

COMP110: Principles of Computing

# **Advanced GitHub Usage**

# Learning outcomes

By the end of this session, you will...

- ▶ Appreciate the usefulness and importance of **version control** in collaborative software projects
- ▶ Understand how to resolve **merge conflicts**, and how to minimise their occurrence
- ▶ Know about **branching**, and how it can fit into your team's workflow

# **Version control: the basics**

# What is version control?

- ▶ A system for managing **changes** to source code
- ▶ Tracks the **history** of changes
- ▶ Allows changes from multiple developers to be **merged**
- ▶ Allows for multiple **forks** and **branches** of the same code

# Why use version control?

Here's what **life without version control** looks like:

- ▶ You need to make **manual backups** of your code
- ▶ It's hard (or impossible) to **undo** a change
- ▶ Maintaining **multiple copies** of the codebase (e.g. production, development, testing) is a hassle
- ▶ Work from **multiple developers** needs to be manually copied and pasted together
- ▶ Code needs to be **transferred** around by dropbox / email / flash drive

Version control (used properly!) **takes care of all of the above**

# What is Git?



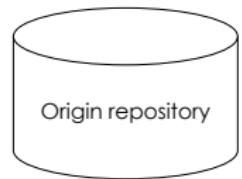
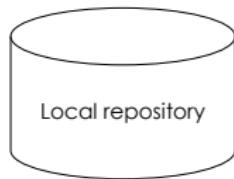
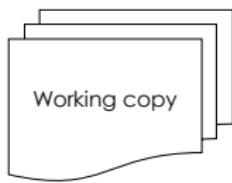
# What is Git?

- ▶ **Git** is an open-source **distributed version control system**
- ▶ Originally developed by Linus Torvalds and other Linux kernel developers
- ▶ **GitHub** is a site that provides free Git hosting

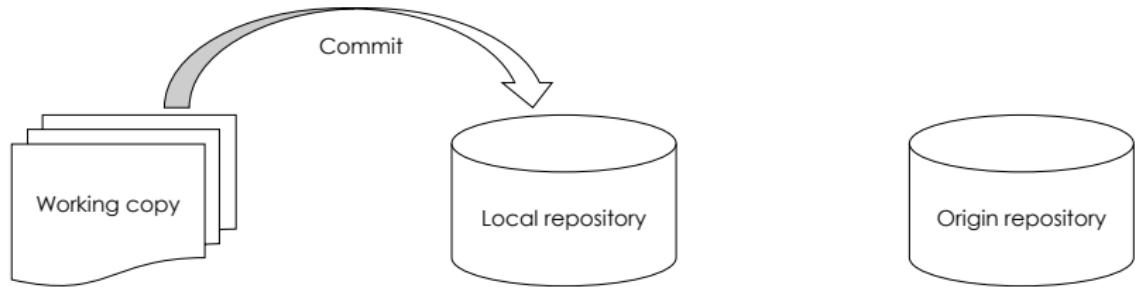
# Basic Git workflow

- ▶ **Origin repository:** the repository on GitHub's servers
- ▶ **Local repository:** a copy of the repository on your local machine (usually stored in a folder named `.git`)
- ▶ **Working copy:** the source files you are editing

# Basic Git workflow

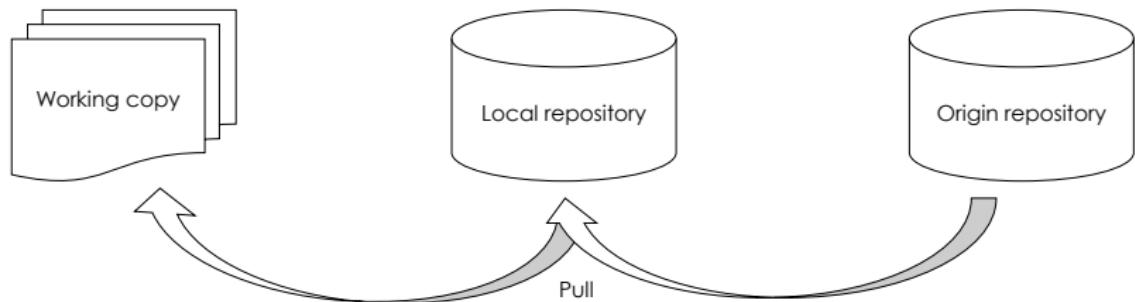


# Basic Git workflow



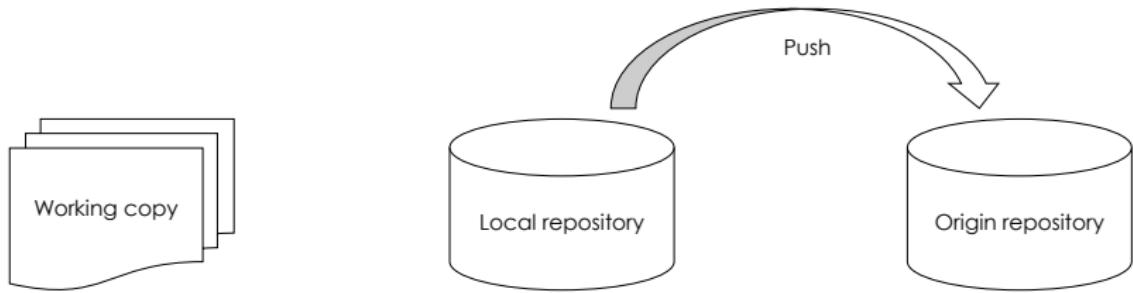
**Commit** adds your recent changes to your local repository

# Basic Git workflow



**Pull** fetches any changes from the GitHub server

# Basic Git workflow



**Push** sends your committed changes to the GitHub server

# Basic Git workflow: recap

- ▶ **Commit** when you have changed something
- ▶ **Pull** to fetch the latest changes from the server
- ▶ **Push** to upload your changes to the server
- ▶ Generally want to stick to doing things in this order, in particular:
  - ▶ Always **pull** before you **push**
  - ▶ Always **commit** before you **pull**
- ▶ Some clients (e.g. GitHub for Windows) combine pull and push into a single **sync** command

# When to commit

- ▶ **Commit early and commit often**
- ▶ If you implement something, test it and it works, commit it!
- ▶ **Rule of thumb:** if it takes more than one sentence to describe what you have changed, you should have committed earlier!

# How (not) to write commit messages

|   | COMMENT                            | DATE         |
|---|------------------------------------|--------------|
| ○ | CREATED MAIN LOOP & TIMING CONTROL | 14 HOURS AGO |
| ○ | ENABLED CONFIG FILE PARSING        | 9 HOURS AGO  |
| ○ | MISC BUGFIXES                      | 5 HOURS AGO  |
| ○ | CODE ADDITIONS/EDITS               | 4 HOURS AGO  |
| ○ | MORE CODE                          | 4 HOURS AGO  |
| ○ | HERE HAVE CODE                     | 4 HOURS AGO  |
| ○ | AAAAAAA                            | 3 HOURS AGO  |
| ○ | ADKFJSLKDFJSOKLFJ                  | 3 HOURS AGO  |
| ○ | MY HANDS ARE TYPING WORDS          | 2 HOURS AGO  |
| ○ | HAAAAAAAAANDS                      | 2 HOURS AGO  |

AS A PROJECT DRAGS ON, MY GIT COMMIT  
MESSAGES GET LESS AND LESS INFORMATIVE.

<https://xkcd.com/1296/>

# Commits are changes

```
56  assignment_briefs/comp140_1/comp140_1.tex
57
58
59
60
61
```

| 55 | 55 | @@ -55,18 +55,29 @@ \subsection*(A. \emph{Design} a novel game controller device)                                 |
|----|----|---|
| 56 | 56 | \textbf{Formative submission:} Arrange a meeting with your tutor to discuss your task board.                      |
| 57 | 57 | \textbf{Summative submission:} Take screenshots of your Trello task board,  |
| 58 |    | - and include them in your summative submission on LearningSpace.   |
|    | 58 | + and add them to your GitHub repository (see below) for inclusion in your summative submission on LearningSpace. |
| 59 | 59 |   |
| 60 | 60 | \subsection*(B. \emph{Build} a prototype of your game controller}   |
| 61 | 61 |   |

- ▶ A commit is a set of **changes** to files
- ▶ A commit **is not** a snapshot of the code (although it can be used to make one)

# Collaborative development

- ▶ Alice changes something in `game.cpp`, commits it and pushes it
- ▶ **Meanwhile**, Bob changes something in `game.cpp` and commits it
- ▶ Bob pulls Alice's changes. What happens?

# Merging changes

- ▶ If Alice **hasn't** edited any of the **same lines of code** as Bob, Git will **merge** the changes automatically
  - ▶ NB: Git is clever about figuring out **where** a change should be applied — e.g. it **doesn't** just use line numbers, so inserting code earlier in the file shouldn't mess it up
  - ▶ Bob will see **both** his and Alice's changes, and so will Alice next time she pulls
- ▶ If they **have** edited the same lines, Git won't be able to merge them automatically — this is a **conflict!**

# Resolving conflicts

- ▶ If there is a conflict, the offending file will look like this:

```
int main(int argc, char* argv[])
{
    std::cout << "Hello world!" << std::endl;
<<<<<< HEAD
    std::cout << "My name is Bob!" << std::endl;
=====
    std::cout << "My name is Alice!" << std::endl;
>>>>> origin/master
    return 0;
}
```

- ▶ Clearly this isn't valid C++ code!
- ▶ You are given **both versions** of the code — you need to **manually edit** the code to make sense again

# Resolving conflicts

```
int main(int argc, char* argv[])
{
    std::cout << "Hello world!" << std::endl;
    std::cout << "Our names are Alice and Bob!" << std ←
        ::endl;
    return 0;
}
```

- ▶ After editing **and testing**, commit and push as normal

# Prevention is better than the cure

- ▶ In a properly managed project, conflicts should be a **rarity**
- ▶ **Coordinate** your efforts so that developers are not working on the same lines of code at the same time
- ▶ Don't make **far-reaching changes** (e.g. refactoring) without first consulting the rest of your team
- ▶ Use a sensible **branching** strategy (see later) to manage and plan the times when code is to be merged

# **Branching**

# Forking vs branching

- ▶ Both represent a set of changes based on, but separate from, the repository's "main" set of commits
- ▶ A **fork** is a completely separate repository
- ▶ A **branch** exists within the same repository
- ▶ Which to use?
  - ▶ We recommend you use branches — they are easier to work with, and keep everything within the same repository
  - ▶ In industry, many (larger) teams use forks, but also many do not

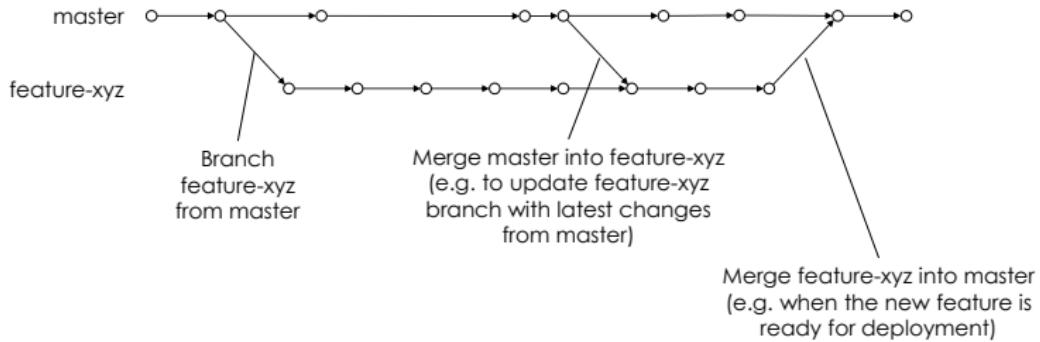
# Branching

- ▶ Every repository starts with one branch called `master`
- ▶ New branches can be created at any time
- ▶ Branches can be **merged**
  - ▶ “Merge A into B”: take the commits (remember: changes) from branch A, and apply them to branch B

# The golden rule

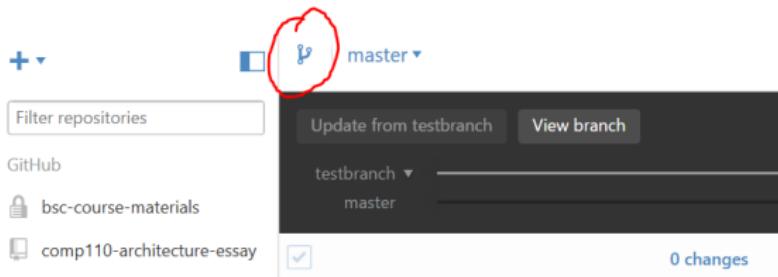
- ▶ The `master` branch should always be **deployable**
  - ▶ No half-finished features
  - ▶ No obvious bugs
  - ▶ **Definitely** no compile errors
  - ▶ If a VIP walked into the room and asked for a demo, `master` is what you'd show them

# Branching



# Creating a branch

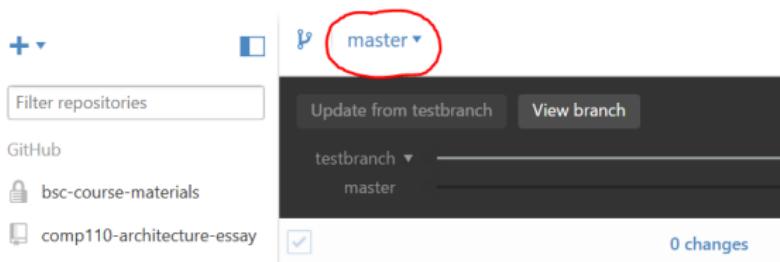
- ▶ GitHub for Windows: click this button



- ▶ TortoiseGit: use the **Create Branch** context menu item

# Switching branches

- ▶ You can switch your **working copy** (source files on disk) to a different branch at any time
- ▶ Always make sure you're committing to the correct branch!
- ▶ GitHub for Windows: use this dropdown menu



- ▶ TortoiseGit: use the **Switch/Checkout** context menu item

# Merging branches

- ▶ GitHub for Windows:
  - ▶ Switch to the branch you want to merge **to**
  - ▶ Click **Compare** and select the branch you want to merge **from**
  - ▶ Click the **Update from <branchname>** button
  - ▶ Resolve any conflicts
- ▶ TortoiseGit:
  - ▶ Switch to the branch you want to merge **to**
  - ▶ Use the **Merge** context menu item
  - ▶ Resolve any conflicts

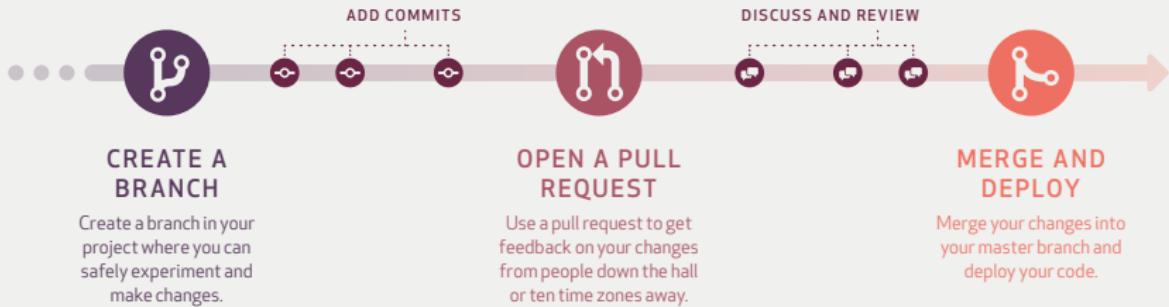
# **Working with branches**

# GitHub flow

<https://guides.github.com/introduction/flow/>

# WORK FAST WORK SMART **THE GITHUB FLOW**

The GitHub Flow is a lightweight, branch-based workflow that's great for teams and projects with regular deployments. Find this and other guides at <http://guides.github.com/>.

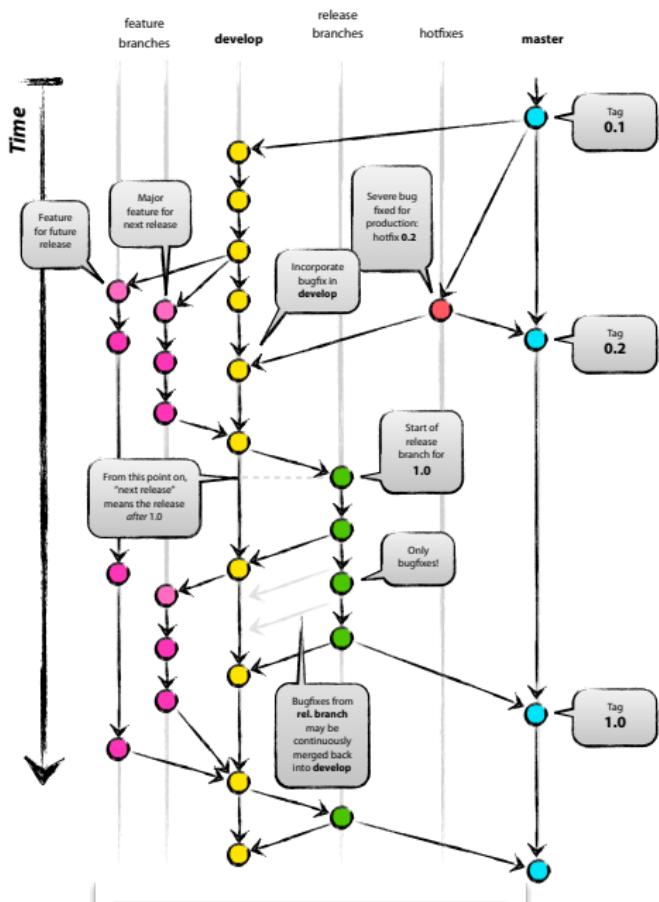


GitHub is the best way to build software together.

GitHub provides tools for easier collaboration and code sharing from any device. Start collaborating with millions of developers today!

# Git flow

[http://nvie.com/posts/  
a-successful-git-branching-model/](http://nvie.com/posts/a-successful-git-branching-model/)



# Discussion

What are the pros and cons of **Git flow** vs **GitHub flow**?

# **Activity: branching and merging**