Agile Software Development Practices

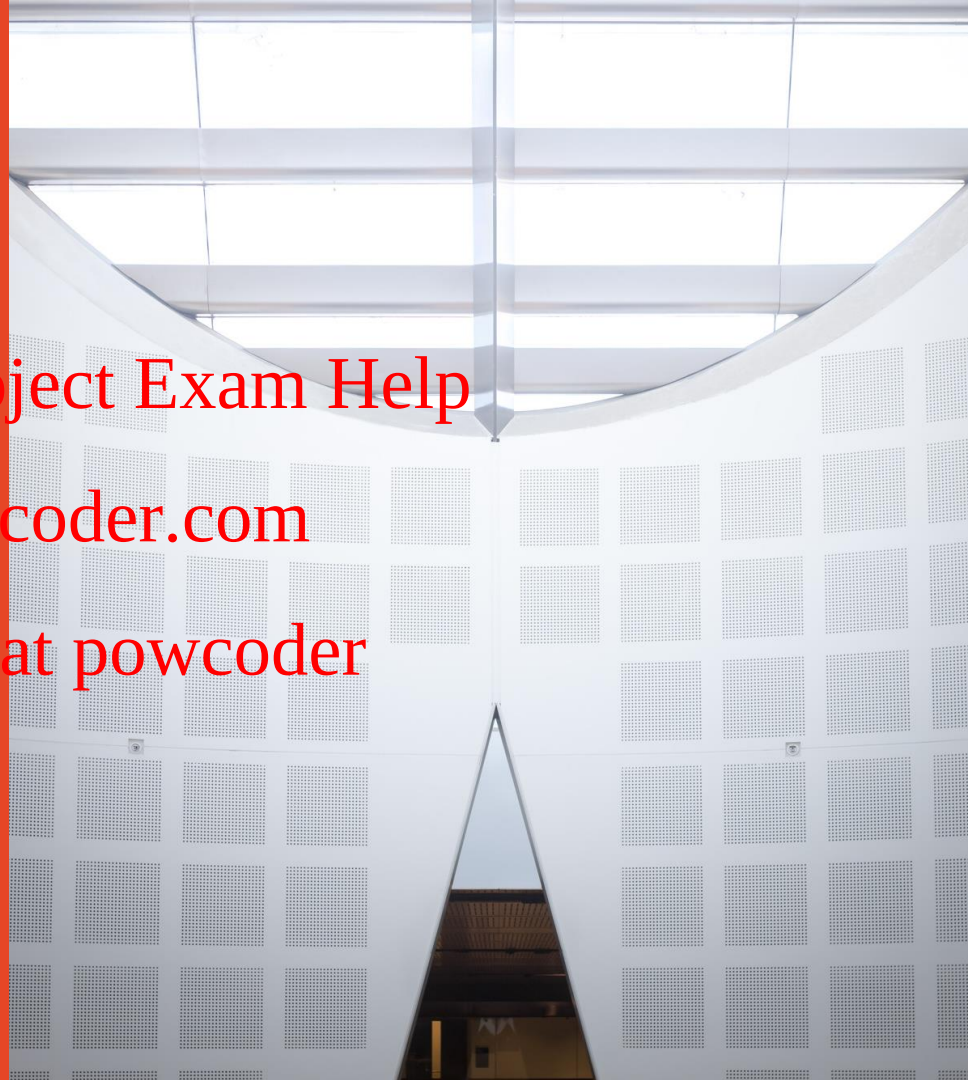
SOF2412 / COMP9412

Tools and Technologies for Controlling Artifacts

Dr. Basem Suleiman

School of Computer Science

THE UNIVERSITY OF SYDNEY



Agenda

- Introduction
  - Software processes (SDLC)
  - Agile Development Model
- Agile Development Tools
  - Software Development – Artifacts
  - Version Control Systems
  - Version Control with Git
  - Using Git Commands

Software Process

The Software Process

- Software Development Process
  - Set of activities required to develop a software
  - Activities are to be done, and in what order
  - Lifecycle for a Software Development project
  - Processes, a set of tools, definitions of the Artifacts, etc.

- Is there a universally applicable software engineering process?
  - Many different types of software systems
  - Companies/engineers claim that they follow “methodology X”, but many times they only do some of what the methodology says
    - Most SW companies developed/customized their own SW process

The Software Process

- Many software development processes, but all include common activities
  - **Specification**
  - **Design and implementation**
  - **Validation**
  - **Evolution**
- Software processes are complex and, rely on people making decisions and judgements
- Activities are complex and include sub-activities
  - E.g., requirements validation, architectural design, unit testing

Software Process Models

- Also known as Software Development Lifecycle (SDLC)

- It presents a description of a process from some particular perspective
  - Describe the activities and their sequence but may not describe the roles of people involved in these activities

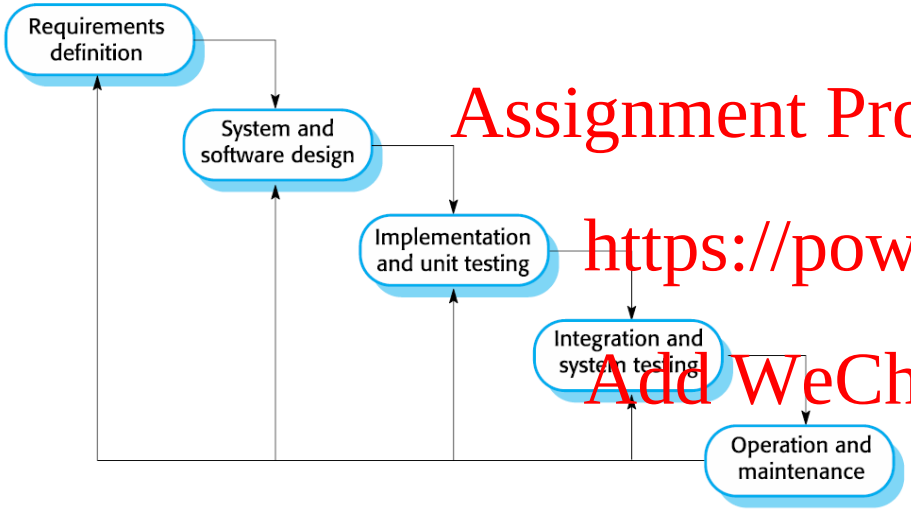
Representative Software Process Models

- **Waterfall Model**
  - Development process activities as process phases
- **Spiral Model**
  - Incremental development risk-driven
- **Agile Model**
  - Iterative incremental process for rapid software development
- **The Rational Unified Process (RUP or UP)**
  - Bring together elements of different process models
  - Phases of the model in timer (dynamic perspectives), process activities (static perspective), good practices (practice perspective)

Waterfall Model Phases

- There are separate identified phases (non-overlapping):
  - Requirements analysis and definition
    - Produces a Requirements document
  - System and software design
    - Requirements document is used to produce a Design document
  - Implementation and unit testing
    - Design document is used to produce code and test it for system components
  - Integration and system testing
    - Software components are integrated and the resulting system is tested
  - Operation and maintenance
- Intensive documenting and planning
- Easy to understand and implement
- Identified deliverables and milestones
- Discovering issues in earlier phases should lead to returning to earlier phase!

Waterfall Model – Heavy-Weight Model



Development activities	Teams
Divide the work into stages	A separate team of specialists for each stage
At each stage, the work is passed from one team to another	Some coordination is required for the handoff from team to team – using "documents"
At the end of all of the stages, you have a software product ready to ship	As each team finishes, they are assigned to a new product

Ian Sommerville. 2016. Software Engineering (10th ed. Global Edition). Pearson

Planning in Software Development

- SW development processes is classified in terms of planning
- **Plan-driven (plan-and-document heavy-weight)**
  - Activities are planned in advance and progress is measured against this plan
  - Plan drives everything and change is expensive
- **Agile processes (light-weight)**
  - Planning is incremental and continual as the software is developed
  - Easier to change to reflect changing requirements
- Most SW processes include elements of both plan-driven and agile
- Each approach is suitable for different types of software
  - There are no right or wrong software processes

Waterfall Model Problems

- **Difficulty of accommodating change** after the process is underway
  - A phase has to be complete before moving onto the next phase

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements
  - Few business systems have stable requirements

- Mostly used for **large systems engineering projects** where a system is developed at several sites
  - The plan-driven nature of the waterfall model helps coordinate the work

Software Failures – Budget, Schedule, Requirements

Project	Duration	Cost	Failure/Status
e-borders (UK Advanced passenger Information System Programme)	2007 - 2014	Over £ 412m (expected), £742m (actual)	Permanent failure - cancelled after a series of delays
Pust Siebel - Swedish Police case management (Swedish Police)	2011 - 2014	$53m (actual)	Permanent failure – scraped due to poor functioning, inefficient in work environments
US Federal Government Health Care Exchange Web application	2013 – ongoing	$93.7m (expected), $1.5bn (actual)	Ongoing problems - too slow, poor performance, people get stuck in the application process (frustrated users)
Australian Taxation Office's Standard Business Reporting	2010 - ongoing	~$1bn (to-date), ongoing	Significant spending on contracting fees (IBM & Fjitsu), significant scope creep and confused objectives

https://en.wikipedia.org/wiki/List_of_failed_and_overbudget_custom_software_projects

Software Evolution

- Software is inherently flexible and can change
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change
- Business software needs to respond to rapidly changing market
  - Time-to-market
- Plan-driven software development processes are not suitable for certain types of SW systems

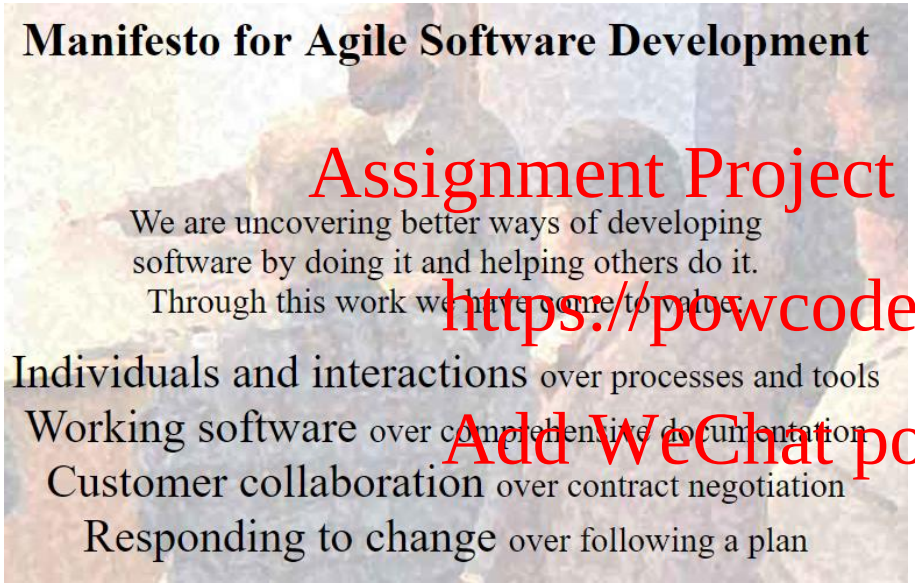
Agile Development Model

THE UNIVERSITY OF SYDNEY

Project Failure – the trigger for Agility

- One of the primary causes of project failure was the **extended period of time it took to develop a system**
- Costs escalated and requirements changed
- Agile methods intend to **develop systems more quickly with limited time spent on analysis and design**

Agile Manifesto (2001) – An Eloquent Statement of Agile Values or Goals



Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

Agile Manifesto: http://agilemanifesto.org/

Agile Process

- Agile advocates believe:
  - Current SW development processes are too heavy-weight or cumbersome
    - Too many things are done that are not directly related to software product being produced
  - Current software development is too rigid
    - Difficulty with incomplete or changing requirements
    - Short development cycles (Internet applications)
  - More active customer involvement needed

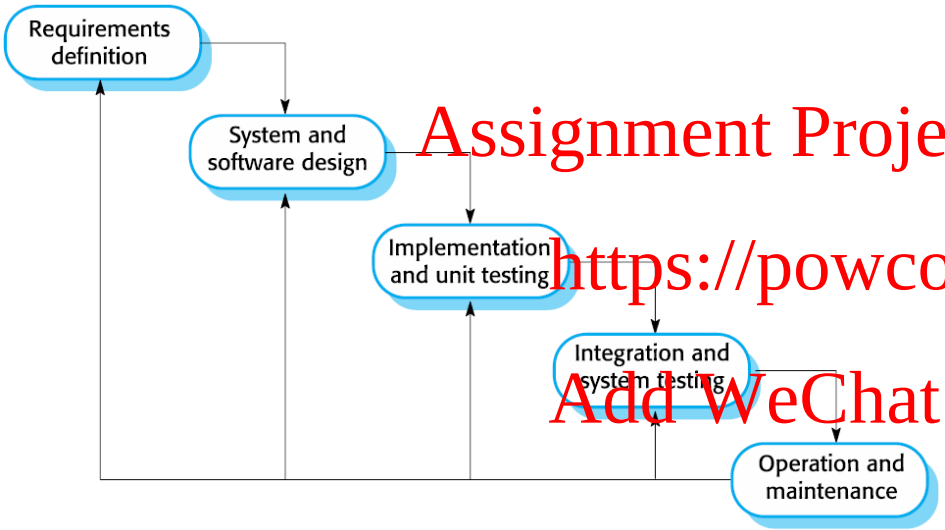
Agile Process

- Agile methods are considered
  - **Light-weight**
  - **People-based** rather than Plan-based
- Several agile methods
  - No single agile method
  - Extreme Programming (XP), Scrum
- Agile Manifesto closest to a definition
  - Set of principles
  - Developed by Agile Alliance

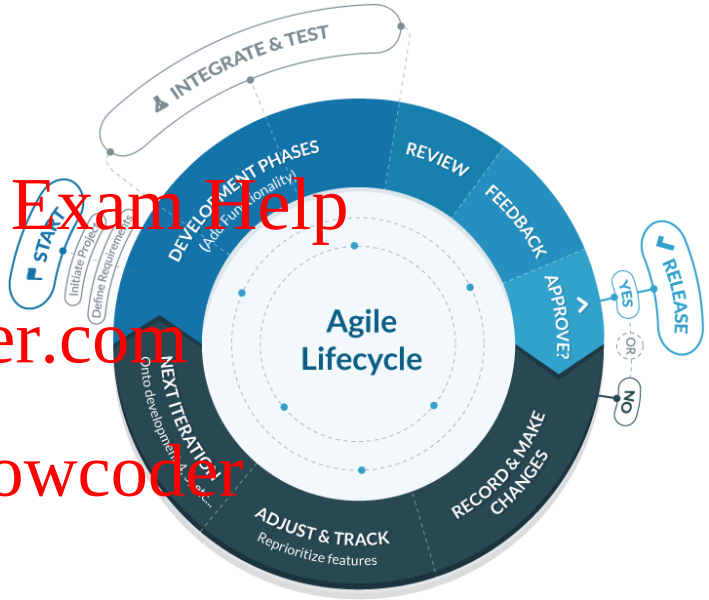
Agile Principles

1. Our highest priority is to **satisfy the customer** through early and **continuous delivery of valuable software**.	5. Build projects around **motivated individuals**. Give them the **environment** and support they need, and **trust** them to get the job done.	9. **Continuous attention** to technical excellence and good design enhances agility.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.	6. The most efficient and effective method of **conveying information** to and within a development team is **face-to-face conversation**.	10. **Simplicity**--the art of maximizing the amount of work not done--is essential.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the **shorter timescale**.	7. **Working software** is the primary **measure of progress**.	11. The best architectures, requirements, and designs emerge from **self-organizing teams**.
4. Business people and developers must **work together daily** throughout the project.	8. Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a **constant pace** indefinitely.	12. At regular intervals, the **team reflects** on how to become more **effective**, then **tunes and adjusts its behavior** accordingly.

Software (Development) Process Models



Waterfall model
plan-driven development



Agile model
Incremental & iterative development

Ian Sommerville. 2016. Software Engineering (10th ed. Global Edition). Pearson

https://blog.capterra.com/agile-vs-waterfall/

Software Development Artifacts

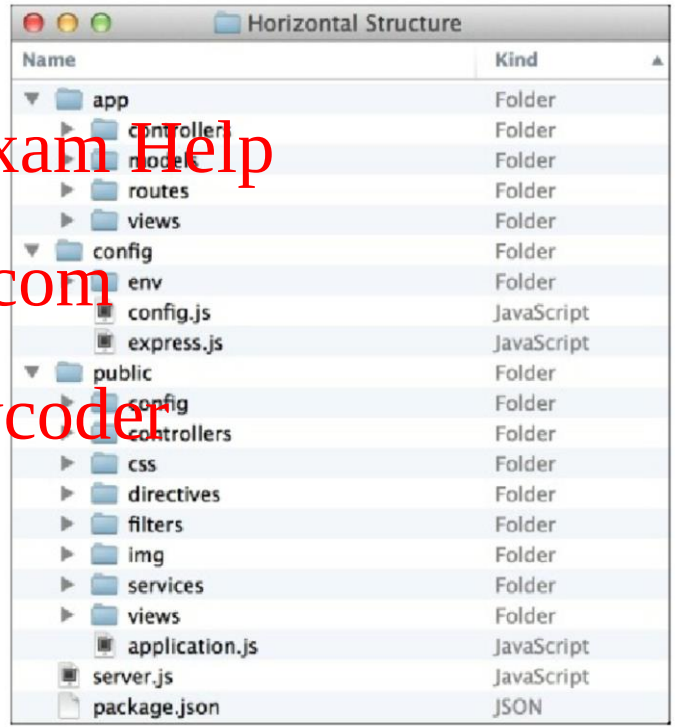
THE UNIVERSITY OF SYDNEY

Artifacts

- Items that represent work done, in ways that others can use
  - Code, requirements specifications
- Artifacts go through **evolution**
- How much impact if these are lost or changes are lost or not tracked?
- How much effort was it to create them?
- The Artifacts have **value** and need to be preserved, communicated, maintained, protected from unauthorized access, etc.

Code Artifacts – Example

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



As source code Artifacts evolve rapidly, it becomes crucial to manage different versions of these Artifacts

Version Control

What is Version Control?

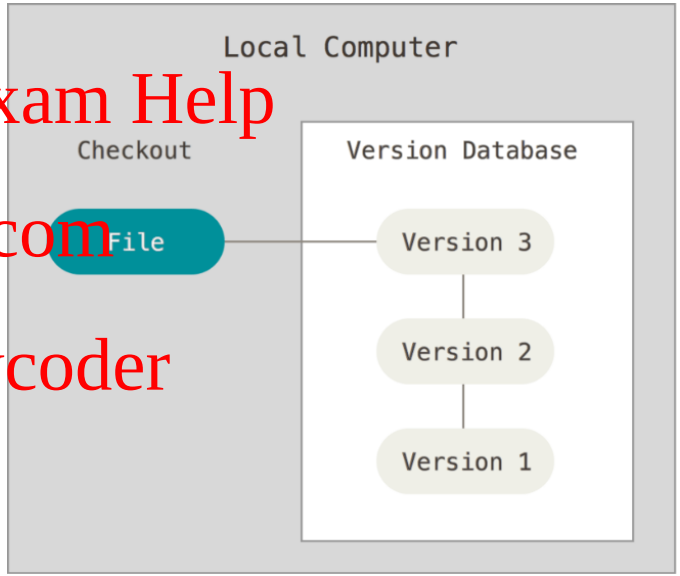
- A method for recording changes to a file or set of files over time so that you can recall specific versions later
  - aka revision control and source control
  - SW development → software source code files
  - Create, maintain and track history of changes during the SDLC for all Artifacts

What is Version Control System (VCS)?

- Category of software tools that help software teams to manage changes to source code over time
  - Keep track of every modification to code in a special kind of storage (*repository*)
  - Revert selected files 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 bugs while minimizing disruption to all team members
  - And more

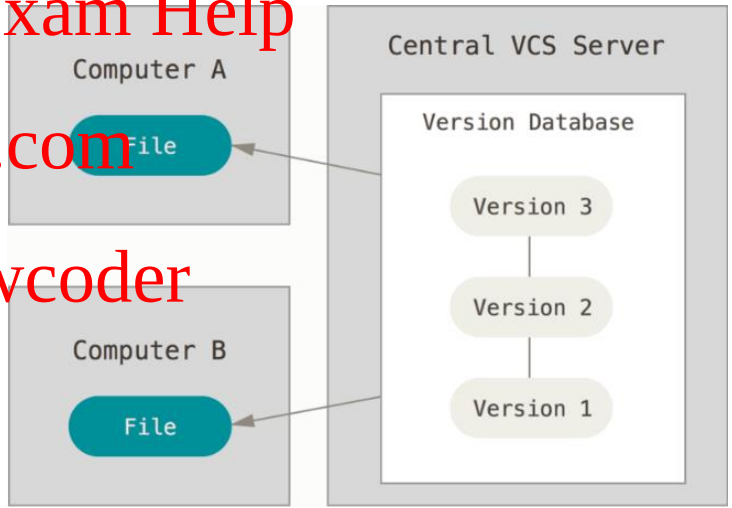
Local Version Control

- Programmers long ago developed local VCSs;
- 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)



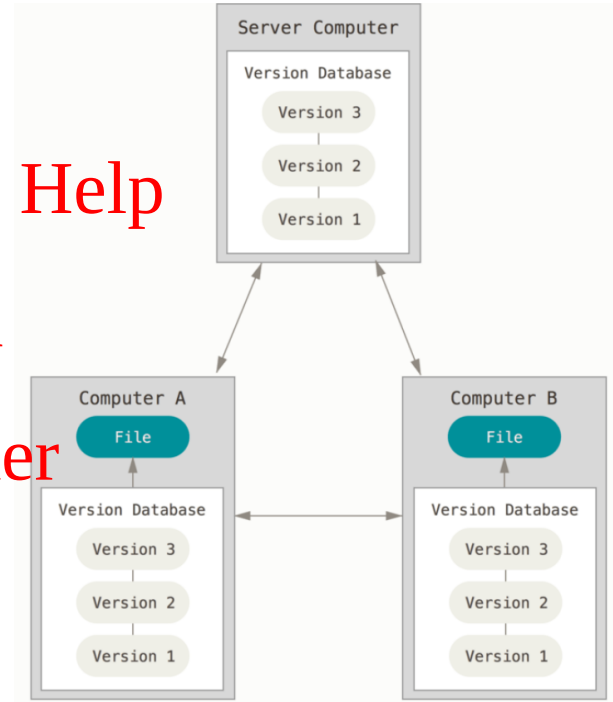
Centralized Version Control (CVC)

- CVCSs support collaborative development
- A single server contains all versioned files and a number of clients check-out files from it
- Better than local VCS
  - Everyone is updated
  - Easier admin - fine-grained control
- Single point of failure
  - Developer's work interrupted!
  - Hard disk becomes corrupted, and no proper/up-to-date backups? entire history lost!



Distributed Version Control (DVC)

- Developers fully mirror the repository including the full history
- 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

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

Git

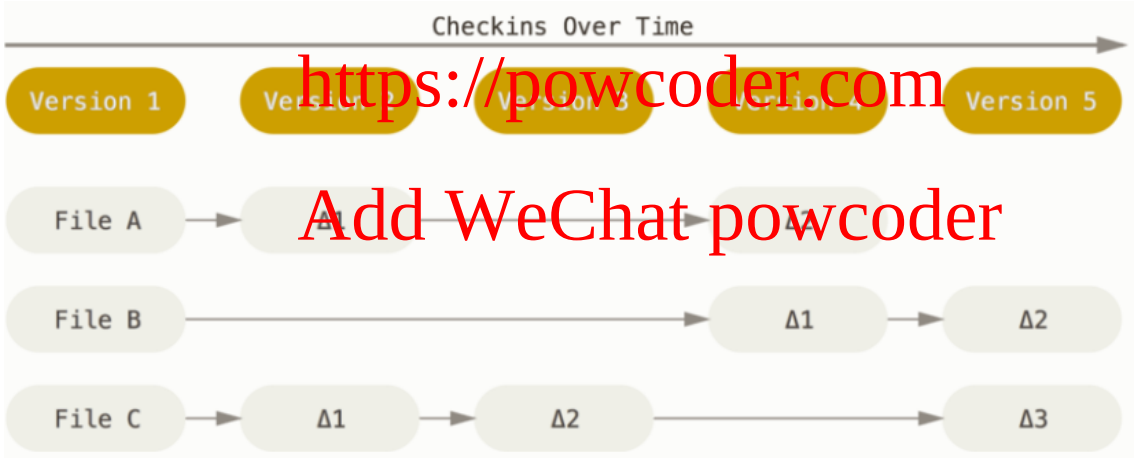
- A version control system that helps development teams to manage changes to source code overtime

- 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 – Snapshots Not Differences

- Git thinks about its data as a ***streams of snapshots*** of a small file system
  - Git doesn't store unchanged files, it just link to previous identical file already stored

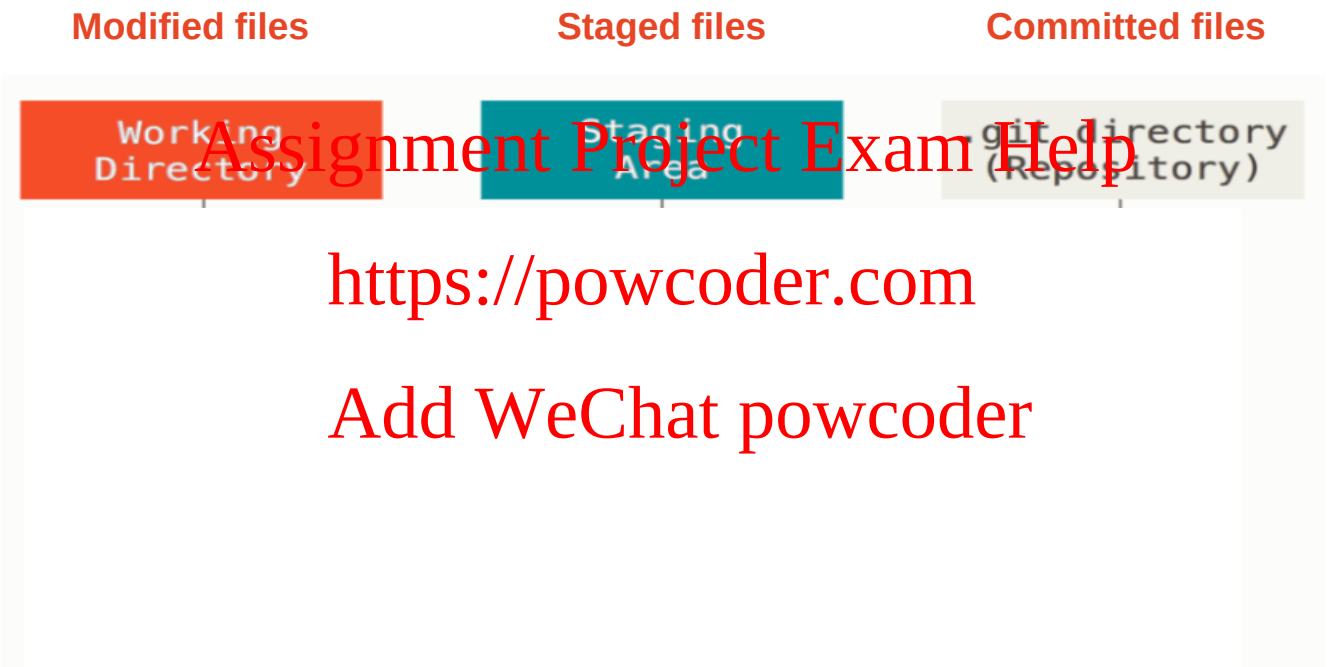
Git – Basics

- **Nearly Every operation is local**
  - E.g., to browse the project history, Git reads it directly from *your local database* — not from the server to display it
  - Work and commit changes to local copies offline
- **Git has built-in Integrity**
  - Everything in Git is check-summed (SHA-1 hash) before it is stored and is then referred to by that checksum
  - 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**
  - ***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

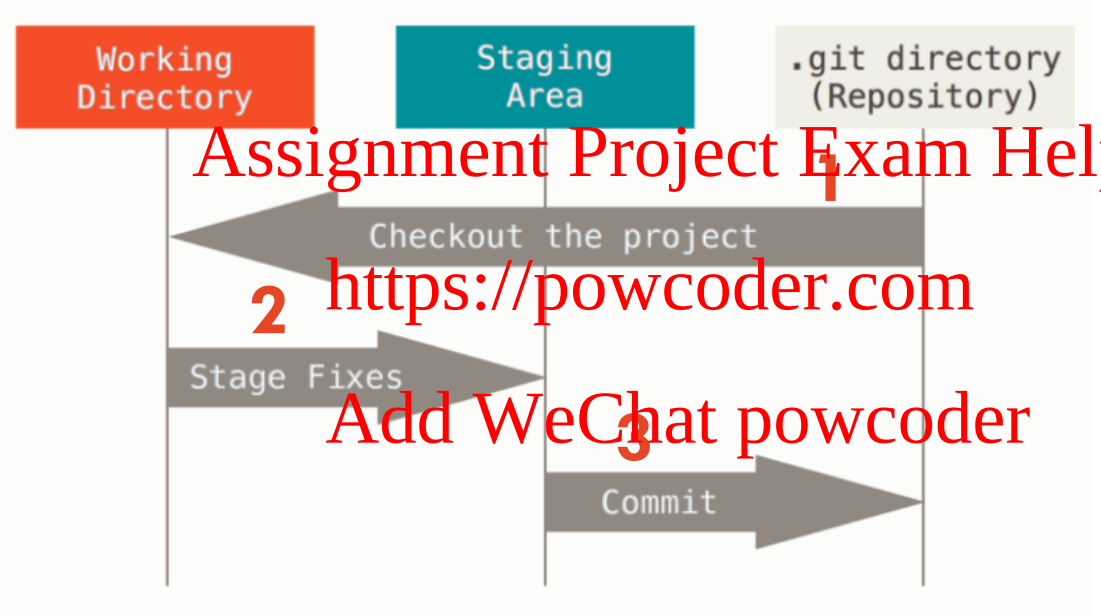
Git – Structure



Git – Structure

- **Git directory (repository)**
  - Metadata and object database
  - 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 and placed on disk for you to use or modify
- **Staging area (index)**
  - a file stores information about what will go into next commit

Git – Basic Workflow



Git Concepts and Scenarios

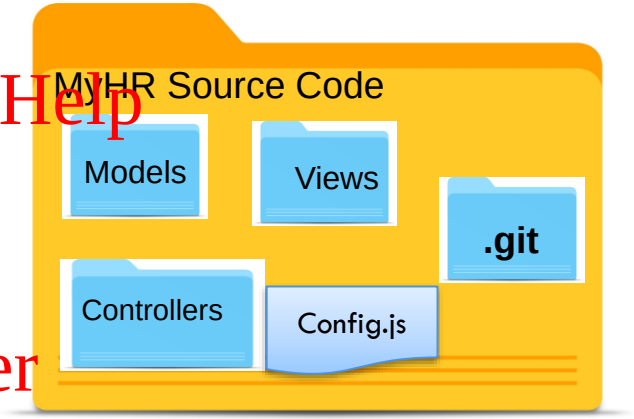
Git Commands and Operations

THE UNIVERSITY OF SYDNEY

Git Repository (Repo)

- A special directory contains project files
  - Where git stores and tracks files (source code)
  - Can be created or cloned
  - Git adds special sub-directory to store **history of changes** about the project's files and directories

- Creating a git repo on your local machine
  - **git init** → will create **.git**
  - Clone an existing git repository from elsewhere
    - **git clone**
    - a full copy of all data that the server has

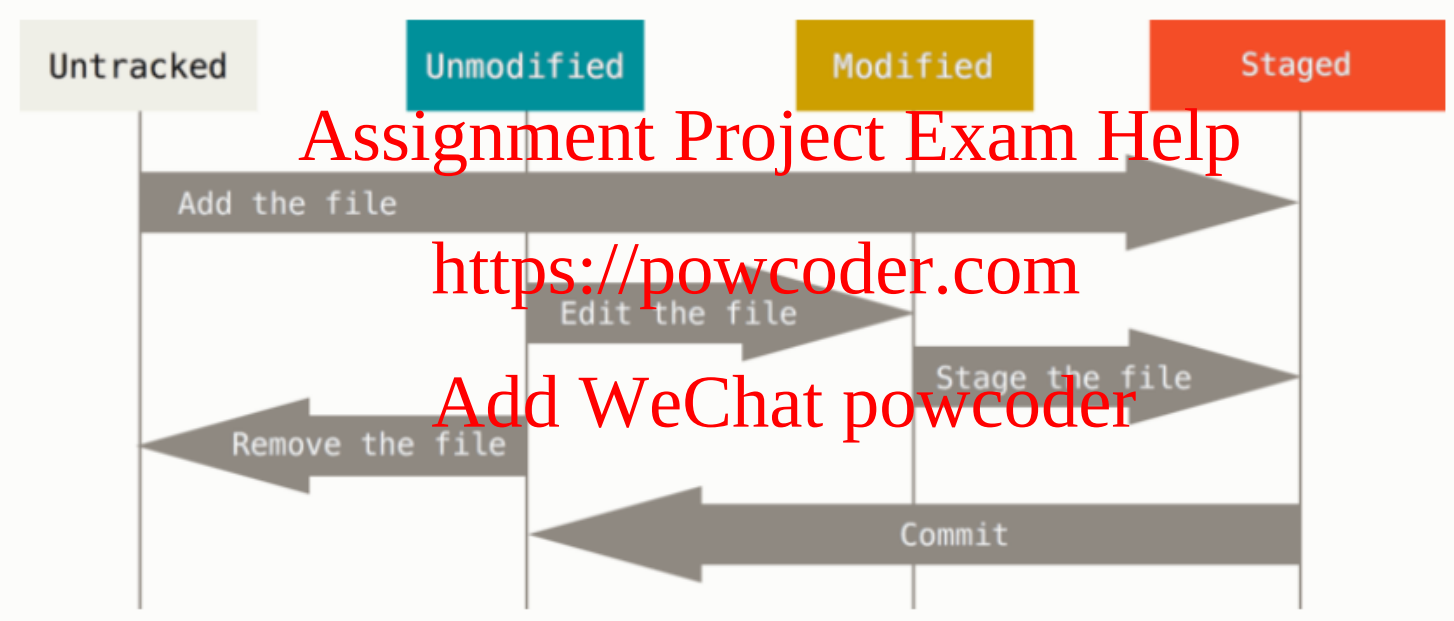


https://www.iconspng.com/

Metadata

- Each version should have:
  - Unique name to refer to it
    - Latest version: **Head**
  - Date
  - Author
- How might you use metadata?

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
- When a repo is cloned then all files are tracked and unmodified
- When you edit files, Git denotes them as modified
- 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

- Git branching is lightweight (nearly instantaneous)
  - 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

- Assume your project directory contains 3 files. Stage all files:
- `$ git add README test.rb LICENSE`
- Staging files: git computes a checksum of each and stores the files in the git repo (as blobs)

Git Branching – Example

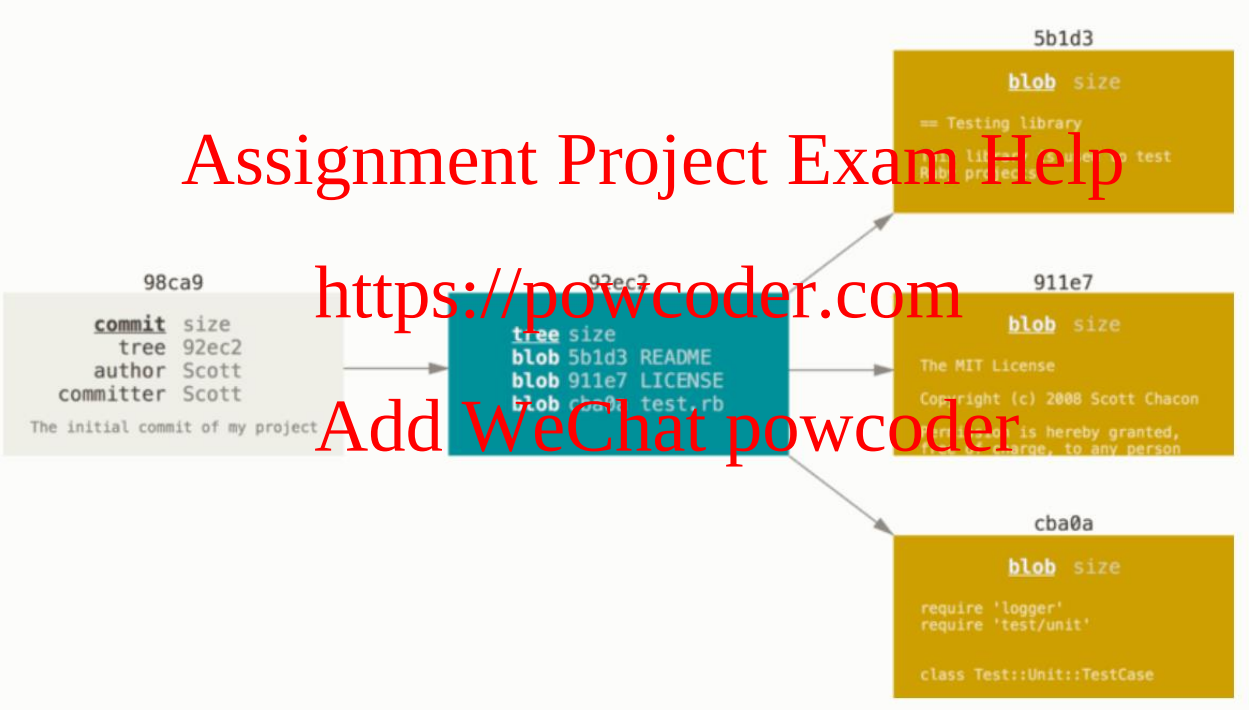
- Create a commit:

```
$ git commit -m 'The initial commit of my project'
```


- git checksums each sub-directory 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

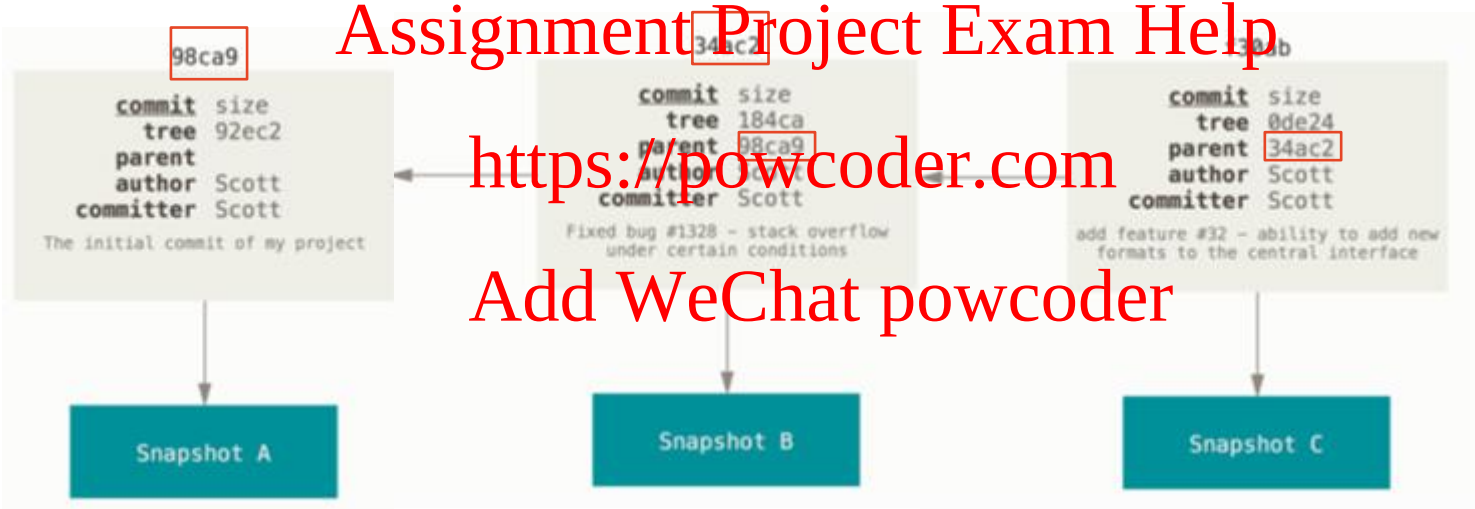
- Our project directory now contains five objects,
  - **Three blobs** (contents of one of the 3 files)
  - One **tree** lists the directory contents and file names as blobs
  - One **commit** with the pointer to the root and the commit metadata

Git Branching – Example



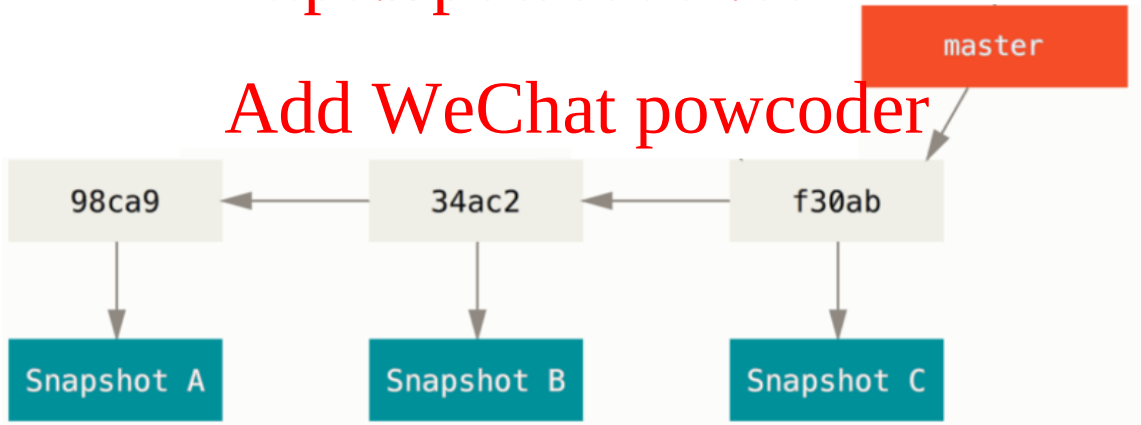
Git Branching – Example

- 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

- *lightweight movable pointer* to one of the commits
- Default branch called "***master***"
- 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

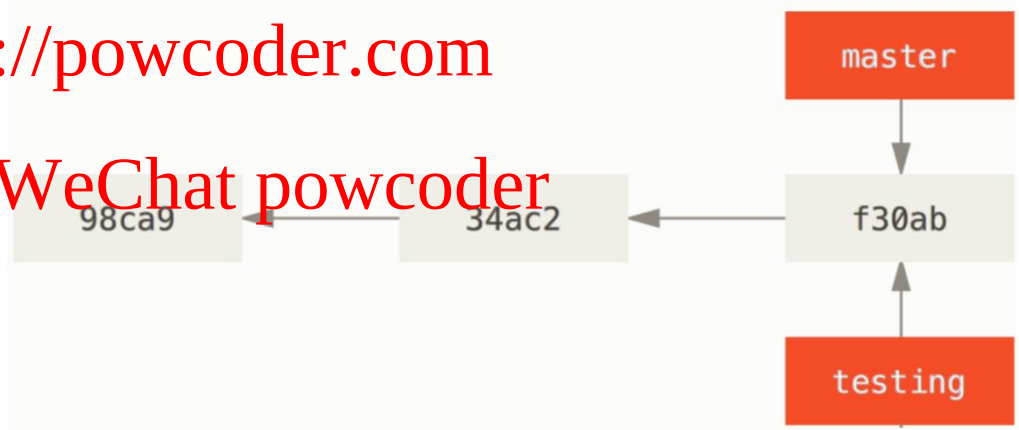


Git – Creating a New Branch

- Example: create a new branch called **testing** in our project

```
$ git branch testing
```


- When you create a new branch, a new pointer will be created – pointing to the same commit we are currently at

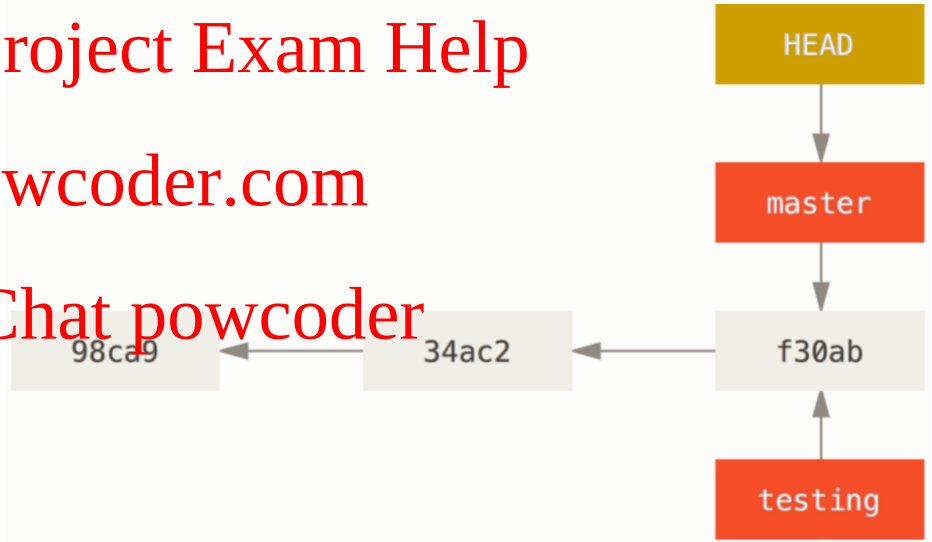


- **How does git know what branch you're currently on?**

Git – Creating a New Branch

- Git knows which branch is the current by maintaining a *special pointer* called "**HEAD**"

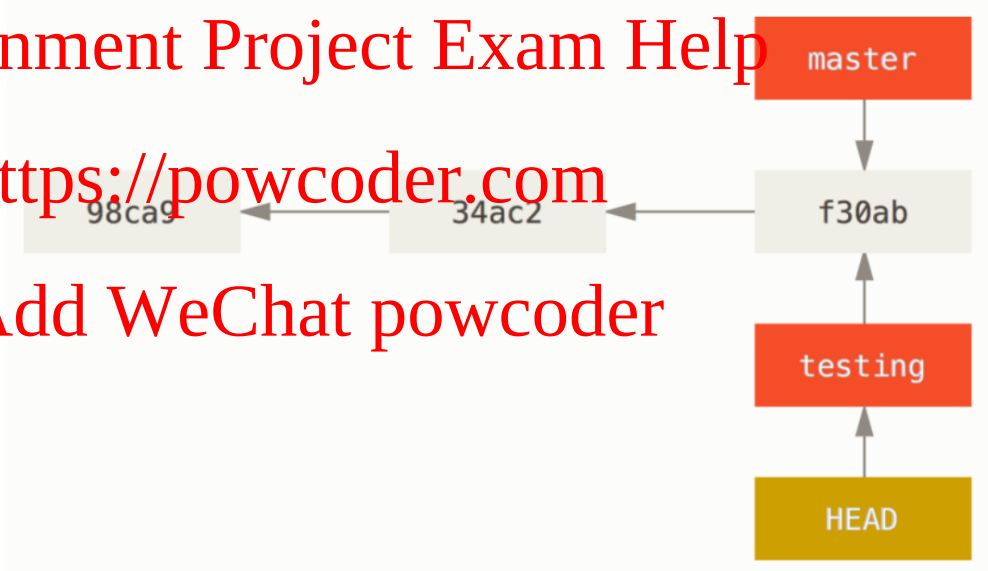
- Creating a branch in Git does not switch the HEAD to the new branch
- **How to switch to another branch ?**



Git – Switching to another Branch (1)

- You can switch to an existing branch using Git *checkout* command

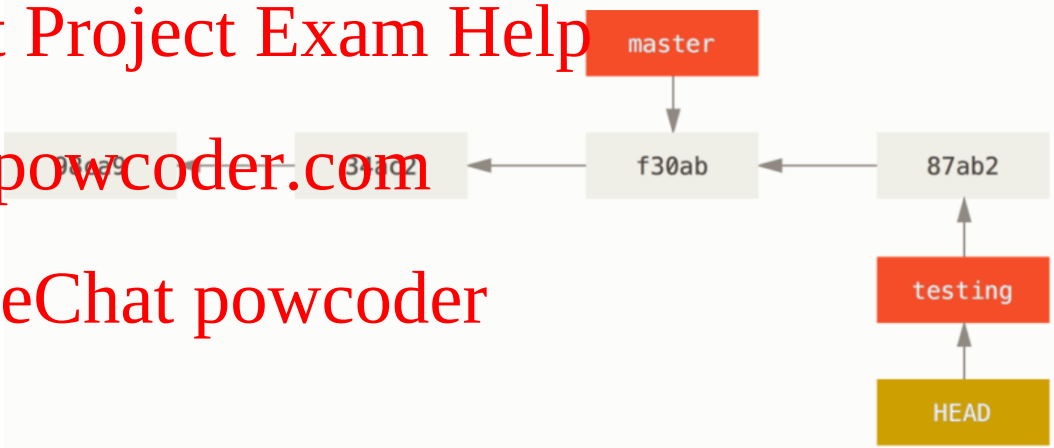
```
$ git checkout testing
```

Git – Switching to another Branch (2)

Assume you edited test.rb file. let's do another commit

$ git commit -a -m 'made a change'

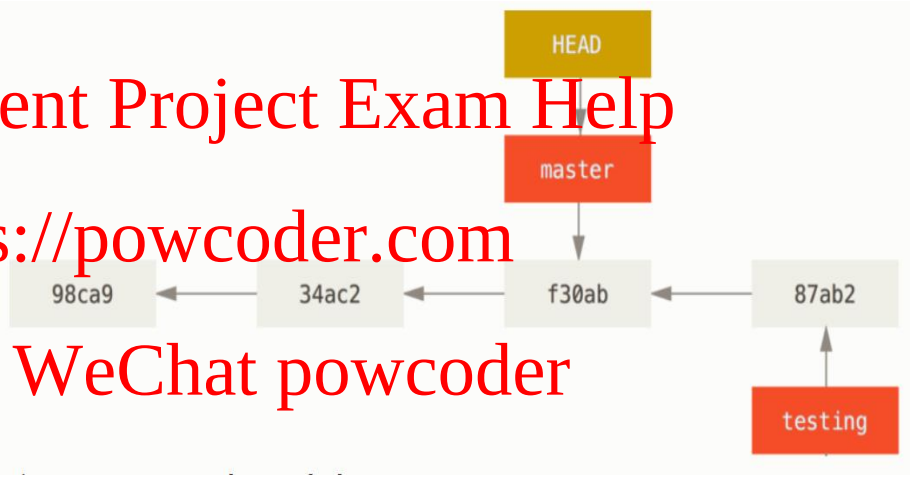


- How would this impact our repo?
- The testing branch has moved forward, but the master branch still points to the commit you were on when you ran git checkout

Git – Switching to another Branch (3)

- What happens if we switch back to the master branch?

```
$ git checkout master
```

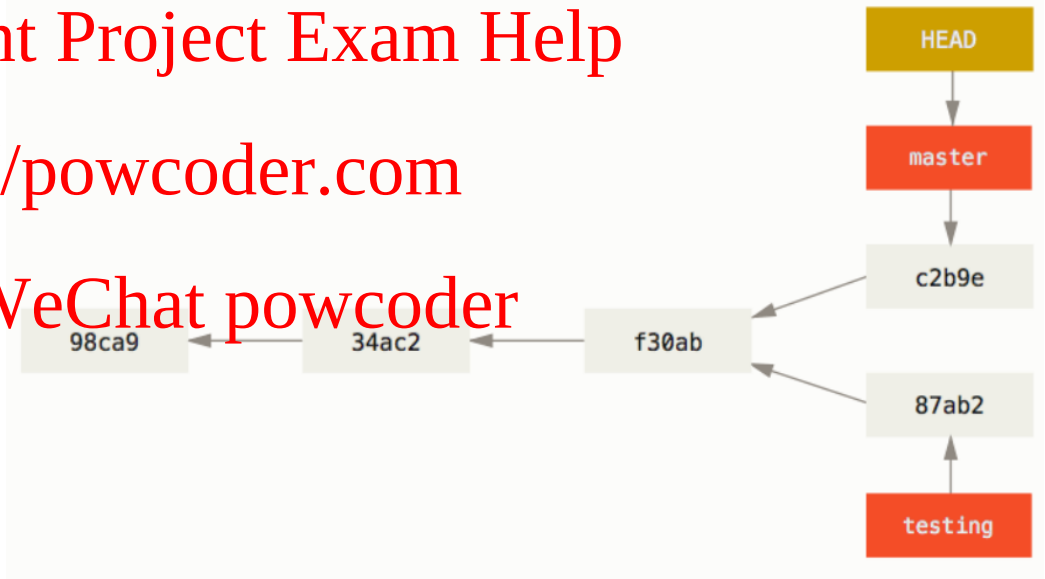
- Any changes we make from this point forward will diverge from an older version of the project
  - You can branch into another direction (not from testing branch)

Git – Switching to another Branch (4)

- Assume test.rb is edited and we want to commit – What happens?

```
$ git commit -a -m 'made other changes'
```


- Both of those changes are isolated in separate branches
  - Switch back and forth between the branches and merge them together when you're ready



- To show the commits log use:

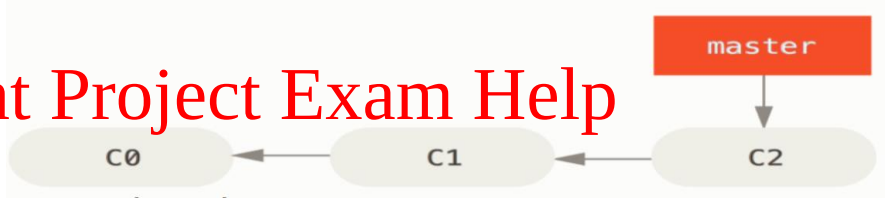
```
$ git log
```

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 your project has a couple of commits already on the master branch

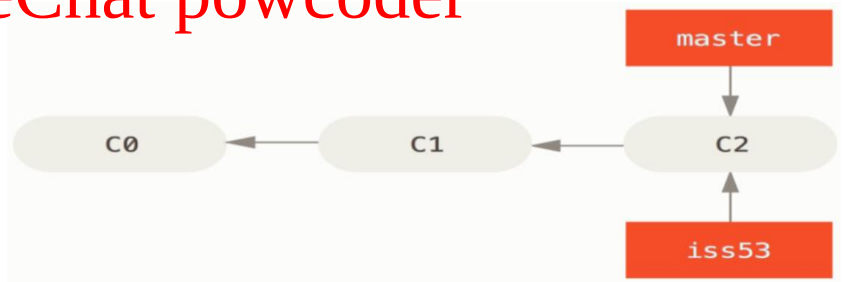


- To work on the issue (say *iss53*) you need to create a new branch and switch to it at the same time (using git *branch* and *checkout*)


```
$ git checkout -b iss53
```

OR

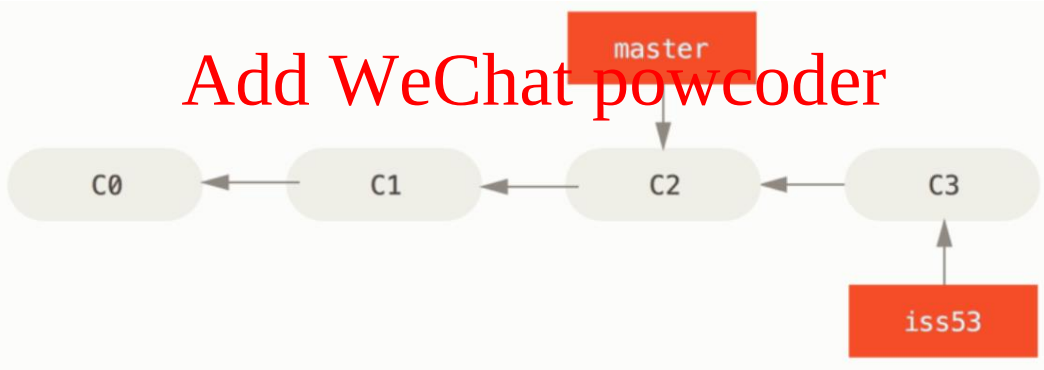
```
$ git branch iss53
$ git checkout iss53
```



Git – Merging Scenario (3)

- You work on your website and do some commits (checked it out)
  - E.g., change index.html to add a new footer


```
$ git commit -a -m 'added a new footer [issue 53]'
```

Git – Merging Scenario (4)

- Imagine you receive a call for urgent issue in the website needs immediate fix
- How would you deal with this scenario?

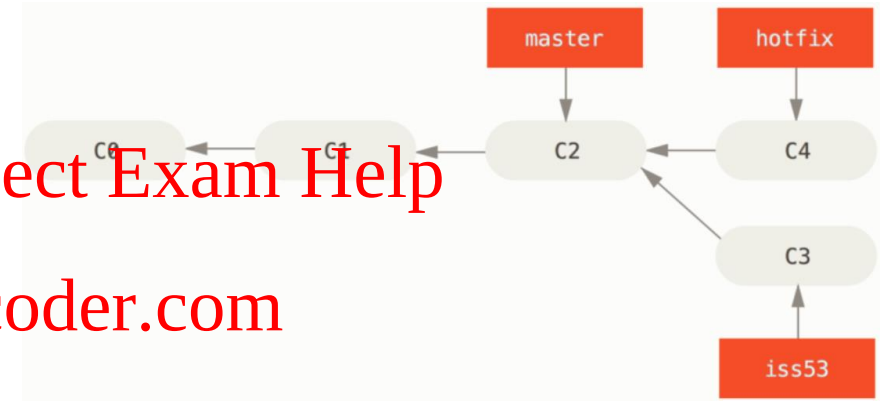
- You can switch back to the master branch
  - But you need to have clean working state before switch branches; i.e., working directory doesn't have uncommitted changes


```
$ git checkout master
```

Git – Merging Scenario (5)

- Create a new branch for fixing the urgent issue, e.g., '*hotfix*'

```
$ git checkout -b hotfix
```



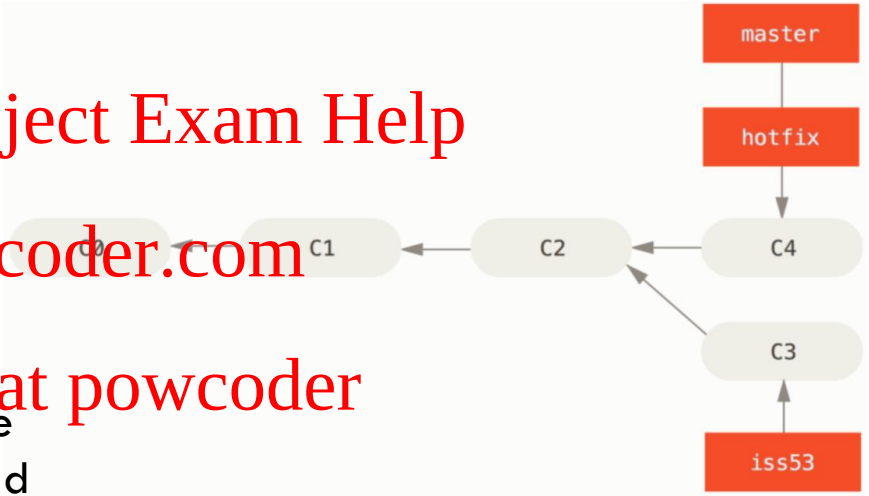
- Edit the index.html to fix broken email address


```
$ git commit -a -m 'fixed the broken email
address'
```

Git – Merging Scenario (6)

- merge the hotfix branch back into your master branch to deploy to production

```
$ git checkout master
$ git merge hotfix
```

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

Git – Merging Scenario (7)

- Delete the hotfix branch (no longer needed):

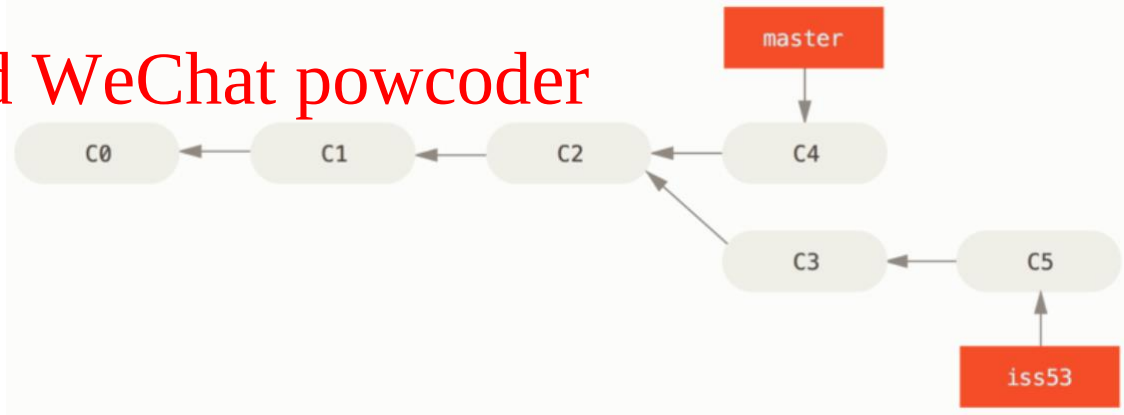
```
$ git branch -d hotfix
```


- Switch back to the *iss53* branch and continue working on it, edit index.html and commit changes you make on *iss53*


```
$ git checkout iss53
```



```
$ git commit -a -m 'finished the
new footer [issue 53]'
```



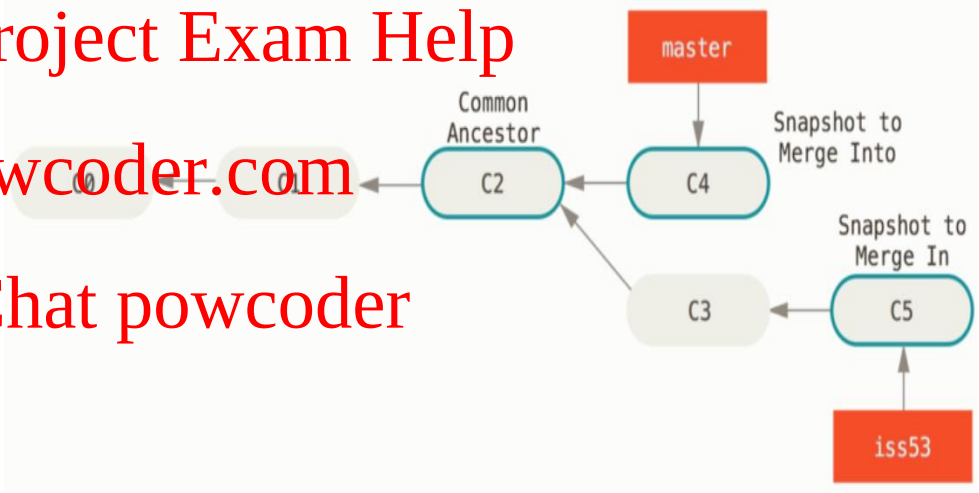
Git – Merging Scenario (8)

- Issue #53 work is complete and ready to be merged into the master branch

```
$ git checkout master
$ git merge iss53
```

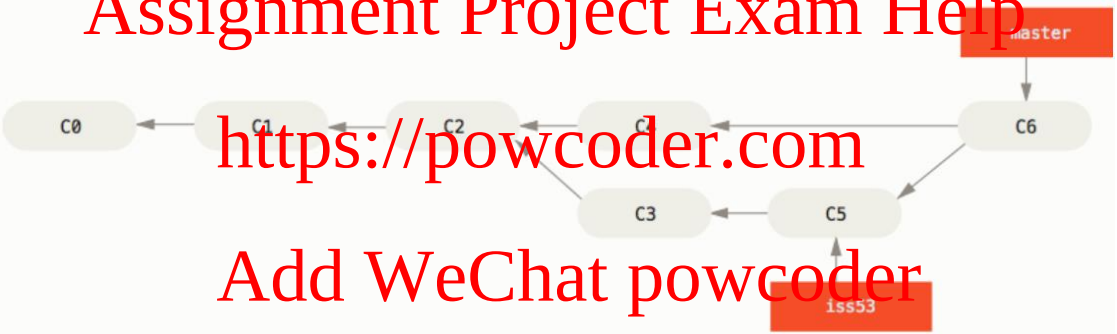

- Note – here the development history has diverged from some older point (commit on the branch we're on isn't a direct ancestor of the branch we're merging in)

- 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 (9)

- 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 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

THE UNIVERSITY OF SYDNEY

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

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

See full list of commands at Git Cheatsheet from Atlassian

Git Log

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

See full list of commands at Git Cheatsheet from Atlassian

Git Commands

Undoing Branches

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

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

Tools and Technologies for Controlling Artifacts

Advanced Git

THE UNIVERSITY OF SYDNEY