LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #4

Version Control

repositories, sandboxes and subversion

Code Development in Science

- is still often characterised by
  - done by a single PhD student alone → "heroic codes" (Heiner Igel)
  - suffers from a lack of professionalism, i.e. it is unplanned, unmanaged, uncoordinated, untested, un...
  - and gives unreliable and unrepeatable? results
- leads to the reliability crisis of computational science
- Problem #1: There is (was?) no culture honouring code development or the quality of scientific codes
- Problem #2: Science students are not being sufficiently educated

Software Carpentry

- Problem #3: Informatics is Computer Science
  - but a Geophysicist wants to do research in Geophysics not Informatics!
- That's where Software Carpentry comes into play
  - its idea is to teach science students the

    10% of modern software engineering that will handle 90% of their needs
  - to make them more productive
  - help to produce reliable and repeatable results
- You'd never let some one work in a chemistry lab without proper instructions

Acknowledgements

- 75% of the following slides are based on the course on Software Carpentry developed by Brent Gorda and Greg Wilson
- The course is publicly available at `http://swc.scipy.org/`
- and was supported by the Python Software Foundation and the University of Toronto
- Should take a look at the material before your master's thesis!

License

applicable to slides of part 4 of the course

Separates the men from the boys

- Four things distinguish professional programmers from amateurs:
  - Using a version control system
  - Automating repetitive tasks
  - Systematic testing
  - Using debugging aids rather than print statements
- We introduce the first of these now
- Later part will deal with aspects of automation
- It's not about coolness, but productivity

Version Control

Version control is as fundamental to programming as accurate notes about lab procedures are to experimental science.

It's what let's you say, "This is how I produced these results," rather than, "Um, I think we were using the new algorithm for that graph — I mean, the old new algorithm, not the new new algorithm."

(Greg Wilson in *Where's the Real Bottleneck in Scientific Computing?", American Scientist, vol. 94, Jan/Feb 2006*)

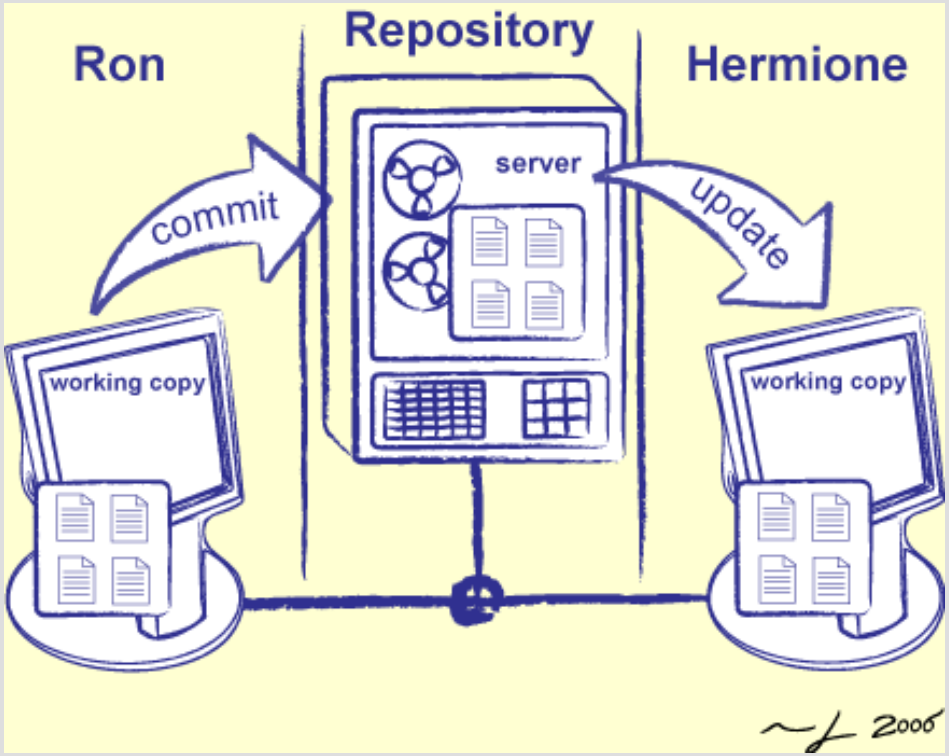
You may fall asleep now, if . . .

- You know what a repository is
- You know how to commit changes
- You know how to merge conflicts
- You know how to roll back a set of changes
- You know what a branch is

Problem #1: Collaboration

- What if two or more people want to edit the same file at the same time?
- Option 1: make them take turns
  - But then only one person can be working at any time
  - And how do you enforce the rule?
- Option 2: patch up differences afterwards
  - Requires a lot of re-working
  - Stuff always gets lost

Solution: Version Control



- The right solution is to use a version control system
- Keep the master copy of the file in a central repository
- Each author edits a working copy (= sandbox)
- When they're ready to share their changes, they commit them to the repository
- Other people can then do an update to get those changes

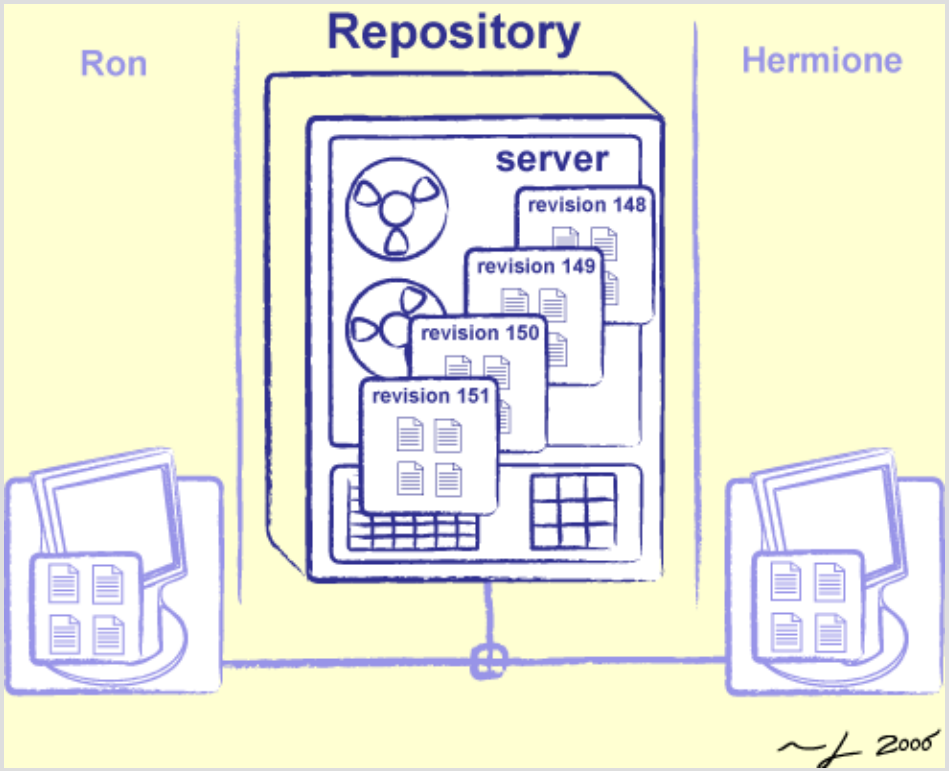
Solution: Version Control (cont.)

- This is also a good way for one person to manage files on multiple machines
  - Keep one working copy on your personal laptop, the lab machine, and the departmental server
  - No more mailing yourself files, or carrying around a USB drive (and forgetting to copy things onto it)
- Line of defence against data loss
  - `rm -rf * .dat` will only delete sandbox,
  - but not files commited to repository
- Not only good for program files, but also for writing a thesis or paper

Problem #2: Undoing Changes

- Often want to undo changes to a file
  - Start work, realise it's the wrong approach, want to get back to starting point
  - Like "undo" in an editor ...
  - ... but keep the whole history of every file, forever
- Also want to be able to see who changed what, when
  - The best way to find out how something works is often to ask the person who wrote it

Solution: Version Control (Again)



- Have the version control system keep old revisions of files
- And have it record who made the change and when
- Authors can then roll back to a particular revision or time

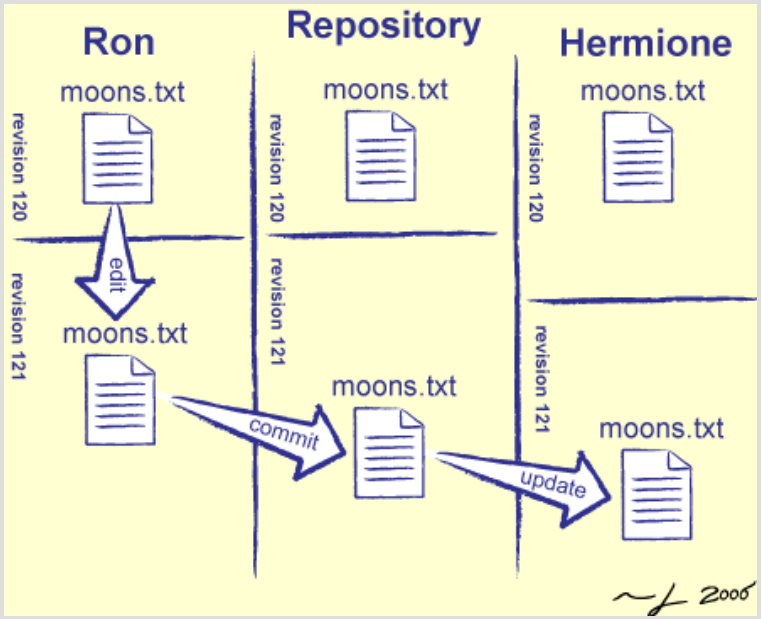
Which Version Control System?

- Many systems are available commercially or freely
- Wikipedia under Comparison of revision control software lists 40 systems (as of revision 10:25, 9 July 2009)
- Subversion is:
  - Open source
  - Reliable
  - Well documented
  - Nicely configurable for demands of larger projects
  - Widely used in Scientific projects
- Also very popular is Git
  - differs in being a distributed VCS
  - see also hosters such as GitHub, BitBucket, or GitLab@LRZ

Subversion

- Subversion developed from 2000 onward . . .
- . . . as a workalike replacement of the older CVS
- Feels the same, but eliminates CVS's major weaknesses
- More info including online version of *O'Reilly book* at `http://subversion.tigris.org/`
- Used to manage slides/material for this course
- Will be used in the pratical part of this course

Basic Usage

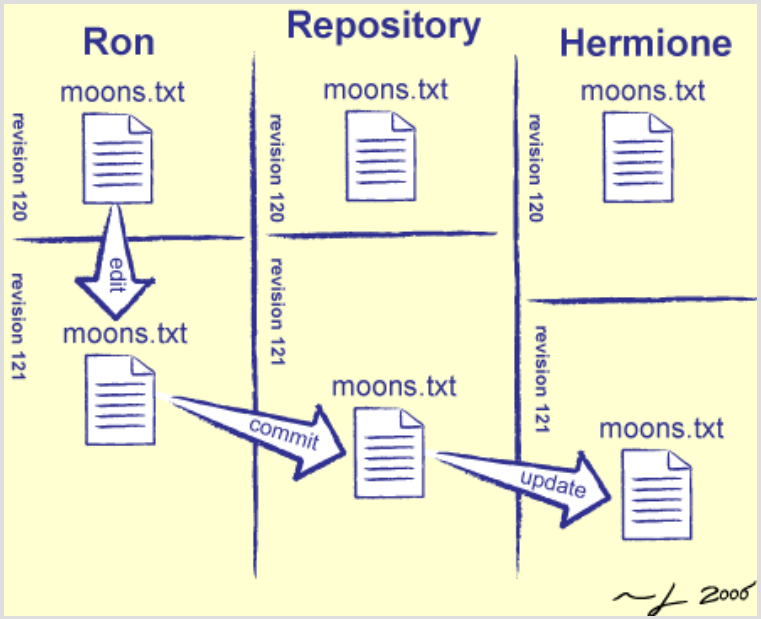


- Ron and Hermione each has a working copy of the solarsystem project repository
- Ron wants to add some information about Jupiter's moons
  - Runs svn update to synchronise his working copy with the repository
  - Goes into the jupiter directory and creates moons.txt

moons.txt

Name	Orbital Radius	Orbital Period	Mass	Radius
Io	421.6	1.769138	893.2	1821.6
Europa	670.9	3.551181	480.0	1560.8
Ganymede	1070.4	7.154553	1481.9	2631.2
Callisto	1882.7	16.689018	1075.9	2410.3

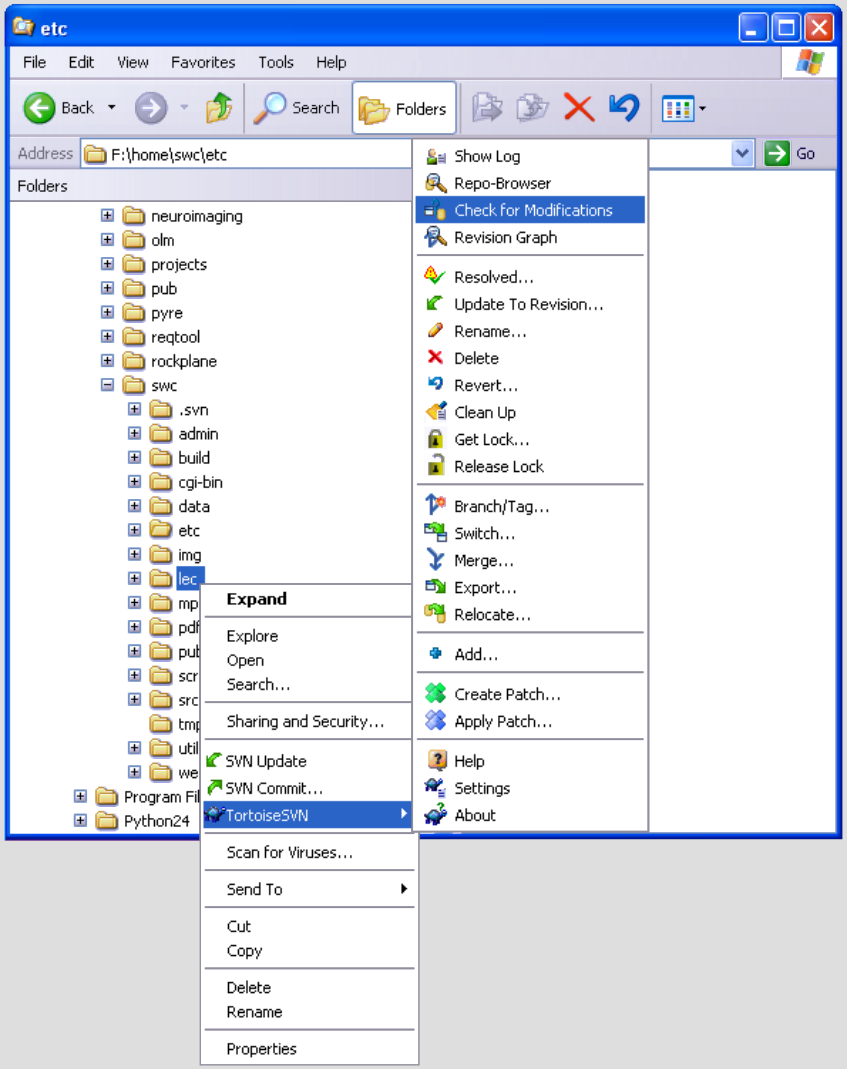
Basic Usage (cont.)



- Ron wants to add some information about Jupiter's moons
  - Runs svn add moons.txt to bring it to Subversion's notice
  - Runs svn commit to save his changes in the repository
- That afternoon, Hermione runs svn update on her working copy
  - Subversion sends her Ron's changes

How To Do It

- Subversion can be used via the command line / shell (of course ;-)
- Alternatively several GUIs and plugins for file browsers exist, e.g.
  - RapidSVN (for Windows, Linux, and Mac)
  - TortoiseSVN (Plugin for Windows Explorer)
  - kdesvn (standalone front-end)
  - ...
- Can also use it from within Emacs



Resolving Conflicts

- Back to the problem of conflicting edits (or, more simply, conflicts)
- Option 1: only allow one person to have a writeable copy at any time
  - This is called pessimistic concurrency
  - Used in Microsoft Visual SourceSafe
- Option 2: let people edit; resolve conflicts afterward by merging files
  - Called optimistic concurrency
  - "It's easier to get forgiveness than permission"
  - Most modern systems (including Subversion) do this

Example: Resolving

- Ron and Hermione are both synchronised with version 151 of the repository
- Ron edits moons.txt and commits his changes to create version 152

Name	Orbital Radius	Orbital Period	Mass	Radius
Io	421.6	1.769138	893.2	1821.6
Europa	670.9	3.551181	480.0	1560.8
Ganymede	1070.4	7.154553	1481.9	2631.2
Callisto	1882.7	16.689018	1075.9	2410.3
Amalthea	181.4	0.498179	0.075	131 x 73 x 67
Himalia	11460	250.5662	0.095	85
Elara	11740	259.6528	0.008	40

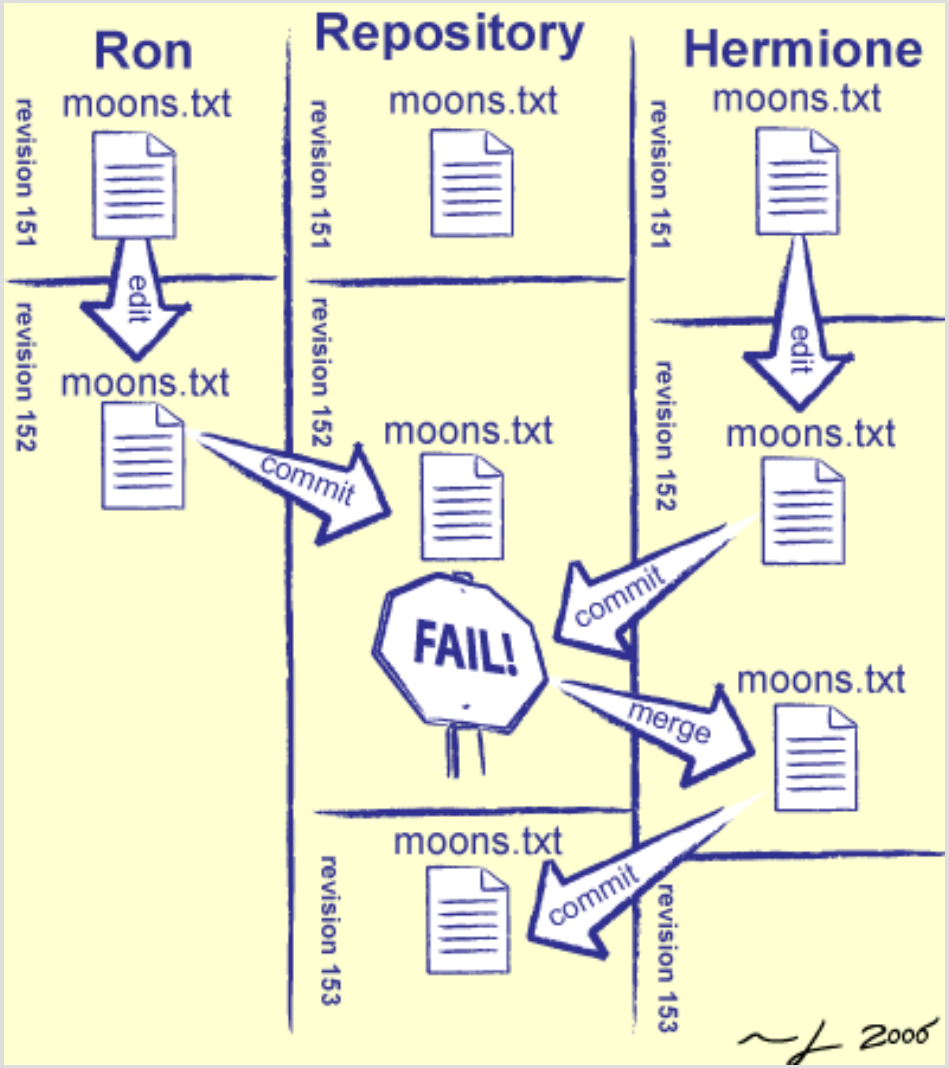
Example: Resolving (cont.)

- Simultaneously, Hermione edits her copy of moons.txt

Name	Orbital Radius (10**3 km)	Orbital Period (days)	Mass (10**20 kg)	Radius (km)
Io	421.6	1.769138	893.2	1821.6
Europa	670.9	3.551181	480.0	1560.8
Ganymede	1070.4	7.154553	1481.9	2631.2
Callisto	1882.7	16.689018	1075.9	2410.3
Amalthea	181.4	0.498179	0.075	131
Himalia	11460	250.5662	0.095	85
Elara	11740	259.6528	0.008	40
Pasiphae	23620	743.6	0.003	18
Sinope	23940	758.9	0.0008	14
Lysithea	11720	259.22	0.0008	12

- When she tries to commit, Subversion tells her there's a conflict

Example: Resolving (cont.)



- A race condition: two or more would-be writers racing to get their changes in first
- Hermione lost: She must now
  - incorporate Ron's changes into her version
  - manually resolve all possible conflicts
- Afterwards she can commit her new version

Example: Resolving (cont.)

- Hermione updates her sandbox and subversion puts Hermione's changes and Ron's in moons.txt
- To help it adds conflict markers to show where they overlapped

```
Name            Orbital Radius  Orbital Period  Mass            Radius
                (10**3 km)      (days)          (10**20 kg)     (km)
Io              421.6           1.769138        893.2           1821.6
Europa          670.9           3.551181        480.0           1560.8
Ganymede        1070.4          7.154553        1481.9          2631.2
Callisto        1882.7          16.689018       1075.9          2410.3
<<<<<<< .mine
Amalthea        181.4           0.498179        0.075           131
Himalia         11460           250.5662        0.095           85
Elara           11740           259.6528        0.008           40
Pasiphae        23620           743.6           0.003           18
Sinope          23940           758.9           0.0008          14
Lysithea        11720           259.22          0.0008          12
=======
Amalthea        181.4           0.498179        0.075           131 x 73 x 67
Himalia         11460           250.5662        0.095           85
Elara           11740           259.6528        0.008           40
>>>>>>> .r152
```

Example: Resolving (cont.)

- Subversion does not understand file's contents, of course. Does the best it can to sensibly arrange differences
- The conflict markers have the following meaning

```
<<<<<<< shows the start of the section from the first file
======= divides sections
>>>>>>> shows the end of the section from the second file
```

- Subversion also creates:
  - moons.txt.mine: contains Hermione's changes
  - moons.txt.151: the version Hermione based her changes on
  - moons.txt.152: the most recent version of the file in the repository

Example: Resolving (cont.)

- At this point, Hermione can:
  - Run `svn revert moons.txt` to throw away her changes
  - Copy one of the three temporary files on top of moons.txt
  - Edit moons.txt to remove the conflict markers
- Once she's done, she runs:
  - `svn resolved moons.txt` to let Subversion know she's done
  - this will also remove the auxiliary files
  - `svn commit` to commit her changes
    $\longrightarrow$ creates version 153 of the repository

Starvation

- But what happens if Ginny commits another set of changes while Hermione is resolving?
  - And then Harry commits yet another set?
- Starvation: Hermione never gets a turn because someone else always gets there first
- This is a management problem, not a technical one
  - Break the file(s) up into smaller pieces
  - Give people clearer responsibilities
  - The version control system is trying to tell you that people on your team are working at cross purposes

Binary Files

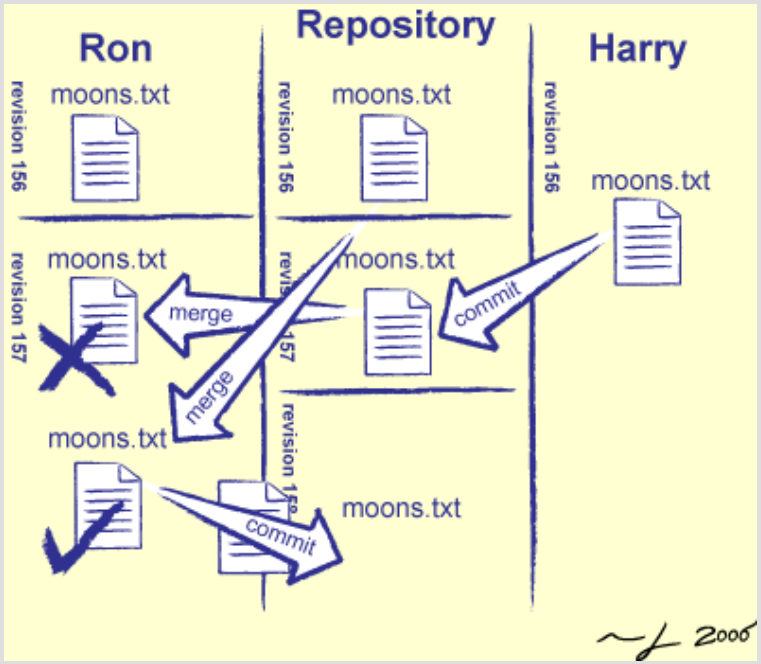
- Subversion can only merge conflicts in text files
- Source code, HTML, LaTeX– basically anything you can edit with Notepad, Vi or (X)Emacs
- But images, video clips, Microsoft Word, OpenOffice and other formats aren't plain text
- When there's a conflict, Subversion saves your copy and the master copy side by side in your working directory
- Up to you to resolve the differences
- It's not Subversion's fault
- Most creators of non-text formats don't provide a way to find or merge differences between files

Reverting

- After doing some more work, Ron decides he's on the wrong path
- `svn diff` shows him which files he has changed, and what those changes are
- He hasn't committed anything yet, so he uses `svn revert` to discard his work
  - I.e., throw away any differences between his working copy and the master as it was when he started
  - Synchronises with where he was, not with any changes other people have made since then
- If you find yourself reverting repeatedly, you should probably go and do something else for a while ...

Rolling Back

- Now Ron decides that he doesn't like the changes Harry just made to moons.txt → Wants to do the equivalent of "undo"



- `svn log` shows recent history
  - Current revision is 157
  - He wants to revert to revision 156
- `svn merge -r 157:156 moons.txt` will do the trick
  - The argument to the -r flag specifies the revisions involved
  - Merging allows him to keep some of Harry's changes if he wants to
  - Revision 157 is still in the repository

Creating a Repository

- To create a repository:
  - decide where to put it (e.g., ~/svn/rotor)
  - go into the containing directory: cd ~/svn
  - `svnadmin create rotor`
- Better think big
  - You can manipulate and checkout parts of you repository
  - But not easily exchange stuff between different repositories
  - Don't use many small repositories
- If you already have some material for the project
  - can now e.g. use `svn import` to put it into the repository

Creating a Sandbox

- One checks out the repository to obtain a working copy/sandbox via one of three ways (protocols) typically
  - Directly through the file system:
    `svn checkout file:///home/mohr/svn/rotor`
  - Through a web server:
    `svn checkout https://www.hogwarts.edu/svn/rotor`
    (requires a web server configured appropriately)
  - Over the internet with SSH:
    `svn co svn+ssh://sshproxy.lmu.de/home/mohr/svn/rotor`
- You use `svn checkout` only once, to initialise your working copy
  - After that, use `svn update` in that directory
  - If you only want part of the repository, use
    `svn co http://www.hogwarts.edu/svn/rotor/engine/dynamics`

SVN Commands

Command Structure

`svn` ⟨`options`⟩ ⟨`action`⟩ ⟨`options`⟩ ⟨`arguments`⟩

options	switches to steer subversion's behaviour, options may be global or specific to a certain action no special order required example: `-v` for enhanced verbosity of output
action	subcommand to be executed, like e.g. `commit`
arguments	for the subcommand, e.g. the files to be committed

Command Reference

name	purpose
svn checkout	Get a fresh working copy of a repository
svn add	Add files and/or directories to version control
svn log	Show history of recent changes
svn commit	Send changes from working copy to repository (inverse of update)
svn status	Show the status of files and directories in the working copy
svn update	Bring changes from repository into working copy (inverse of commit)

Command Reference (cont.)

name	purpose
svn delete	Remove files and/or directories from version control
svn help	Get help (in general, or for a particular command)
svn merge	Merge two different versions of a file into one
svn mkdir	Create a new directory and put it under version control
svn rename	Rename a file or directory, keeping track of history
svn revert	Undo changes to working copy (i.e., resynchronise with repository)

Some actions have short forms, e.g.

- `svn ci` $\longrightarrow$ `svn commit`
- `svn co` $\longrightarrow$ `svn checkout`

Reading SVN Output

- `svn status` prints status of working copy files and directories
  - with no arguments, prints only locally modified items (⟶ no repository access)
  - prints one line for each file that's worth talking about

```
==> svn status
M jupiter/moons.txt
C readme.txt
```

  - jupiter/moons.txt has been modified
  - readme.txt has conflicts

Reading SVN Output

- `svn update` prints one line for each file or directory it does something to

```
==> svn update
A saturn/moons.txt
U mars/mars.txt
```

- saturn/moons.txt has been added
- mars/mars.txt has been updated (i.e. someone else modified it)

Commit Messages

- Whenever you submit something to the repository you are asked for a commit message
  - integral part of version control
  - intended to describe what you have changed and why
- Without this documentation
  - it becomes nasty to solve questions like
    - "when did we exchange algorithm A for B?" (answer: rev 160!)
    - "why did we insert this line in rev 182?" (answer: to fix a bug!)
  - since you always have to compare differences between revisions explicitly (not possible for binary files!)
- You tell your collaborators what you did ($\rightarrow$ automatic commit mails)

Example: Commit Messages

```
==> svn log unit12.tex
------------------------------------------------------------------------
r852 | mohr | 2009-07-10 18:01:36 +0200 (Fri, 10 Jul 2009) | 4 lines

- Finished motivation, overview on version control and basic usage
- Want to complete this with outlook on advanced stuff, like branches and
  trac.

------------------------------------------------------------------------
r850 | mohr | 2009-07-09 18:29:05 +0200 (Thu, 09 Jul 2009) | 3 lines

This will become a unit on version control using subversion. Strongly based
on the respective part from the Software Carpentry Project by Greg Wilson.

------------------------------------------------------------------------
```

Tags

Tags

A tag is a symbolic name given to a snapshot of a project in time.

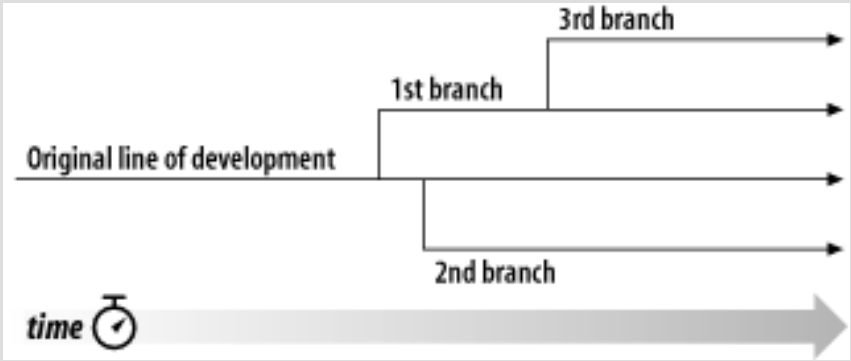
- Using Version Control we can always recreate a project's state via
  - a revision number (e.g. rev 350)
  - or a time stamp (e.g. 04/15/2009)
- but a symbolic name (like PRE_EGU_2009) is of course easier to remember.
- Conceptually we don't want to change a tag in the future.

Branches

- Assume you are working on a Mantle Convection code
  - code is running, but may still contain bugs
  - students are using it for production runs
- Now you want to experiment with a new mineral–physics model
  - you do not want to break production code with unfinished and/or untested new code
  - still want to do bug-fixes to production code

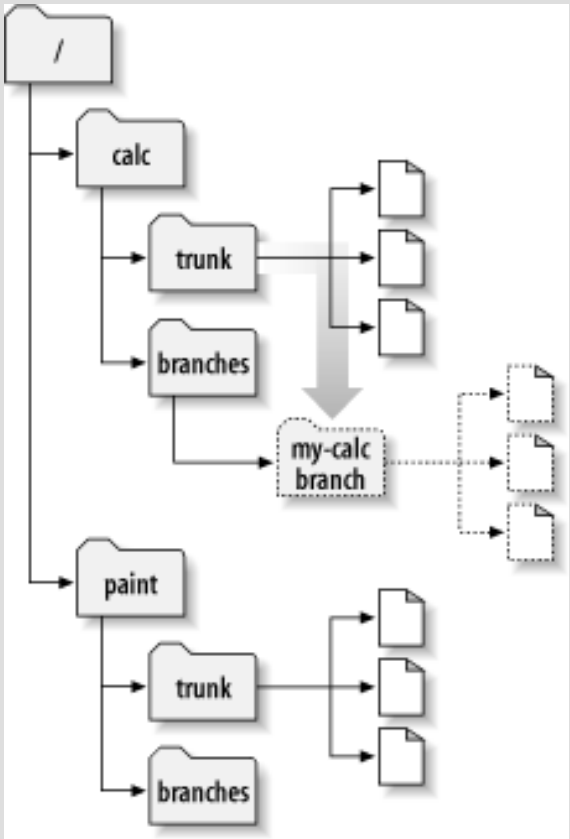
Branches (cont.)

- Solution for this
  - generate a branch for separate implementation of new features
  - once these are tested merge them back into the trunk
- On the trunk you can still do bug-fixes to production code



(Collins–Sussmann et al., Version Control with Subversion)

Branches (cont.)



(Collins–Sussmann et al.)

- Branching is the action of generating a duplicate of a project's current state.
- The branch is not static, but intended to be actively used for development.
- The original line of development is called trunk.
- It's possible to generate branches from branches, but rarely necessary.
- A branch might live-on independently forever (e.g. code variant for MacOS)
- or be retired and merged into the trunk at one point in time

Tags & Branches in SVN

- Subversion has no internal concept for tags or branches, it only knows how to deal with copies.
- In subversion tags and branches exist as normal filesystem directories in the repository, not in an extra dimension (different from many other systems)
- Generation of a tag works as

```
svn copy file:///home/mohr/svn/myProj/trunk
         file:///home/mohr/svn/myProj/tags/PRE_EGU_2009
```

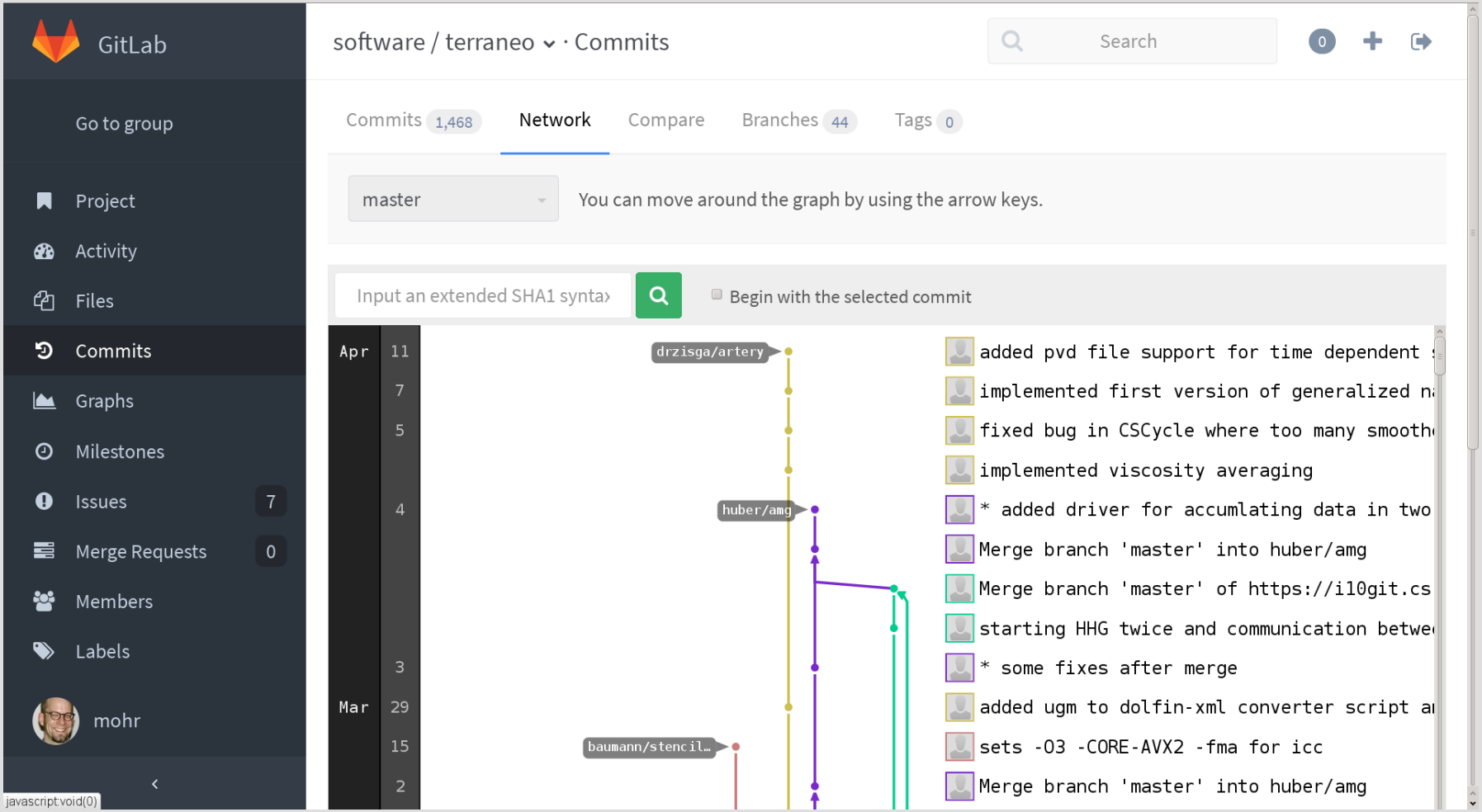
- Meaning of tag or branch is attached to the copy by the user not svn.
- Most people split the project up into three subdirectories (trunk, branches, tags)

Project Management

> Project management is the discipline of planning, organising and managing resources to bring about the successful completion of specific project goals and objectives. It is often closely related to and sometimes conflated with program management. (Wikipedia)

- Professional software projects today rely on tools for
  - project management
  - bug tracking
  - issue tracking
- such as e.g.
  - bugzilla
  - jira
  - trac

Example: Gitlab (1/2)



Example: Gitlab (2/2)