CSE P 504

Advanced topics in Software Systems

Fall 2022

Best practices and version control

October 10, 2022

Today

- Logistics
- Best practices (or how to avoid bugs)
- Version control with git
- In-class exercise 1

Logistics

- Course material, schedule, etc. on website: https://homes.cs.washington.edu/~rjust/courses/CSEP504
- Submission of assignments via Canvas: https://canvas.uw.edu
- Discussions on Slack: https://csep504.slack.com
- Poll everywhere: https://pollev.com/renejust859

Having trouble accessing any of those – let us know!

Logistics: in-class exercises

How to get the most out of these exercises?

1. **Prepare**
   - Follow set-up instructions: ready to go on Monday.
2. **Participate**
   - Work as a team: focus on problem solving and discussions.
3. **Reflect**
   - Revisit notes;
   - Submit deliverables;
   - Identify and raise open questions.

Logistics: in-class exercises

How to get the most out of these exercises?

1. **Prepare**
   - Follow set-up instructions: ready to go on Monday.
2. **Participate**
   - Work as a team: focus on problem solving and discussions.
3. **Reflect**
   - Revisit notes;
   - Submit deliverables;
   - Identify and raise open questions.

In-class due dates – Friday vs. Sunday?

Best practices

How to avoid bugs in your code?

How to avoid bugs in your code?

It's super simple...don't introduce them during coding!

"Everybody makes mistakes except for me…

But then, there is just one of me."

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
2. Correctness: get it right the first time
3. Bug visibility

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
      i. Ever had a use-after free bug in a garbage-collected language?
      ii. Ever had an assignment bug (String to Integer) in a statically typed language? (Even stronger guarantees with custom types and pluggable type systems.)
   b. Libraries and protocols
      i. TCP vs. UDP
      ii. No overflows in BigInteger

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
   b. Libraries and protocols
2. Correctness: get it right the first time
   a. A program without a spec is bug free
   b. Keep it simple, modular, and testable
   c. Defensive programming and conventions (discipline)

How to avoid bugs in your code?

A more realistic approach: 3 steps

1. Make certain bugs impossible by design
   a. Programming language
   b. Libraries and protocols
2. Correctness: get it right the first time
   a. A program without a spec is bug free
   b. Keep it simple, modular, and testable
   c. Defensive programming and conventions (discipline)
3. Bug visibility
   a. Assertions (pre/post conditions)
   b. (Regression) testing
   c. Fail fast

Quiz: setup and goals

- 3-4 students per team
- 4 code snippets
- 2 rounds
  - **First round**
    - For each code snippet, decide whether it represents good or bad practice.
    - **Goal:** discuss and reach consensus on good or bad practice.
  - **Second round** (known “solutions”)
    - For each code snippet, try to understand why it is good or bad practice.
    - **Goal:** come up with an explanation or a counter argument.

Round 1: good or bad?

Snippet 1: good or bad?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

Snippet 2: good or bad?

```
public enum PaymentType {DEBIT, CREDIT}
public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

Snippet 3: good or bad?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

Snippet 4: good or bad?

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return this.x;
   }
   public int getY() {
      return this.y;
   }
}
```

Round 2: why is it good or bad?

My take on this

- Snippet 1: bad
- Snippet 2: good
- Snippet 3: bad
- Snippet 4: good

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

Null references...the billion dollar mistake.

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

```
File[] files = getAllLogs();
for (File f : files) {
   …
}
```

Don’t return null; return an empty array instead.

Snippet 1: this is bad! why?

```
public File[] getAllLogs(Directory dir) {
   if (dir == null || !dir.exists() || dir.isEmpty()) {
      return null;
   } else {
      int numLogs = … // determine number of log files
      File[] allLogs = new File[numLogs];
      for (int i=0; i<numLogs; ++i) {
         allLogs[i] = … // populate the array
      }
      return allLogs;
   }
}
```

No diagnostic information.

Snippet 2: this is good, but why?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

Snippet 2: this is good, but why?

```
public enum PaymentType {DEBIT, CREDIT}

public void doTransaction(double amount, PaymentType payType) {
  switch (payType) {
    case DEBIT:
       … // process debit card
       break;
    case CREDIT:
       … // process credit card
       break;
    default:
       throw new IllegalArgumentException("Unexpected payment type");
   }
}
```

Type safety using an enum; throws an exception for unexpected cases (e.g., future extensions of PaymentType).

Snippet 3: Java API, but still bad! why?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

Snippet 3: Java API, but still bad! why?

```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add("Hello");
l.add("World");
l.remove(index);
```

What does the last call return?

Snippet 3: Java API, but still bad! why?


```
public class ArrayList<E> {
   public E remove(int index) {
      …
   }
   public boolean remove(Object o) {
      …
   }
   …
}
```



```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add("Hello");
l.add("World");
l.remove(index);
```


Avoid method overloading, which is statically resolved.
Autoboxing/unboxing adds additional confusion.

Snippet 4: this is good, but why?

```
public class Point {
   private final int x;
   private final int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return this.x;
   }
   public int getY() {
      return this.y;
   }
}
```

Snippet 4: this is good, but why?


```
public class Point {
   private final int x;
   private final int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() {
      return this.x;
   }
   public int getY() {
      return this.y;
   }
}
```


Good encapsulation; immutable object.

Version control

Why use version control?

Common App Essay

11:51pm

Why use version control?

Common App Essay

11:51pm

Common App Essay FINAL

11:57pm

Why use version control?

Common App Essay

Common App Essay EDITED FINAL

Common App Essay FINAL FINAL

Common App Essay FINAL REVISED

Common App Essay FINAL

Common App Essay OKAY THIS IS THE FINAL ONE

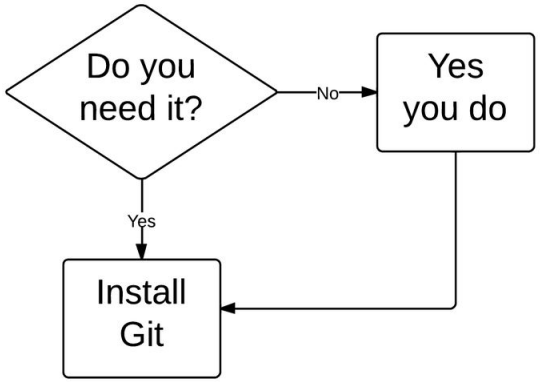
Common App Essay REVISED FINAL

Common App Essay REVISED

Just kidding... this is far more realistic.

Version control

Version control records changes to a set of files over time.
This makes it easy to review or obtain a specific version (later).



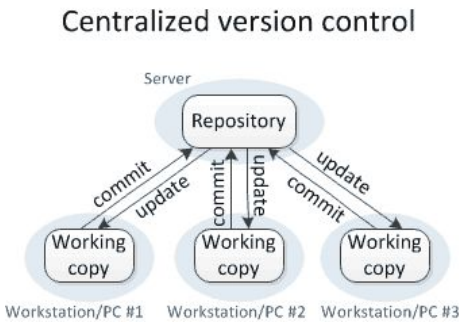
Who uses version control?

Example application domains

- Software development
- Research (infrastructure and data)
- Applications (e.g., (cloud-based) word processors)

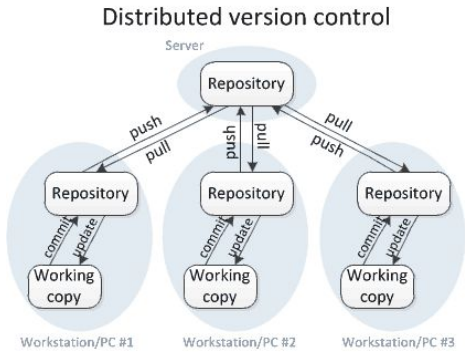
Centralized version control

- **One central repository.**
- All users **commit** their changes to a **central repository**.
- Each user has a working copy. As soon as they commit, the repository gets updated.
- Examples: SVN (Subversion), CVS.



Distributed version control

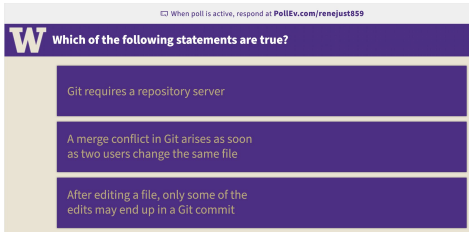
- **Multiple copies of a repository**.
- Each user **commits** to a **local** (private) repository.
- All committed changes remain local unless **pushed** to another repository.
- No external changes are visible unless **pulled** from another repository.
- Examples: Git, Hg (Mercurial).



Version control with Git
(aka the best thing since sliced bread)

- “I see Subversion as being the most pointless project ever started”
- " ‘what would CVS never ever do’-kind of approach”

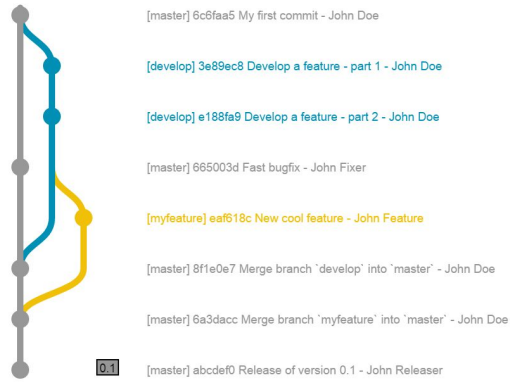
A little quiz



Branch vs. Clone vs. Fork

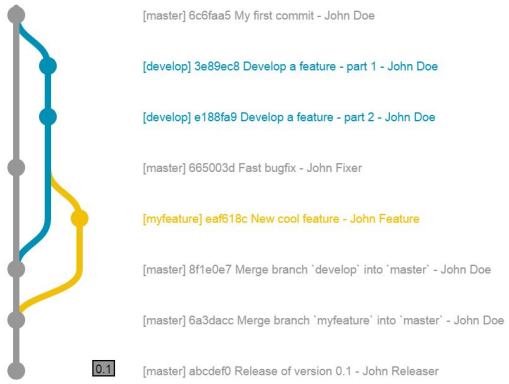
Branches

- One **main** development **branch** (**main**, master, trunk, etc.).
- Adding a new feature, fixing a bug, etc.: create a new **branch** -- a **parallel line of development**.
- **Lightweight** branching (**branch**).
- **Heavyweight** branching (**clone**).
- **Forking** (clone + metadata).

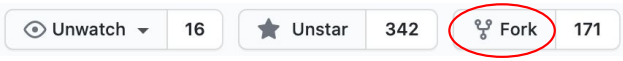


Branches

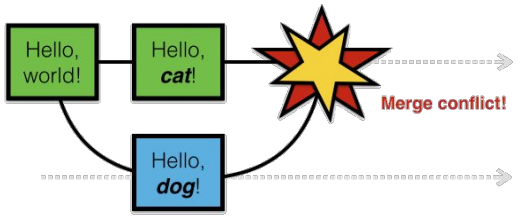
- One **main** development **branch** (**main**, master, trunk, etc.).
- Adding a new feature, fixing a bug, etc.: create a new **branch** -- a **parallel line of development**.
- **Lightweight** branching (**branch**).
- **Heavyweight** branching (**clone**).
- **Forking** (clone + metadata).



Branch and clone are common version control commands; forking is a concept used by GitHub etc.



Conflicts



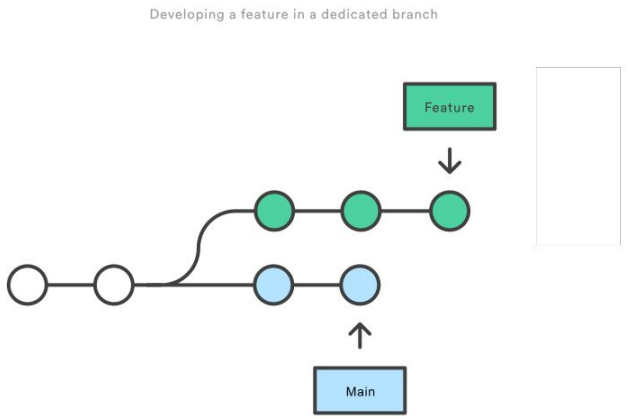
- **Conflicts** arise when two users **change the same line** (or two adjacent lines) of a file.
- When a conflict arises, the last committer needs to resolve it.

How to avoid merge conflicts?

Conflicts

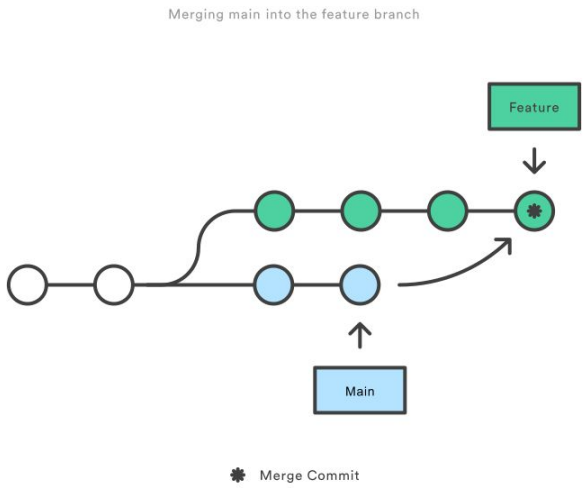
Merge vs. Rebase
(vs. Interactive Rebase)

Merge vs. Rebase



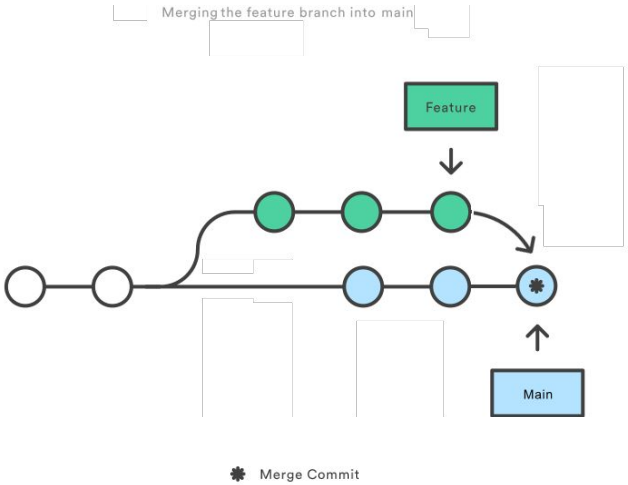
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Merge (integrating changes **from main**)



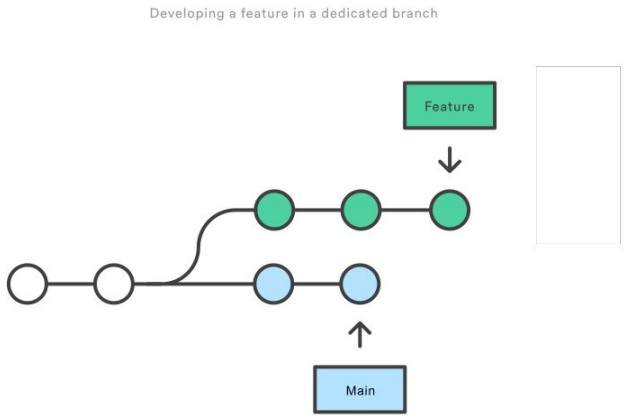
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Merge (integrating changes **into main**)



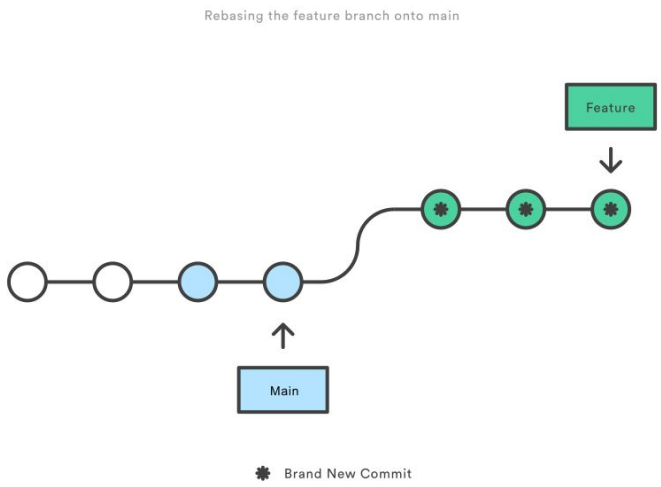
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Merge vs. Rebase



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

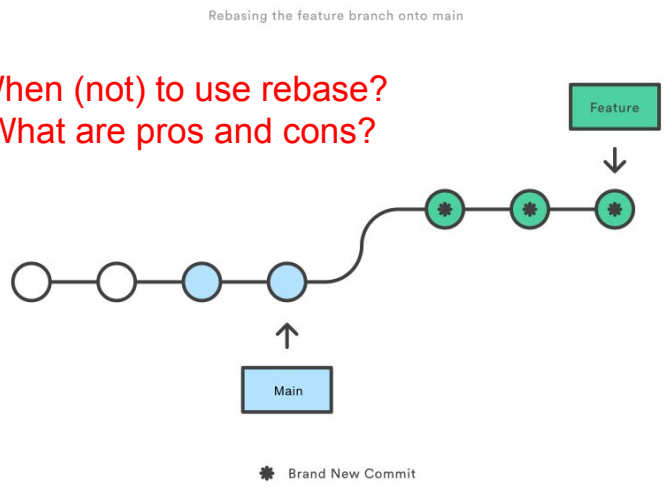
Merge vs. **Rebase**



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

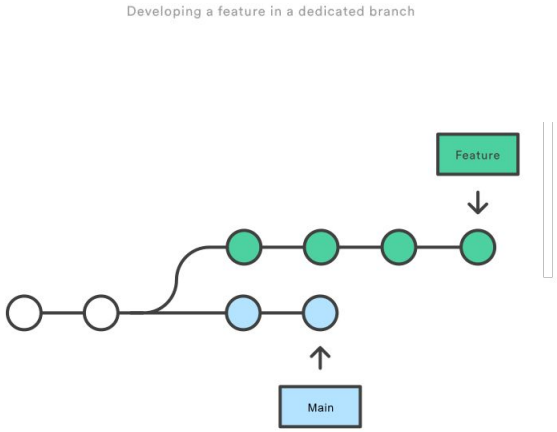
Merge vs. **Rebase**

When (not) to use rebase?
What are pros and cons?



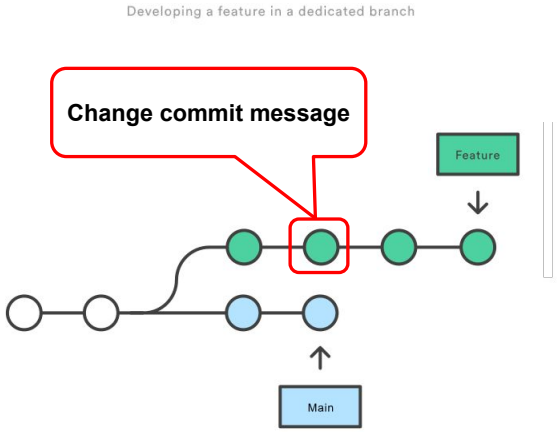
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase



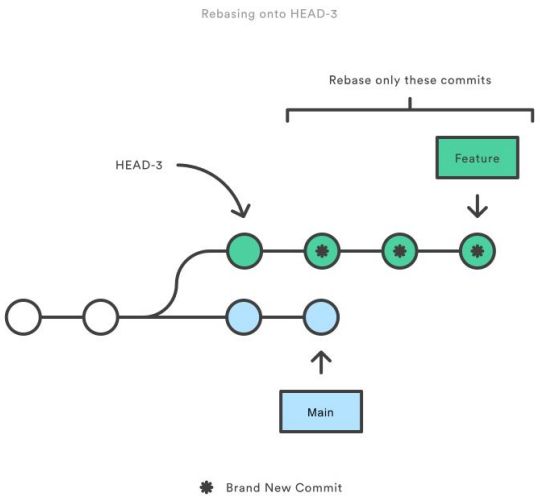
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (reword)



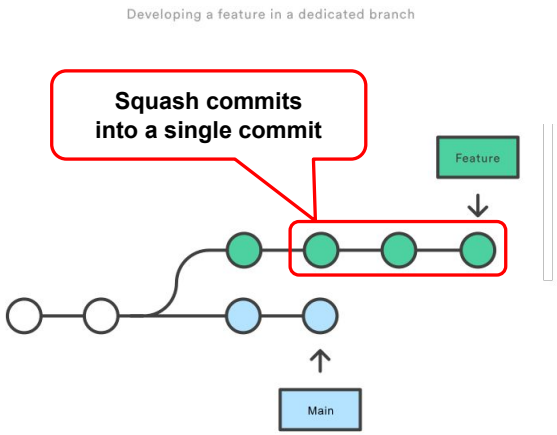
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (reword)



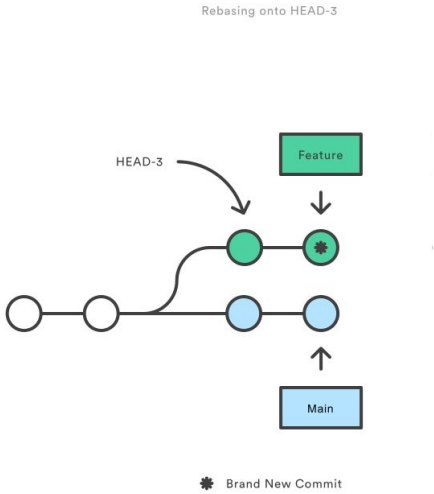
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (squash)



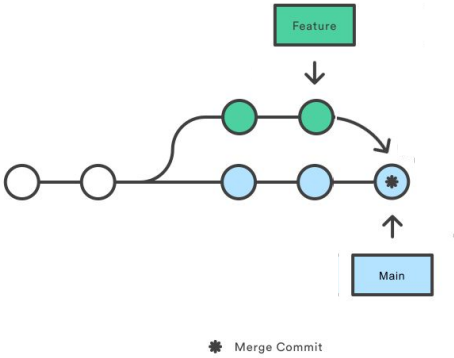
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (squash)



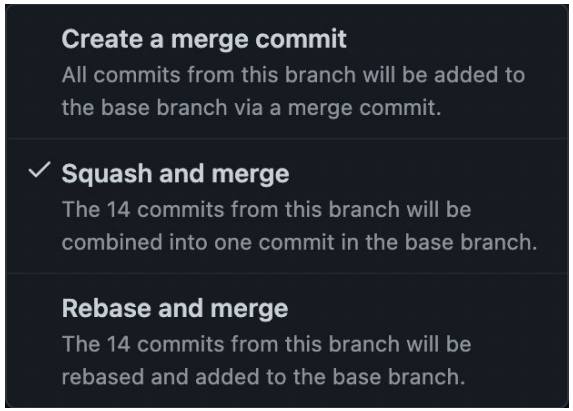
https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Interactive Rebase (squash & merge)

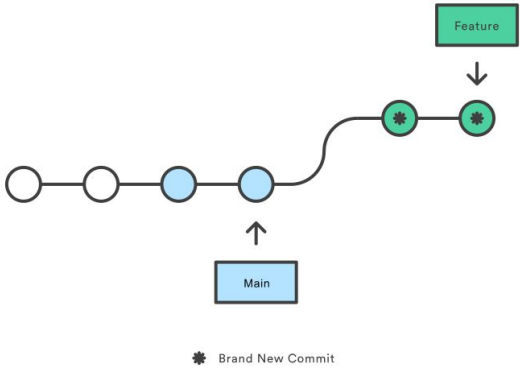


https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Squash & merge on GitHub



Interactive Rebase (squash & rebase)

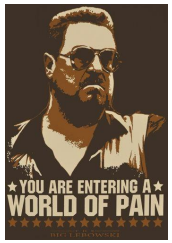


https://www.atlassian.com/git/tutorials/merging-vs-rebasing

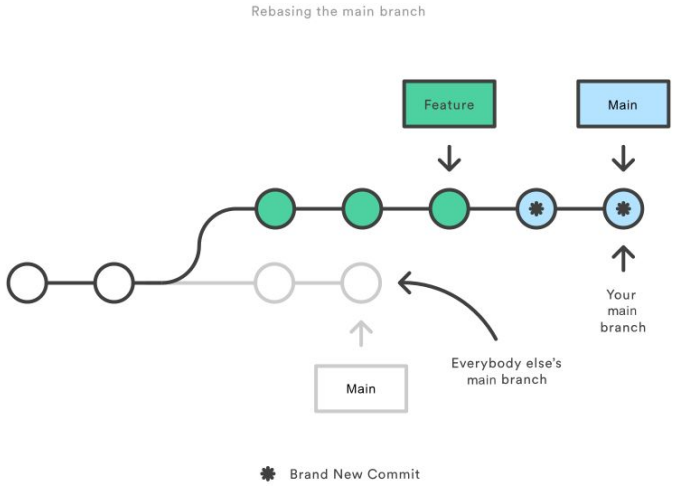
Rebase: a powerful tool, but …

- Results in a sequential commit history.
- Interactive rebasing often used to squash commits.
- **Changes the commit history!**

Do not rebase public branches with a force-push!



Rebase: a powerful tool, but …



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Git concepts and terminology

Motivating Example: What is this Git command?

NAME

git-_____ - _____ file contents to the index

SYNOPSIS

git _____ [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]

DESCRIPTION

This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically _____s the current content of existing paths as a whole, but with some options it can also be used to _____ content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

Motivating Example: What is this Git command?

NAME

git-add - Adds file contents to the index

SYNOPSIS

git add [--dry-run | -n] [--force | -f] [--interactive | -i] [--patch | -p]

DESCRIPTION

This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically adds the current content of existing paths as a whole, but with some options it can also be used to add content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.

Git: concepts and terminology

SYNOPSIS

git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION

git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

Git: concepts and terminology

SYNOPSIS
git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION
git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

SYNOPSIS
git-allocate-remote *[--derive-head | --massage-link-head | --abduct-commit]*

DESCRIPTION
git-allocate-remote allocates various non-branched local remotes outside added logs, and the upstream to be packed can be supplied in several ways.

SYNOPSIS
git-resign-index *[--snap-file] [--direct-change]*

DESCRIPTION
git-resign-index resigns all non-stashed unstaged indices, and the --manipulate-submodule flag can be used to add a branch for the upstream that is counted by a temporary submodule.

Git: concepts and terminology

SYNOPSIS
git-diff-index [-m] [--cached] [<common diff options>] <tree-ish> [<path>…]

DESCRIPTION
git-diff-index compares the content and mode of the blobs found in a tree object with the corresponding tracked files in the working tree, or with the corresponding paths in the index.

SYNOPSIS
git-allocate-remote *[--derive-head | --massage-link-head*

DESCRIPTION
git-allocate-remote allocates
be packed can be

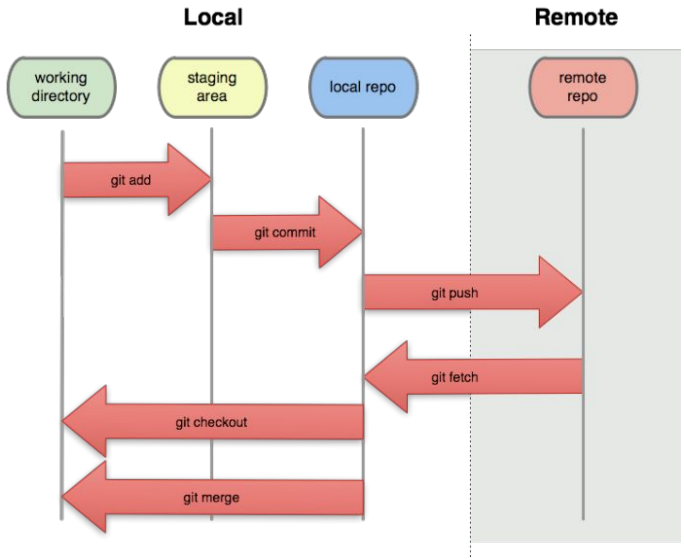
SYNOPSIS
git-resign-index

DESCRIPTION
git-resign-index resig
used to add a branch for the upstream that is counted by a temporary submodule.

Git: vocabulary

- **index**: staging area (located .git/index)
- **content**: git tracks **what is in a file, not the file itself**
- **tree**: git's representation of a file system
- **working tree**: tree representing the local working copy
- **staged**: ready to be committed
- **commit**: a snapshot of the working tree (a database entry)
- **ref**: pointer to a commit object
- **branch**: just a (special) ref; semantically: represents a line of dev
- **HEAD**: a ref pointing to the working tree

Git: concepts and terminology



In-class exercise 1