Version Control

V22.0474-001 Software Engineering
Lecture 12, Spring 2008

Clark Barrett, New York University

Configuration Management

- *Configuration Management* refers to a set of procedures for managing an evolving software system. It typically includes the following:
  - Version control
  - Support for automated system building
  - Support for automated system testing/bug-tracking
  - Support for multiple platforms
  - Release management
- In this lecture, we focus on version control. We will come back to the others later.

All Software Has Multiple Versions

- Different releases of a product
- Variations for different platforms
  - Hardware and software
- Versions within a development cycle
  - Test release with debugging code
  - Alpha, beta of final release
- Each time you edit a program

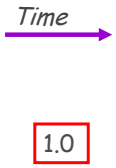
Version Control

- *Version control* tracks multiple versions
- In particular, allows
  - old versions to be recovered
  - multiple versions to exist simultaneously
- Why use version control?

Why Use Version Control?

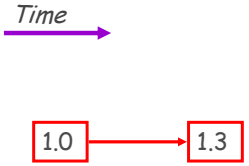
- To allow more than one developer to work on the code
- Can easily recreate old versions
- Change log
- Comparison with old versions very useful for debugging
- May need multiple versions of the same project

Scenario I: Bug Fix



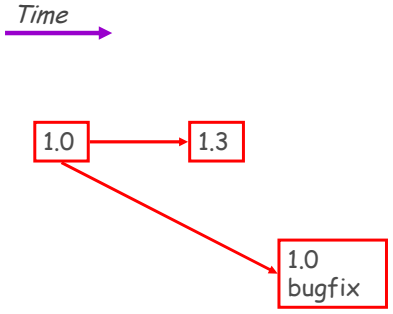
First public release of the hot new product

Scenario I: Bug Fix



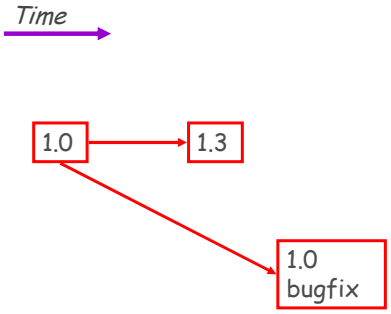
Internal development continues, progressing to version 1.3

Scenario I: Bug Fix



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

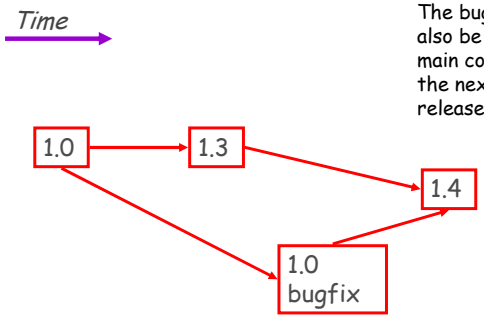
Scenario I: Bug Fix



Note that there are now two lines of development beginning at 1.0.

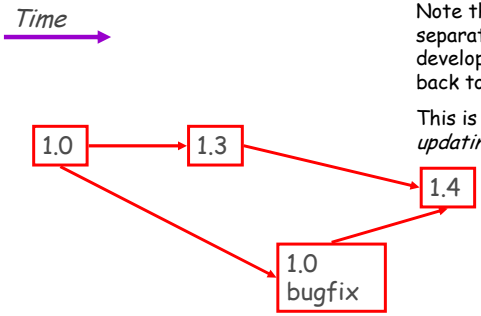
This is *branching.*

Scenario I: Bug Fix



The bug fix should also be applied to the main code line so that the next product release has the fix.

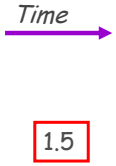
Scenario I: Bug Fix



Note that two separate lines of development come back together in 1.4.

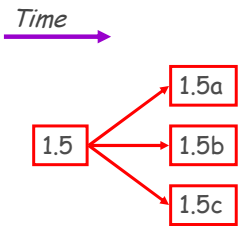
This is *merging* or *updating.*

Scenario II: Normal Development



You are in the middle of a project with three developers named a, b, and c.

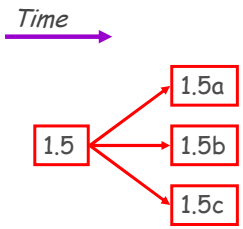
Scenario II: Normal Development



At the beginning of the day everyone *checks out* a copy of the code.

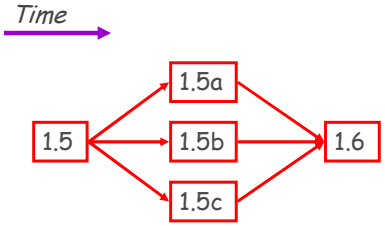
A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.

Scenario II: Normal Development



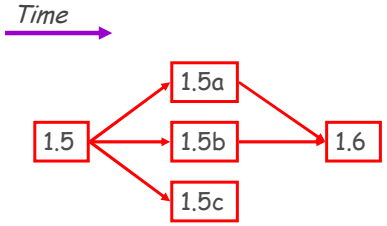
The local versions isolate the developers from each other's possibly unstable changes. Each builds on 1.5, the most recent stable version.

Scenario II: Normal Development



At 4:00 pm everyone *checks in* their tested modifications. A check in is a kind of merge where local versions are copied back into the version control system.

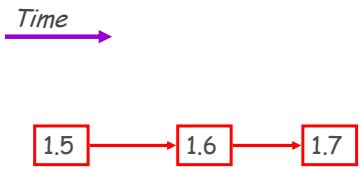
Scenario II: Normal Development



In many organizations check in automatically runs a test suite against the result of the check in. If the tests fail the changes are not accepted.

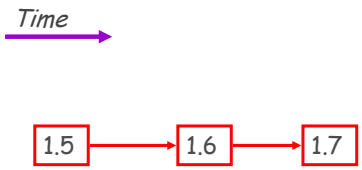
This prevents a sloppy developer from causing all work to stop by, e.g., creating a version of the system that does not compile.

Scenario III: Debugging



You develop a software system through several revisions.

Scenario III: Debugging



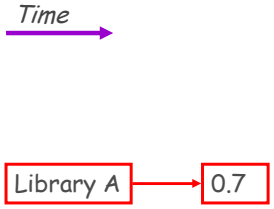
In 1.7 you suddenly discover a bug has crept into the system. When was it introduced?

With version control you can check out old versions of the system and see which revision introduced the bug.

Scenario IV: Libraries

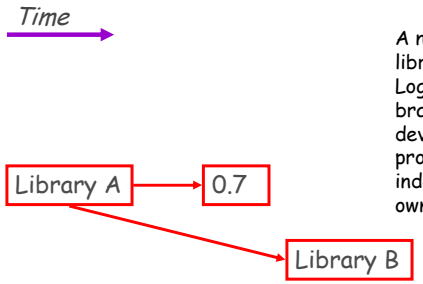
You are building software on top of a third-party library, for which you have source.

Scenario IV: Libraries



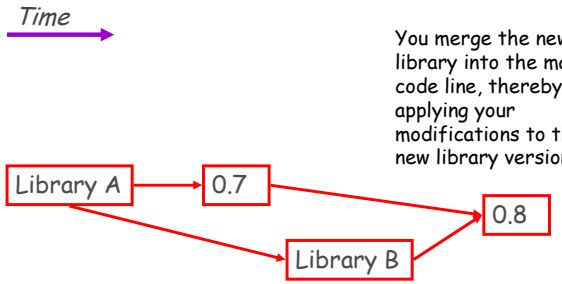
You begin implementation of your software, including modifications to the library.

Scenario IV: Libraries



A new version of the library is released. Logically this is a branch: library development has proceeded independently of your own development.

Scenario IV: Libraries



You merge the new library into the main code line, thereby applying your modifications to the new library version.

CVS

- CVS stands for "Concurrent Versions System"
- It is a commonly-used open-source program for version control.
- I recommend you use CVS for your project.
- More information available at http://ximbiot.com/cvs/

Concepts

- Repository
- Projects
- Working Copy
- Revisions
- Merging
- Conflicts
- Branches

Repository

- In order to use CVS, you must have a *repository*, a location which CVS can use to store information about multiple versions
- Typically, one repository may hold many different projects

Creating a Repository Using CVS

- To create a repository, you must specify an *absolute* pathname
- Create a repository using the following CVS command: cvs –d <directory/name> init
- For example, the following command creates a repository in the current directory:
  > cvs –d $PWD/repository init

Sharing a Repository

- If you want others in your group to be able to access your repository, your home directory must be group writable:
  > chmod g+w ~/.
- Also, the repository must belong to the group and be writable:
  > chgrp –R <nutlog|game> repository
  > chmod g+rwx repository

Projects

- A *project* is a set of files in version control
- Version control doesn't care what files
  - Not a build system
  - Or a test system
    - Though there are often hooks to these other systems
  - Just manages versions of a collection of files

Creating a Project in CVS

- Suppose you want to use CVS to track changes to a set of files in directory <myproj>
  - > cd <myproj>
  - > cvs –d <repository> import –m "Imported sources" <myproj> <myproj> start
- To create an empty project, just import from an empty directory:
  - > mkdir myproj
  - > cd myproj
  - > cvs –d <repository> import –m "Imported sources" myproj myproj start

Creating a Project in CVS

- If you get an error about being unable to create a temporary file, set the shell environment variable TMPDIR as follows:
  - > export TMPDIR=/tmp
- OR, if you are running csh or tcsh:
  - > setenv TMPDIR /tmp

Creating a Working Copy

- Once a project has been imported into the CVS repository, anyone can create a working copy:
  - > newgrp <nutlog|game>
  - > cvs –d <repository> co <myproj>
  - > cd <myproj>
  - > ls
- The working copy contains a directory CVS which stores information about the project and the repository.

Adding a file

- To add a file to the project, first create it, then use the cvs add and cvs commit commands
- Example
  - > emacs main.cpp
  - > <Type in code and then Ctrl-X Ctrl-C>
  - > cvs add main.cpp
  - > cvs commit –m "initial revision" main.cpp

Adding a directory

- Adding a directory is almost the same as adding a file:
- Example
  - > mkdir newModule
  - > cvs add newModule
  - > cvs commit –m "initial revision" newModule
- Note that you may need to fix the permissions in the repository after adding a directory:
  - > cd <repository>
  - > chgrp –R <nutlog|game> .
  - > chmod g+rwx *

Working with Multiple Copies

- CVS allows multiple users to edit files at the same time. The process works like this:
- First, update your working copy:
  - > cvs update –d
- Next, make changes to your working copy
- Finally, commit your changes:
  - > cvs commit –m "My changes"

Revisions

- Consider
  - Check out a project
  - Edit some files
  - Check the files back in
- This creates a new version of each file
  - Usually increment minor version number
  - E.g., 1.5 -> 1.6

Revisions (Cont.)

- Observation: Most edits are small
- For efficiency, don't store entire new file
  - Store diff with previous version
  - Minimizes space
  - Makes check-in, check-out potentially slower
    - Must apply diffs from all previous versions to compute current file
    - In practice, not significant

Revisions (Cont.)

- With each revision, system stores
  - The diffs for that version
  - The new minor version number
  - Other metadata
    - Author
    - Time of check in
    - Log file message

Merging

- Start with a file, say main.cpp, revision 1.1
- Alice makes changes A to 1.1
- Bob makes changes B to 1.1
- Assume Bob checks in first
  - Current revision is 1.2 = apply(B,1.1)

Merging (Cont.)

- Now Alice checks in
  - System notices that Alice had checked out 1.1
  - But current version is 1.2
  - Alice has not made her changes in the current version!
- The system complains
  - Alice is told to *update* her local copy of the code

Merging (Cont.)

- Alice does an update
  - This applies Bob's changes B to Alice's code
    - Remember Alice's code is apply(A,1.1)
    - Current version is 1.2 = apply(B, 1.1)
- Two possible outcomes of an update
  - Success
  - Conflicts

Success

- Assume that
  apply(A,apply(B,1.1)) = apply(B,apply(A,1.1))
- Then order of changes didn't matter
  - Same result whether Bob or Alice checks in first
  - The version control system is happy with this
- Alice can now check in her changes
  - Obtaining 1.3 = apply(A, 1.2) = apply(A, apply(B,1.1))

Failure

- Now, assume Alice and Bob make new changes to version 1.3 and
  apply(A',apply(B',1.3)) ≠ apply(B',apply(A',1.3))
- There is a *conflict*
  - The order of the changes matters
  - Version control will complain

Conflicts

- Arise when two programmers edit the same piece of code
  - One change overwrites another

1.3:	"Hello world"
Alice:	"Hello world, my name is Alice"
Bob:	"Hello world, my name is Bob"

The system doesn't know what should be done, and so complains of a conflict.

Conflicts (Cont.)

- System cannot apply changes when there are conflicts
  - Final result is not unique
  - Depends on order in which changes are applied
- Version control shows conflicts on update
- Conflicts must be resolved by hand
  - Symptom of bad communication

Conflicts are Syntactic

- Conflict detection is based on "nearness" of changes
  - Changes to the same line will conflict
  - Changes to different lines will likely not conflict
- Note: Lack of conflicts *does not* mean Alice's and Bob's changes work together

Example With No Conflict

- Revision 1.6: `int f(int a, int b) { … }`
- Alice: `int f(int a, int b, int c) { … }`
  add argument to all calls to `f`
- Bob: add call `f(x,y)`
- Merged program
  - Has no conflicts
  - But will not even compile

Don't Forget

- Merging is syntactic
- Semantic errors may not create conflicts
  - But the code is still wrong
  - You are lucky if the code doesn't compile
    - Worse if it does . . .
  - Rare in practice, if you maintain good team communication

Branches

- A branch is just two separate revisions of a file that do not get merged
  - Two people check out 1.5
  - Check in 1.5.1
  - Check in 1.5.2
- Notes
  - Normally checking in does not create a branch
    - Changes merged into main code line
  - Must explicitly ask to create a branch
  - Must explicitly merge branches

CVS Tags

- Some operations require a snapshot of the global project state
  - Branching
  - Major releases
- CVS can *tag* a project with a name

CVS Remote Repository

- Normally, repository is on the local file system
  - Hard to collaborate between distributed teams
- CVS can run in client-server mode
  - Server machine runs ssh server and keeps the repository
  - Client machine queries the server
  - Only diffs are being sent => fast even on slow net
  - Converts automatically MSDOS/Unix files
  - Use it to keep source files synchronized