Version Control

Martin Kellogg

Testing (part 3)

Today's agenda:

- Reading Quiz
- Finish up slides from last lecture
- Test input generation (**fuzzing**)
- Test oracle generation
- Test prioritization & test suite minimization

This is how far we got on 9/20/24. Before we cover today's topic (version control), we're going to finish this.

Test input generation

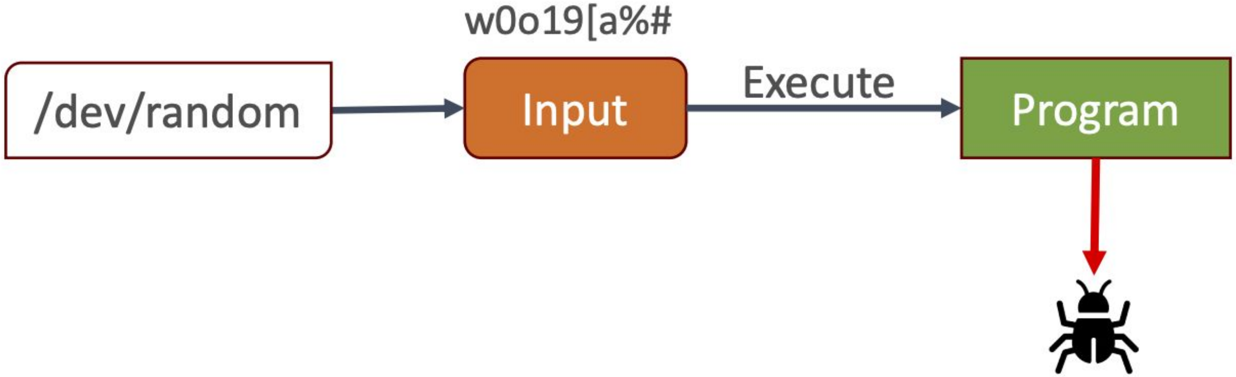
- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Logic**: choose inputs that will maximize coverage
  - Lens of **Statistics**: choose inputs "at random"
  - Lens of **Adversity**: choose inputs that kill mutants

Lens of Statistics: fuzzing and random testing

Key idea: provide inputs “at random” to the program and use an implicit oracle

Lens of Statistics: fuzzing and random testing

Key idea: provide inputs “at random” to the program and use an implicit oracle



Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
- totally random input rarely works well

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
- totally random input rarely works well
    - most programs have **structured input**

Lens of Statistics: fuzzing and random testing

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
- totally random input rarely works well
  - most programs have **structured input**
  - so modern fuzzers use some kind of **semi-random, directed search**

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
    - start with a ***seed pool*** of valid or useful inputs

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
    - start with a ***seed pool*** of valid or useful inputs
    - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:
    - when an input **increases coverage** (or some other test goal), choose more inputs like that (e.g., add it to the seed pool)

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:
  - when an input **increases coverage** (or some other test goal), choose more inputs like that (e.g., add it to the seed pool)
- **combination with path predicates**:

Lens of Statistics: fuzzing: input structure

Modern fuzzers deal with structured input in a few ways:

- **mutating seed inputs**:
  - start with a ***seed pool*** of valid or useful inputs
  - new test cases are **evolved** from old ones
- **reward** or **fitness functions**:
  - when an input **increases coverage** (or some other test goal), choose more inputs like that (e.g., add it to the seed pool)
- **combination with path predicates**:
  - add inputs that are guaranteed to increase coverage to the seed pool

Lens of Statistics: fuzzing in practice

Lens of Statistics: fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources

Lens of Statistics: fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources
- Fuzzing is **machine-intensive**
  - most inputs aren't useful

Lens of Statistics: fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources
- Fuzzing is **machine-intensive**
  - most inputs aren't useful
- Fuzzing **finds real bugs**
  - especially useful for finding security bugs

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Logic**: choose inputs that will maximize coverage
  - Lens of **Statistics**: choose inputs "at random"
  - Lens of **Adversity**: choose inputs that kill mutants

Lens of Adversity: killing mutants

- Actually, **not as useful as it seems** for automatic test generation
    - still need to use either path predicates or fuzzing to choose inputs

Lens of Adversity: killing mutants

- Actually, **not as useful as it seems** for automatic test generation
  - still need to use either path predicates or fuzzing to choose inputs
- Can be a useful **fitness function** or guide for other automated test input generation approaches

Testing (part 3)

Today's agenda:

- Reading Quiz
- Finish up slides from last lecture
- Test input generation (fuzzing)
- Test oracle generation
- **Test prioritization & test suite minimization**

Too many tests

- At this point, we may actually have **too many** test cases

Too many tests

- At this point, we may actually have **too many** test cases
    - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools
  - Which many produce many tests but lower-quality ones than humans would produce

Too many tests

- At this point, we may actually have **too many** test cases
  - Surprisingly, this is **normal in industry**: you almost always have far too few or far too many!
- This is especially true when using automated test generation tools
  - Which many produce many tests but **lower-quality** ones than humans would produce
  - A **big cost problem**!

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1, 6

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1, 6

Which of these tests would you pick to minimize the number that need to be run?

Test suite minimization

Definition: given a set of test cases and coverage information for each one, the ***test suite minimization problem*** is to find the minimal number of test cases that still have the maximum coverage.

Example:

- ~~T1 covers lines 1,2,3~~
- T2 covers lines 2,3,4,5
- ~~T3 covers lines 1,2~~
- T4 covers lines 1, 6

Which of these tests would you pick to minimize the number that need to be run?

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?
    - theory strikes again!

Test suite prioritization

Definition: given a budget of time, number of tests to run, or similar, the ***test suite prioritization problem*** is deciding which tests to run to maximize coverage while staying within the budget

- very similar to test suite minimization (same techniques are useful for both)
- **question**: how **hard** are these problems?
  - theory strikes again!
  - answer: it's "hard" (similar "traditional" problem that you might consider a reduction to: **knapsack**)

Version Control

Today's agenda:

- **Reading Quiz**
- How does a version control system work?
- How to use your VCS
- GitHub workflows

Reading Quiz: version control

Q1: When does the author of “Version control concepts and best practices” argue it is appropriate to rewrite history?

A. when you have committed something unintentionally
B. when you have committed a generated file
C. when you want to avoid a merge conflict
D. it is never appropriate to rewrite history

Q2: **TRUE** or **FALSE**: The author of “My favourite Git commit” calls it his favourite because of the technical, complex code in the commit.

Reading Quiz: version control

Q1: When does the author of “Version control concepts and best practices” argue it is appropriate to rewrite history?

A. when you have committed something unintentionally
B. when you have committed a generated file
C. when you want to avoid a merge conflict
D. it is never appropriate to rewrite history

Q2: **TRUE** or **FALSE**: The author of “My favourite Git commit” calls it his favourite because of the technical, complex code in the commit.

Reading Quiz: version control

Q1: When does the author of “Version control concepts and best practices” argue it is appropriate to rewrite history?

A. when you have committed something unintentionally
B. when you have committed a generated file
C. when you want to avoid a merge conflict
D. it is never appropriate to rewrite history

Q2: **TRUE** or **FALSE**: The author of “My favourite Git commit” calls it his favourite because of the technical, complex code in the commit.

Version Control

Today’s agenda:

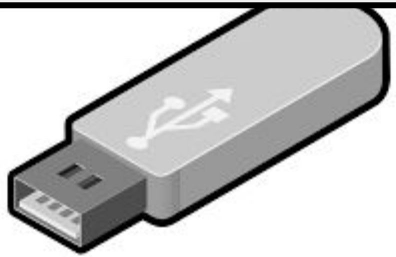
- Reading Quiz
- **How does a version control system work?**
- How to use your VCS
- GitHub workflows

Let’s share a file

Let’s share a file

Let’s share a file

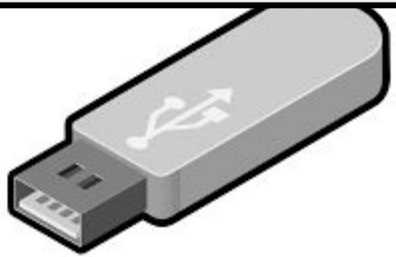
These systems are fine for “**binary blobs**”: files that you don’t intend to change once shared



Let's share a file

These systems are fine for "**binary blobs**": files that you don't intend to change once shared

- **but not for code**



Goals of version control

Goals of version control

- Keep a **history** of your work
  - Explain the purpose of each change
  - Checkpoint specific versions (known good state)
  - Recover specific state (fix bugs, test old versions)

Goals of version control

- Keep a **history** of your work
  - Explain the purpose of each change
  - Checkpoint specific versions (known good state)
  - Recover specific state (fix bugs, test old versions)
- **Coordinate**/merge work between team members
  - Or yourself, on multiple computers, or multiple features

What is version control

Definition: a ***version control system*** is a program that manages many versions of one or more text-based documents by storing diffs between them

What is version control

Definition: a ***version control system*** is a program that manages many versions of one or more text-based documents by storing diffs between them

- can be either ***centralized*** or ***distributed***

What is version control

Definition: a ***version control system*** is a program that manages many versions of one or more text-based documents by storing diffs between them

- can be either ***centralized*** or ***distributed***

one main repository, many
remotes with working copies

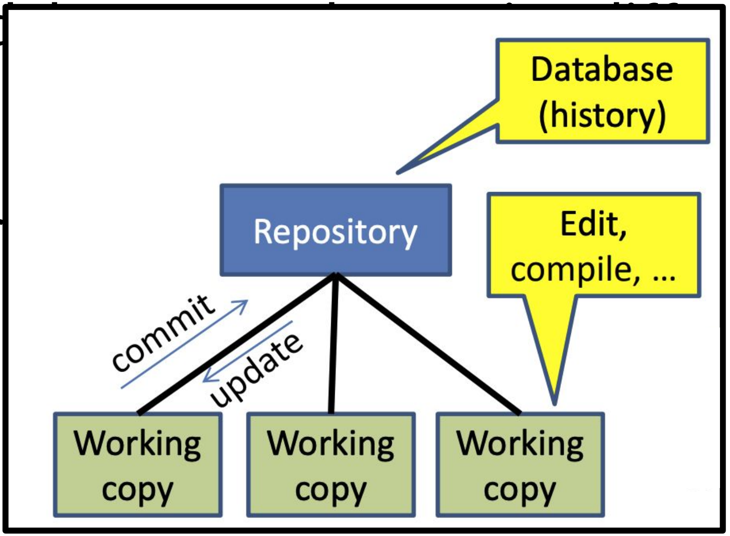
What is version control

Definition: a ***version control system*** is a program that manages many versions of one or more text-based

between them

- can be either ***centralized*** or

one main repository, many remotes with working copies



What is version control

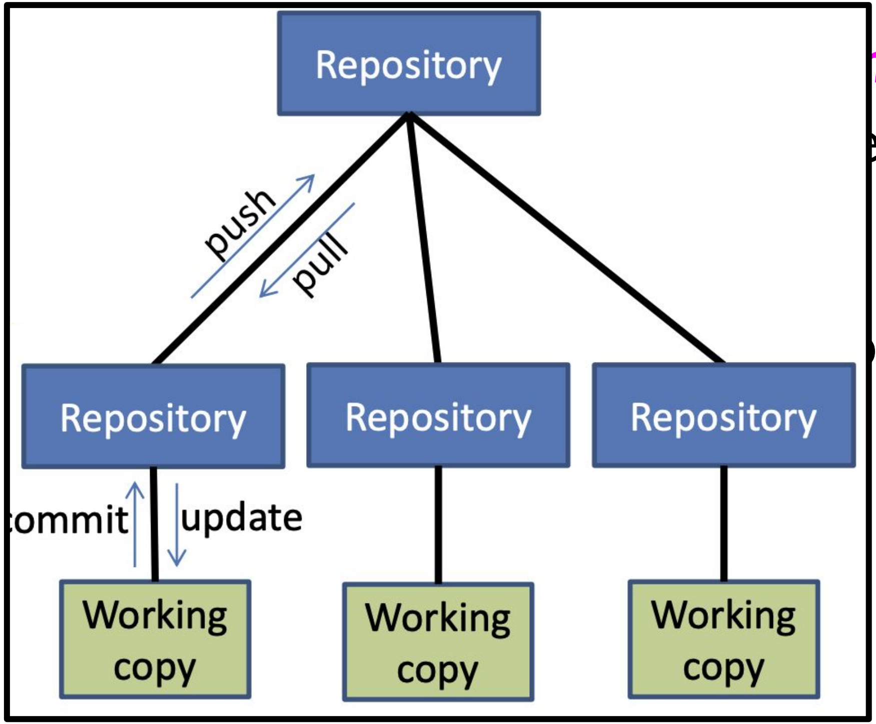
Definition: a ***version control system*** is a program that manages many versions of one or more text-based documents by storing diffs between them

- can be either ***centralized*** or ***distributed***

one main repository, many remotes with working copies

many repositories, each repository has a working copy

What is version control

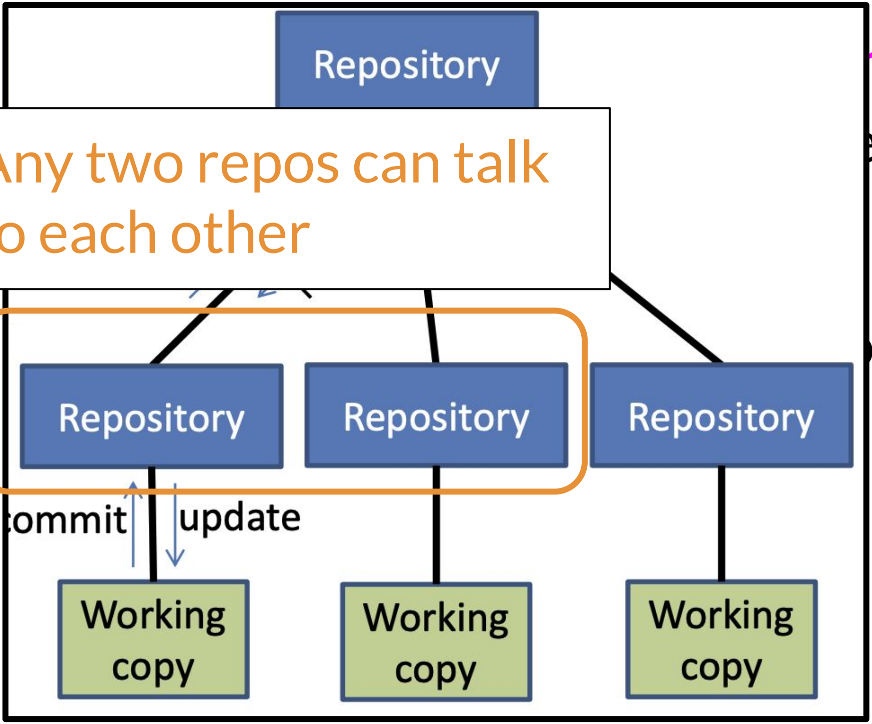


m is a program that manages many
ed documents by storing diffs

r ***distributed***

many repositories, each
repository has a working copy

What is version control



Any two repos can talk to each other

m is a program that manages many
ed documents by storing diffs

r *distributed*

many repositories, each repository has a working copy

What is version control

Definition: a ***version control system*** is a program that manages many versions of one or more text-based documents by storing diffs between them

- can be either ***centralized*** or ***distributed***

one main repository, many remotes with working copies

many repositories, each repository has a working copy

typical setup: distributed VCS with a single, privileged main

Advantages of distributed VCS

- checkpoint work without publishing to teammates
- commit, examine history when not connected to the network
- more accurate history
- more effective merging algorithms

Advantages of distributed VCS

- checkpoint work without publishing to teammates
- commit, examine history when not connected to the network
- more accurate history
- more effective merging algorithms

Less important in CS 490:

- share changes selectively with teammates
- flexibility in repository organization and workflow
- faster performance

Advantages of distributed VCS

- checkpoint work without publishing to teammates
- commit, examine history when not connected to the network
- more accurate history
- more effective merging algorithms

Less important in CS 490:

- share changes selectively with teamm
- flexibility in repository organization a
- faster performance

Distributed VCS is now the **industry standard** (e.g., git, hg). (Some organizations do still use centralized, though.)

Distributed VCS prevents some operations

- No update if uncommitted changes exist: must commit first
- No push if not ahead of remote: must pull & merge first
- No partial update (e.g., updating just one directory)
  - update gets all changes in a changeset (= a commit)

Distributed VCS prevents some operations

- No update if uncommitted changes exist: must commit first
- No push if not ahead of remote: must pull & merge first
- **No partial update** (e.g., updating just one directory)
  - update gets all changes in a changeset (= a commit)

Why might this be a problem in a large company?

Monorepos

Distributed VCS prevents some operations

- No update if uncommitted changes exist: must commit first
- No push if not ahead of remote: must pull & merge first
- **No partial update** (e.g., updating just one directory)
  - update gets all changes in a changeset (= a commit)

Why might this be a problem in a large company?
Monorepos

Distributed VCS prevents some operations

- No update if uncommitted changes exist: must commit first
- No push if not ahead of remote: must pull & merge first
- No partial update (e.g., updating just one directory)
  - update gets all changes in a changeset (= a commit)
- Rationale:
  - Maintain more accurate, complete history
  - Keep all users in sync
  - Avoid painful conflicts
  - Avoid loss of work

Coordinating with others

- `pull` incorporates others' changes into your repository
  - (update brings changes into your working copy)
  - (N.b.: `git pull` does pull, merge, and update)

Coordinating with others

- `pull` incorporates others' changes into your repository
  - (update brings changes into your working copy)
  - (N.b.: `git pull` does pull, merge, and update)
- If you are **behind**, nothing more to do
  - Behind = your history is a **prefix** of master history

Coordinating with others

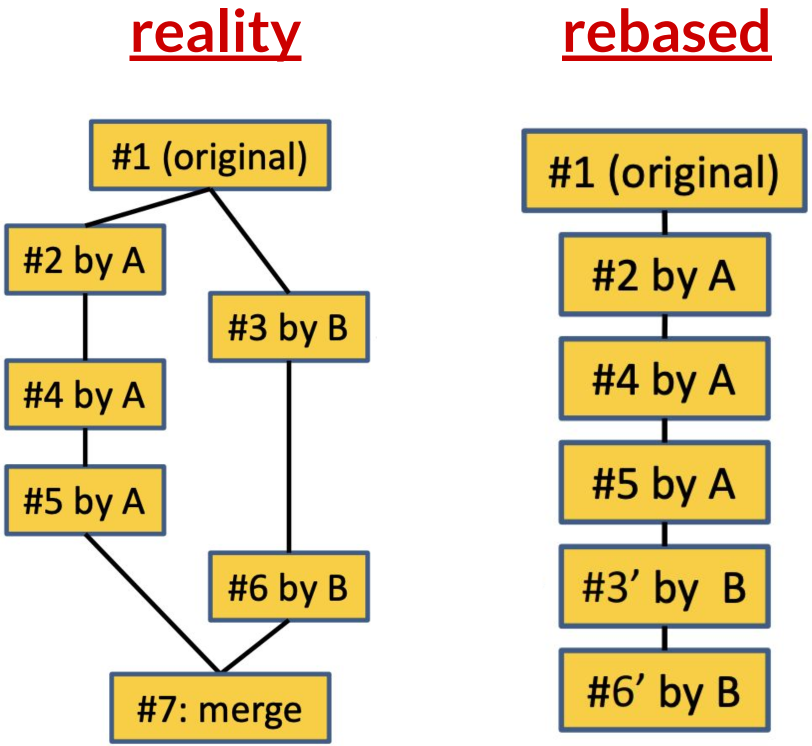
- `pull` incorporates others' changes into your repository
  - (update brings changes into your working copy)
  - (N.b.: `git pull` does pull, merge, and update)
- If you are **behind**, nothing more to do
  - Behind = your history is a **prefix** of master history
- If you have made changes in parallel, you must **merge**
  - Merge = create a new version incorporating all changes

Coordinating with others: rebasing

- `rebase` **rewrites** history

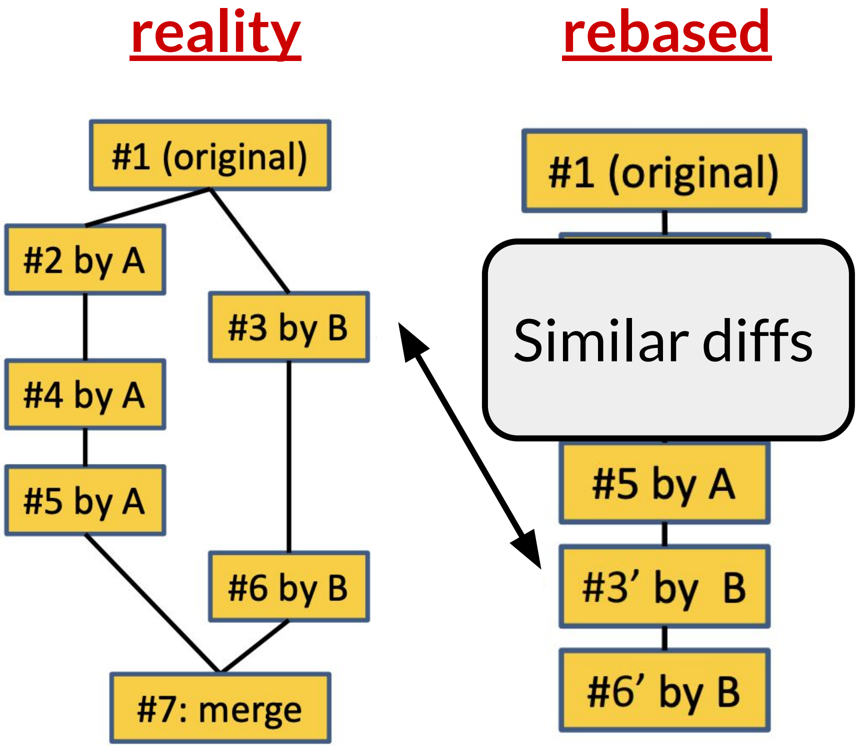
Coordinating with others: rebasing

- `rebase` **rewrites** history



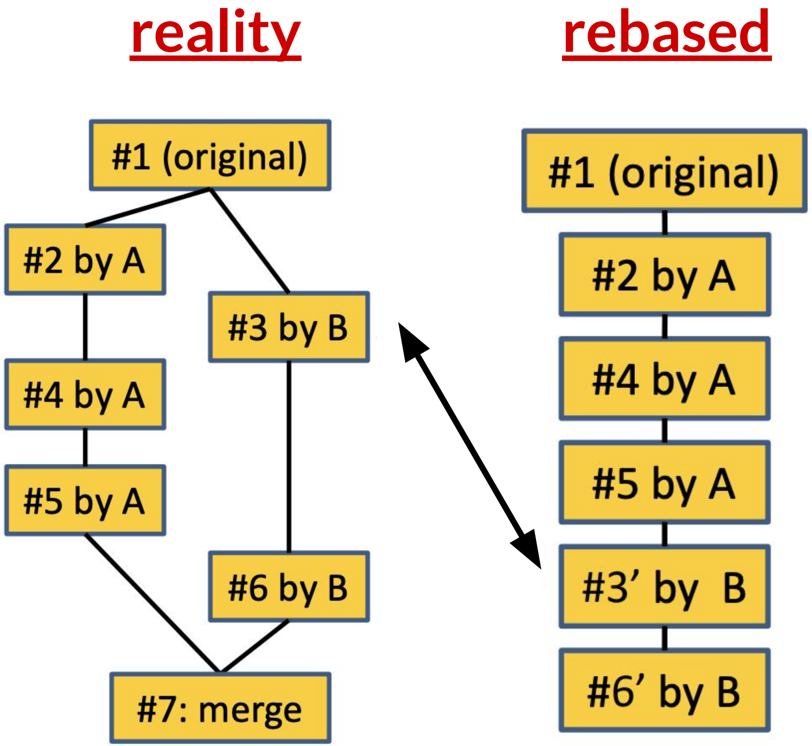
Coordinating with others: rebasing

- `rebase` **rewrites** history



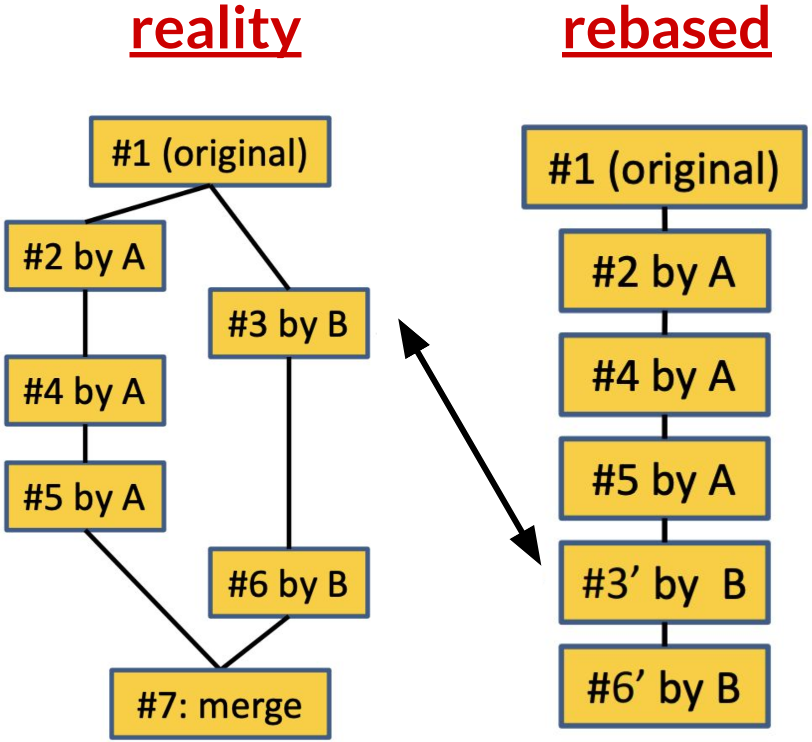
Coordinating with others: rebasing

- `rebase` **rewrites** history
- **Cleaner** history, easier to read



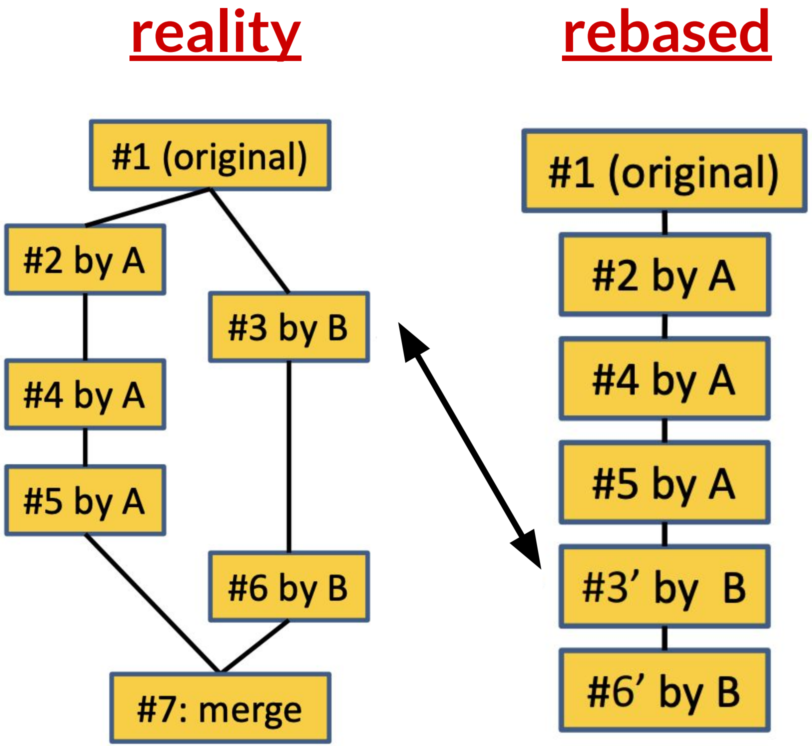
Coordinating with others: rebasing

- `rebase` **rewrites** history
- **Cleaner** history, easier to read
- Mixes commits #3 and #7
- Does not show context for change #3



Coordinating with others: rebasing

- `rebase` **rewrites** history
- **Cleaner** history, easier to read
- Mixes commits #3 and #7
- Does not show context for change #3
- Squash-and-merge is a safer form of rebasing



Coordinating with others: conflicts

Two changes can either be:

- **Conflict-free**:
- **Conflicting**:

Coordinating with others: conflicts

Two changes can either be:

- **Conflict-free**: changes are to different lines of a file
- **Conflicting**:

Coordinating with others: conflicts

Two changes can either be:

- **Conflict-free**: changes are to different lines of a file
- **Conflicting**:
  - Simultaneous changes to the same lines of a file
  - Requires manual conflict resolution

Coordinating with others: conflicts

Two changes can either be:

- **Conflict-free**: changes are to different lines of a file
- **Conflicting**:
  - Simultaneous changes to the same lines of a file
  - Requires manual conflict resolution

"Conflict-free" is a **textual**, **not semantic**, notion

- A heuristic about when to get the user involved
- Could yield compile errors or test failures

Aside: false positives and false negatives

Can X **actually** happen?

YES	NO

Aside: false positives and false negatives

Can X **actually** happen?

Did a tool **warn** us about X?	YES	NO
YES		
NO		

Aside: false positives and false negatives

Can X **actually** happen?

Did a tool **warn** us about X?	YES	NO
YES	**True positive**	
NO		

Aside: false positives and false negatives

		Can X **actually** happen?	
		YES	NO
Did a tool **warn** us about X?	YES	**True positive**	**False positive**
	NO		

Aside: false positives and false negatives

Can X **actually** happen?

Did a tool **warn** us about X?

Did a tool warn us about X? \ Can X actually happen?	YES	NO
YES	**True positive**	**False positive**
NO	**False negative**	

Aside: false positives and false negatives

Did a tool **warn** us about X? / Can X **actually** happen?	YES	NO
YES	True positive	False positive
NO	False negative	True negative

Aside: false positives and false negatives

Did a tool **warn** us about X?	Can X **actually** happen? YES	Can X **actually** happen? NO
YES	**True positive**	**False positive**
NO	**False negative**	**True negative**

Useful tool for thinking about anything that might warn us about a problem

Coordinating with others: conflicts

Two changes can either be:

- **Conflict-free**: changes are to different lines of a file
- **Conflicting**:
  - Simultaneous changes to the same
  - Requires manual conflict resolutic

False positives, false negatives, both, or neither?

"Conflict-free" is a **textual**, **not semantic**, notion

- A heuristic about when to get the user involved
- Could yield compile errors or test failures

Coordinating with others: conflicts

Two changes can either be:

- **Conflict-free**: changes are to different lines of a file
- **Conflicting**:
  - Simultaneous changes to the same
  - Requires manual conflict resolutio

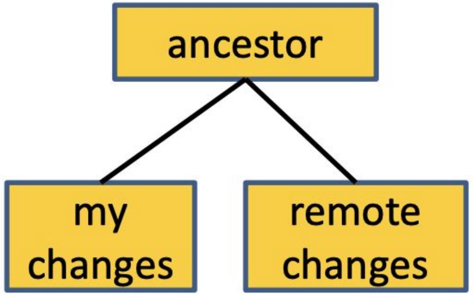
False positives,
false negatives,
both, or neither?

"Conflict-free" is a **textual**, **not semantic**, notion

- A **heuristic** about when to get the user involved
- Could yield compile errors or test failures

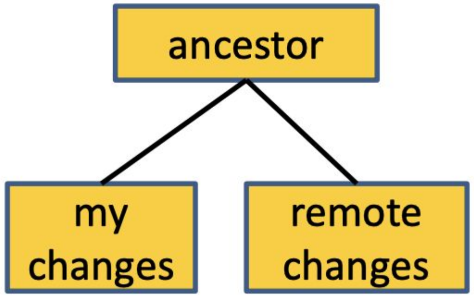
Coordinating with others: resolving conflicts

- There are **three versions** of the file:
- **You decide** which version to keep or how to merge them



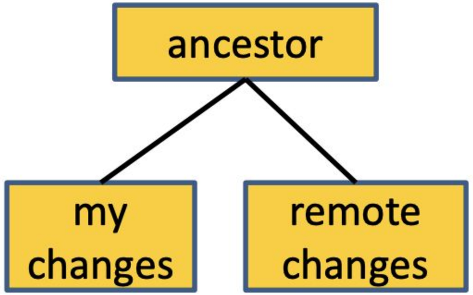
Coordinating with others: resolving conflicts

- There are **three versions** of the file:
- **You decide** which version to keep or how to merge them
- Many merge tools exist
- Configure your DVCS to use the merge tool that you prefer
  - **Practice** this ahead of time!



Coordinating with others: resolving conflicts

- There are **three versions** of the file:
- **You decide** which version to keep or how to merge them
- Many merge tools exist
- Configure your DVCS to use the merge tool that you prefer
  - **Practice** this ahead of time!
- **Don't panic!** Instead, think.
- **You can always** bail out of the merge and **start over**
  - You have the full local and remote history



Version Control

Today’s agenda:

- Reading Quiz
- How does a version control system work?
- **How to use your VCS**
- GitHub workflows

Version Control: advice and best practices

Best practice: don’t commit binary files

Best practice: don't commit binary files

- The history database records **changes**, not the entire file every time you commit

Best practice: don't commit binary files

- The history database records **changes**, not the entire file every time you commit
- **Avoid** binary files for content (especially simultaneous editing)
  - Word .docx files, Excel .xlsx files, other **proprietary formats**

Best practice: don't commit binary files

- The history database records **changes**, not the entire file every time you commit
- **Avoid** binary files for content (especially simultaneous editing)
  - Word .docx files, Excel .xlsx files, other **proprietary formats**
- Do not commit **generated files**, such as:
    - Binaries (e.g., .class files), etc.
    - IDE files (your teammates might use other tooling)

Best practice: don't commit binary files

- The history database records **changes**, not the entire file every time you commit
- **Avoid** binary files for content (especially simultaneous editing)
    - Word .docx files, Excel .xlsx files, other **proprietary formats**
- Do not commit **generated files**, such as:
        - Binaries (e.g., .class files), etc.
        - IDE files (your teammates might use other tooling)
    - Wastes space in repository
    - Causes merge conflicts

Best practice: feature branch development

- Whenever you start working on something new, create a branch
    - colloquially called a ***feature branch***, even when it’s not a feature

Best practice: feature branch development

- Whenever you start working on something new, create a branch
  - colloquially called a ***feature branch***, even when it's not a feature
- Pros:
  - features developed in isolation (less risk of main being broken)
  - encourages small PRs
- Cons:
  - large features can make integration difficult

Best practice: feature branch development

- Whenever you start working on something new, create a branch
  - colloquially called a *feature branch*
    feature
- Pros:
  - features developed in isolation (le
  - encourages small PRs
- Cons:
  - large features can make integration difficult

Advice: use feature branch development model iff your team typically ships features quickly

Advice: synchronize with teammates often

- Pull often

Advice: synchronize with teammates often

- Pull often
  - **Avoid getting behind** the main repo or your teammates
  - Avoid difficult and/or complex merges

Advice: synchronize with teammates often

- Pull often
    - **Avoid getting behind** the main repo or your teammates
    - Avoid difficult and/or complex merges
- Push as often as practical

Advice: synchronize with teammates often

- Pull often
  - **Avoid getting behind** the main repo or your teammates
  - Avoid difficult and/or complex merges
- Push as often as practical
  - Don’t let your teammates get behind you!
  - Don’t destabilize the main build
  - Avoid long periods working on a branch
    - but do work in a feature branch - don’t work directly on main!

Advice: commit messages

- Always write a commit message **yourself**

Advice: commit messages

- Always write a commit message **yourself**
    - **never** use an auto-generated message from a tool like "update *filename(s)*" from GitHub's GUI

Advice: commit messages

- Always write a commit message **yourself**
  - **never** use an auto-generated message from a tool like "update *filename(s)*" from GitHub's GUI
- Commit messages should be **descriptive**

Advice: commit messages

- Always write a commit message **yourself**
  - **never** use an auto-generated message from a tool like "update *filename(s)*" from GitHub's GUI
- Commit messages should be **descriptive**
- Don't write a novel: **summarize**. The code documentation in the commit should cover the rest.

Advice: commit messages: good or bad?

```
commit 763fe9cc335bb78ca45a608fa1f4c606713d5b44
Author: [redacted]
Date:

    Simplify `getImmediateSubcheckerClasses()` implementation (#5579)
```

Advice: commit messages: good or bad?

```
commit 763fe9cc335bb78ca45a608fa1f4c606713d5b44
Author:
Date:

    Simplify `getImmediateSubcheckerClasses()` implementation (#5579)
```

GOOD: short and to the point. Contains link to the PR it was merged in

Advice: commit messages: good or bad?

```
commit 123317b24a72215071a0f02e08635ee4b5b9669a
Author: [redacted]oreply.github.com>
Date:   [redacted]

    Update the code (#5)
```

Advice: commit messages: good or bad?

```
commit 123317b24a72215071a0f02e08635ee4b5b9669a
Author: [redacted] oreply.github.com>
Date: [redacted]

    Update the code (#5)
```

NOT SO GOOD:
description is vague
(looks auto-generated!)

Advice: commit messages: good or bad?

```
commit ddb6ab4df36a6bac3d4b118d40278f3428029f0c
Author: [redacted]@virginia.edu>
Date:   [redacted] 2014 -0500

    Comments? My code is self documenting.
```

Advice: commit messages: good or bad?

```
commit ddb6ab4df36a6bac3d4b118d40278f3428029f0c
Author: [redacted]@virginia.edu>
Date:   [redacted] 2014 -0500

    Comments? My code is self documenting.
```

NOT SO GOOD: while the humor is nice, this message is content-free

Advice: commit early and often

Advice: commit early and often

- Make **many small commits**, not one big one

Advice: commit early and often

- Make **many small commits**, not one big one
  - **Easier** to understand, review, merge, revert

Advice: commit early and often

- Make **many small commits**, not one big one
    - **Easier** to understand, review, merge, revert
- How to make many small commits:

Advice: commit early and often

- Make **many small commits**, not one big one
    - **Easier** to understand, review, merge, revert
- How to make many small commits:
    - Do only **one task at a time** and commit after each one

Advice: commit early and often

- Make **many small commits**, not one big one
    - **Easier** to understand, review, merge, revert
- How to make many small commits:
    - Do only **one task at a time** and commit after each one
    - Do multiple tasks in one working copy
        - Commit only a subset of files (use git's staging area)
        - Error-prone

Advice: commit early and often

- Make **many small commits**, not one big one
  - **Easier** to understand, review, merge, revert
- How to make many small commits:
  - Do only **one task at a time** and commit after each one
  - Do multiple tasks in one working copy
    - Commit only a subset of files (use git's staging area)
    - Error-prone
  - Create a branch for each simultaneous task
    - Need to keep track of all your branches, merge
    - Easier to share unfinished work with teammates

Advice: ways to avoid merge conflicts

- **Modularize** your work
  - Divide work so that individuals or subteams "**own**" a module
  - Other team members only need to understand its specification (abstractions!)
  - Requires good documentation and testing

Advice: ways to avoid merge conflicts

- **Modularize** your work
  - Divide work so that individuals or subteams "**own**" a module
  - Other team members only need to understand its specification (abstractions!)
  - Requires good documentation and testing

Bonus: this kind of modularization improves **observability** for management: it's easier to see who is being productive

Advice: ways to avoid merge conflicts

- **Modularize** your work
  - Divide work so that individuals or subteams "**own**" a module
  - Other team members only need to understand its specification (abstractions!)
  - Requires good documentation and testing
- **Communicate** about changes that may conflict
  - Don't overwhelm the team with such messages

Advice: always use version control

Advice: always use version control

- Still worthwhile, **even when working alone**
  - backups
  - feature branches are still useful when working on multiple parts of a system in parallel
  - sharing work across multiple computers

Advice: always use version control

- Still worthwhile, **even when working alone**
  - backups
  - feature branches are still useful when working on multiple parts of a system in parallel
  - sharing work across multiple computers
- Use **private repos** for things that should be private (e.g., your IP0/1/2 solutions…)
  - GitHub will give you free private repos because you're students

Advice: always use version control

- Still worthwhile, **even when working alone**
  - backups
  - feature branches are still useful when working on multiple parts of a system in parallel
  - sharing work across multiple
- Use **private repos** for things that ur IP0/1/2 solutions...)
  - GitHub will give you free priv students

I use **text-based formats** for many files so that I can version control them

Version Control

Today’s agenda:

- Reading Quiz
- How does a version control system work?
- How to use your VCS
- **GitHub workflows**

How to make a PR on GitHub

- start by creating a ***fork*** of the project
    - a new repository controlled by you, connected to the main

How to make a PR on GitHub

- start by creating a ***fork*** of the project
  - a new repository controlled by you, connected to the main
- in your fork, create a **feature branch**

How to make a PR on GitHub

- start by creating a ***fork*** of the project
  - a new repository controlled by you, connected to the main
- in your fork, create a **feature branch**
- write code + tests

How to make a PR on GitHub

- start by creating a ***fork*** of the project
  - a new repository controlled by you, connected to the main
- in your fork, create a **feature branch**
- write code + tests
- commit **early and often**, push to your fork

How to make a PR on GitHub

- start by creating a ***fork*** of the project
  - a new repository controlled by you, connected to the main
- in your fork, create a **feature branch**
- write code + tests
- commit **early and often**, push to your fork
- **prepare** for code review: follow code review author's best practices

How to make a PR on GitHub

- start by creating a ***fork*** of the project
    - a new repository controlled by you, connected to the main
- in your fork, create a **feature branch**
- write code + tests
- commit **early and often**, push to your fork
- **prepare** for code review: follow code review author's best practices
    - we'll discuss how to do a code review in a few weeks

How to make a PR on GitHub

- start by creating a ***fork*** of the project
  - a new repository controlled by you, connected to the main
- in your fork, create a **feature branch**
- write code + tests
- commit **early and often**, push to your fork
- **prepare** for code review: follow code review author's best practices
  - we'll discuss how to do a code review in a few weeks
- open PR against "**main**" repository from your fork's feature branch

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
    - a new repository controlled by you, unconnected to the main

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch
- write code (if there are already tests, don't bother to run them)

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch
- write code (if there are already tests, don't bother to run them)
- commit **all** of your code at once, when you're done

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch
- write code (if there are already tests, don't bother to run them)
- commit **all** of your code at once, when you're done
- **don't bother** to check if you've followed best practices

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch
- write code (if there are already tests, don't bother to run them)
- commit **all** of your code at once, when you're done
- **don't bother** to check if you've followed best practices
- **email** your changes to the maintainer of the original project

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch
- write code (if there are already tests, don't bother to run them)
- commit **all** of your code at once, when you're done
- **don't bother** to check if you've followed best practices
- **email** your changes to the maintainer of the original project
  - bonus points: email the full working copy, not just the diffs

How NOT to make a PR on GitHub

- start by creating a ***hard fork*** of the project
  - a new repository controlled by you, unconnected to the main
- do all of your work on the repository's **main** branch
- write code (if there are already tests, don't bother to run them)
- commit **all** of your code at once, whe
- **don't bother** to check if you've follo
- **email** your changes to the maintaine
  - bonus points: email the full work

I've seen people make all of these mistakes (and more)!

Takeaways: version control

- Understand what your VCS is good for (storing text files, collaboration) and what it isn't good for (storing binaries!)
- Understand your VCS: don't just thoughtlessly use the GUI
- Follow best practices when using your VCS:
    - don't push straight to main
    - practice resolving merge conflicts
    - use process to try to avoid merge conflicts, if possible
    - commit early and often
    - pull as often as you can