

# Conduite de Projet

## Cours 2 — Version Control

Stefano Zacchioli  
zack@irif.fr

Laboratoire IRIF, Université Paris Diderot

2018–2019

URL <https://epsilon.cc/zack/teaching/1819/cproj/>  
Copyright © 2012–2019 Stefano Zacchioli  
License Creative Commons Attribution-ShareAlike 4.0 International License  
<https://creativecommons.org/licenses/by-sa/4.0/>



# Sommaire

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)
- 6 Subversion
- 7 Git

# Outline

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)
- 6 Subversion
- 7 Git

# Change

During the life time of a software project, *everything* changes:

- bugs are discovered and have to be fixed (code)
- system requirements change and need to be implemented
- external dependencies change
  - ▶ e.g. new version of hardware and software you depend upon
- competitors might catch up

Most software systems can be thought of as a set of **evolving versions**

- potentially, each of them has to be maintained concurrently with the others

# Configuration management

## Definition (Configuration Management)

*Configuration Management (CM)* is concerned with the **policies**, **processes**, and **tools** for managing changing software systems.  
(Sommerville)

### Why?

- it is easy to lose track of which changes have been incorporated in each version
  - ▶ things get even messier with versions which have to be maintained in parallel
- minimize risks of working on the wrong version
- useful for *solo* projects ⇒ backup on steroids + it's easy to forget which change has been made and why
- useful for team project ⇒ help in praising(, blaming), know who to ask

# Configuration management activities

**Change management** keep track of request for changes (from both customers and developers), evaluate costs/risks/benefits, making commitment to change

**Version management** (or *version control*, revision control, etc.) keeping track of multiple version of (software) components and ensure unrelated changes do not interfere

**System building** assembling program components, data, and libraries into executable systems

**Release management** preparing software for external release and keep track of which version is in use at which customer site

**Version management** (or *version control*, revision control, etc.)  
keeping track of multiple version of (software)  
components and ensure unrelated changes do not  
interfere

# Outline

- 1 Configuration management
- 2 diff & patch**
- 3 Version control concepts
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)
- 6 Subversion
- 7 Git

## Before version control: diff & patch

The Swiss army knife of change management: diff & patch

`diff` compute the difference  $D$  among a file  $A$  and a file  $B$

- can be applied recursively to directories

`patch` apply a difference  $D$  (usually computed using `diff`) to a file  $A$  (possibly producing a new file  $B$ )

# Demo

- patches are (were) usually conveyed via email messages to the main software maintainer
- best practices
  - ▶ add to emails clear and concise explanations of the purpose of the attached patch
  - ▶ do the same in the source code added by the patch
    - ★ nothing new: usual good coding practice; it becomes more important only because the number of software authors grows...
  - ▶ <http://tldp.org/HOWTO/Software-Release-Practice-HOWTO/patching.html>

## Poor man's version control

Projects by a license student often look like this:

```
lucien> ls  
a.out  
projet.ml  
projet-save.ml  
projet-hier.ml  
projet-marche-vraiment.ml  
projet-dernier.ml
```

- what are the differences among the 5 source files?
- what are the *relationships* among them?
- hard to answer without specific utilities

## Poor men's version control (plural)

Project by *a group* of license students:

```
lucien> ls ~joel/projet
```

```
a.out
```

```
module.ml
```

```
module-de-julien-qui-marche.ml
```

```
projet.ml
```

```
projet-save.ml
```

```
projet-hier.ml
```

```
projet-marche-vraiment.ml
```

```
projet-dernier.ml
```

```
lucien> ls ~julien/projet
```

```
a.out
```

```
module.ml
```

```
projet.ml
```

```
projet-recu-de-joel.ml
```

```
module-envoye-a-joel.ml
```

What is the right combination of projet.ml and module.ml to obtain a good grade at the exam?

## diff & patch to the rescue

To exchange projet.ml and module.ml a group of students can rely on emails, diff, and patch (a huge improvement!)

Julien

```
lucien> diff -Nurp projet-hier.ml projet.ml > mescorrections
lucien> mail -s "Voici mes modifs" joel@lucien < mescorrection
```

Joel

```
lucien> mail
Mail version 8.1.2 01/15/2001. Type ? for help.
> 1 julien@home Fri Sep 13 20:06 96/4309 voici mes modifs
& s 1 /tmp/changes
& x
lucien> patch < /tmp/changes
```

Julien's changes between projet-hier.ml and projet.ml are now **integrated** in Joel's copy of projet.ml (hoping no conflicting changes have been made by Joel...)

## diff & patch: except that...

Nonetheless, on exam day nothing works, although it worked just the day before. *Panicking*, you'll try to understand:

- **what** has changed
- **who** did the change
  - ▶ probably you don't care about **why**, but still...
- **when** it has been done
- which state, not including that change, works properly
- **how** to get back to that state

⇒ you (badly) need a *real* Version Control System

# Outline

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts**
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)
- 6 Subversion
- 7 Git

# Version Control System (VCS)

## A version control system

- manage specific artifacts which form your source code
  - ▶ files, directories, their attributes, etc.
- is able to store changes to those artifacts (a VCS implements the notion of version for source code)
  - ▶ who has done a change
  - ▶ wrt which state
  - ▶ why
  - ▶ when
- can show the differences among different (stored) states
- can go back in time and restore a previous state
- can manage concurrent work among developers, distributing the changes among them

# Basic VCS concepts

A few basic concepts are shared across VCSs: <sup>1</sup>

**revision** (or **version**) a specific state, or point in time, of the content tracked by the VCS

- granularity and scope vary

**history** a set of revisions, (partially) ordered

---

1. although the actual naming changes from system to system; we'll stick to the naming presented here

## Basic VCS concepts (cont.)

A few basic concepts are shared across VCSs: <sup>1</sup>

**repository** (or **depot**) where the tracked content and all its history, as known to the VCS, is stored

- might be local or remote

**working copy** a local copy of a revision, which might be acted upon

- where the “real” work happens

**checkout** (or **clone**) the action of creating a working copy from a repository

---

1. although the actual naming changes from system to system; we'll stick to the naming presented here

## Basic VCS concepts (cont.)

A few basic concepts are shared across VCSs: <sup>1</sup>

**change** (or **delta**) a specific modification to (or *with respect to*) the content tracked by the VCS

- granularity vary

**commit** (as a verb) the act of writing a change performed in the working copy back to the repository

- = adding a new revision to the history

**commit** (as a substantive) same as change, for changes that have been committed

**diff** the act of (or the result of) inspecting the differences among two revisions, or among a revision and the working copy

- inspection format is usually diff

---

1. although the actual naming changes from system to system; we'll stick to the naming presented here

# Branching and merging

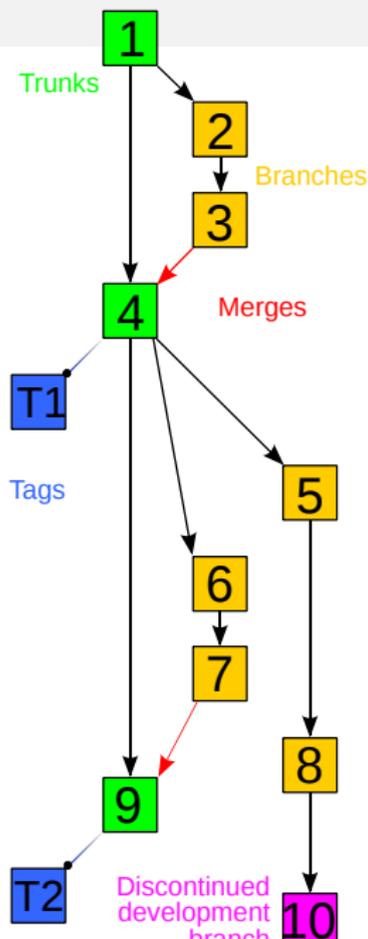
**branch** (verb) the act of duplicating (or “forking”) a specific revision in history, to open up a new line of development

- branches are usually named

**branch** (substantive) subset of history rooted at a fork point and extending until the next merge point

**merge** (verb) the act of joining together multiple lines of development, reconciling all their changes together

**merge** (substantive) the point in history where the merge happens

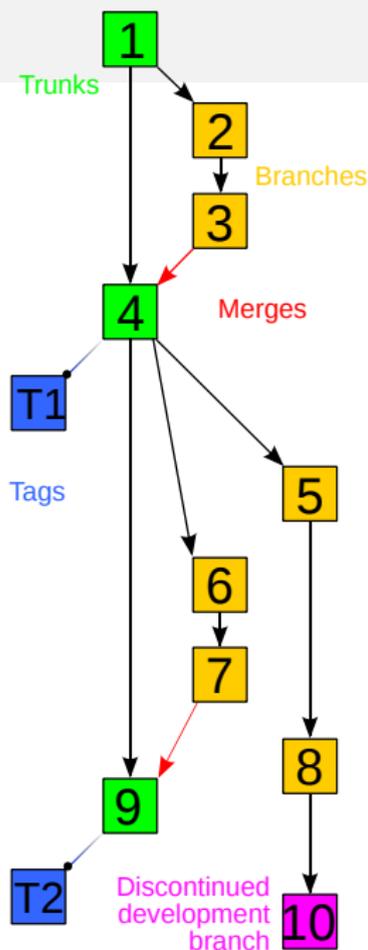


## Branching and merging (cont.)

- assuming an idealized purely functional model, content history can then be depicted as a direct acyclic graph
- parallel changes may or may not be compatible...

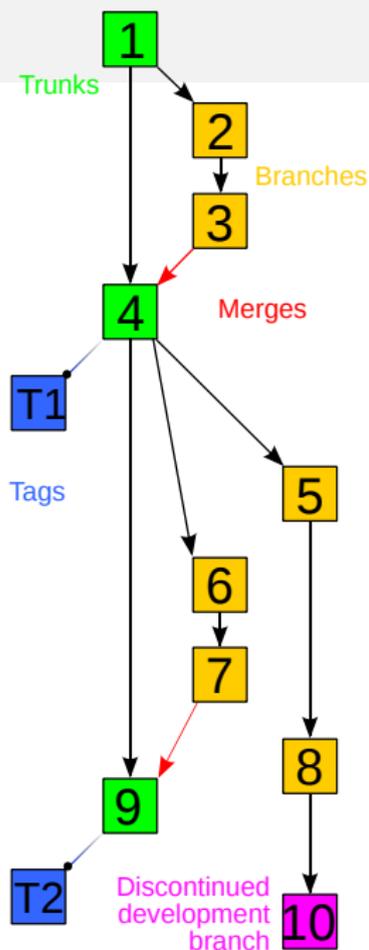
**conflict** the situation occurring when, upon a merge attempt, changes from involved branches cannot be reconciled

**solving** a conflict means applying extra changes to combine non (automatically) reconcilable changes or choose a subset of them



## Branching and merging (cont.)

- tag** (or **label**) a symbolic name attached to a particular revision in history
- head** (or **tip**) the (moving) tag always associated to the most recent commit; might be limited to a specific “special” branch, such as:
- trunk** (or **master**) the unique line of development which is not a branch
  - peculiar: treating a specific branch as special is not necessary for the idealized model to work



## Brief history of VCSs

- 1972 **SCCS** (Source Code Control System), commercial (AT&T) UNIX-es, part of the Single UNIX Specification; scope: file; modern clone (for compatibility only): **cssc**
- 1982 **RCS** (Revision Control System) GNU-based UNIX-es; scope: file; Free-er and generally considered more evolved than SCCS, currently maintained by the GNU Project
- 1990 **CVS** (Concurrent Version System), client-server paradigm; scope: set of files
- late 1990's **TeamWare**, **BitKeeper**; early attempt at distributed version control system; proprietary
- 2000 **Subversion** (SVN); client-server, addressing many defects of CVS
- 2001– Distributed VCS (**DVCS**) golden age: **GNU arch** (2001), **Darcs** (2002), **SVK** (2003), **Monotone** (2003), **Git** (2005), **Mercurial** (2005), **Bazaar** (2005) [more on this later...](#)

# Outline

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts
- 4 Revision Control System (RCS)**
- 5 Concurrent Versions System (CVS)
- 6 Subversion
- 7 Git

# Revision Control System (RCS)

- one of the oldest system (1982)
- typical of the commercial UNIX era
- scope: single file
  - ▶ although the repositories for several files can be stored in the same “shared” place
- repository
  - ▶ file,v where file is the original name of the checked in file
  - ▶ changes are stored as incremental reverse diffs
  - ▶ minimization of secondary storage: delta compression is triggered by deletion of intermediate revisions
- concurrency model:
  - ▶ pessimistic approach: one have to acquire explicit locks before making modification; by default working copies are read-only
  - ▶ as the working copy is shared among users, this enforced a rather heavy mutual exclusion discipline

## RCS — basic operations

`commit` `ci` FILE (without lock)  
`ci` -l FILE (with lock)

`checkout` `co` FILE (without lock)  
`co` -l FILE (with lock)

`diff` `rcsdiff` -rVERSION1 -rVERSION2 FILE

`history` `rlog` FILE

`acquire lock` `rcs` -l FILE

## Demo



# RCS — branching and merging operations

**branch** `ci -rVERSION FILE`

example: `ci -r2 foo.ml`

**branch checkout** `co -rVERSION FILE`

**merge** `rcsmerge -p -rVERSION1 -rVERSION2 FILE > RESULT`

- performs a 3-way diff (a-la `diff3`) among old (common) `VERSION1`, and the two new versions: `VERSION2` and the current state of `FILE`

## Example

```
rcsmerge -p -r1 -r3 foo.ml > foo.ml.new
```

merges the differences among branch 1 and 3 of `foo.ml`, with differences among branch 1 and the current version of `foo.ml`; save the result to `foo.ml.new`

## Demo

# Outline

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)**
- 6 Subversion
- 7 Git

# Concurrent Versions System (CVS)

- a significant (r)evolution in the history of VCS
- designed to address the (too) constraining mutual exclusion discipline enforced by RCS (hence the emphasis on *concurrent*)
- client-server model
  - ▶ enforce decoupling of repository and working copy
  - ▶ several working copies exist—generally one for each developer—and can be acted upon independently
  - ▶ commands and processes to:
    - ★ “push” local changes from working copies to the repository
    - ★ “pull” changes (made by others) from the repository and merge it with the local (uncommitted) changes
    - ★ deal with conflicts and try to avoid they hit the repository
  - ▶ note: the repository is used as the orchestrator and as the sole data storage
    - ★ “local commits” are not possible
    - ★ disconnected operations are heavily limited

## CVS — some details

- scope: a project (i.e., a tree of file and directories)
- built as a set of scripts on top of RCS
  - ▶ each file has its own ,v file stored in the repository
  - ▶ each file has its own set of RCS versions (1.1, 2.3, 1.1.2.4, etc.)
  - ▶ a very cool hack, but still a hack
- the repository can be either local (i.e., on the same machine of the working copy) or remote (accessible through the network; common scenario)
- concurrency model:
  - ▶ optimistic approach: working copies can be acted upon by default; given that working copies are independent from each other, work is concurrent by default
  - ▶ conflicts are noticed upon commit and must be solved locally (i.e., commits prior to merges are forbidden)
  - ▶ explicit locks are permitted via dedicated actions
- curiosity: one of the first popular UNIX commands relying on sub-commands

# CVS — basic operations

**repository setup** export CVSROOT=SOME/DIR  
cvs init

**create a project** cd PROJECT-DIR  
cvs import -d NAME VENDOR-NAME RELEASE-NAME  
example: cvs import -d coolhack zack initial

**checkout** cvs checkout NAME

**status** get information about the status of the working copy  
with respect to (the last contact with) the repository

**commit** cvs commit [FILE...]  
example: cvs commit -m 'fix segmentation fault' foo.c

**update** (merge changes from the repository in the local copy)  
cvs update -d

## CVS — basic operations (cont.)

**history** cvs log

**diff** cvs diff [FILE...] (among working copy and last update)

**diff** cvs diff -rVERSION1 -rVERSION2 [FILE...]

**remove file** cvs rm FILE (*schedule* removal; needs commit)

**add file** cvs add FILE (*schedule* addition; needs commit)

## Demo

# CVS — branching and merging operations

tag `cvstag TAG-NAME`

## branch

```
cvstag -b BRANCH-NAME  
# work on trunk continues  
cvsup -r BRANCH-NAME  
# work on branch ...  
cvsup -A  
# go back working on trunk
```

## merge

```
cvsup -j BRANCH-NAME # merge changes (wc)  
cvsc
```

## Demo

## CVS — discussion

revolutionary for its time

affected by severe limitations nonetheless:

- revisions are per file, i.e., there is no knowledge of repository-wide revisions (they can be emulated by tags, but...)
- no knowledge of several common file-system features (e.g., attributes, symlink, file move)
- files are considered textual by default; ad-hoc and limited support for binary content
- branch operations are expensive (underlying assumption: most of the work happens in trunk)
- commits are not atomic operations
- very little support for disconnected operations (e.g., painful when you're hacking on a plane)

# Outline

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)
- 6 Subversion**
- 7 Git

## Subversion (SVN) — context

Started in 2000 to overcome CVS limitations.

Historical context:

*In the world of open source software, the Concurrent Version System (CVS) has long been the tool of choice for version control. And rightly so. CVS itself is free software, and its non-restrictive modus operandi and support for networked operation—which allow dozens of geographically dispersed programmers to share their work—fits the collaborative nature of the open-source world very well. CVS and its semi-chaotic development model have become cornerstones of open-source.*

— Collins-Sussman

# Subversion (SVN)

- same paradigm of CVS: client-server + independent working copies
- features (i.e., “bug fixes” w.r.t. CVS):
  - ▶ atomic commits
  - ▶ tracking (in the history!) of file-system level operations (copy, move, remove, etc.)
  - ▶ global versioning (rather than per-file versioning)
  - ▶ support for symlinks and (some) file-system level metadata
  - ▶ cheap (server-side) branches
  - ▶ some (but not much) support for disconnected operations (most notably: diff among working copy and HEAD)

## SVN — basic operations

To increase adoption chances within CVS circles, SVN command line interface has been designed to be as compatible as possible to CVS interface. The strategy has worked very well! Most commands work as in CVS.

**basic operations** `svn checkout`, `svn status`, `svn add`, `svn remove`,  
`svn commit`, `svn diff`, `svn log`, `svn update`, ...

**repository setup** `svnadmin create REPO-PATH`

**create a project** (actually: create a directory in a repository)

`svn checkout REPO-PATH`

`svn mkdir DIR`

`svn commit`

## Demo

## SVN — branching and merging

- branches in SVN are part of the versioned tree
- to create a branch, one makes a copy of an existing directory to a new path
  - ▶ development can then proceed independently in the original and new directory
  - ▶ branches are cheap (“shallow copies”) on the repository side (but not on the client side)
  - ▶ partial checkouts are possible, to avoid forcing clients to keep all branches at once
- tags work in the same way; the only difference is that a tag doesn't (i.e., shouldn't) be committed to
- specific path conventions are suggested. . .

# SVN — path conventions

```
project/  
  trunk/  
    main.ml  
    module-foo.ml  
    quux/  
    ...  
  branches/  
    feature1/  
      main.ml  
      module-foo.ml  
      quux/  
      ...  
    feature2/  
      main.ml  
      module-foo.ml  
      quux/  
      ...  
  tags/  
    1.0-rc1/  
      main.ml  
      module-foo.ml  
      quux/  
      ...  
    1.0/  
      main.ml  
      module-foo.ml  
      quux/  
      ...  
    1.1/  
      main.ml  
      module-foo.ml  
      quux/  
      ...
```

Recommended path conventions.

No strict requirement (but still recommended for uniformity across projects).

## SVN — branching and merging example

- branching off trunk

```
$ svn cp -m "Creating branch for feature A" \  
  /trunk/component1 /branches/zack-component1-featureA  
$ svn log -v /branches/zack-component1-featureA  
# prints revision number, say 123
```

- updating the branch

```
$ svn merge -r 123:HEAD /trunk/component1 .  
# test new code from master  
$ svn commit -m'sync with master'  
# Revision 256.
```

- merging into trunk

```
$ cd /trunk/component1  
$ svn merge -r 123:HEAD /branches/zack-component1-featureA  
$ svn commit -m'integrate feature A by Zack'  
$
```

## SVN — branching and merging example (better)

```
$ cd /trunk/component1
$ svn merge --reintegrate REPO-PATH/branches/zack-component1
$ svn commit -m'integrate feature A by Zack'
$
```

- keep track of merged revisions via `svn:mergeinfo` property

## Demo

# Outline

- 1 Configuration management
- 2 diff & patch
- 3 Version control concepts
- 4 Revision Control System (RCS)
- 5 Concurrent Versions System (CVS)
- 6 Subversion
- 7 Git**

We will follow the excellent tutorial:

*Git 101*

Scott Chacon

GitHub

<https://www.slideshare.net/chacon/git-101-presentation>

# References



Aiello, Sachs

*Configuration Management Best Practices: Practical Methods that Work in the Real World*

Addison-Wesley, 1st edition, 2010



Chacon

*Pro Git*

Apress, 2005. <http://progit.org/book/>