Prifysgol Abertawe
Swansea University

CS-230 Software Engineering

L10: Version Control Systems

Dr. Liam O'Reilly

Semester 1 – 2020

Many of these slides have been adapted from Clark Barrett (New York University) and Prof. Necula (Berkeley)

Previously in CS-230 …

Wrong

```
static public void main(String[] args) {
    ...
```

Correct

```
public static void main(String[] args) {
    ...
```

Coding Conventions

Previously in CS-230… (2)

What are coding conventions?

- Rules for:
  - Laying out code, e.g., spaces.
  - Naming variables/classes/etc.
  - Limiting length of lines, methods, etc.
  - Controlling the basic style of code.

How many coding conventions are there?

- Every company has it’s own style.

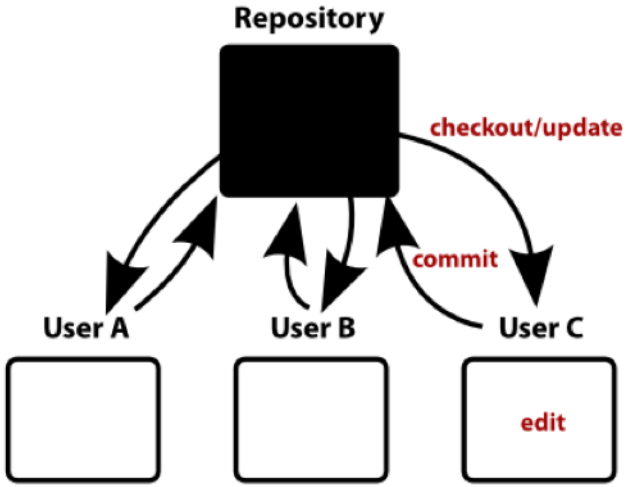
Previously in CS-230… (2)

Why should we bother with coding conventions?

- Makes code easier to read once you are used to the standard.
- Good coding conventions can help avoid bugs.
- Makes working with version control systems easier.
- Helps make code more reuseable.
- Coding conventions will be marked as part of A2.

Today

Version Control Systems



Outline

- What is version control?
  - And why use it?
  - Scenarios
- Basic concepts
  - Projects
  - Branches
  - Merging
    - Conflicts
- Two systems
  - SVN (Subversion)
  - Git

Nice Video

- Git and Subversion (svn) are open source version control systems.
- Below is a link to a nice video which explains the Git Version Control System.
- https://www.youtube.com/watch?v=OqmSzXDrJBk
- Subversion is a bit old now. I would recommend using Git.

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 and final releases.
- Each time you edit a program you create a new version.

Version Control

- Version control tracks changes to a set of files.

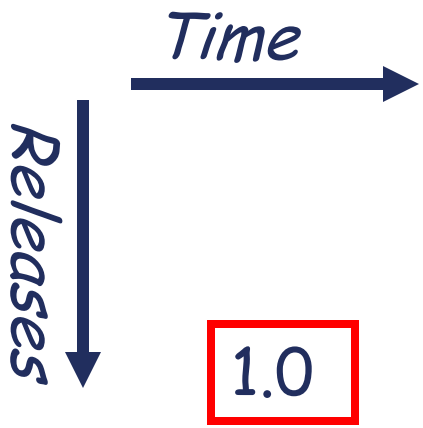
- In particular such system generally, allows
  - To work on a local checkout.
  - To turn back time on the local checkout and see/recover what it looked like at certain points in past.
  - To branch the development.
  - To merge existing branches.
  - To see (blame?) who made certain changes.
  - Allow all versions to exist simultaneously in a central repository.
  - Store the changes efficiently taking up minimal space.

Why Use Version Control?

- Because it is useful:
  - You will want old/multiple versions.
  - Without version control, can't recreate project history.
  - It allows teams to work together on the same code.

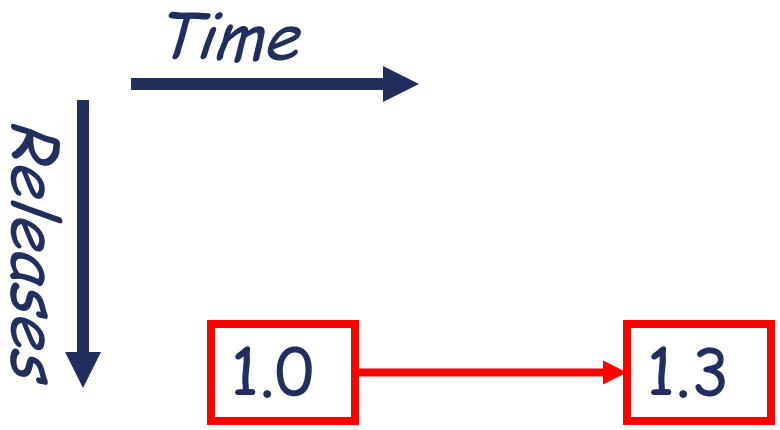
- Because we require it:
    - It really a core software development tool these days.
  - Everyone is using one of them.
  - After using one properly, you will wonder how people managed code without them.

Scenario I: Bug Fix



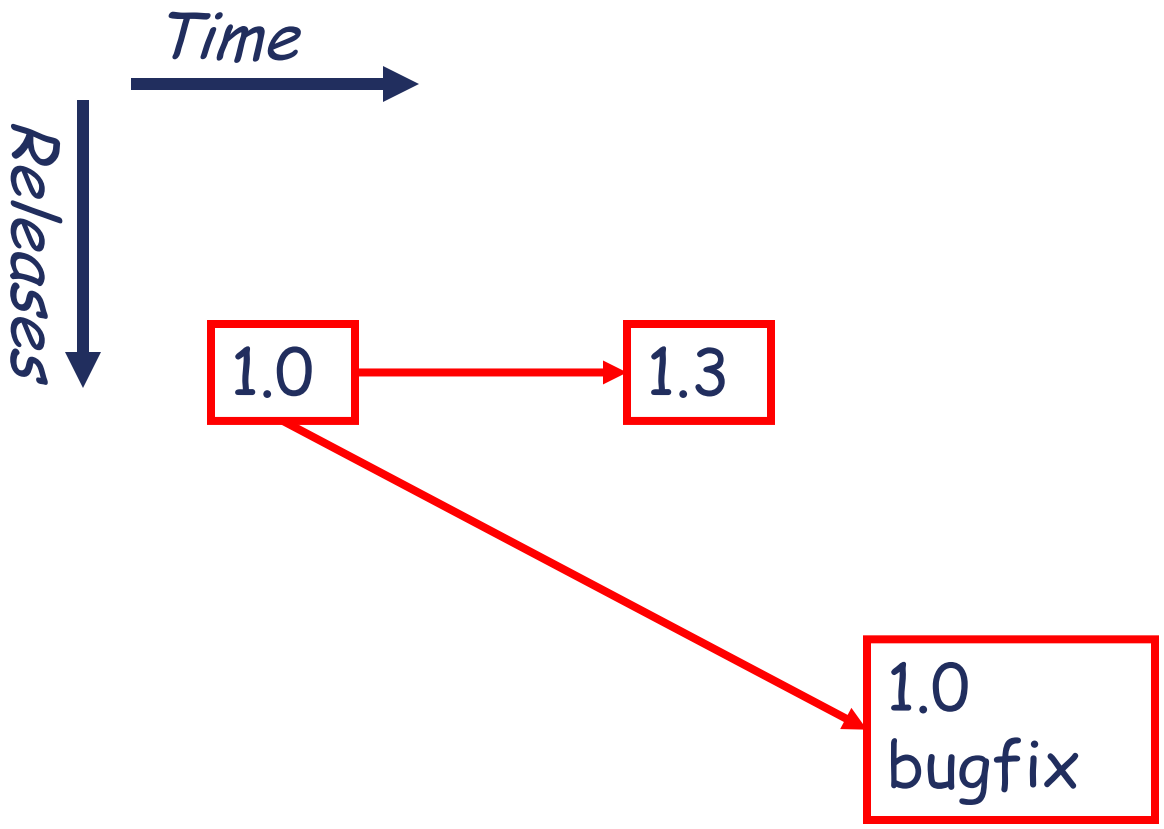
First public release of the hot new product

Scenario I: Bug Fix



Internal development continues, progressing to version 1.3

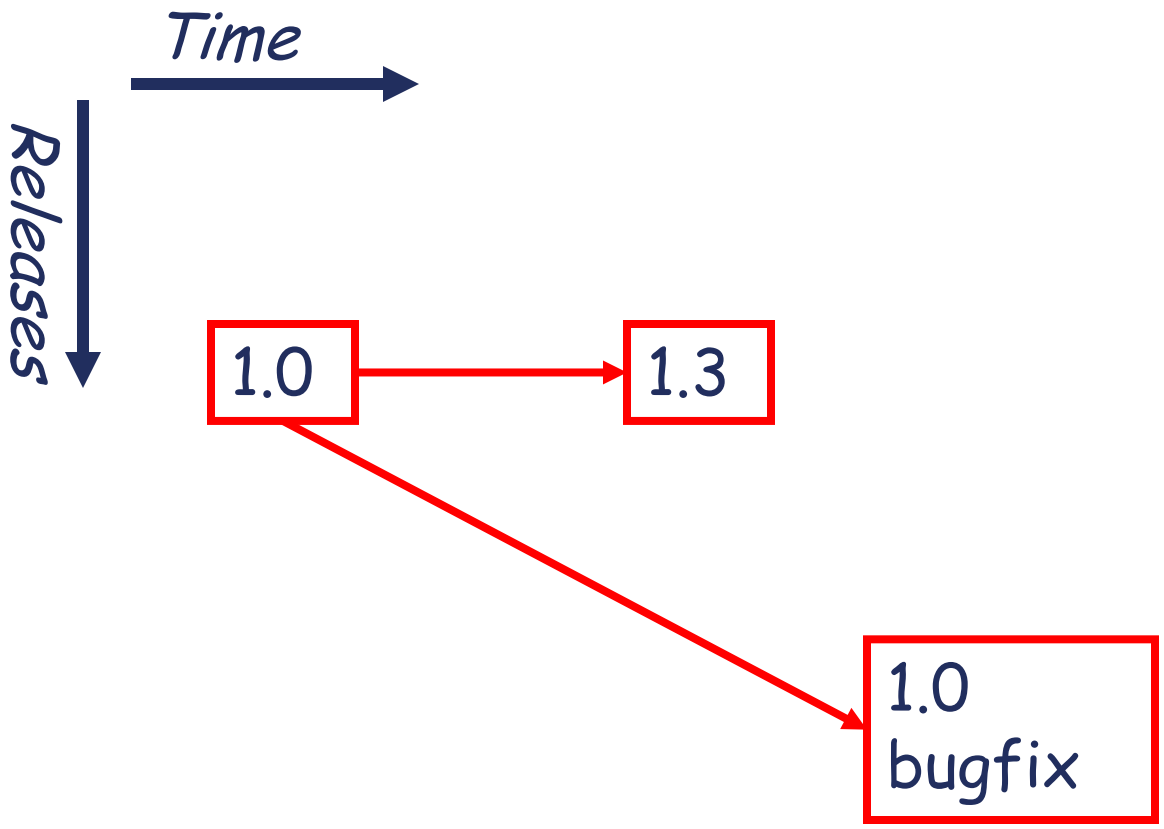
Scenario I: Bug Fix



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.

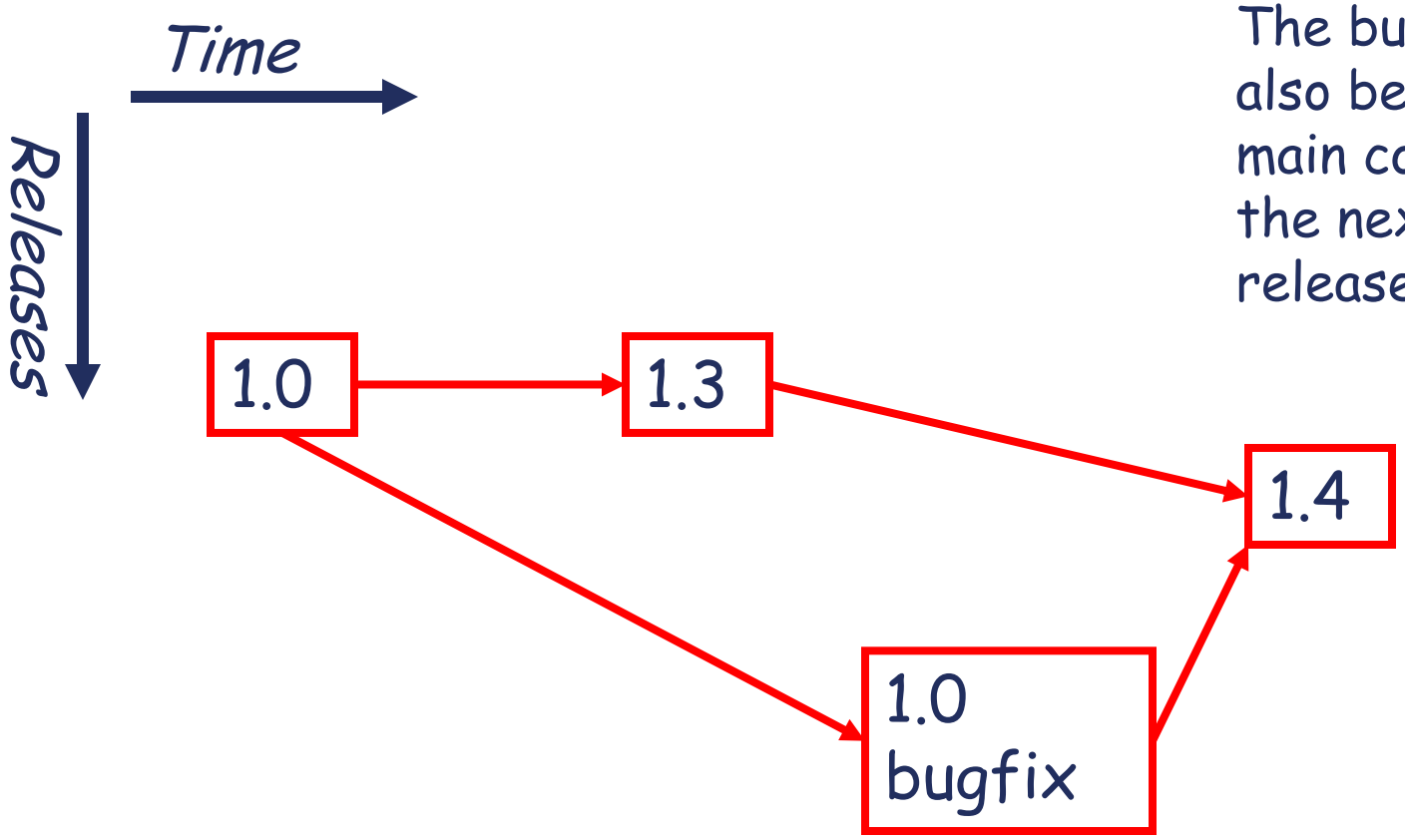
Scenario I: Bug Fix



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

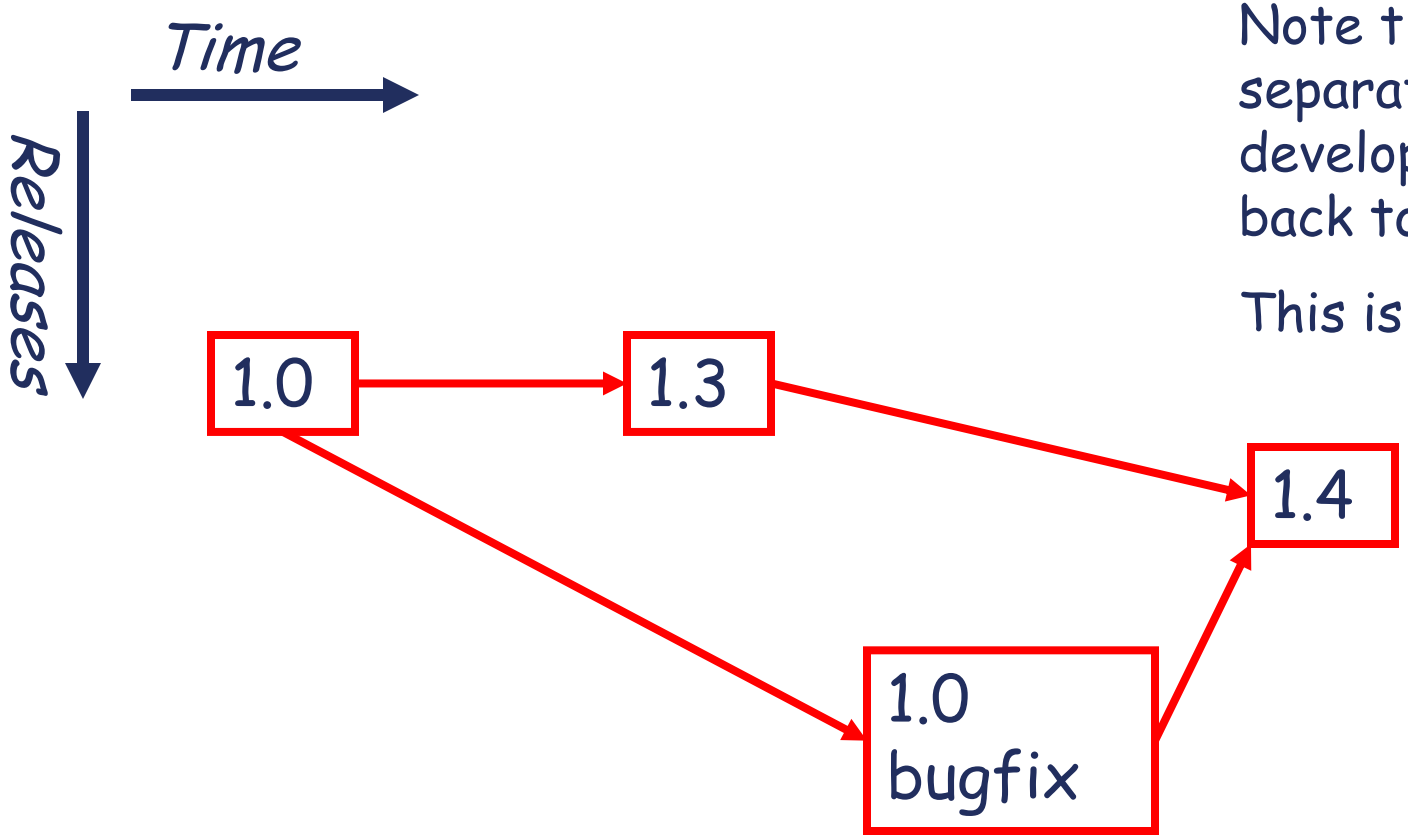
This is *branching*.

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.

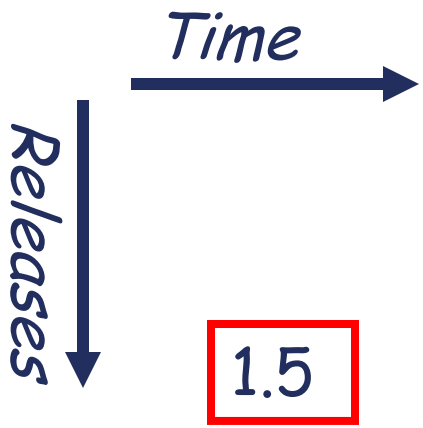
Scenario I: Bug Fix



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

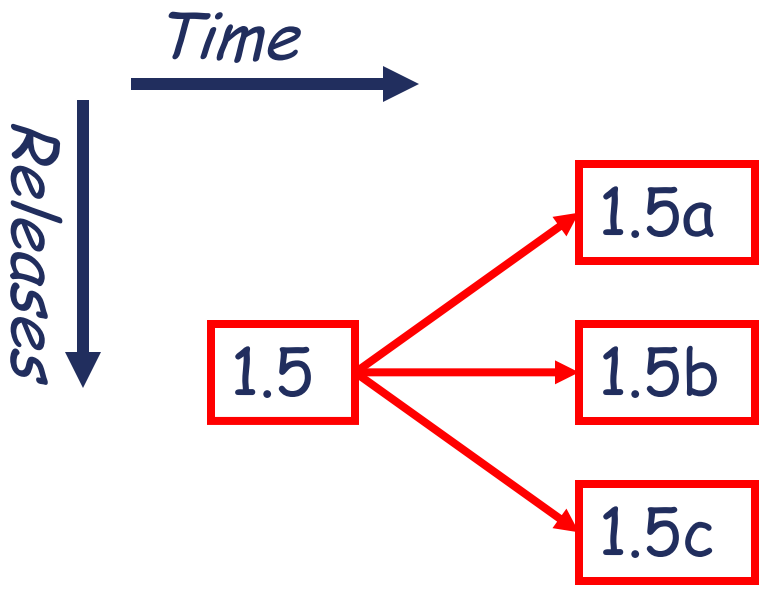
This is merging.

Scenario II: Normal Development



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

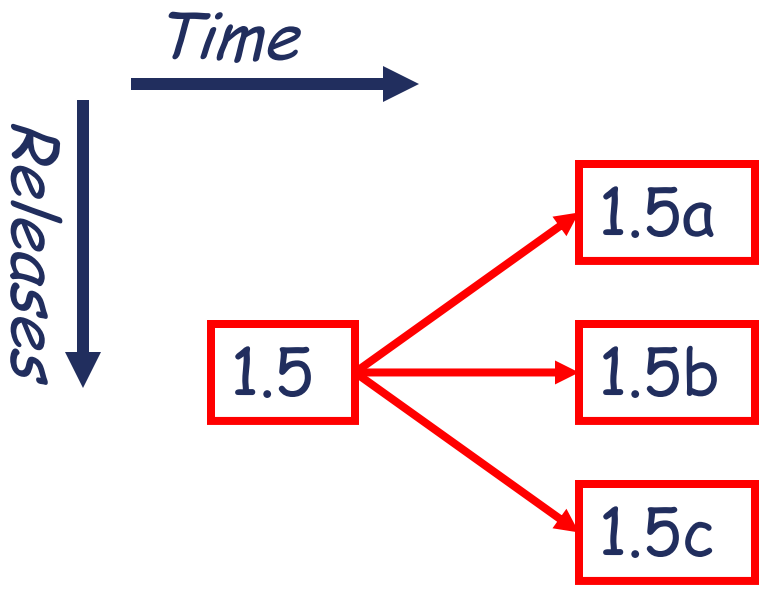
Scenario II: Normal Development



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

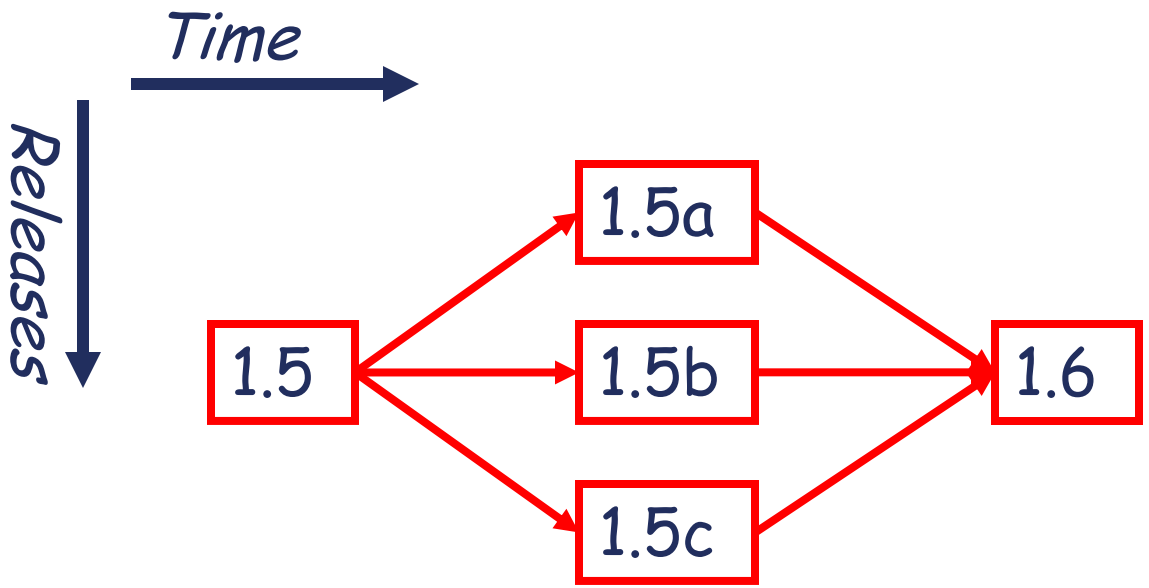
A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.

Scenario II: Normal Development



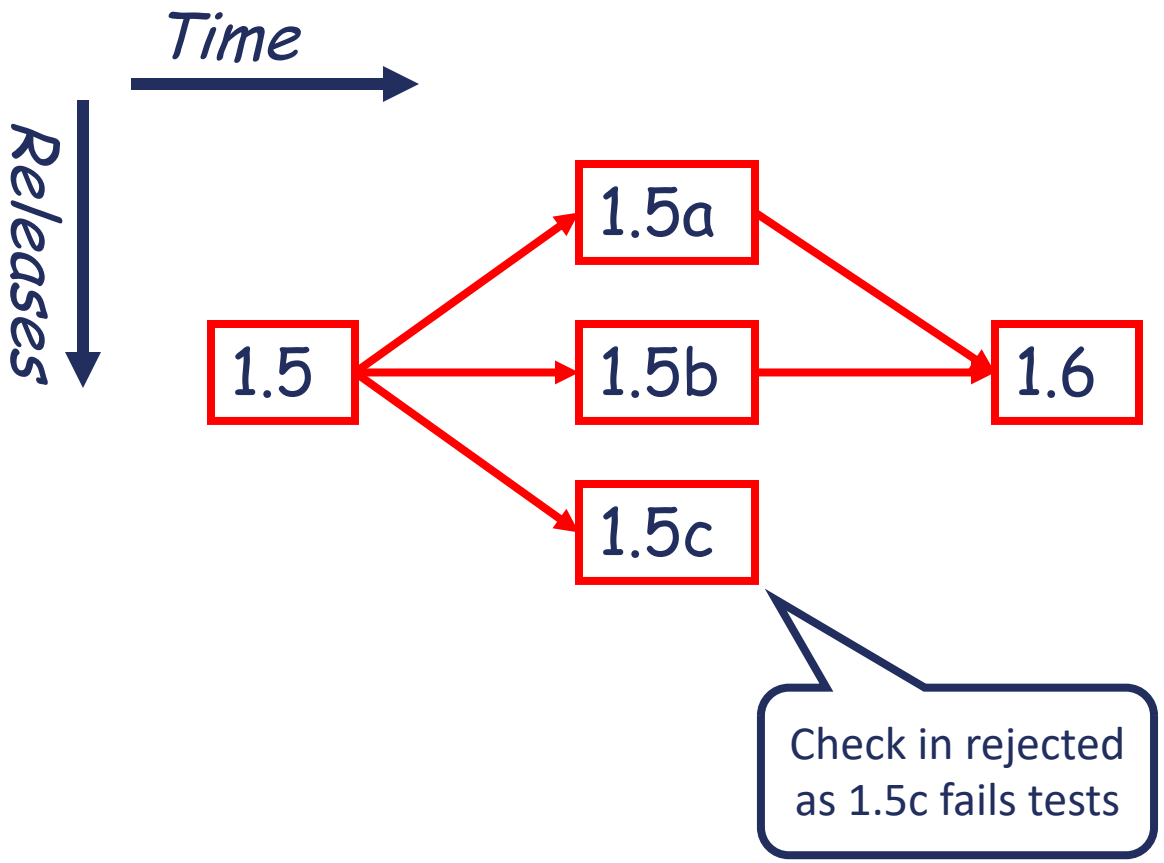
The local versions isolate the developers from each other's possibly unstable changes. Each builds on 1.5, the most recent stable version.

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.

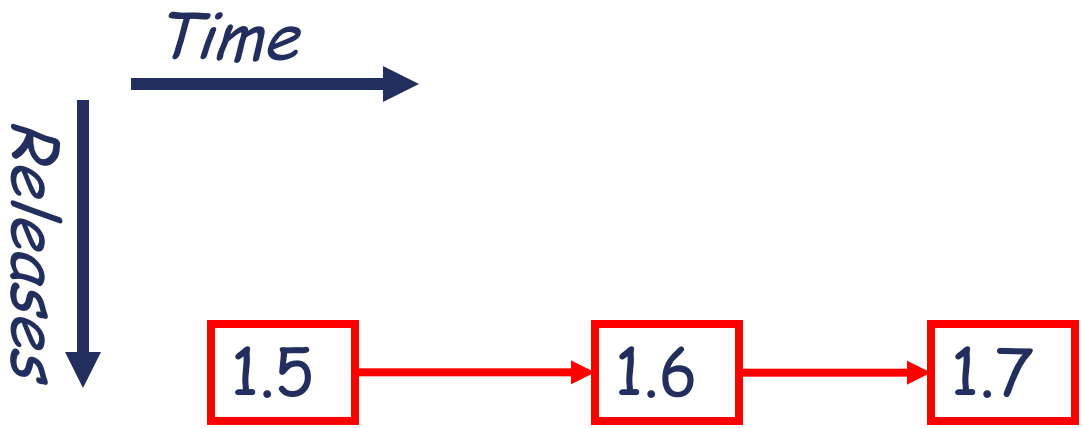
Scenario II: Normal Development



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

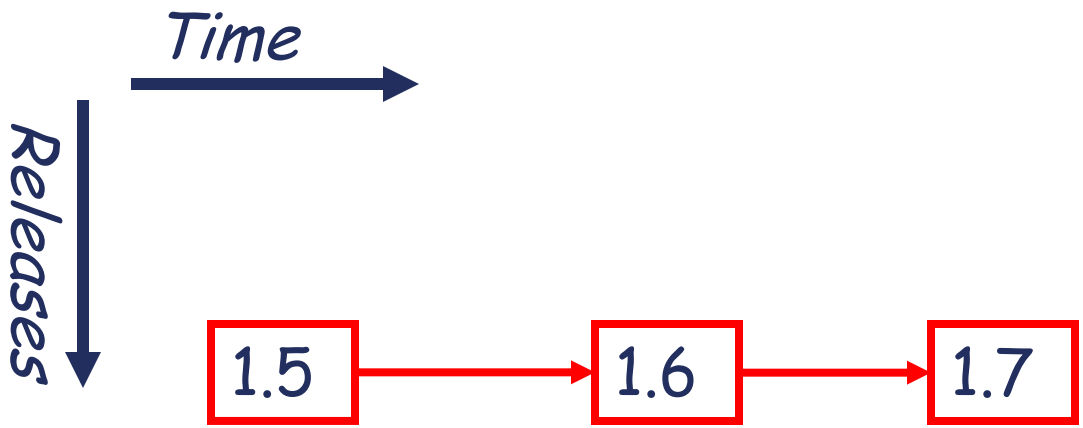
This prevents a sloppy developer from causing all work to stop by, e.g., creating a version of the system that does not compile.

Scenario III: Debugging



You develop a software system through several revisions.

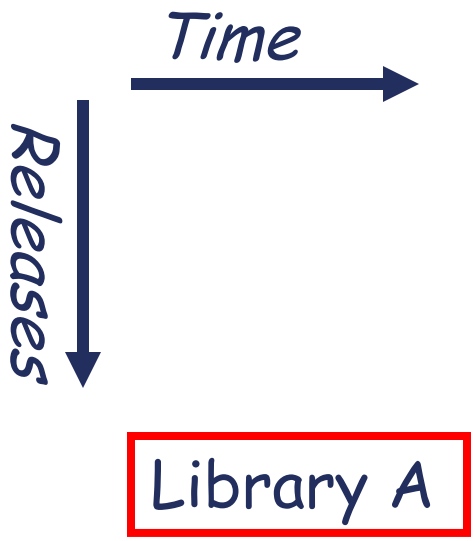
Scenario III: Debugging



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

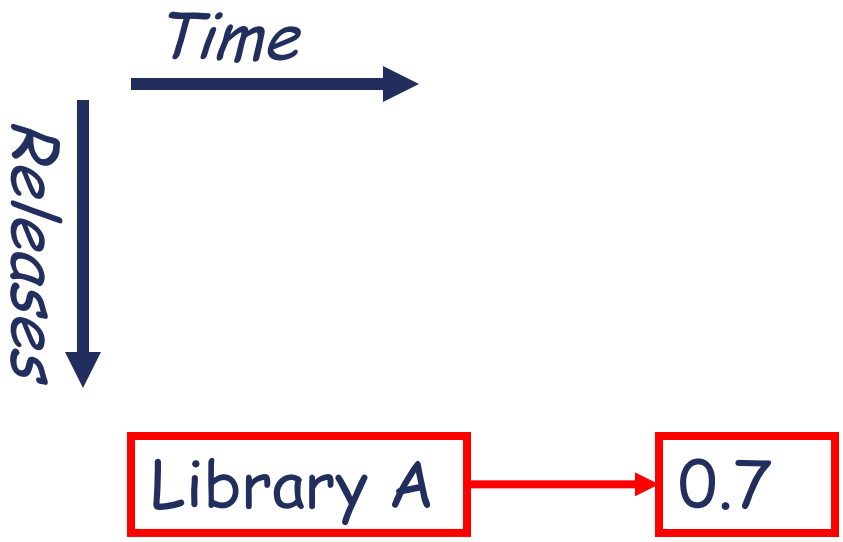
With version control you can check out old versions of the system and see which revision introduced the bug.

Scenario IV: Libraries



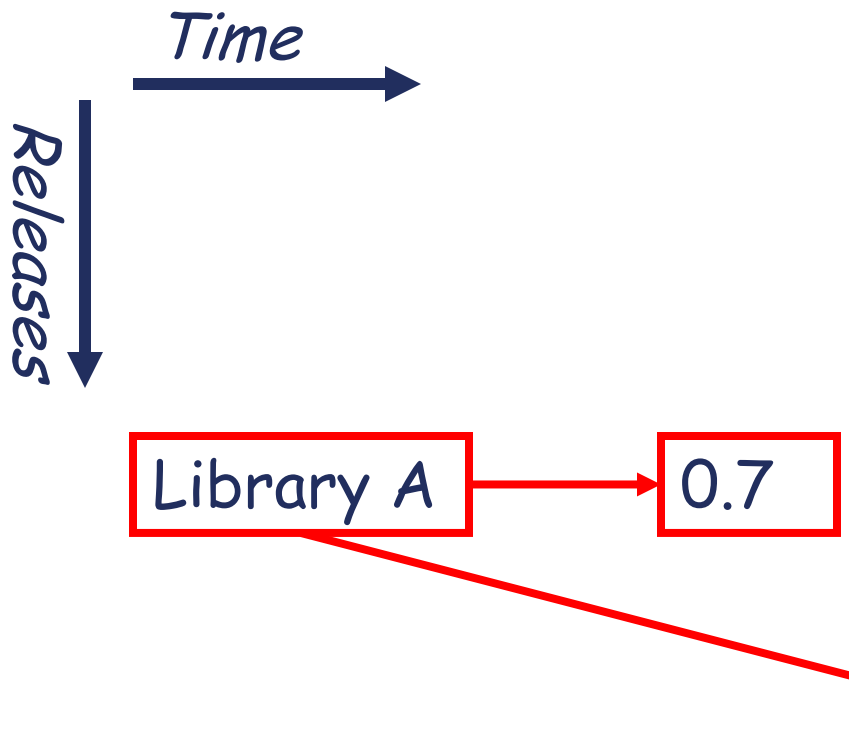
You are building software on top of a third-party library, for which you have source.

Scenario IV: Libraries



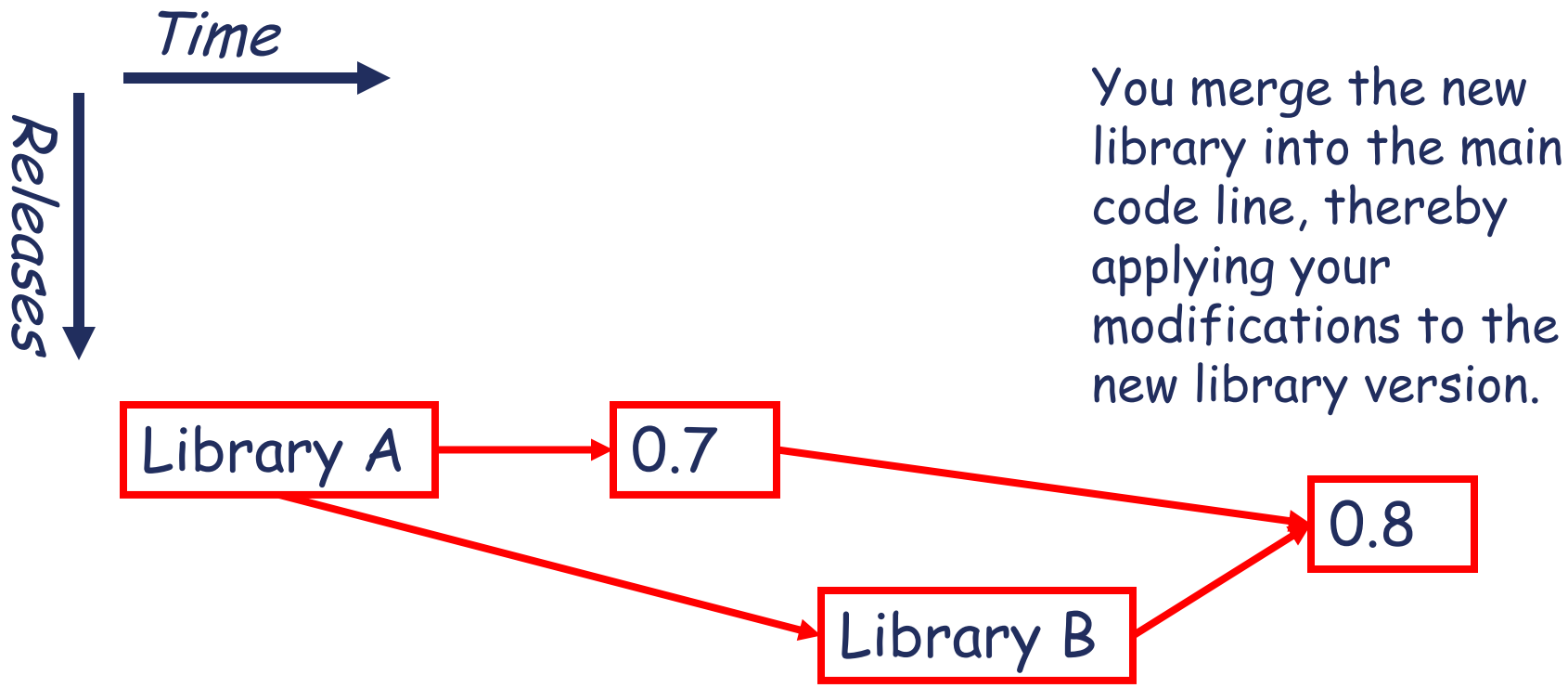
You begin implementation of your software, including modifications to the library.

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.

Concepts

- Projects
- Revisions
- Branches
- Merging
- Conflicts

Projects

- A project/repository 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.
  - Generally, version control works better on text files (e.g., .java, .c, .txt, .xml); as opposed to binary files (e.g., .class, .exe, .jpg)

Revisions

- Consider:
  - Check out a file from a central repository.
  - Edit it.
  - Check the file back in to the central repository.

- This creates a new version of the file.
  - Usually increment some version number.
  - E.g., 1.5 -> 1.6

Revisions (Cont.)

- Observation: Most edits are small (and should be!)
  - They should change one thing in one way. Then we can write a change/commit message that describes the one change.

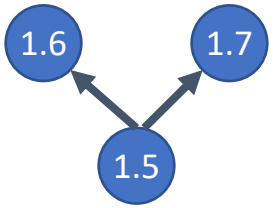
- For efficiency, (most) version control systems do not store the whole file each time a change is made:
  - Stores diff with previous version.
  - Minimizes space.
  - Makes check-in, check-out potentially slower.
    - Must apply diffs from all previous versions to compute current file.

Revisions (Cont.)

- With each revision, system stores
  - The diffs for that version.
  - The new version number.
  - Other metadata:
    - Author/person checking file in.
    - Time of check in.
    - Log file message – absolutely critical information for serious development.

Branches

- A branch is just two different revisions of a file that are tracked slightly differently.
  - Two people check out 1.5.
  - User A checks in 1.6.
  - User B checks in 1.7.



- Note:
  - Normally checking in does not create a branch.
    - Changes merged into main code line.
  - Must explicitly ask to create a branch:
    - Branches need to be controlled; It would not be nice to accidentally create them all over the place.

Assumption

- Consider a project with 1 file for the next slides.
- We will return to the multiple file case later.

Merging

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

Merging (Cont.)

- Now Alice attempts to checks in her changes.
  - Version control system knows that Alice checked out 1.5.
  - But current version is 1.6.
  - Alice has not made her changes in the current version!
- The version control system reports an error:
  - Alice cannot check in her changes.
  - Alice is told to update her local copy of the code with what is in the central repository.

Merging (Cont.)

- Alice does an update.
  - This attempts to apply Bob's changes B to Alice's code.
    - Remember Alice's code is apply(A,1.5).

- Two possible outcomes of this update:
  - Success.
  - Failure due to conflicts.

Merging – Success

- Assume that:

  apply(A, apply(B,1.5) = apply(B, apply(A,1.5))

- Then then order of changes didn't matter:
  - Same result whether Alice or Bob checks in first.
  - The version control system is happy with this.

- Alice successfully merges Bob's changes to the central repository with her local checkout.
- She can now check in her changes as she is up-to-date.

Merging – Failure

- Assume now instead that:

  $$\text{apply}(A, \text{apply}(B, 1.5) \neq \text{apply}(B, \text{apply}(A, 1.5))$$

- There is a conflict:
  - The order of the changes matters.
  - Version control will complain and expect Alice to manually fix the conflict.

Conflicts

- Conflicts arise when two programmers edit the same piece of code in different ways:
  - One change overwrites another
  - 1.5: `a = b;`
  - Alice: `a = b + 1;`
  - Bob: `a = b - 1;`
  - 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 from central repository.
  - 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.
  - Code conventions help prevent unnecessary conflicts.
  - Version control systems can be configured to ignore some differences in white space.

- 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 and Bob now make changes at the same time.
  - Alice: int f(int a, int b, int c) { … }
    add argument c 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 syntactic conflicts.
  - But the code is still wrong.
  - You are lucky if the code doesn’t compile.
    - Worse if it does compile . . .
      - Better hope you have (automated) unit tests.

Various Version Control Systems

- We now look at an overview of:
  - Subversion (svn):
    - A popular centralised version control system
  - Git:
    - A newer, hugely popular distributed version control system

- Others:
  - Bazaar – made by Canonical (the company behind Ubuntu)
  - Mercurial
  - Team Foundation – made by Microsoft

Subversion Overview

Subversion Model

- Central server stores the main repository.
- Users checkout a local working copy.
- Users makes changes to local working copy.
- Users commit (check-in) changes to the main repository.

- Operations are on the project.
  - Not on individual files.
  - During a commit, either all selected files are checked in, or none are. It is atomic.

- Repository has a version number which is incremented with every commit (aka check-in).

Subversion Model: Example

- Repository version number: 1654
- Check out repository to create a local working copy
- Update file foo.bar
- Commit changes (this time there are no conflicts)
- Repository version number: 1655
- <Time passes – 1 month, others modify repository>
- Update local working copy to bring in changes from main repository. Repository version number: 1680
- Update file foo.bar
- Commit changes (but changes are refused as someone else changed the file foo.bar, repository now at version 1681)
- Update local working copy to bring in changes from main repository. Repository version number: 1681
- However, changes could not be merged automatically – conflict!!!
- Resolve conflict manually
- Commit changes (this time you were fast enough and no one else changed the repository)
- Repository version number: 1682

Subversion Model

- Changes to individual files treated as changes to the project
- Every state of the repository has a version number
  - E.g., 1655
- Makes it possible to recover a whole project to as it was in the past.
  - E.g., To as it was at version 1650.

Subversion: Features

- Branches
- Tags – Can label a repository version with a unique name e.g., “Release v2.1”

Subversion: Issues

- As it is centralised, if the server goes down, no one can commit changes.
  - This results in people continuing development in their local working copy without commit. Result: when they do commit – they commit everything in one huge lump!
  - Rolling back is then a problem – undo everything or nothing.
- Tags are actually branches, that you promise not to perform further development on!
- You cannot commit locally:
  - E.g., if you are developing offline with no network connection.
  - Also, if you get muddled up during a merge, then you can not get back to how the code was just before the merge.

Git Overview

Git

- A distributed version control system:
  - No central repository required.
  - People commit locally, then at some point push their commits to other repositories.
    - You can designate a repository to be a central one and then use git as a centralised system if you want.
      - E.g., github, bitbucket, or your own server, etc.
  - People can push their commits to other peoples local repositories.
  - People can pull changes from other people.
  - Advanced features: I can commit locally, say A,B, and C. Then I can combine (squish) all the changes into just one commit which I push, i.e., I can change the history.

Git: Features

- Branches
- Tags – Give a unique name to a commit.
  Unlike subversion, this is just a label (it is not a branch)
- Commits have SHA1 identifiers.
- Conflicts are easier to handle than svn, as changes are committed locally first (impossible (ish)) to get a conflict. Then commits are pushed elsewhere. If there is a conflict on a push, we can also resort back to our local commit.
- Massive flexibility as it is distributed:
  - Many ways to work with it
  - This can make it hard to learn.
- Many many more features:
  - Cherry picking
  - Rebasing
  - Git is complex and therefore can be hard to learn.

Summary

When working on code you need version control systems:

- For any code which is more complex than trivial.
- For working in teams.

- Recommendation: Use Git for new projects (not subversion).