Software Design and Construction
159.251

Version Control and version management

Amjed Tahir

a.tahir@massey.ac.nz

Remember these question from last lecture….

- I cannot find the latest version of this program or document?
- A *disaster*: the latest version of the code was ***overwritten*** by an old version!!
  - I've lost all my latest changes – two days of work!!!
- Nobody knows **which version** of the program is final.

Managing Source code

- Files change over time
  - can be useful to have older versions accessible
  - programming is especially fraught
    - *seemingly small changes can stop projects from working*
    - *often a large number of interdependent files*
- Ad-hoc backups
- Renaming the files with the date
- Zipping the folder

File sets have a temporal dimension

- To build a project the correct versions of files are needed
  - not just files of the same name
  - these may not be the latest version
  - may not be able to use current release of a library
  - files may be present but moved

"Revision Control Systems" manage versions of files

- Originally designed for textual documents
- Keeps latest version and deltas for earlier revisions
- Works only on single files, not projects (large set of files)
- No server - data held in subdirectory
- Introduced in 1982, still in use on some Wikis (user editable sites)

Single vs. ***shared*** development

- **Single developers are (relatively) easy**
  - *Mostly* no problem with clashes
- **Shared development is common**
  - Work with multiple developers.
  - Sometimes, they are located in different locations (i.e., the outsourcing model).
  - Can be quite busy (i.e., frequent development).

What's Version Control (VC)?

- VC combines procedures and tools to manage and control different versions of the software, while production.

- A **version control system** (VCS) records changes to a file or set of files over time so that you can recall specific **milestone** later.
- Version:
  - A well-defined state of a configuration item at a given point of time.

- A baseline is a ***snapshot*** of the system that has been formally reviewed and agreed upon (approved).

Overview of VC

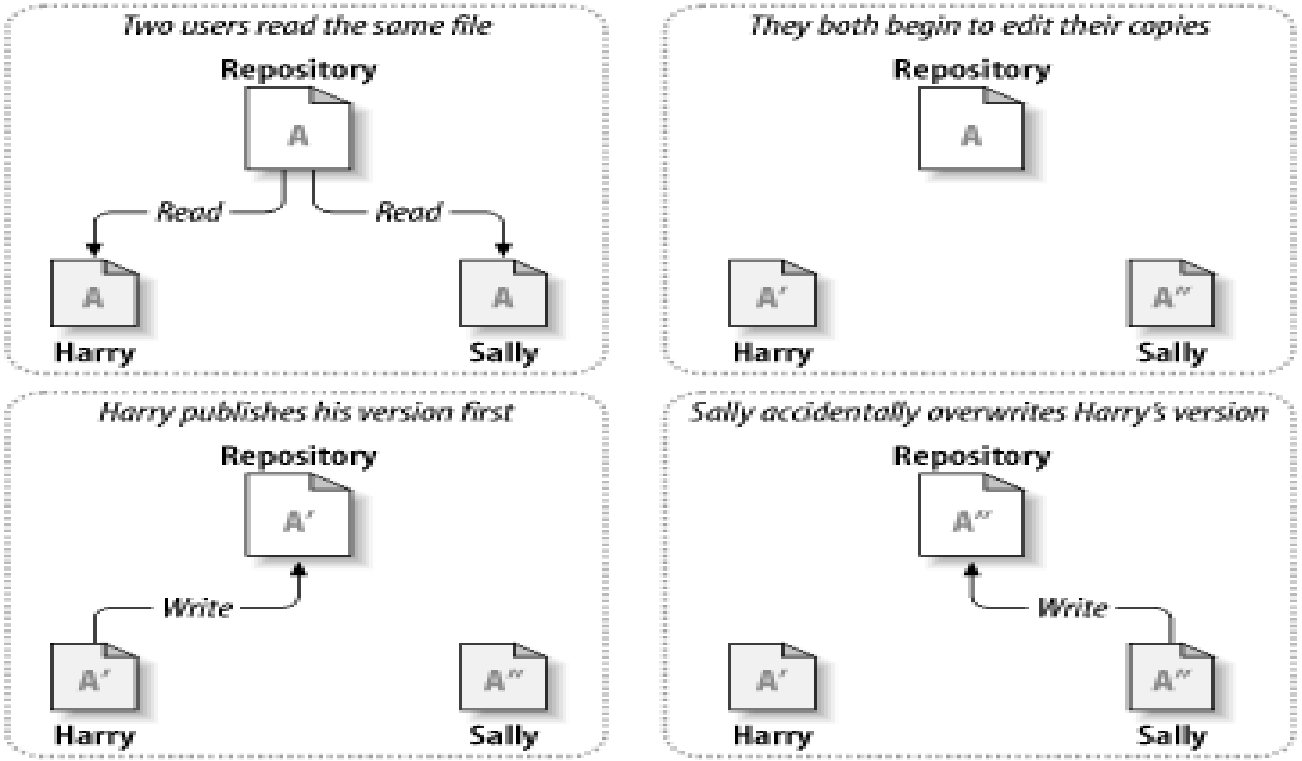
- Part of SCM process….
- VCS is a method to record all changes made to a file (or a set of files) over time so that you can recall specific versions later.
- In software development, the term VC mostly refers to source-code VC.
  - But this shouldn't be always the case.
- A VCS allows to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time and control developers progress.

History of VC in software development

- Initially, version control was done manually (i.e., manually record all changes by all developers) – and mostly for the purpose of tracking bugs.
- Traditional revision control systems use a centralized model - all the revision control take place on a shared server.
  - If two developers try to change the same file at the same time (without some method of managing access), the developers may end up overwriting each other's work.

The centralized model

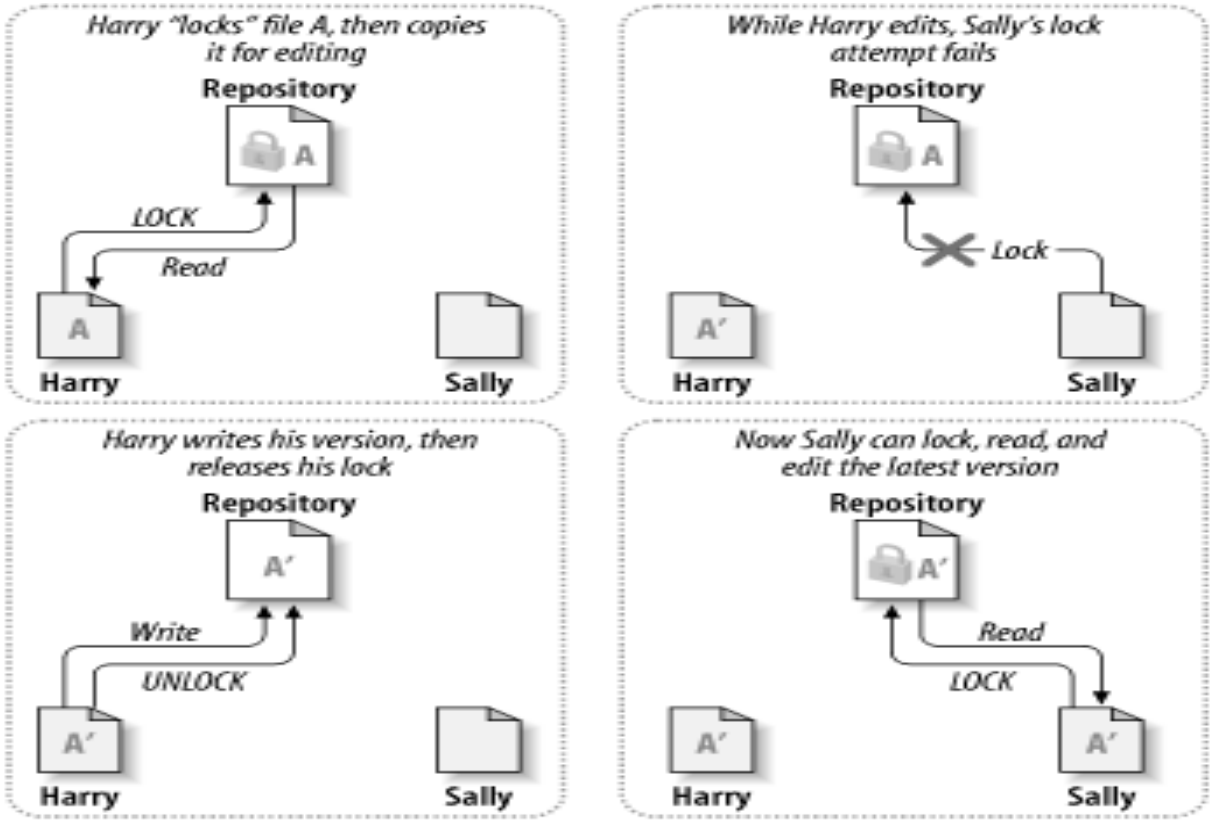
- The client-server model…



Problems ?

The “Lock-Modify-Unlock” model

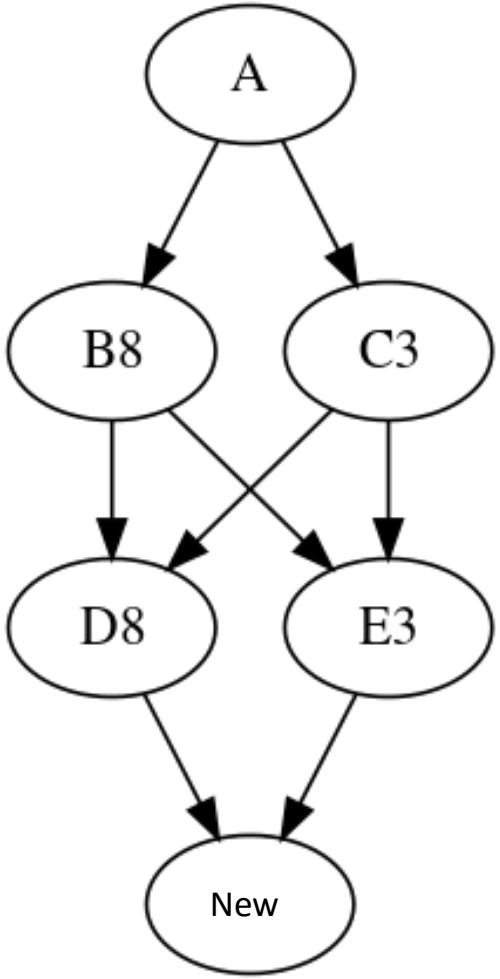
- Allows ***only one*** developer to access a file at a time.



Problems?

The “version-merging” model

- Multiple developers can deposit the file into a central repository at the same time.
- Preserve the changes from the first developer when other developers check in.
- Merging is easy with simple structure files such as text files, but quite difficult with source code files.



Distributed Versions Control (DVC)

- A decentralised model
  - follows a peer-to-peer approach.
- All multiple developers to work on the same file without requiring everyone to share the same network.
- Every developer work in his own repository (no one central repository).
- Check-in, Check-out and commits are faster (you don’t need to communicate to the central server).

Distributed Versions Control

Cont'd

- Multiple "central" repositories
- Each working copy with each developer can effectively function as a backup, protecting against data loss.
- **central server** are not essential
- Users can work off-line
- Main issue:
  - Slow *cloning* of the repository – compared to the centralised check-in model.

Widely used VCS

- There are a number of versions control systems out there.
- Depending on your model of preference.
- A light comparison of VCSs is provided here

  https://en.wikipedia.org/wiki/Comparison_of_version_control_software
- Be away of the development status of your VCS.

Examples of VCS

- **CVS**
  - A centralised VCS – last release was in 2008.
- **SVN**
  - A distributed model – widely used within the OSS community
- **Git**
  - Was for the development of Linux kernel following an issue with BitKeeper.
- **Mercurial**
  - implemented using the Python! One the newest.
- **BitKeeper**
  - A distributed VC system- previously used by Linux kernel developers.
- **GNU Bazaar**
  - One of a few VCS that supports both distributed and client–server revision control system

Concurrent Versioning System (CVS)

- Released in 1990
- uses a client-server model
  - *the server is centralised **repository** of changes*
  - *users must be able to access the server*
- CVS labels a single project (set of related files) that it manages as a module.
- A CVS server stores the modules it manages in its repository.
- Check the CVS main repository (for the original CVS project!) here: http://cvs.savannah.gnu.org/viewvc/?root=cvs

How does CVS works?

A client–server architecture

- A server stores the **current version(s)** of a project and its history,
- Clients connect to the server in order to "check out" a complete copy of the project,
- Clients work on their local copies of the files
- Later clients "check in" their changes.
- Typically, the client and server connect over a LAN or over the Internet …
- The server software normally runs on Unix.

Why projects use CVS?

  - Fully free and open-source
  - Ubiquity of CVS (used to be!)
  - Large number of legacy users
  - The development team provided a solid documentation
- Developments have stopped in 2008 (latest stable release)!
- Most current OS projects migrated their source code to other DVCS systems.

Apache Subversion (SVN)

- Subversion: designed as a successor to CVS. (aka: *CVS done right!*)
- Started in 2000.
- Still under development – latest release on April 2016
- Atomic commits (unlike CVS)
- Complete directory & metadata versioning
- Handles binary files
- Multiple access methods: designed to minimise network traffic
- Also uses Apache HTTP server as a network server.

file:// - via file system
http:// - with web server
svn:// (with dedicated svn server "svnserve/svnserve.exe")

Subversion

Cont’d

- partial checkouts
- locking is possible
- some files aren't mergeable
- files can be moved and renamed, with history
- ignore lists
- svn: keywords (author, date, revision)

Subversion - difficulties

- **Branches have different filesystem path**

```
---+
   |----- Trunk
   |       main.py
   |
   +---- new-Feature-Branch
         |
         +- main.py
         |
         +- new.py
```

- **Need access to central repository**
  - difficulties when offline
  - all commits MUST be public (everyone can see the commit history)

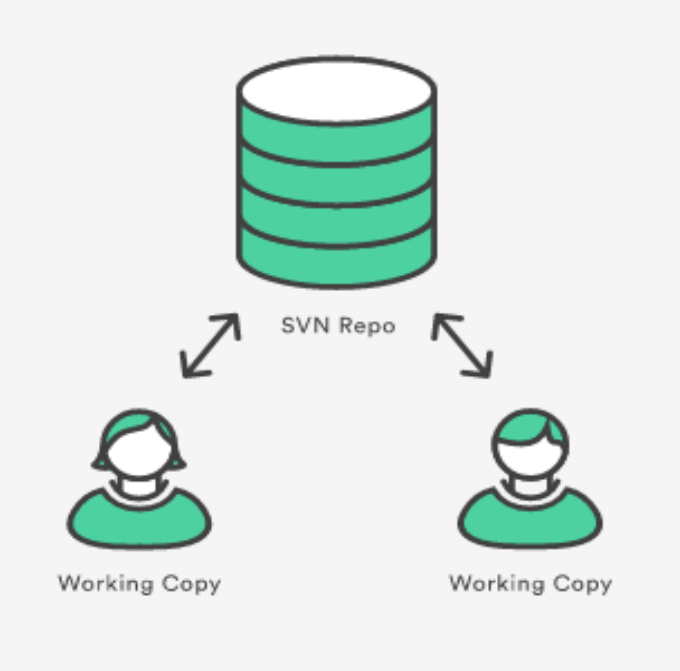
Git

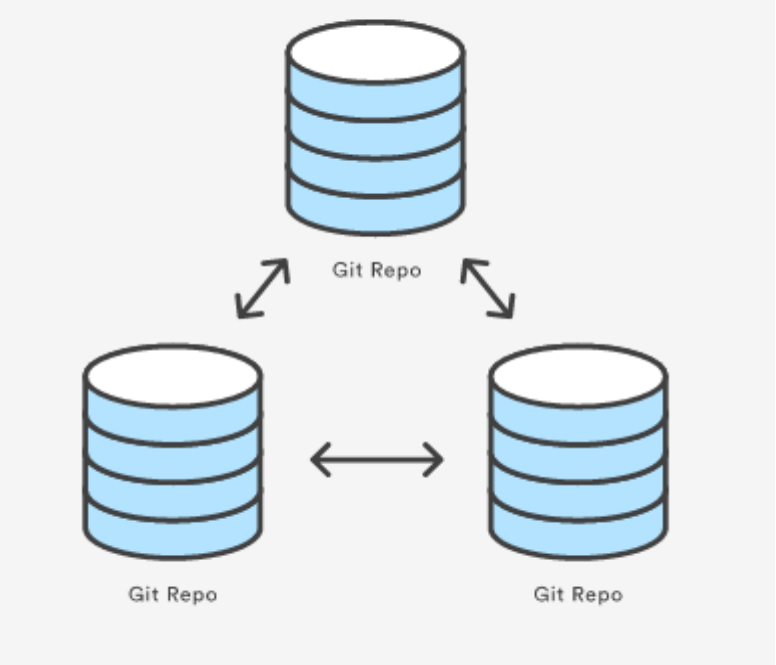
- Designed by Linus Torvalds to handle development of the Linux Kernel (was build in a few weeks!)
- It is generally assumed that the community of Linux kernel developers is composed by 5000 or 6000 members.
- The first version of Linux kernel (v.0.01) had 10K lines of code, where the latest release (v.4.1) had over 19M lines of code contributed by 14K developers! http://en.wikipedia.org/wiki/Linux_kernel#Development

Characteristics of git

- Designed to allow multiple concurrent developers
- Supports the notion of a **change author** and a **change committer**
- Based on a checkout lastest, and merge model (no locking)
- Primarily intended for use with source/text files
  - *not intended for very large files*

- The main difference between Git and SVN





Two approaches to learn git

- **Learn a recipe of commands**
  - watch/read tutorials
  - fine for getting started
  - not the best long term
- **Understanding how it works**
  - doesn't immediately help with using it
  - much more productive long term
- **An aside - on productivity**
  - learn to put effort in for the longer term return
  - systems keep changing, this requires constant relearning
  - try to pick "systems" that'll have a long life

  And
    - *it takes time and effort to become really good with a tool …*

Concepts in git (and other DVCS)

- **repository (repo)**
  - where all the history is kept. It may be may be local or remote
- working copy
  - the directory containing your files, the older versions of which are in the repo
- Commit
  - telling the VCS to "save the state of the working copy (or a subset)
- commit ID
  - a id string of some sort that identifies a particular commit
    - can be used to retrieve that commit later
- commit message
  - a useful comment about what was changed/fixed within a commit
- loga
  - list of the commit messages and commit IDs

Concepts

cont'd

- Branch
  - an alternate path of development
    - the "Main" branch (in git) is called **master**
- Merge
  - combining all the changes from one or more branches into one line of development
- pull/update
  - pull (get) changes from the repo so the working copy matches the specified commit (usually the **HEAD**)
- repository public key
  - a method of accessing a remote repository without having to specify passwords usually using *ssh*.

Using Git from command line

- Assuming you've installed git (instructions are on Stream)
  - Type **git** in the command line to check if git is installed!

- **Configure Git**
  - You'll set up many repositories on your PC, some options are global (apply to all repos)
  - Global options are stored in a **.gitconfig** file in your home directory

%HOME%\.gitconfig	in Windows
~/.gitconfig	in Linux & MacOS/X

- **Tell git your name & email address**
  - Git requires minimal configuration, but to provide meaningful commit messages, it needs to know your name and email address

```
git config --global user.name "ATahir“
git config --global user.email a.tahir@massey.ac.nz
```

- Git will use the default text editor of your system, unless you configure it.
- The following command will change the editor into Notepad++, for example.

```
git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession"
```

- To review your configuration:

```
git config --list
```

```
C:\Users\atahir>git config --list
core.symlinks=false
core.autocrlf=true
core.fscache=true
color.diff=auto
color.status=auto
color.branch=auto
color.interactive=true
help.format=html
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt
diff.astextplain.textconv=astextplain
rebase.autosquash=true
credential.helper=manager
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.required=true
user.name=Amjed Tahir
user.email=a.tahir@massey.ac.nz
core.editor='C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession
```

Where to find your configuration file?

- Git configuration file allows you to set configuration variables that control how Git operates.
- Windows:
  - .gitconfig file can be found in your home directory i.e., (C:\Users\UserXYZ)
  - there is also a system-level config file at C:\ProgramData\Git\config
- In Unix-like systems
  - /etc/gitconfig file: Contains values for every user on the system and all their repositories.
  - ~/.gitconfig or ~/.config/git/config file: Specific to your user. You can make Git read and write to this file specifically by passing the --global option.
  - config file in the Git directory (.git/config) of whatever repository you're currently using: Specific to that single repository.

Using Git

- There are few concepts that you need to know:
  - Create a repository
  - Committing
  - Branching
  - Tagging
  - Merging
  - Cloning

Creating a repository

- Repositories are just like folders in your machine.
- It's not possible to commit just a single file
- The complete state of a directory (and subdirectories) are committed
  - *the commit is a **snapshot of a set files/dirs** in that folder at a point in time*
- ***git init*** turns any folder into a repository
  - creating a repository adds a **.git** folder (with subfolders)
  - existing content won't be altered

```
E:\>git init
Initialized empty Git repository in E:/.git/
```

Deposit Changes (commit)

- First, add your files to the repository
-

```
git add leapYear.py
git add LICENSE.txt
git add leapYear.java
```

OR

```
git add .
        # (to add all files to the depository)
```

- Then, commit these files

```
git commit
```

Once successful, you'll see the following message

```
E:\gitProject>git commit
[master (root-commit) 369501f] ABC
 3 files changed, 34 insertions(+)
 create mode 100644 LICENSE.txt
 create mode 100644 leapYear.java
 create mode 100644 leapYear.py
```

Commit

- Individual commits are each given a unique hash value (i.e., ID)
- The commit ID is based on:
  - the author's name. email and timestamp
  - the committer's name, email and timestamp
  - the commit message for this commit
  - the hash of the parent commit(s)
- Commit ID can help tracking issues and also productivity….
- Review the change in your commit using `git log -p filename`

- Check the status of the commit (after committing the files)

```
git status
```

```
E:\>git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   LICENSE.txt
        new file:   leapYear.java
        new file:   leapYear.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        $RECYCLE.BIN/
        backup/
        gitProject/
```

Git index - the staging area

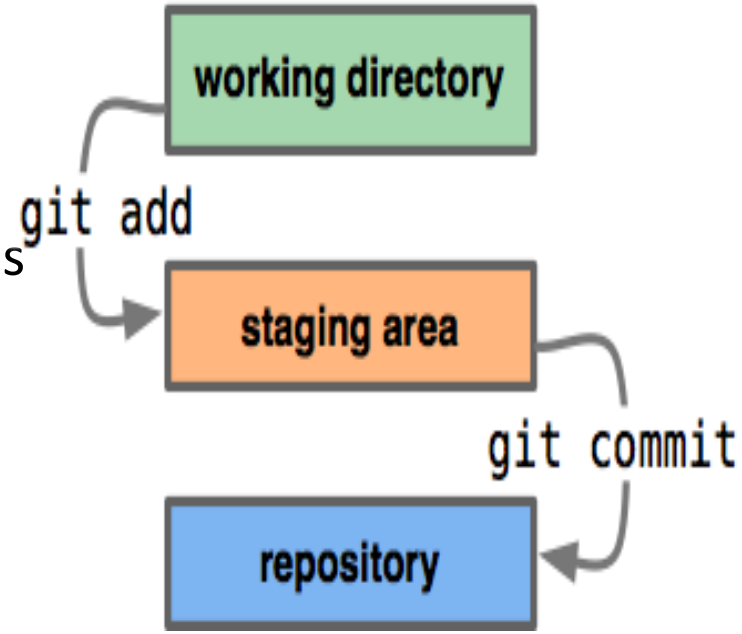
- 'index' is where you place files you want committed to the git repository.
- Act like *cache.*

- Before you commit files to the git repository, you need to first place the files in the git index.

- Important:
  - The index isn't your "Working Directory"
  - The index isn't your " Git Repository"

- Use the following command to check with files are in the git index

```
git ls-files
```



- Deleting a repository is easy
  - Remember: a repository is just a directory with working copy.

- Just delete the .git subfolder and the repository will be deleted!

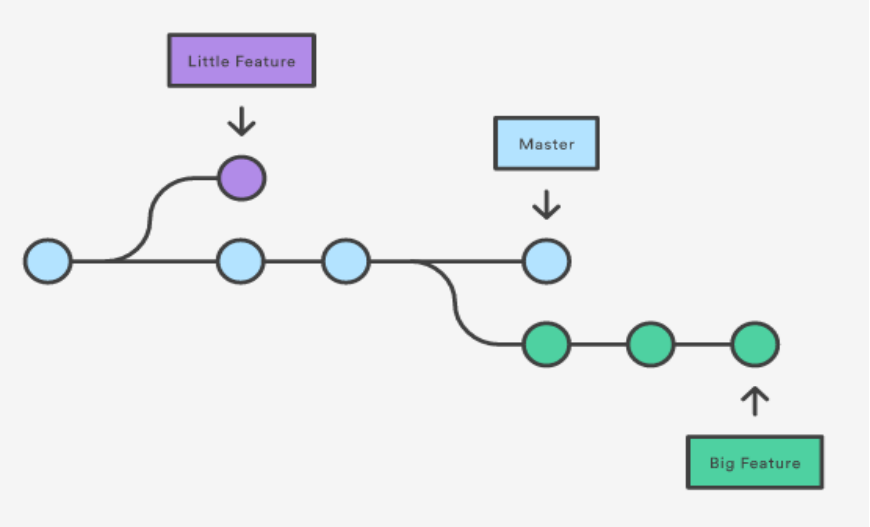
Ignoring files - *.gitignore

- Many files that don't need to be committed
  - temp/backup files
  - generated files (e.g., ***.class*** files)
  - executables
  - log files
- You can specify which files you want to ignore by updating the ***.igitignore*** settings.
  - See more information here: https://help.github.com/articles/ignoring-files/

Branching

- A ***branch*** is an independent line of development.
- Essential for *collaborative* and *parallel* development.
- Think of a branch as a way to request a *brand new* working directory from the repository.

A ***master branch*** is the default branch created when you initialised your git repository.
It points to the last commit you made.



Branching commands

`git branch`	To list all of the branches in ytour repository
`git branch <newBranch>`	To create a new branch (called: newBranch)
`git branch -m <newBranch>`	Rename the current branch
`git branch -d < newBranch >`	Delete a specific branch (replace “d” with “D” to ***force delete***)
`git checkout`	To navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch
`git checkout -b newBranch`	Create a branch before checking out

Example

- Assume that we are adding a new feature to a program in a repository:
  - Branch the new feature
  - The checkout the feature
  - Then add your files
  - And finally, commit

```
git branch Feature
```

```
git checkout Feature
```

```
git add <files>
```

```
git commit <files>
```

Example of branch, checkout and commit

All of these are recorded
in freshBranch, which is completely isolated from the master branch.

All the commits here are not going to effect other branches.

```
E:\>git branch freshBranch

E:\>git checkout freshBranch
D       LICENSE.txt
D       leapYear.java
D       leapYear.py
Switched to branch 'freshBranch'

E:\>git add NewFeature.xyz




E:\>git commit -m "new feature development has started"
[freshBranch 271d826] new feature development has started
 4 files changed, 34 deletions(-)
 delete mode 100644 LICENSE.txt
 create mode 100644 NewFeature.xyz
 delete mode 100644 leapYear.java
 delete mode 100644 leapYear.py
```

Tagging

- Tags can be created to mark particular commits
  - e.g., to tag specific points in history as being important.
  - Release v.5.2.
- To list all available tags

```
git tag
```

- You can also search for a particular tag a particular pattern
- Two types of tags:

```
git tag -l "v5.5*"
```

  - **Lightweight**: just a pointer to a specific commit (just the name of the commit)
  - **Annotated**: stored as full objects in the Git database (full details: tagger name, email, date and a massage)

Tagging Examples

Lightweight Tags

```
E:\>git tag v2.2

E:\>git tag


v2.2
```

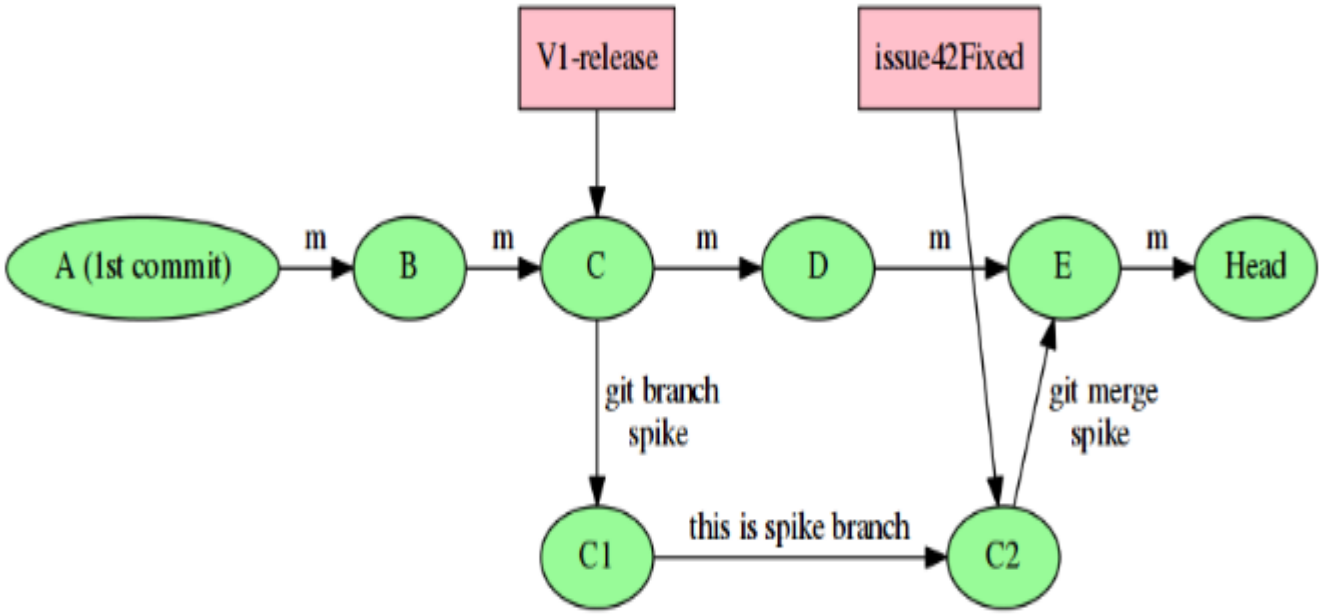
Annotated Tags

```
E:\>git tag -a v.3.4 -m "new tag with new features"

E:\>git tag
v.3.4
v1.1
v1.4
v2.2
```

You can see the tag data along with the commit that was tagged

git tag -a v5.2 -m "new tag with new features"

```
E:\>git tag -a v5.2 -m "new tag with new features"

E:\>git show v5.2
tag v5.2
Tagger: ATahir <a.tahir@massey.ac.nz>
Date:   Wed Aug 10 16:39:07 2016 +1200

new tag with new features

commit 3a3e8bdbd3a487a5570351fe60c02758f01387a0
Author: ATahir <a.tahir@massey.ac.nz>
Date:   Wed Aug 10 15:44:01 2016 +1200

    asdf

diff --git a/test.test b/test.test
new file mode 100644
index 0000000..e69de29
```


V1-release
issue42Fixed
A (1st commit)
m
B
m
C
m
D
m
E
m
Head
git branch
spike
git merge
spike
C1
this is spike branch
C2

Merging

- Branches are only useful if you can merge lines of development together.
- The ***git merge*** command allows you take the independent lines of development created by ***git branch*** and integrate them into a single branch.

```
git merge <branch>
```

- Example
  - ***Create a branch*** called “feature”
  - ***Checkout*** the branch (i.e., make it the working directory)
  - ***Add*** files
  - ***Commit*** changes
  - ***Merge*** the new branch with the master branch
  - Finally, delete the branch

git branch feature #new branch

git checkout feature

git add leapYear.java leapYear.py License.txt NewFeature.xyz

git commit leapYear.java leapYear.py License.txt NewFeature.xyz

git checkout master #return to the master branch

git merge feature #merging 'feature' with 'master'

git branch d- feature #delete 'feature'

```
E:\>git merge feature
Updating 1d61a3e..ba3d5ba
Fast-forward
Auto packing the repository in background for optimum performance.
See "git help gc" for manual housekeeping.
Counting objects: 16, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (16/16), done.
Total 16 (delta 4), reused 0 (delta 0)
Checking connectivity: 11601, done.
warning: There are too many unreachable loose objects; run 'git prune' to remove them.
 NewFeature.xyz | 0
 test.test      | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 NewFeature.xyz
 create mode 100644 test.test

E:\>git branch -d feature
Deleted branch feature (was ba3d5ba).
```

Working with remote repositories

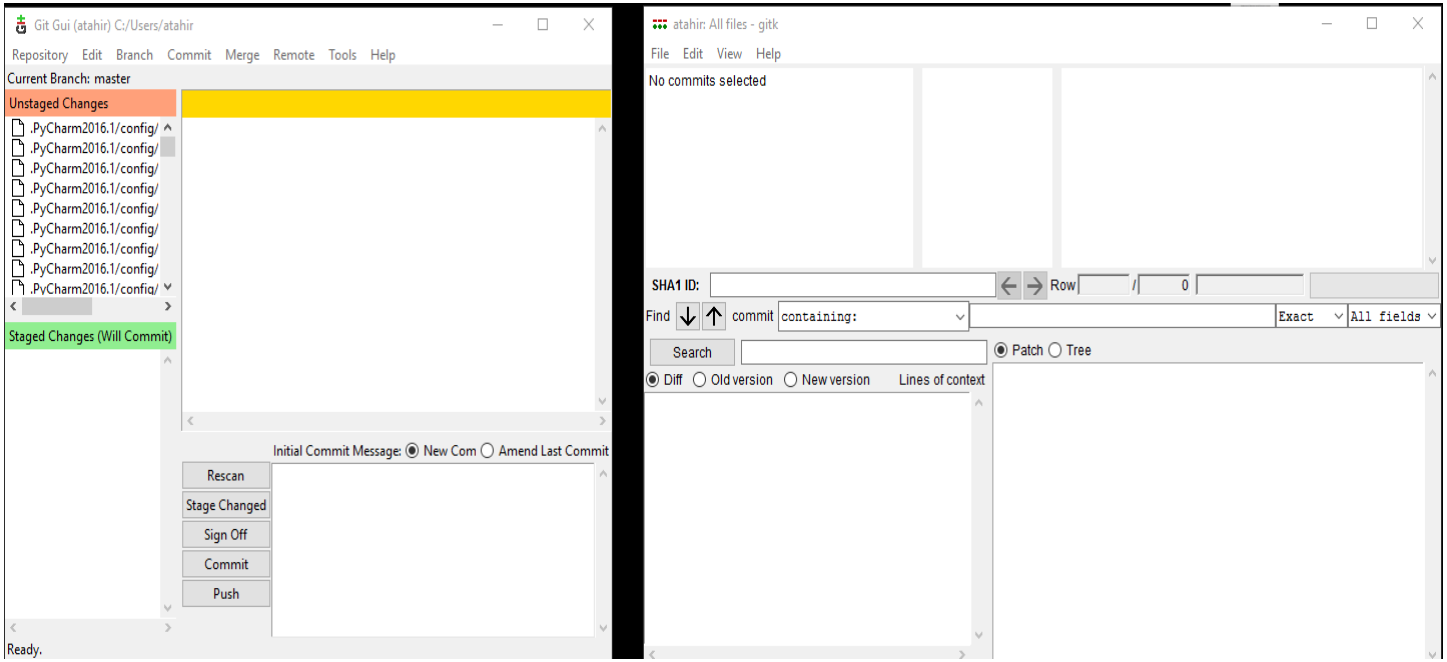
- Clone is used to retrieve a copy of an existing Git repository.
  - makes a local copy of another repository

```
git clone <repo> <directory>
```

    - Repo: can be located on the local filesystem or on a remote location (accessible via HTTP or SSH)
- Cloning depends on the size of the files and your connection as well

Git clients

- Git comes with with two GUI components:
  - git-gui for committing
  - gitk for browsing



- There are a number of places where you can host your Git repositories.
- Choose your hosting repository service based on your needs..
- A small list of well-known sites (supports git)
  - GitHub
    - supports git, Mercurial and SVN - private repository are not free
  - Bitbucket
    - Supports private repositories
  - GitLab
    - Open-source, widely used by large agencies and corporations such as NASA and CERN.
  - And many others
    - See this comparison : https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities

Git clients

- A number of GUI clients are available to manage your Git repositories…
- Make it ‘easy’ to manage your large repositories (especially when you have many tens and tens of branches!)
- Well-known:
  - GitHub Desktop : Windows and Mac
  - SourceTree: Windows and Mac
  - git-cola: Linux, Windows and Mac
  - GitKranken: Linux, Windows and Mac

Resources

- Additional resources on GIT:
  - Git official documentation

    https://git-scm.com/doc
  - Detailed tutorial on Git from Atlassian

    https://www.atlassian.com/git/tutorials/
  - A simple step-by-step guide on Git

    http://www.gitguys.com