Introduction to Software Engineering (ISAD1000)

Lecture 4: Version Control

Updated: 15th February, 2022

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Version Control

- Almost every software project uses *version control*.
  - The ones that don't...*should!*
- It's a way of tracking:
  - A complete history of all work done so far.
  - Multiple simultaneous versions of the product.
- To help us, we need a *version control system* (VCS).
  - The most well-known one is probably "Git".
  - Others include "Mercurial", "Subversion", "Perforce", and many more.
  - We cannot realistically do version control without a VCS.
- But it's not all automated. The VCS is a tool, and we must learn how to use its powers for good.

Git is not Github

- The VCS (Git, Mercurial, etc.) is just a piece of software.
  - You download and install it on your computer.
- You may know of websites like GitHub.com, Bitbucket.org, or others.
  - These are *services* that happen to use Git, Mercurial, etc.
  - They provide a central place to store ("host") your project code.
  - They also provide features to help your team track its progress, discuss issues, and conduct code reviews.
- The VCS and the hosting service are separate things.
- You can use the VCS *by itself* (just on your own computer), or in conjunction with a hosting service.

Undoing Mistakes

- At some point, you may experience the following:
  1. It works!
  2. Now I'll just add the next feature...
  3. Damn. Now it's broken.
  4. Argh! Now I can't figure out how to get back to the original version.
  5. Start again from scratch.
- The easy solution is to *save a copy* before making drastic changes.
- But this leads to another problem:
  - After a while, I have directories called "new", "original", "working", "working.3", "old", "new/old", "old/old/new", etc.
  - Each one is a particular version, and they build up over time.
  - But I *can't remember* what is what.

But text editors can undo changes. . .

- Any remotely reasonable text editor has an "undo" feature.
- But this isn't good enough! A few reasons:

1. You often want *both* the old and new versions.
   - Don't throw away the new version just because it's broken.
   - You may yet figure out how to fix it!
2. The editor's "undo buffer" is only temporary (in general).
   - Editors forget all the undo information when you close them.
3. There's still too much for *you* to remember.
   - Since the last working version, you may have inserted/deleted hundreds of words across several files.
   - Will you remember how much you need to undo in each file?
   - Will you even remember *which files* you've modified?

How does a VCS help?

- For every update ("commit") made to a software project, the VCS records:
  - What was changed.
  - The date and time.
  - The person responsible (since you're probably working in a team).
  - A description entered by that person.
- You can tell the VCS to:
  - Show the list of commits in order.
  - Retrieve the code as it existed at a specific time in the past.
  - Show the exact differences between the code at two different times.
- All you have to do is: tell the VCS whenever you make a change (a "commit"), and enter a description.
- Now, you can safely make any changes you like, and always have the ability to undo them later on.

Repositories

- A *repository* ("repo") stores the complete project (all its versions).
- Older VCSs had a single "centralised" repository.
  - Stored on a central server.
  - Each team member can "check out" a few files, like borrowing books from a library.
  - Later they "check in" the updates they've made.
  - Nobody else can modify the same files at the same time.
- Newer VCSs (e.g. Git and Mercurial) are "distributed".
  - Each team member has their own "local" repository.
  - Typically there's *also* a central repository.
    - Often hosted on GitHub.com, Bitbucket.org, etc.
  - The different repositories are periodically kept in sync with each other.
    - People *can* update the same files at the same time.
    - Intelligent algorithms help "merge" their updates together.

Local Repository vs. Working Directory

- Your "working directory" stores the version of the code that you're currently working on.
  - *Probably* the latest version (but see later discussion on branches).
  - This is just straightforward, ordinary directory.
- Your local repository stores all versions of the code (except any *uncommitted* changes in the working directory).
  - Typically the repo is stored in a sub-directory in the working directory ("`.git/`" for Git, "`.hg/`" for Mercurial, etc.).
  - VCS-specific format – not directly human-understandable.
- The VCS knows how to (among other things):
  - Save the working directory (or parts of it) into the repository.
  - Load a particular version from the repository into the working directory.

Git

- Created by Linus Torvalds, to help manage the Linux OS kernel (which he also initially created):
  - The Linux kernel is a vast project: 22 million lines of code, 4,600 new lines added *per day*, over 13,500 developers, and running since 1991 [1].
  - This is what Git was created to handle.
- We'll use Git via the command-line.
  - There are lots of GUI and web tools available too.
  - But the command-line is better for learning.
- Git has a whole suite of little commands:

```
[user@pc]$ git add ...
```

```
[user@pc]$ git commit ...
```

```
[user@pc]$ git reset ...
```

[1] https://www.linux.com/infographic/25-years-linux-kernel-development

Git – Setting Up a Local Repository

- Before you use Git for anything, set up your identity:

```
[user@pc]$ git config -global user.name "Your Name"
```

```
[user@pc]$ git config -global user.email "me@xyz.com"
```

  - This will help identify your work in the repository (since there could be other people involved too).
  - "`-global`" applies this to *all* projects (for this login account).
- To actually create a repository for a project:

```
[user@pc]$ mkdir myproject
```

```
[user@pc]$ cd myproject
```

```
[user@pc]$ git init
```

This will create and populate the `.git` directory (hidden, due to the starting dot).

Basic Git Usage – Staging and Committing

1. Create/modify some .java files in your project.
2. Tell Git to *stage* the updates (to prepare for a commit):

```
[user@pc]$ git add MyCode.java MyOtherCode.java
```

```
[user@pc]$ git add YetMoreCode.java
```

3. Tell Git to *commit* the staged files:

```
[user@pc]$ git commit -m "Fixed input validation bug."
```

- Always provide a meaningful description of what you did. To list files that are (1) staged, (2) new/modified since the last commit but not staged, or (3) unchanged:

```
[user@pc]$ git status
```

To see all un-staged code changes in detail:

```
[user@pc]$ git diff
```

More Notes on Staging and Committing

- A commit should represent "one thing" – one bug fix, one small feature, etc.
  - If you make two distinct changes, make two separate commits.
  - Makes it easier for others (and yourself in the future) to see what you did.
- Files can be un-staged, prior to a commit:

```
[user@pc]$ git reset MyCode.java
```

```
[user@pc]$ git reset
```

- If you mess up a commit, fix it by making *another* commit.
  - The thing about the ultimate "undo" tool...is that it has to remember everything you do, including the stupid things![2]

[2]There are ways of changing already-made commits, but this is risky, and can defeat the purpose of version control. In ISE, we'll assume commits are irrevocable.

Logs and Diffs – Viewing Past Work

- To get a summary of all commits:

```
[user@pc]$ git log --graph
```

  - ("`--graph`" is optional, but useful when we get to branching.)
- To see the entire history of a particular file:

```
[user@pc]$ git log -p MyCode.java
```

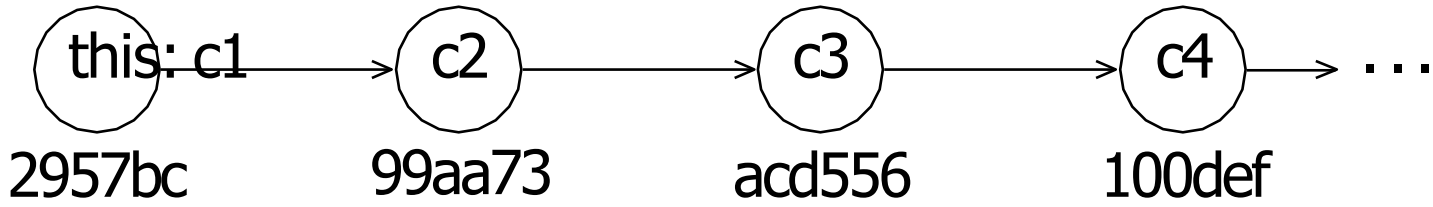
  - "**+**" marks a line that was added.
  - "**-**" marks a line that was deleted.
  - "**@@**" identifies the location of changed lines (though you can usually tell anyway).
- To see the differences between two commits (across all files):

```
[user@pc]$ git diff 46b9bc 5f14b6
```

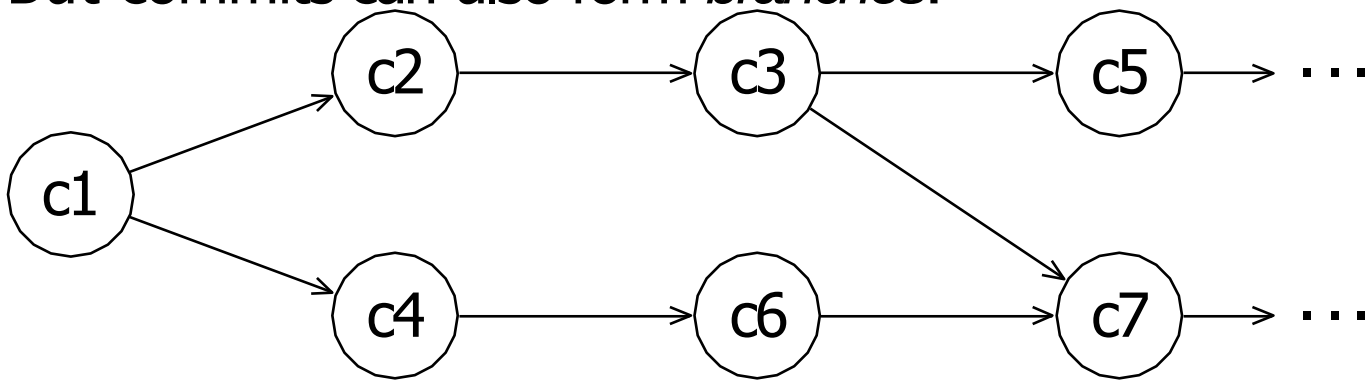
  - Each commit is identified by a "hash" code (shown by `git log`).
  - 40-chars long, but you can abbreviate them.

Branches

- You can think of a sequence of commits like



  - 4 commits in a row, each based on the previous one.
- But commits can also form *branches*:



  - We have two branches, originating from c1.
  - Each branch has several commits, until they are merged at c7.

Branching – Why?

- Branching allows *multiple simultaneous versions*.
- In a simple case, you have:
  - A "master" branch for the main "it-works!" version.
  - One or more "feature" branches for whatever crazy experimental stuff you're adding/fixing.
- In a team, different developers work on different feature branches.
- Branching is vital because you need easy access to these different versions.
  - People will ask you for the latest working version, so they can actually use it.
  - But you can still make experimental changes without risking (a) what you're already done, and (b) what everyone else is working on.

Branching in Git (1)

- By default, you have one branch called "master".
- To make a new branch, based on the last commit in the current branch:

```
[user@pc]$ git branch mynewbranch
```

- To switch branches:

```
[user@pc]$ git checkout mynewbranch
```

  - You must commit any uncommitted changes first.
  - This will *delete and replace* your code with the latest version in "mynewbranch".
  - "`git checkout -b mynewbranch`" will both create and checkout a new branch.
- Now, your next commit will be in the new branch.
  - Switch back to "master", and you'll see the old version.

Branching in Git (2)

- To list the existing branches, and see which one is current:

```
[user@pc]$ git branch
```

  - "`git status`" will also show the current branch name.
- You can create a new branch based on *any* commit:

```
[user@pc]$ git branch mynewbranch ff823e
```

  - If you made a big mistake in an existing commit, you can create a branch based on the *previous* commit, and "try again".
- You can rename branches:

```
[user@pc]$ git branch -m mynewbranch featurexyz
```

(Renames "mynewbranch" to "featurexyz".)

- You can also delete branches, but that's associated with *merging*.

Merging

- Branching would be useless without later being able *merge* branches together.
  - Feature branches are where most work is done.
  - But this work must end up in the "master" branch somehow.
- "`git merge`" merges another branch into the current one:

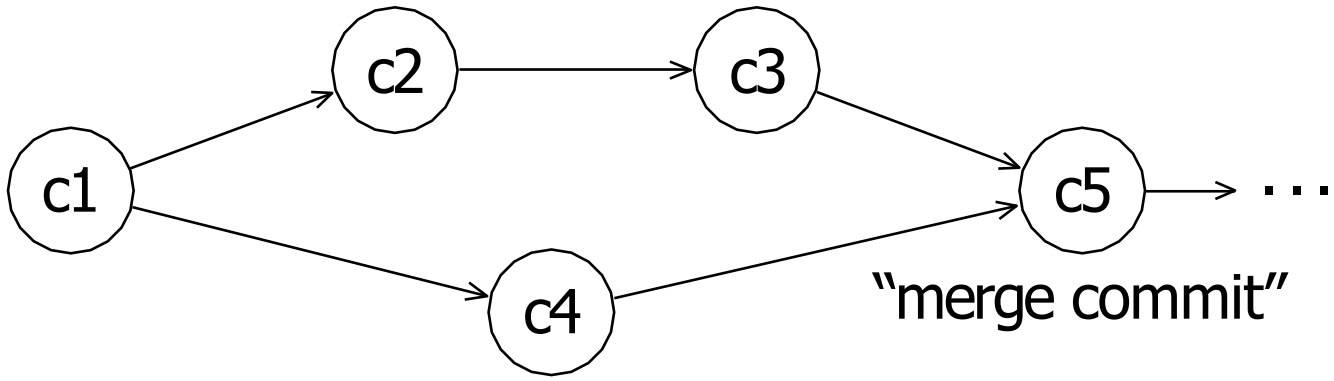
```
[user@pc]$ git checkout master
```

```
[user@pc]$ git merge mynewbranch
```

- Here we merge "mynewbranch" into "master" (in Git):
  - We're finished with mynewbranch.
  - All our work is now in "master" (until we make another new branch).
- Merging creates a special "merge commit": a mixture of the last commits in "master" and "mynewbranch".

Merging – WTF?

- How does Git *know* how to merge things?
- Merging seems like can't possibly be automated. Consider this:



  - These branches *both* have commits, which could be anything!
  - We're asking Git to combine two different pieces of code and make a workable result!
  - This ought to be a *hard* problem!

Merging – How it Works

- Git can't *always* merge things automatically, but often it can.
- First, it finds the "common ancestor": the point where the branches first separated.
- Second, Git executes the "diff3" algorithm. This tells it:
  - What has changed in each branch since the common ancestor.
  - What has stayed the same in each branch.
- Often the two branches change *different sections of code*.
  - Git will automatically commit these changes.
- If the two branches both change the *same section of code*, this is a "merge conflict".
  - Merge conflicts must be resolved manually.

Merge Conflicts and Manual Merging

- A "`git merge`" may report something like this:

```
CONFLICT (content): Merge conflict in Xyz.java
```

  - Both branches have changed the *same parts* of Xyz.java.
  - Git couldn't work out how to merge both changes.
- Git will leave both changes in Xyz.java, with notes on which branch each one comes from.
- You must:
  1. Edit Xyz.java, see both changes, and figure out for yourself how to reconcile them.
  2. Stage and commit your new changes:

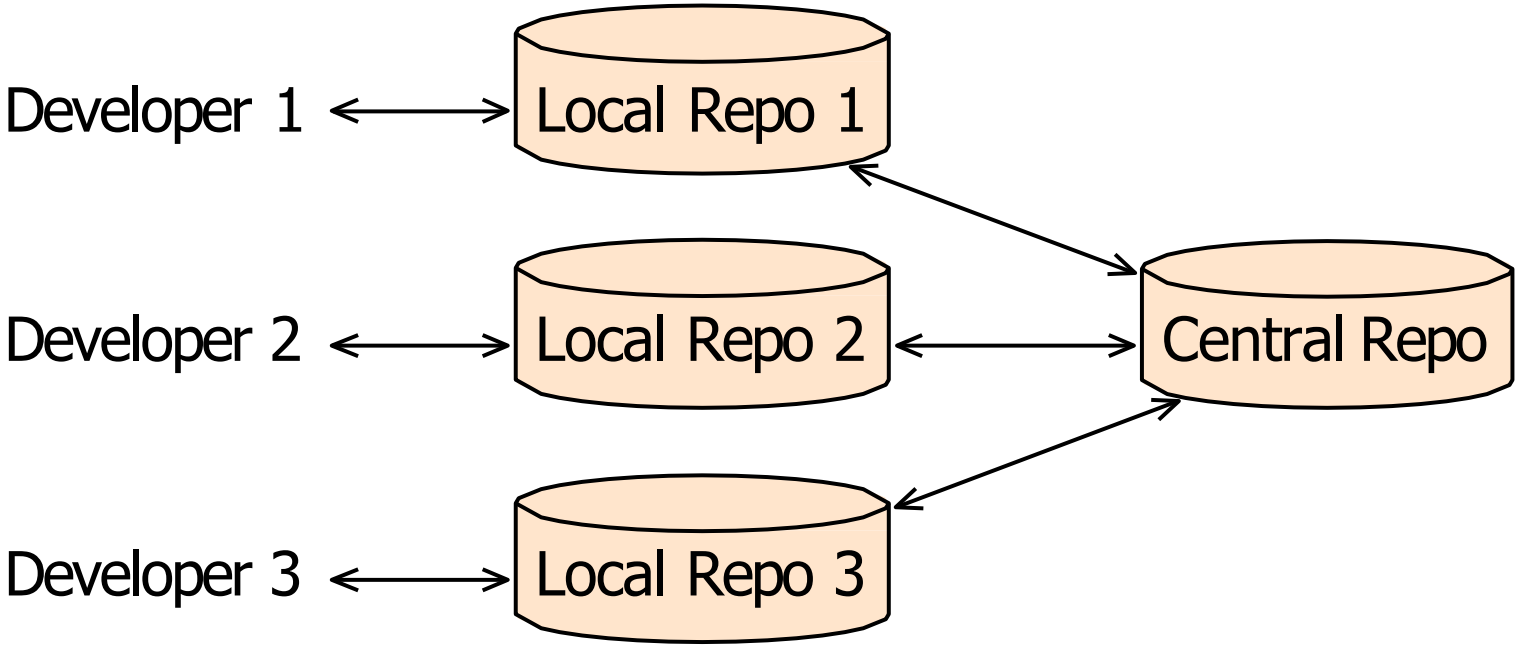
```
[user@pc]$ git add Xyz.java
```

```
[user@pc]$ git commit -m "Merged abc-xyz changes."
```

- (nb. "`git status`" also shows what file(s) need attention.)

Multiple Repositories

- Recap: you typically use *multiple* repos (in a distributed VCS).
- Typically:
  - Each developer has a "local" repo. This is what we've been working with so far.
  - There's also a central repo.



Multiple Repositories – Why?

One repo keeps track of everything. So why *more* than one?

1. Safety.
   - With a centralised VCS, if the repo gets corrupted/deleted, you could lose everything!
   - Distributed version control is a natural backup system.
2. Reliability. If/when the central server goes down, then:
   - With a distributed VCS (e.g. Git) you still have a local repo.
   - With a centralised VCS, nobody can do any work – massive lost productivity.
3. Performance. For large teams and large projects:
   - A centralised VCS can get swamped, both by CPU intensive tasks and network traffic.
   - A distributed VCS doesn't need the network most of the time, and spreads around the CPU load. Team size is irrelevant to performance, because each developer uses their own PC.

Remote Repositories in Git

- Each repository keeps track of "remotes".
  - Typically there's a remote called "origin" – the central repo.
  - Git is flexible, though:
    - You could have *no* central repo at all, and instead sync up with other team members' local repos directly.
    - You could have several central repos in a complicated hierarchy.
- The central repo (if it exists) is mostly for coordination:
  - Removes any doubt as to *who* has the latest version.
  - Minimises the amount of syncing needed – you only need to keep your local repo in sync with *one* other (typically).

Cloning

- Cloning *creates* a new repository based on an existing one.
- Also sets up the original repository as the "origin" (a remote repo) for the new one.
- Typically used to create a new local repo based on an existing central repo:

```
[user@pc]$ git clone https://xyz.com/myproject.git
```

  - This assumes we have an existing (central) repo on xyz.com.
- We can also create a 2nd local repo:

```
[user@pc]$ git clone myproject myproject2
```

```
[user@pc]$ cd myproject2
```

  - Assumes we're one level up from the `myproject/` directory.
  - One local repo should really be "bare" – see the prac worksheet.

Pushing and Fetching

- You can *push* and *fetch* to keep different repos in sync.
  - Typically done on *one branch* at a time.
  - Copies any commits not already copied.
- **Pushing TO a repote repo:**
  - Upload "mynewbranch" to "origin" (and make "origin" the current remote):

```
[user@pc]$ git push -u origin mynewbranch
```

  - Upload the *current* branch to *current* remote:

```
[user@pc]$ git push
```

- **Fetching FROM a remote repo:**
  - Get "master" from "origin":

```
[user@pc]$ git fetch origin master
```

  - Get *everything* from the *current* remote:

```
[user@pc]$ git fetch
```

Don't Panic

- Can push and fetch overwrite things?
- What if multiple developers push changes at the same time?
- You don't have to worry!
  - Each commit is considered "immutable" (unchangeable).
  - Therefore, anything "pushed" or "fetched" must in the form of *new* commits.
  - New commits never overwrite existing ones – they're just added.

Working as a Team

Git can be used in different ways, but here's a reasonable scenario:

1. The central repo contains a "master" branch.
2. Developer 1 wants to work on a new feature:
    1. *Fetches* the master branch (so the local repo is up-to-date).
    2. Creates a new "featureX" branch in their local repo.
    3. Gets featureX working, making some commits.
    4. *Pushes* featureX (the whole branch) to the central repo.
3. Developer 2 performs a code review:
    1. *Fetches* featureX from the central repo.
    2. Inspects the code.
    3. If anything is wrong, we repeat from step 2.3.
4. On the central repo, the team now *merges* featureX into master.
5. All developers *fetch* master from the central repo (so as to be up-to-date).

That's all for now!