IN323 Software Engineering
Software Configuration Management with Subversion

Christophe Garion

DMIA – ISAE

License CC BY-NC-SA 3.0

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmite) and to Remix (adapt) this work under the following conditions:

Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial – You may not use this work for commercial purposes.

Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See `http://creativecommons.org/licenses/by-nc-sa/3.0/`.

First problem: manage change

Problem

How to « remember » changes made to an application?

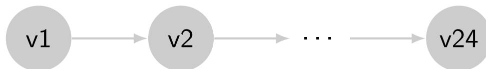
Why?

- « go back » to remove bad changes
- be able to propose a stable version of the application when continuing its development
- be able to propose an older version of the application by starting back from an older version
- ...

First problem: discussion...

Idea

Have an history of the application using **versions**.



But...

- which types of file can you manage?
  ➥ **source codes**, configuration and build files
- how to manage history?
  - create directories with version numbers et copy **all** necessary files at each time
    ➥ not usable!
  - add version numbers on file names
    ➥ how to guarantee numbers coherence?
    ➥ not possible for instance in Java for source files!

Second problem: share code

Problem

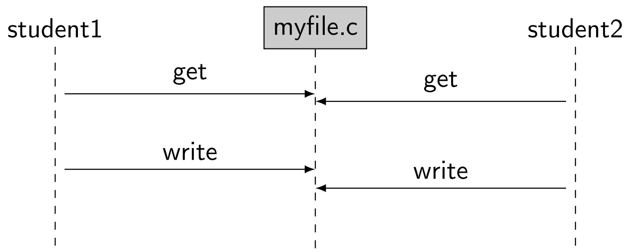
How to allow several developers/designers to share code/documents to work **at the same time**?

Why?

- to allow a team to work easily on the same project, particularly on the source code
- to manage **conflicts** when two persons work on the same document

Second problem: students solutions...

1. sharing documents by **email**
   - ➥ completely unmanageable
2. **open rights** on one student's account
   - ➥ unsecured...
   - ➥ cannot manage conflicts



Second problem: students solutions. . .

3. use Dropbox or equivalent system
   - not really a solution
   - limited history
   - no diffs, no commit messages
   - limited conflict management

Outline

1. **Revision control**
2. Subversion
3. Process

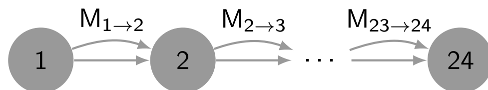
Revision control

To solve the previous problems, we will use a **revision control** system.

A revision control software (RCS) allows to easily manage:

- changes made on the project files
- multiple users working on the project
- branches to develop experimental features or correct bugs without changing the application main version

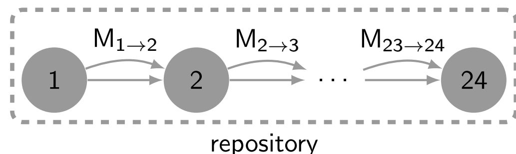
Revision control: concepts



Documents evolutions are represented by **revisions**.
Revisions are often denoted by natural numbers: revision 1, revision 2 etc.

To go from a revision to another, **changesets** are applied to the project files.
A changeset can change **several files**: it represents the transition from a "coherent" state of the project to another "coherent" state.
Revision control softwares **only keep changesets**.

Revision control: concepts

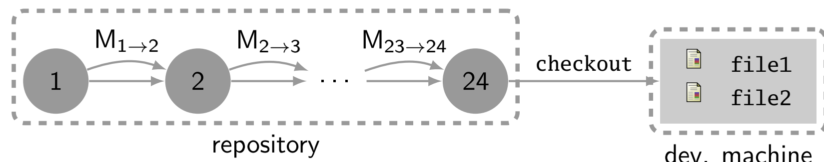


Files managed by the RCS are kept in a **repository**.

The repository, like a DBMS, respects the **ACID** properties to ensure the atomicity and coherence of changes:

- **atomicity** of **changesets**
- **consistency**
- **isolation** from other changes
- **durability**

Revision control: concepts

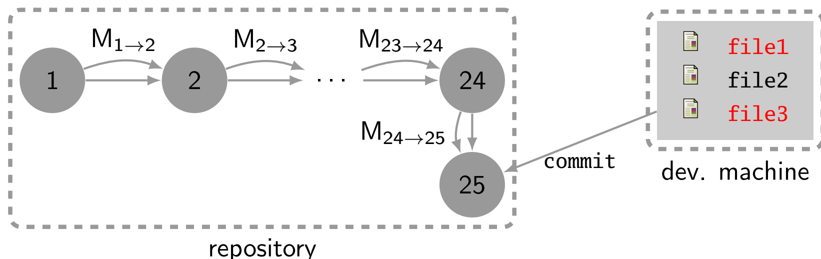


Files managed by the RCS are kept in a **repository**.

To work on the project, you have to make a **local copy** of the repository. You will obtain by default the last revision of the repository, buy you can choose.

This operation is called a **checkout**.

Revision control: concepts



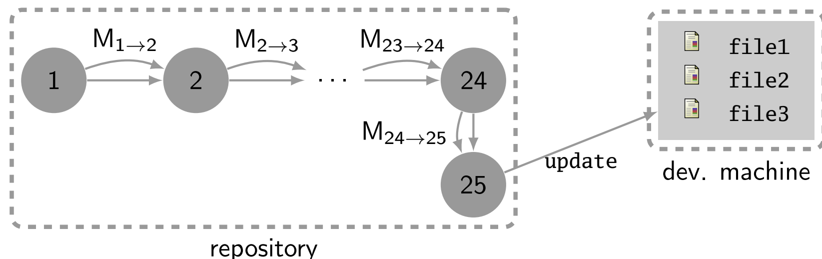
When you have made the desired changes on the files, you can submit your changes to the repository.

This operation is called a **commit**.

N.B.

Conflicts may arise during this operation!

Revision control: concepts



When you want your local copy to be up-to-date with the repository, you make an **update** operation.

N.B.

Conflicts may arise during this operation!

Outline

1 Revision control

2 **Subversion**
- Basic usage
- Conflicts management
- Viewing logs and changes
- Branches

3 Process

Subversion

The RCS we will use at ISAE is Subversion, a free software available on multiple platforms.

The Apache Software Foundation (2013).
Apache Subversion.
http://subversion.apache.org/.

Collins-Sussman, B., B. W. Fitzpatrick, and C. Michael Pilato (2004).
Version control with Subversion.
O'Reilly.
http://svnbook.red-bean.com/.

Mason, Mike (2006).
Pragmatic Version Control Using Subversion.
2nd edition.
Pragmatic Programmers.

Outline

1 Revision control

2 **Subversion**
- Basic usage
- Conflicts management
- Viewing logs and changes
- Branches

3 Process

Usual commands

~alice/ – rev. 1

file1.txt
Coucou

REPOSITORY – rev. 1

file1.txt
Coucou

~bob/ – rev. 1

file1.txt
Coucou

Alice copies the repository (idem for Bob):

shell (alice)

```
[alice@computer]~ $ svn checkout URL_REPOSITORY
A    scm/alice/file1.txt
Checked out revision 1.
```

Usual commands

~alice/ – rev. 1	REPOSITORY – rev. 1	~bob/ – rev. 1
file1.txt Coucou	**file1.txt** Coucou	**file1.txt** Coucou
file2.txt Hello		

Alice creates a new file.

Usual commands

~alice/ – rev. 1

file1.txt

Coucou

file2.txt

Hello

REPOSITORY – rev. 1

file1.txt

Coucou

~bob/ – rev. 1

file1.txt

Coucou

She can verify that her local copy is not identical to the repository.

shell (alice)

```
[alice@computer]~ $ svn status
?       file2.txt
```

Usual commands

~alice/ – rev. 2

file1.txt

Coucou

file2.txt

Hello

REPOSITORY – rev. 2

file1.txt

Coucou

file2.txt

Hello

~bob/ – rev. 1

file1.txt

Coucou

She can then **add** the file and **submit** it to the repository.

shell (alice)

```
[alice@computer]~ $ svn add file2.txt
A         file2.txt
Adding         file2.txt
[alice@computer]~ $ svn commit -m "adding file2.txt"
Transmitting file data .
Committed revision 2.
```

Usual commands

~alice/ – rev. 2

file1.txt

Coucou

file2.txt

Hello

REPOSITORY – rev. 2

file1.txt

Coucou

file2.txt

Hello

~bob/ – rev. 2

file1.txt

Coucou

file2.txt

Hello

Bob can update his local copy.

shell (bob)

```
[bob@computer]~ $ svn update
Updating '.':
A    file2.txt
Updated to revision 2.
```

Usual commands

~alice/ – rev. 2	REPOSITORY – rev. 3	~bob/ – rev. 3
file1.txt Coucou	**file1.txt** Coucou	**file1.txt** Coucou
file2.txt Hello	**file2.txt** Hello	**file2.txt** Hello
	file3.txt c'est moi	**file3.txt** c'est moi

Bob adds a file and submit it.

Usual commands

~alice/ – rev. 4

file1.txt

Coucou

file2.txt

Bonjour

REPOSITORY – rev. 4

file1.txt

Coucou

file2.txt

Bonjour

file3.txt

c'est moi

~bob/ – rev. 3

file1.txt

Coucou

file2.txt

Hello

file3.txt

c'est moi

Alice modifies `file2.txt` and submit it.

shell (alice)

```
[alice@computer]~ $ svn commit -m "changing Hello in file2.txt"
Sending        file2.txt
Transmitting file data .
Committed revision 4.
```

Outline

1. Revision control
2. **Subversion**
   - Basic usage
   - Conflicts management
   - Viewing logs and changes
   - Branches
3. Process

Conflicts

~alice/ – rev. 1

file1.txt
Coucou

REPOSITORY – rev. 1

file1.txt
Coucou

~bob/ – rev. 1

file1.txt
Coucou

Conflicts

~alice/ – rev. 2	REPOSITORY – rev. 2	~bob/ – rev. 1
file1.txt	**file1.txt**	**file1.txt**
Bonjour	Bonjour	Coucou

Alice modifies `file1.txt` and commits her version.

Conflicts

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 2

file1.txt

Bonjour

~bob/ - rev. 1

file1.txt

Hello

Bob modifies `file1.txt` and wants to commit his version.

shell (bob)

```
[bob@computer]~ $ svn commit -m "changing Coucou to Hello in file1.txt"
Sending        file1.txt
svn: E155011: Commit failed (details follow):
svn: E155011: File '/home/tof/Cours/IN323/bob/file1.txt' is out of date
svn: E160028: File '/file1.txt' is out of date
```

Conflicts

~alice/ – rev. 2

file1.txt

Bonjour

REPOSITORY – rev. 2

file1.txt

Bonjour

~bob/ – rev. 1

file1.txt

Hello

Bob can update his local copy.

shell (bob)

```
[bob@computer]~ $ svn update
Updating '.':
C    file1.txt
Updated to revision 2.
Summary of conflicts:
  Text conflicts: 1
```

Conflicts

~alice/ - rev. 2

file1.txt

Bonjour

REPOSITORY - rev. 2

file1.txt

Bonjour

~bob/ - rev. 1

file1.txt

Hello

Bob chooses to edit (e) this file. He obtains a **temporary** file containing both his version and the repository version.

file1.txt.tmp

```
<<<<<<< .mine
Hello
=======
Bonjour
>>>>>>> .r2
```

Conflicts

~alice/ - rev. 2	REPOSITORY - rev. 2	~bob/ - rev. 2
file1.txt	**file1.txt**	**file1.txt**
Bonjour	Bonjour	Hello

Bob modifies the temporary file to keep his version. He choose to mark the conflict as **resolved**.

shell (bob)

```
[bob@computer]~ $ svn resolved file1.txt
Resolved conflicted state of 'file1.txt'
```

Conflicts

~alice/ - rev. 2	REPOSITORY - rev. 3	~bob/ - rev. 3
file1.txt	**file1.txt**	**file1.txt**
Bonjour	Hello	Hello

Bob can then commit its changes to the repository.

shell (bob)

```
[bob@computer]~ $ svn commit -m "changing Coucou to Hello in file1.txt"
Sending        file1.txt
Transmitting file data .
Committed revision 3.
```

Conflicts

~alice/ – rev. 2

file1.txt

Guten Tag

REPOSITORY – rev. 3

file1.txt

Hello

~bob/ – rev. 3

file1.txt

Hello

Alice modifies her local copy of `file1.txt`. She updates her copy, discovers the conflict and chooses to **postpone** (`p`) the conflict management.

shell Alice

```
[alice@computer]~ $ svn update
Updating '.':
C    file1.txt
Updated to revision 3.
Summary of conflicts:
  Text conflicts: 1
```

Conflicts

~alice/ - rev. 2

```
file1.txt
<<<<<<< .mine
Guten Tag

=======
Hello
>>>>>>> .r3
```

REPOSITORY - rev. 3

```
file1.txt
Hello
```

~bob/ - rev. 3

```
file1.txt
Hello
```

Subversion has created several files corresponding to different versions of `file1.txt`: one for revision 2, one for revision 3 and the local copy (`file1.txt.mine`).
`file1.txt` has the same syntax as presented previously.

shell Alice

```
[alice@computer]~ $ ls
file1.txt
file1.txt.mine
file1.txt.r2
file1.txt.r3
```

Conflicts

~alice/ – rev. 2

```
file1.txt
---------
Guten Tag
```

REPOSITORY – rev. 3

```
file1.txt
---------
Hello
```

~bob/ – rev. 3

```
file1.txt
---------
Hello
```

Alice can choose the file she wants or modify `file1.txt`. She can specify to Subversion that she wants to keep her local copy to solve the conflict. She has to commit her changes after (not done here!).

shell Alice

```
[alice@computer]~ $ svn resolve --accept mine-full file1.txt
Resolved conflicted state of 'file1.txt'
```

Outline

1. Revision control
2. Subversion
   - Basic usage
   - Conflicts management
   - Viewing logs and changes
   - Branches
3. Process

Logs

Obtaining the commit messages for a specific file:

shell (bob)

```
[bob@computer]~ $ svn log file1.txt
------------------------------------------------------------------------
r3 | bob | 2014-07-09 11:09:05 +0200 (Wed, 09 Jul 2014) | 1 line

changing Coucou to Hello in file1.txt
------------------------------------------------------------------------
r2 | alice | 2014-07-09 11:09:01 +0200 (Wed, 09 Jul 2014) | 1 line

changing Coucou to Bonjour in file1.txt
------------------------------------------------------------------------
r1 | tof | 2014-07-09 11:08:58 +0200 (Wed, 09 Jul 2014) | 1 line

initial import of file1.txt
------------------------------------------------------------------------
```

Diffs

Obtaining the set of changes between two revisions for a file:

shell (bob)

```
[bob@computer]~ $ svn diff -r2:3 file1.txt
Index: file1.txt
===================================================================
--- file1.txt   (revision 2)
+++ file1.txt   (revision 3)
@@ -1 +1 @@
-Bonjour
+Hello
```

N.B.

The result of **diff** is called a **patch**: those are the changes to apply on `file1.txt` to go from revision 2 to revision 3.

Outline

1. Revision control
2. **Subversion**
   - Basic usage
   - Conflicts management
   - Viewing logs and changes
   - Branches
3. Process

What are branches useful for?

Definition (branch)

A **branch** is a development line that exists independently of other lines.

Branches in Subversion allow to:

- create multiple versions of the same product
- create a branch for debugging
- create a branch for experimental features
- mix and match different lines of development
- maintain a release branch for production code
- ...

What are tags useful for?

Definition (tag)

A **tag** is a symbolic name for a set files.

Tags in Subversion allow to:

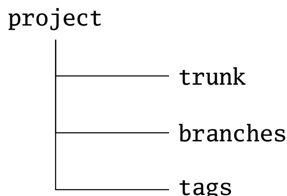
- have a symbolic name for a set of files, each with a particular revision number
- put milestones in your project
- eventually mix revision numbers

Branches and tags: example

- release version 1.0 of your project:
  - create a tag `REL-1.0`
  - create a branch `RB-1.0` to eventually work on this release
- fix the bug number 3035:
  - create a branch `BUG-3035`
  - create a tag `PRE-3035`
  - after correcting the bug, create a tag `POST-3035`
- experiment with a new GUI:
  - create a branch `TRY-new-GUI`
- ...

Organizing your project

A classical repository organization for Subversion projects:



- `trunk`: contains the main development line
- `branches`: contains branches ☺
- `tags`: contains tags ☺

Creating the necessary directories

Bob wants to create a release branch for his project. He needs first to create the `branches` directory:

shell (bob)

```
[bob@computer]~ $ svn mkdir URL/branches -m "creating branches dir."

Committed revision 4.
```

N.B.

Use **`svn mkdir`** with URL to create directories to be managed by subversion, it is easier and faster.

Creating a branch

Creating branches or tags in Subversion is just copying directories!

Bob wants to create a branch using the trunk main development line:

shell (bob)

```
[bob@computer]~ $ svn copy -m "creating branch for DEV 1.0" URL/trunk
    URL/branches/DEV-1.0

Committed revision 5.
```

Now he can checkout the branch as usual:

shell (bob)

```
[bob@computer]~ $ svn checkout URL/branches/DEV-1.0 dev-1.0
A    dev-1.0/file1.txt
Checked out revision 5.
```

Switching to a branch

Bob can also switch to a branch from another one (here from `trunk` for instance):

shell (bob)

```
[bob@computer]~ $ svn switch URL/branches/DEV-1.0
At revision 5.
```

All changes made here are taken into account in the `DEV-1.0` branch.

Merging branches

Suppose now that Bob changed `file1.txt` in the `DEV-1.0` branch and wants to merge its changes into the main development line (execute this in a working copy of the `trunk`):

shell (bob)

```
[bob@computer]~ $ svn merge URL/branches/DEV-1.0
--- Merging r5 through r6 into '.':
U    file1.txt
[bob@computer]~ $ svn commit -m "merging branch DEV-1.0 into trunk"
--- Recording mergeinfo for merge of r5 through r6 into '.':
 U   .
Sending        .
Sending        file1.txt
Transmitting file data .
Committed revision 7.
```

N.B.

Beware of conflicts, update your working copy before merging!

Outline

1. Revision control
2. Subversion
3. **Process**

How to use Subversion?

checkout

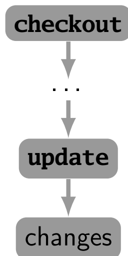
...

update

Verifications

- code compiles
- tests passed

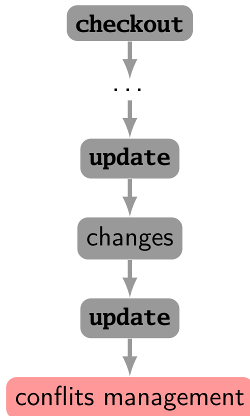
How to use Subversion?



Verifications

- code compiles
- tests passed

How to use Subversion?



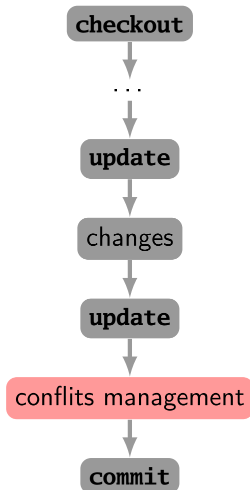
Verifications

- ➥ code compiles
- ➥ tests passed

Conflicts management

Working with other dev. needed!

How to use Subversion?



Message

Use **explicit commit** message!
Describe **what** is concerned with the commit, not **how** you achieve the modifications (that is the function of the diff/patch).

When committing

- each commit must be **coherent**: your application should compile and work normally
- commit **each time** you add/correct a single functionality. Do not commit big changesets
- to be able to do regression tests, you must have small changesets