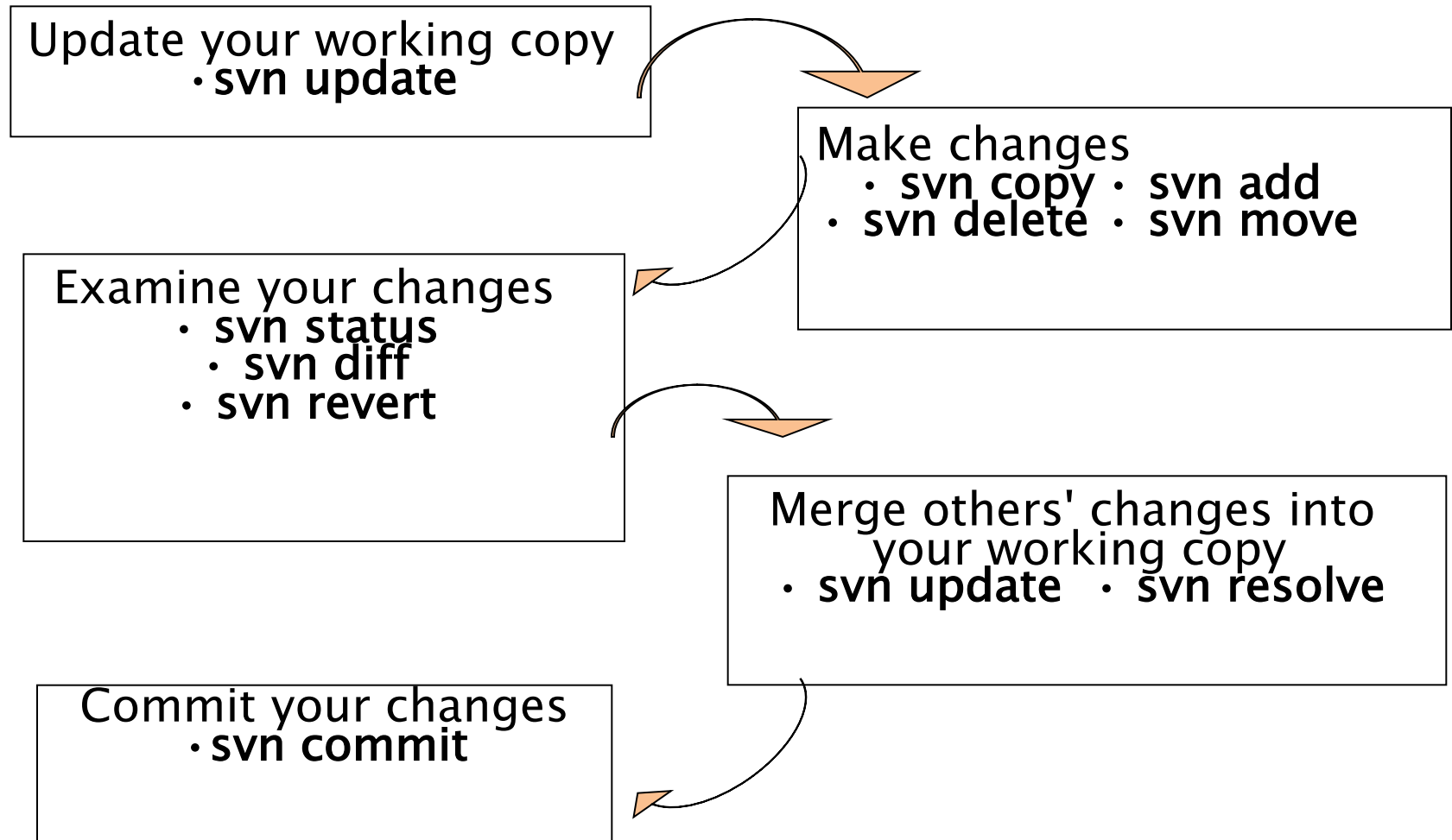
- Once the conflict has been composed you ought to signal it has been resolved

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

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

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

Typical work cycle

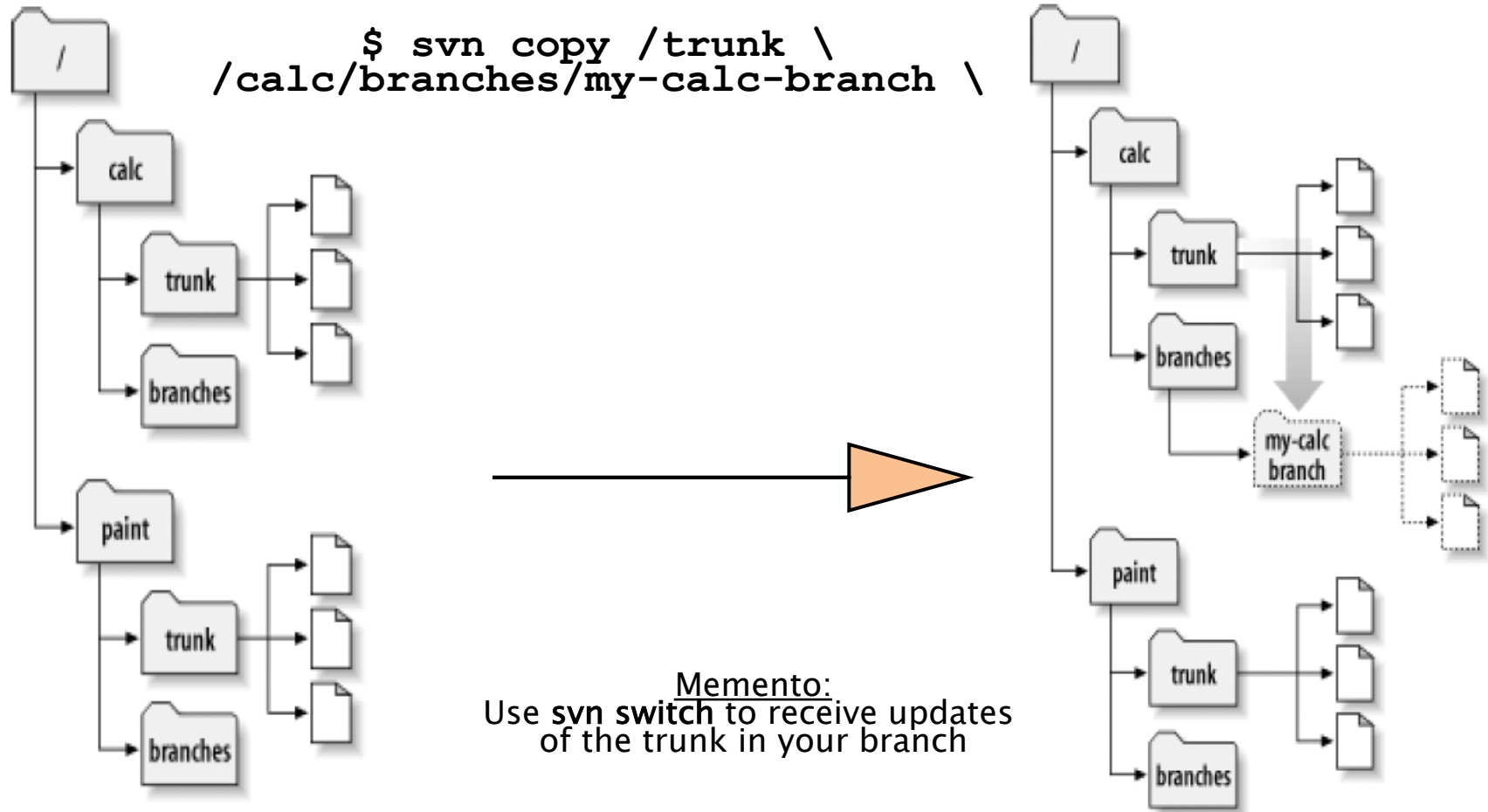


Branches in Subversion

- 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

Branches in Subversion

You create a branch with `svn copy`:



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

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

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

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>