Construction:
Configuration, build, unit testing, debugging

Nigel Goddard

School of Informatics
University of Edinburgh

Boring(?) but *really useful* things

Configuration management means managing all the source code, object code, compiled code. . . keeping track of changes, allowing developers to cooperate, automatic building, etc. etc. Version control is a fundamental part of this task. We'll also talk about build tools.

Testing of various kinds is an essential ingredient of software engineering. Test your components (*unit testing*); test that they work together (*system testing*); when you change something, check that everything still works (*regression testing*), . . .

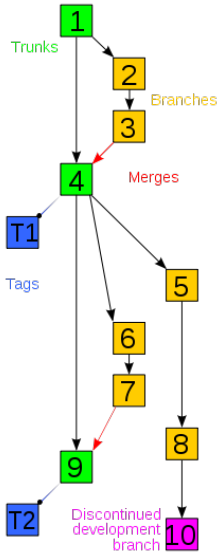
Debugging sometimes has to be done . . . how?

Naturally Eclipse provides support for all of these.

Software configuration management

1. Version control tools are important even to individual programmers (you should be using VC on all your assignments, written or coded!)
2. Configuration management tools have additional features to support teams
3. There's more to software configuration management than picking a tool. . .

Version control



Version control

is the core of configuration management.

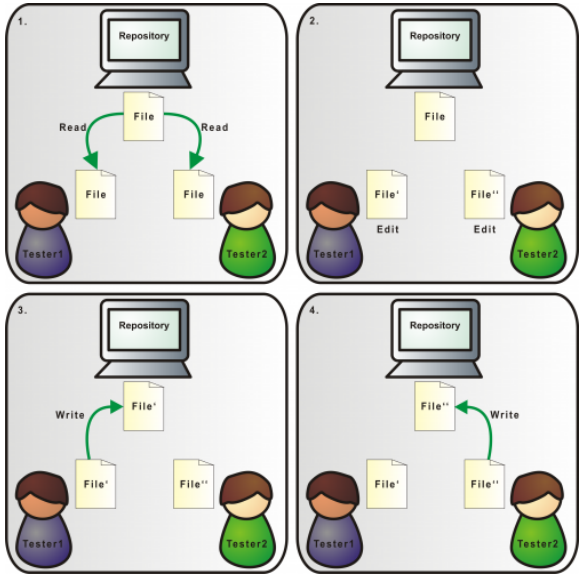
- keep copies of every version (every edit?) of files
- provide change logs
- somehow manage situation where several people want to edit the same file
- provide *diffs*/*deltas* between versions
- etc.

Local history in Eclipse

Eclipse has built in to its core a very simple version control system. Eclipse keeps local history for each file: copy of file every time it is saved. Can compare or restore versions.

For details and how to use, see Eclipse help page,
Workbench User Guide : Concepts : Workbench : Local history

Avoiding Race Conditions



RCS

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

Suitable for small projects, where only one person works on a file at a time.

Works by *locking* files, preventing *check-out* by other developers. *Check in* files when done editing – changes now available to others. Lock-Modify-Unlock model.

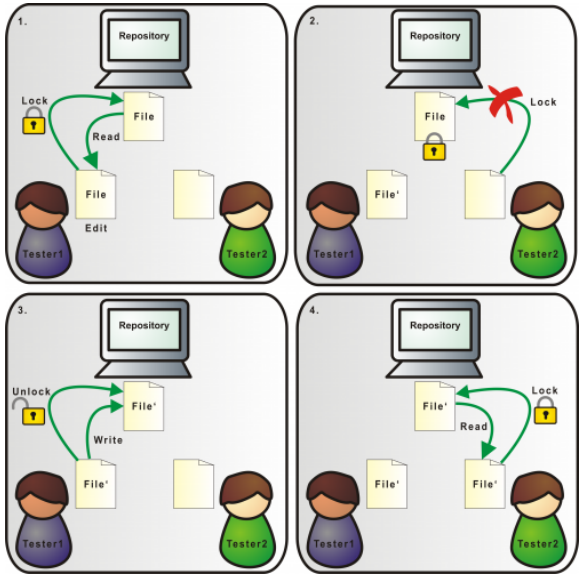
Keeps deltas between versions; can restore, compare, etc. Can manage multiple *branches* of development.

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

RCS is included in all Unix/Linux distributions.

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.

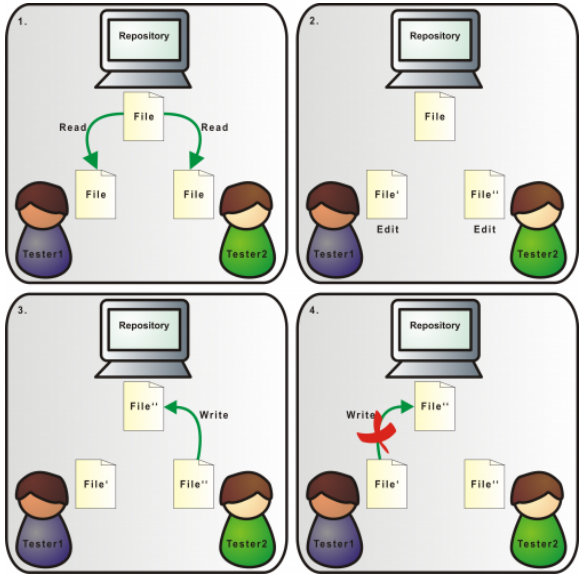
If conflicting updates are checked in, CVS automatically merges changes, if it can. Otherwise flags problem to second user.

Pattern of use: *check out* entire project (or subdirectory) (not individual files). Edit files. Do *update* to get latest versions of everything from repository and check for conflicting updates. *Check in* your edits.

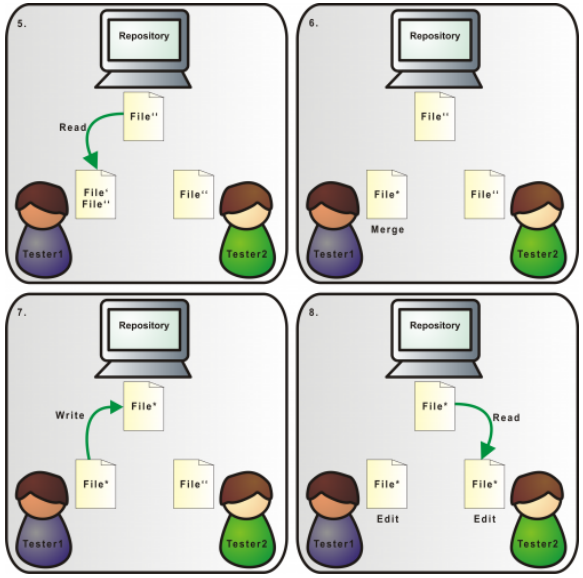
Central repository may be on local filesystem, or remote.

Many additional features.

Copy-Modify-Merge



Copy-Modify-Merge



Distributed version control

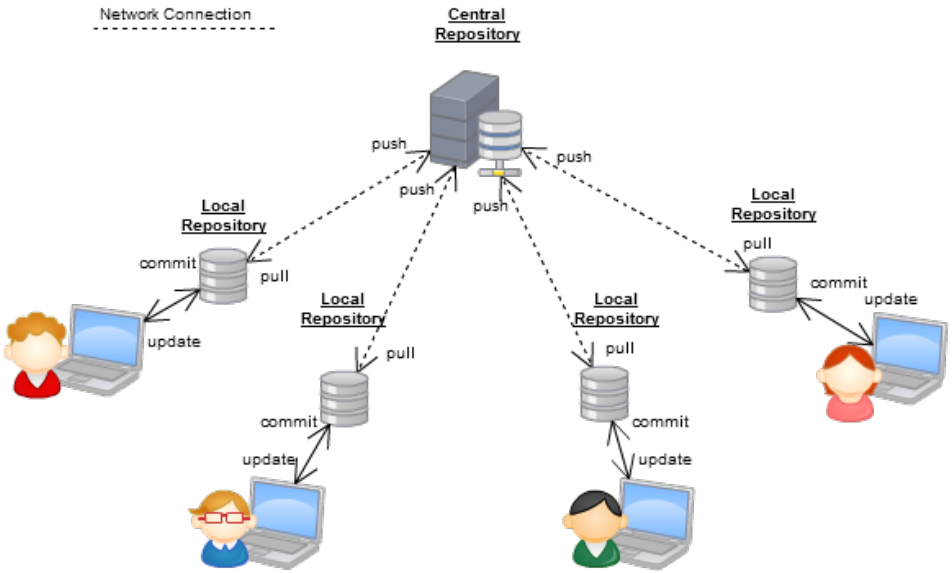
e.g. Darcs, Git, Bazaar, Mercurial.

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.

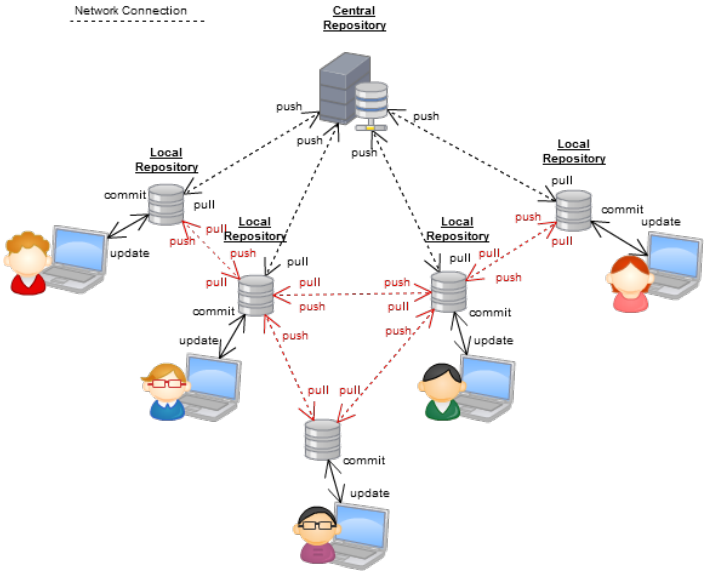
Distributed version control systems (dVCS) allow many repositories of the same software to be managed, merged, etc.

- + 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...

Software configuration management process

Key activities (from SWEBOK Ch7):

- software configuration identification – what needs to be controlled, what are the relationships, what constitutes a release?
- software configuration control – processes for agreeing to make a change (see later lecture on Deployment and Maintenance)
- software configuration status accounting – where's the product at? what's in the latest release? how fast are change requests being dealt with?
- software configuration auditing – is it actually being done right?
- software release management and delivery – we'll talk about build tools, but see also later lecture on Deployment and Maintenance.

Dependencies

Much of software engineering can be seen as managing dependencies, in the most general sense:

A depends on B if, when B changes, it's possible A may need to change as a consequence

Some of this is captured in the software configuration identification (how much depends on just how you do that).

Some is general, e.g., Foo.class will depend on Foo.java.

If the change that may be forced is, e.g., a change to the code, a human will have to make it.

If it's just that something needs to be recompiled, we can automate it provided a tool has the dependency information.

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 capatibilities 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.

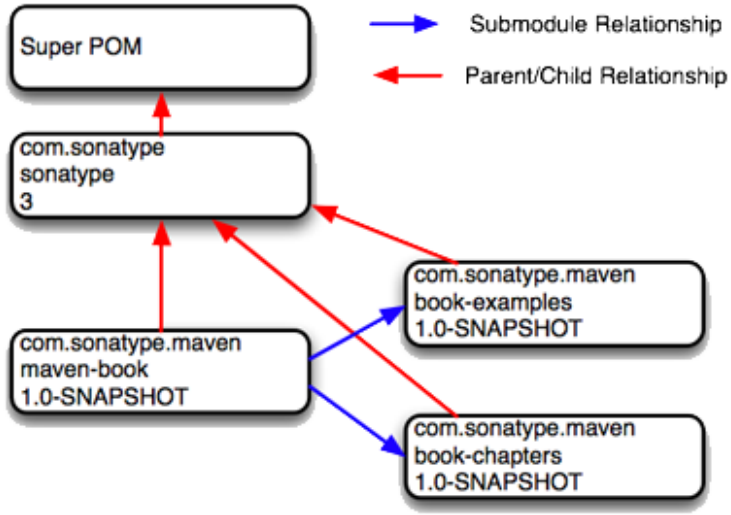
Maven is considerably more complex than Ant, and considerably more useful for projects using many external libraries (i.e., most Java projects).

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://www.w3.org/2001/
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
  <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`.)

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

Testing

is vital. And not just at the end!

(Unit) Tests should be written before the code – a mantra from Extreme Programming, but widely accepted.

More about testing later on – but do it now!

JUnit

JUnit is a simple framework for writing unit tests for Java programs.

For each class, write a *test class* which exercises methods and checks for correct functioning.

The framework makes it easy to run all tests frequently (many times per day).

For now, see junit.org for more information and introductory tutorial.

Eclipse has a plug-in to integrate JUnit.

Debugging

When tests fail, you'll probably have to debug.

Revise Inf1 lab on the Java debugger included with Eclipse.

When your attempt at debugging fails

You run your code inside a debugger; it still does something wrong; you don't understand why. Now what?

Some strategies to hone (in no particular order):

1. RTFM
2. isolate the bug: make the smallest, simplest subset of your code that still exhibits the problem, and then debug that
3. google for someone else having the same problem (especially good if you get an obscure error message from a widely used thing: if you don't understand it, there'll be others who've asked about it)
4. explain to a friend, in great detail, why your code should work – they usually won't have to say anything, or even listen!
5. ask someone, or post to a newsgroup (with your small simple example)
6. think of another way to do it
7. work on something else instead for now
8. experience...

Skills

(If you haven't already as you do the coursework...) Make sure you can:

- use SVN or CVS or GIT to retrieve a copy of a Sourceforge etc. project
- set up and use some kind of version control on your own projects, e.g. assignments
- debug a Java program in Eclipse or an IDE of your choice
- (need not be this week) write and run JUnit unit tests for a Java program
- build a project that uses a makefile or Ant/Maven build script
- (optional) understand and write makefiles and Ant/Maven build scripts.

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`

Suggested: browse `http://www.junit.org`.

Quote of the day

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Dijkstra