# CSC 2700: Scientific Computing

## Record and share your work: revision control systems

Dr Frank Löffler

Center for Computation and Technology
Louisiana State University, Baton Rouge, LA

Feb 13 2014

# Overview

# Version control systems

Version Control Systems:

- Most commonly stand-alone applications
- Also embedded into various other software, e.g.
  - word processors
  - spreadsheets
  - content management systems
  - wikis

Changes:

- Might be identified by number or letter code
- Termed as “revision number”, “revision level”, or simply “revision”
- Associated with time-stamp, user and log message
- Can be compared, restored, and with some types of files, merged

# Version control systems

Traditionally: centralized model - solve conflicts with one of two methods:

- File locking
  - grant write access to only one checkout at a time
  - benefit: can provide some protection against difficult conflicts
  - drawback: when locked for too long, developers are tempted to bypass system
- Version merging
  - Simultaneous edit by multiple developers allowed
  - Merge necessary when transferring change to central server
  - Automatic merging available for simple files: text

Often both options are available, but locking is typically not used (anymore).

# Branches - Trunk - Tags

Branch

- Duplication of a (larger) object under revision control
- Usually complete directory tree

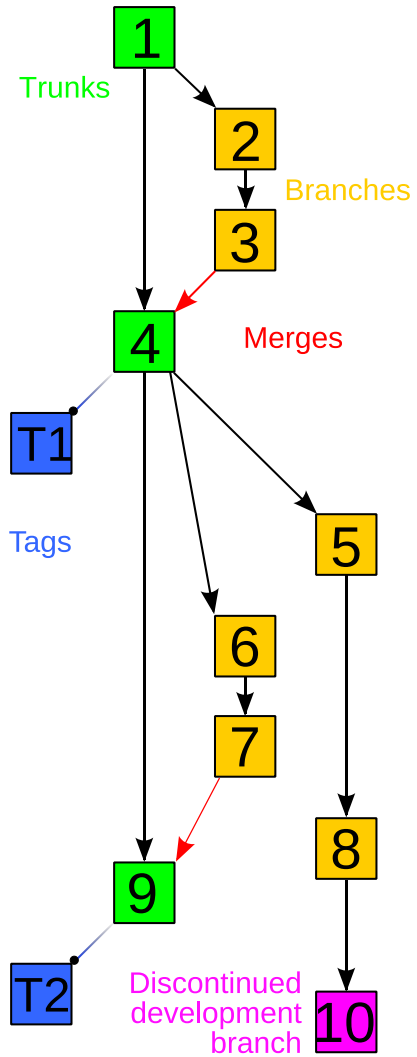
Trunk / Master

- Main, central development branch
- Upstream branch

Tag

- Repository snapshots
- Used especially for releases
- Synonyms: labels, baselines

Some repository systems treat all three identically.



# Version control systems - Branches

Branching: duplication of an object under revision control

- Modifications can happen in parallel along multiple branches.
- Branch: also known as trees, streams or code-lines
- Originating branch: *parent* branch or *upstream*
- Branch without parent: *trunk* or *mainline*
- Branches might be merged, especially into *trunk*
- Branches not intended to be merged usually called *fork*

# Distributed revision control

- Peer-to-peer approach
- Theoretically no central repository
- Repositories synchronized by exchanging change-sets (patches)
- Communication only necessary for exchange with other copies
- Some common operations are fast, because local
- In practice, projects tend to have one designated central, "official" repository and only limited exchange between user copies.

# Checkouts - Exports - Update - Commits

Checkout (synonym: working copy)

- Act of creating a local working copy from the repository
- User may specify a specific revision or obtain the latest

Export

- Act of obtaining only files from the repository
- Similar to checkout, but creates clean directory tree without version control metadata.

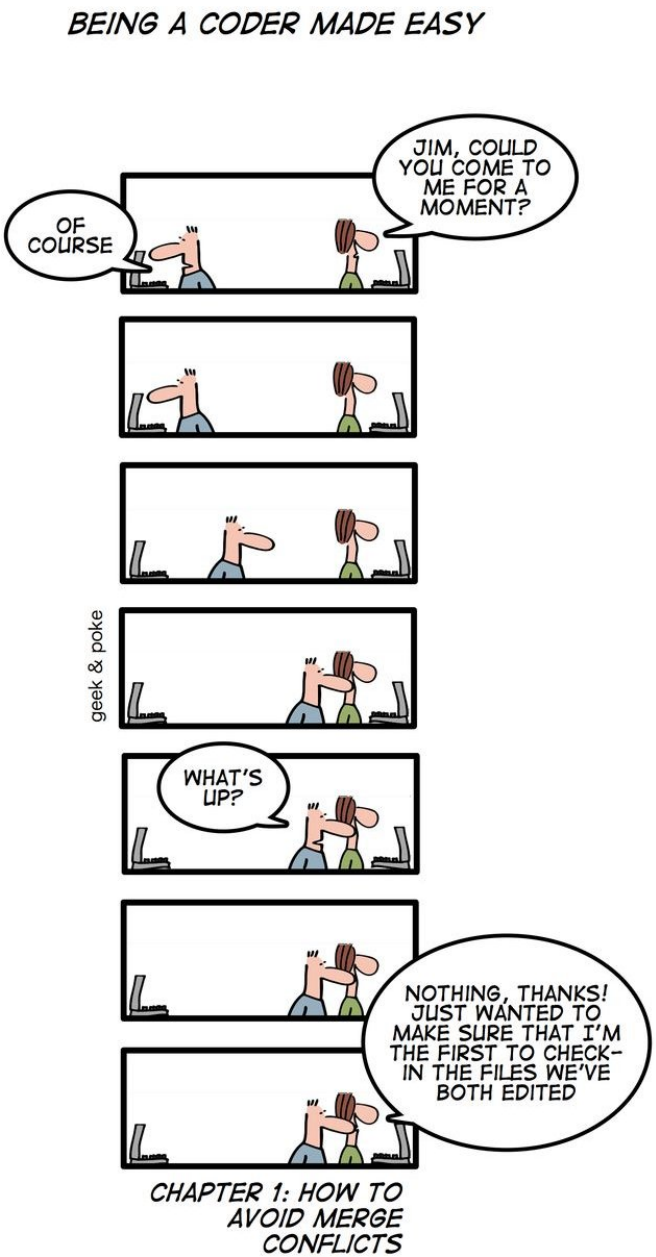
Update

- Merges changes in the repository into the local working copy
- Might create conflicts with local changes that might have to be resolved manually

Commit

- Action of writing or merging the changes made in the working copy back to the repository.
- Might not be possible without an up-to-date checkout
- Also: New revision that is created by committing

# Some day on Geek & Poke



# Revision control - Log messages

*"If you have nothing to say about what you are committing, you have nothing to commit."*

Log messages serve at least three important purposes

- To speed up the reviewing process.
- To help to write a good release note.
- To help the future maintainers to find out why a particular change was made (that might be you!)

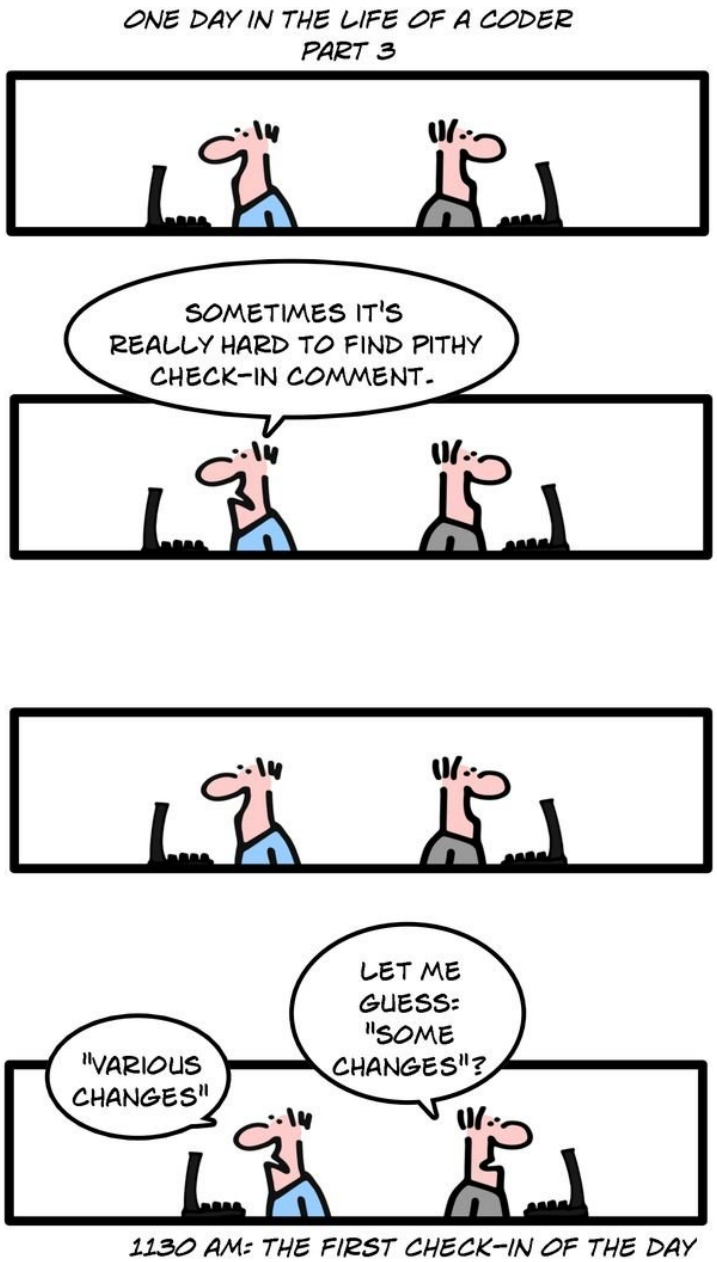
At least try to

- Summarize clearly in one line what this commit is about
- Describe the change, not how it was made

Even better

- Write one-line summary, followed by an empty line and a longer description
- Line-break the commit message

# Some day on Geek & Poke



# Revision control - example work-flows

Simple change to central repository

1. Checkout
2. Change file locally
3. Test change
4. Update - shows no remote changes
5. Commit change

# Revision control - example work-flows

Change to central repository with conflicts

1. Checkout
2. Change file locally
3. Test change
4. Update shows changes and merge conflicts
5. Resolve conflict
6. Test change
7. Repeat updating until success
8. Commit change

# Revision control - Conflicts

Updating from repository will

- Try to merge remote changes with local changes
- Create a conflict if this fails
  - Binary files: only option between 'theirs full' and 'mine full'
  - Text files: fine-grained control of multiple changes in single file

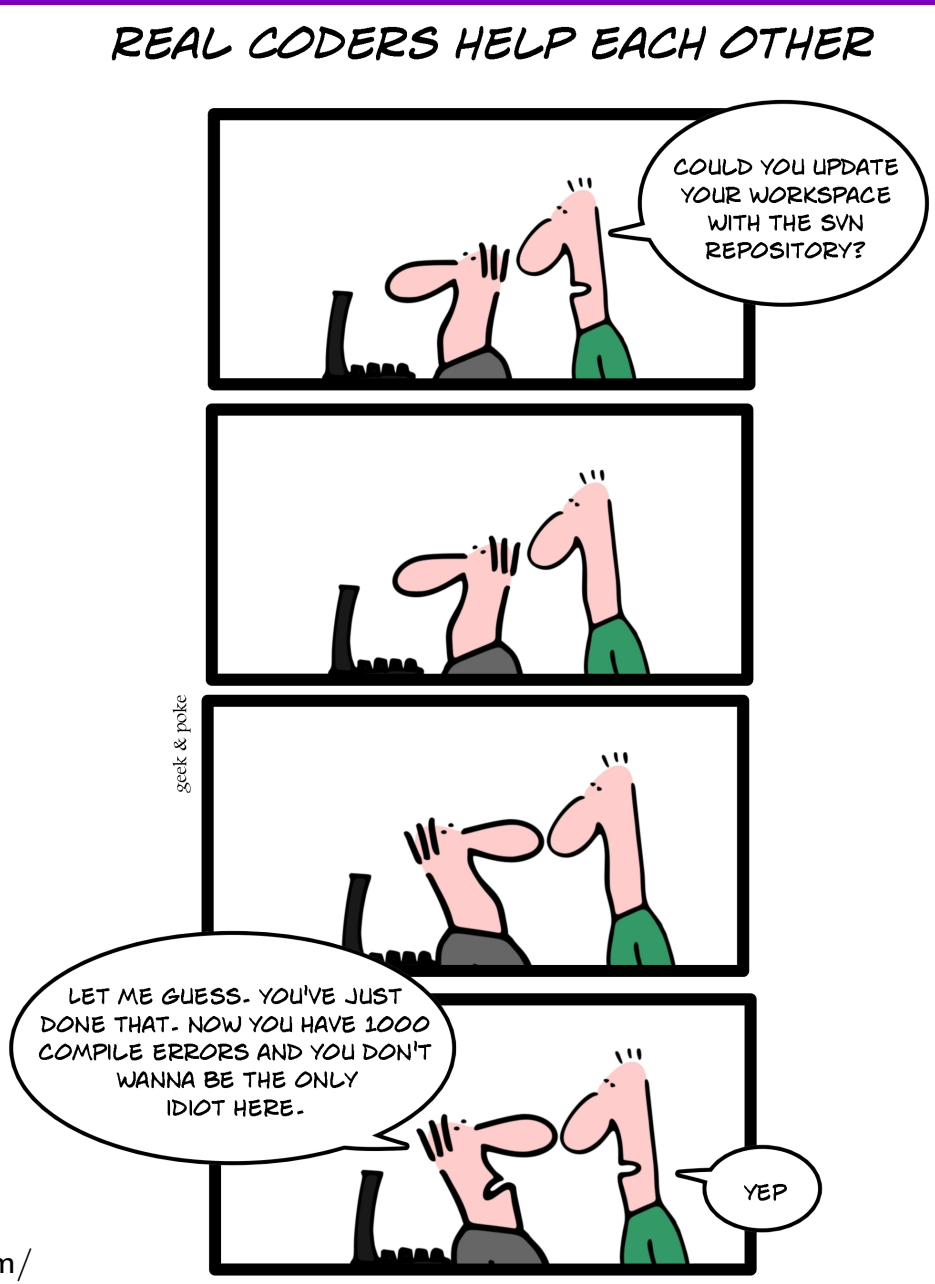
Example:

```
$svn update
  U index.html
  G changed-b.html
  C rubbish-b.html
  Updated to revision 46.
```

- U - Updated
- G - Merged
- C - Conflict

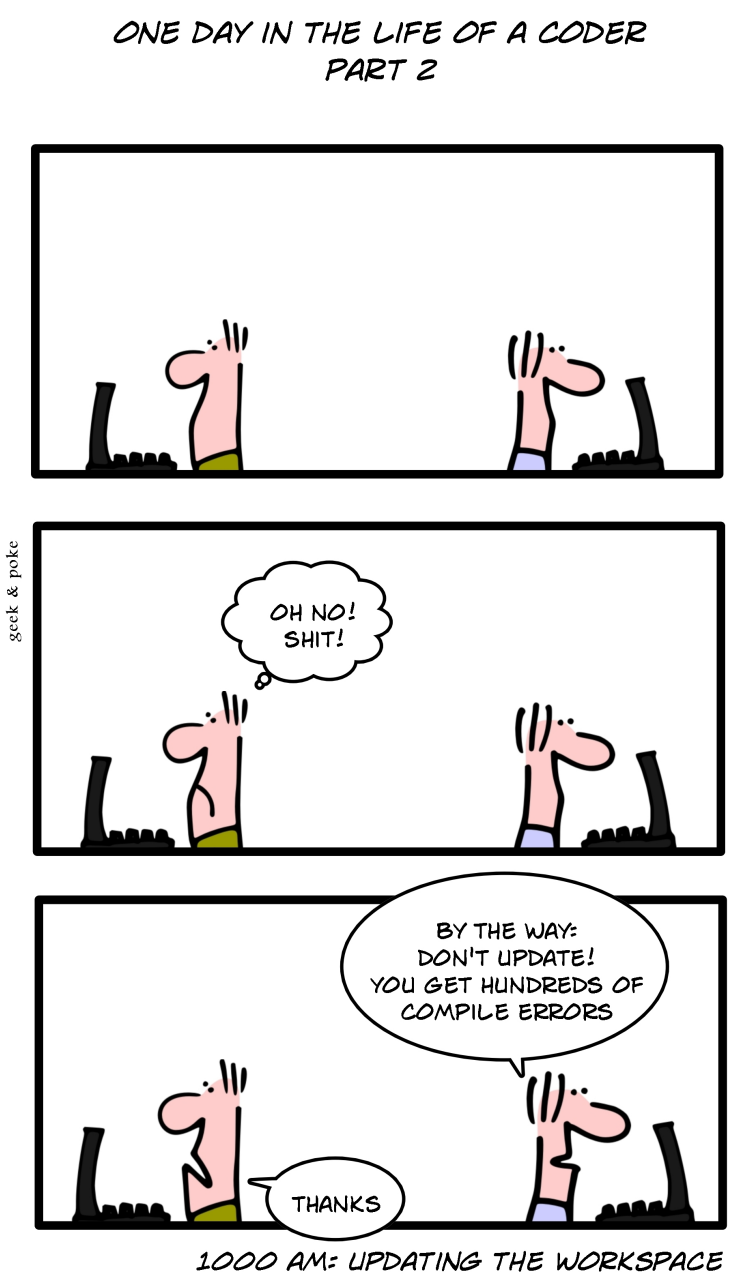
- Test after conflict resolving!

# Some day on Geek & Poke




http://geekandpoke.typepad.com/

# Some day on Geek & Poke




http://geekandpoke.typepad.com/

# Revision control - Conflicts

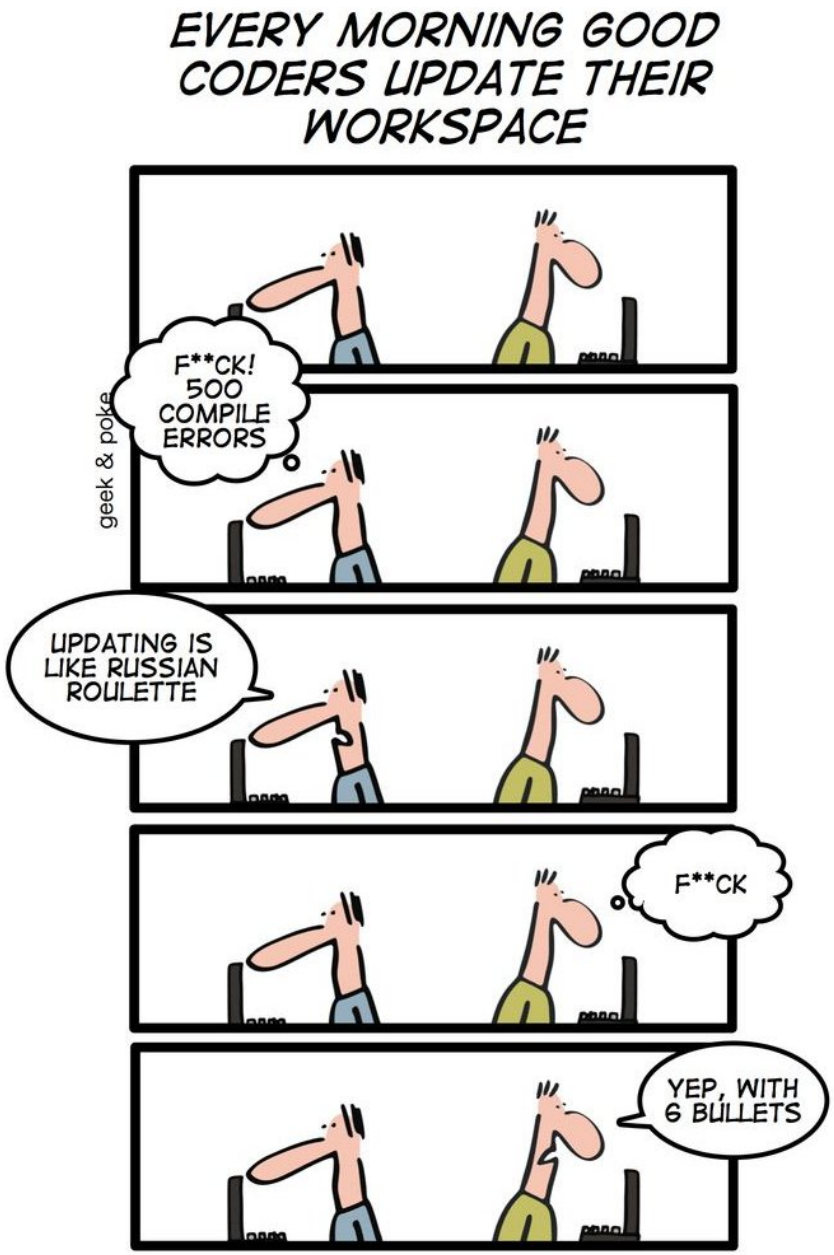
Conflict markers in text files:

```
<<<<<<< filename
    your changes
=======
    code merged from repository
>>>>>>> revision
```

Resolve conflict by

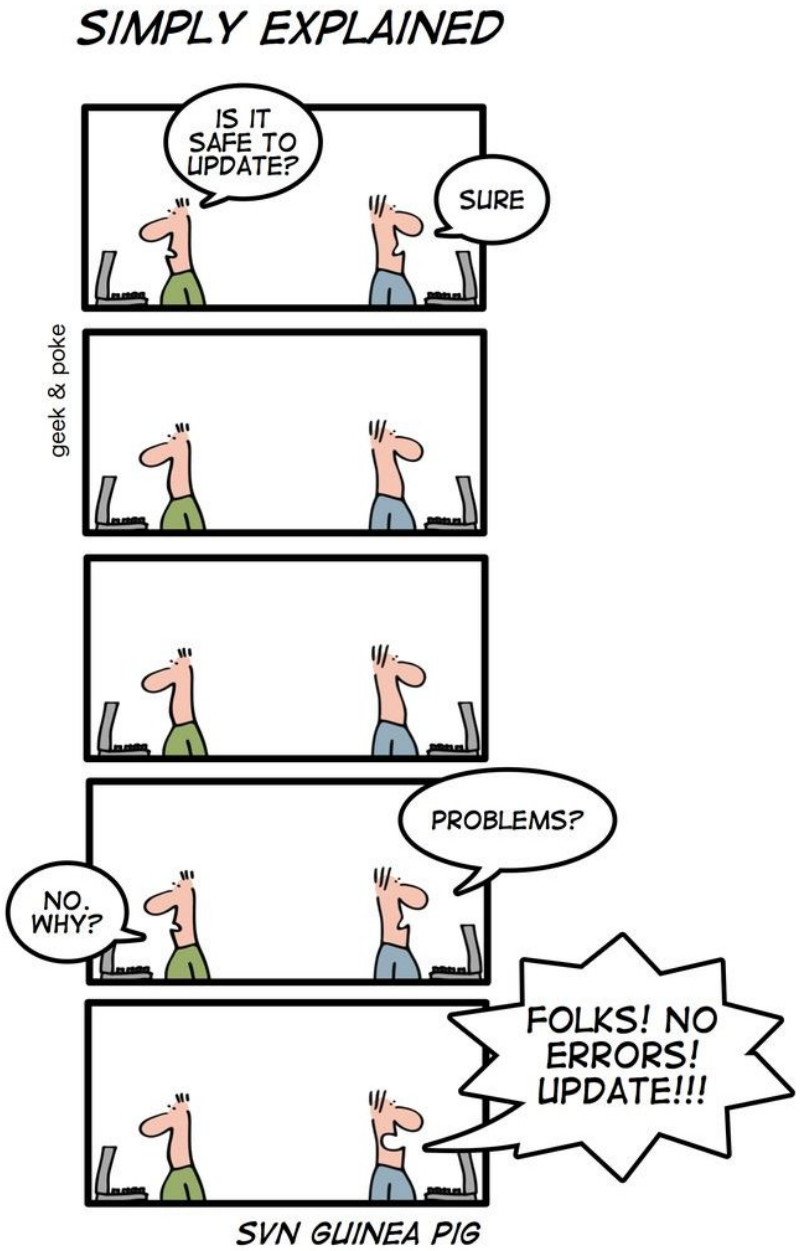
- Reviewing both changes and manually merging them
- Test merged version
- Tell RCS that conflict is resolved
- Commit

# Some day on Geek & Poke




http://geekandpoke.typepad.com/

# Some day on Geek & Poke




http://geekandpoke.typepad.com/

# Revision control - example workflows

Simple branch example

1. Create branch from trunk
2. Checkout branch
3. Change files locally, test and commit like on trunk
4. Possibly merge changes from trunk, resolve conflicts as usual
5. Eventually, merge changes from branch back into trunk
6. Remove branch

# Revision control - example workflows

Applying change using distributed RCSs

1. Checkout / Clone
2. Change file locally
3. Test change
4. Commit change (locally)
5. Update from other working copy (e.g. "central" repository)
6. Resolve possible conflicts, commit needed changes
7. Test again
8. Repeat until success
9. Push commit(s) to other working copy (e.g. "central" repository)

# Revision control - history examination

History in RCSs can be used to find out

- Why something was implemented (log messages)
- When something was implemented
- Who did a certain change

Typical example:

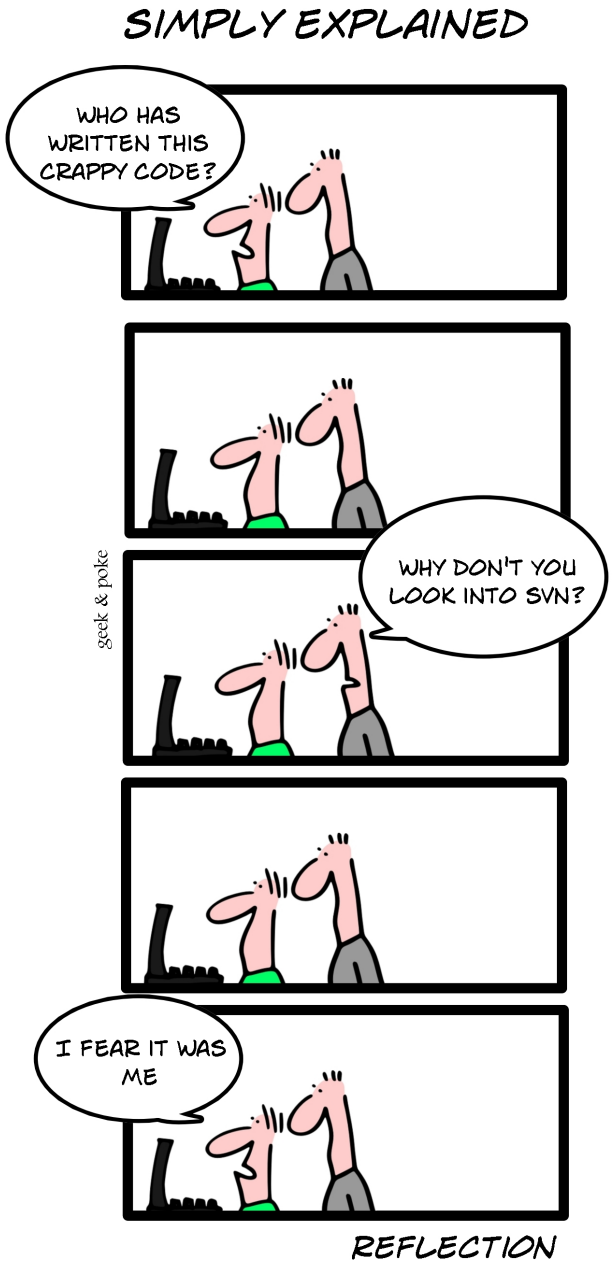
- Test of checkout: ok
- Test of checkout after local changes: ok
- Test after update from repository: failure

Narrow down location of bug by

- List log messages since last known working version
- Test without some of remote changes (go back in history)

Tools can help with this process.

# Some day on Geek & Poke




http://geekandpoke.typepad.com/

# Summary - Best RCS usage

Basic

- Use it, learn (much) about it!
- Put as much as possible under version control
- Only put original source in version control, not built objects
- Test changes before committing
- Commit often and in logical chunks
- Update as often as possible (close open files beforehand!)
- Write meaningful commit messages

Advanced

- Branch only when necessary
- Don't copy when you mean to branch
- Branch late
- Propagate / Merge early and often

Reading: Subversion, Git, Mercurial

# Course Work I

- Create public svn/git repository using free hosting service
- possibilities: bitbucket, github, google code, assembla, ...
- Commit/push coursework text file there
- Send email with (public) link to us

# Course Work II

`http://svn.cactuscode.org/flesh/trunk/src`

- What was the commit message of revision 7? Show how you found out.
- Which revision removed the macro `GROUP_SCALAR`? (mentioned in the commit message)
- Do a checkout and an export of that repository. Report about the size of both. Which one is larger and why? Which version of svn are you using?

`https://github.com/lsuits/moodle`

- How many forks does this project have on github?
- What is the (full) hash of the last commit, and the commit message?
- Show the 'diff' of commit cc95aed777c75fe899992168fcd40031b45cb9b0