Version Control

James Kermode
Department of Engineering

Outline

- What is version control?
- Why should you care about it?
- Centralised and distributed revision control
- Version control software: `rcs`, `cvs`, `svn` and `git`
- TCM's `cvs/svn` server and web interface
- Case study: 'Learn on the Fly' project

Version Control

- Version control (or revision control) is the management of multiple revisions of the same unit of information
- Usually applied to source code

  (But not necessarily: also widely used in engineering for blueprints and electronic design, and great for writing collaborative papers)
- Co-ordinate efforts of team of people to work on a complex task, avoiding duplication

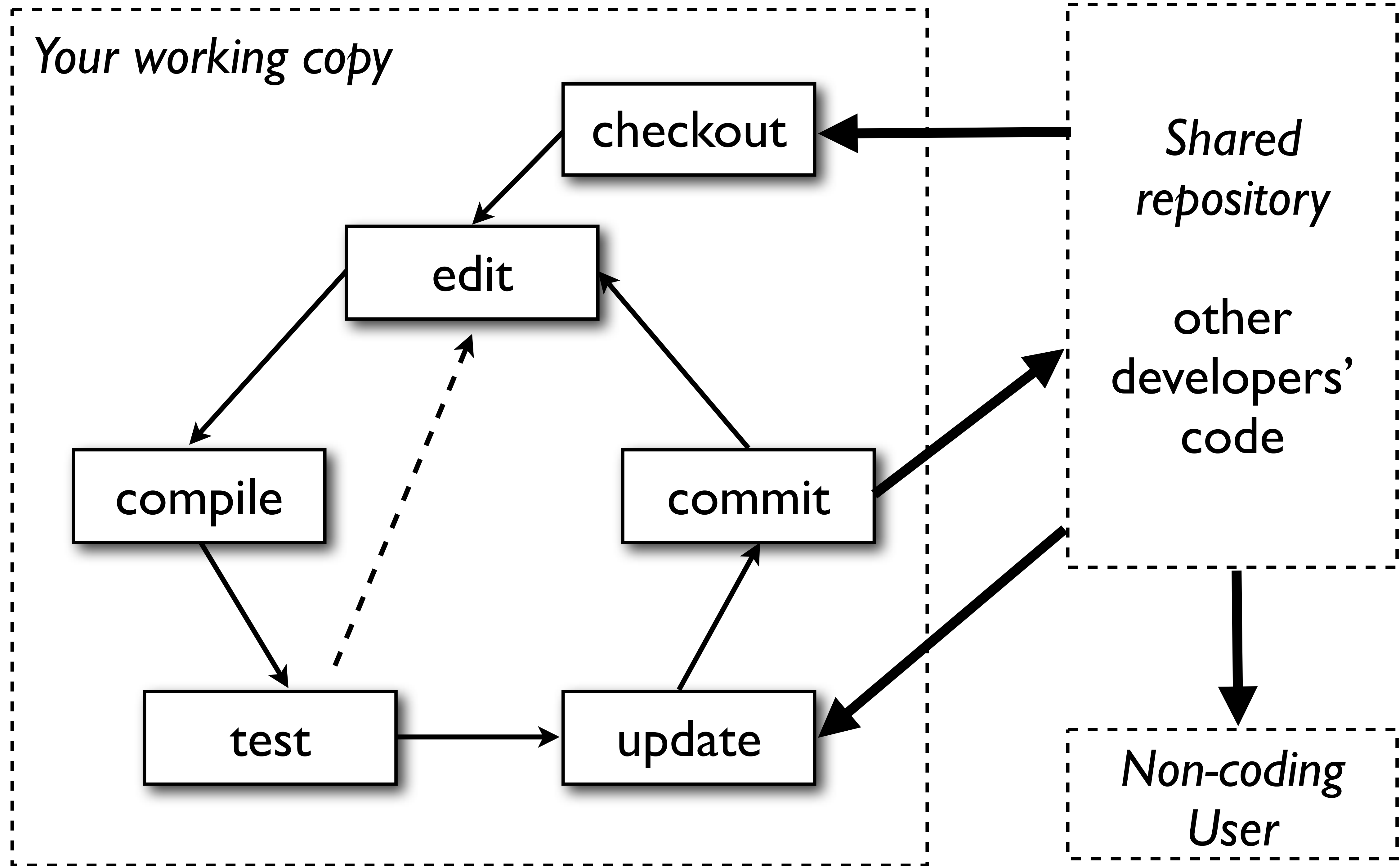
Manual Version Control

- Simplest approach
- Sequentially numbered revisions
- Requires manual maintenance of many near-identical versions of source code
- Self discipline required, so leads to mistakes
- Hard to co-ordinate between geographically diverse team of developers

Centralised Version Control

- Most version control systems work by storing differences between revisions: *delta compression*
- Centralised client-server model uses a single shared source code *repository*
- Source management models
- Two main approaches to concurrent access
  - File locking
  - Version merging (e.g. `cvs`, `svn`)

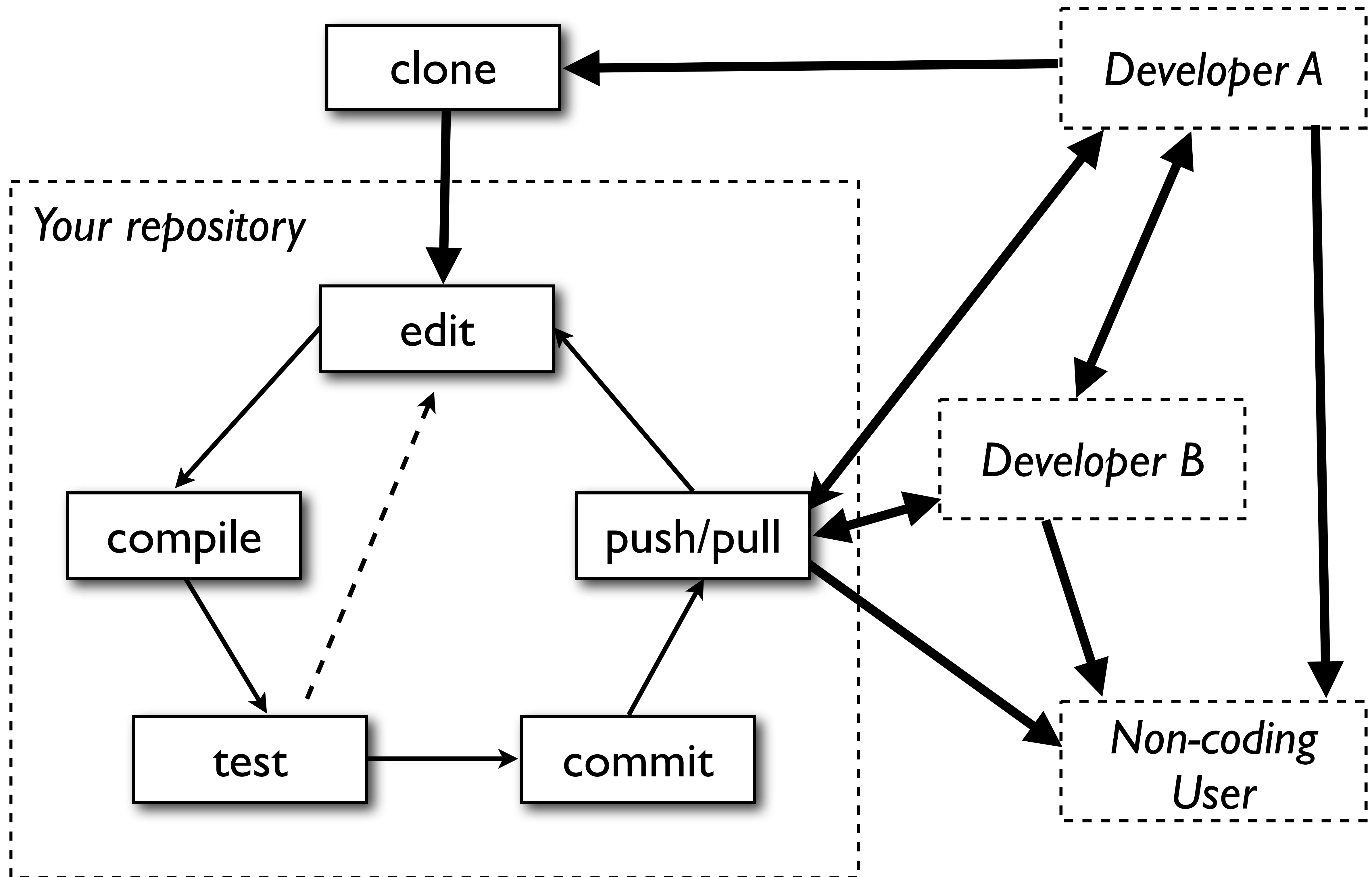
Centralised Version Control



Distributed Version Control

- Peer-to-peer approach
- Every working copy is a repository; there is no canonical version
- Every working copy is a separate branch
- Synchronisation is achieved by exchanging *patchsets*, leading to distributed 'Web of Trust'
- Examples
  - git created by Linus Torvalds, used by Linux kernel
  - mercurial used by Mozilla

Distributed Version Control



Distributed Version Control

Advantages

- Developers can be productive without network connection
- Usually faster, since no network involved
- Forking is easier
- Multiple redundant backups
- Fosters meritocratic culture; no ‘commitor’ status
- Can use version control for private work

Disadvantages

- Requires more merge conflict resolution by hand
- Project may need centralised control (especially true in scientific computing)
- Effect can be to replace central server with central person - is this an improvement?
- More complexity for developers

Concurrent Versions System

- In the early 1980s the Revision Control System `rcs` was developed
- `cvs` was originally a set of shell scripts to add support for multiple files and branches. Now full client-server system, still using RCS file format
- Major limitations
  - Does not version moving or renaming files or directories
  - No symbolic links
  - No atomic commits - can fail part way through, e.g. if there's a conflict in one file or network goes down
  - Branches are expensive

Subversion (svn)

- ‘State of the Art’ in centralised revision control
- Mostly-compatible successor to ageing cvs
- Very widely used (gcc, Apache, KDE, GNOME, Python, Google, ...) and hence stable
- Free (Apache license)
- Also gaining distributed features: currently status checks and diffs don’t need network, local commits coming soon

Improvements over cvs

- Commits are atomic
- Database storage - improved performance
- One revision number for entire repository
- Renamed/copied/moved/removed files retain full history
- Support for file metadata, symbolic links, directories and binary files
- Cheap branching and tagging (copy-on-write)

git

"I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git." - Linus Torvalds

- Initially developed in ~3 days in April 2005 for Linux kernel in response to BitKeeper changing license
- Emphasis on being fast, with strong support for non-linear development
- Set of core C programs plus associated shell scripts
- In many ways, closer to a filesystem with history than a traditional revision control system
- Can be used on top of `svn` via `git-svn`

cvs.tcm

- Michael has set up a dedicated cvs and svn server in TCM, called cvs.tcm.phy.cam.ac.uk
- Inside TCM, there is host-based access. From outside TCM, use RSA authentication (with a passphrase and ssh-agent)
- To check out a module:

```
$ export REPO=svn+ssh://cvs/home/jrk33/repo
$ svn checkout $REPO/Module Module
```

- Once you've checked out, don't need long path for subsequent operations

svn commands

- svn import
- svn checkout
- svn copy
- svn switch
- svn status
- svn commit
- svn update
- svn log
- svn diff
- svn add
- svn mv
- svn delete

Web Interface

- There is a read-only web interface, using `viewvc`
- Multiple levels of access control
  - Anonymous (public)
  - Raven
  - Per-user (or per-project) password file
  - Fine control to allow different sets of people to have access to different parts of the repository
- Generates very useful coloured diffs

Common svn problems

- `svn import` does not modify the directory it imports at all. After importing sources for the first time you have to checkout a fresh copy in a new directory
- Committing to a repository with a newer version of the `svn` client stops older clients from being able to access the repository, leading to version inflation

svn branches

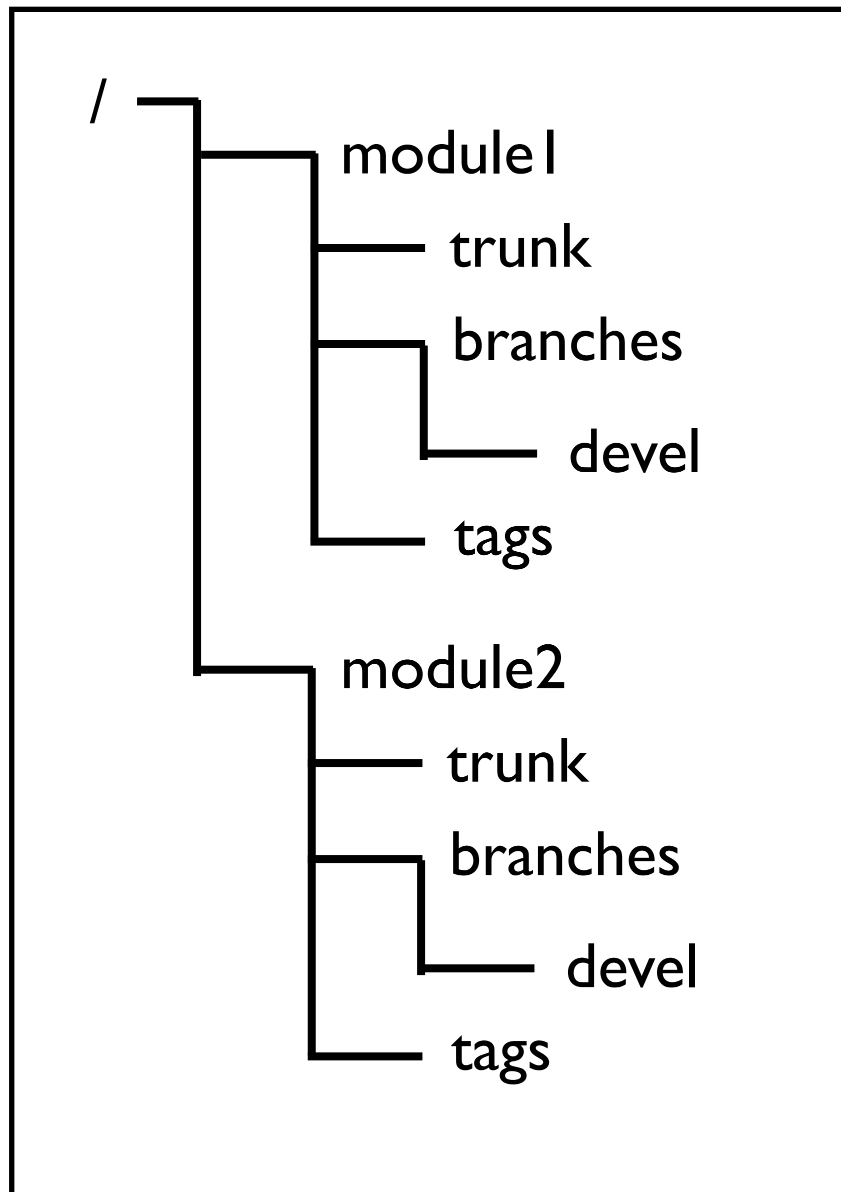
- Branches are easy to make in `svn`:

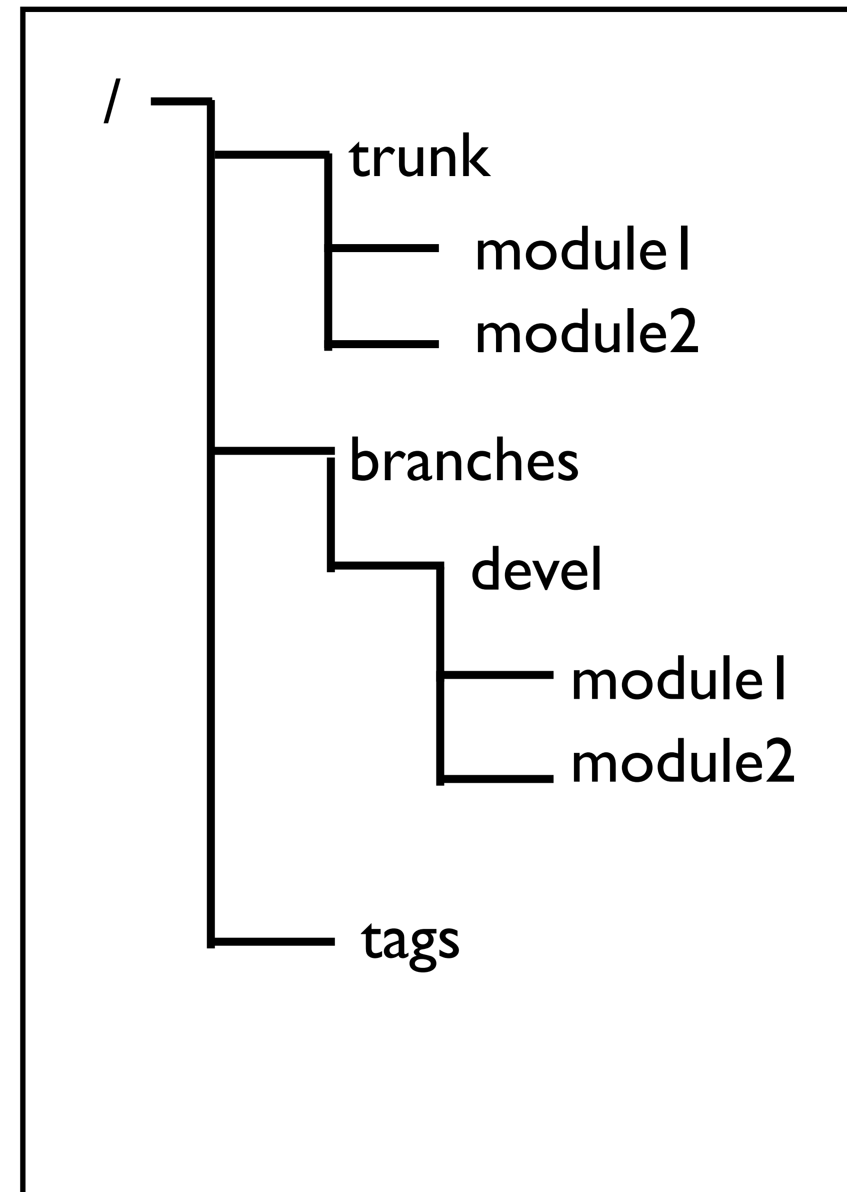
  `svn copy . $REPO/branches/new-branch`

  `svn switch $REPO/branches/new-branch`
- Copy-on-write means copies are cheap
- Recommended repository layout includes
  - `trunk` - for stable version
  - `branches` - where branches live
  - `tags` - for snapshots, e.g. releases
- It's common to have a `devel` branch for developers

svn branches

Repositories typically have one of two structures, depending on how tightly integrated various modules are





'Learn on the Fly'

- 2005 ~ 4 developers, all in TCM
- Single `cvs` repository on `/rscratch` adequate
- Continued like this until 2008. Problems we faced:
  - No branches (not really a `cvs` limitation...)
  - Necessity to do manual backups
  - No support for code reorganisation (moving/renaming loses `cvs` history)
  - All users and developers need TCM shell accounts

‘Learn on the Fly’

- Currently ~100 000 lines of Fortran 95
- ~8 active developers in disparate locations
- Plus users who want up-to-date, read-only access
- Some parts of code public, others private
- Options: stay with `cvs`, move to `svn`, move to `git`
- Moving to `svn` repository on `cvs.tcm` addressed all our problems (and `cvs2svn` tool does a good job of migration)

Our Branch Philosophy

- We have two main branches
  - `trunk` - for non-coding users and public access
  - `branches/devel` - for active developers
- Plus individual developers create `devel-User`
- Bugs get fixed in trunk, then we use svnmerge.py:
  - *Regularly*: bug fixes from `trunk` into `devel`
  - *Regularly*: updates from `devel` into `devel-User`
  - *When stable*: from `devel-User` to `devel`
  - *Periodically, when well tested*: from `devel` to `trunk`

Conclusions

- Version control is a Good Idea
- Version control is not difficult
- Try to avoid using `rcs` and `cvs`
- Consider using `svn`
- Consider using `cvs.tcm`

Acknowledgments and References

- Michael Rutter
- All the LOTF people who've been using the repository for the last few months
- Wikipedia, from where much of my introductory material has been adapted

- The `svn` book http://svnbook.red-bean.com/

svnmerge.py

- `svnmerge.py` is a useful add-on which improves merge tracking support (not necessary with svn 1.5)
- After you first make a new branch do:

```
svnmerge.py init $REPO/Module/trunk
svn commit -F svnmerge-commit-message.txt
```

- Then when you want to see what's available to merge, complete with log messages

```
svnmerge.py avail -l
```

- Finally, to do the merge

```
svnmerge.py merge
svn commit -F svnmerge-commit-message.txt
```

- More complex (bidirectional) merges are supported
  See http://www.orcaware.com/svn/wiki/Svnmerge.py

svn scripts

Here's a script called svnall to run an svn command across a series of subdirectories

```
#!/bin/bash

dirs=""

# Try to interpret first arguments as SVN controlled directories
while [[ -d $1 && -d $1/.svn ]]; do
    dirs="${dirs} $1"
    shift
done

# Find all SVN controlled directories one level down from cwd.
if [[ $dirs == "" ]]; then
    dirs=$(find . -path '*/.svn' -maxdepth 2 | sed -e 's//.svn//' -e 's/.///')
fi

# Find length of longest directory name (just for pretty formatting)
maxlen=$(python -c 'import sys; print max([len(a) for a in sys.argv[1:]])' $dirs)

# Execute all the svn commands, labelling with names of directories
for dir in $dirs; do
    olddir=$(pwd)
    cd $dir && svn $(echo $@ | sed -e "s/%d/$dir/") | awk '{printf "%-'$maxlen's : %sn","'$dir'",$0}' && cd $olddir
done
```

Use svnall status | grep -v '? :' for a quick view of what's modified

svn scripts

This wrapper function will prevent you from accidentally committing all modified files in a directory

```
function svn() {
    svn=$(which svn)
    if [[ $1 != "ci" && $1 != "commit" ]]; then
      $svn "$@"
      return $?
    else
      gotfile=0
      for a in $@; do
          [[ -f $a || -d $a ]] && gotfile=1
      done
      if (( gotfile == 0 )); then
          echo Committing without specifying any files is not allowed!
          echo If you need to commit everything then you should use
          echo
          echo '  svn ci -m message * .'
          echo
            echo "(this will be necessary if you are using svnmerge.py)"
            echo
          return 1
      fi
      $svn "$@"
      return $?
    fi
}
```