Software Design and Construction

159.251

Version control and version management

Amjed Tahir

a.tahir@massey.ac.nz

- What version control is and its use
- Types of version control systems
  - Centralized
  - Distributed
- **git**, a widely used version control system
  - Concepts
  - Usage

Problems in managing source code

- **Disaster**! the latest version of the code was *overwritten* by an old version! Lost all my latest changes! Lost old version of code.
  - What changes fixed some bug?
  - Who made this change?
  - Where's the source code for this release?
- **Confusion**: What's the current version of the code?

Managing source code

- Files/code changes over time (bug fixes, new/deprecated features, refactoring)
  - Collaboration (Single vs shared development)
  - Small changes can *stop projects from working*
  - Often *a large number of interdependent files*
  - Useful to have older versions accessible
    - Regressions
    - Traceability and comprehension
    - Support/maintenance
- Solutions:
  - Ad-hoc backups
  - Renaming the files with the date
  - Zipping the folder

Overview of version control systems (VCS)

- Part of software configuration management (SCM) - (see previous topic).
- VCS are used to record all changes made to a file (or a set of files) over time so that you can recall specific versions later. Also records metadata associated with change.
- In software development, the term VC mostly refers to source-code.
  - But this shouldn't be always the case.
- A VCS allows reverting files back to a previous state, revert the entire project back to a previous state, compare changes over time and control developer's progress.
- A baseline is a ***snapshot*** or version of the system that has been formally reviewed and approved.

Benefits of VCS

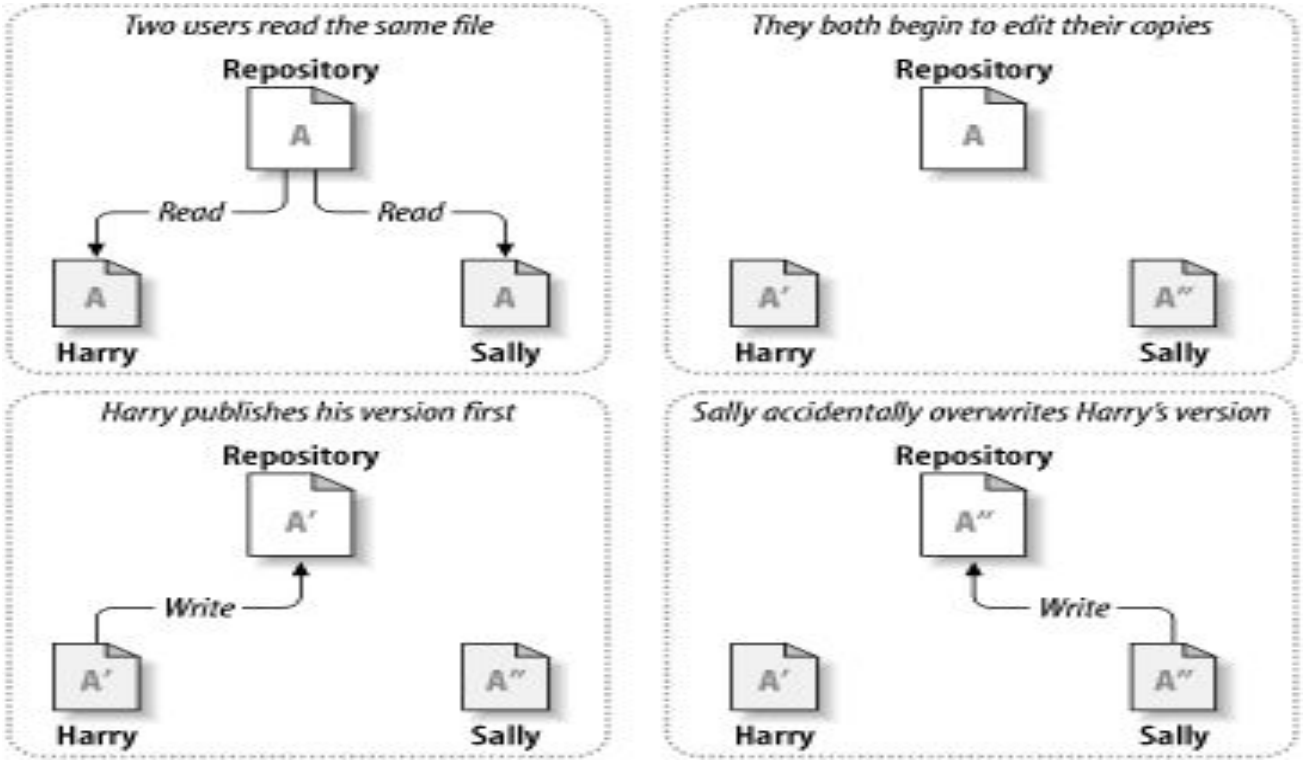
- Work in isolation on a project and share changes at any time.
- Ensures temporary or incomplete edits don’t interfere with other developer’s work.
- Automatically merge changes by different developers
- Easier to collaborate with others.
- Easier to work on /execute code-base from multiple computers ((e.g., deployment across machines.)

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.
- **Stand-alone**
  - Source Code Control System (SCCS), Revision Control System (RCS)
    - Single developer, file-focussed, text files only
- **Centralized**
  - Concurrent Versions System (CVS)
    - Centralized repository, file-focused
  - Subversion (SVN)
    - Tracks directory structure changes, binary files, atomic operations
  - **Distributed**
    - No shared central repository

The centralized model

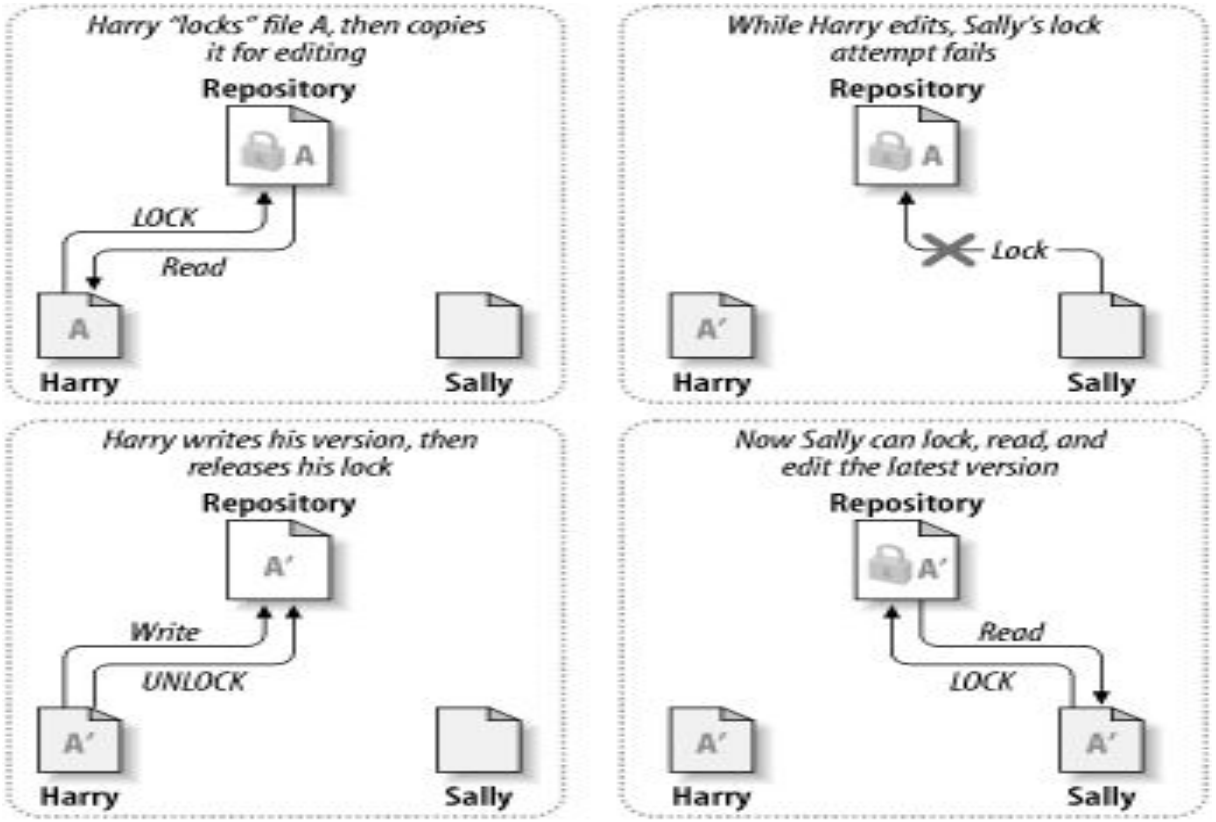
- The client-server model…



Problems?

The “Lock-Modify-Unlock” model

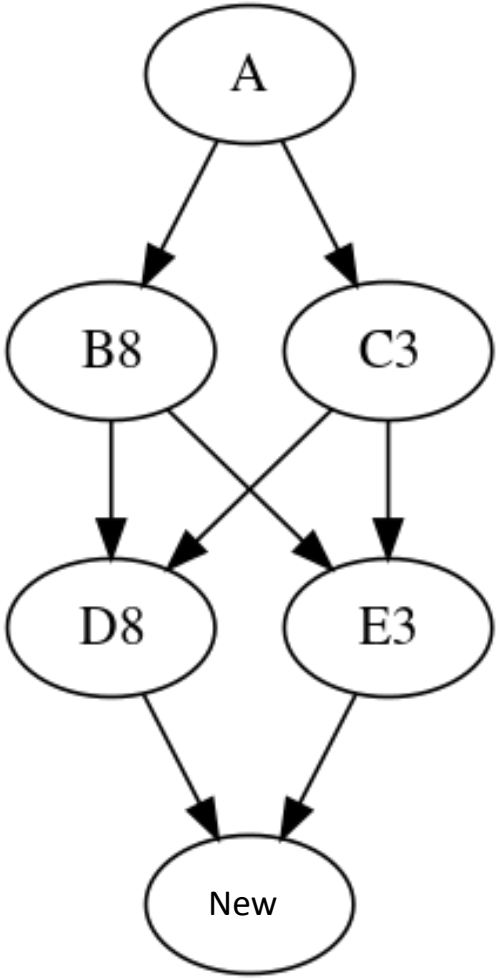
- Allows ***only one*** developer to access a file at a time.



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.
- Multiple developers to work on the same file without requiring everyone to share the same network.
- Every developer work in their 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** is not essential
- Users can work offline
- 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 aware 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
- **BitKeeper**
  - A distributed VC system- previously used by Linux kernel developers.
- **Git**
  - Was for the development of Linux kernel following a licensing issue with BitKeeper .
- **Mercurial**
  - One of the newest. Similar to Git but with different branching structure.
- **GNU Bazaar**
  - One of a few VCS that supports both distributed and client–server revision control system

and many more …

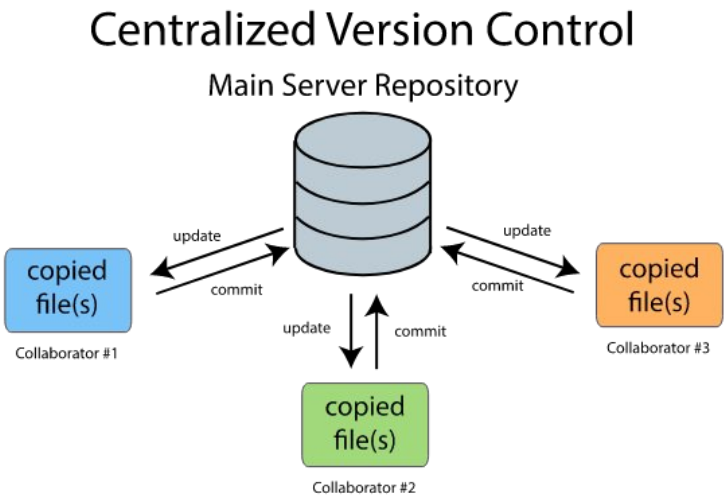
Concurrent Versioning System (CVS)

- 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 work?

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 normally runs on Unix.



- Fully free and open-source
- Large number of legacy users and projects
- Solid documentation

But outdated now …

CVS is not widely used anymore...

- Developments have stopped in 2008 (latest stable release)!
- Most current OS projects migrated their source code to other DVCS systems, mainly because centralised VC systems have many disadvantages… (see next slides)

Apache Subversion (SVN)

- Designed as a successor to CVS. (aka: *CVS done right!*)
- Started in 2000.
- Still in development- latest long-term support (LTS) release (1.14.2) on April 2022.
- Atomic commits (unlike CVS) => *indivisible* and *irreducible (all or nothing!)*
- 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")
```

SVN difficulties

- **Branches have different filesystem path**

```
---+
   |----- Trunk
   |       file.java
   |
   +---- new-Feature-Branch
         |
         +- file.java
         |
         +- new.py
```

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

Git

- Developed by Linus Torvalds (creator of Linux) to handle development of the Linux Kernel (was built in a few weeks!)
- Community of Linux kernel developers is composed of around 15,600 (number of contributors since 2019).
- The first version of Linux kernel (v.0.01) had 10K lines of code, where the latest releases had 27m lines of code contributed by >4k developers!
  http://en.wikipedia.org/wiki/Linux_kernel#Development

wiki

Characteristics of git

- Designed to allow multiple ***concurrent*** developers
- Supports the notion of a **change author** and a **change committer**
- Based on a *checkout latest*, and merge model (no locking)
- Primarily intended for use with source/text files
  - *Multiple revisions of large files increase the clone and fetch times for other users of a repository.*
  - *Not intended for very large files → special version of git aka Git Large File Storage (LFS) that improves how large files are handled.*

Concepts in git (and other DVCS)

- **repository (repo)**
  - where all the history is kept. It may be local or remote
- **working copy**
  - the local 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**
  - identifies a particular commit
    - can be used to retrieve that commit later
- **commit message**
  - a useful comment about what was changed/fixed in a commit
- **log**
  - list of the commit messages and commit IDs

Concepts

- **branch**
  - an alternate path of development
    - the "Main" branch (in git) is called **main**
- **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

- **gitconfig** is the global git configuration file
- **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 "user"
git config --global user.email user@email.com
```

- 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"
```

linux like

```
open -a "Atom" .gitconfig
```

→ open the file in Atom text editor

- To review your configuration:

```
git config --list
```

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., (../UserXYZ)
  - there is also a system-level config file at ../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
  - Commit changes
  - Branching
  - Tagging
  - Merging
  - Cloning

Creating a repository

- Repositories are just like directories 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

```
git-example $git init
Initialized empty Git repository in /Users/atahir/git-example/.git/
```

Commit changes

- First, add your files to the repository

```
git add file1.java
git add LICENSE.txt
git add file2.java
```

OR

```
git add .
```

to add all files to the depository

- Then, commit these files

```
git comm -m "message"
```

Once successful, you'll see the following message

```
[git-example $git commit -m "added files"
[main (root-commit) 4d91b44] added files
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 LICENSE.txt
 create mode 100644 file1.java
 create mode 100644 file2.java
```

to check status

```
git status
```

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`

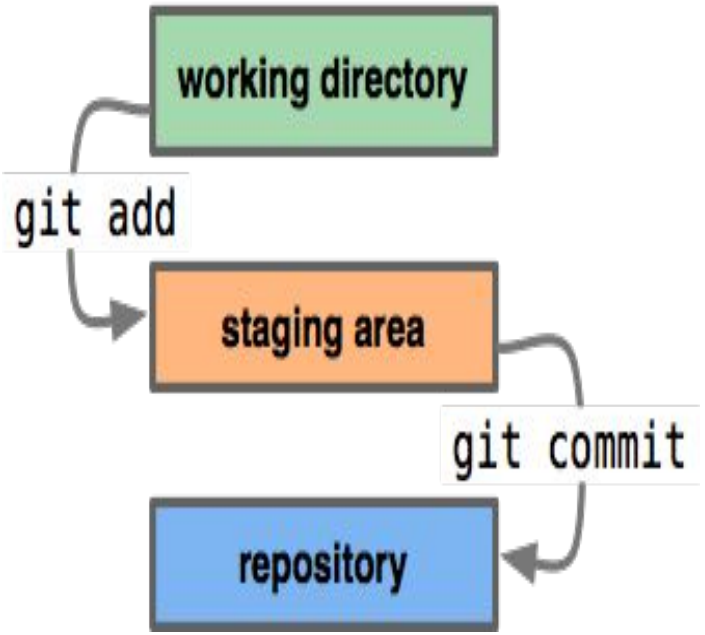
Git index - the staging area

- '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!

Warning: the following command will delete the entire git repository. Make sure that you run this from the correct folder

```
rm -rf .gitc
```

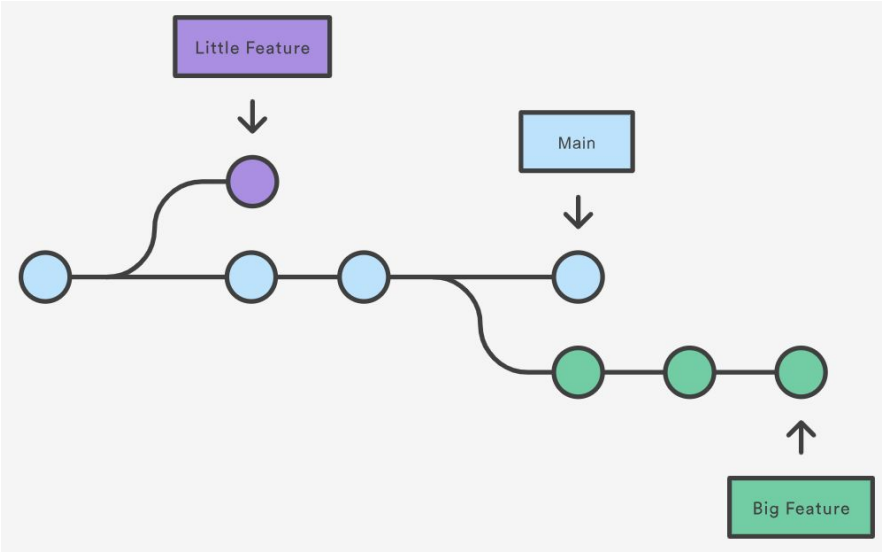
Ignoring files - .gitignore

- we usually have a set of files in each program that we do not really want to commit, such as
  - temp/backup files
  - generated binary files (e.g., ***.class*** files)
  - log files
- You can specify which files you want to ignore by updating the ***.gitignore*** settings.

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 ***main branch*** is the default branch created when you initialise your git repository.
- It points to the last commit in that branch.



Branching commands

`git branch`	To list all of the branches in your repository
`git branch newBranch`	To create a new branch with the name newBranch
`git branch -m newBranch newBranch2`	Rename the current branch
`git branch -d newBranch`	Delete a specific branch (replace “d” with “D” to ***force delete***)
`git checkout newBranch`	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

```
git branch Feature
```

  - The checkout the feature

```
git checkout Feature
```

  - Then add your files

```
git add <files>
```

  - And finally, commit

```
git commit <files>
```

note: default branch is `main`

Example of branch, checkout and commit

All of these are recorded in `Feature`, which is completely isolated from the main branch.

All commits here are not going to impact other branches.

```
git-example $git add .
git-example $git status
On branch Feature
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file3.java

git-example $git commit -m "added files to new branch"
[Feature 2907e4b] added files to new branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file3.java
```

to display the commit graph

```
git log --all --decorate --oneline --graph
```

```
git-example $git log --all --decorate --oneline --graph
* 2907e4b (HEAD -> Feature) added files to new branch
* 863ba54 Revert "Revert "added files""
* 980d4e4 Revert "added files"
* 4d91b44 (newBranch, main) added files
git-example $
```

Tagging

- Tags can be created to mark particular commits (like bookmarks)
  - e.g., to tag specific points in history as being important.
  - Release v.5.2.
  - They are fixed unlike branches
- To list all available tags

```
git tag
```

- You can also search for a particular tag a particular pattern

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

- Two types of tags:
  - **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 message)

Tagging Examples

Lightweight Tags

```
git-example $git tag v2
git-example $git tag
v2
```

Annotated Tags

```
git-example $git tag -a v3 -m "new tag"
git-example $git tag
v2
v3
```

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

```
git tag -a v2 -m "new tag"
```

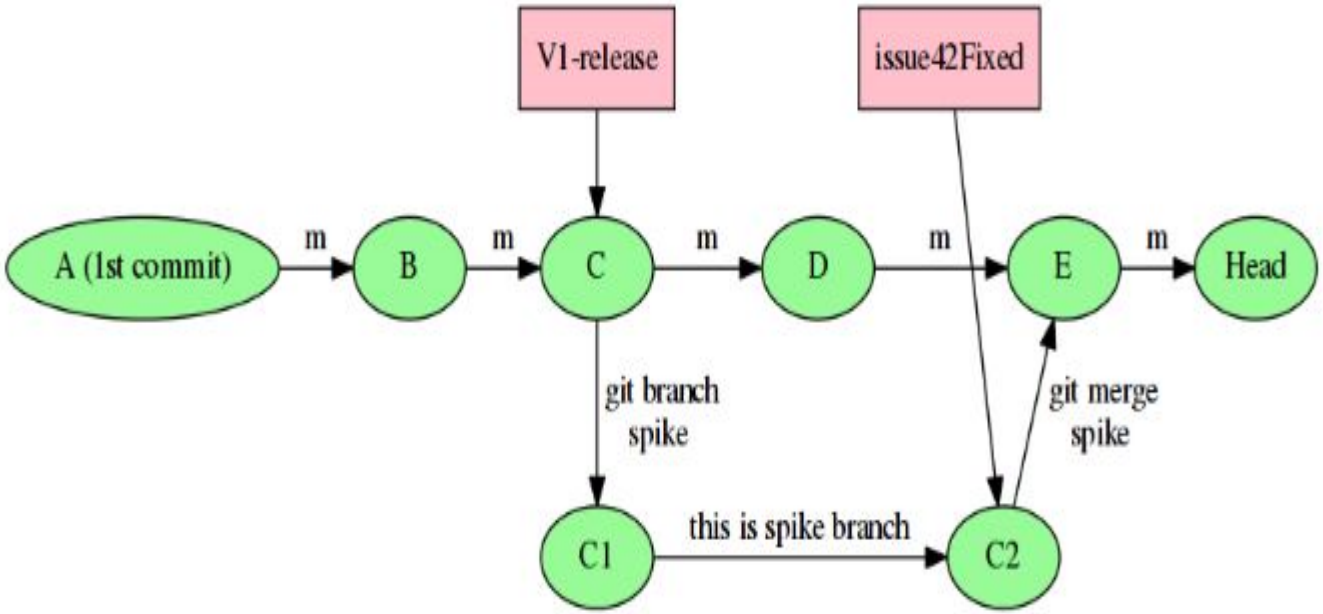
```
git-example $git show v3
tag v3
Tagger: Amjed Tahir <amjedtahir@gmail.com>
Date:   Mon Jul 24 10:29:03 2023 +1200

new tag

commit 2907e4bf7a273338869ee6f9b424f1a32926c055 (HEAD -> Feature, tag: v3, tag:
v2)
Author: Amjed Tahir <amjedtahir@gmail.com>
Date:   Mon Jul 24 10:25:29 2023 +1200

    added files to new branch

diff --git a/file3.java b/file3.java
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>
```

- Two important *merge* commands:
  - if merge results in conflicts (e.g., two branches are committing the same file), then use git merge --abort to abort the merge process and try to reconstruct the pre-merge state commit.
  - After a git merge stops due to conflicts you can conclude the merge by running git merge --continue

Merging

- Merge types, git provides two types of merging
  - fast-forward merging : when there is a linear path from the current branch.
  - 3-way merging: when the branches have diverged. Create a new commit using common ancestor

```
git-example $git checkout main
Switched to branch 'main'
git-example $git merge Feature
Updating 4d91b44..2907e4b
Fast-forward
 file3.java | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file3.java
```

- Merge conflict
  - if branches have changes in the same part of a file. Git adds visual markers in the file which can be manually resolved before commiting.

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>
```

  - Repository: can be located on the local filesystem or on a remote location (accessible via HTTP or SSH)

```
repos $git clone https://github.com/jhy/jsoup.git
Cloning into 'jsoup'...
remote: Enumerating objects: 21638, done.
remote: Counting objects: 100% (3052/3052), done.
remote: Compressing objects: 100% (225/225), done.
remote: Total 21638 (delta 2906), reused 2868 (delta 2805), pack-reused 18586
Receiving objects: 100% (21638/21638), 4.75 MiB | 1.27 MiB/s, done.
Resolving deltas: 100% (9865/9865), done.
```

- Cloning depends on the size of the files and your connection as well

cloning from another repository

- using the command of **git clone** will create a copy of the repository in the local machine (aka the local working copy)
- makes a local copy of another repository
- 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
  - first navigate to the repository local folder
  - example: clone source code of apache ant project

```
git clone https://github.com/jhy/jsoup.git
```

```
repos $git clone https://github.com/jhy/jsoup.git
Cloning into 'jsoup'...
remote: Enumerating objects: 21638, done.
remote: Counting objects: 100% (3052/3052), done.
remote: Compressing objects: 100% (225/225), done.
remote: Total 21638 (delta 2906), reused 2868 (delta 2805), pack-reused 18586
Receiving objects: 100% (21638/21638), 4.75 MiB | 1.27 MiB/s, done.
Resolving deltas: 100% (9865/9865), done.
```

- 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
    - now supports private repositories with up to three collaborators
  - 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 (personal favourite!)
    - git-cola: Linux, Windows and Mac
    - GitKranken: Linux, Windows and Mac
- Use git client functionality in IDEs (e.g., VS Code, IntelliJ)

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
  - Guidance on contributing to projects using git (workflows, writing good commit messages .. etc)

    https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project