COLLABORATIVE DEVELOPMENT

Christine Julien

EE461L

Based on materials from Miryung Kim and Adnan Aziz

Motivation for Version Control

- Software is constantly evolving
  - We often need a history of these changes
- Multiple developers on any real project
  - Must restrict access, handle merges, attribute changes
- Added bonus: central repository

Outline

- What is version control and why do we use it?
  - Best understood in terms of scenarios:
    - One developer/multiple developers
    - One branch/multiple branches
- Basic concepts
  - VC operations
    - import, co, ci, up, diff, log, merge, info, blame, …
  - Branches
  - Merging
- We will use SVN as a VCS tool and discuss alternatives later…

Why do we need version control?

- Start with one developer, one file
- The file system stores the current snapshot of the file

- Without version control, we cannot recreate the project history
  - We cannot revert to an older version
    - To discard recent changes (e.g., started work an a new implementation of an API and gave up)
    - To investigate when a bug was introduced
  - We cannot compare a current version to an older one
  - We cannot figure out when a particular section of code was added/ updated and why

Version Control

- Version control tracks multiple versions
  - Different releases
  - Different versions of the same release
  - Each time the project is edited
- Version control allows
  - Old versions to be easily recovered
  - Optimistic locking – allows multiple versions to exist simultaneously (more later)
- Version control is implemented using low level primitives like **diff**, **patch**, **locking**
  - Note: remember “longest common subsequence” and “edit distance”?

Version Control: Basic Operations

- Version control systems store all versions of the project

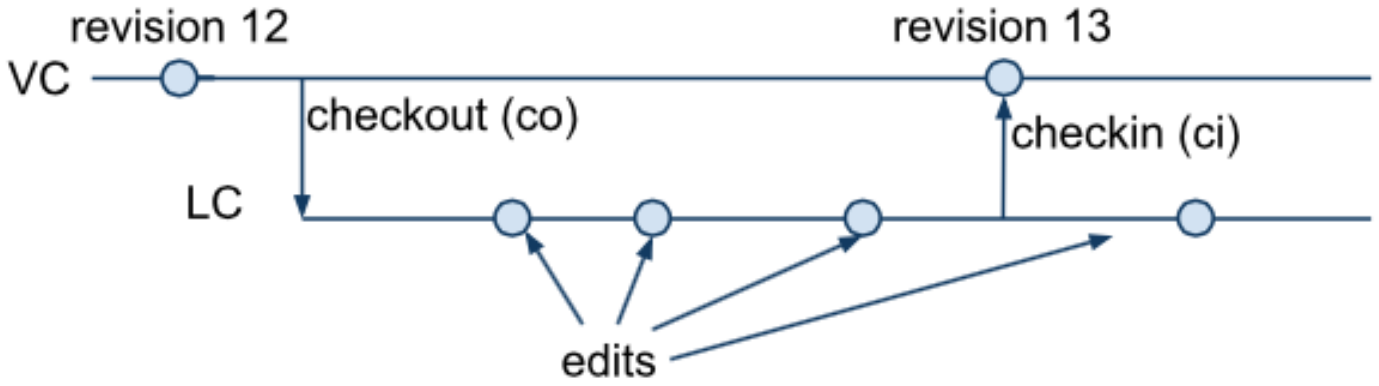
- Checkout (e.g., `svn co`)
  - Copies the current version of the project from version control (also called the **repository**) into a **local copy** (also called the **working copy**)

Do This

- svn co https://subversion.assembla.com/svn/ut461lsvnexample/

Version Control: Basic Operations

- Editing is always done on the local copy



- A **checkin** stores a snapshot of the local copy
  - Can add comment, author, data
  - Can configure to have mail sent to watchers or to automatically run tests
  - Creates a new **revision**
  - Also know as a **commit** or **putback**

Version Control: History

- You can see the history of revisions by examining the log (`svn log`)

From **svnlog,** Featureful e-mail commit notification from svn
1.5 (2005-05-26)
Pass the correct arguments into build_diff.
Lines changed: +7 -7
1.4 (2005-05-26)
Fix the debbugs support. Remove trailing slashes from directory names in the Subject header (but not elsewhere). Limit the size of the diff in the mail message to 200KB.
Lines changed: +40 -21
1.3 (2005-05-08)
Fix unterminated B<> in the documentation.
Lines changed: +2 -2
1.2 (2005-05-08)
Changed the output format to look a bit more like mailer's output, but with the change message before the list of affected files. Skip diffstat output if we don't have anything interesting to feed to it. Fix loop when calculating prefixes when the only affected files are directories. Add additional metainformation about property changes and tree or file copies to the log message. Make the debugging output a bit more useful.
Lines changed: +103 -42
1.1 (2005-04-17)
Initial working version

Version Control: Inspecting Changes

- diff(VC12, VC13)
  - List of edits needed to change revision 12 to be the same as revision 13
  - These are the changes made in revision 13
  - `svn diff -r 12:13 [TARGET]` or `svn diff -c 13 [TARGET]`
- diff(VC13, LC)
  - List of edits needed to change revision 13 to be the same as the local copy
    - These are your local edits
  - `svn diff -r 13 [TARGET]` (or just `svn diff [TARGET]`)
- `svn help diff`
  - Gives arguments, options

diff

- **diff** program: given text files A and B, determine sequence of edits that changes A into B
  - `1,2c1,2` : change lines 1 and 2
  - < remove
  - > add

- General form:
  - `n1,m1cn2,m2`
    - If only one line `n1cn2,m2` or `n1,m1cn2` or `n1cn2`
    - `5a6,7` append lines 6 and 7 after line 5
    - `6,7d5` delete lines 6 and 7 (the files will match from line 5 onwards)

- Can also diff directories (matches files by name)

diff example

```
The change impact was minimized
by localizing the most
important variables,
saving approximately
1,500 pounds.
```

```
The change was minimised
by localising the most
important variables,
saving approximately
1,500 pounds.
(1500 pounds are approximately $2,100)
```

```
sh% diff A.en_us A.en_uk
```

```
1,2c1,2
< The change impact was minimized
< by localizing the most
---
> The change was minimised
> by localising the most
5a6
> (1500 pounds are approximately $2,100)
```

patch example

- patch takes a diff output and applies its commands on a file: inserting, deleting, and changing lines as mandated by the commands

```
sh-3.2$ diff A.en_us A.en_uk > A.patch
sh-3.2$ cp A.en_us tmp
sh-3.2$ patch tmp < A.patch
patching file tmp
sh-3.2$ cat tmp
The change was minimised
by localising the most
important variables,
saving approximately
1,500 pounds.
(1500 pounds are approximately $2,100)
```

- patch does not always succeed
  - It leaves portions that cannot be applied in `tmp.reg`; you have to take care of these manually

Contextual diff

```
sh$ diff -c A.en_us A.en_uk
*** A.en_us    2014-01-24 20:15:25.000000000 -0600
--- A.en_uk    2014-01-24 20:15:14.000000000 -0600
***************
*** 1,5 ****
! The change impact was minimized
! by localizing the  most
  important variables,
  saving approximately
  1,500 pounds.
--- 1,6 ----
! The change was minimised
! by localising the most
  important variables,
  saving approximately
  1,500 pounds.
+ (1500 pounds are approximately $2,100)
```

Start of a “hunk”

The line should be replaced

The line should be added

Contextual patch

- Repeat the patch example using contextual diff…

Getting differences for multiple files

- diff also works on multiple files at the same time
- Go to your Lecture5 folder under version control
- Inside, there should be two folders:
  originaldirectory/ and updated directory/
- Try these commands and observe the differences
  - `diff originaldirectory/ updateddirectory/`
  - `diff -c originaldirectory/ updateddirectory/`

Patching multiple files

- Generate the patch the same as before (make a contextual patch)
  - `diff -c originaldirectory updateddirectory/ > patchfile.patch`
- To avoid stomping on your original directory, make a copy of `originaldirectory/` and `patchfile.patch` someplace else
- Now patch
  - `patch –i patchfile.patch`
- Oops.
  - Read the error and fix it.
  - Hint: use `man patch` if you need help

diff and Version Control

```
[adnan@quad0] 1084 svn diff svn-example.c -r 3
Index: svn-example.c
===================================================================
--- svn-example.c    (revision 3)
+++ svn-example.c    (working copy)
@@ -1,7 +1,10 @@
 #include <stdio.h>
+#include <math.h>

 int main( int argc, char **argv ) {
    int x = 42;
-   printf("The number %d is a magical number\n", x );
+   double y = 3.14;
+   printf("The number %d is a very magical number\n", x );
+   printf("The number %f is even more magical\n", y );
    return 0;
 }
```

This is `svn diff`.

A diff is made out of “changed hunks” (aka **chunks**)

This diff is between the local copy and revision 3

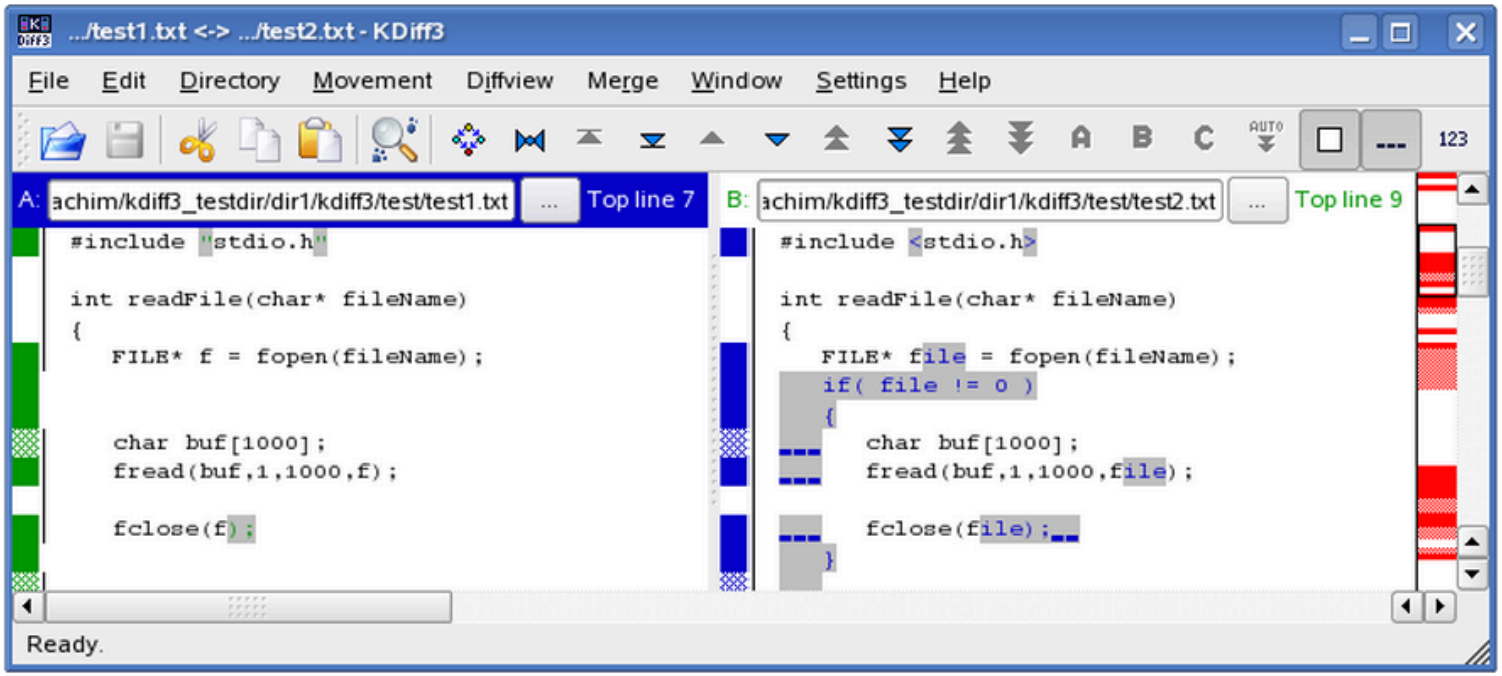
Lines starting with - are in revision 3; lines starting with + are only in the local copy

Do this

- Go to your Lecture5 directory under version control
- `svn diff -r 2:3 A.en.uk`

Some notes on diff

- The order of arguments to diff matters
  - diff(VC13, VC12) gives the edits needed to change revision 13 into revision 12
    - These are the steps needed to revert the changes in VC13
- There are lots of nice graphical tools for diff

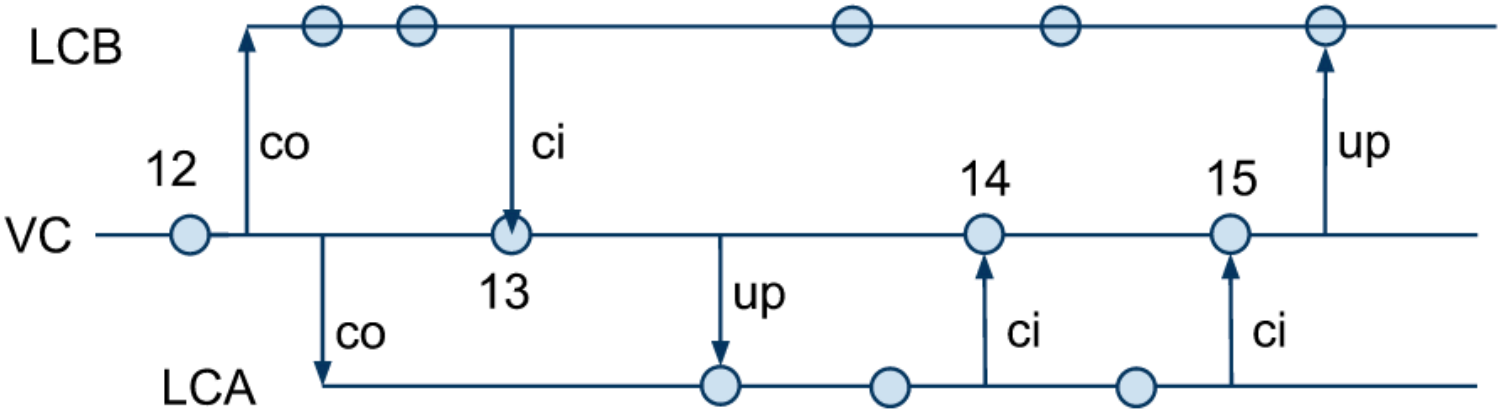


How does this apply to version control?

- Well, it will be much more interesting in a bit…
- But if someone sends you a “patch” suggesting an edit to your local copy
  - You can apply the patch (after reviewing it, of course)
  - Then commit your changes
- Things are about to get a lot more interesting, though, as we add in more developers…

Version Control: Concurrent Development

- Assume that we have two developers, A and B, each with his own local copies (LCA and LCB)



- `svn update` brings new revisions from version control into the local copy
  - This essentially performs a **merge**
    - i.e., it performs a patch using the diff… all automatically

Version Control: Under the Hood

- For each file, the local copy records
  - The version control revision number it is based on (the last update or checkin)
  - The time of the last update or checkin
- The local copy classifies files into categories:
  - **unchanged, current**
    - There are no changes in either the local copy or version control
    - Both `svn up` and `svn ci` would do nothing
  - **locally changed, current**
    - There are changes in the local copy only
    - `svn up` would do nothing; `svn ci` would commit the edits that have been made
  - **unchanged, outdated**
    - No changes in the local copy, but the file in version control has been updated
  - **locally changed, outdated**
    - Both the local copy and the copy under version control have changed

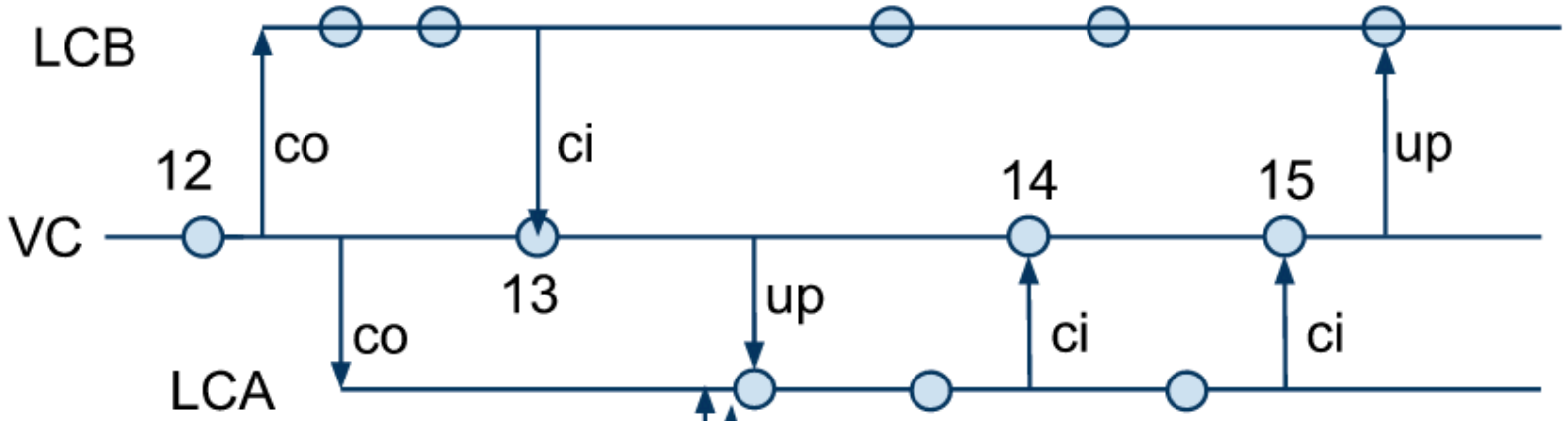
SVN and Conflicts

- If developers A and B simultaneously make edits that conflict
  - Whoever is second to check in will have to resolve the conflict

```
$ svn update
Conflict discovered in 'test-file.txt'
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

  - Selecting `df` will show you the diff
  - Selecting `e` will launch an editor for you to resolve the conflict by hand

A More Complex Scenario




```
diff(VC12, VC13)
--- foo.c (revision 12)
+++ foo.c (revision 13)
@@ -12,14 +17,19 @@
  int i;
  -i = 2;
  +i = 3;
  print(i);
```

LC status

```
20: int i;
21: i = 2;
22: print(i+1);
```

```
diff(VC12, LC)
--- foo.c (revision 12)
+++ foo.c (working copy)
@@ -12,15 +20,22 @@
  int i;
  i = 2;
  -print(i);
  +print(i+1);
```

A More Complex Scenario (cont.)

- A can diff the local copy with the repository:

```
$ svn diff example.c
--- example.c (revision 13)
+++ example.c (working copy)
@@ -1,3 +1,3 @@
 int i;
-i=3;
-print(i);
+i=2;
+print(i+1);
```

- There will be no merge conflicts on update

```
$ svn up
G example.c
Updated to revision 13.
```

G: merGed

More on Merge Conflicts

- Merge will complain when it cannot match the original hunk in the file to be changed
  - This makes merge safe: it will not silently discard your local changes
- Merge creates 4 files for each conflict in the local copy
  - foo.c.r12 – the version control revision 12 of foo.c (what we originally checked out)
  - foo.c.r13 – the version control revision 13 of foo.c (what B checked in)
  - foo.c.mine – the local copy of foo.c before the merge
  - foo.c – the partially merged foo.c

More on Merge Conflicts (cont.)

- You can keep your copy

```
cp -f foo.c.mine foo.c
```

- You can keep the copy checked into version control

```
cp -f foo.c.r13 foo.c
```

- You can edit the conflicts in the partially merged file
  - These conflicts are identified by markers in the file
  - Note: not all differences are conflicts!

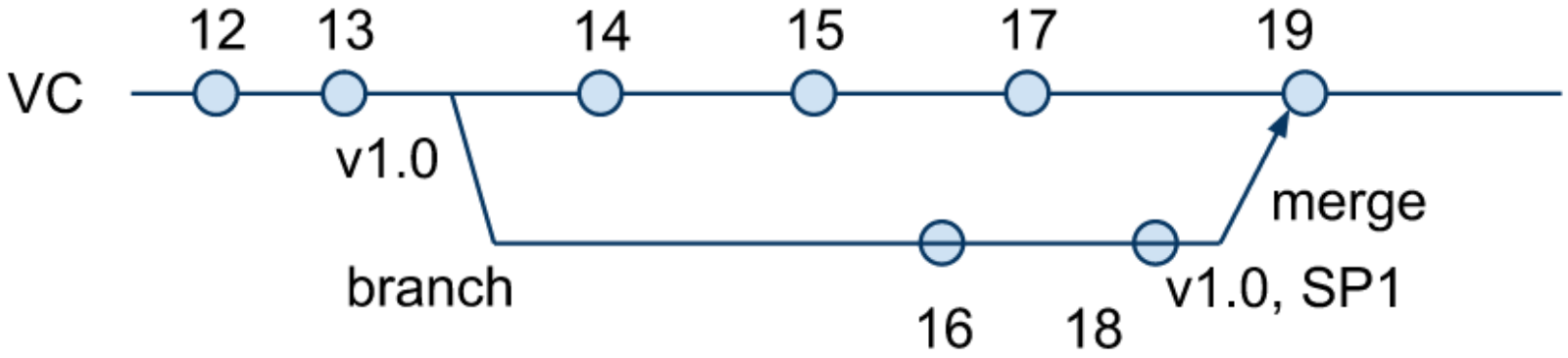
- Best practice is to use a merge tool (e.g., kdiff3) to aid conflict resolution

Syntactic Conflicts

- Conflict detection is based on the “nearness” of changes (based on lines in the file)
  - Changes to the same line will conflict
  - Changes to different lines will likely not conflict
- A lack of conflicts does not mean that changes made by A and B will work together!
  - Lack of merge conflicts is not a certificate of correctness
    - Best case scenario: compile fails
    - Second best scenario: tests fail (because you have a test suite, right?)
    - Worst case: semantic conflict makes it to the field

Version Control and Branching

- Consider the following scenario:



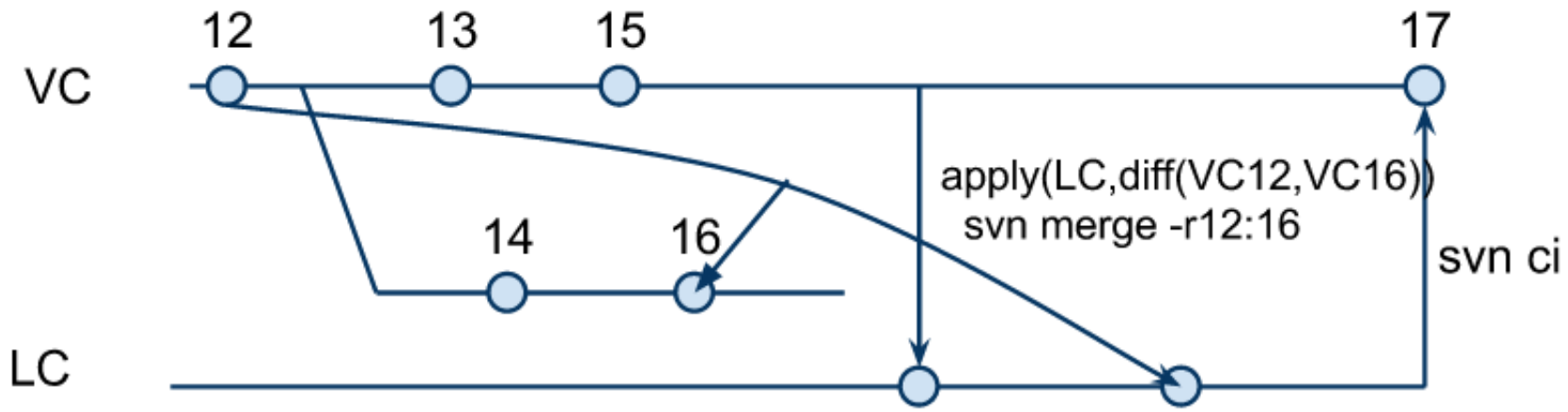
- Revision 13 is released as v1.0
  - A bug is reported in v1.0 and we need to make a service pack
  - We cannot do so without a branch
    - A branch is a parallel version control with a shared history

Other Uses for Branches

- Temporary/private versions
  - Implement new and experimental features
  - Isolate changes
  - Eventually merge changes back
- Snapshot of the code for testing
  - Development continues on the main trunk
  - Testing and hot-fixes on the main testing branch
  - Eventually all hot-fixes merged back to trunk
- Separate branch for custom release
  - Surprisingly common

Merging Branches

- In SVN merging of branches is always done on a local copy
  - Use diff to collect the changes, then apply the changes to the local copy
  - Commit the changes back to the main branch



AN ALTERNATIVE…

Distributed Version Control

slides adapted from those created by Ruth Anderson

images from http://git-scm.com/book/en/

Git Resources

- At the command line: (where verb = config, add, commit, etc.)

  $ git help <verb>

  $ git <verb> --help

  $ man git-<verb>

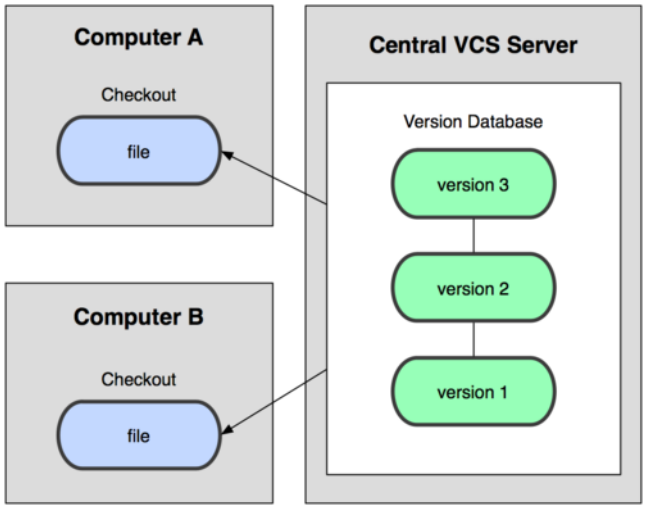
- Free on-line book: http://git-scm.com/book
- Git tutorial: http://schacon.github.com/git/gittutorial.html
- Reference page for Git: http://gitref.org/index.html
- Git website: http://git-scm.com/

Git History

- Came out of Linux development community
- Linus Torvalds, 2005
- Initial goals:
  - Speed
  - Support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like Linux efficiently

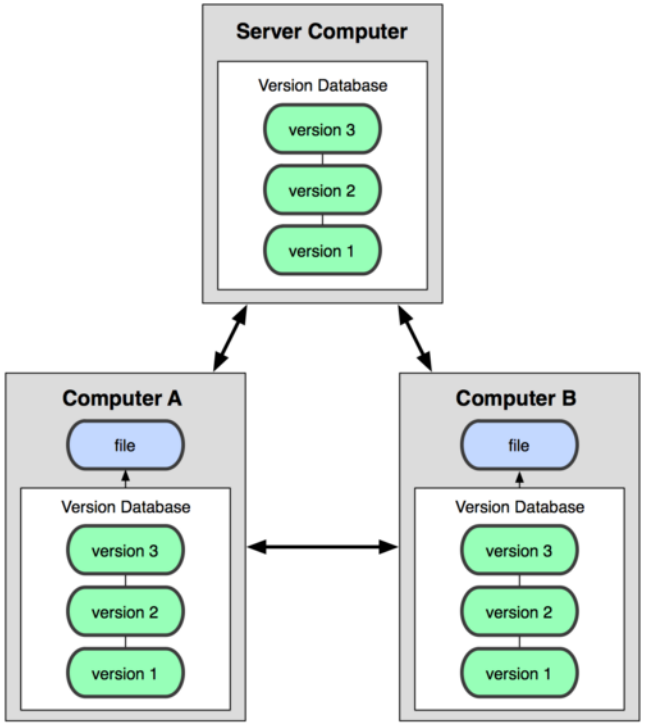
Git uses a distributed model

Centralized Model



(CVS, Subversion, Perforce)

Distributed Model

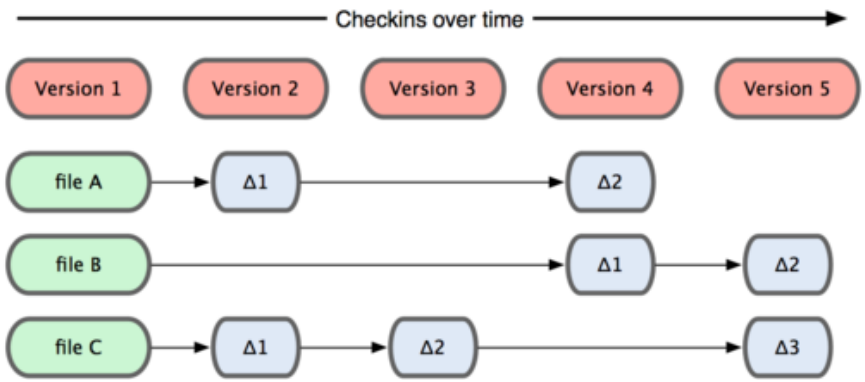


(Git, Mercurial)

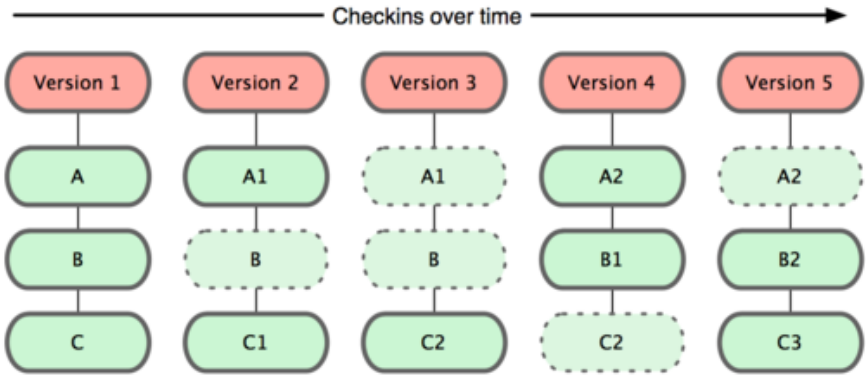
Result: Many operations are local

Git takes snapshots

Subversion



Git



Git uses checksums

- In Subversion each modification to the **central** repo increments the version # of the overall repo.
- How would this numbering scheme work **when each user has their own copy of the repo**, and commits changes to their local copy of the repo before pushing to the central server?

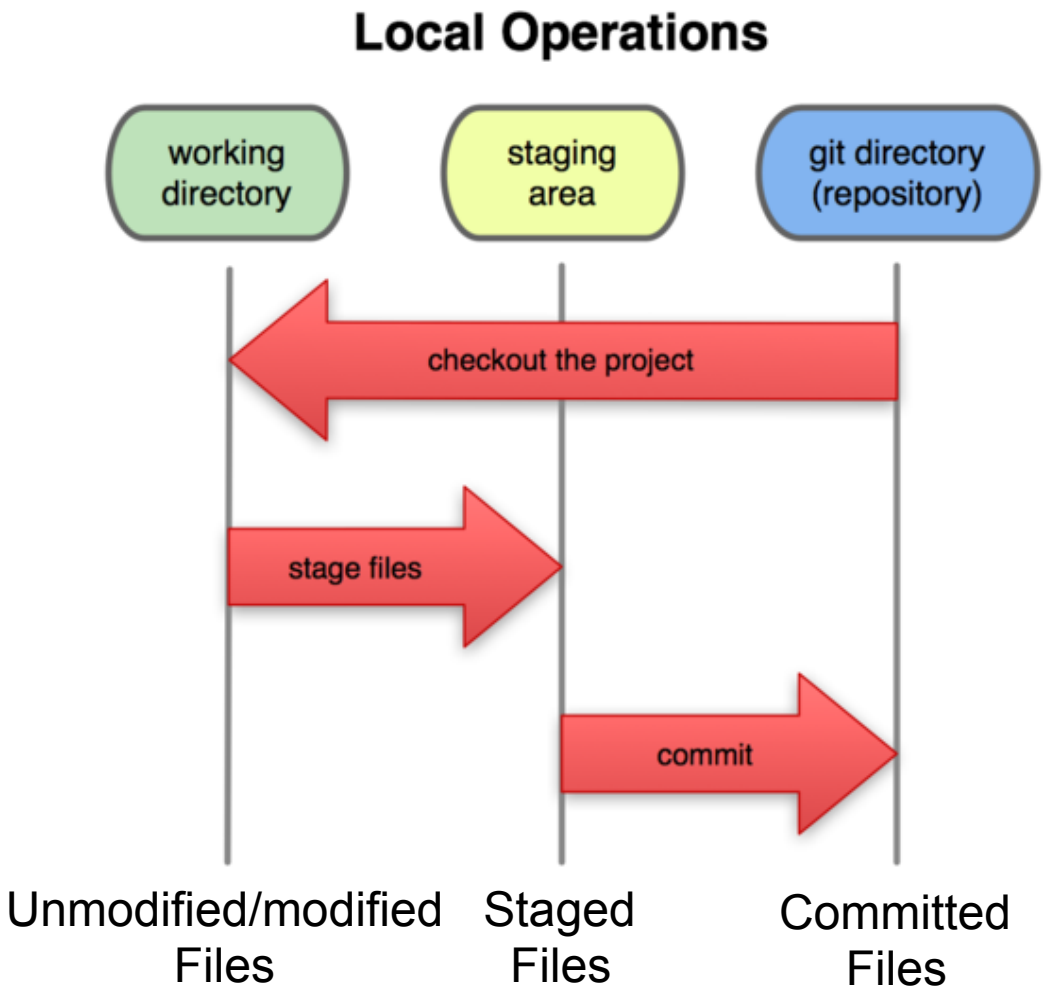
- Instead, Git generates a unique SHA-1 hash – 40 character string of hex digits, for every commit. Refer to commits by this ID rather than a version number. Often we only see the first 7 characters:

  1677b2d Edited first line of readme

  258efa7 Added line to readme

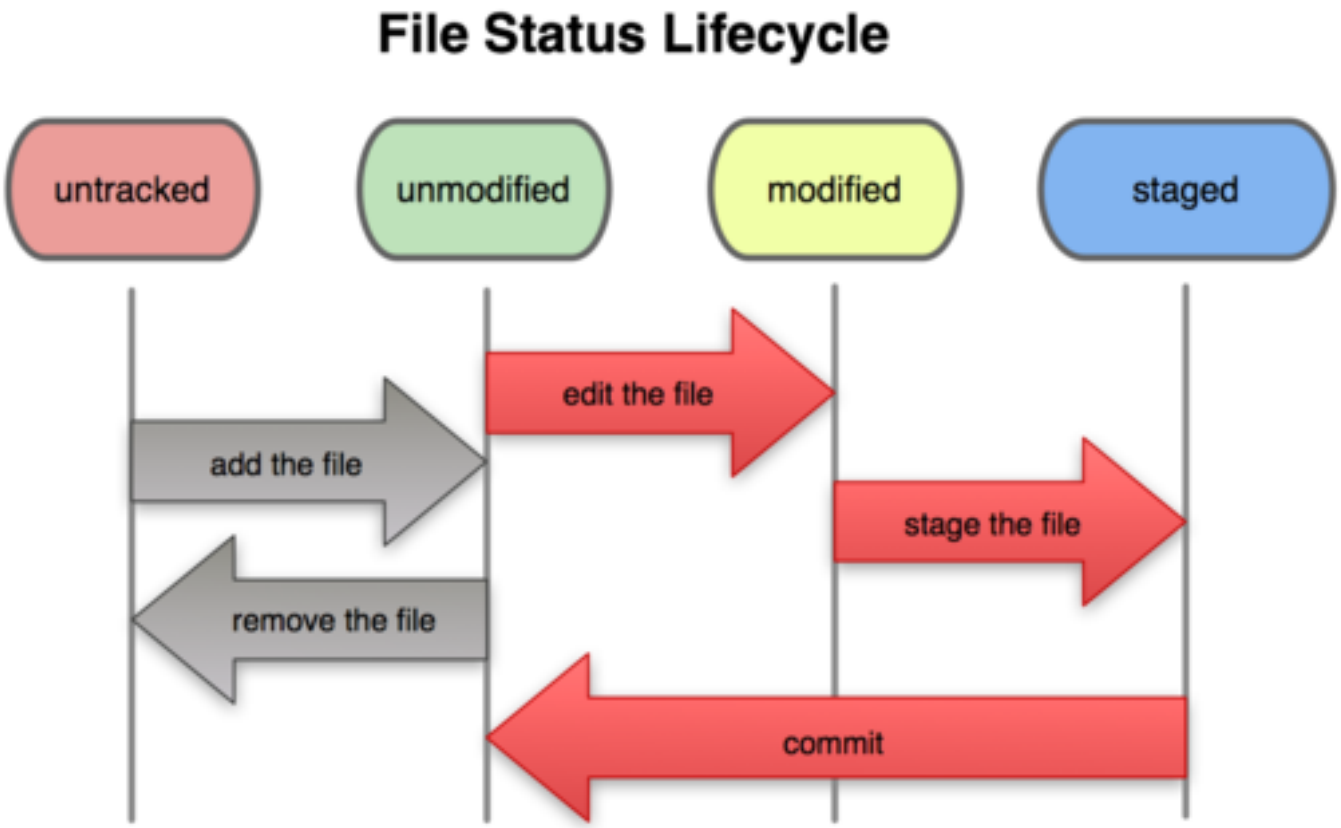
  0e52da7 Initial commit

A Local Git project has three areas



Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

Git file lifecycle



Basic Workflow

Basic Git workflow:

1. **Modify** files in your working directory.
2. **Stage** files, adding snapshots of them to your staging area.
3. Do a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

- Notes:
  - If a particular version of a file is in the **git directory**, it's considered **committed**.
  - If it's modified but has been added to the **staging area**, it is **staged**.
  - If it was **changed** since it was checked out but has not been staged, it is **modified**.

Aside: So what is github?

- GitHub.com is a site for online storage of Git repositories.
- Many open source projects use it, such as the Linux kernel.
- You can get free space for open source projects or you can pay for private projects.

Question: Do I have to use github to use Git?

Answer: No!

- you can use Git completely locally for your own purposes, or
- you or someone else could set up a server to share files, or
- you could share a repo with users on the same file system

Get ready to use Git!

1. Set the name and email for Git to use when you commit:

```
$ git config --global user.name "Bugs Bunny"
$ git config --global user.email bugs@gmail.com
```

- You can call **`git config –list`** to verify these are set.
- These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the **`--global`** flag.
- You can also set the editor that is used for writing commit messages:
  $ git config --global core.editor emacs (it is vim by default)

Create a local copy of a repo

2. Two common scenarios: (only do one of these)
   a) To **clone an already existing repo** to your current directory:

```
$ git clone <url> [local dir name]
```

This will create a directory named *local dir name*, containing a working copy of the files from the repo, and a **`.git`** directory (used to hold the staging area and your actual repo)

   b) To **create a Git repo** in your current directory:

```
$ git init
```

This will create a **`.git`** directory in your current directory.

Then you can commit files in that directory into the repo:

```
$ git add file1.java
$ git commit –m “initial project version”
```

Git commands

command	description
`git clone` ***`url [dir]`***	copy a git repository so you can add to it
`git add` ***`file(s)`***	adds file(s) contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help` ***`[command]`***	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

Committing files

- The first time we ask a file to be tracked, *and* **every time before we commit a file** we must add it to the staging area:

```
$ git add README.txt hello.java
```

This takes a snapshot of these files at this point in time and adds it to the staging area.

- To move staged changes into the repo we commit:

```
$ git commit –m "Fixing bug #22"
```

Note: To unstage a change on a file before you have committed it:

```
$ git reset HEAD  --  filename
```

Note: To unmodify a modified file:

```
$ git checkout  --  filename
```

Note: These commands are just acting on **your local version of repo**.

Status and Diff

- To view the **status** of your files in the working directory and staging area:

```
$ git status
```
or
```
$ git status –s
```
(`-s` shows a short one line version similar to svn)

- To see what is modified but unstaged:

```
$ git diff
```

- To see staged changes:

```
$ git diff --cached
```

After editing a file…

```
[rea@attu1 superstar]$ emacs rea.txt
[rea@attu1 superstar]$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rea.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
[rea@attu1 superstar]$ git status -s
 M rea.txt                                   ← Note: M is in second column = "working  tree"
[rea@attu1 superstar]$ git diff              ← Shows modifications that have not been staged.
diff --git a/rea.txt b/rea.txt
index 66b293d..90b65fd 100644
--- a/rea.txt
+++ b/rea.txt
@@ -1,2 +1,4 @@
 Here is rea's file.
+
+One new line added.
[rea@attu1 superstar]$ git diff --cached     ← Shows nothing, no modifications have been staged
yet.
[rea@attu1 superstar]$
```

After adding file to staging area…

```
[rea@attu1 superstar]$ git add rea.txt
[rea@attu1 superstar]$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   rea.txt
#
[rea@attu1 superstar]$ git status -s
M  rea.txt                                   ← Note: M is in first column = "staging area"
[rea@attu1 superstar]$ git diff              ← Note: Shows nothing, no modifications that have not
been staged.
[rea@attu1 superstar]$ git diff --cached     ← Note: Shows staged modifications.
diff --git a/rea.txt b/rea.txt
index 66b293d..90b65fd 100644
--- a/rea.txt
+++ b/rea.txt
@@ -1,2 +1,4 @@
 Here is rea's file.
+
+One new line added.
```

Viewing logs

To see a log of all changes in your local repo:

- `$ git log` or
- `$ git log --oneline` (to show a shorter version)

```
1677b2d Edited first line of readme
258efa7 Added line to readme
0e52da7 Initial commit
```

- `git log -5` (to show only the 5 most recent updates, etc.)

Note: changes will be listed by commitID #, (SHA-1 hash)

Note: changes made to the remote repo before the last time you cloned/pulled from it will also be included here

Pulling and Pushing

Good practice:

1. **Add** and **Commit** your changes to your local repo
2. **Pull** from remote repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
3. **Push** your changes to the remote repo

To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:

```
$ git pull origin master
```

To push your changes from your local repo to the remote repo:

```
$ git push origin master
```

Notes: `origin` = an alias for the URL you cloned from

`master` = the remote branch you are pulling from/pushing to,
(the local branch you are pulling to/pushing from is your current branch)

Note: On attu you will get a Gtk-warning, you can ignore this.

Branching

To create a branch called experimental:

- `$ git branch experimental`

To list all branches: (* shows which one you are currently on)

- `$ git branch`

To switch to the experimental branch:

- `$ git checkout experimental`

Later on, changes between the two branches differ, to merge changes from experimental into the master:

- `$ git checkout master`
- `$ git merge experimental`

Note: `git log --graph` can be useful for showing branches.

Note: These branches are in *your local repo*!

SVN vs. Git

- SVN:
  - central repository approach – the main repository is the only "true" source, only the main repository has the complete file history
  - Users check out local copies of the current version
- Git:
  - Distributed repository approach – every checkout of the repository is a full fledged repository, complete with history
  - Greater redundancy and speed
  - Branching and merging repositories is more heavily used as a result

QUESTIONS?