Construction: version control and system building

Paul Jackson

School of Informatics
University of Edinburgh

The problem of systems changing

- Systems are constantly changing through development and use
  - Requirements change and systems evolve to match
  - bugs found and fixed
  - new hardware and software environments are targeted
- Multiple versions might have to be maintained at each point in time
- Easy to lose track of which changes realised in which version
- Help is needed in managing versions and the processes that produce them.

Software Configuration Management to the rescue

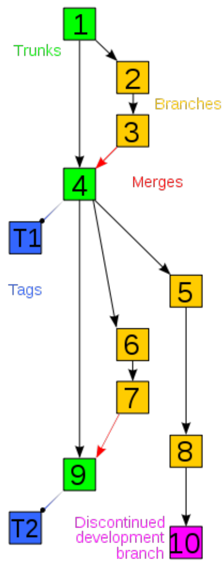
CM is all about providing such help.

Common CM activities:

- Version control
  - tracking multiple versions,
  - ensuring changes by multiple developers don't interfere
- System building
  - assembling program components, data and libraries,
  - compiling and linking to create executables
- Change management
  - tracking change requests,
  - estimating change difficulty and priority
  - scheduling changes
- Release management
  - preparing software for external release
  - tracking released versions

Focus on first two today

Version control



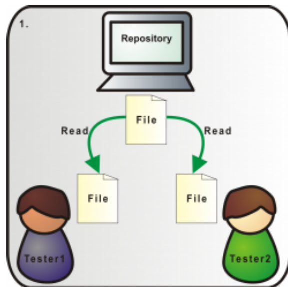
Version control

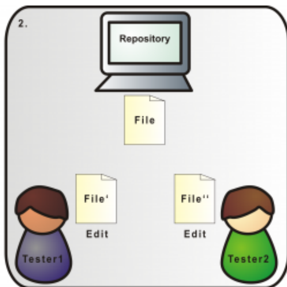
The core of configuration management.

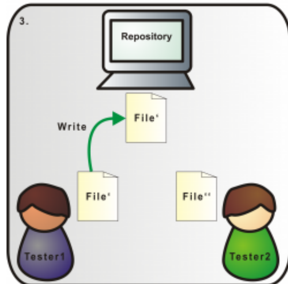
The idea:

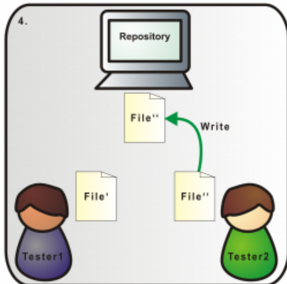
- keep copies of every version (every edit?) of files
- provide change logs
- somehow manage situation where several people want to edit the same file
- provide *diffs*/*deltas* between versions

Avoiding Race Conditions









RCS

RCS is an old, primitive VC system, much used on Unix.

Suitable for small projects, where only one person edits at a time.

Lock-Modify-Unlock model:

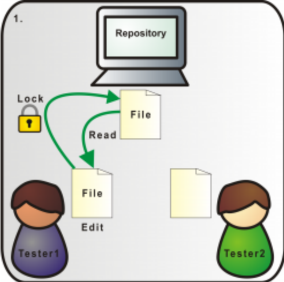
- Editor *checks-out* a file
  - Editor locks file
  - Others can check-out, but only for reading
- Editor makes changes
- Editor *checks-in* modified file
  - Lock is released
  - Changes now viewable by others
  - Others now can make their own changes

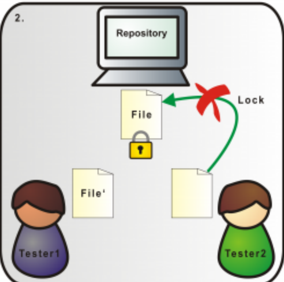
Keeps deltas between versions; can restore, compare, etc. Can manage multiple *branches* of development.

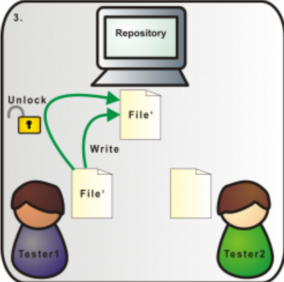
Remains a very useful tool for personal projects – and articles, lectures, essays, etc.

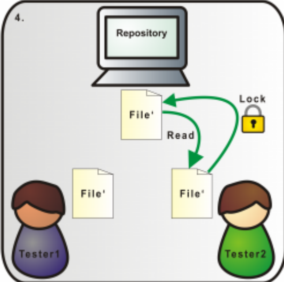
Further reading: *man rcsintro* on DICE.

Lock-Modify-Unlock









CVS and SVN

CVS is a much richer system, (originally) based on RCS.
Subversion (SVN) very similar.

Handles entire directory hierarchies or projects – keeps a single master *repository* for project.

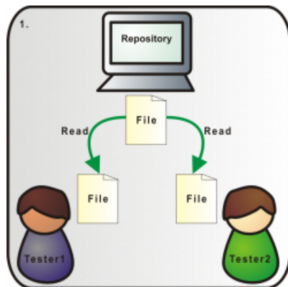
Is designed for use by multiple developers working simultaneously – Copy-Modify-Merge model replaces Lock-Modify-Unlock.

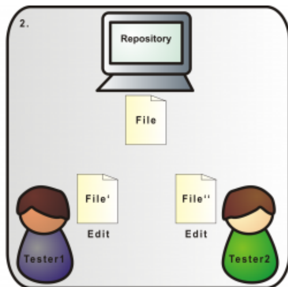
Pattern of use for Copy-Modify-Merge:

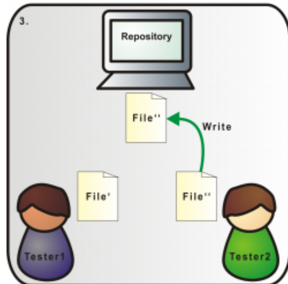
- *check out* entire project (or subdirectory) (not individual files).
- Edit files.
- Do *update* to get latest versions of everything from repository
  - system merges non-overlapping changes
  - user has to resolve overlapping changes - conflicts
- *check-in* version with merges and resolved conflicts

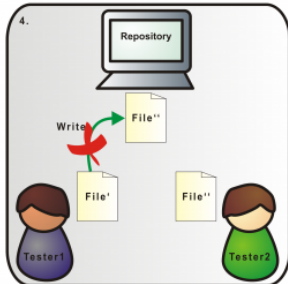
Central repository may be on local filesystem, or remote.

Copy-Modify-Merge

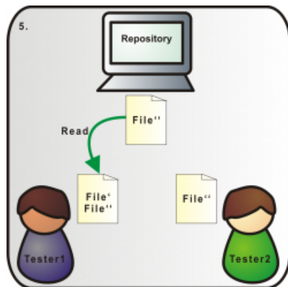


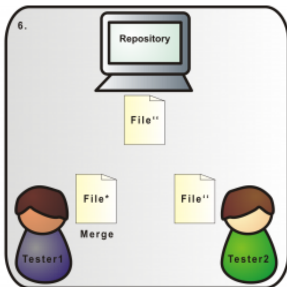


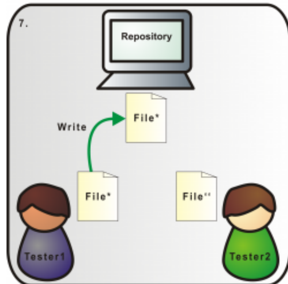


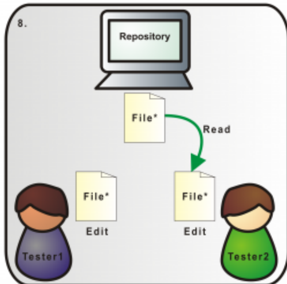


Copy-Modify-Merge









Distributed version control

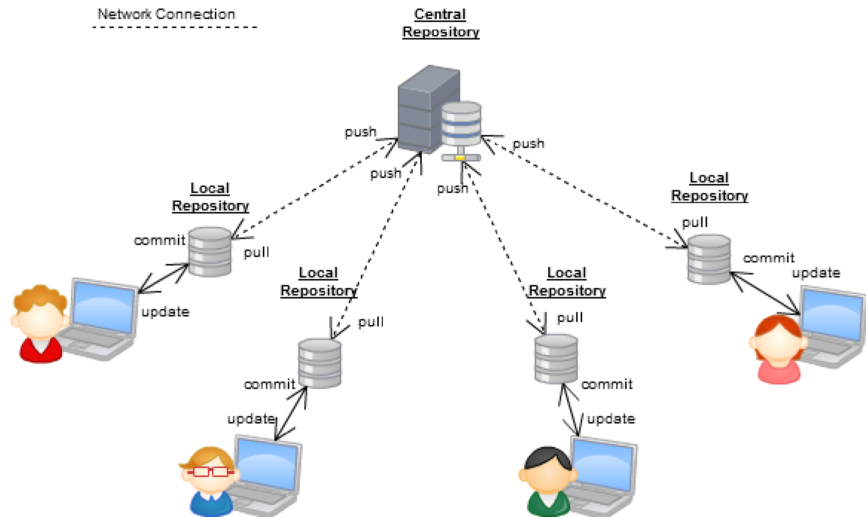
E.g. Git, Mercurial, Bazaar.

All the version control tools we've talked about so far use a single central repository: so, e.g., you cannot check changes in unless you can connect to its host, and have permission to check in.

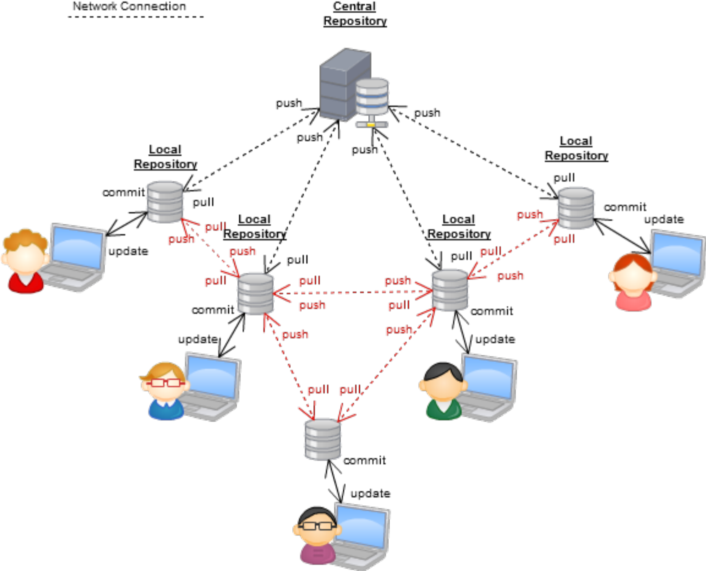
Distributed version control systems (dVCS) allow many repositories of the same software to be managed, merged, etc.

- reduces dependence on single physical node
- allows people to work (including check in, with log comments etc.) while disconnected
- much faster VC operations
- much better support for branching
- makes it easier to republish versions of software
- But... much more complicated and harder to understand

Distributed VCS



Distributed VCS



Branches

Simplest use of a VCS gives you a single linear sequence of versions of the software.

Sometimes it's essential to have, and modify, two versions of the same item and hence of the software: e.g., both

- the version customers are using, which needs bugfixes, and
- a new version which is under development

As the name suggests, branching supports this: you end up with a tree of versions.

What about merging branches, e.g., to roll bugfixes in with new development?

With CVS/SVN, this is very hard. Distributed version control systems support it much better, so developers use branches a lot more.

Releases

Releases are configurations packaged and released to users.

alpha release for friendly testers only: may still be buggy, but maybe you want feedback on some particular thing

beta release for any brave user: may still have more bugs than a real release

release candidate sometimes used (e.g. by Microsoft) for something which will be a real release unless fatal bugs are found

bugfix release e.g. 2.11.3 replaces 2.11.2 - same functionality, but one or more issues fixed

minor release e.g. 2.11 replacing 2.10: basically same functionality, somehow improved

major release e.g. 3.0 replacing 2.11: significantly new features.

Variants, e.g. even (stable) versus odd (development) releases...

Build tools

Given a large program in many different files, classes, etc., how do you ensure that you recompile one piece of code when another than it depends on changes?

On Unix (and many other systems) the `make` command handles this, driven by a *Makefile*. Used for C, C++ and other 'traditional' languages (but not language dependent).

part of a Makefile for a C program

```
OBJS = ppmtomd.o mddata.o photocolcor.o vphotocolcor.o dyesubcolcor.o
ppmtomd: $(OBJS)
        $(CC) -o ppmtomd $(OBJS) $(LDLIBS) -lpnm -lppm -lpgm -lpbm -lm

ppmtomd.o: ppmtomd.c mddata.h
        $(CC) $(CDEBUGFLAGS) -W  -c ppmtomd.c

mddata.o: mddata.c mddata.h
```

Makefiles list the *dependencies* between files, and the commands to execute when a depended-upon file is newer.

`make` has many baroque features – and exists in many versions. (Just use GNU Make.)

Like version control, a Makefile is something *every* C program should have if you want to stay sane.

Ant

`make` can be used for Java.

However, there is a pure Java build tool called Ant.

Ant *Buildfiles* (typically `build.xml`) are XML files, specifying the same kind of information as `make`.

There is an Eclipse plugin for Ant.

part of an Ant buildfile for a Java program

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name ="Dizzy" default = "run" basedir=".">
<description>
    This is an Ant build script for the Dizzy chemical simulator. [...]
</description>
<!-- Main directories -->
<property name = "source"      location = "${basedir}/src"/> [...]
<!--General classpath for compilation and execution-->
<path id="base.classpath">
  <pathelement location = "${lib}/SBWCore.jar"/> [...]
</path> [...]
<target name = "run" description = "runs Dizzy"
                     depends =" compile, jar">
  <java classname="org.systemsbiology.chem.app.MainApp" fork="true">
    <classpath refid="run.classpath" />
    <arg value="." />
  </java>
</target> [...]
</project>
```

Maven

Maven extends Ant's capabilities to include management of dependencies on external libraries

Maven *buildfiles* (typically `pom.xml`) are XML files, specifying the same kind of information as `Ant` buildfiles but also which classes and packages depend on which versions of which libraries.

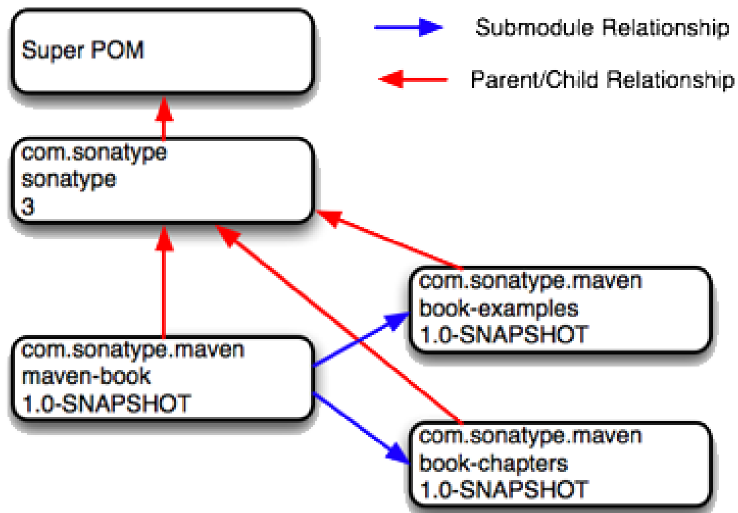
Maven is considerably more complex than Ant, and considerably more useful for projects using many external libraries (i.e., most Java projects).

There is an Eclipse plugin for Maven (actually, two).

A Maven buildfile for a Java program

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://ww
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.ap
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Maven Parent-Child



Per-platform code configuration

Different operating systems and different computers require code to be written differently. (Incompatible APIs. . .). Writing portable code in C (etc.) is hard.

Tools such as GNU Autoconf provide ways to automatically extract information about a system. The developer writes a (possibly complex) configuration file; the user just runs a shell script produced by autoconf.

(Canonical way to install Unix software from source: `./configure; make; make install`.)

A newer tool is CMake.

Problem is less severe with Java. (Why?) But still tricky to write code working with all Java dialects.

Reading

Required: Chapter 1, Fundamental Concepts, of the SVN book
`http://svnbook.red-bean.com/`

Suggested: man rcsintro

Suggested: Eclipse Team Programming with CVS (see above)

Suggested: Tutorial about dVCS, `http://hginit.com/00.html`