Configuration Management

Object-Oriented Programming

http://softeng.polito.it

Version 0.2

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

Configuration Management

- A discipline applying technical and administrative direction and surveillance 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
- What is the correct set of documents for a specific need?
  - configuration
- Who can access and change what?
  - Change control
- How the system is obtained?
  - build

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 version are numbered and kept
  - User decides to change version number with specific operation (commit)
  - It is possible to retrieve any previous version

Version

- An initial release or re-release of a configuration item
- Instance of CI
  - Ex Req document 1.0
  - Req document 1.1

Version identification

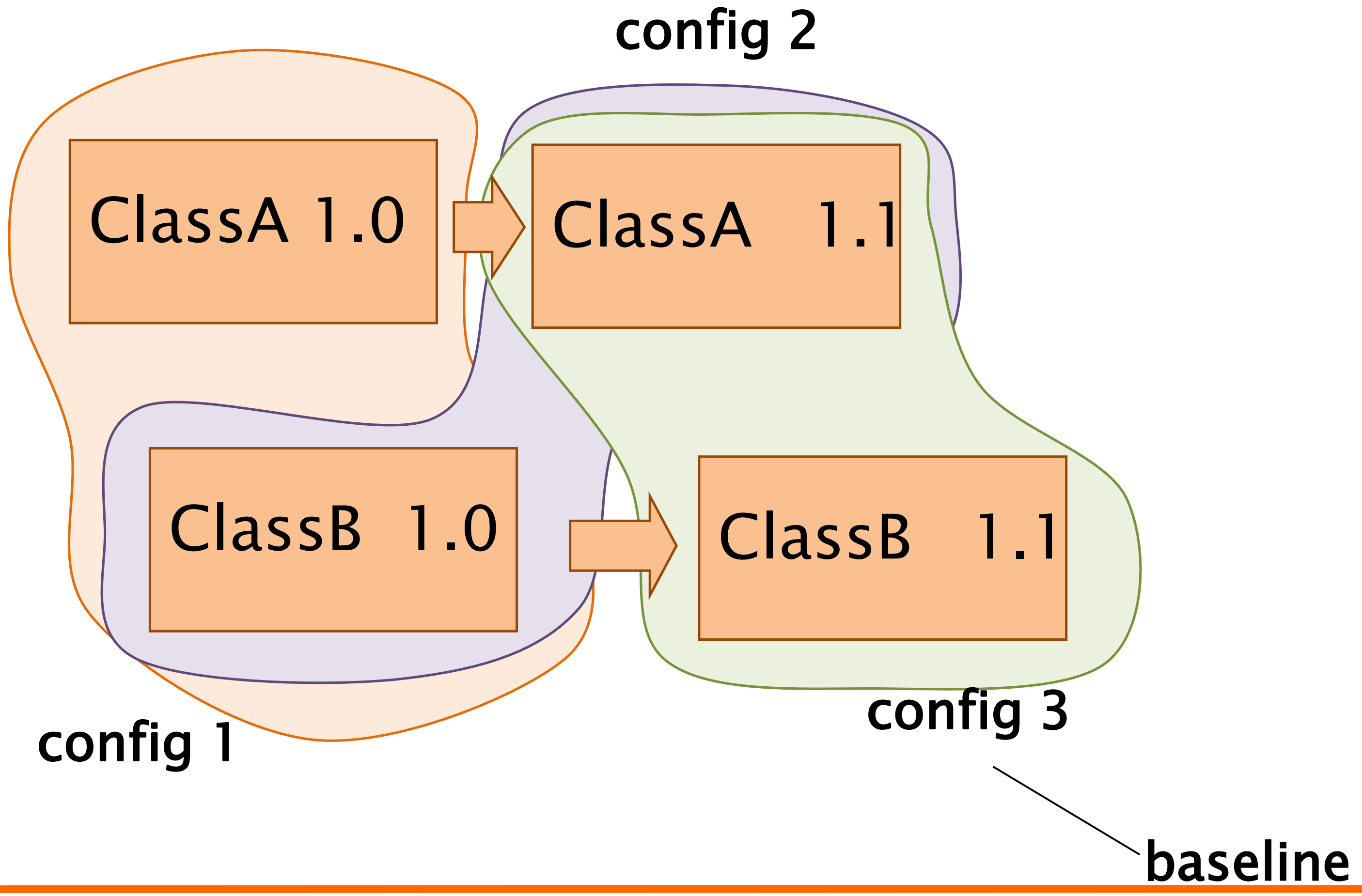
- Procedures for version identification should define an unambiguous way of identifying component versions
- basic techniques for component identification
  - Version numbering
  - Attribute-based identification

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 certain version
  - Same CI may appear in different configurations
  - Also configuration has version

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

CHANGE CONTROL

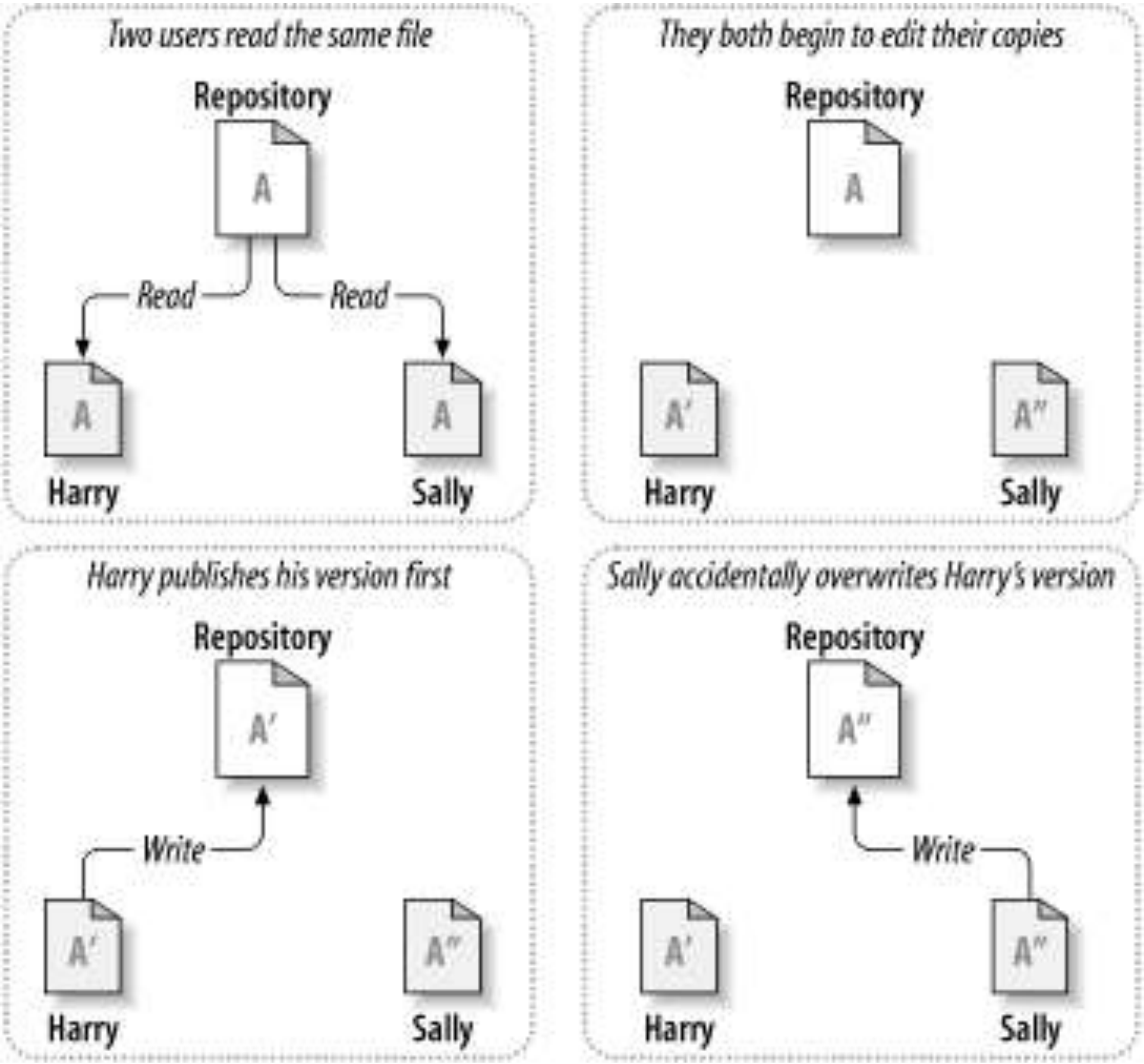
Repository

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

Typical situation

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

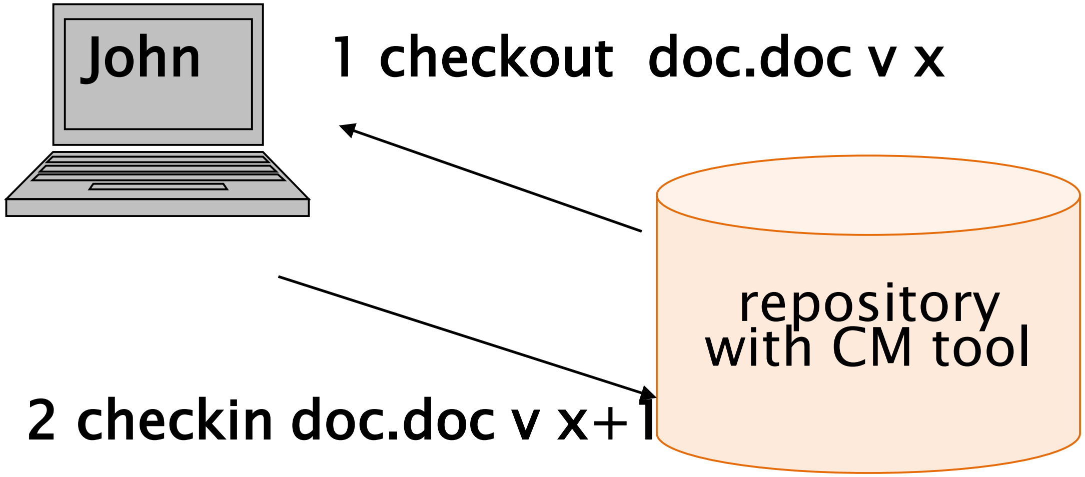
File system limitations


Two users read the same file
Repository
A
Read
Read
A
Harry
A
Sally
They both begin to edit their copies
Repository
A
A′
Harry
A″
Sally
Harry publishes his version first
Repository
A′
Write
A′
Harry
A″
Sally
Sally accidentally overwrites Harry's version
Repository
A″
Write
A′
Harry
A″
Sally

Check-in / check-out

- Check-out
  - Extraction of CI from repository
    - with goal of changing it or not
    - After checkout next users are notified
- Check-in (or commit)
  - Insertion of CI under control

Repository – check in checkout



John

1 checkout doc.doc v x

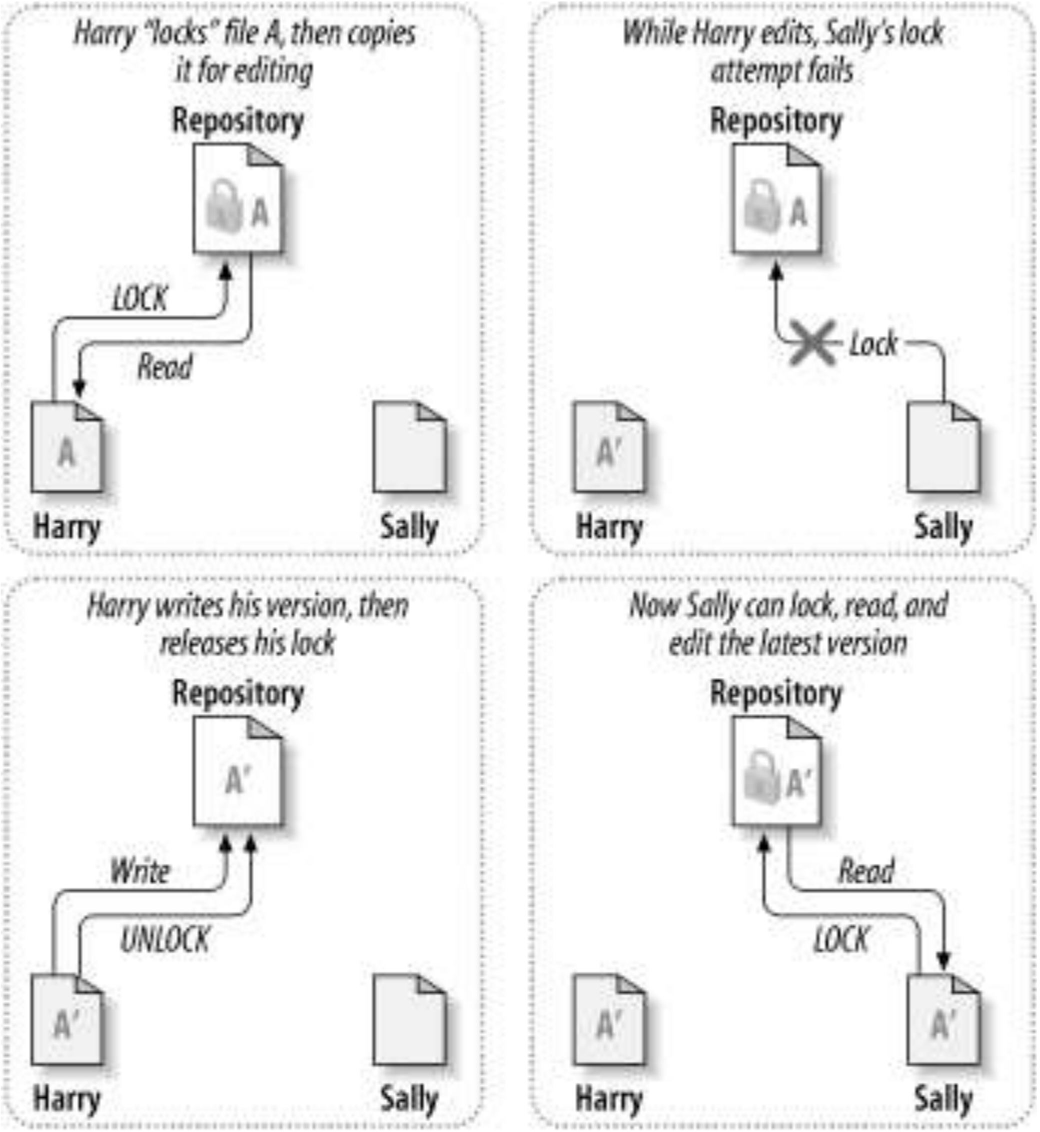
repository with CM tool

2 checkin doc.doc v x+1

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

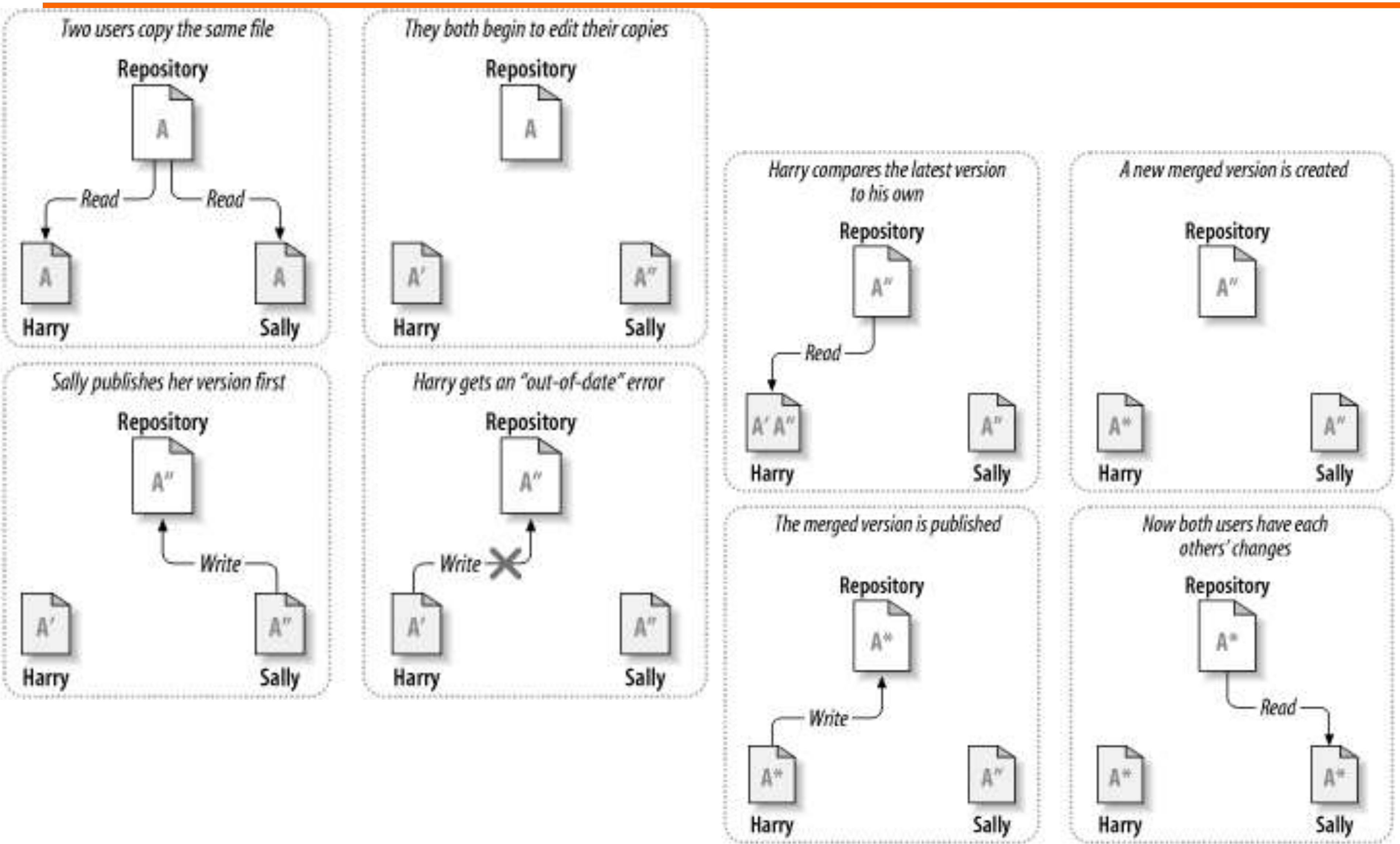

Harry "locks" file A, then copies it for editing
Repository
A
LOCK
Read
A
Harry
Sally
While Harry edits, Sally's lock attempt fails
Repository
A
Lock
A'
Harry
Sally
Harry writes his version, then releases his lock
Repository
A'
Write
UNLOCK
A'
Harry
Sally
Now Sally can lock, read, and edit the latest version
Repository
A'
Read
LOCK
A'
Harry
A'
Sally

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

Lock-Modify-Unlock

- Pro
  - Conflicts are impossible
- Con
  - No parallel work is possible, delays con be introduced
  - Developers can possibly forget to unlock so blocking the whole team

Copy-Modify-Merge


Two users copy the same file
Repository
A
Read
Read
A
A
Harry
Sally
They both begin to edit their copies
Repository
A
A′
A″
Harry
Sally
Sally publishes her version first
Repository
A″
Write
A′
A″
Harry
Sally
Harry gets an "out-of-date" error
Repository
A″
Write
A′
A″
Harry
Sally
Harry compares the latest version to his own
Repository
A″
Read
A′ A″
A″
Harry
Sally
A new merged version is created
Repository
A″
A*
A″
Harry
Sally
The merged version is published
Repository
A*
Write
A*
A″
Harry
Sally
Now both users have each others' changes
Repository
A*
Read
A*
A*
Harry
Sally

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

Copy-Modify-Merge

- Pro
  - More flexible
  - Several developers can work in parallel
  - No developer can block others
- Cons
  - Requires care to resolve the conflicts

Tools

- CM + VM
  - RCS
  - CVS
  - SCCS
  - PCVS
  - Subversion
  - BitKeeper
  - Git

VERSION CONTROL WITH SUBVERSION

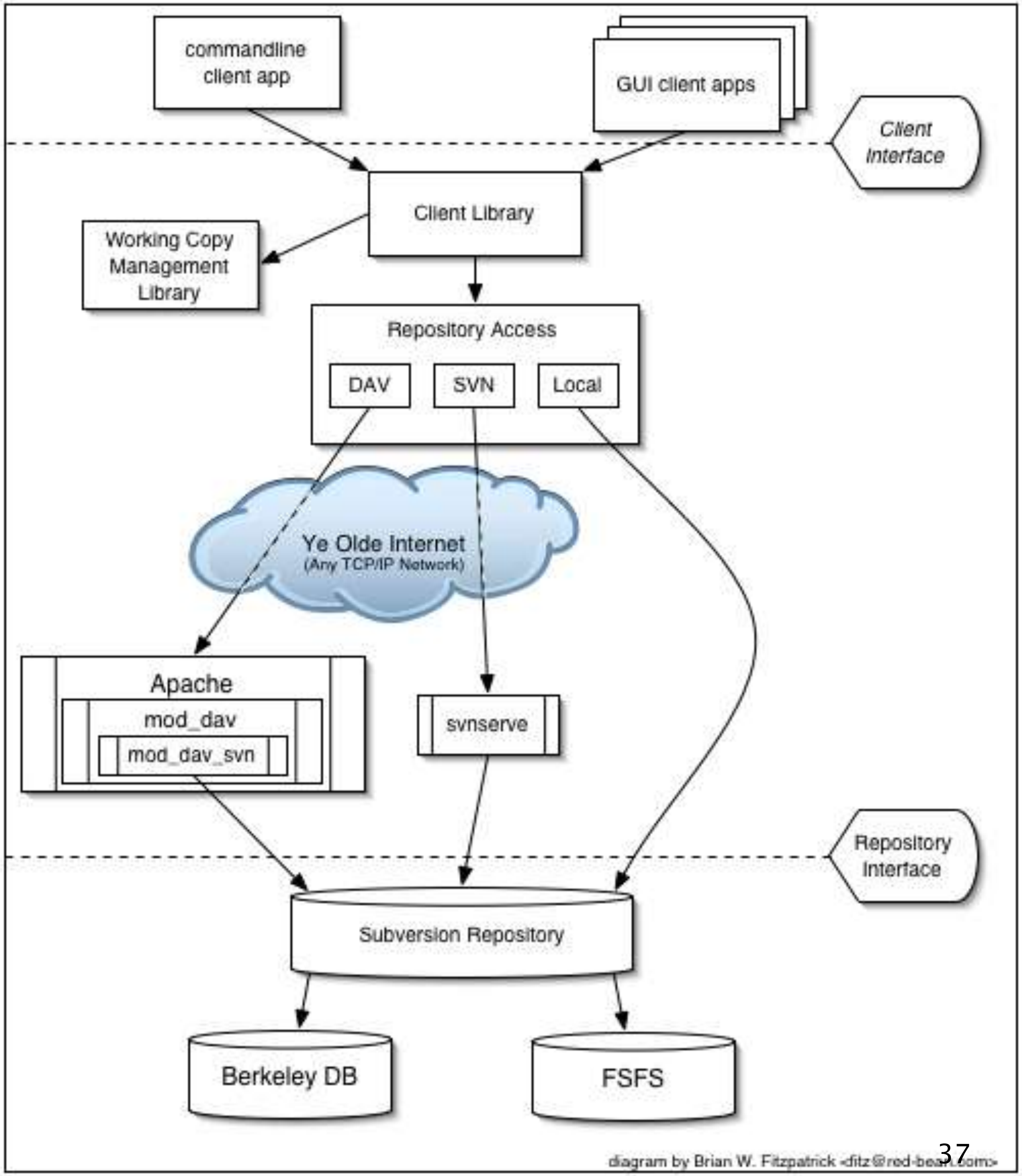
What is Subversion

- Free/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

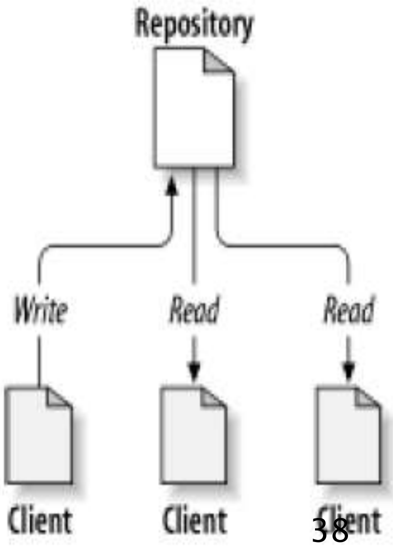
- Directory versioning and true version history
- Atomic commits
- Metadata versioning
- Several topologies of network access
- Consistent data handling
- Branching and tagging
- Usable by other applications and languages

Architecture



The 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, this creates a new state of the filesystem tree, called a revision.
- Global revision numbers: each revision is assigned a unique natural number, one greater than the number of the previous revision
  - An empty, freshly created repository has revision zero
- The whole repo gets a new revision number
  - Revision *N* represents the state of the repository after the *N*th commit.

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.

Basic Procedure

- Create working copy from a repository
  - `svn checkout <repository>`

  When ready…
- Synchronize contents of WC with repo
  - `svn update`

  Work on WC
- Possibly add new files
  - `svn add <file list>`
- Push work to repository
  - `svn commit -m ”<Log message>"`

Conflicts

- A conflict arise, upon commit, if the file has been updated in the meanwhile
- A conflict occurs if:
  - $M > N$ and
  - revision M differs from revision N
- Where
  - N: the revision (BASE) that was modified (the repo revision at the time of last update)
  - M: the current revision (HEAD) in the repository ($\geq$N)

Conflicts

- For every conflicted file, Subversion places three extra unversioned files in your working copy:
  - **`filename.mine`** : This is your file as it existed in your working copy before you updated your working copy—that is, without conflict markers. This file has only your latest changes in it.
  - **`filename.r`***OLDREV* : This is the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.
  - **`filename.r`***NEWREV* : This is the file that your Subversion client just received from the server when you updated your working copy. This file corresponds to the HEAD revision of the repository.

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

- This is what you get in your working copy

```
$ ls
sandwich.txt
sandwich.txt.mine
sandwich.txt.r1
sandwich.txt.r2
```

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

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`** now is

```
Top piece of bread
Mayonnaise
Lettuce
Mortadella
Prosciutto
Grilled Chicken
Creole Mustard
Bottom piece of bread
```

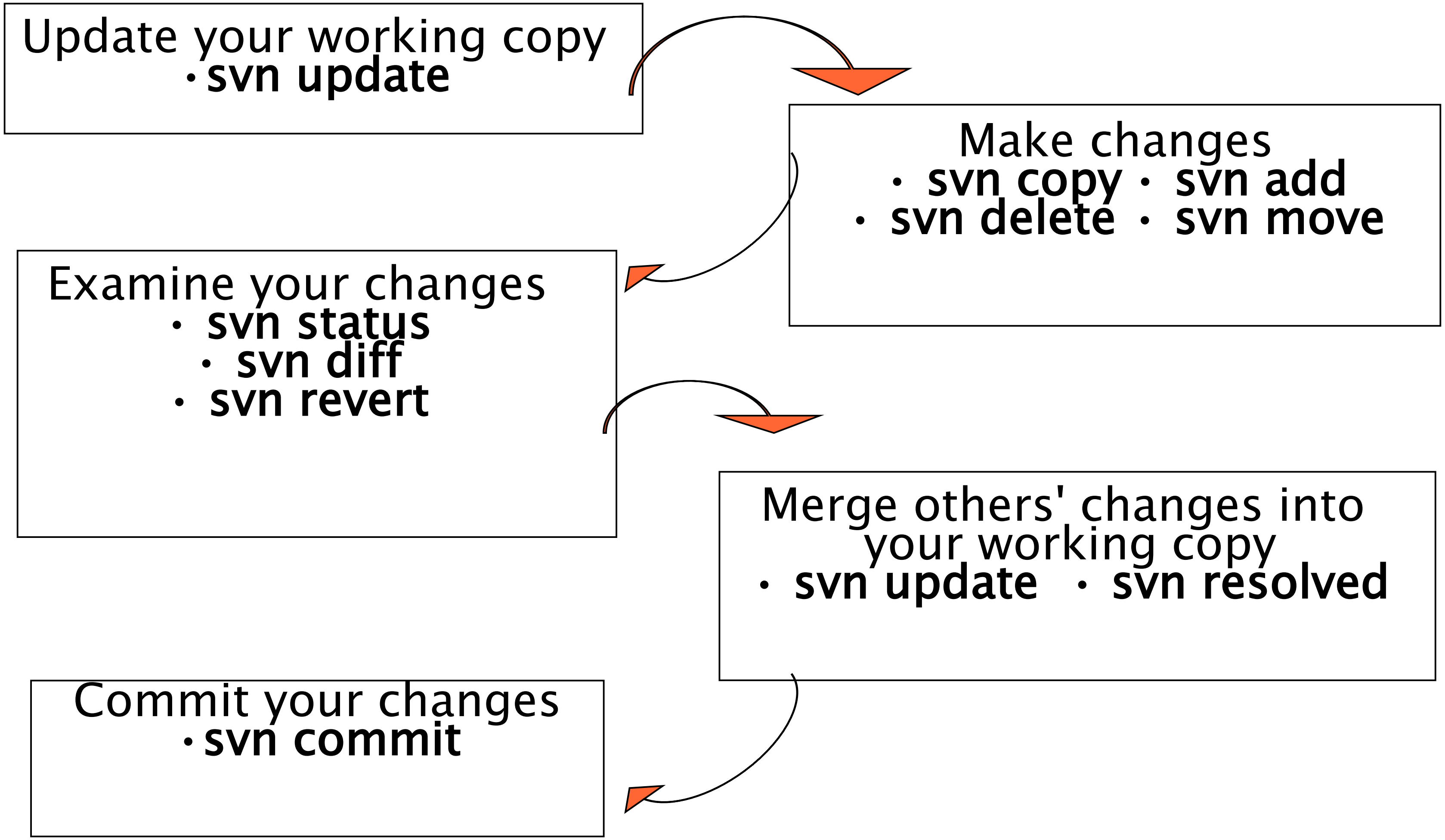
Pick and choose “by hand”

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 ”Pick and choosen.”
```

Typical work cycle



Update your working copy

- svn update

Make changes

- svn copy
- svn add
- svn delete
- svn move

Examine your changes

- svn status
- svn diff
- svn revert

Merge others' changes into your working copy

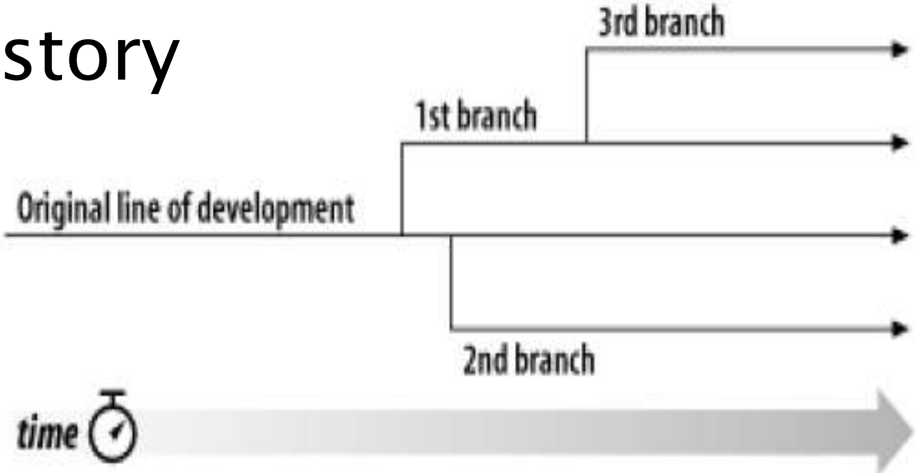
- svn update
- svn resolved

Commit your changes

- svn commit

Branches: general concept

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

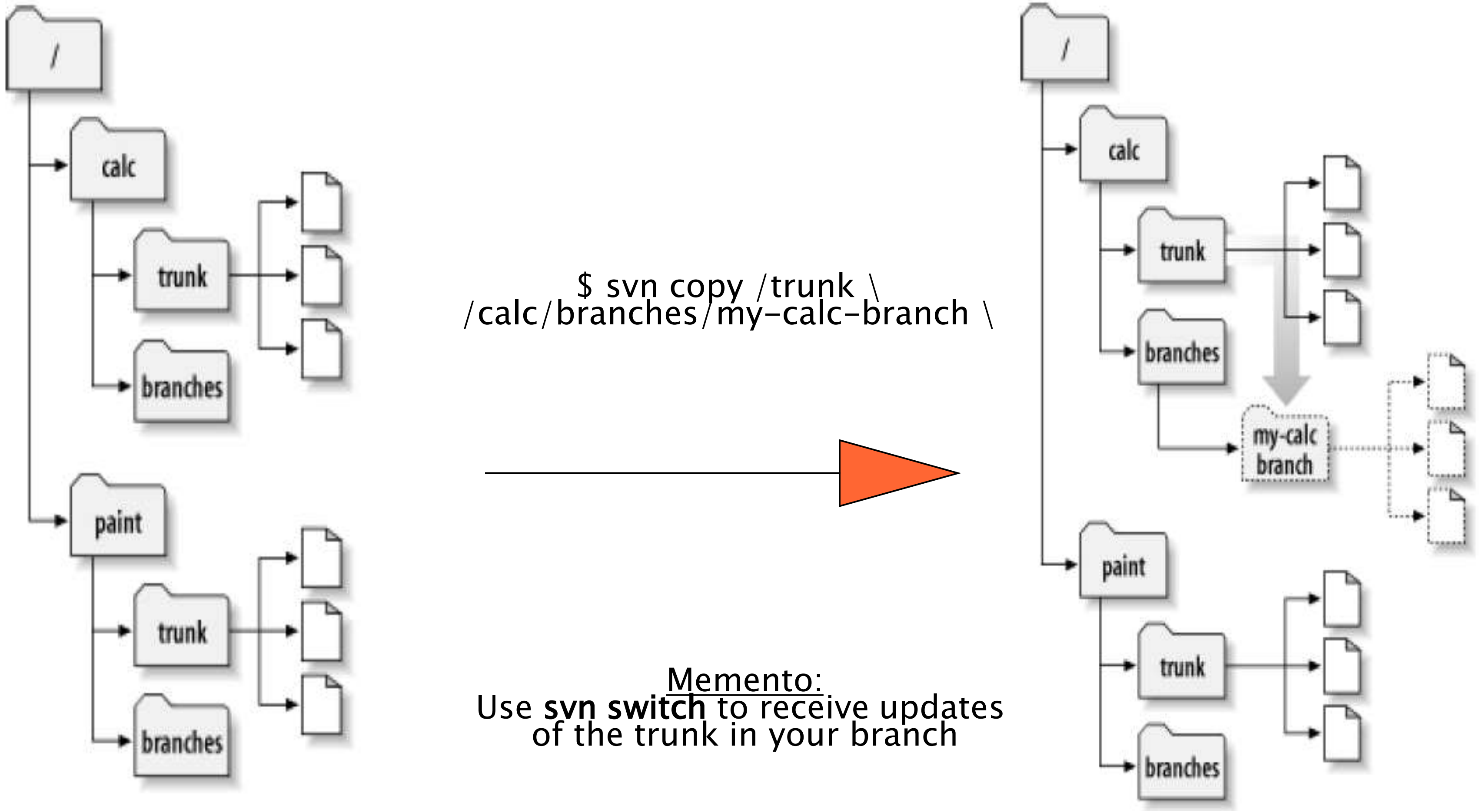


Branches in Subversion

- Unlike many other version control systems, Subversion's branches exist as normal filesystem directories in the repository, not in an extra dimension. These directories just happen to carry some extra historical information.
- Subversion has no internal concept of a branch—only copies. When you copy a directory, the resulting directory is only a “branch” because you attach that meaning to it. You may think of the directory differently, or treat it differently, but to Subversion it's just an ordinary directory that happens to have been created by copying.

Branches in Subversion

You create a branche with **svn copy**:



```
$ svn copy /trunk \
/calc/branches/my-calc-branch \
```

Memento:
Use **svn switch** to receive updates
of the trunk in your branch

Merge

- When you finish your work in your branch, you need to merge it in the trunk. This is done by **svn merge** command.
- This command is a very close cousin to the **svn diff** command.
- **Svn merge**, instead of printing the differences to your terminal, however, it applies them directly to your working copy as local modifications. **Svn diff** command ignores ancestry, **svn merge** does not.
- A better name for the command might have been **svn diffand-apply**, because that's all that happens: two repository trees are compared, and the differences are applied to a working copy.
- Conflicts may be produced by svn merge: you need to solve them.

Wrap-up session

Comments

- Structure of the comment

```
<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

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**