**Version Control**

CS195  
Lecture 10

---

**Outline**

- What is version control?  
  - And why use it?  
  - Scenarios

- Basic concepts  
  - Projects  
  - Branching  
  - Merging  
  - Conflicts

- Two systems  
  - RCS  
  - CVS

---

**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?**

- Because everyone does  
  - A basic software development tool

- Because it is useful  
  - You will want old/multiple versions  
  - Without version control, can’t recreate project history

---

**Scenario 1: Bug Fix**

- First public release of the hot new product
Scenario I: Bug Fix

Time

Releases

1.0 – 1.3

Internal development continued, progressing to version 1.3

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

Scenario II: Normal Development

Time

Releases

1.0 – 1.3 – 1.4

Note that two separate lines of development came back together in 1.4. This is merging or updating

Scenario I: Bug Fix

Time

Releases

1.0 – 1.3

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.

1.0 bugfix

Scenario I: Bug Fix

Time

Releases

1.0 – 1.3

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

1.0 bugfix

1.4
**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.

---

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

---

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.

---

In many organizations check in automatically run 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 coding 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.

---

In 1.7 you suddenly discover a bug that 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.

Library A

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

Library A

0.7

Library B

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

Library A

0.7

Library B

0.8

Concepts

- Projects
- Revisions
- Branches
- Merging
- Conflicts

Projects

- A project is a set of files in version control
  - Called a module in CVS

- 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
Assumption

- Consider a project with 1 file
- We will return to the multiple file case later

Revisions

- Consider
  - Check out a file
  - Edit it
  - Check the file back in
- This creates a new version of the 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

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
    - Results of "smoke test"

Branches

- A branch is just two revisions of a file
  - 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

Merging

- Start with a file, say 1.5
- Bob makes changes A to 1.5
- Alice makes changes B to 1.5
- Assume Alice checks in first
  - Current revision is 1.6 = apply(8,1.5)
Merging (Cont.)

- Now Bob checks in
  - System notices that Bob checked out 1.5
  - But current version is 1.6
  - Bob has not made his changes in the current version!
- The system complains
  - Bob is told to update his local copy of the code.

Success

- Assume that
  \[ \text{apply}(A, \text{apply}(B,1.5)) = \text{apply}(B, \text{apply}(A,1.5)) \]
- Then then order of changes didn’t matter
  - Same result whether Bob or Alice checks in first
  - The version control system is happy with this
- Bob can now check in his changes
  - Because \( \text{apply}(B, \text{apply}(A,1.6)) = \text{apply}(B,1.6) \)

Failure

- Assume
  \[ \text{apply}(A, \text{apply}(B,1.5)) \neq \text{apply}(B, \text{apply}(A,1.6)) \]
- 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

  | 1.5: | a = b; |
  | Alice: | a = b++; |
  | Bob: | a = ++b; |

  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
  - Generally based on diff3
- Conflicts must be resolved by hand
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 1.5: `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 . . .

Two Systems

- We discuss
  - CVS
    - De facto free software standard for version control
  - PRCS
    - Hitfinger, et al.
- For single file projects, these are the same
  - Except for administration

PRCS Model

- Operations are on the project
  - Not on individual files
- Example
  - Project version 1.5
  - Check out
  - Update file foo.bar
  - Check in
  - Project version is now 1.6

PRCS Model (Cont.)

- Changes to individual files treated as changes to the project
- Every state of the project has a name
  - E.g., 1.6
- Makes it possible to recover any point in the project history
**CVS Model**

- Operations are on files
- Example
  - Check out
  - Modify foo.bar revision 2.7
  - Check in
  - foo.bar now revision 2.8

**CVS Tags**

- Some operations require a snapshot of the global project state
  - Branching
  - Major releases
- CVS can tag a project with a name
  - A separate operation to do what PRCS does for every change

**Administration**

- PRCS has a simple administrative model
  - One file with all metadata in a standard format
    - Really, a small project programming language
  - Administration done by text editing
  - The administrative file is under version control, too
    - Get old project versions by checking out old adms in files
- CVS administration is much more complex
  - Numerous files, information scattered throughout
    - One admin file per file under CVS
    - Makes renaming, moving files awkward

**Design**

- Version control of projects is about snapshots of sets of files
  - PRCS represents this directly
  - CVS is oriented toward individual files
    - And it shows in complexity
  - A lesson here for those interested in software design . . .

**Trade-offs**

- CVS has many more features than PRCS
  - In particular, remote repositories
  - Allows distributed work over ssh
  - If you don’t need remote check in/check out, PRCS may be a better choice