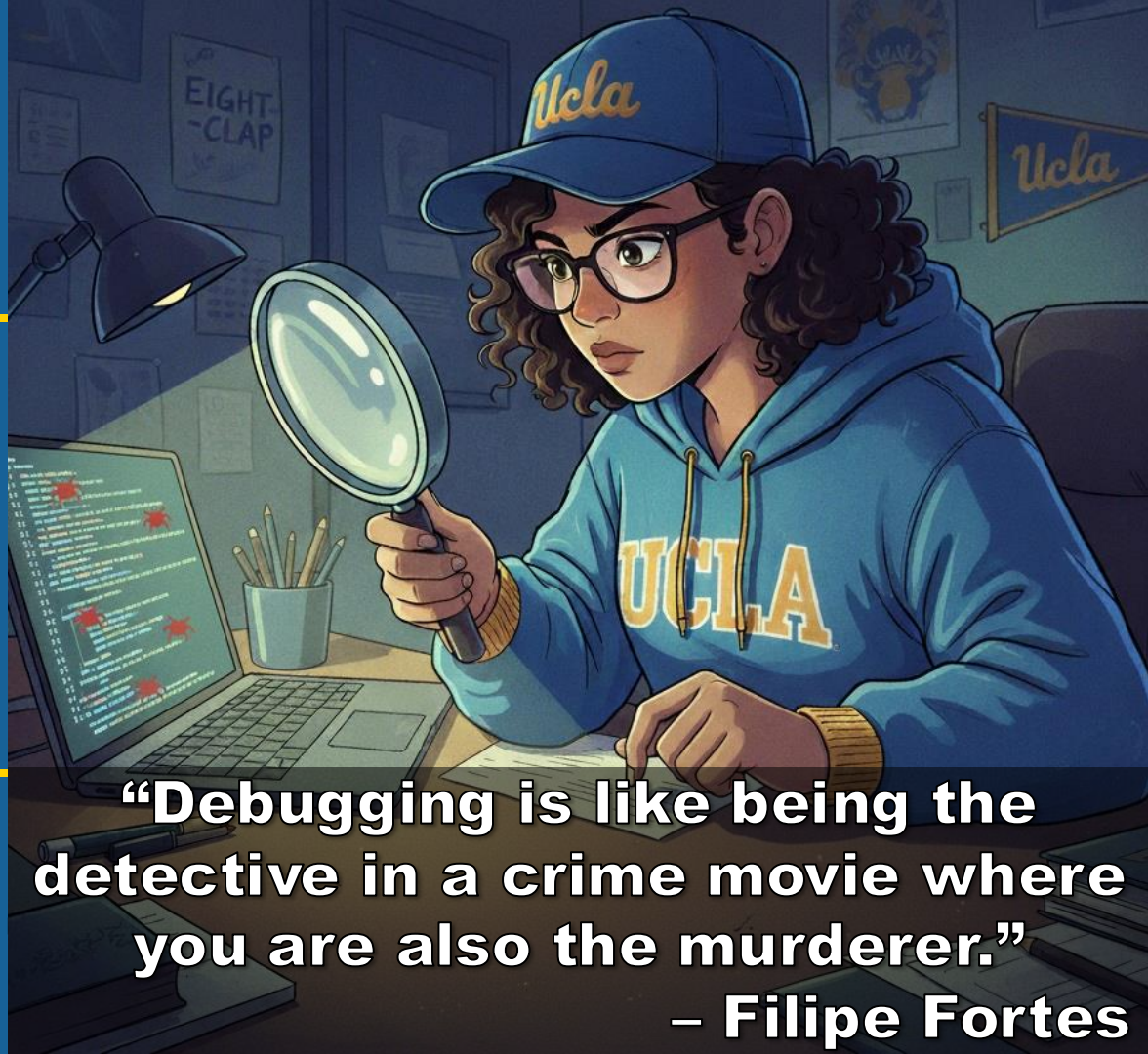
UCLA Samueli School of Engineering

CS 35L Software Construction

Lecture 10 – Debugging & Git

Tobias Dürschmid
Assistant Teaching Professor
Computer Science Department



You see this Error. What do you do next?!

Talk to your Neighbor(s)!

```
mysql> CREATE DATABASE my_awesome_database
    -> CHARACTER SET utf8mb4
    -> COLLATE utf8mb4_unicode_ci;
```

```
ERROR 3680 (HY000): Failed to create schema directory
'my_awesome_database' (errno: 2 - No such file or directory)
```

Ask **search engines** (e.g., Google) and **AI tools** (e.g., Gemini, ChatGPT)

Software Construction Practice: Search For The Error Message

Problem

You see an **error message** from your framework, library, or external service **(not your own code)** that does not directly point you towards a solution.

Context

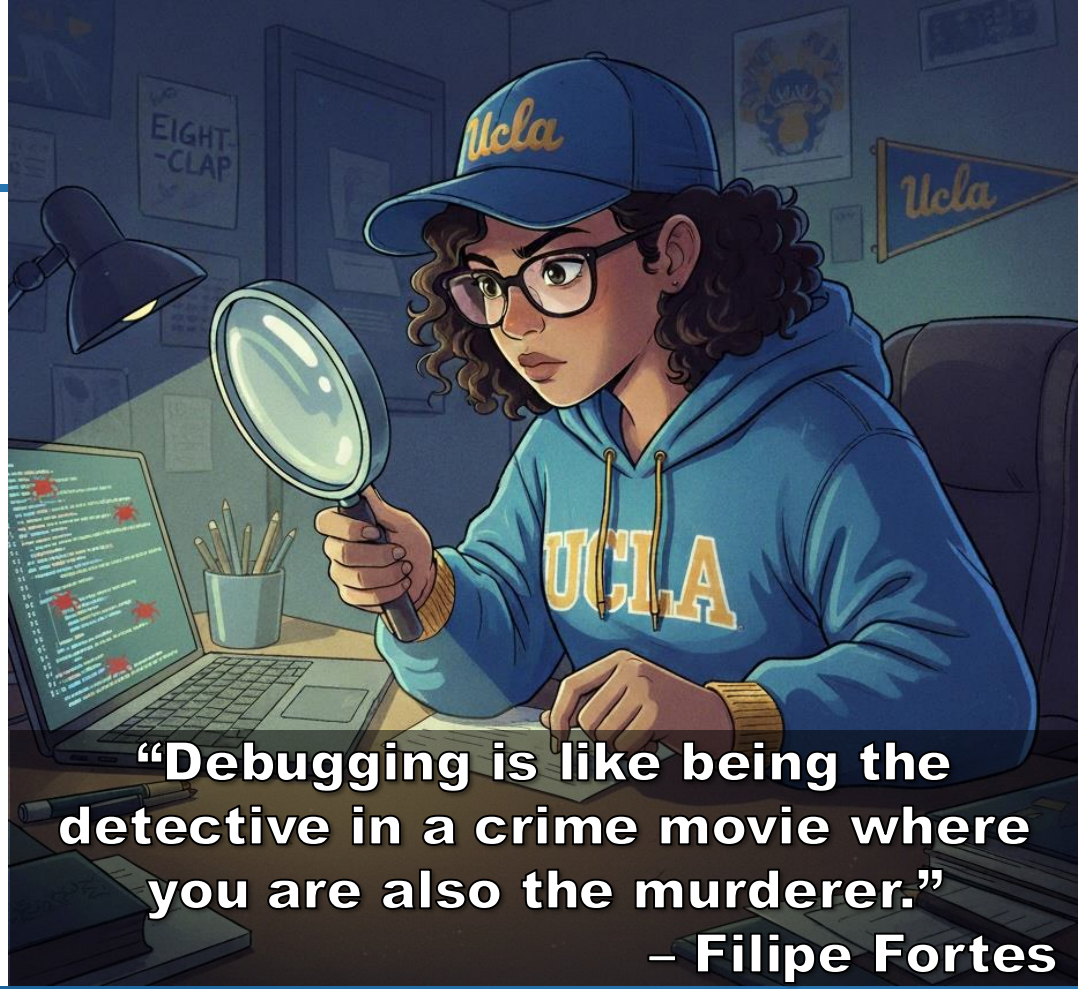
As a **human developer**, you do not have infinite knowledge & experience. Nobody expects you to understand every error message.

Solution

1. **Remove** all **project-specific** identifiers from you're the input & output
2. **Copy** & **paste** input & output into search engine / AI tool
3. Carefully **study** the results, (cautiously) experiment with suggested solutions
4. (Only) after you cannot get any more useful information from online sources, **ask** your more experienced co-worker

What is Debugging?

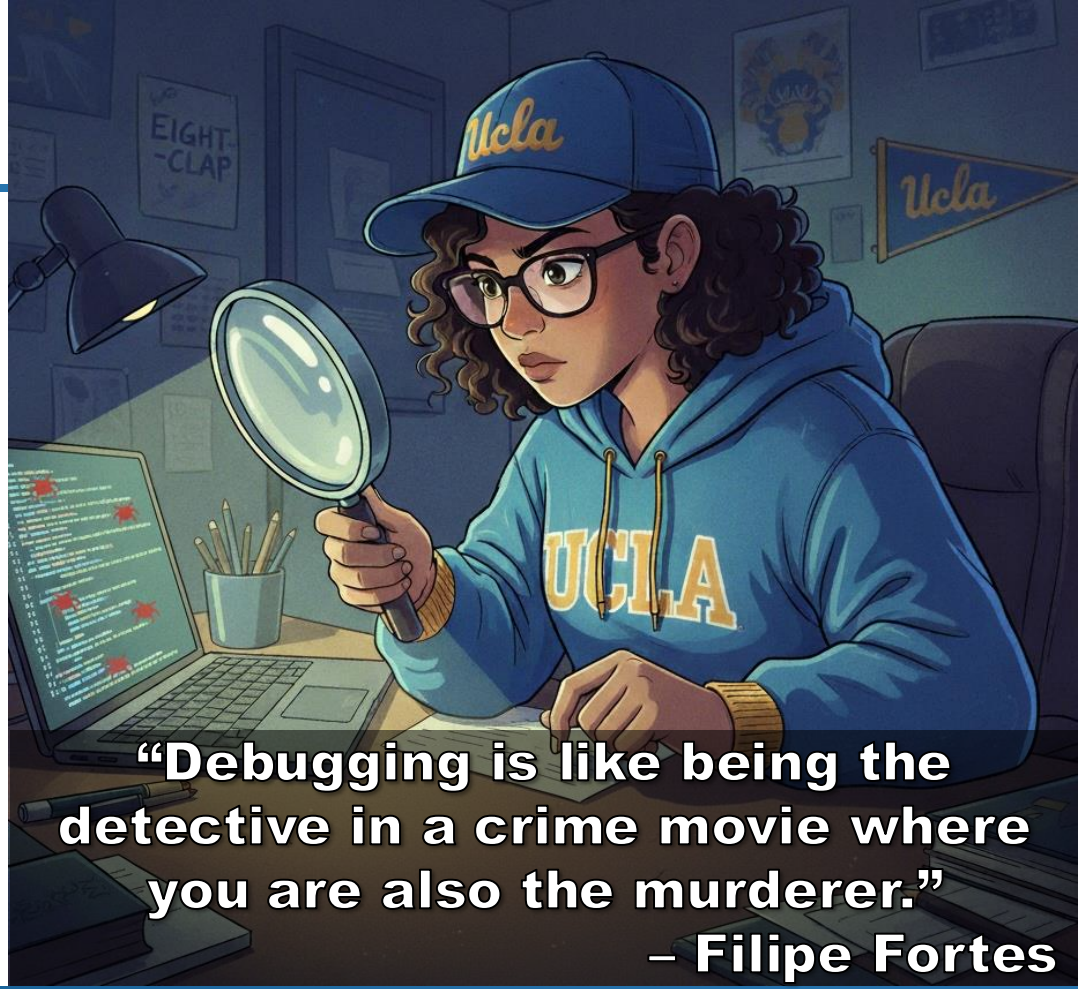
- The **systematic process** of **finding** & **fixing faults**, (aka. “bugs”) in a program's source code
  - 1) **Investigating** symptoms to Reproduce the Bug
  - 2) **Locating** the faulty code
  - 3) Determining the **root cause** of the bug
  - 4) Implementing and verifying a **fix**



Do Debugging Skills Matter?

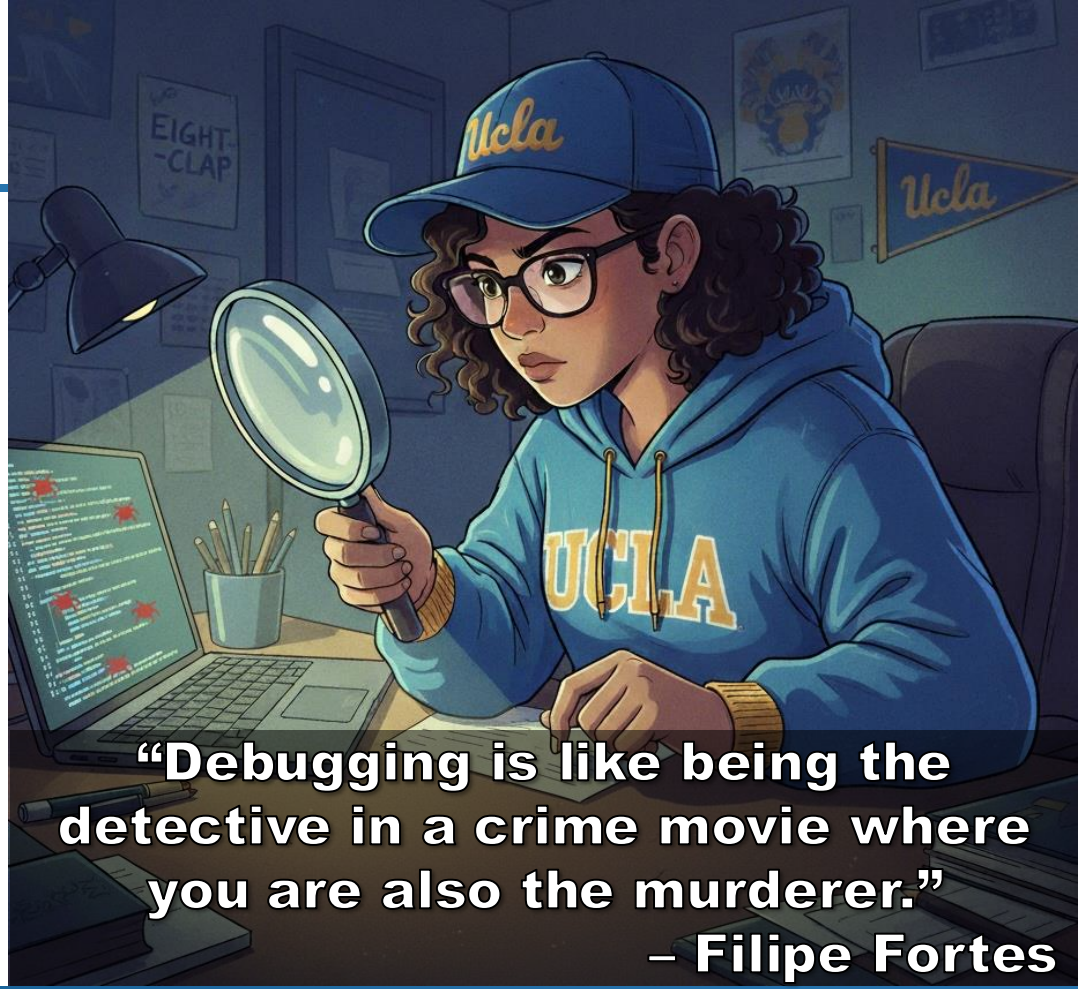
- Software bugs cost ≈ **60 billion USD / year**
- Validation (including debugging) can take up to **50-75% of development time**
- When debugging, some developers are **three times as efficient as others**

Hopefully you!



What is Debugging?

- The **systematic process** of **finding** & **fixing faults**, (aka. “bugs”) in a program's source code
  - 1) **Investigating** symptoms to Reproduce the Bug
  - 2) **Locating** the faulty code
  - 3) Determining the **root cause** of the bug
  - 4) Implementing and verifying a **fix**



Bug Terminology

Fault

The erroneous **location** in the code
(e.g., variable not initialized)

Program execution

Error

An **incorrect state** during program execution
(e.g., variable has the wrong value)

Error Reaches System Boundary

Failure

Observed incorrect **outside behavior**
(e.g., system crashes, incorrect output displayed)

How can we prevent this error from becoming a failure?

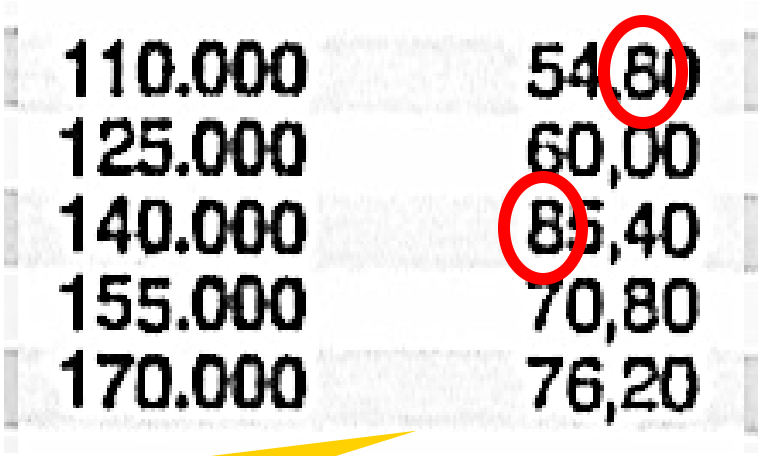
```
import sys
import math
def cal_circumference(radius):
  diameter = 2 * radius
  circumference = diameter * math.pi
  return circumference

def __main__():
  try:
    input_radius = sys.argv[1]
    C = cal_circumference(input_radius)
    print(f"The circumference of a circle
    with radius {input_radius} is: {C}")
  except:
    print("An error occurred
          but there is no failure")

__main__()
```

How can we Debug these Xerox Scans?

Document:

110.000	54,60
125.000	60,00
140.000	65,40
155.000	70,80
170.000	76,20

Scan:



Are we looking at a fault, error, or failure?

Source: https://www.dkriesel.com/en/blog/2013/0802_xerox-workcentres_are_switching_written_numbers_when_scanning

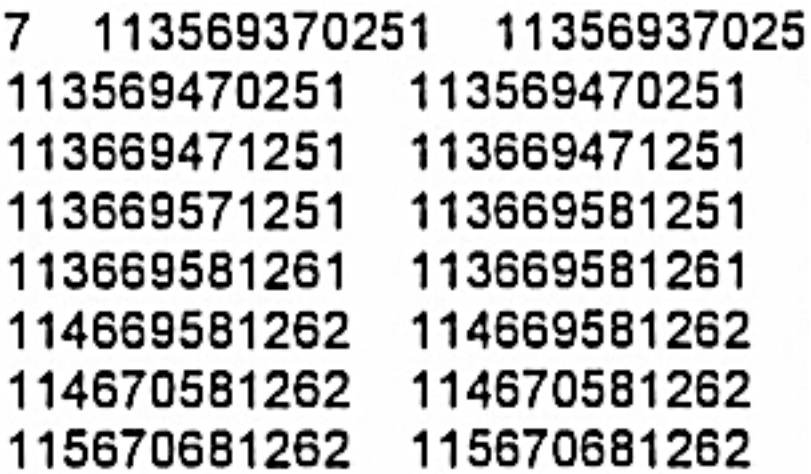
Step 1: Reproduce the Bug

How to reproduce the bug? What should we ask the customer?

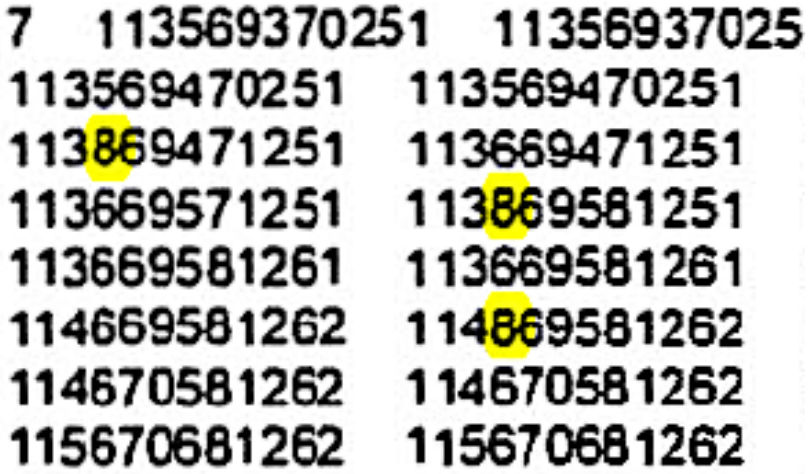
Goal / Motivation

Reproduce the bug to be able to observe it and check whether you've fixed it

Document:



Scan:



Source: https://www.dkriesel.com/en/blog/2013/0802_xerox-workcentres_are_switching_written_numbers_when_scanning

Reproduce the Bug & Write a Bug Report

Reproduce the *problem environment*

- Problem environment = **setting in which the problem occurs**
- **Configuration of the machine** (e.g., hardware, operating system, settings, run-time dependencies, software versions, …)
- Try to **re-create the problem environment** on a different machine

Reproduce the *problem history*

- Problem history = **steps necessary to re-create the problem**
- Record a sequence of **data inputs**, **user interactions**, and **communications** with other components
- Additional variables: timing, randomness, physical influences, …

See "Why Programs Fail – A Guide to Systematic Debugging" by Andreas Zeller 2009

If Possible, Write an Automated Bug Reproduction Test

Goal / Motivation

Automated Tests allow you to **check faster whether you've fixed the bug**

Write a Bug Reproduction Test

Automate the steps of reproducing the bug & checking if the bug is present.

Run your test several times during debugging

Simplify your Test Case

Find out what is **relevant** & **remove** all apparently **irrelevant steps** (e.g., details in the input that do not seem to be connected to the bug and that do not affect whether the failure is present or not)

What is Debugging?

- The **systematic process** of **finding** & **fixing faults**, (aka. “bugs”) in a program's source code
  - 1) **Investigating** symptoms to Reproduce the Bug
  - 2) **Locating** the faulty code
  - 3) Determining the **root cause** of the bug
  - 4) Implementing and verifying a **fix**

Investigate Symptoms Using Logs

- **Logging Libraries help recording symptoms**
  - Inputs (particularly unexpected ones)
  - State changes
  - Communication with other components

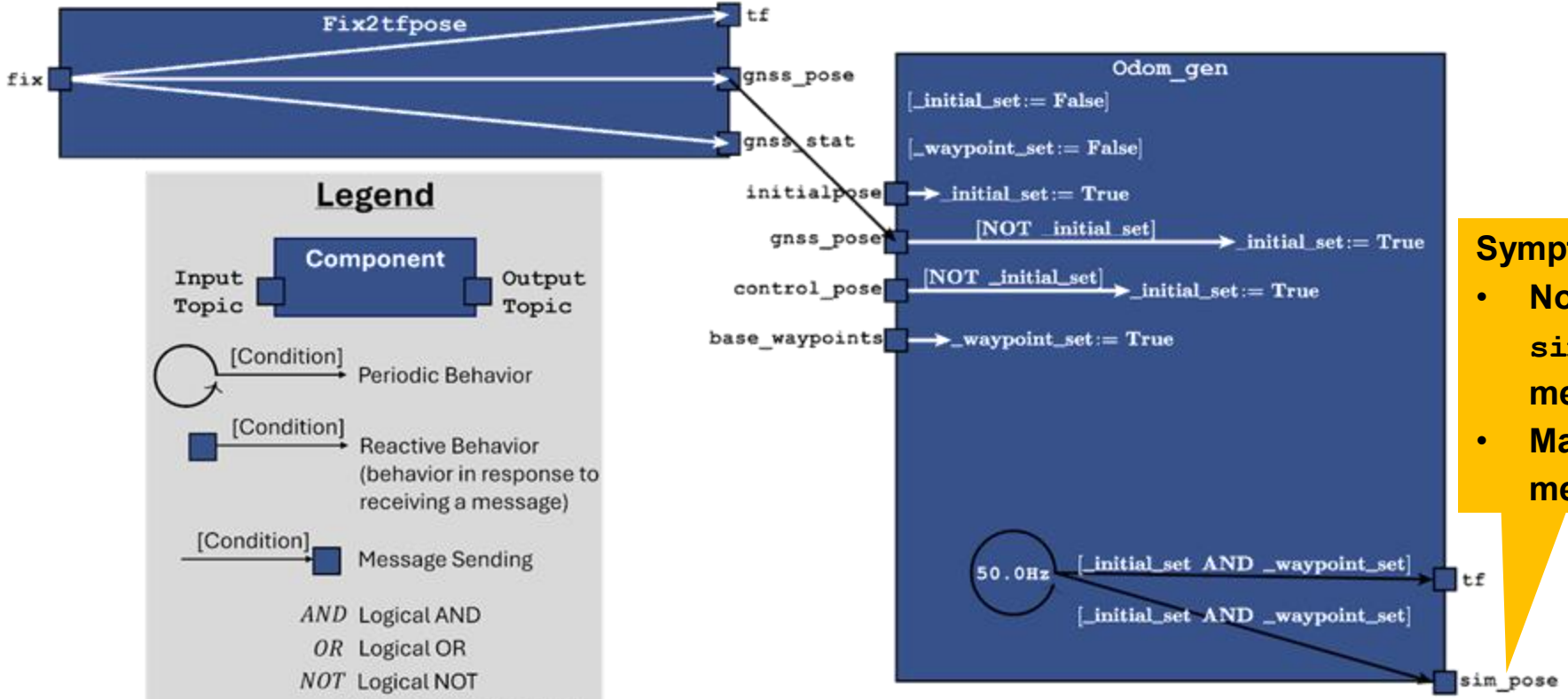
Python Logging

```
import logging

# Create a log format
fmt = logging.Formatter(
    "%(name)s: %(asctime)s | %(levelname)s |
%(filename)s:%(lineno)s | %(process)d >>>
%(message)s"
)

logging.debug("A debug message")
logging.info("An info message")
logging.warning("A warning message")
logging.error("An error message")
logging.critical("A critical message")
```

Output

```
example: 2023-07-23 14:42:18,599 | INFO | main.py:30 | 187901 >>> Server started listening
on port 8080
example: 2023-07-23 14:14:47,578 | WARNING | main.py:28 | 143936 >>> Disk space on drive
'/var/log' is running low. Consider freeing up space
example: 2023-07-23 14:14:47,578 | ERROR | main.py:34 | 143936 >>> Failed to connect to
database: 'my_db'
Traceback (most recent call last):
  File "/home/ayo/dev/betterstack/demo/python-logging/main.py", line 32, in <module>
    raise Exception("Failed to connect to database: 'my_db'")
Exception: Failed to connect to database: 'my_db'
```

Use Visual Diagrams to Locate Bugs



Symptoms:

- No `sim_pose` messages
- Many `fix` messages

Focus on the Most Likely Origins

Code Smells

Bugs are more likely to be in **poorly written.**

Get rid of code smells via refactoring before debugging

Look for common bugs

e.g., uninitialized variables, unused values, unreachable code, memory leaks, interface misuse, null pointers, inconsistent types, …

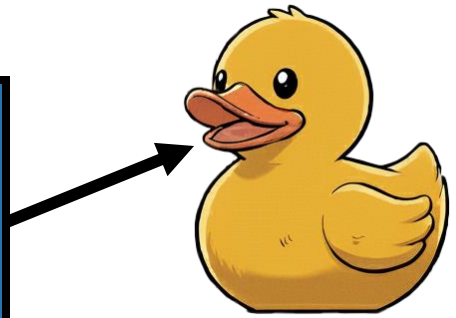
Assertions

Add assertions to your code to **prevent errors from propagating**
(e.g., `assert input > 0`, `assert file != null`)

What is Debugging?

- The **systematic process** of **finding** & **fixing faults**, (aka. “bugs”) in a program's source code
  - 1) **Investigating** symptoms to Reproduce the Bug
  - 2) **Locating** the faulty code
  - 3) Determining the **root cause** of the bug
  - 4) Implementing and verifying a **fix**

Your most valuable root cause analysis tool



Rubber Duck Debugging Can Help you Find Bugs

Curse of Knowledge

Having a **mental model** of your solution, you read what you ***intended*** to write, not what you ***actually*** **wrote**.

Explain Your Code to Your Duck

- Place a **rubber dock on your desk.**
- Explain to the duck **what your code is supposed to do, line by line**.
- At some point, you will tell the duck what your code should be doing next & **realize** that this is **not** what it **actually does**.

Use Break Points For Debugging

Goal

You want to observe the execution of the program step by step and see variable values

Break Point

An intentional **stopping point**

How it works

Whenever the program execution reaches the break point, it stops and opens interactive control over the program to step through and watch

```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16           but there is no failure")
17
18 __main__()
```

See https://code.visualstudio.com/docs/debugtest/debugging

Define Inputs & Run Configuration

Debug Configuration in VS Code

launch.json

```
"version": "0.2.0",
"configurations": [
 {
  "name": "Python Debugger:
           Current File",
  "type": "debugpy",
  "request": "launch",
  "args": ["10"],
  "program": "${file}",
  "console": "integratedTerminal"
 }
]
```

inputs

```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16            but there is no failure")
17
18 __main__()
```

See https://code.visualstudio.com/docs/debugtest/debugging

Observe Program Step by Step

Indicates where we are in the program execution

Allows you to evaluate expressions in the scope of the program

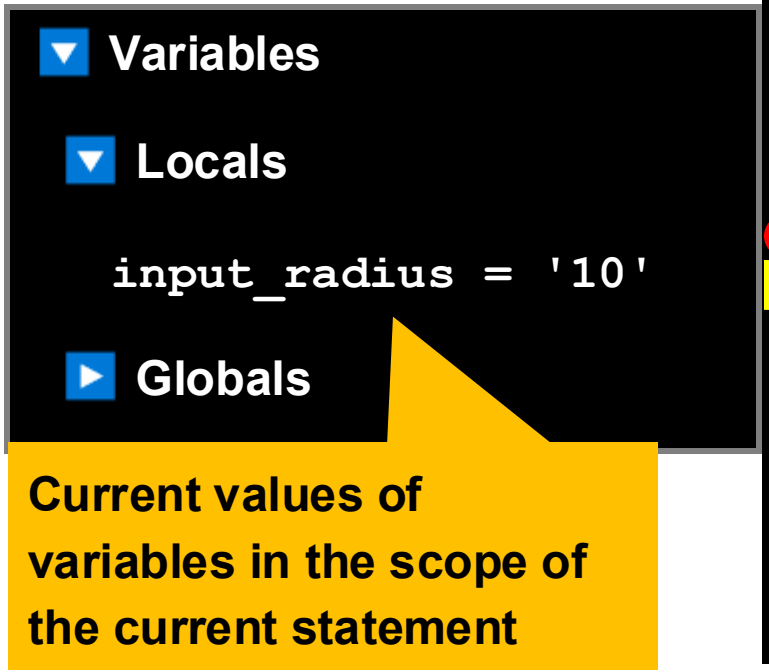
Debug Console

```
→ sys.argv[1]
  '10'
→ input_radius
  NameError: name
  'input_radius'
  is not defined
```

```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16           but there is no failure")
17
18 __main__()
```

See https://code.visualstudio.com/docs/debugtest/debugging

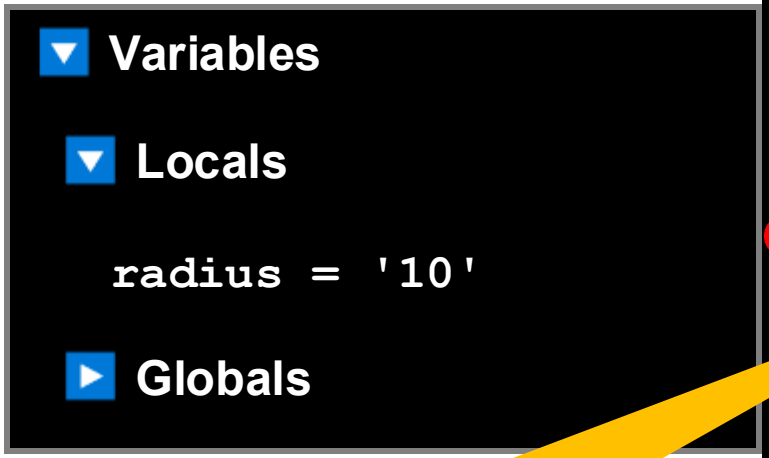
Observe Program Step by Step




```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16           but there is no failure")
17
18 __main__()
```

See https://code.visualstudio.com/docs/debugtest/debugging

Observe Program Step by Step

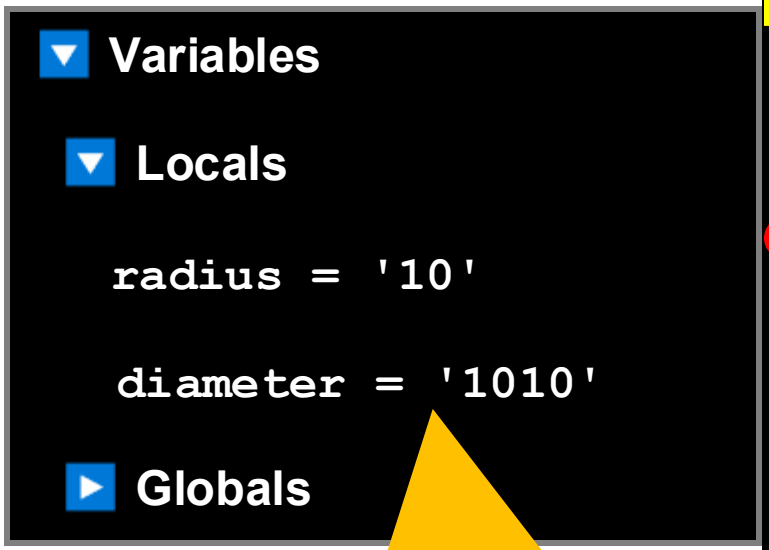


We have stepped into this Function

```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16           but there is no failure")
17
18 __main__()
```

See https://code.visualstudio.com/docs/debugtest/debugging

Observe Program Step by Step

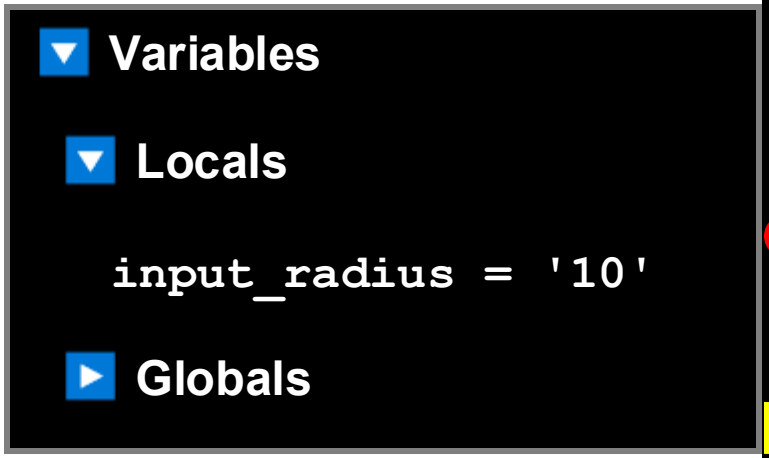


New variables appear as they are assigned

```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16           but there is no failure")
17
18 __main__()
```

See https://code.visualstudio.com/docs/debugtest/debugging

Observe Program Step by Step




```
1  import sys
2  import math
3  def cal_circumference(radius):
4    diameter = 2 * radius
5    circumference = diameter * math.pi
6    return circumference
7
8  def __main__():
9    try:
10     input_radius = sys.argv[1]
11     C = cal_circumference(input_radius)
12     print(f"The circumference of a circle
13     with radius {input_radius} is: {C}")
14   except:
15     print("An error occurred
16           but there is no failure")
17
18 __main__()
```

Exception handling is triggered by multiplying a string with a float

See https://code.visualstudio.com/docs/debugtest/debugging

Conditional Break Points – Skip Directly to the Execution You Want to Observer

Goal

You want to observe the program only **if certain conditions are true** (e.g., the input is very large or the input contains a certain character, or the loop variable has reached 10).

Conditional Break Points

Break points that trigger only when a **given expression evaluates to true**.

How it works

Right-click on the break point to edit it. The expression can include functions and variables that exist in the code of this statement.

See https://code.visualstudio.com/docs/debugtest/debugging

What is Debugging?

- The **systematic process** of **finding** & **fixing faults**, (aka. “bugs”) in a program's source code
  - 1) **Investigating** symptoms to Reproduce the Bug
  - 2) **Locating** the faulty code
  - 3) Determining the **root cause** of the bug
  - 4) Implementing and verifying a **fix**

Document The Bug Fix & Run Your Tests

Add Assertions

After fixing the bug, **add assertions** to **detect nearby bugs & run tests**

Document your Bug Fix

In the code, explain **why your fix was necessary** in a **comment**

Reference the bug in your **git commit message**

Document your solution in the bug report to allow **historical traceability**

After Fixing, **Keep all new Tests** for **Regression Testing**

Regression Testing = **re-running existing tests** after code changes to ensure that new updates, bug fixes, or features haven't introduced new problems into existing functionality

UCLA Samueli School of Engineering

CS 35L Software Construction

Lecture 10 – Advanced Git Techniques

Tobias Dürschmid
Assistant Teaching Professor
Computer Science Department

Rapid Fire Quiz on Git

What is `HEAD`?

The pointer to the **currently selected commit / branch**

What is the staging area?

Code that is about to be committed in the next commit

What is a branch

A pointer to the most recent commit of a sequence of separately developed code

What is a merge conflict

The attempted merge of lines in the code that have been changed by both versions

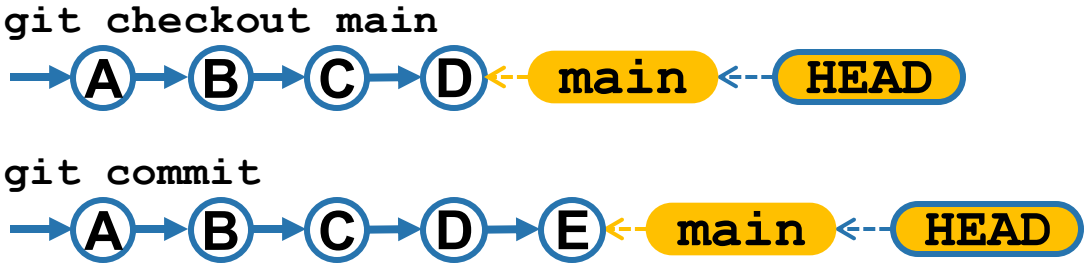
What makes a merge commit different?

It has two parents

Detached HEAD

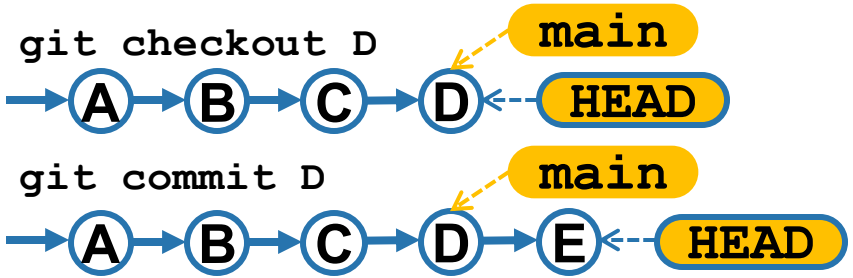
Non-Detached HEAD

`HEAD` is attached to a **branch.** Most Git commands, such as **`git commit`**, will move the branch & **`HEAD`.**



Detached HEAD

`HEAD` is attached to a commit, not a **branch.** Git commands, will NOT move the branch.



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

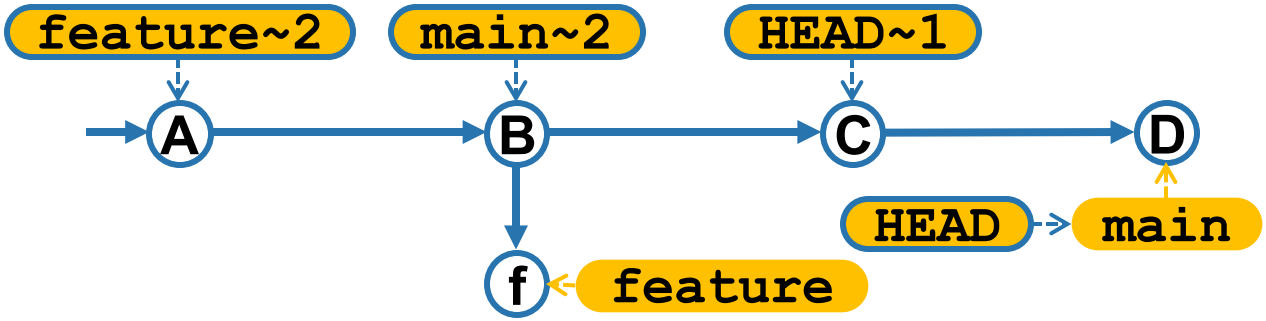
Relative Commit Addresses

Goal

You want to describe the **parent commit** (or grand parent commit, or great grand parent commit) of a commit **without having to get the commit ID from the log.**

How it works

`BRNACH~n` is the n-th commit before `BRANCH`.



See https://www.geeksforgeeks.org/git/git-cherry-pick/

Git Cherry-Pick

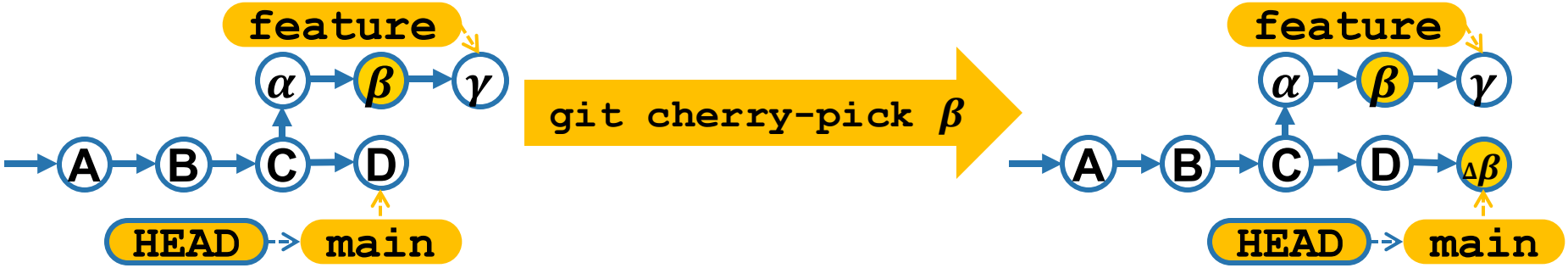
Goal

You want to selectively **"copy" one particular commit** from one branch to another branch without bringing over the entire history.

How it works

`git cherry-pick` β calculates the difference (patch) for commit β **and creates a new commit** that **applies the same patch** to the current base.

Note: This may result in **merge conflicts** if the patch cannot be applied directly.



See https://www.geeksforgeeks.org/git/git-stash/

Git Stash & Pop

Problem

You want to checkout a branch or commit while you have **uncommited changes**

Context

You might **need** your uncommited **changes later**

Solution

Use `git stash` to move your changes to a separate stashing location and use `git stash pop` to bring them back

How it works

Git maintains a stack of stashes changes. Every time you call git stash it adds these patches to the stashing stack. Pop removes them from the stack. `git stash list` shows you the entire stack. `git stash apply <id>` applies a non-top stash patch

`git blame` – Which Commit Last Changed a Dubious Line?

See https://www.geeksforgeeks.org/git/how-to-use-git-blame/

Goal

For a given line you want to identify **why it exists** / why it has been **implemented** the way it has, or **who has most recently changed this line**.

How it works

`git blame <filename>` shows you, for every line in the file:

- **Commit id** of the last commit that modified this line
- **Author** of that commit
- **Timestamp** of that commit

GitHub visualizes this very well

```
git blame .\my_stack.py
ae721935 (TD 10-25 15:04  1) class MyStack:
562ad4a6 (TD 10-25 15:06  2)     """
562ad4a6 (TD 10-25 15:06  3)     A simple implementation of a Stack data str
562ad4a6 (TD 10-25 15:06  4)
562ad4a6 (TD 10-25 15:06  5)     A Stack operates on the Last-In, First-Out
562ad4a6 (TD 10-25 15:06  6)     the last element added to the stack is the
562ad4a6 (TD 10-25 15:06  7)     """
562ad4a6 (TD 10-25 15:06  8)
6f33a948 (TD 10-25 18:58  9)     class PopOnEmptyStackException(Exception):
6f33a948 (TD 10-25 18:58 10)         """
6f33a948 (TD 10-25 18:58 11)         Exception raised when trying to call po
6f33a948 (TD 10-25 18:58 12)         """
6f33a948 (TD 10-25 18:58 13)         def __init__(self, message="Cannot pop
6f33a948 (TD 10-25 18:58 14)             super().__init__(message)
6f33a948 (TD 10-25 18:58 15)
07768051 (TD 10-25 16:29 16)     def __init__(self):
47c50cfe (TD 10-25 16:44 17)         """
```

`git bisect` – Which Commit Introduced the Fault?

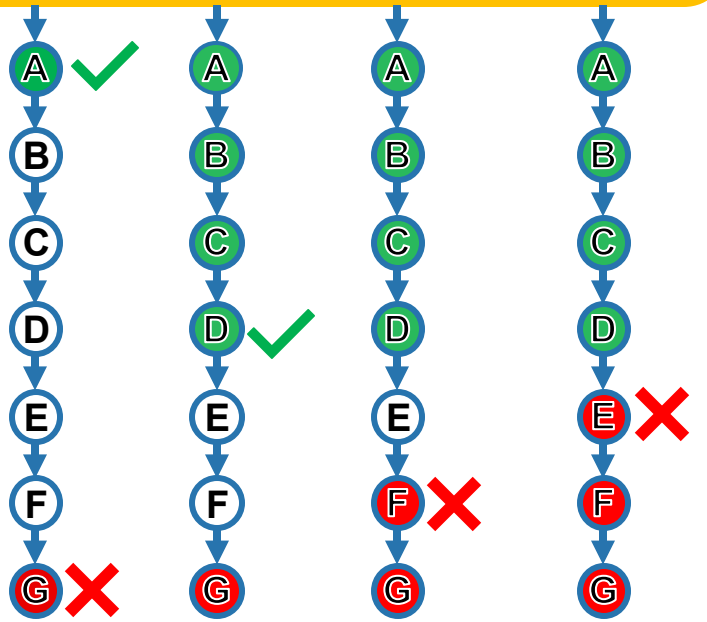
See https://www.geeksforgeeks.org/git/git-bisect/

Goal

Out of hundreds of commit, how to **find the one that broke the tests?**

How it works

1. Identify a past commit `c_good` that worked
2. Write a command `test` that tests whether a commit works and that **can run on past commits**
3. Run: `git bisect start c_good && git bisect run test`
4. Git will run a **binary search** and stop on the commit that **broke the tests**



`git revert` – The Undo Button For Changes in the Past

See https://www.geeksforgeeks.org/git/how-to-revert-a-commit-with-git-revert/

Goal

You want to "**undo" a commit** in the history of your branch (e.g., it introduced a bug)

How it works

`git revert B` calculates the difference (patch) for commit B and **creates a new commit** that **applies the inverse patch** to the current base. B is still in the commit history. The default commit message for the reverting commit is "Revert B".

Note: This may result in **merge conflicts** if the inverse patch cannot be applied directly.

This command is **non-destructive** (preserves the entire history).

`git rebase -i`
Rewrite Your Own History

See https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

Destructive! Be very careful

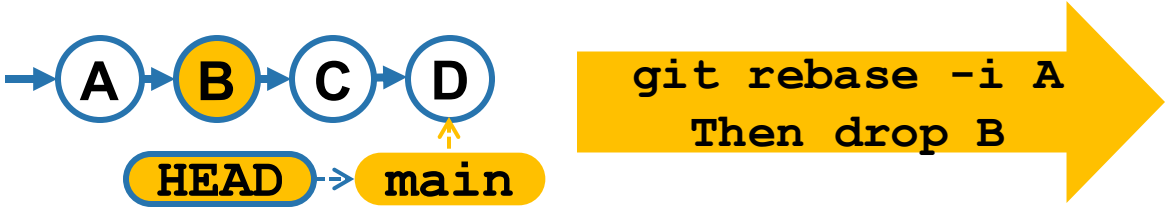
Goal

You want to "**undo" one or more commits** in **without introducing a revert commit.**

How it works

`git rebase -i A` rewrites the commit history starting from commit A. If changes are made (e.g., commits should be dropped or added within a break) git creates new commits with the same patches. This command is **destructive** (the result is **a new commit history** which may require **`push --force`** to update)

```
pick B "Let's see if they find my bug"
pick C "Harmless changes"
pick D "Implement magical feature"

# Rebase A..D onto A (3 commands)
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but
edit the commit message
# b, break = stop here (continue rebase
later with 'git rebase --continue')
# d, drop <commit> = remove commit
[…]
```



`git rebase -i`
Rewrite Your Own History

Data Loss Risk

If you get confused, mess up a step, or accidentally drop a commit you needed, you could **lose work**. Always ensure your changes are committed (or stashed) before starting a rebase.

The Safe Zone

When working **in teams**, only use **`git rebase -i`** on commits that exist **only on your local machine** or on a feature branch that **only you** are working on.

See https://www.geeksforgeeks.org/git/rebasing-of-branches-in-git/

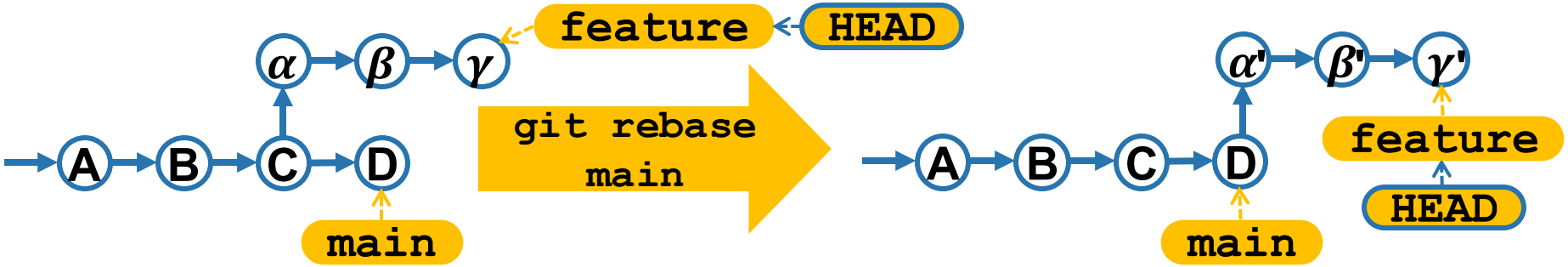
Simple Rebase – An Alternative to Merge

Goal

You want to merge a feature branch but instead of a merge commit with two parents you just want to **"copy" the individual commits**

How it works

`git rebase f` combines the patches from all commits on the feature branch into one instead of creating a merge commit with two parents



See https://www.geeksforgeeks.org/git/rebasing-of-branches-in-git/

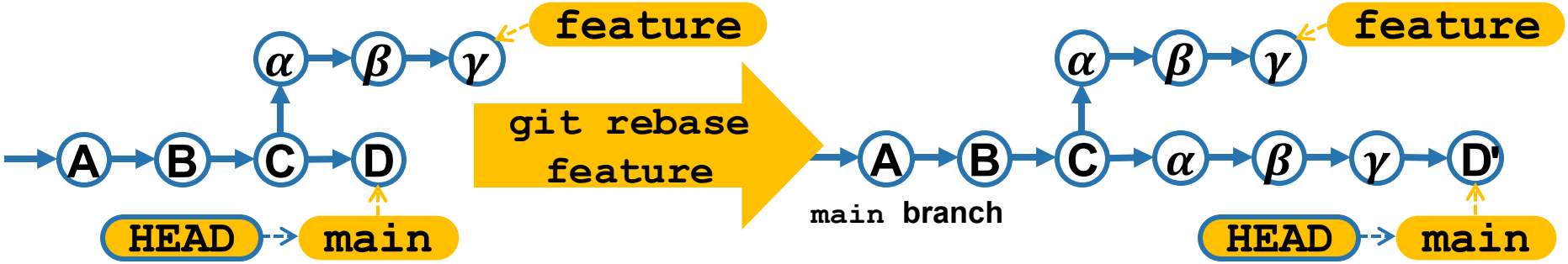
Simple Rebase – An Alternative to Merge

Goal

You want to merge a feature branch but instead of a merge commit with two parents you just want to **"copy" the individual commits**

How it works

`git rebase f` combines the patches from all commits on the feature branch into one instead of creating a merge commit with two parents



See https://www.geeksforgeeks.org/git/git-squash/

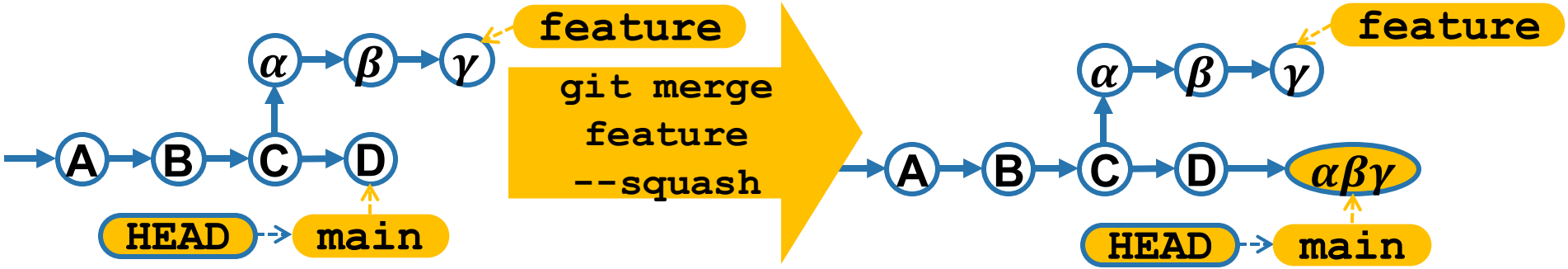
Squash Merge keeps your History Simple and Clean

Goal

You want to merge a feature branch but instead of keeping all individual commit you want **just one single commit for the new feature**

How it works

`git merge --squash` combines the patches from all commits on the feature branch into one instead of creating a merge commit with two parents



`Git Submodules` – Perfect for Large Projects with Multiple Repositories

See https://www.geeksforgeeks.org/git/git-submodule/

Goal

You want to include **external libraries** or **shared modules** within your project while maintaining their **history** and keeping them **separate from your main repository**.

How it works

Conceptually, git submodules keep a **Git repository as a subdirectory of another Git repository**. Internally, they are represented as a file that **points to the commit ID** that should be checked out in the submodule. You change the content of the submodule by committing this file (which has the same name as the directory).

Examples

https://github.com/cmu-rss-lab/rosdiscover-evaluation/tree/main/deps

https://github.com/UCLA-CS-35L/submodule_example_parent/

Common Submodule Commands (Yes, this Initially Confusing)

See https://www.geeksforgeeks.org/git/git-submodule/

Add a Submodule

`git submodule add <repo-url> <path>`

Clones `<repo-url>` into `<path>` and starts keeping track of it as a submodule

Clone a Repository that has Submodules

`git clone --recursive <repo-url>`

Clones `<repo-url>` and their transitive submodules and checks out the selected commit

Reset Submodule Content to the Selected Commit

`git submodule update`

Clones all submodule repos (and their submodules) and checks out the selected commit

Commit Content inside the Submodule

Commit the changes within the submodule folder & push

Change the directory to outside of the submodule, commit the submodule & push

Please fill out your Exit Tickets on Bruin Learn!

Question 1 — 1 pts

In your own words, please summarize **three key insights** you learned about **Debugging & Git** today. (Do not copy phrases from the lecture slides)

Question 2 — 1 pts

What is the difference between Testing & Debugging? Please describe the difference based on an example.

Question 3

Please leave any questions that you have about today's material and things that are still unclear or confusing to you (if none, simply write N/A)

Credits: These slide use images from Flaticon.com (Creators: Freepik)

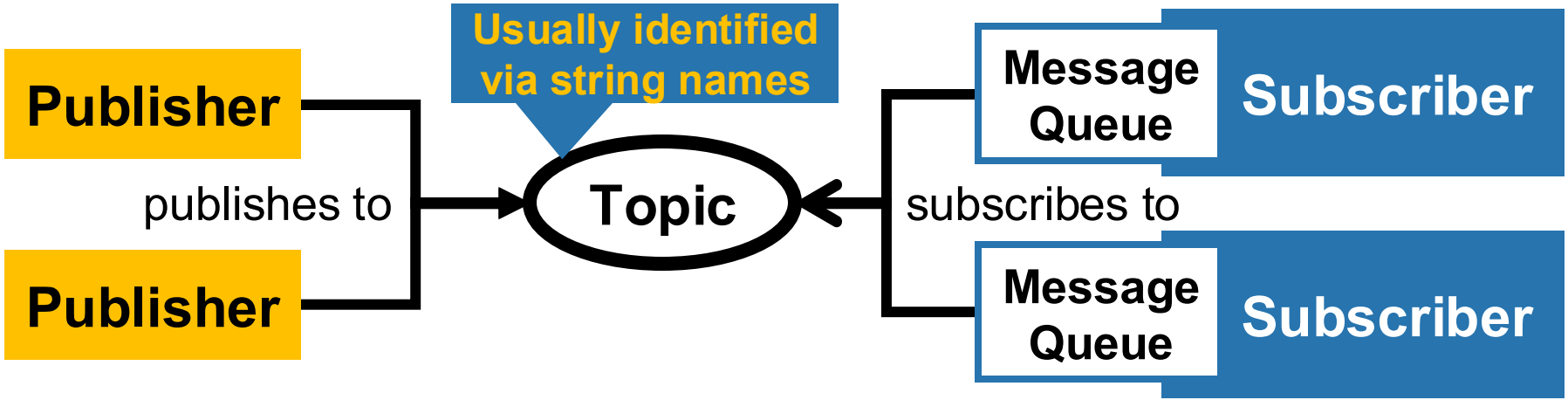
Publish-Subscribe implements an N-to-M Messaging Channel

Problem

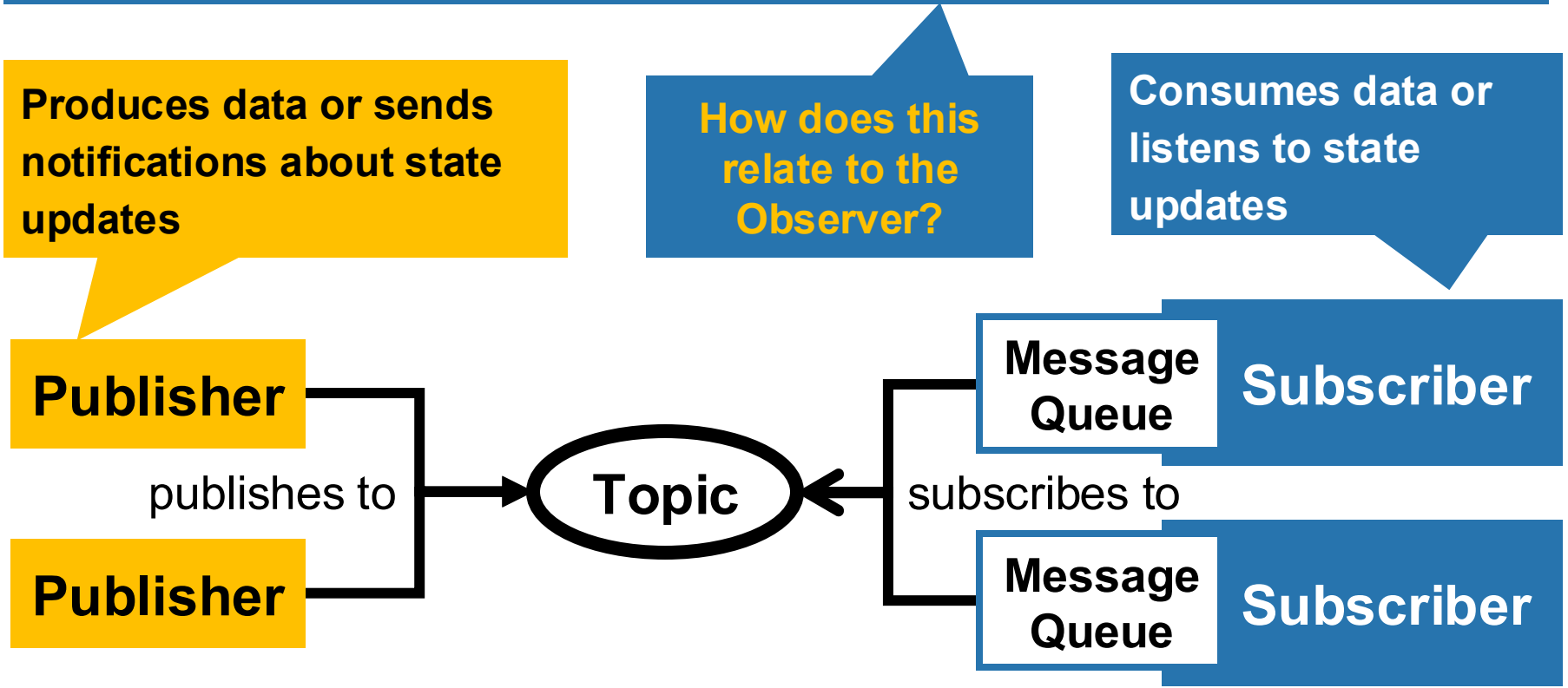
How to keep the **state** of **cooperating components synchronized**?

Context

Components should be **loosely coupled and reduce direct dependencies**



Publish-Subscribe implements an N-to-M Messaging Channel



Publish-Subscribe implements an N-to-M Messaging Channel

```
pub = nh.advertise("topic_name");
…
pub.publish(msg)
```

```
nh.subscribe("topic_name", 10, callback);
```