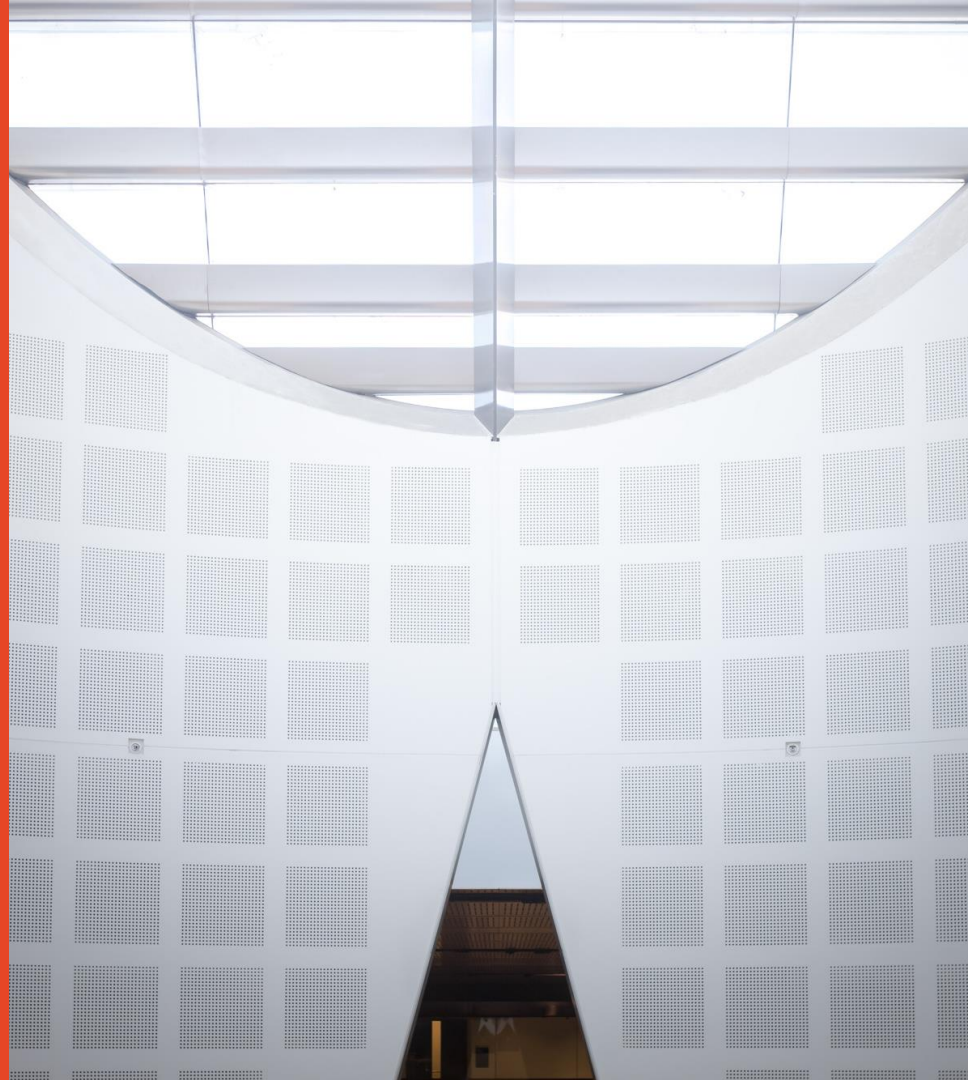


Agile Software Development Practices SOF2412 / COMP9412

Tools and Technologies for
Controlling Artefacts

Dr. Basem Suleiman

School of Information Technologies



Agenda

- Software Development – Artefacts
- Software Configuration Management
- Version Control Systems
- Version Control with Git

Software Development Activities

- Whilst the sequencing is quite different, the essential activities within processes are similar...
- How many primary software project activities can you identify?
 - Contract negotiation Requirements elicitation
 - Scoping Analysis
 - Design Data modelling
 - Coding Integration
 - Testing (at various levels) Validation
 - Documentation Delivery
 - Training Maintenance (and issue tracking)
 - Refactoring ...

Activities are supported by tools

Different tools suit different activities

- Requirements elicitation IBM Rational DOORS
- Scoping SCOPE (function point analysis)
- Analysis iRise
- Design Rhapsody
- Data modelling Oracle SQL Developer
- Coding Eclipse
- Integration Bamboo
- Testing (at various levels) JUnit
- Validation AVAT
- Documentation Doxygen
- Delivery RPM
- Training ??
- Maintenance / Issues Atlassian JIRA
- Refactoring Klocwork
- ...

Artefacts

- Any software development project produces items that represent work done, in ways that others can use
 - Code is one prominent example of an artefact, requirements specifications
 - Some are text, others are diagrams (models, designs)
- As an artefact, or collection of artefacts, is being developed, we can see an **evolution**, or ongoing sequence of changed states
- How much impact if these are lost or changes are lost or not tracked?
- How much effort was it to create them?
- The artefacts have **value** and need to be preserved, communicated, maintained, protected from unauthorized access, etc.
 - High value → need for **management** (storage, disaster recovery, access control, query/search capacity?)

Requirements Artefact

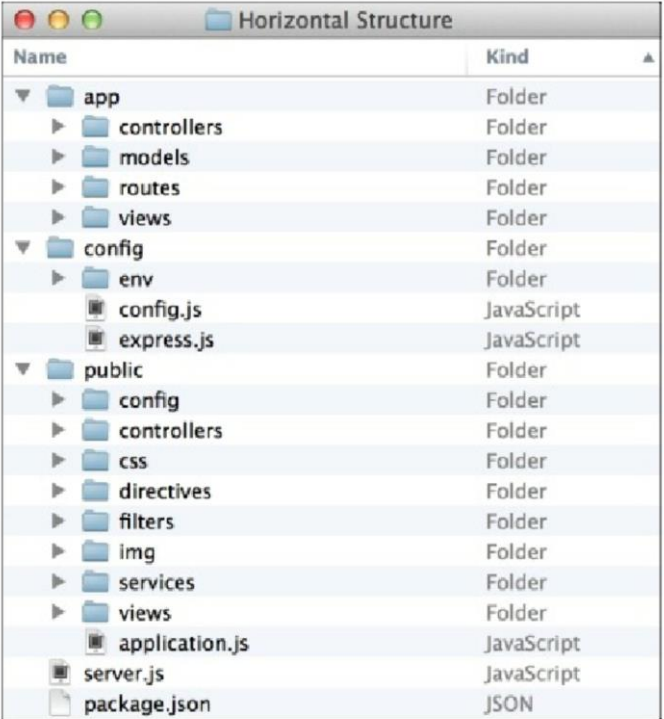
- Eliciting these from users was a lot of effort
- Different formats in various process methodologies
 - In agile, often as “User stories” (plain text, stylized format, can be manipulated by tools)
 - In traditional methodologies, often huge Word documents in a standard template
 - Software Requirements Specification (SRS)
 - Signed off as obligation on the developers
- Track changes and versions of requirements is very important

Code

- The source code for a project is typically spread over different files, and perhaps in a directory structure
- Language conventions or requirements
 - Eg Java has a file for each class
 - Package hierarchy: namespace matches directory structure
- Documentation may be derived from the source code
- Maintainability and extensibility are among the key non-functional properties for a software
 - Design and architecture play important role
 - E.g., Model-View-Controller
 - Layered architecture

Artefacts – Source Code

- Example of source directory structure
 - Web application code architecture (Node/Express.js)
 - Model-View-Architecture (MVC) architecture



The screenshot shows a file explorer window titled "Horizontal Structure". It displays a directory tree with the following structure:

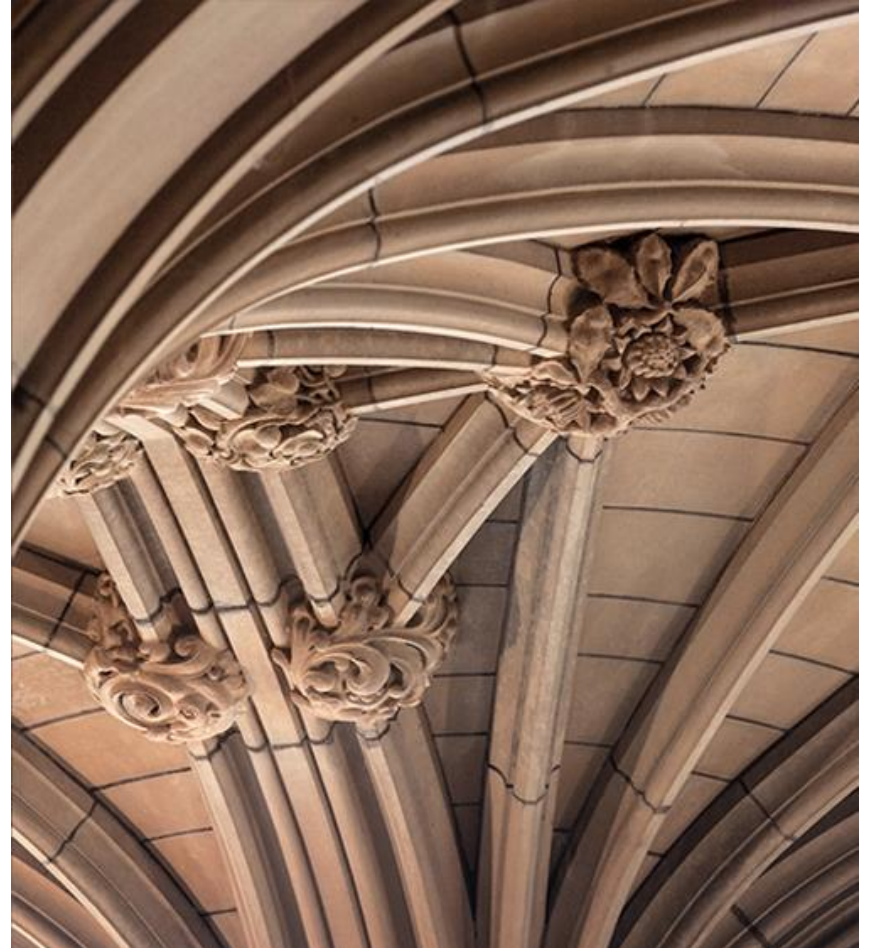
Name	Kind
app	Folder
controllers	Folder
models	Folder
routes	Folder
views	Folder
config	Folder
env	Folder
config.js	JavaScript
express.js	JavaScript
public	Folder
config	Folder
controllers	Folder
css	Folder
directives	Folder
filters	Folder
img	Folder
services	Folder
views	Folder
application.js	JavaScript
server.js	JavaScript
package.json	JSON

Artefacts – Dependencies

- A project has multiple artefacts, and some can depend on the state of others
 - Eg A source file calls a method from another file
 - Eg A source file uses declarations from a header file
 - Eg A test is used to see if a requirement is met
 - Eg a source file fixes a bug reported in an incident
 - Eg A piece of documentation refers to a set of source files
 - Eg A test was run in a particular environment configuration
- Great problems occur if artefacts are not in consistent state
 - Eg modify one file but not the others

Software Configuration Management

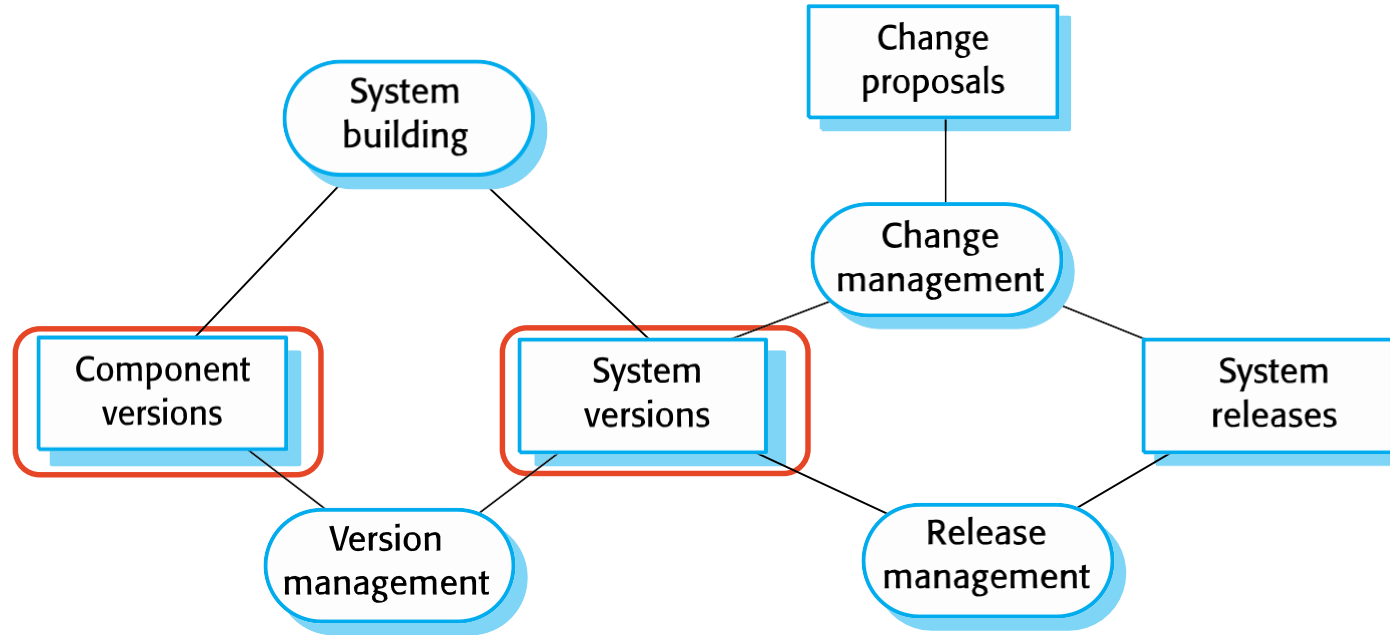
Software version and version management



Configuration Management (CM)

- Software systems are constantly changing during development and use
- Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems
- You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- CM is essential for team projects to control changes made by different developers

Configuration Management Activities



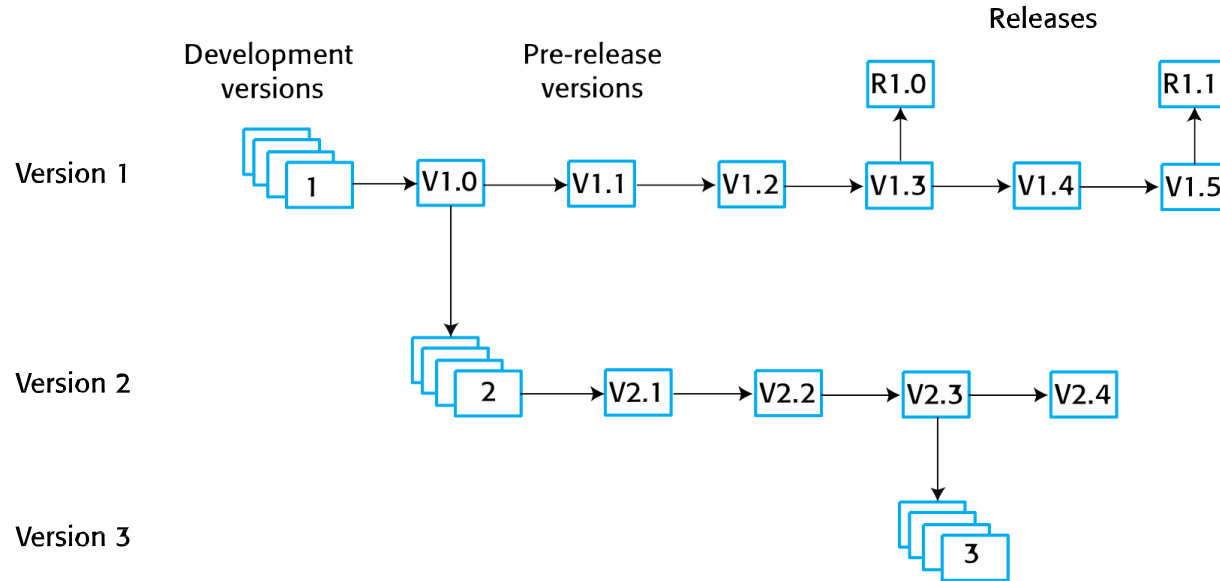
Configuration Management Activities

- **Version management:** Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- **System building:** the process of assembling program components, data and libraries, then compiling these to create an executable system.
- **Change management:** keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- **Release management:** preparing software for external release and keeping track of the system versions that have been released for customer use.

Multi-version Systems

- For large systems, there is never just one ‘working’ version of a system.
- There are always several versions of the system at different stages of development.
- There may be several teams involved in the development of different system versions.

Multi-version System Development



CM Related Terminology

Term	Explanation
Baseline	A collection of component versions that make up a system. Baselines are controlled; i.e., the versions of the components making up the system cannot be changed. It is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A set of versions of a software component and other configuration items on which that component depends.
Mainline	A sequence of baselines representing different versions of a system.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Version Management (VM)

- The process of keeping track of different versions of software components or configuration items and the systems in which these components are used
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other
- Therefore version management can be thought of as the process of managing codelines and baselines.

Version Management (VM)

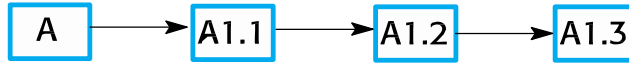
- The process of keeping track of different versions of software components or configuration items and the systems in which these components are used
 - It also involves ensuring that changes made by different developers to these versions do not interfere with each other
 - VM can be thought of as the process of managing codelines and baselines
- **Codeline:** a sequence of versions of source code with later versions in the sequence derived from earlier versions
 - Normally apply to components of systems so that there are different versions of each component
- **Baseline:** a definition of a specific system.
 - Specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

Baselines

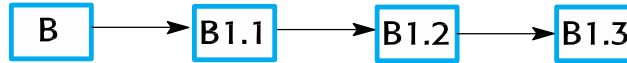
- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- Baselines are important because you often have to recreate a specific version of a complete system
 - For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired

Baselines

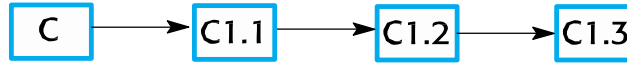
Codeline (A)



Codeline (B)



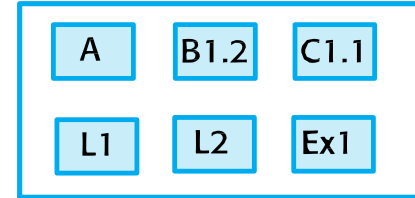
Codeline (C)



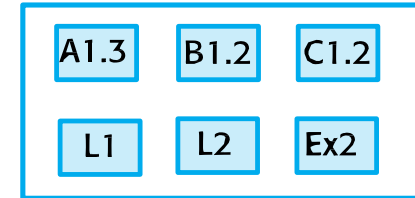
Libraries and external components



Baseline - V1



Baseline - V2



Mainline

Semantic Versioning (SemVer)

- Set of rules and requirements that determine how version numbers should be assigned and incremented for software being developed
 - Semantic numbers; numbers with meaning in relation to a certain version
- Why SemVer?
 - Helps managing versioning numbers in a meaningful and standard way
 - Managing dependencies: the bigger your system grows and the more packages/libraries/plugins you integrate into your software
- Given a version number **MAJOR.MINOR.PATCH**, increment the:
 1. MAJOR version when you make incompatible API changes,
 2. MINOR version when you add functionality in a backwards-compatible manner,
 3. PATCH version when you make backwards-compatible bug fixes.
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

SemVer Example

Stage	Code	Rule	Example
New product	1 st release	Start with 1.0.0	1.0.0
Patch Release	Bug fixes, other minor changes	Increment the 3 rd digit	1.0.1
Minor Release	New features that do not break existing features	Increment the 2 nd digit	1.1.0
Major Release	Changes that break backward compatibility	Increment the 1 st digit	2.0.0

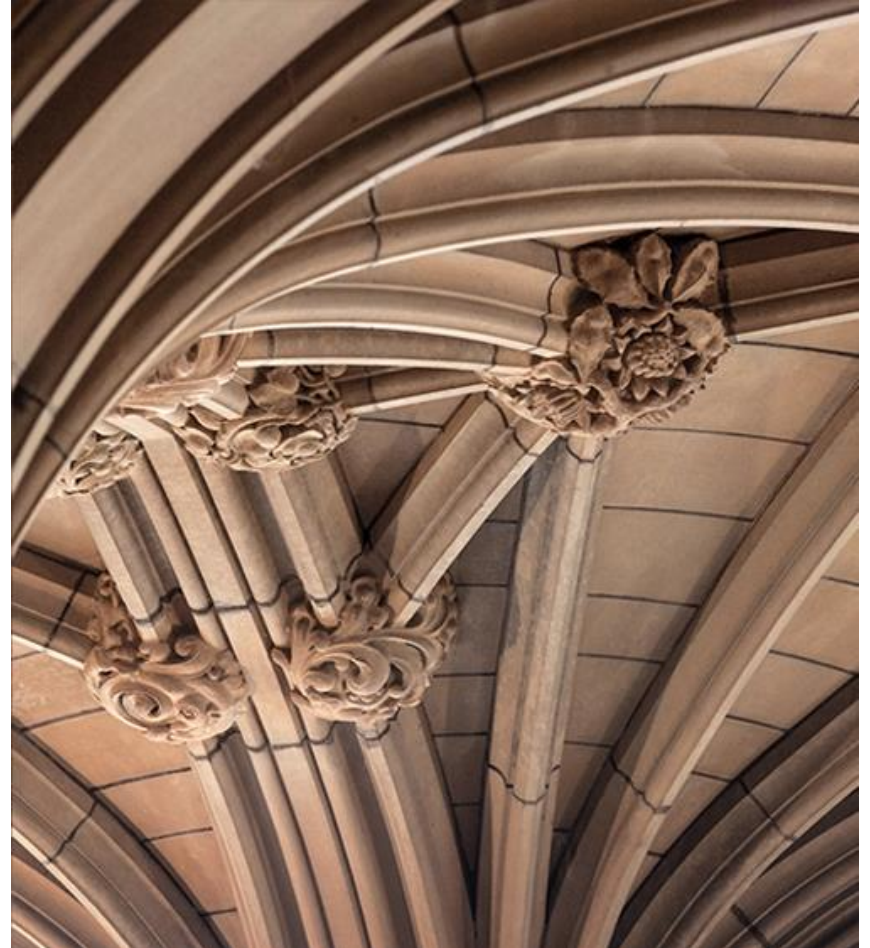
<https://docs.npmjs.com/getting-started/semantic-versioning>

SemVer Example – Dependencies

- Consider a library called “**Firetruck**.” It requires a Semantically Versioned package named “**Ladder**.”
- At the time that **Firetruck** is created, **Ladder** is at version 3.1.0.
- Since **Firetruck** uses some functionality that was first introduced in 3.1.0, you can safely specify the **Ladder** dependency as greater than or equal to 3.1.0 but less than 4.0.0.
- Now, when **Ladder** version 3.1.1 and 3.2.0 become available, you can release them to your package management system and know that they will be compatible with existing dependent software.

Version Control

History of Version Control Systems (VCSs)



History of VCSs

- Version control systems(VCS) have a long history
 - SCCS done at Bell Labs in 1972
 - Dominant in early Unix systems
 - Many alternatives explored in mid 1980s
 - At same time as related ideas for eg CAD systems
 - Subversion (svn) around 2000 was very popular
 - Still used in many places
 - Now git, mercurial (hg) are the main ones
 - These are “Distributed VCS”
 - Microsoft offers Team Foundation Version Control (TFVC)

What is Version Control?

- A method for recording changes to a file or set of files over time so that you can recall specific versions later
 - Also known as revision control and source control
 - Source code is the most important intellect to be created in SW development
 - Files → software source code files
 - Create, maintain and track history of changes during the SDLC for all artefacts

What is Version Control System?

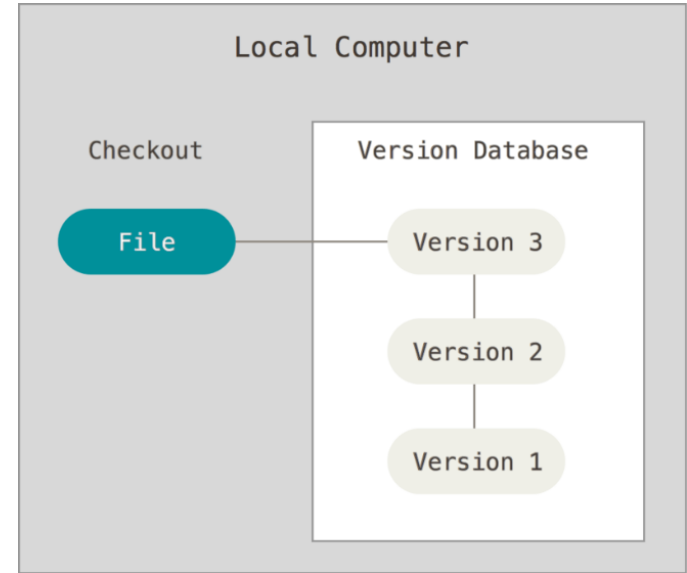
- Version Control System (VCS)
 - Category of software tools that help software teams to manage changes to source code over time
 - It allows developers to keep track of every modification to the code in a special kind of storage (repository)
 - It allows developers to:
 - Revert selected files (or the entire project) back to a previous state
 - Compare changes over time
 - See who last modified something that might be causing a problem
 - Who introduced an issue and when
 - compare earlier versions of the code to help fix mistakes while minimizing disruption to all team members
 - And more

Version Control – SW Development Scenarios

- Multiple versions of the same software deployed in different sites and SW developers working simultaneously on updates
- Developers fixing some bugs/issues may introduce some others as the program develops
 - Bugs or features of the SW often only present in certain versions
- Two versions of the software may be developed concurrently
 - One version has bugs fixed, but no new features
 - While the other version is where new features are worked on
- And many more

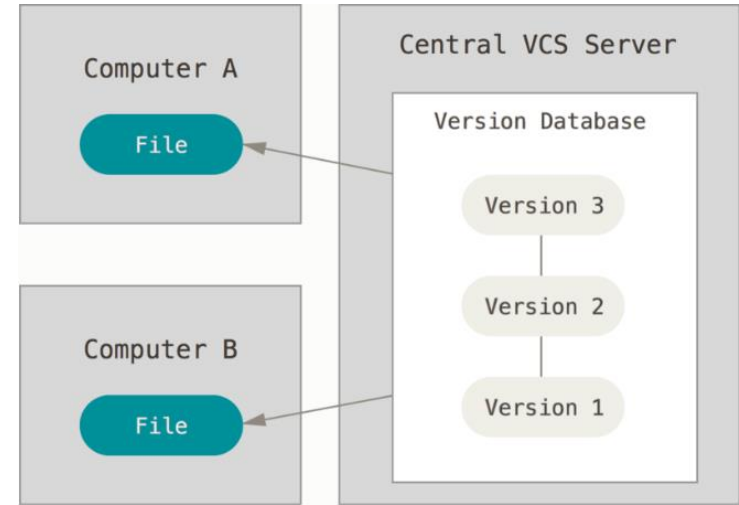
Local Version Control

- Programmers long ago developed local VCSs; a simple database that kept all the changes to files under revision control
- Popular example of such VCS tools is RCS - which is still distributed with many computers today
- RCS works by keeping patch sets (i.e., the differences between files) in a special format on disk it can then re-create what any file looked like at any point in time by adding up all the patches



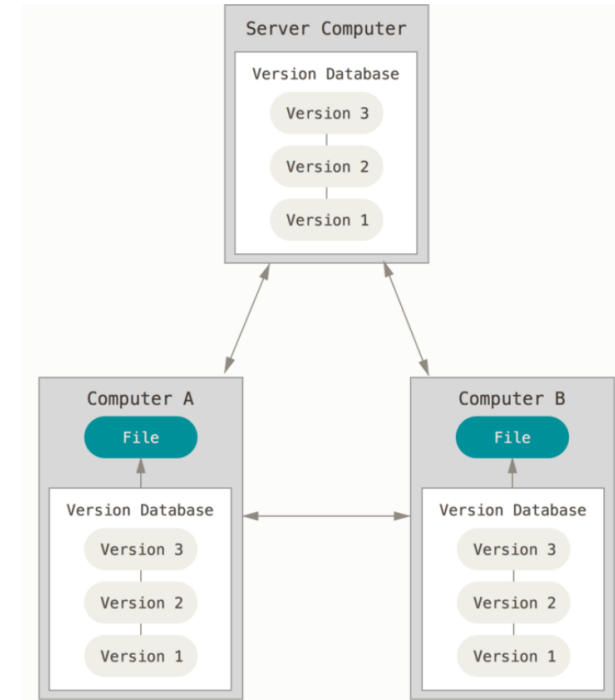
Centralized Version Control (CVC)

- CVCSs (e.g., CVS, subversion) developed to allow developers to collaborate
- It has a single server that contains all versioned files and a number of clients that check-out files from the central server
- Better than local VCS
 - Everyone knows to a certain degree what everyone else on the project is doing
 - Easier admin - fine-grained control over who can do what (rather than dealing with local database)
- Single point of failure – central server is down!
 - Developer's work interrupted!
 - Hard disk becomes corrupted, and no proper/up-to-date backups? entire history lost!



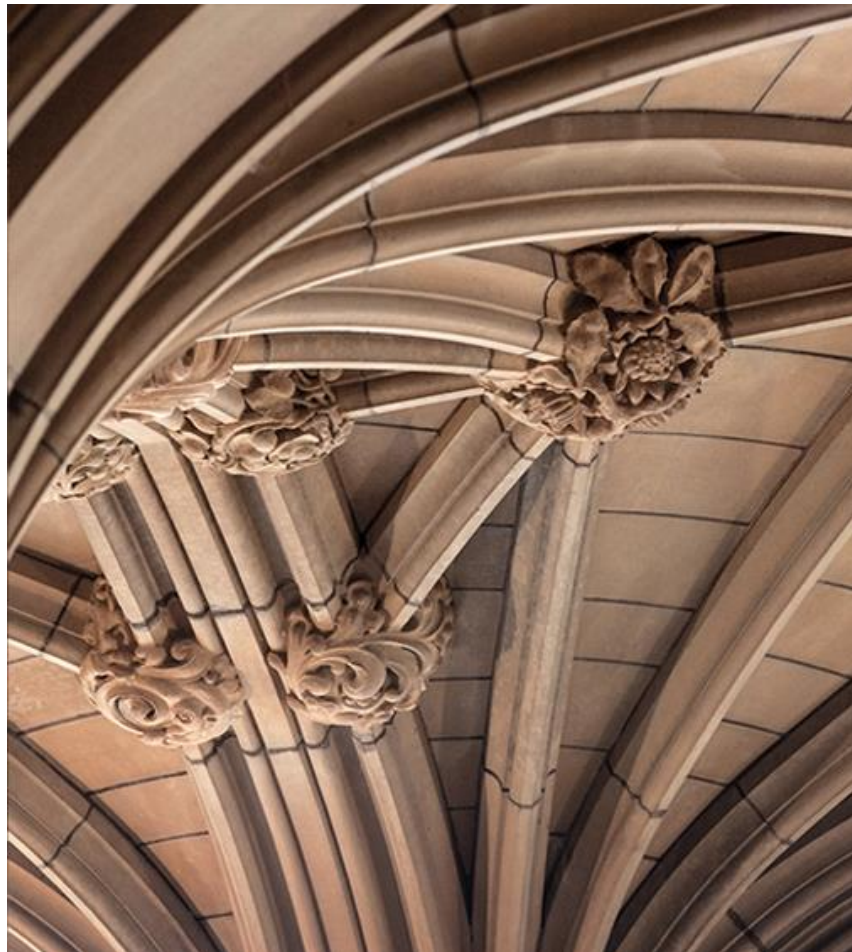
Distributed Version Control (DVC)

- In DVCSs developers fully mirror the repository including the full history
 - Every clone is a full backup of all the data, if a server dies any of the client repositories can be copied back to the server
- Several remote repositories
 - Developers can collaborate with different groups of people in different ways simultaneously with the same project
 - Can setup several types of workflows (not possible in CVC)



Git Fundamentals

Based on Pro Git Pro book – see references

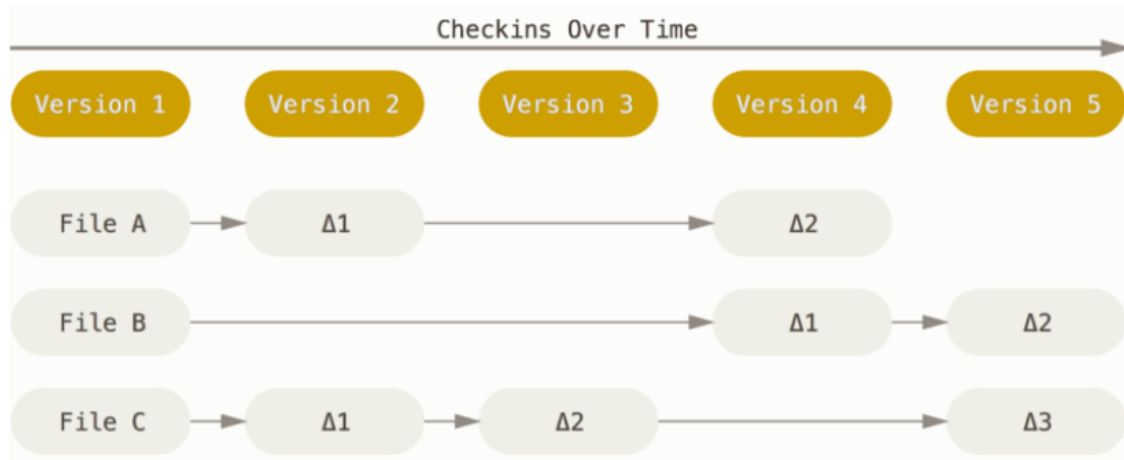


Git

- A version control system that helps development teams to manage changes to source code overtime
 - Most widely used modern VCS
- Web-based (online) central repository of code and track of changes
 - Tracing history of changes, commits, branches, merges, conflict resolution,
 - Collaborate and update repository through command-line and GUI
- Public and private repositories

Delta-based VCSs (Differences)

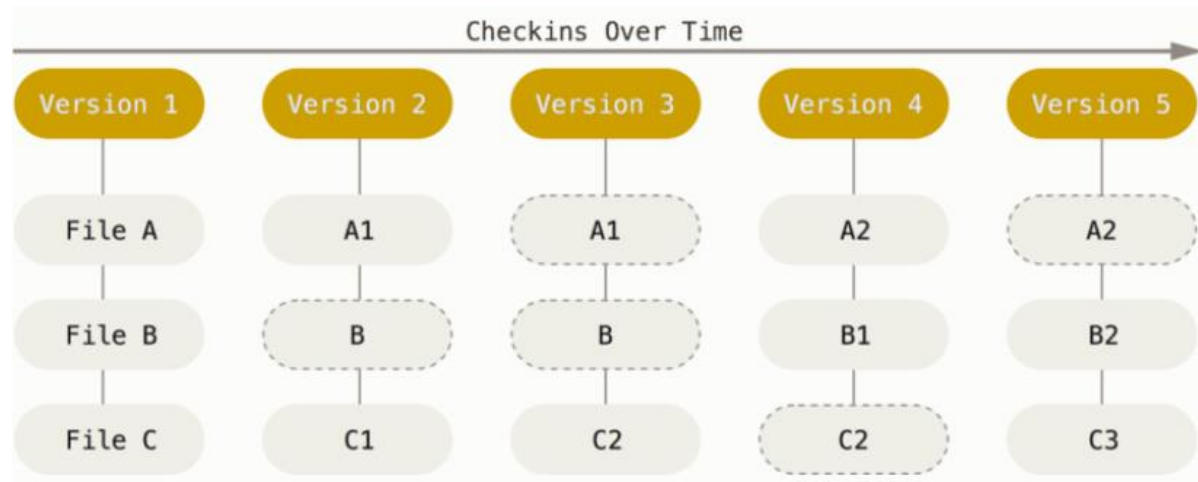
- VCSs that store as a set of files and the changes made to each file over time
 - Example: CVS and subversion



- Git stores and thinks about information in a very different way (not as differences)

Git – Snapshots Not Differences

- Git thinks about its data as a *streams of snapshots* of a small file system
- Every time a developer commits (or save state of the project) Git takes a picture of what all files look like at that moment and stores references to that snapshot
 - Git doesn't store unchanged files, it just link to previous identical file already stored



Git – Basics (1)

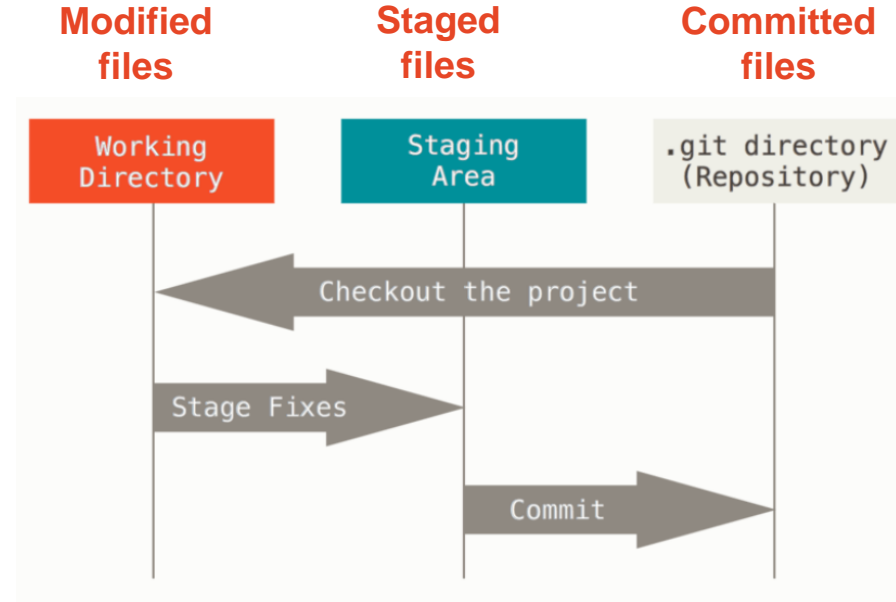
- **Nearly Every operation is local**
 - Most operations in Git need only local files and resources to operate
 - E.g., to browse the project history, Git reads it directly from your local database — it doesn't get the history from the server to display it
 - Developers can work and commit their changes to their local copies offline (on train)
- **Git has built-in Integrity**
 - Everything in Git is check-summed before it is stored and is then referred to by that checksum
 - You can't lose information in transit or get file corruption without Git being able to detect it using SHA-1 hash check-sum
 - Git stores everything in its database by the hash value of its contents (not the file name)

Git – Basics (2)

- **Git generally only adds data**
 - Git allows to undo things; after committing a snapshot, it is very difficult to lose, especially if you regularly push your database to another repository
- **Git has three states that your files can reside in**
 - *Committed*: file is safely stored in your local database
 - *Modified*: file has been changed but not committed it to the local database
 - *Staged*: a *modified file has been* marked in its current version to go into next commit snapshot

Git – Basics (3)

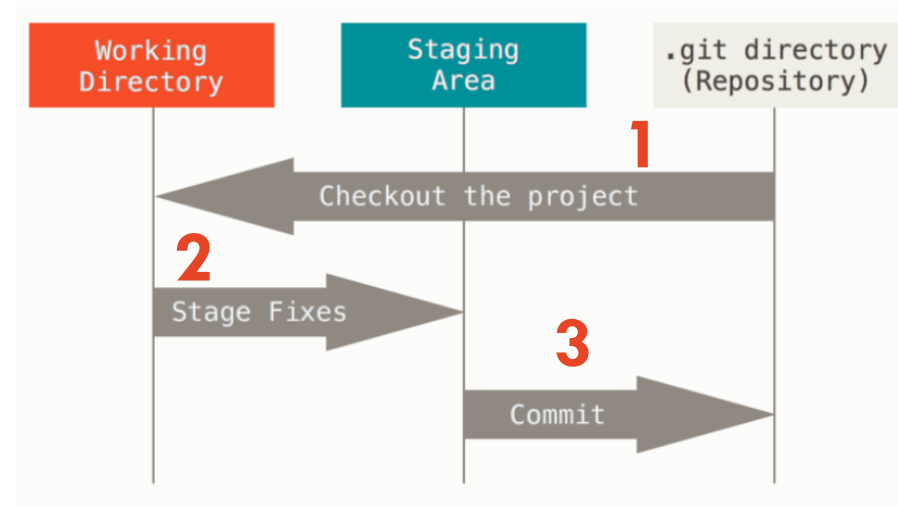
- **Git directory (repository)**
 - Where Git stores the metadata and object database for your project
 - What is copied when you *clone* a repository from another computer
- **Working directory (tree)**
 - A single checkout of one version of the project
 - These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify
- **Staging area (index)**
 - a file, generally contained in your Git directory, that stores information about what will go into your next commit



Git – Basics (4)

Basic Git workflow

1. After check-out a project from .git, you modify files in your working directory
2. You selectively stage just those changes you want to be part of your next commit, which adds *only* those changes to the staging area
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to the Git directory



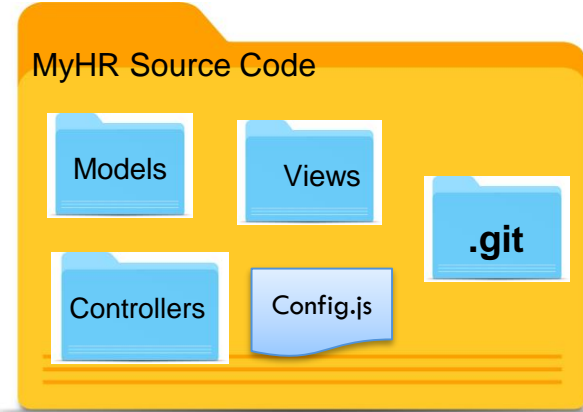
Git Concepts and Scenarios

Git Commands and Operations



Git Repository (Repo)

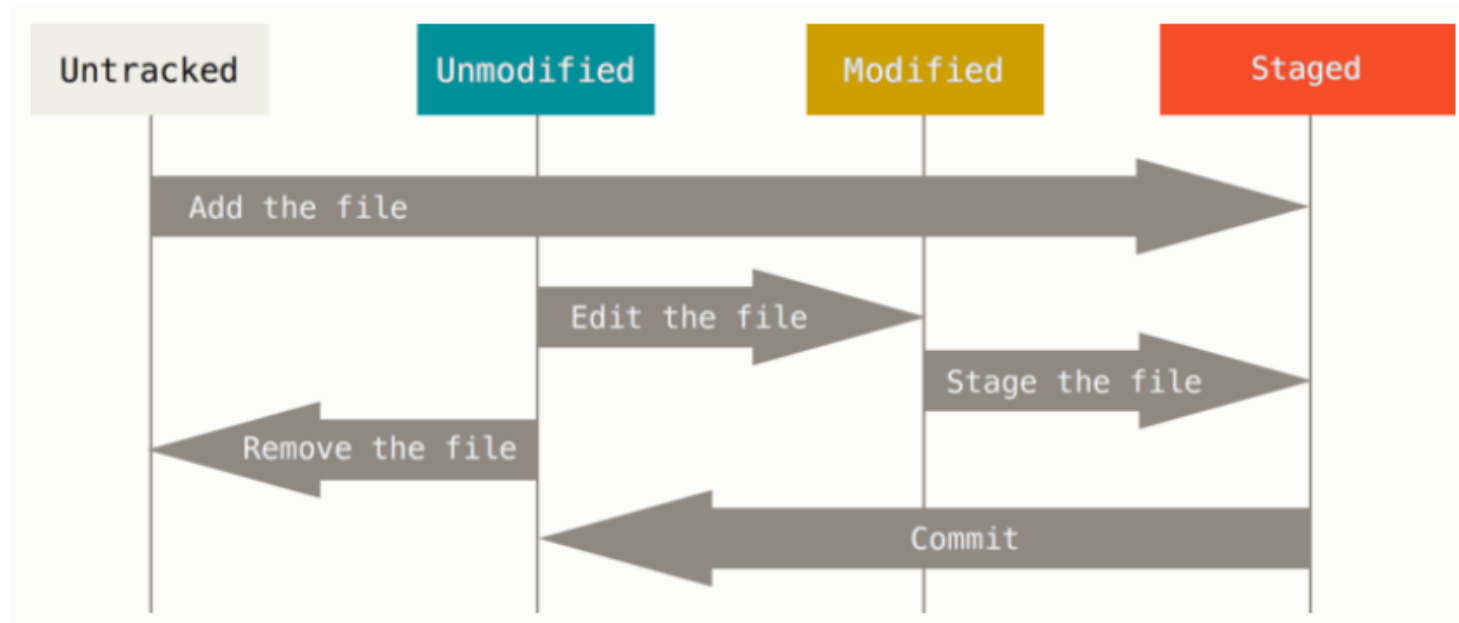
- A special directory that contains project files (and sub-directories)
 - A place where Git stores and tracks files (source code)
 - Can be created or cloned
 - Once created/cloned, Git adds special sub-directory to store all information (**history of changes**) about the project's files and directories
- Creating a Git repo on your local machine
 - Take a local directory that is currently not under version control, and turn it into a Git repository
 - Git operation “*init*” → will create **.git**
 - Clone an existing Git repository from elsewhere
 - Git operation “*clone*”
 - a full copy of nearly all data that the server has



Metadata

- Each version should have a unique name to refer to it
 - The current, latest version is often called the head
- Each version should have a date
- Each version should have an author
- How might you use metadata?
 - Access a version by name
 - Find author for explanations, checking, blaming
 - Access the version that was used elsewhere

Git – Recording changes to a Repo



Git – Recording changes to a Repo

- Each file in the project (working) can be in one of two states:
 - *Tracked*: Git knows about it (unmodified, modified, or staged)
 - *Untracked*: Git doesn't know about it; neither in the last snapshot nor in the staging area
- When a repo is cloned (and nothing has been changed) then all files are tracked and unmodified (Git just checked them out)
- When you edit files, Git denotes them as modified (changed since last commit)
 - When you stage these modified files and commit all those staged changes, you have a clean directory
- Git has operations to check the *status* of files, track new files, adding and removing files to/from staging area, commit changes, view commit history, undoing things

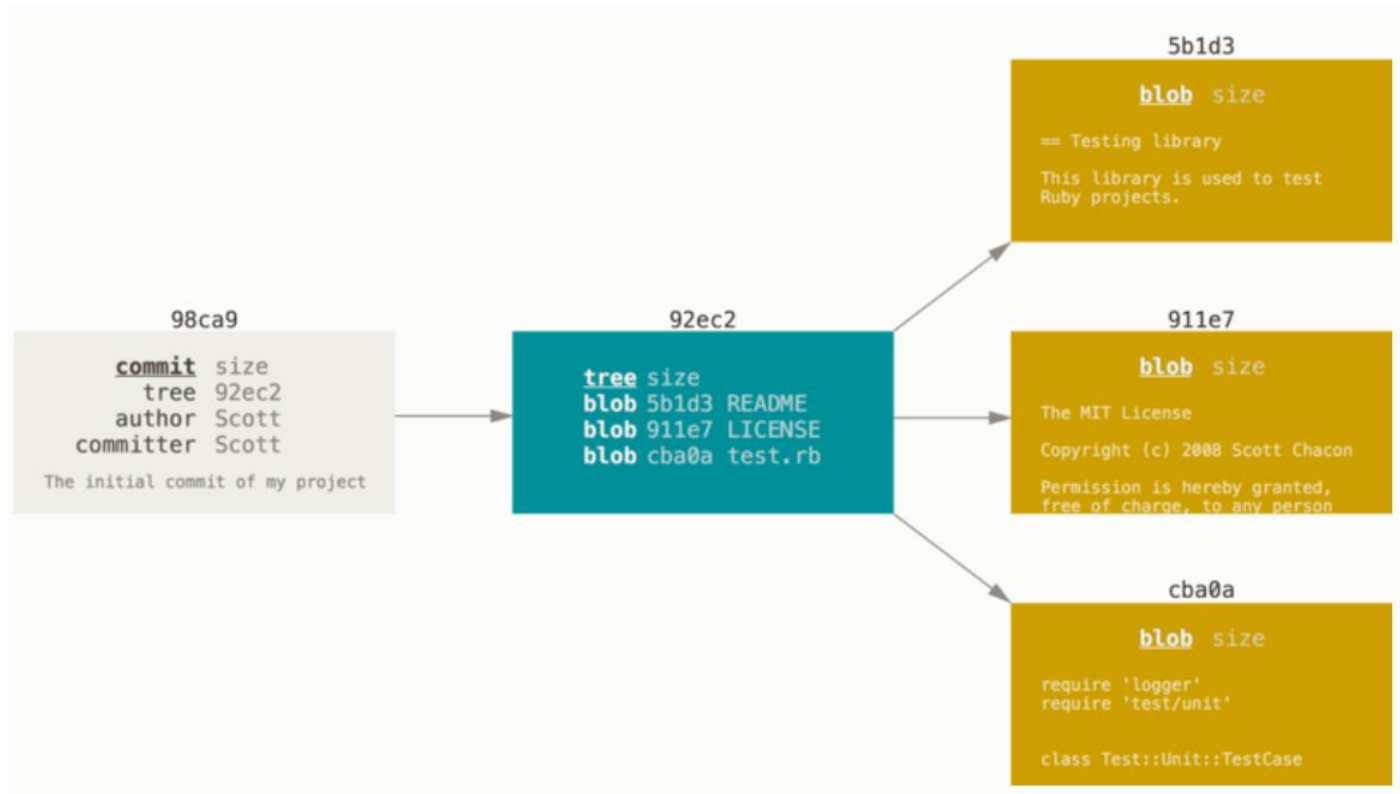
Git – Branching

- Diverging from the main line of development and continue to do work without messing with that main line
 - Expensive process; often requires you to create a new copy of your source code directory, which can take a long time for large projects
- Git’s “*killer feature*” as Git branching is lightweight (nearly instantons)
 - With every commit, Git stores a *commit object* that contains
 - A pointer to the staged snapshot, author’s name and email, commit message, and commit/commits before this commit parent/parents commits

Git – Branching Example (1)

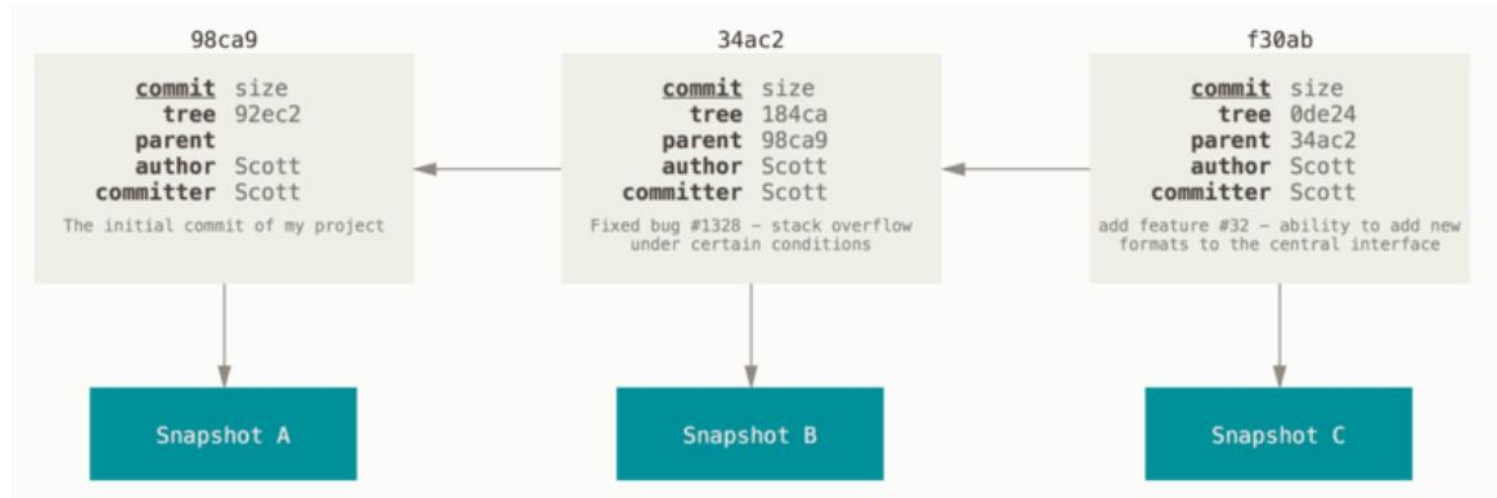
- Assume your project directory contains 3; all of them staged and committed
- When you stage all files, Git computes a checksum of each (SHA-1 hash) and stores the files in the Git repo (as blobs)
- When you commit all files, git checksums each sub-directory (root project dir in this case) and stores those tree objects in the Git repo
- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed
- Our project directory now contains five objects;
 - Three blobs (contents of one of the 3 files)
 - One tree lists the directory contents and file names as blobs
 - One commit with the pointer to the root and the commit metadata

Git – Branching Example (2)



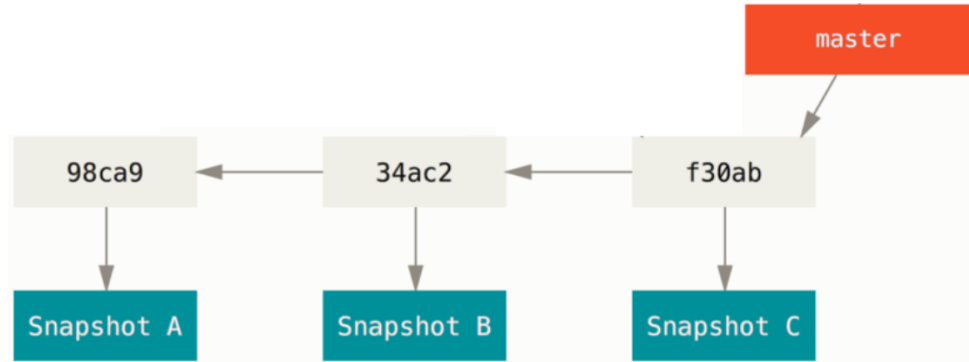
Git – Branching Example (3)

- If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



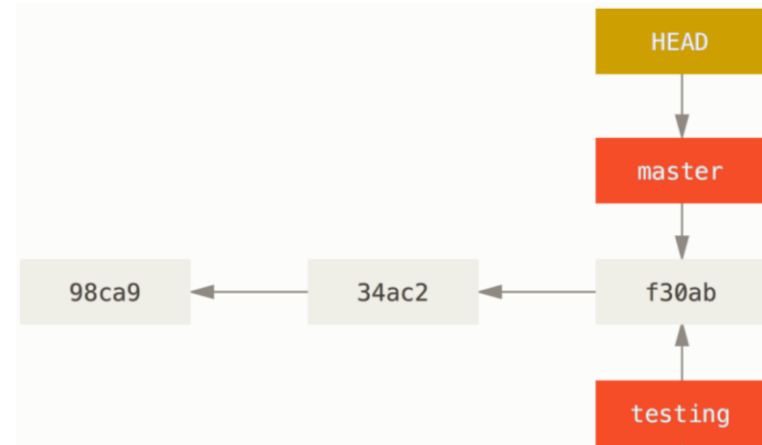
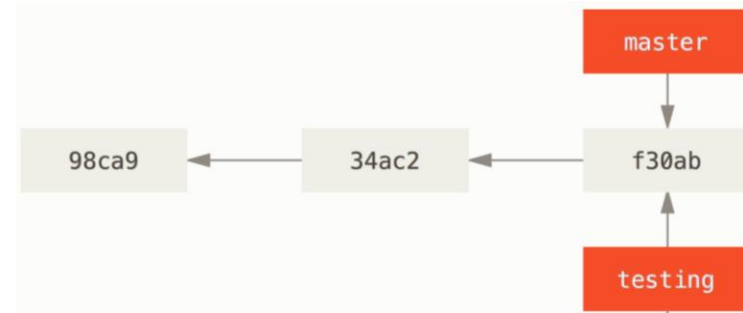
Git – Branching (pointer's perspective)

- A branch in Git is like a *lightweight movable pointer* to one of these commits
- The default branch name in Git is “*master*”
 - Every Git repo has it as it's created by default when a directory is initiated with Git “init”
- As you start making commits, you're given a master branch that points to the *last commit* you made
- Every time you commit, the *master branch pointer* moves forward automatically



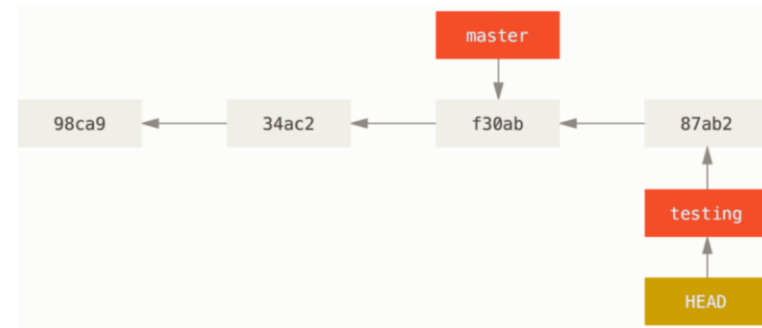
Git – Creating a New Branch

- When you create a new branch, a new pointer will be created – pointing to the same commit we are currently at
- Example: create a new branch called testing in our project
- Git knows which branch is the current by maintaining a special pointer called “HEAD”
 - Pointer to the local branch you’re currently on (master)
 - Creating a branch in Git does not switch the HEAD to the new branch



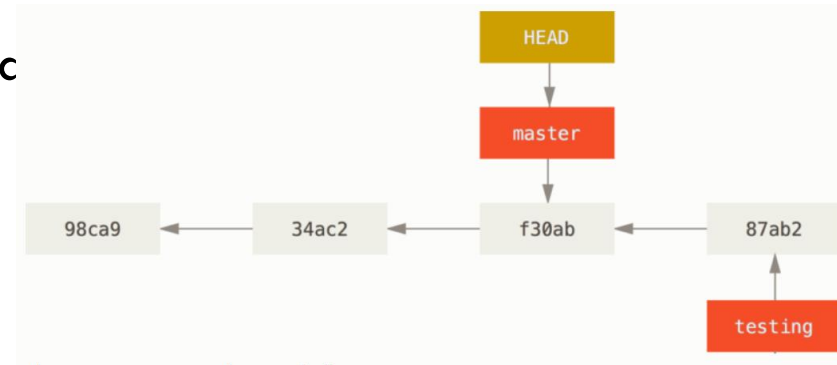
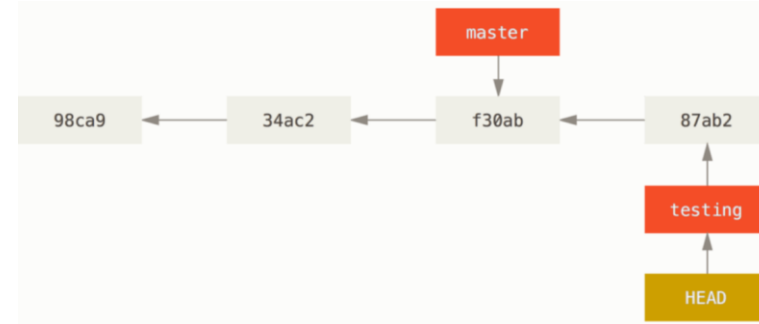
Git – Switching to another Branch (1)

- You can switch to an existing branch using Git *checkout* operation
- How would this impact our repo – let's do another commit
 - The testing branch has moved forward, but the master branch still points to the commit you were on when you ran git checkout to switch branches



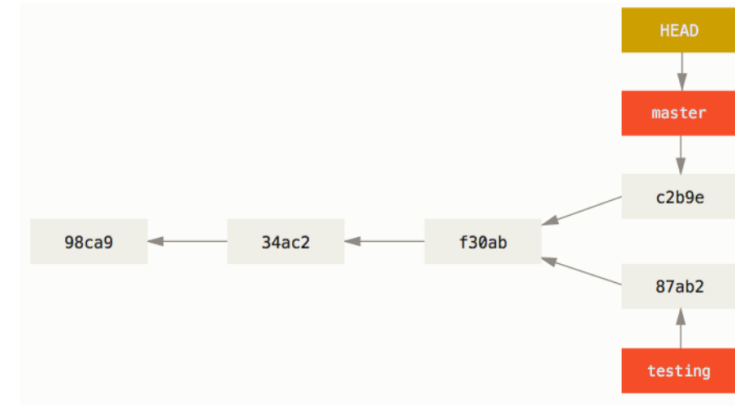
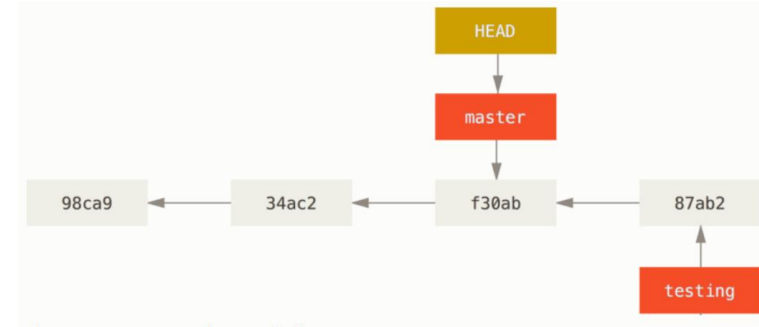
Git – Switching to another Branch (2)

- What happens if we switch back to the master branch (using checkout)?
- Two things:
 - The HEAD pointer moved back to the master branch
 - The files in our working directory reverted back to the snapshot that master points to
- Any changes we make from this point forward will diverge from an older version of the project
 - You can branch into another direction (not from testing branch)



Git – Switching to another Branch (3)

- While the pointer is on the master, let's edit one file and commit – What happens?
- Both of those changes are isolated in separate branches
 - You can switch back and forth between the branches and merge them together when you're ready
- You can see the history of all commits at any point of time

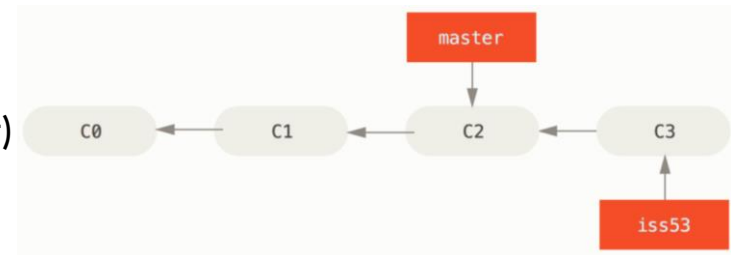
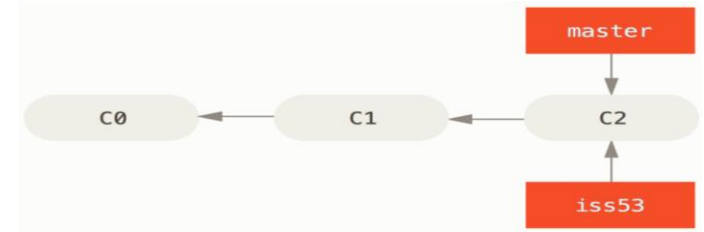
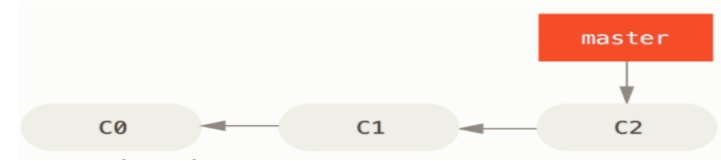


Git – Merging Scenario (1)

- Consider the following real-world scenario
 - Do some development on a website
 - Create a branch for a new user story you're working on
 - Do some development in that branch
- At this stage, assume you receive a call that another issue is critical and you need a hotfix. You'll do the following:
 - Switch to your production branch
 - Create a branch to add the hotfix
 - After it's tested, merge the hotfix branch, and push to production
 - Switch back to your original story and continue working

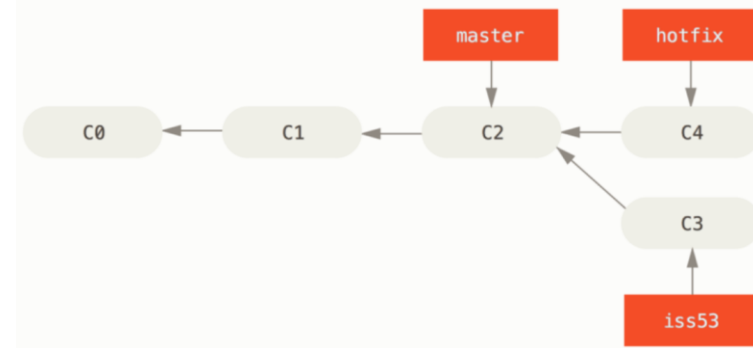
Git – Merging Scenario (2)

- Assume you're working on your project and have a couple of commits already on the master branch
- To work on the issue (say *iss53*) you need to create a new branch and switch to it at the same time (using *git branch* and *checkout* operations)
- You work on your website and do some commits (checked it out)
 - This moves the *iss53* branch forward (HEAD point to it)



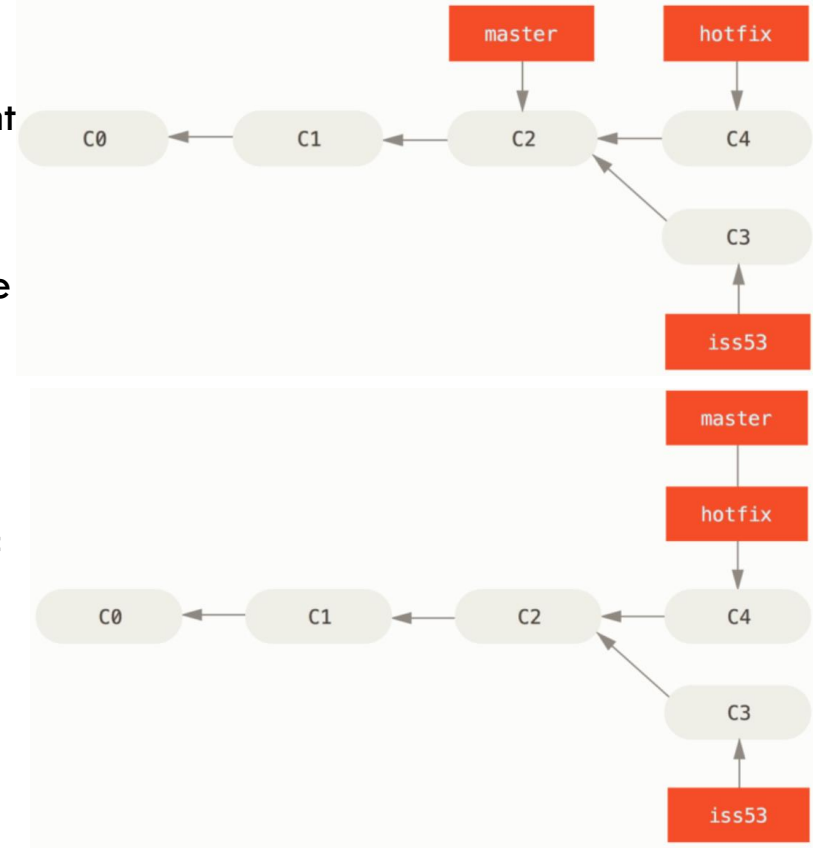
Git – Merging Scenario (3)

- Imagine you receive a call for urgent issue in the website that need immediate fix
- How would you deal with this scenario?
 - Deploy your fix along with the iss53 changes you've made?
 - Revert those changes before you can work on applying your fix to what's in production?
- You can switch back to the master branch
 - But you need to have clean working state before switch branches; i.e., working directory doesn't have uncommitted changes



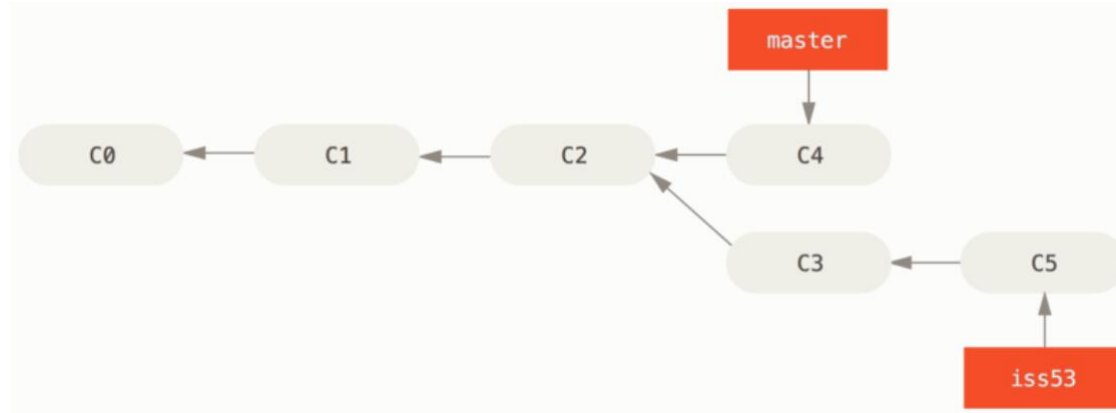
Git – Merging Scenario (4)

- Ready to create a new branch for fixing the urgent issue, let's call 'hotfix'
- Work on the 'hotfix' and make tests and merge the hotfix branch back into your master branch to deploy to production
 - Using the git merge operation
- **Fast-forward:** the commit C4 pointed to by the branch hotfix we merged in was directly ahead of the commit C2, Git moves the pointer forward
- Ready to deploy the fix – the change is in the snapshot of the commit pointed to by the master branch



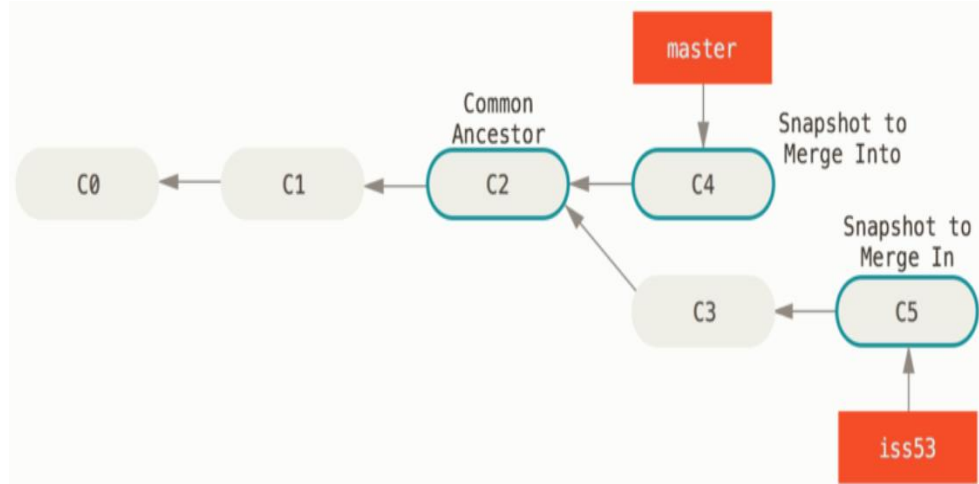
Git – Merging Scenario (5)

- Now you can switch back to the *iss53* work – but first you should delete the hotfix branch (no longer needed)
 - The master branch already points to the same commit
- Switch back to the *iss53* branch and continue working on it, and commit changes you make on *iss53*



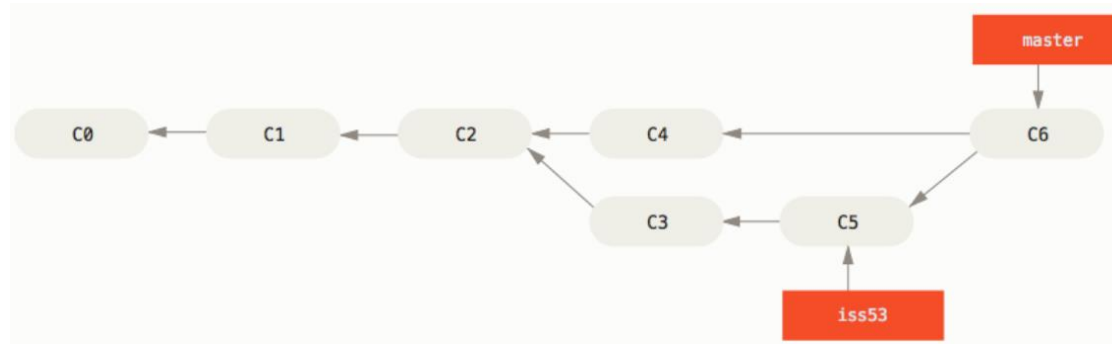
Git – Merging Scenario (6)

- Suppose that issue #53 work is complete and ready to be merged into the master branch
 - Merge the iss53 branch into the master branch
- Suppose that issue #53 work is complete and ready to be merged into the master branch
 - Merge the iss53 branch into the master branch
- Note – here the development history has diverged from some older point (commit on the branch we're on isn't a direct ancestor of the branch we're merging in)
- Instead of moving the branch pointer forward, Git makes “three-way merge” using two snapshots as shown in the figure



Git – Merging Scenario – Three-way Merge (7)

- In the three-way merger, Git creates a new snapshot and automatically creates a new commit that points to it
 - This ‘special’ merge has more than one parent and it’s referred to as “merge commit”



- As the work has been merged in there’s no need for *iss53* branch and the issue can be recorded as fixed, so the branch *iss53* can be deleted

Git – Conflict Resolution (1)

- Commits, branching and merging workflows can get complicated (we discuss happy scenarios)
- For example, a developer may make changes to some part of a file in the *iss53* branch and another developer make changes to the same part of the same file on the *hotfix* branch
 - What happens if we try to merge both *iss53* and *hotfix* branches?
 - This will lead to a **Merge Conflict** and requires **Conflict Resolution**
 - Git will not make automatic merge; it will raise a conflict require human intervention to resolve the conflict manually
 - Git keeps anything that has merge conflicts and hasn't been resolved is listed as unmerged

Git – Conflict Resolution (2)

- **Conflict-resolution markers:** special markers added automatically by Git to the files that have conflicts to guide you where the conflicts

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

- The version in the master branch HEAD (everything above the ===), while the version in iss53 branch everything in the bottom part
- To resolve the conflict, you have to either choose one side or the other or merge and remove those special markers
- After resolving conflicts, add each file to staging area – Git marks it as resolved

Git – Conflict Resolution (3)

- **Graphical format:** visual representation of merges and conflicts (Git opendiff *is the default*)
- Other available tools opendiff, diffuse, diffmerge, codecompare
- When you exit the merge tool, Git asks if the merge was successful – if you confirm that, it stages the file to mark it as resolved and then you can commit the merge

Using Git

Git Commands and Operations



Git – The Command Line vs GUI

- **Command-line tools**

- The only place you can run *all* Git commands
- If you know how to run the command-line version, you can probably also figure out how to run the GUI version

- **Graphical User Interface (GUI)**

- Most of the GUIs implement only a partial subset of Git functionality for simplicity

Git Basics

<code>git init <directory></code>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
<code>git clone <repo></code>	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
<code>git config user.name <name></code>	Define author name to be used for all commits in current repo. Devs commonly use <code>--global</code> flag to set config options for current user.
<code>git add <directory></code>	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
<code>git commit -m "<message>"</code>	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
<code>git status</code>	List which files are staged, unstaged, and untracked.
<code>git log</code>	Display the entire commit history using the default format. For customization see additional options.
<code>git diff</code>	Show unstaged changes between your index and working directory.

See full list of commands at [Git Cheatsheet](#) from Atlassian

Git Log

<code>git log --<limit></code>	Limit number of commits by <limit>. E.g. <code>git log -5</code> will limit to 5 commits.
<code>git log --oneline</code>	Condense each commit to a single line.
<code>git log -p</code>	Display the full diff of each commit.
<code>git log --stat</code>	Include which files were altered and the relative number of lines that were added or deleted from each of them.
<code>git log --author="<pattern>"</code>	Search for commits by a particular author.
<code>git log --grep="<pattern>"</code>	Search for commits with a commit message that matches <pattern>.
<code>git log <since>..<until></code>	Show commits that occur between <since> and <until>. Args can be a commit ID, branch name, HEAD, or any other kind of revision reference.
<code>git log -- <file></code>	Only display commits that have the specified file.
<code>git log --graph --decorate</code>	<code>--graph</code> flag draws a text based graph of commits on left side of commit msgs. <code>--decorate</code> adds names of branches or tags of commits shown.

See full list of commands at [Git Cheatsheet](#) from Atlassian

Git Branching/Merging

<code>git branch</code>	List all of the branches in your repo. Add a <code><branch></code> argument to create a new branch with the name <code><branch></code> .
<code>git checkout -b <branch></code>	Create and check out a new branch named <code><branch></code> . Drop the <code>-b</code> flag to checkout an existing branch.
<code>git merge <branch></code>	Merge <code><branch></code> into the current branch.

- Hands-on exercises and tasks in this weeks lab/tutorial. Make sure you attend and practice

See full list of commands at [Git Cheatsheet](#) from Atlassian

References

- Ian Sommerville. 2016. Software Engineering (10th ed.) Global Edition. Pearson, Essex England
- Scott Chacon. 2014. Pro Git (2nd ed.) Apress
 - Free online book – download from <https://git-scm.com/book/en/v2>
- Additional Resources – Paper
 - H-Christian Estler, Martin Nordio, Carlo A. Furia and Bertrand Meyer: *Awareness and Merge Conflicts in Distributed Software Development*, in proceedings of ICGSE 2014, 9th International Conference on Global Software Engineering, Shanghai, 18-21 August 2014, IEEE Computer Society Press (best paper award),
 - http://se.ethz.ch/~meyer/publications/empirical/awareness_icgse14.pdf