Source Code Control

- Managing Source Code
- Tagging and Branching
- Merging between branches
- Source Code Collaboration
- CVS and Subversion
- Git
- Netbeans Git integration

Managing Source Code

- Programmers create, modify, delete, and/or arrange lines of code in order, usually spread across more than one source code file, to form a program
- Programmers often need to work with more than one *version* at a time:
  - Every single change made to the code entails two versions:
    - The version of the program before the change
    - The version of the program after the change

Changes and Commits

- The programmer decides what constitutes a **change**
  - For example, each single keystroke is not usually considered to be a change
  - Programmers naturally group keystroke-level modifications logically into changes
    - The modifications needed to fix a bug could be considered a change
    - The modifications needed to implement a feature could be considered a change
    - The programmer usually has a goal which dictates the required modifications that together form a change that achieves the goal
    - The programmer may identify a part of one of the above kinds of change to be a change
- When a programmer finishes making a change, they **commit** the change with a description of the change, adding the resulting version to the source code **repository**
- The simple case, after three commits, can be visualized like this

first version -> second version -> third version

Source Code Control (cont'd)

- The fundamental goal of source code control is to enable the programmer(s) to manage all of these changes
- Programmer working on a new program or version can decide
  - What changes should be included, no matter when that change occurred
  - What changes should be excluded, no matter when that change occurred
  - No change is ever lost
- All versions of a program are retrievable
- Best Practice is to supply a description of the change (comment) with each commit
- The versions that are **released** (copied for endusers) are distinguished from the other versions, and given names like Version 1.0, Version 1.01, Version 2.0, OSX 10.13.6, etc
  - Version naming has a lot to do with marketing and doesn't necessarily follow any logic or rules: OSX, Windows 10, High Sierra, Marshmallow

Tagging

- Source code control systems support assigning a name or label called a **tag** to a version
- Tags are arbitrary (can be anything)
- Tags help a programmer manage all the different versions, by distinguishing some of them in the repository

Branching

- Different versions of one program can be developed simultaneously, using branches
  - Common situation is that
    - a version will be released, say, Version 1.0, but it still needs development work
      - Bugs in Version 1.0 need to be fixed
      - Sometimes small features are added to a release
      - The bug fixes and small features correspond to 1.1, 1.2, 1.3, etc (or 1.01, 1.02, etc)
      - Often these are called **minor** versions, or minor releases
    - The next **major** release also needs to be developed at the same time (in parallel)
      - The next major release starts off untested, maybe unstable, maybe it won't even compile
      - After the next major release is finished, it will be called something like Version 2.0
      - Programmers need to work on Version 2.0's code without interfering with Version 1.0's code
- A simple branch situation in the repository looks like this

commit -> commit -> commit (Version 1.0) -> commit ->commit

\

MyBranch: -> commit->commit

Merging

- **Merging** allows the programmer to copy changes from one branch to another
- The bug fixes done on the Version 1.0 branch after Version 1.0 was released are also needed in the Version 2.0 branch, and vice versa
- It would be bad practice (inefficient and error-prone) to attempt to make the same bug-fix changes on both branches independently
- We make the bug-fix change on one branch, and **merge** that change to any other branches that should have that change
- Version 1.0.x and (what will become) Version 2.0 usually have many lines of code in common
- A change to Version1.0.x's "line 10" would merge to Version 2.0 by simply turning Version 2.0's new "line 10" into Version 1.0.x's changed "line 10"

Conflicts while Merging

- Sometimes, “line 10” (from the previous slide) will be changed into something in the Version 1.0.x copy of “line 10” and it will be changed into something else in the Version 2.0 copy of “line 10”
- This is a **conflict**
- There is no way, in general, for a source code control system (without a programmer’s intelligence) to decide how best to combine the two changes
- Example:

Line 10: happy -> line 10: happier (on 1.0.x branch)

Line 10: happy -> line 10: happiest (on 2.0 branch)

The merge will raise an error (conflict) and give us both ”line 10”s in the branch we’re merging to:

>> happier

--

<< happiest

- The programmer needs to decide what the one ”line 10” should now be in the branch we’re merging to (called resolving the conflict)

Repositories and working copies

- Source Code Control Systems use a repository where all the information about all the changes (commits) are stored
- Programmers can retrieve any single one of the versions from the repository – this is the programmers **working copy**
- Usually the working copy is retrieved by copying the most recent version of some branch
- Changes are made to the working copy
- When the programmer is satisfied that the change is complete, they **commit** their changes, creating a new most recent version of the branch
- With just one programmer, there is no need to worry that other changes have been committed to the branch in the meantime (no need to worry about those other changes being lost)

Collaboration with Source Code Control

- With collaboration between two or more programmers, now we need to consider that other changes might have been committed to the branch since we checked out (the most recent version on) that branch
- CVS and Subversion are source code control systems that use a central repository: let's talk about that first
- Consider this problem:

Mary checks out BranchA

Bill checks out BranchA

Bill adds FeatureX to his working copy of BranchA

Mary adds BugFixA to her working copy of BranchA

Bill commits his working copy to BranchA

Mary commits her working copy to BranchA

- After this, the most recent version on BranchA includes BugFixA but not FeatureX, and the second-most-recent version on BranchA includes FeatureX, but not BugFixA

Collaboration with Source Code Control (cont'd)

- Intuitively, the solution to Mary and Bill's problem is this (no matter what source code control system is being used)

1. Programmer obtains working copy from the appropriate Branch
2. Programmer makes their changes to their working copy
3. **Programmer merges into their working copy all of the changes that other programmers have made to the Branch since Step 1**
4. Programmer commits their working copy to the Branch

- If Mary had done Step 3, she would have committed a version that contained both BugFixA and FeatureX, which is normally what we want

Collaboration Steps

- Let's call Steps 1-4 on the previous slide the **Collaboration Steps**
- With CVS, at a Linux command line the Collaboration Steps become

```
% CVSROOT=:pserver:jrandom@cvs.foobar.com:/usr/local/cvs  # set repository
% export CVSROOT.                    # now the cvs command knows where repo is
% cvs checkout TheGoodBranch    # step one, working copy here now
%  <make all your changes>          # step two
% cvs update                        # step three, a merge, with potential conflicts
%   <fix conflicts if necessary>
% cvs commit                        # step four
```

Git Book

- Find the git book here: https://git-scm.com/book/en/v2

Collaboration Steps with Git

- On the previous slide, with CVS and a central repository:
  - all developers check out and check in to the same physical repository
  - The developers probably access the single repository remotely
  - Each developer has just their working copy of a single version – not a copy of the whole repository
  - Developers commit their working copy to the central repository to make their work visible to others
- Git is a distributed system:
  - All developers have a complete local copy of the repository (at least one)
  - Often, one of the copies of the repository is deemed special, for example, the one that is stored on a repository sharing facility (GitHub, BitBucket, etc)
  - A developer begins by cloning a copy of the repository (origin) to their local machine (often they'll be cloning the "special" repository but it could be any one)
  - The developer checks out a version from their local (cloned) repository and works on that, checking their work into their local copy of the repository
  - Developers publish their work by pushing their local repository content to the remote one (origin) possibly merging
  - Developers can see the other programmers' work by pulling from the remote repository (origin) to the local copy of the repository

Collaboration tutorial

- There is a good tutorial on Collaboration with Git at

https://www.atlassian.com/git/tutorials/syncing

We will continue at the above URL. To summarize:

git remote : for managing which repositories you're syncing with your own

git fetch : to copy content from origin into your own repository

git push : to copy content from your own repository into a remote

git pull : to copy and merge content from a remote into your own repository

Pull request

- A **pull request** is not a git operation
- The pull request feature is provided by an external facility, and its purpose is to facilitate communication between developers while syncing their repositories
- A pull request is actually a request that some else will pull content **from** you
- If you are communicating with a developer by other means, like Slack, or face-to-face, then you can coordinate your syncing through that channel (syncing means using “git push” and/or “git pull” and/or “git fetch”) as described in the Collaboration tutorial linked on the previous slide.