Lecture 04

ECE 1145: Software Construction and Evolution

Configuration Management

Announcements

- Remote lectures and recordings will be available if you cannot attend class due to illness or required quarantine or isolation
- Iteration 1 due Sept. 19
  - Utility.java, TestIterators.java
  - http://hamcrest.org/JavaHamcrest/tutorial
  - Groups/Teams now on Canvas
- Resources
  - Relevant Exercises: **33.4**
  - Hotel Safe TDD example on Canvas (Week 2 module)
  - https://dangitgit.com/
  - Branching models
    - https://nvie.com/posts/a-successful-git-branching-model/
    - https://guides.github.com/introduction/flow/

Project: Team Roles

- Coordinator: schedule meetings, delegate tasks, keep meetings on track, monitor completion of tasks
- Recorder: take meeting notes, organize report, submit report
- Checker/Monitor: observe team dynamics, ensure input from all team members, facilitate team decisions, review report submissions

For 2-person teams:

- Coordinator/Recorder
- Checker/Monitor

Project: Team Roles

- Coordinator: schedule meetings, delegate tasks, keep meetings on track, monitor completion of tasks
- Recorder: take meeting notes, organize report, submit report
- Checker/Monitor: observe team dynamics, ensure input from all team members, facilitate team decisions, review report submissions

For 2-person teams:

- Coordinator/Recorder
- Checker/Monitor

Everyone should write code!!!

→ choose from test list tasks that can be done in parallel, or utilize pair programming / team programming

Project: Team Roles

For Iteration 1: Assign by who woke up earliest

- From earliest to latest: Coordinator, Recorder, Checker/Monitor
- Coordinator/Recorder, Checker/Monitor for two-person teams

Rotate each iteration

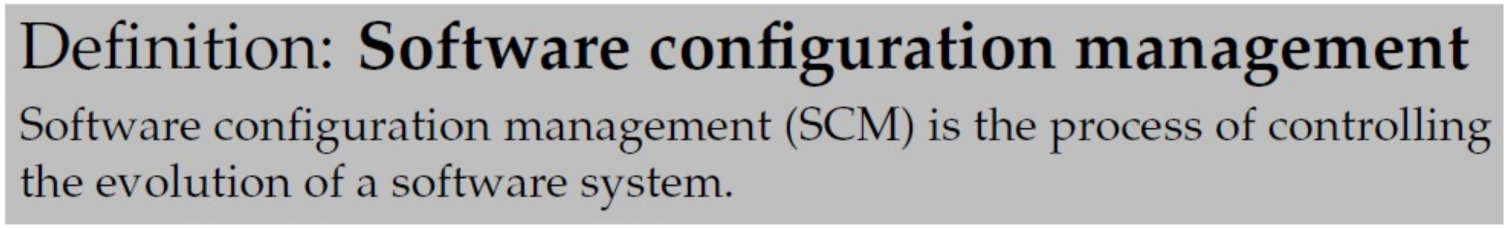
Describe roles/contributions in each report

Questions for Today

How do we manage multiple developers on a single code base?

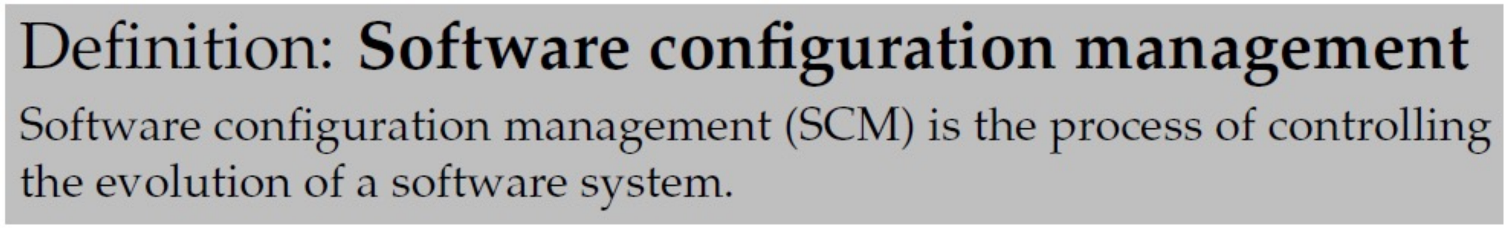
How do we track and maintain software versions?

Configuration Management


Definition: **Software configuration management**

Software configuration management (SCM) is the process of controlling the evolution of a software system.

Configuration Management

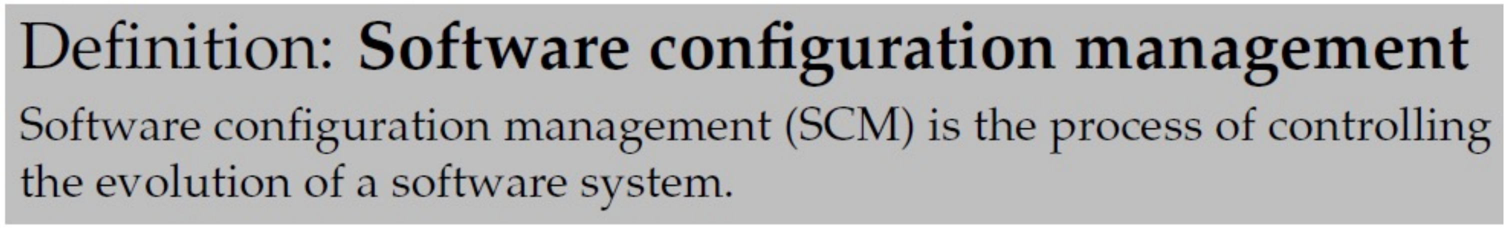

Definition: **Software configuration management**

Software configuration management (SCM) is the process of controlling the evolution of a software system.

Configuration Management helps to solve many problems in development!

- Team collaboration on large code bases
- Developing multiple features in parallel
- Managing release vs development versions
- Managing breaking changes
- Iterative software development

Configuration Management



Definition: **Software configuration management**

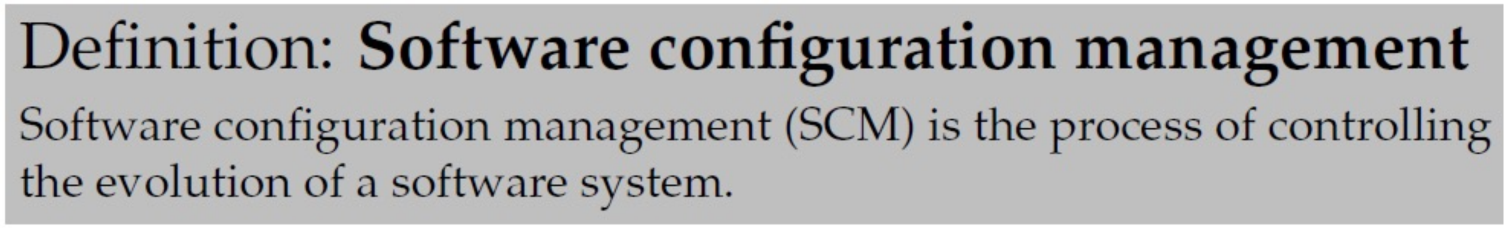
Software configuration management (SCM) is the process of controlling the evolution of a software system.

Configuration Management helps to solve many problems in development!

- Team collaboration on large code bases
- Developing multiple features in parallel
- Managing release vs development versions
- Managing breaking changes
- Iterative software development

We will use **Git**

Configuration Management



Definition: **Software configuration management**

Software configuration management (SCM) is the process of controlling the evolution of a software system.

Standards

- IEEE 828: Software Configuration Management Plans
- IEEE 1042: Guide to Software Configuration Management

Configuration Management

Definition: **Configuration item**

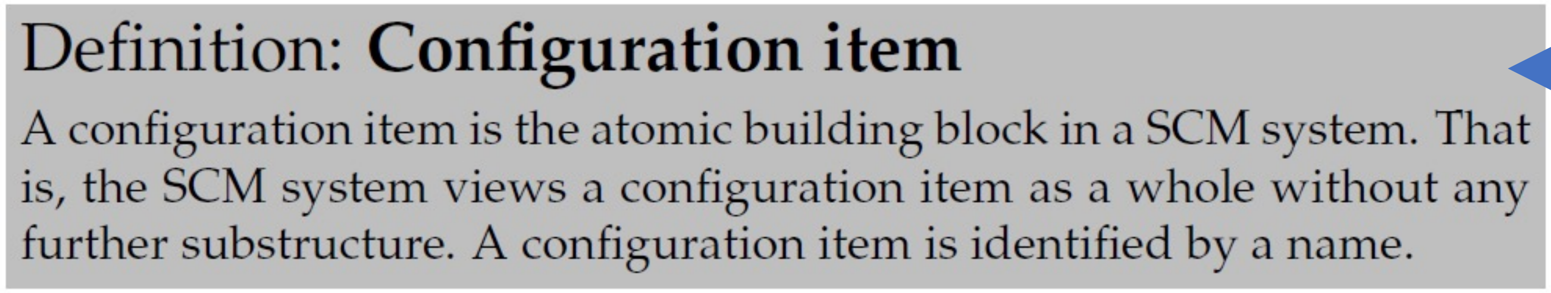
A configuration item is the atomic building block in a SCM system. That is, the SCM system views a configuration item as a whole without any further substructure. A configuration item is identified by a name.

Definition: **Configuration**

A configuration is a named hierarchical structure that aggregates configuration items and configurations.

Configuration Management

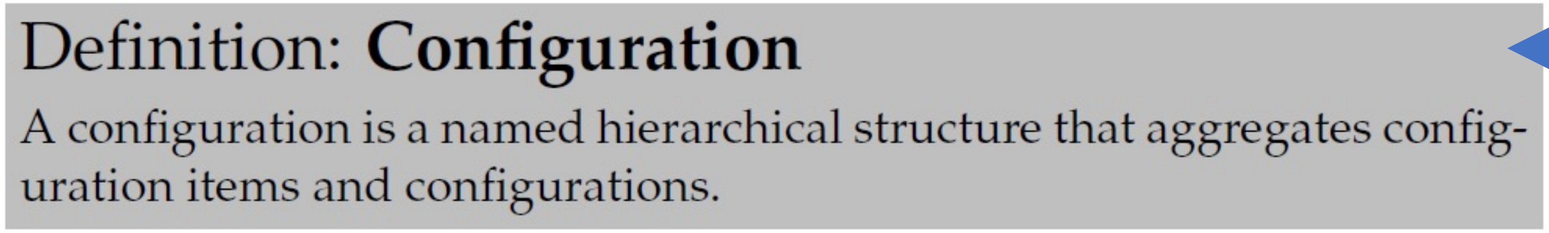
Using Git:



Definition: **Configuration item**

A configuration item is the atomic building block in a SCM system. That is, the SCM system views a configuration item as a whole without any further substructure. A configuration item is identified by a name.

File



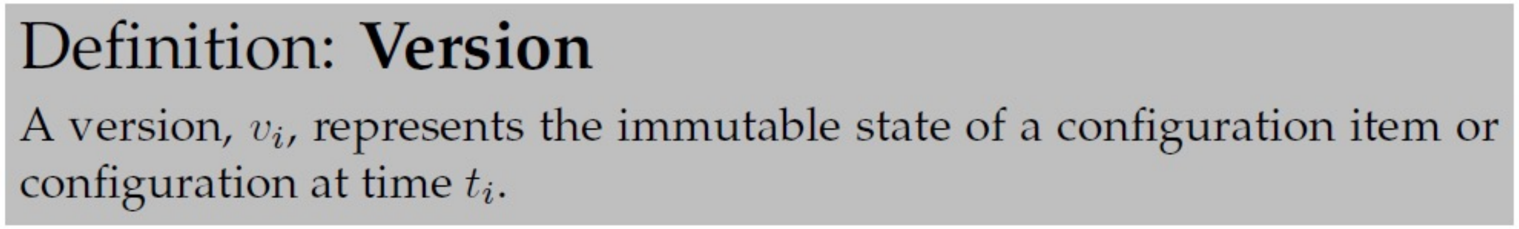
Definition: **Configuration**

A configuration is a named hierarchical structure that aggregates configuration items and configurations.

Files + Folder structure

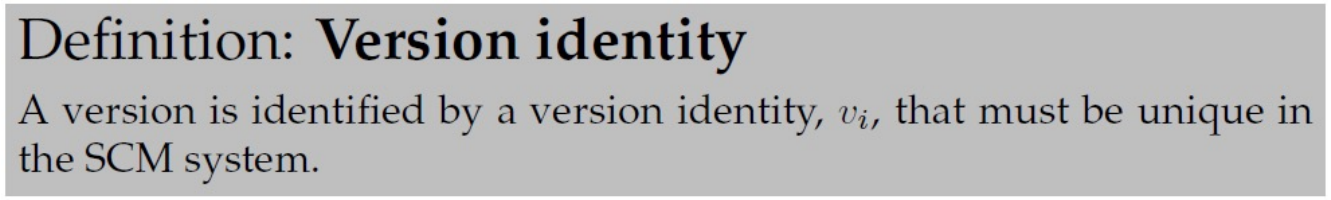
Version Control

Version control is synonymous with configuration management



Definition: **Version**

A version, v_i, represents the immutable state of a configuration item or configuration at time t_i.



Definition: **Version identity**

A version is identified by a version identity, v_i, that must be unique in the SCM system.

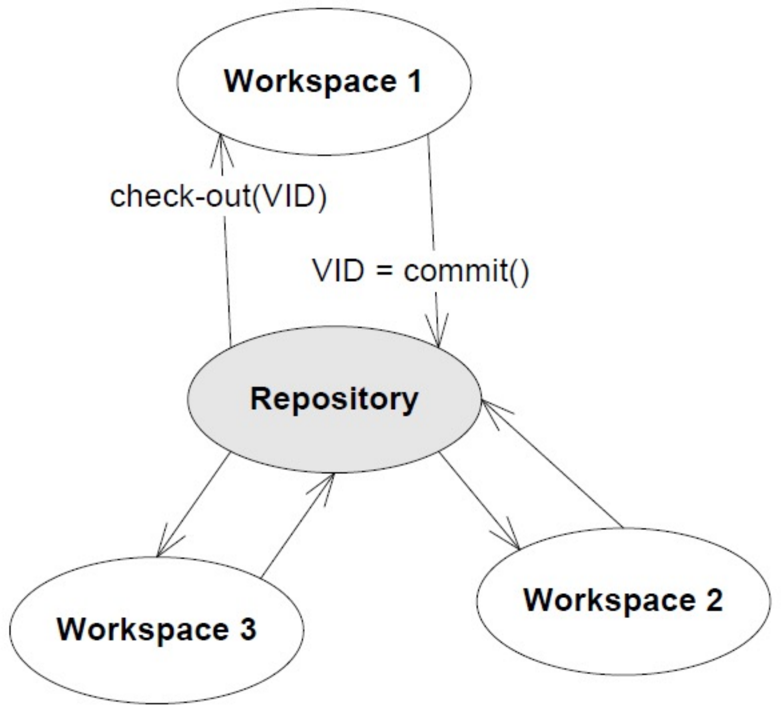
Version Control

Definition: Repository

The repository is a central database, maintained and controlled by the SCM system that stores all versions of all controlled entities.

Definition: Workspace

A workspace is a local file system in which individual versions of entities can be modified and altered. Only one version of a given entity is allowed at the same time in the workspace.



Version Control

Definition: Repository

The repository is a central database, maintained and controlled by the SCM system that stores all versions of all controlled entities.

Definition: Workspace

A workspace is a local file system in which individual versions of entities can be modified and altered. Only one version of a given entity is allowed at the same time in the workspace.

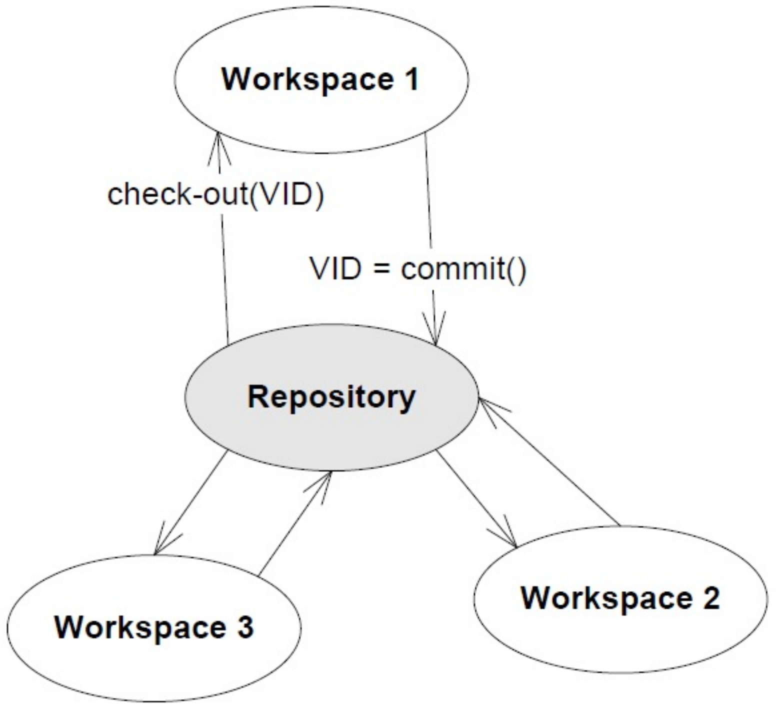
Definition: Commit

A commit is an operation that

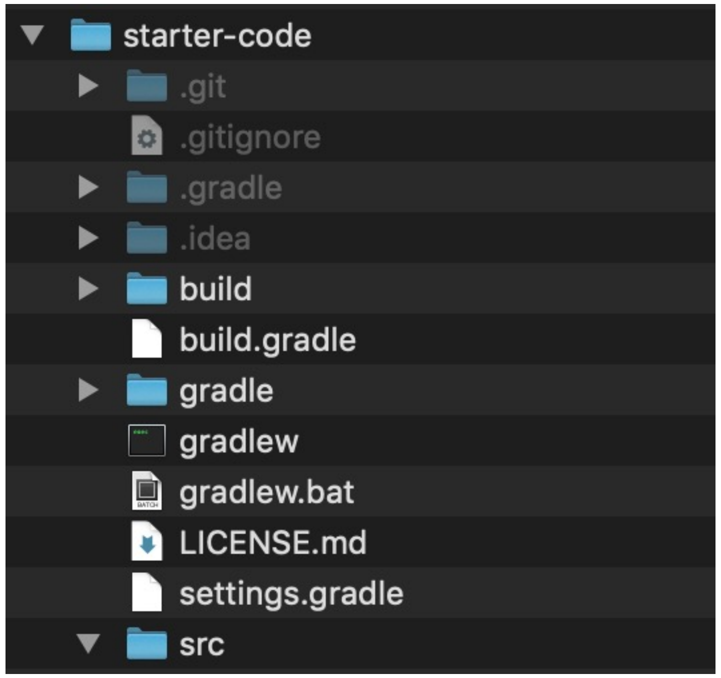
1. generates a new, unique, version identity for the given entity.
2. stores a copy/snapshot of the entity under this identity.

Definition: Check-out

A check-out is an operation that, given a unique version identity, is able to retrieve an exact copy of an entity as it looked when the given version identity was formed during a commit.



Version Control: Git


```
▼ starter-code
    ▶ .git
      .gitignore
    ▶ .gradle
    ▶ .idea
    ▶ build
      build.gradle
    ▶ gradle
      gradlew
      gradlew.bat
      LICENSE.md
      settings.gradle
    ▼ src
```

Repository information is stored in '.git'

- Workspace is standard folder/package structure

git commit
creates snapshot of the current version in the repository

git checkout <commit ID>
retrieves a specific snapshot into the workspace

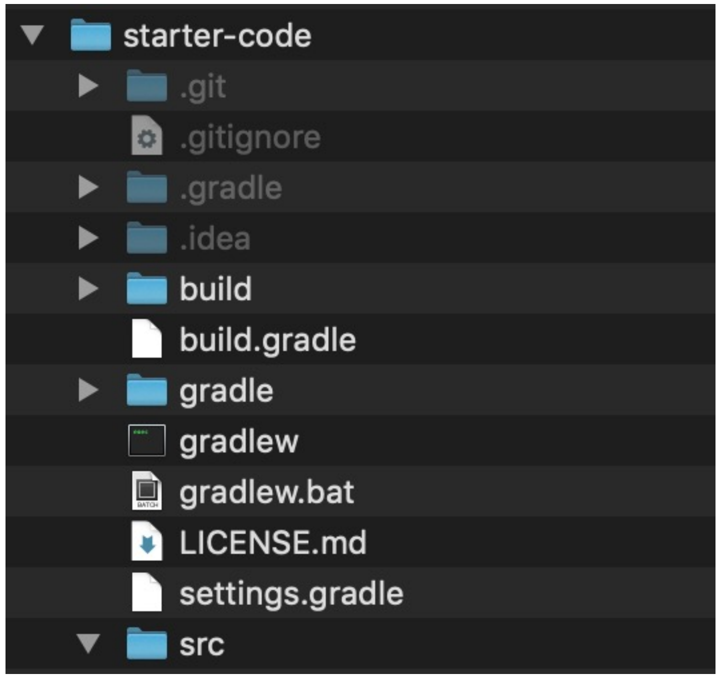
Version Control: Git

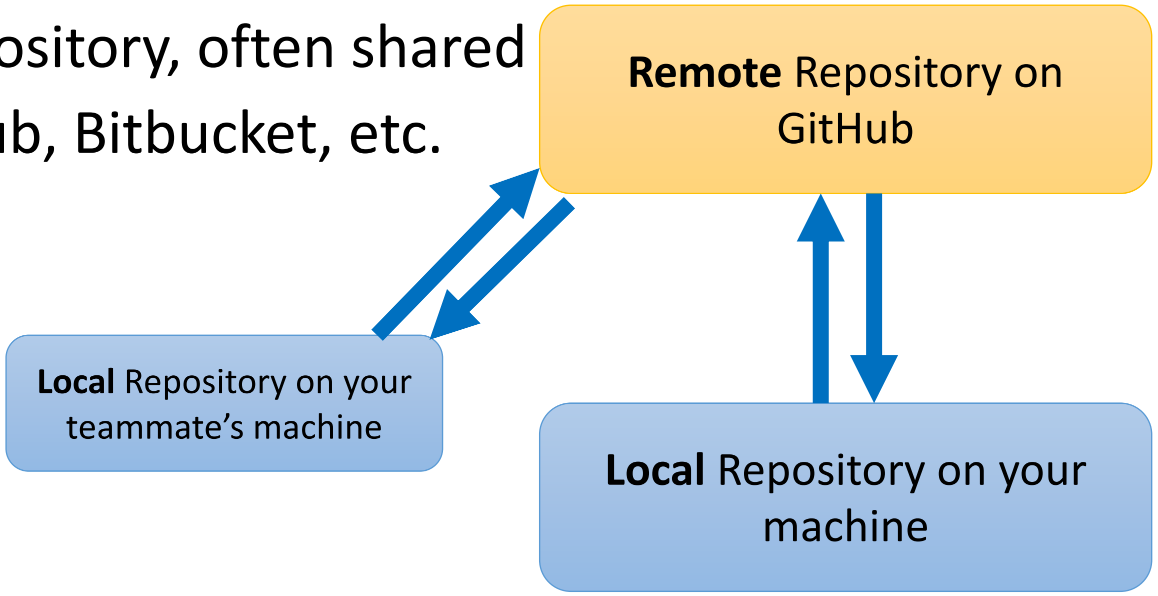
Repository information is stored in '.git'

- Workspace is standard folder/package structure

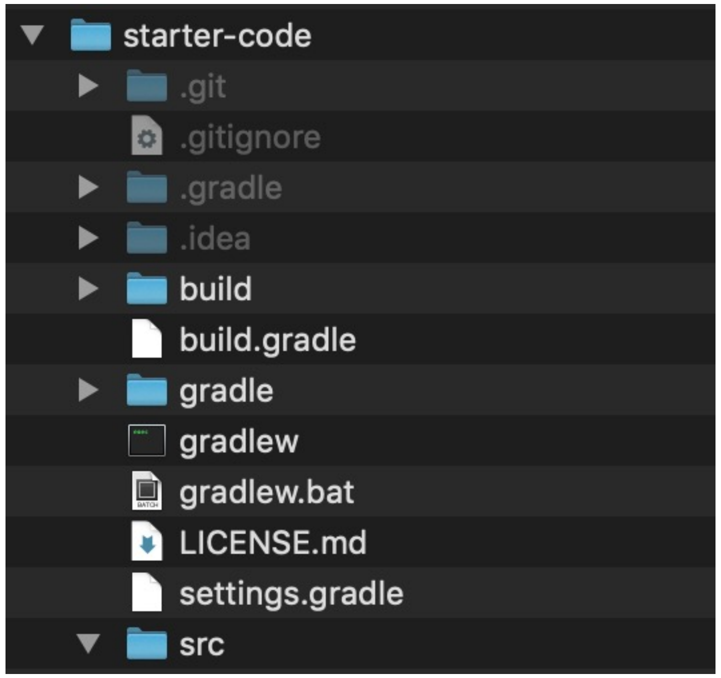
Typically have multiple repositories; default is 'local' and 'origin'

- Origin is usually a remote repository, often shared
- Remote repo hosted on GitHub, Bitbucket, etc.




Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine

Version Control: Git



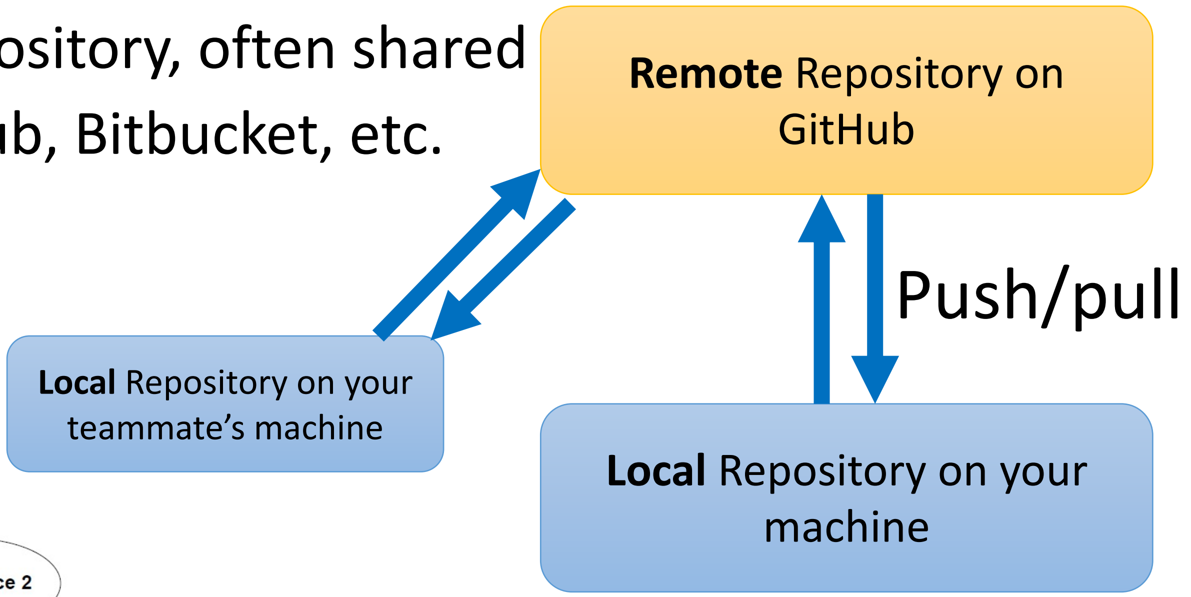
```
▼ starter-code
    ▶ .git
      .gitignore
    ▶ .gradle
    ▶ .idea
    ▶ build
      build.gradle
    ▶ gradle
      gradlew
      gradlew.bat
      LICENSE.md
      settings.gradle
    ▼ src
```

Repository information is stored in '.git'

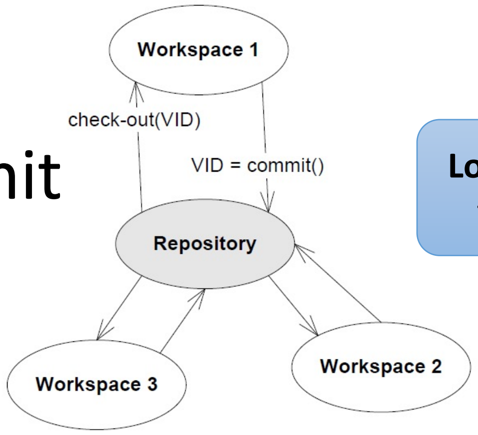
- Workspace is standard folder/package structure

Typically have multiple repositories; default is 'local' and 'origin'

- Origin is usually a remote repository, often shared
- Remote repo hosted on GitHub, Bitbucket, etc.


Remote Repository on GitHub
Local Repository on your teammate's machine
Push/pull
Local Repository on your machine

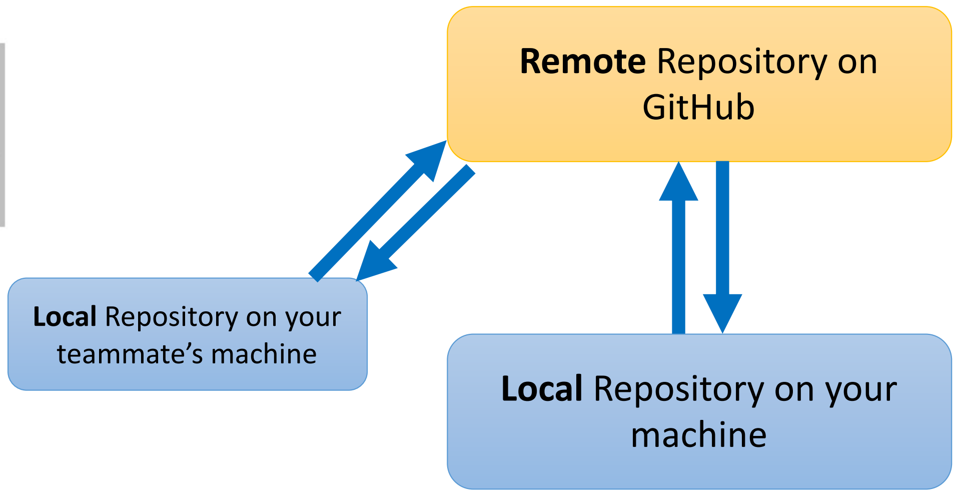
Checkout/commit

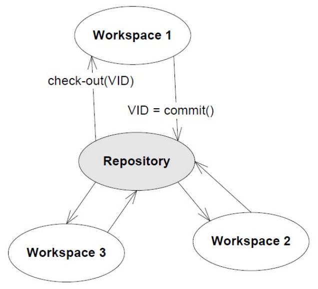


Version Control

Definition: Conflict

A conflict is a situation where the same piece of code has been changed at the same time in two or more different workspaces.


Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine



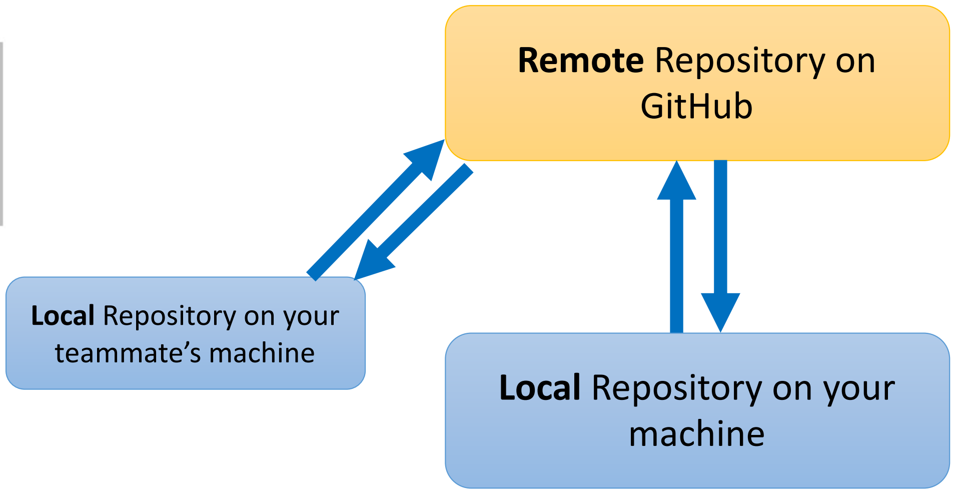
Version Control

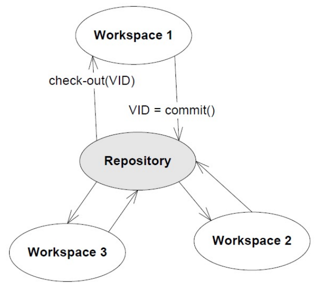
Definition: Conflict

A conflict is a situation where the same piece of code has been changed at the same time in two or more different workspaces.

Definition: Pessimistic concurrency

Ensure strict sequential modifications by *locking* configuration items during modification.


Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine



Version Control

Definition: Conflict

A conflict is a situation where the same piece of code has been changed at the same time in two or more different workspaces.

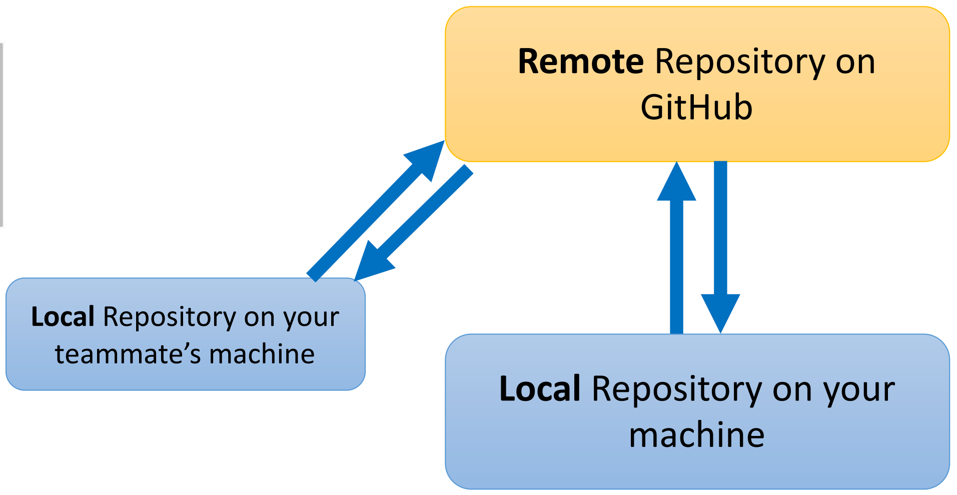
Definition: Pessimistic concurrency

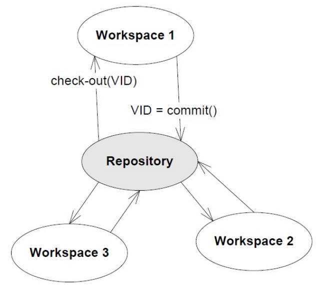
Ensure strict sequential modifications by *locking* configuration items during modification.

Definition: Optimistic concurrency

Allow for parallel modification and handle conflicts by merging.

← Git


Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine



Version Control

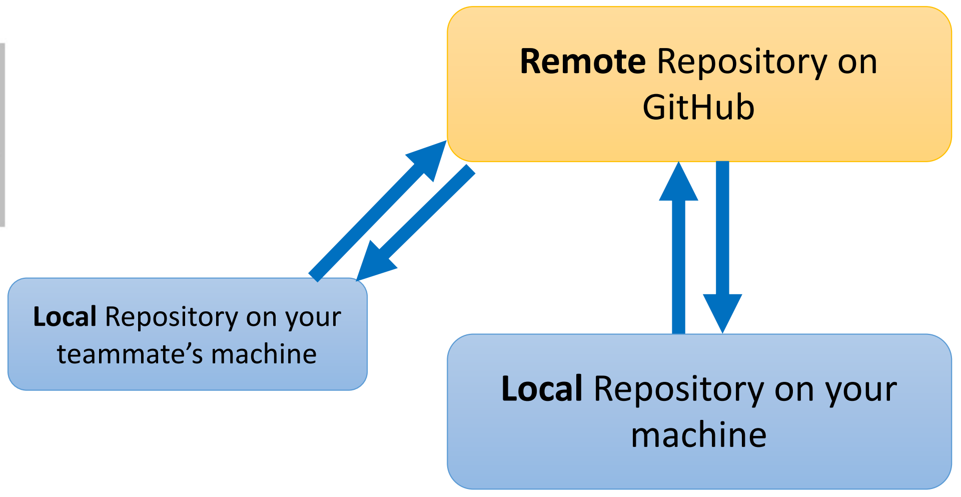
Definition: Conflict
A conflict is a situation where the same piece of code has been changed at the same time in two or more different workspaces.

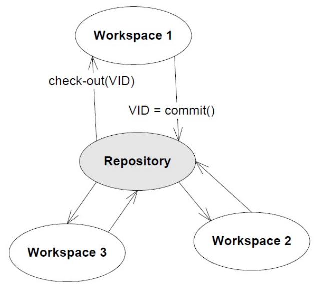
Definition: Pessimistic concurrency
Ensure strict sequential modifications by *locking* configuration items during modification.

Definition: Optimistic concurrency
Allow for parallel modification and handle conflicts by merging.

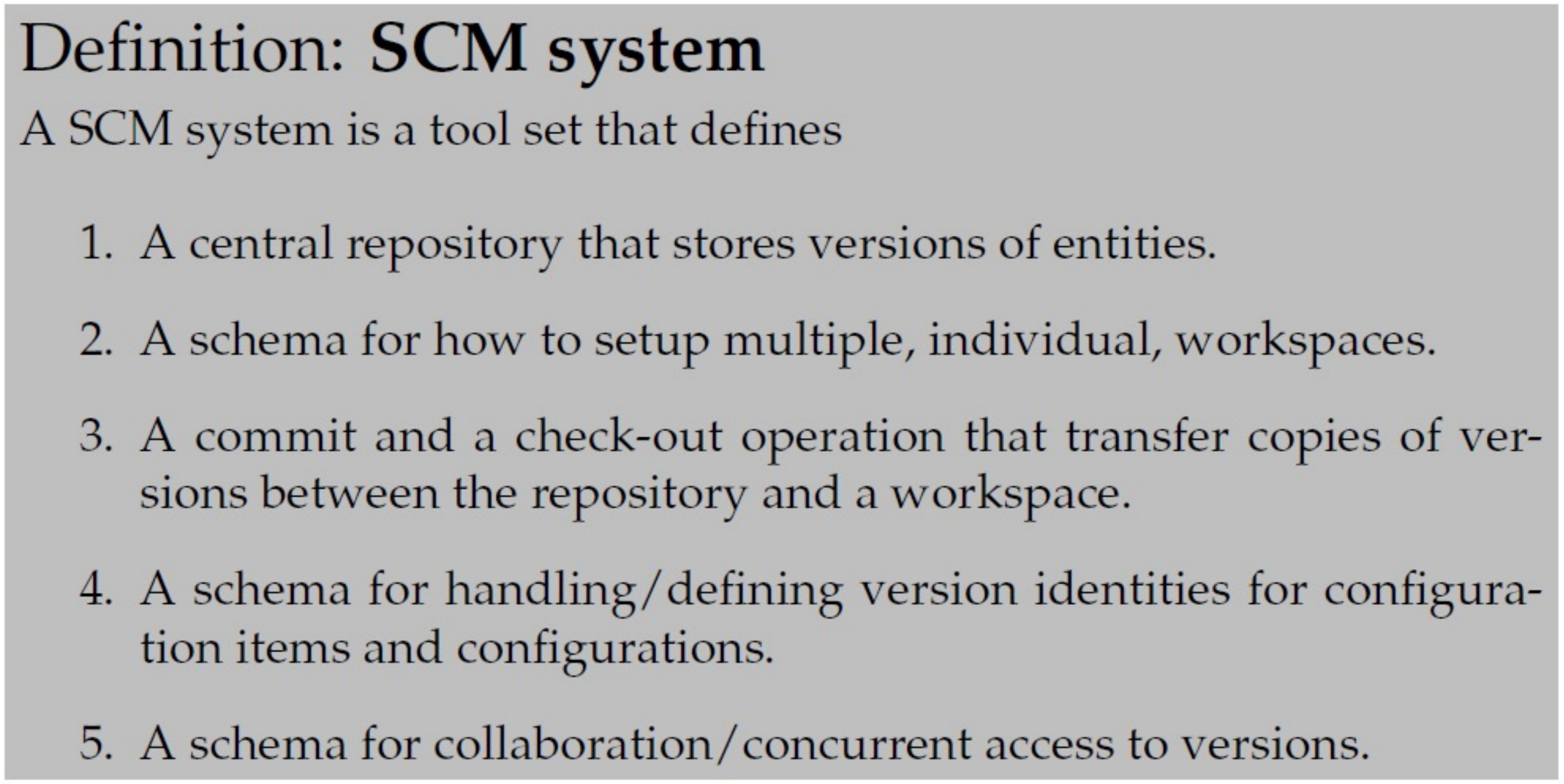
← Git

Definition: Merge
A merge is a operation where the sum of changes since the last common ancestor in the version graph is included in a configuration item or configuration.


Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine



Version Control: Git

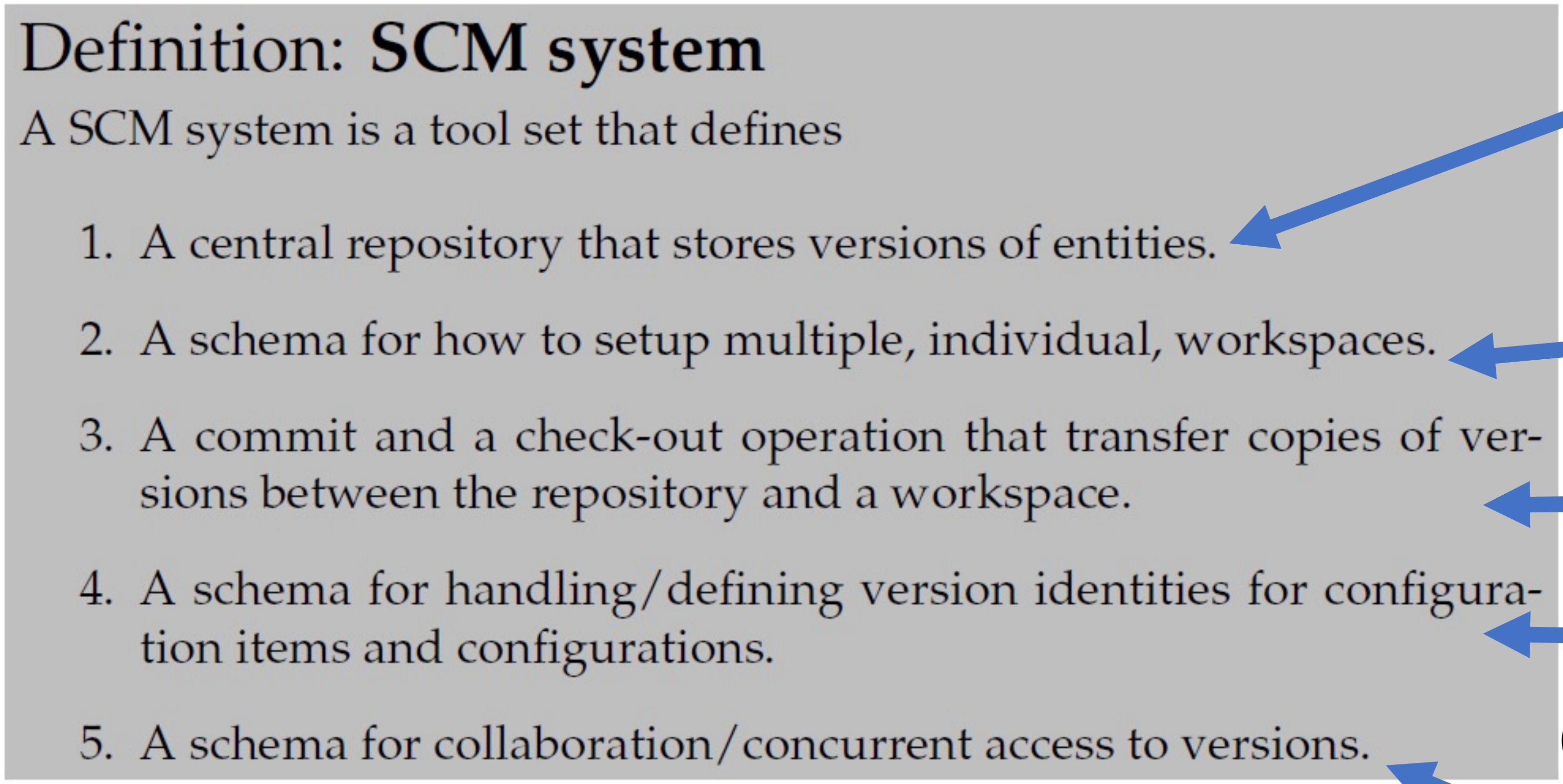


Definition: **SCM system**

A SCM system is a tool set that defines

1. A central repository that stores versions of entities.
2. A schema for how to setup multiple, individual, workspaces.
3. A commit and a check-out operation that transfer copies of versions between the repository and a workspace.
4. A schema for handling/defining version identities for configuration items and configurations.
5. A schema for collaboration/concurrent access to versions.

Version Control: Git



Definition: SCM system

A SCM system is a tool set that defines

1. A central repository that stores versions of entities.
2. A schema for how to setup multiple, individual, workspaces.
3. A commit and a check-out operation that transfer copies of versions between the repository and a workspace.
4. A schema for handling/defining version identities for configuration items and configurations.
5. A schema for collaboration/concurrent access to versions.

.git

git clone

git commit/checkout

Commit IDs (hashes)

Optimistic concurrency
clone/checkout
merge

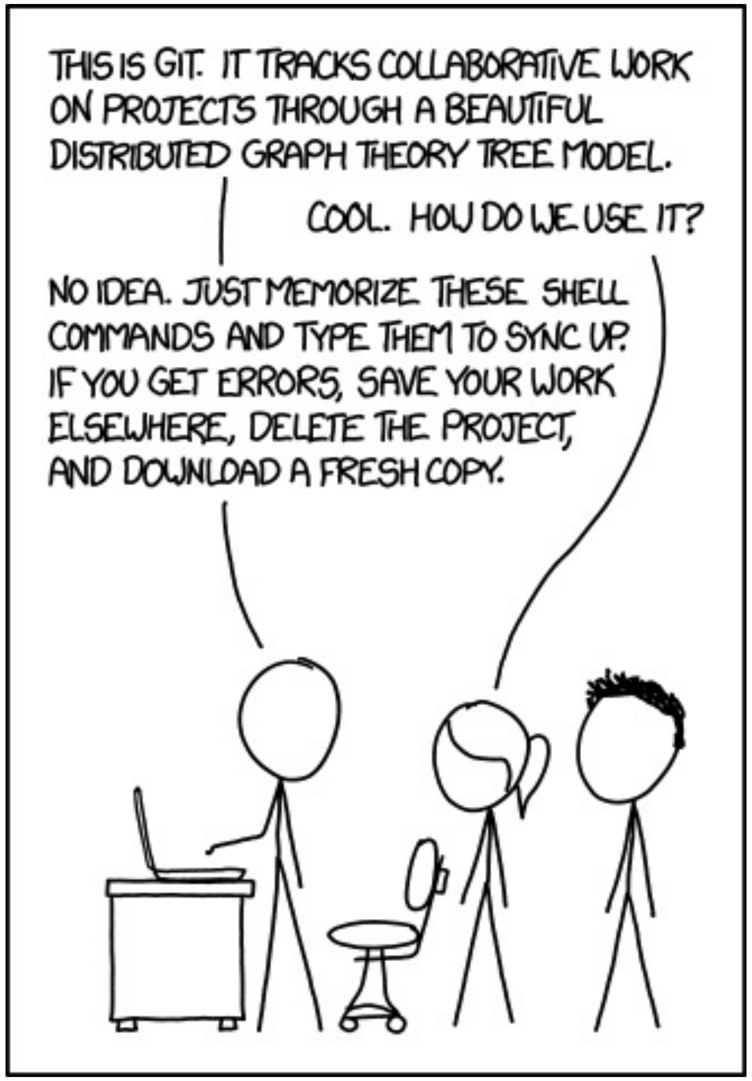
Git is a **distributed** software configuration management system

- Every workspace holds a complete copy of the repository (repo)

Version Control: Git

Git can get complicated!

You will mostly only need a few commands


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Version Control: Git

Git can get complicated!

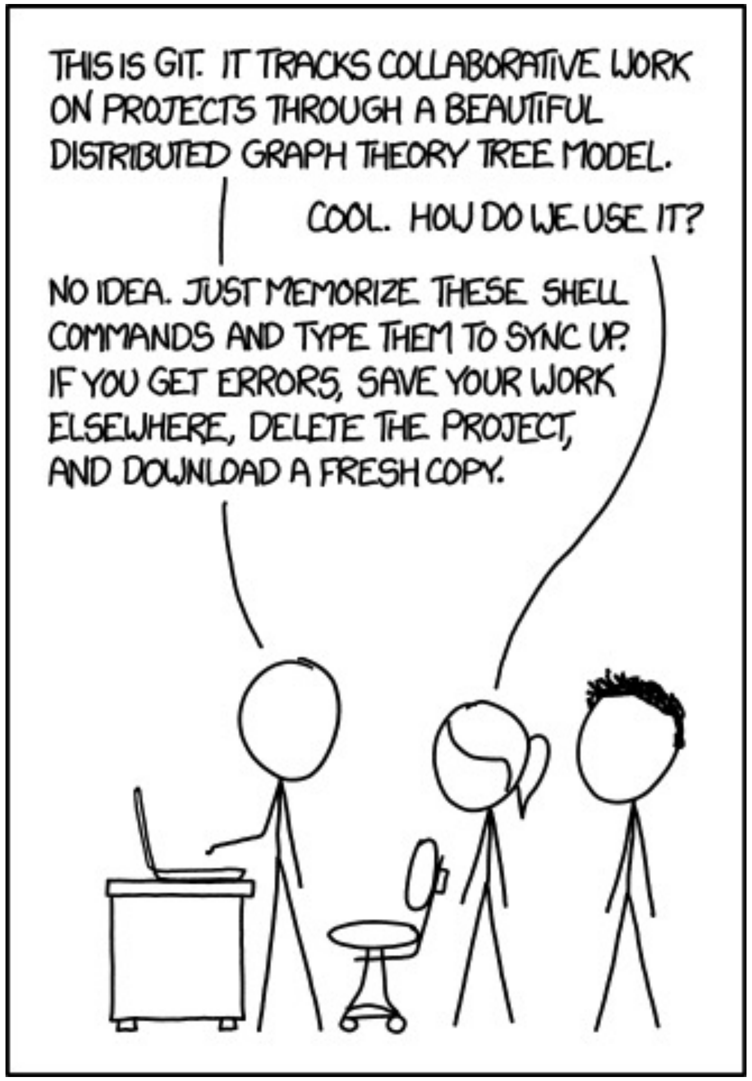
You will mostly only need a few commands

git commit
creates snapshot on the **local repository**

git push
Merges all changes from local repo into remote repo

git pull
Merges all changes from remote repo into local repo

git status
see changed/added files, see changes staged for commit, see sync status with remote


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Version Control: Git

To be included in a commit, a new or modified file in workspace must be added to the Git **staging area** with **git add**

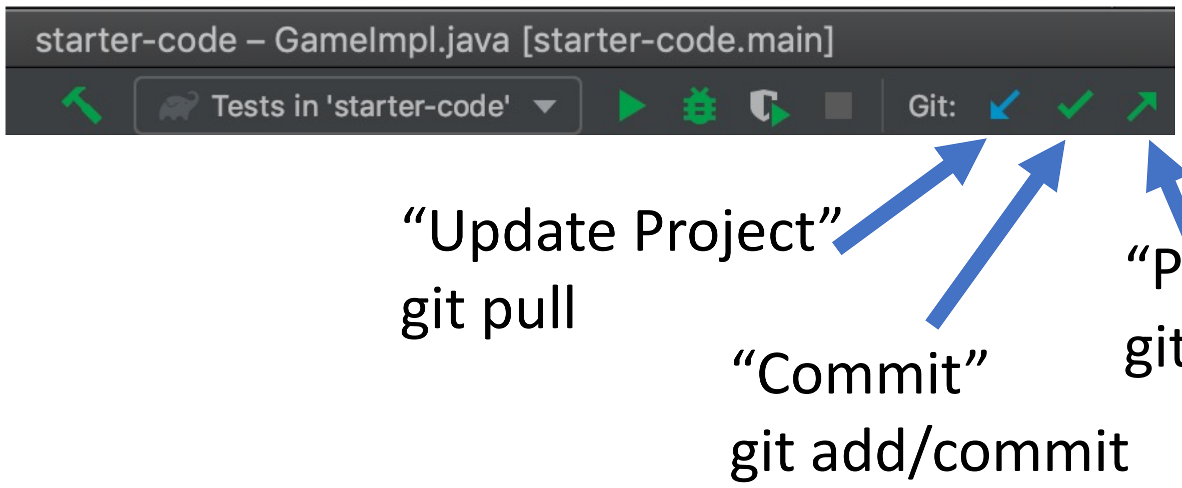
Workflow:

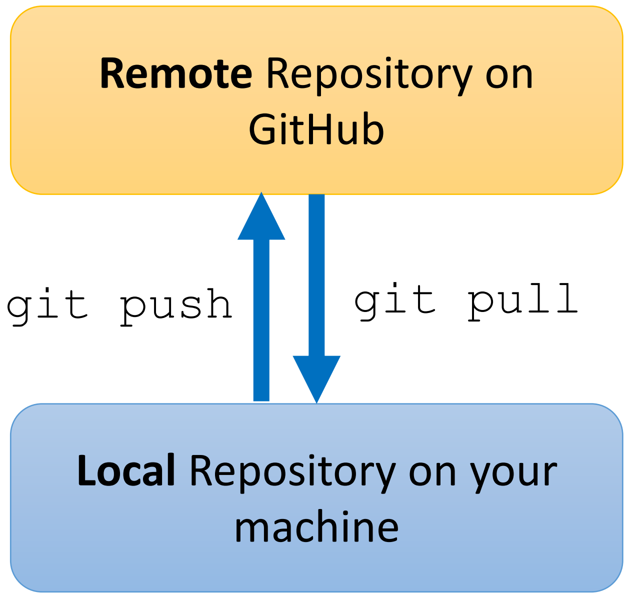
- Working locally, modify README.md
- See a list of changes: **git status**
- Add files to the staging area: **git add README.m**d
- Commit to the local repo: **git commit –m ”modified readme”**
- Merge local commits into remote repo: **git push**

Version Control: Git

Run git commands in a shell or a graphical interface (e.g., SourceTree)

- Controls are built into many IDEs, including IntelliJ




Remote Repository on GitHub
git push
git pull
Local Repository on your machine

```
git add <files>
git commit –m "made these changes"
```

Version Control: Git

Git enables **fine-grained version control**

Case: In iteration 755 I added feature x to my fabulous program

- … which uses the ‘doSuperStuff’ method that I spotted a bug in!

With Git:

- Add (stage) just the files related to the bug fix; commit bugfix
- Then add the rest of the files and commit feature

Version Control: Git

git log

- See a list of previous commit messages (and hashes), going back in time

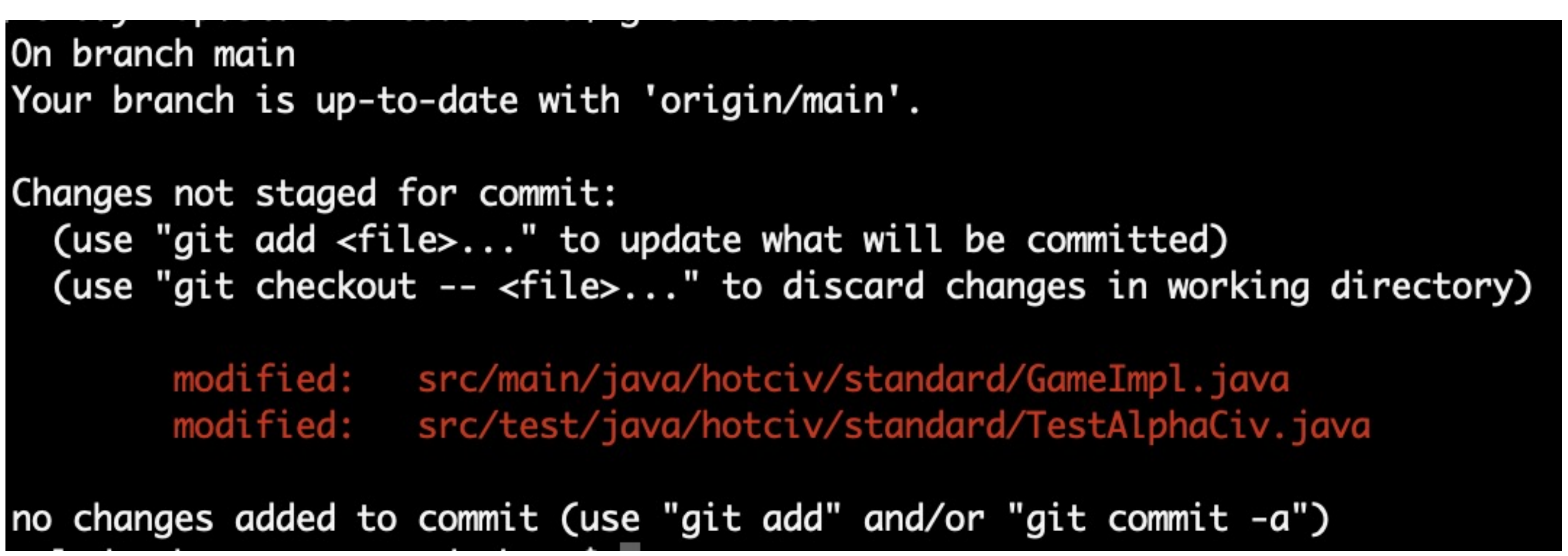
git log -3

- See the 3 most recent commit messages

Version Control: Git Workflow

Just finished a TDD iteration, all tests passing…

```
git status
```

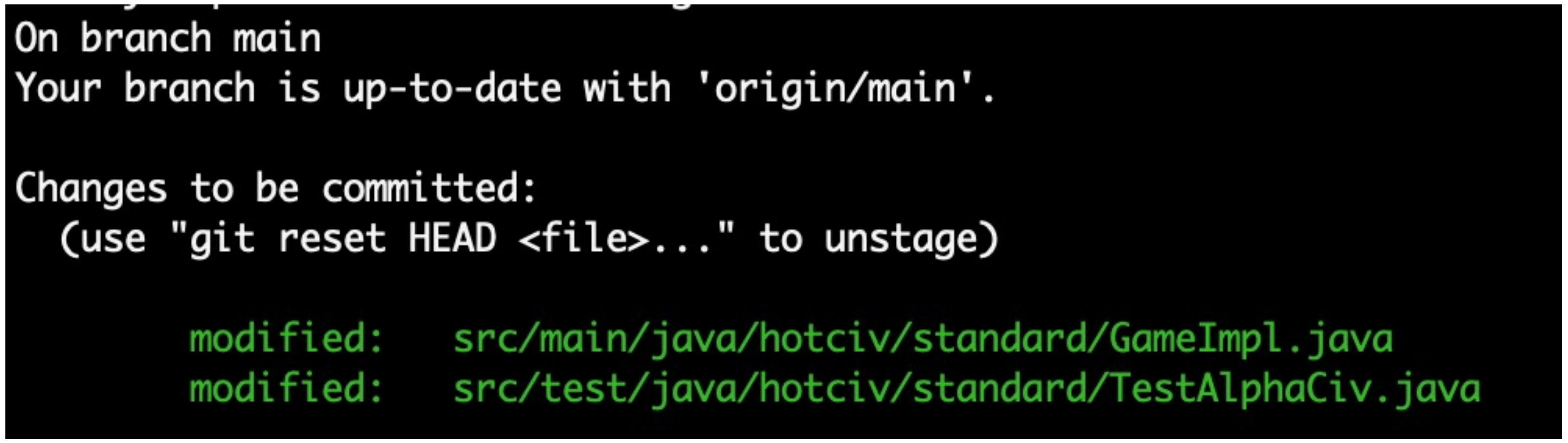


```
On branch main
Your branch is up-to-date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   src/main/java/hotciv/standard/GameImpl.java
        modified:   src/test/java/hotciv/standard/TestAlphaCiv.java

no changes added to commit (use "git add" and/or "git commit -a")
```

Version Control: Git Workflow

Just finished a TDD iteration, all tests passing…

```
git add .
git status
```



```
On branch main
Your branch is up-to-date with 'origin/main'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   src/main/java/hotciv/standard/GameImpl.java
        modified:   src/test/java/hotciv/standard/TestAlphaCiv.java
```

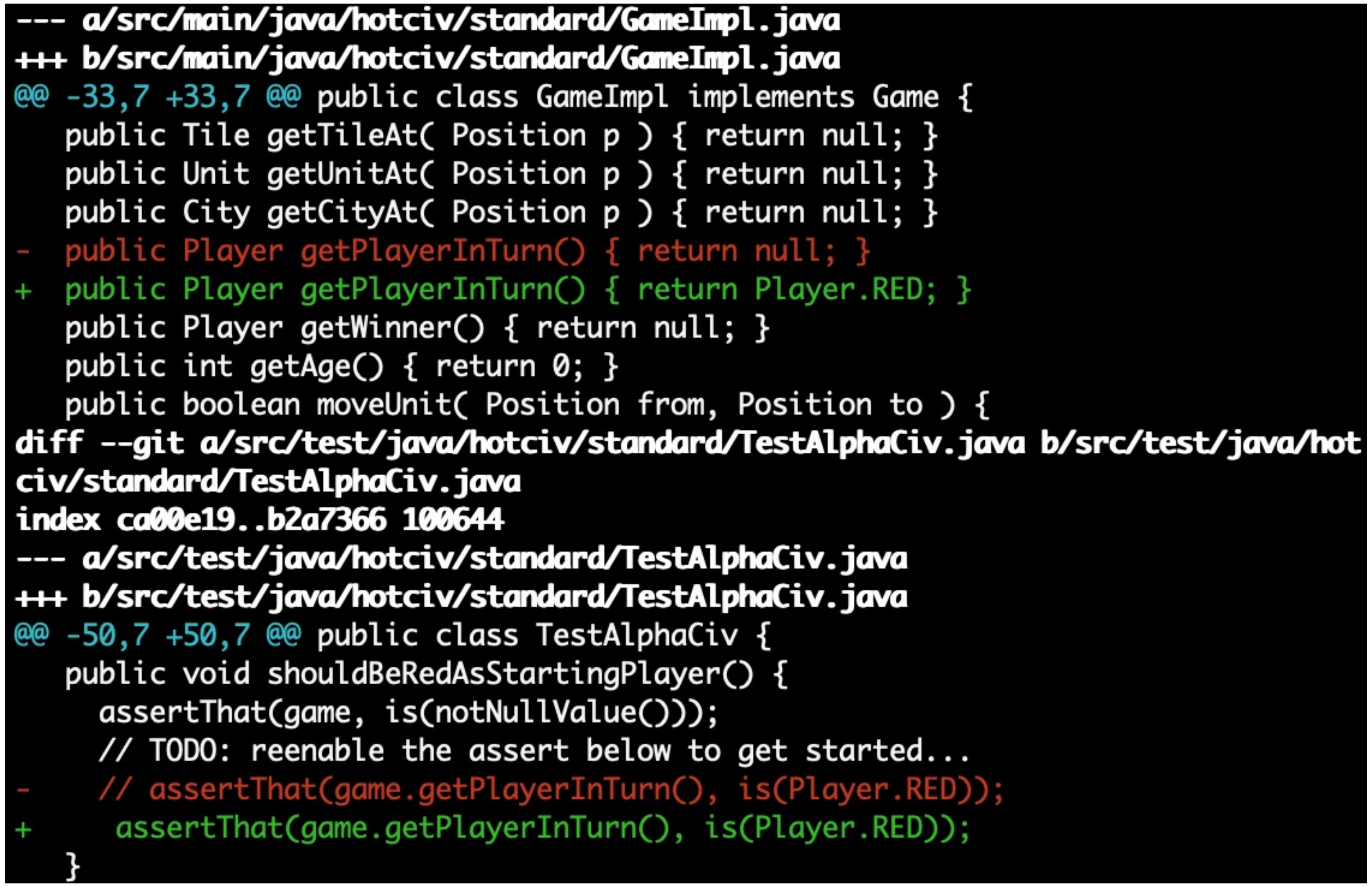
Version Control: Git Workflow

Just finished a TDD iteration, all tests passing…

```
git commit -m “first player is red”
git push
```

Useful Git Commands

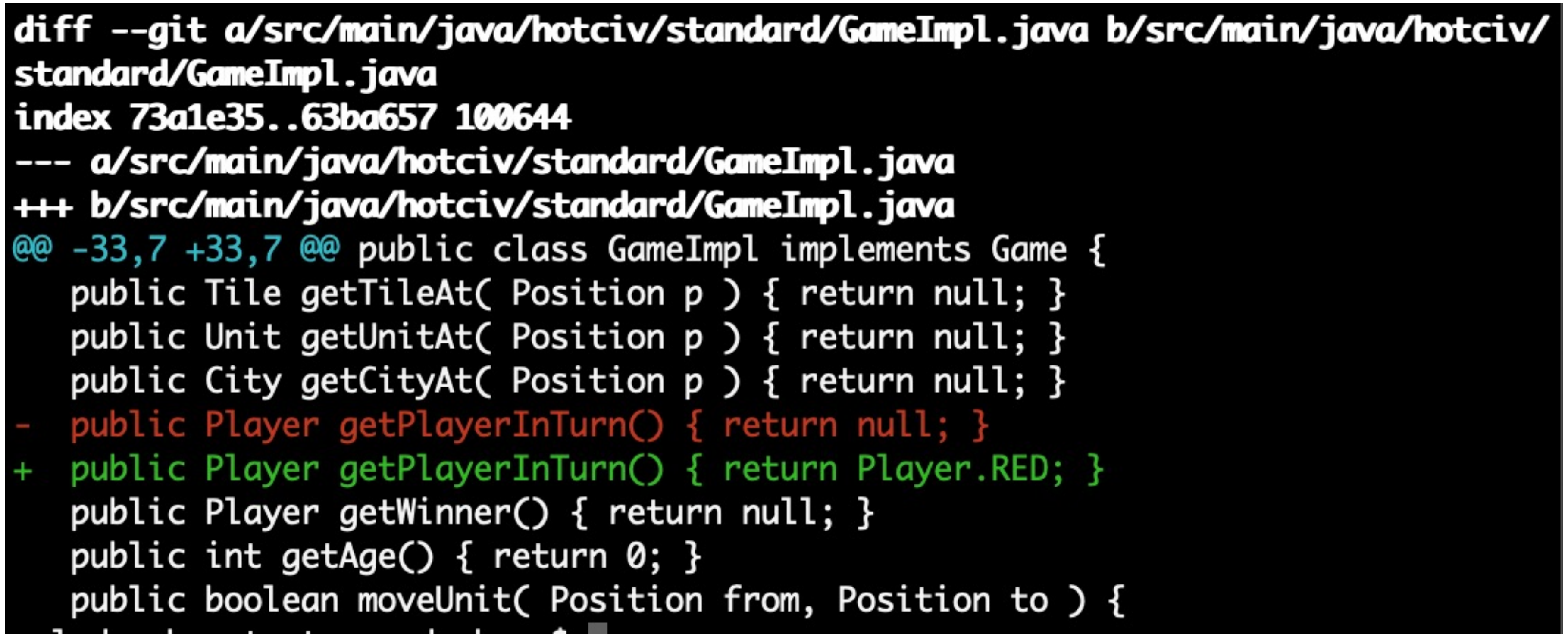
`git diff`



```
--- a/src/main/java/hotciv/standard/GameImpl.java
+++ b/src/main/java/hotciv/standard/GameImpl.java
@@ -33,7 +33,7 @@ public class GameImpl implements Game {
   public Tile getTileAt( Position p ) { return null; }
   public Unit getUnitAt( Position p ) { return null; }
   public City getCityAt( Position p ) { return null; }
-  public Player getPlayerInTurn() { return null; }
+  public Player getPlayerInTurn() { return Player.RED; }
   public Player getWinner() { return null; }
   public int getAge() { return 0; }
   public boolean moveUnit( Position from, Position to ) {
diff --git a/src/test/java/hotciv/standard/TestAlphaCiv.java b/src/test/java/hot
civ/standard/TestAlphaCiv.java
index ca00e19..b2a7366 100644
--- a/src/test/java/hotciv/standard/TestAlphaCiv.java
+++ b/src/test/java/hotciv/standard/TestAlphaCiv.java
@@ -50,7 +50,7 @@ public class TestAlphaCiv {
   public void shouldBeRedAsStartingPlayer() {
     assertThat(game, is(notNullValue()));
     // TODO: reenable the assert below to get started...
-    // assertThat(game.getPlayerInTurn(), is(Player.RED));
+     assertThat(game.getPlayerInTurn(), is(Player.RED));
   }
```

Useful Git Commands

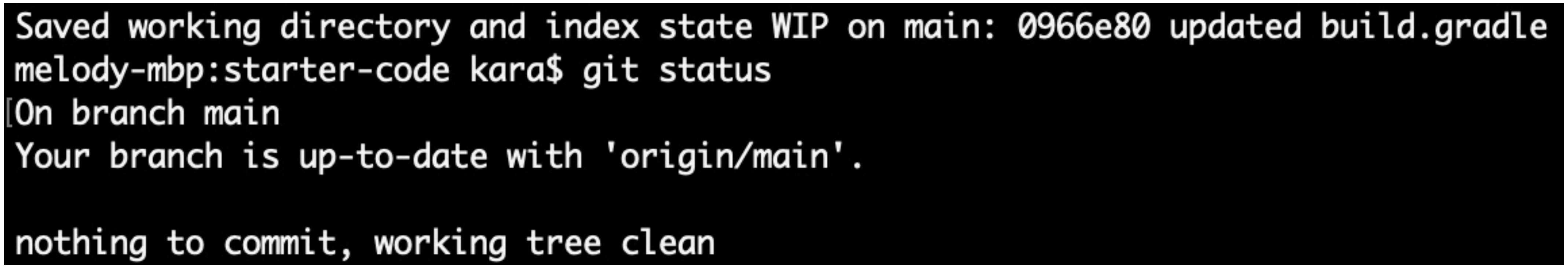
```
git diff src/main/java/hotciv/standard/GameImpl.java
```



```
diff --git a/src/main/java/hotciv/standard/GameImpl.java b/src/main/java/hotciv/
standard/GameImpl.java
index 73a1e35..63ba657 100644
--- a/src/main/java/hotciv/standard/GameImpl.java
+++ b/src/main/java/hotciv/standard/GameImpl.java
@@ -33,7 +33,7 @@ public class GameImpl implements Game {
   public Tile getTileAt( Position p ) { return null; }
   public Unit getUnitAt( Position p ) { return null; }
   public City getCityAt( Position p ) { return null; }
-  public Player getPlayerInTurn() { return null; }
+  public Player getPlayerInTurn() { return Player.RED; }
   public Player getWinner() { return null; }
   public int getAge() { return 0; }
   public boolean moveUnit( Position from, Position to ) {
```

Useful Git Commands

```
git stash
```



```
Saved working directory and index state WIP on main: 0966e80 updated build.gradle
melody-mbp:starter-code kara$ git status
[On branch main
Your branch is up-to-date with 'origin/main'.

nothing to commit, working tree clean
```

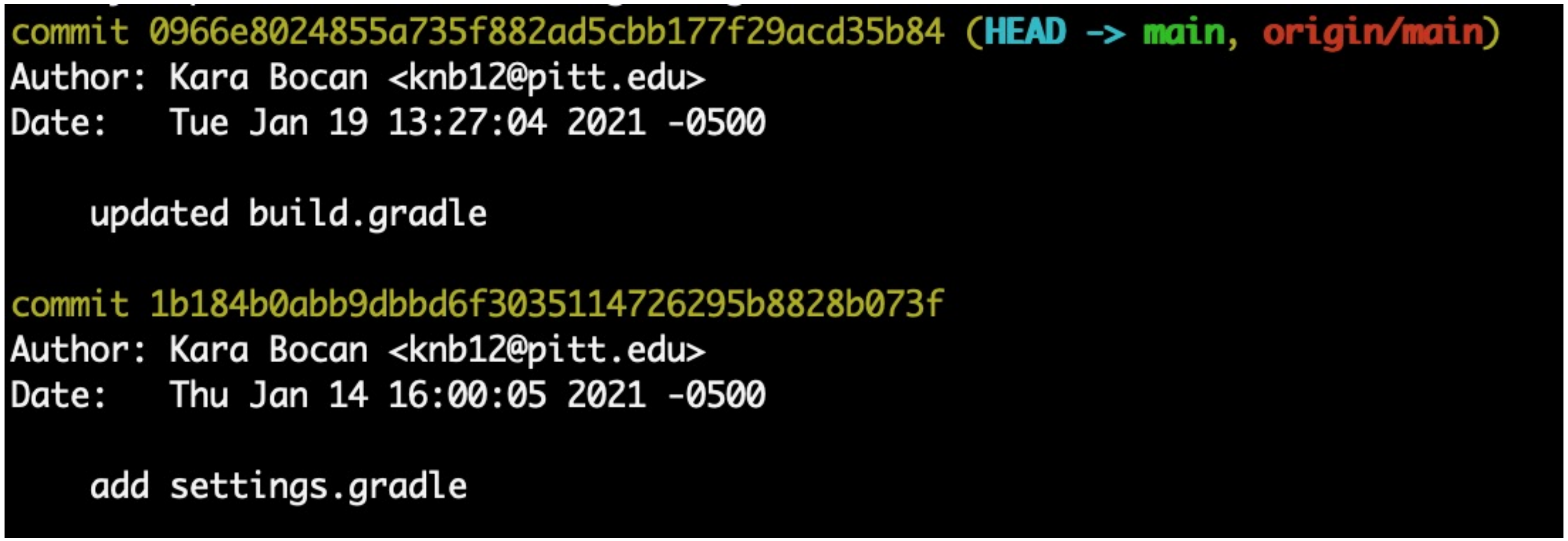
Temporarily undo **and store** all changes since the last commit

```
git stash apply
```

Reapply changes stored in stash

Useful Git Commands

`git log`

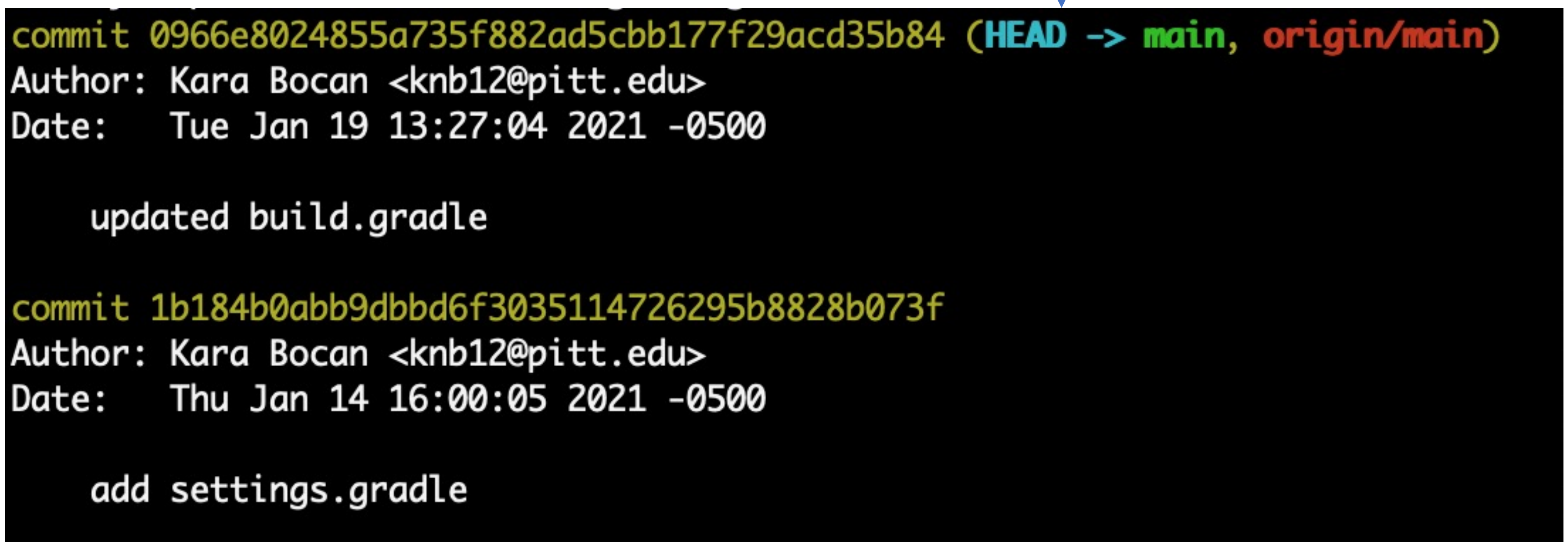


```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle

commit 1b184b0abb9dbbd6f3035114726295b8828b073f
Author: Kara Bocan <knb12@pitt.edu>
Date:   Thu Jan 14 16:00:05 2021 -0500

    add settings.gradle
```

Useful Git Commands

HEAD points to the current snapshot

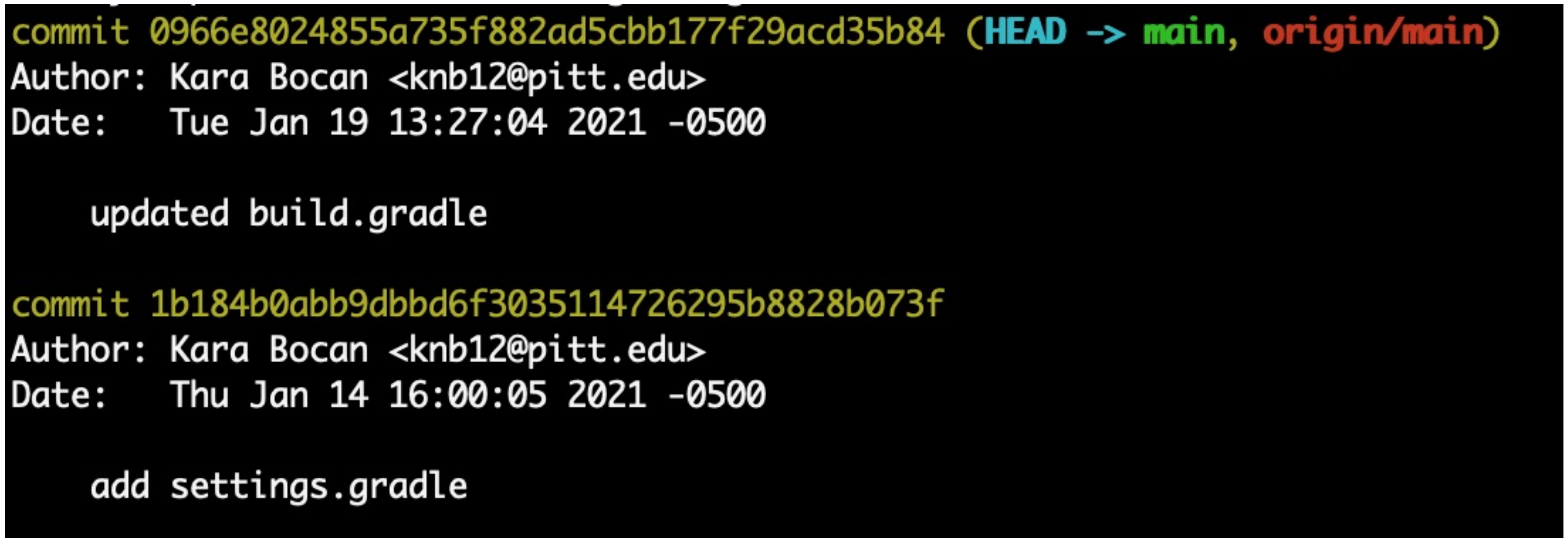
```
git log
```



```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle

commit 1b184b0abb9dbbd6f3035114726295b8828b073f
Author: Kara Bocan <knb12@pitt.edu>
Date:   Thu Jan 14 16:00:05 2021 -0500

    add settings.gradle
```

Useful Git Commands

`git checkout <commit id>`

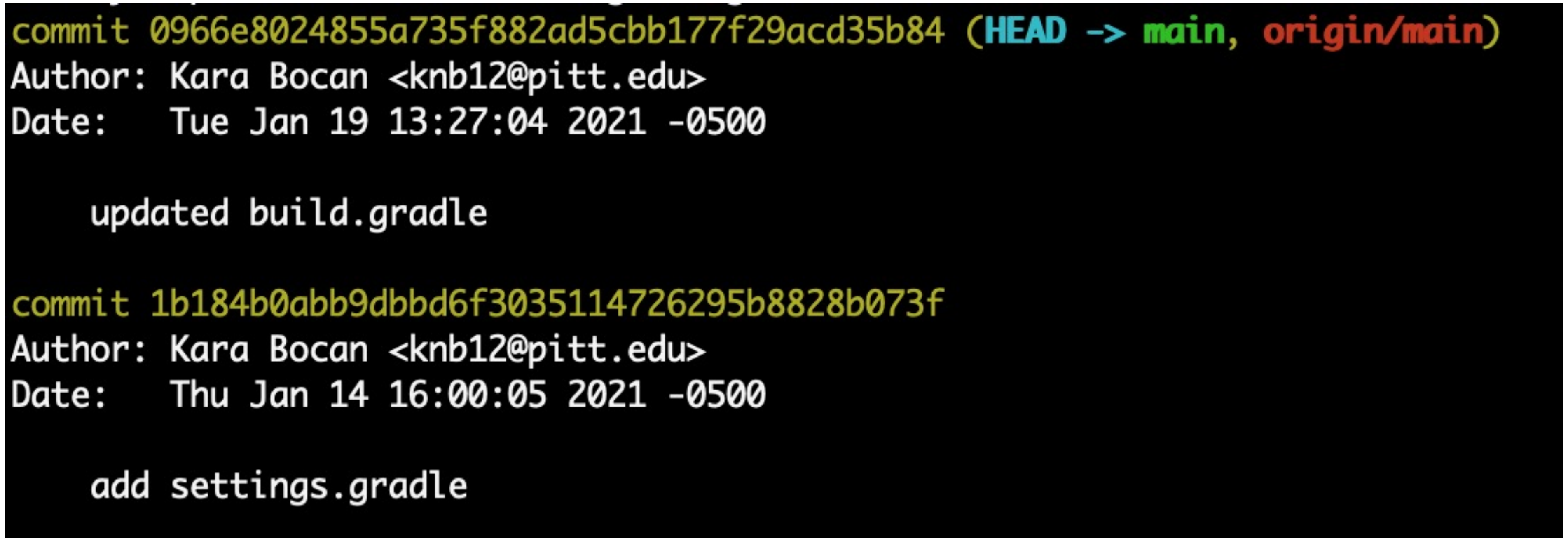


```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle

commit 1b184b0abb9dbbd6f3035114726295b8828b073f
Author: Kara Bocan <knb12@pitt.edu>
Date:   Thu Jan 14 16:00:05 2021 -0500

    add settings.gradle
```

Only use for inspection – causes "detached HEAD state"
Only make changes when HEAD is on the most recent commit

Useful Git Commands

```
git revert <commit id>
```

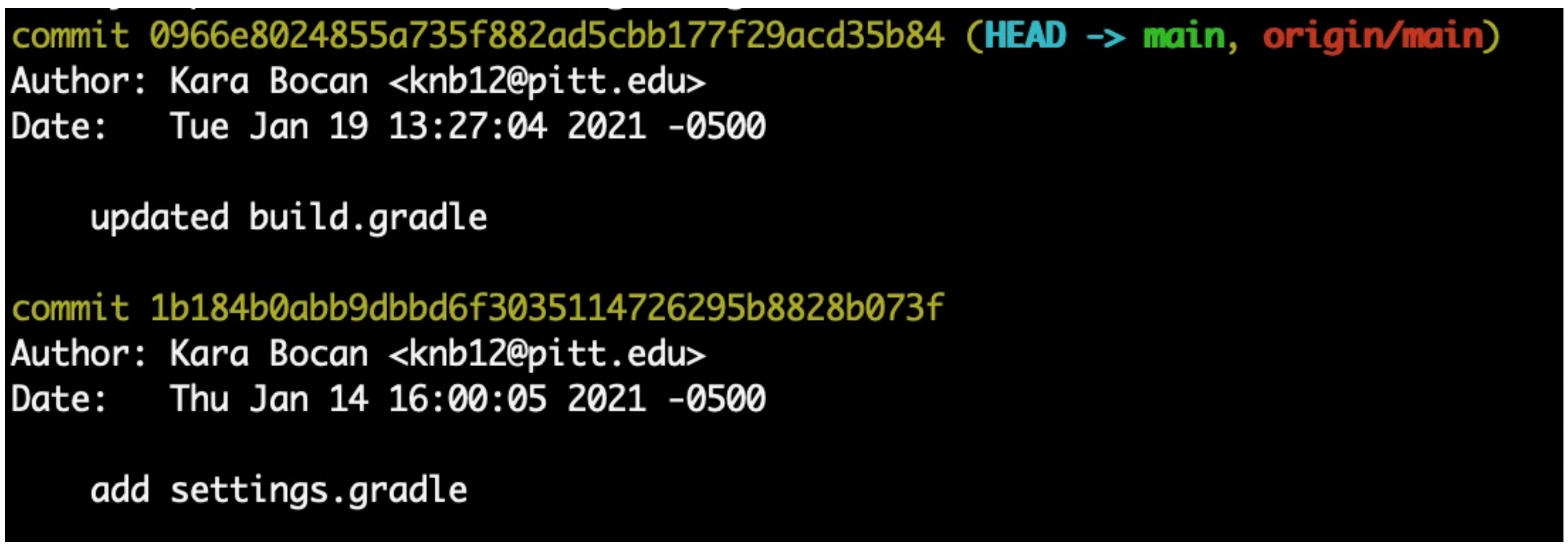


```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle

commit 1b184b0abb9dbbd6f3035114726295b8828b073f
Author: Kara Bocan <knb12@pitt.edu>
Date:   Thu Jan 14 16:00:05 2021 -0500

    add settings.gradle
```

Undoes the specified commit
(makes a new commit with reverse changes)

Useful Git Commands

```
git checkout -- <file>
```

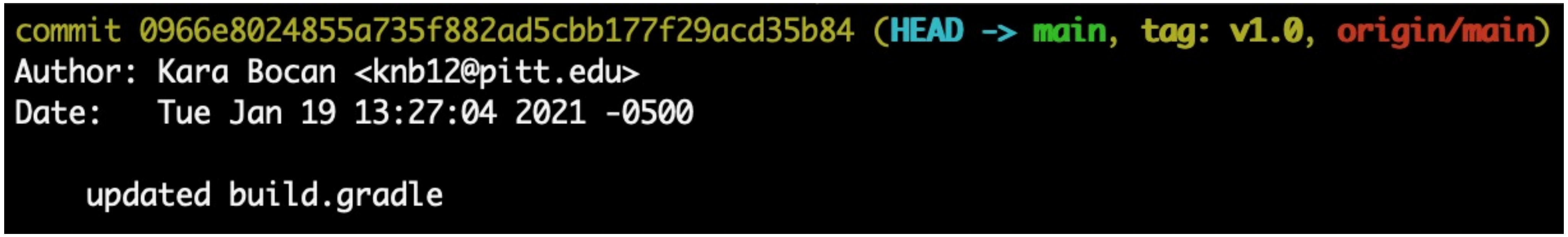


```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle

commit 1b184b0abb9dbbd6f3035114726295b8828b073f
Author: Kara Bocan <knb12@pitt.edu>
Date:   Thu Jan 14 16:00:05 2021 -0500

    add settings.gradle
```

Discard changes to <file> since the last commit

Useful Git Commands

```
git tag “v1.0”
```



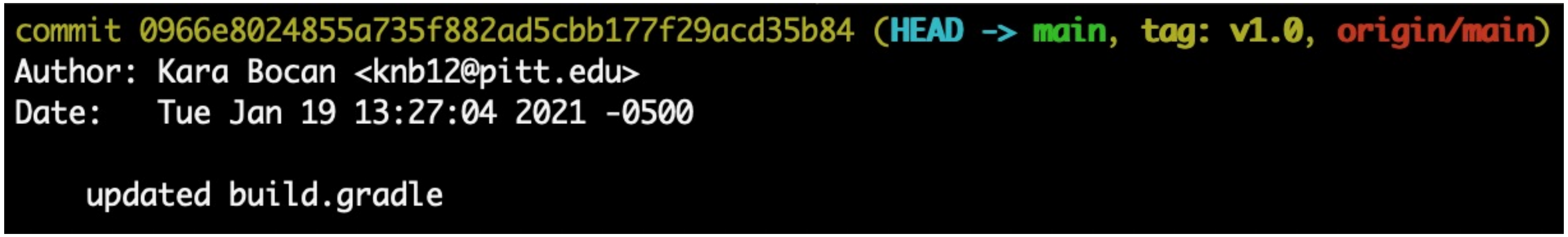
```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, tag: v1.0, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle
```

Tag a commit with a more readable name, then:

```
git checkout v1.0
```

Useful Git Commands

```
git tag "v1.0"
```



```
commit 0966e8024855a735f882ad5cbb177f29acd35b84 (HEAD -> main, tag: v1.0, origin/main)
Author: Kara Bocan <knb12@pitt.edu>
Date:   Tue Jan 19 13:27:04 2021 -0500

    updated build.gradle
```

Tag information is stored locally – remember to push to origin:

```
git push origin <tag name>
```

or

```
git push --tags
```

best practice

Useful Git Commands

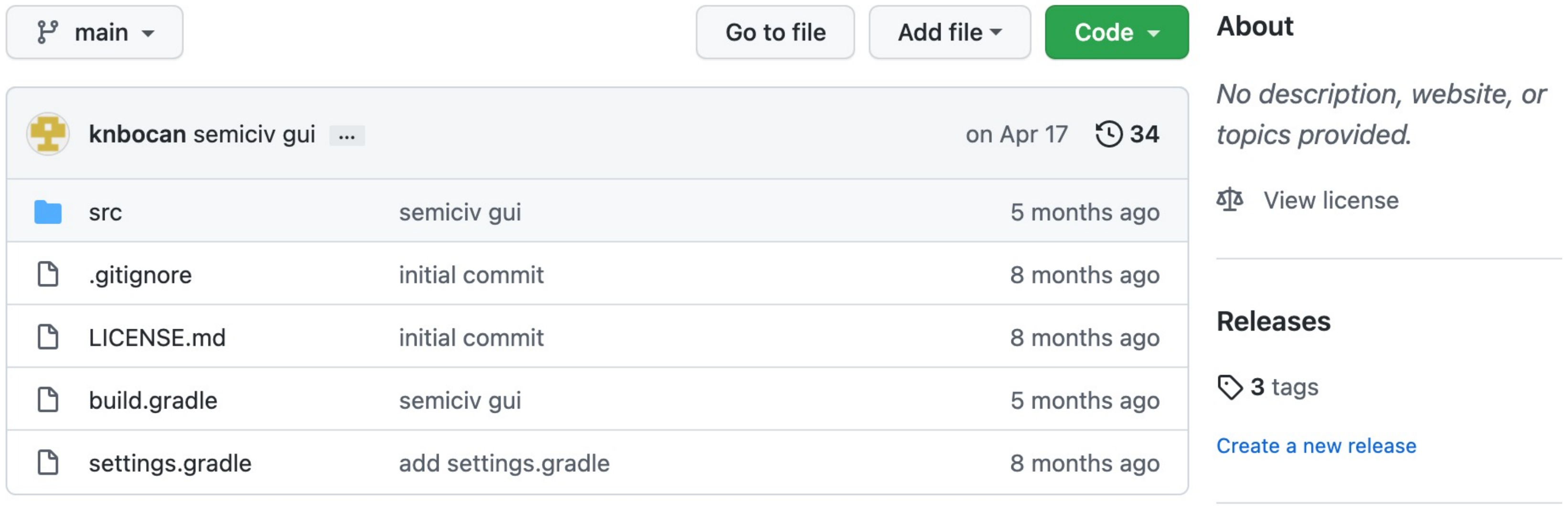
```
git pull
```

- Merge any new commits from the remote into your local repo

```
git fetch
```

- Get branches and any changes from the origin but **do not merge them locally,** can ‘git diff main origin/main’

Useful Git Commands



main | Go to file | Add file | Code

knbocan semiciv gui ... on Apr 17 34

src	semiciv gui	5 months ago
.gitignore	initial commit	8 months ago
LICENSE.md	initial commit	8 months ago
build.gradle	semiciv gui	5 months ago
settings.gradle	add settings.gradle	8 months ago

About

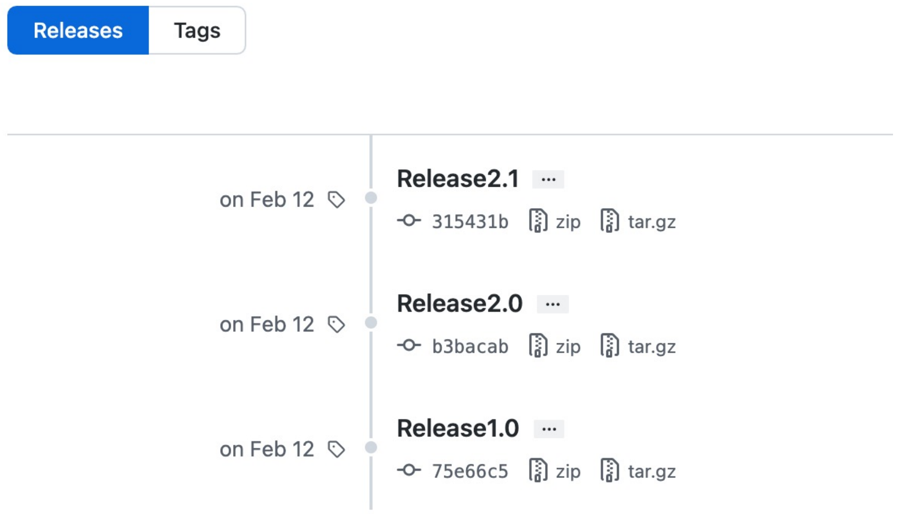
No description, website, or topics provided.

View license

Releases

3 tags

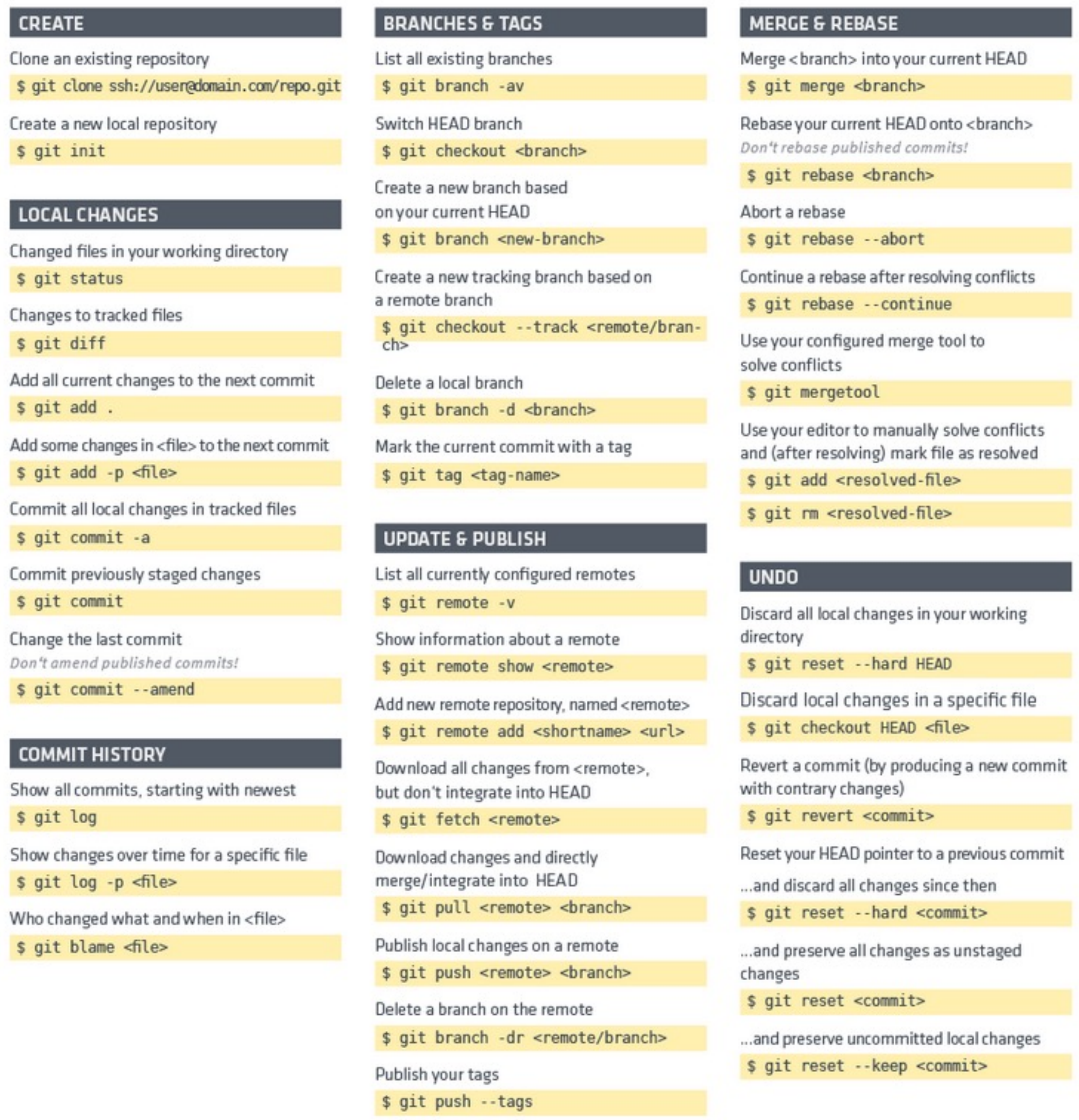
Create a new release



Releases | Tags

on Feb 12	**Release2.1**	315431b zip tar.gz
on Feb 12	**Release2.0**	b3bacab zip tar.gz
on Feb 12	**Release1.0**	75e66c5 zip tar.gz

Git Commands Summary



CREATE

Clone an existing repository

```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repository

```
$ git init
```

LOCAL CHANGES

Changed files in your working directory

```
$ git status
```

Changes to tracked files

```
$ git diff
```

Add all current changes to the next commit

```
$ git add .
```

Add some changes in <file> to the next commit

```
$ git add -p <file>
```

Commit all local changes in tracked files

```
$ git commit -a
```

Commit previously staged changes

```
$ git commit
```

Change the last commit

Don't amend published commits!

```
$ git commit --amend
```

COMMIT HISTORY

Show all commits, starting with newest

```
$ git log
```

Show changes over time for a specific file

```
$ git log -p <file>
```

Who changed what and when in <file>

```
$ git blame <file>
```

BRANCHES & TAGS

List all existing branches

```
$ git branch -av
```

Switch HEAD branch

```
$ git checkout <branch>
```

Create a new branch based on your current HEAD

```
$ git branch <new-branch>
```

Create a new tracking branch based on a remote branch

```
$ git checkout --track <remote/bran-
ch>
```

Delete a local branch

```
$ git branch -d <branch>
```

Mark the current commit with a tag

```
$ git tag <tag-name>
```

UPDATE & PUBLISH

List all currently configured remotes

```
$ git remote -v
```

Show information about a remote

```
$ git remote show <remote>
```

Add new remote repository, named <remote>

```
$ git remote add <shortname> <url>
```

Download all changes from <remote>, but don't integrate into HEAD

```
$ git fetch <remote>
```

Download changes and directly merge/integrate into HEAD

```
$ git pull <remote> <branch>
```

Publish local changes on a remote

```
$ git push <remote> <branch>
```

Delete a branch on the remote

```
$ git branch -dr <remote/branch>
```

Publish your tags

```
$ git push --tags
```

MERGE & REBASE

Merge <branch> into your current HEAD

```
$ git merge <branch>
```

Rebase your current HEAD onto <branch>

Don't rebase published commits!

```
$ git rebase <branch>
```

Abort a rebase

```
$ git rebase --abort
```

Continue a rebase after resolving conflicts

```
$ git rebase --continue
```

Use your configured merge tool to solve conflicts

```
$ git mergetool
```

Use your editor to manually solve conflicts and (after resolving) mark file as resolved

```
$ git add <resolved-file>
```

```
$ git rm <resolved-file>
```

UNDO

Discard all local changes in your working directory

```
$ git reset --hard HEAD
```

Discard local changes in a specific file

```
$ git checkout HEAD <file>
```

Revert a commit (by producing a new commit with contrary changes)

```
$ git revert <commit>
```

Reset your HEAD pointer to a previous commit

...and discard all changes since then

```
$ git reset --hard <commit>
```

...and preserve all changes as unstaged changes

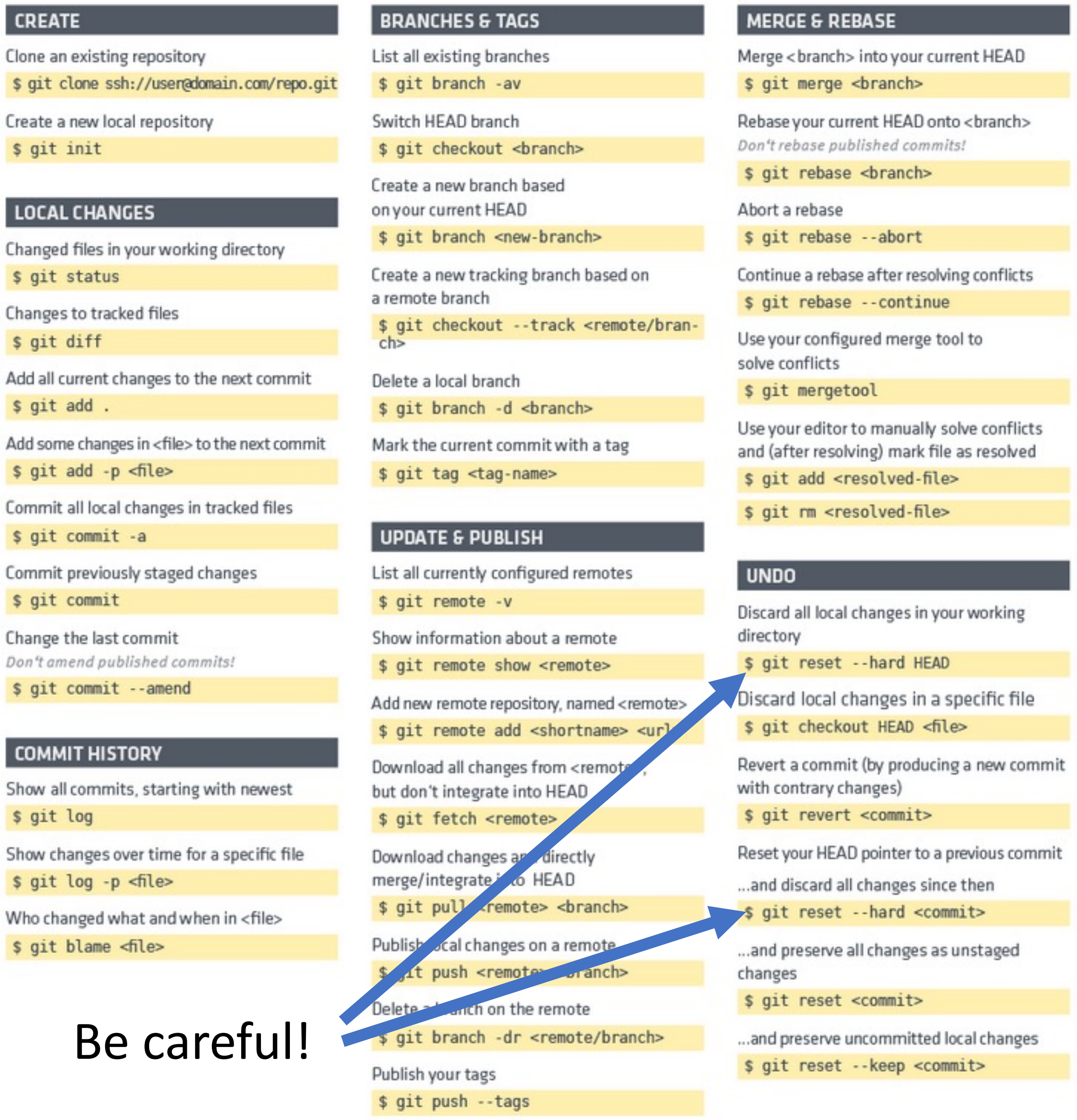
```
$ git reset <commit>
```

...and preserve uncommitted local changes

```
$ git reset --keep <commit>
```

Git Commands Summary

Generally best to undo changes with revert so there is still history in git log



CREATE

Clone an existing repository
```
$ git clone ssh://user@domain.com/repo.git
```
Create a new local repository
```
$ git init
```

LOCAL CHANGES

Changed files in your working directory
```
$ git status
```
Changes to tracked files
```
$ git diff
```
Add all current changes to the next commit
```
$ git add .
```
Add some changes in <file> to the next commit
```
$ git add -p <file>
```
Commit all local changes in tracked files
```
$ git commit -a
```
Commit previously staged changes
```
$ git commit
```
Change the last commit
Don't amend published commits!
```
$ git commit --amend
```

COMMIT HISTORY

Show all commits, starting with newest
```
$ git log
```
Show changes over time for a specific file
```
$ git log -p <file>
```
Who changed what and when in <file>
```
$ git blame <file>
```

BRANCHES & TAGS

List all existing branches
```
$ git branch -av
```
Switch HEAD branch
```
$ git checkout <branch>
```
Create a new branch based on your current HEAD
```
$ git branch <new-branch>
```
Create a new tracking branch based on a remote branch
```
$ git checkout --track <remote/bran-
ch>
```
Delete a local branch
```
$ git branch -d <branch>
```
Mark the current commit with a tag
```
$ git tag <tag-name>
```

UPDATE & PUBLISH

List all currently configured remotes
```
$ git remote -v
```
Show information about a remote
```
$ git remote show <remote>
```
Add new remote repository, named <remote>
```
$ git remote add <shortname> <url>
```
Download all changes from <remote>, but don't integrate into HEAD
```
$ git fetch <remote>
```
Download changes and directly merge/integrate into HEAD
```
$ git pull <remote> <branch>
```
Publish local changes on a remote
```
$ git push <remote> <branch>
```
Delete a branch on the remote
```
$ git branch -dr <remote/branch>
```
Publish your tags
```
$ git push --tags
```

MERGE & REBASE

Merge <branch> into your current HEAD
```
$ git merge <branch>
```
Rebase your current HEAD onto <branch>
Don't rebase published commits!
```
$ git rebase <branch>
```
Abort a rebase
```
$ git rebase --abort
```
Continue a rebase after resolving conflicts
```
$ git rebase --continue
```
Use your configured merge tool to solve conflicts
```
$ git mergetool
```
Use your editor to manually solve conflicts and (after resolving) mark file as resolved
```
$ git add <resolved-file>
$ git rm <resolved-file>
```

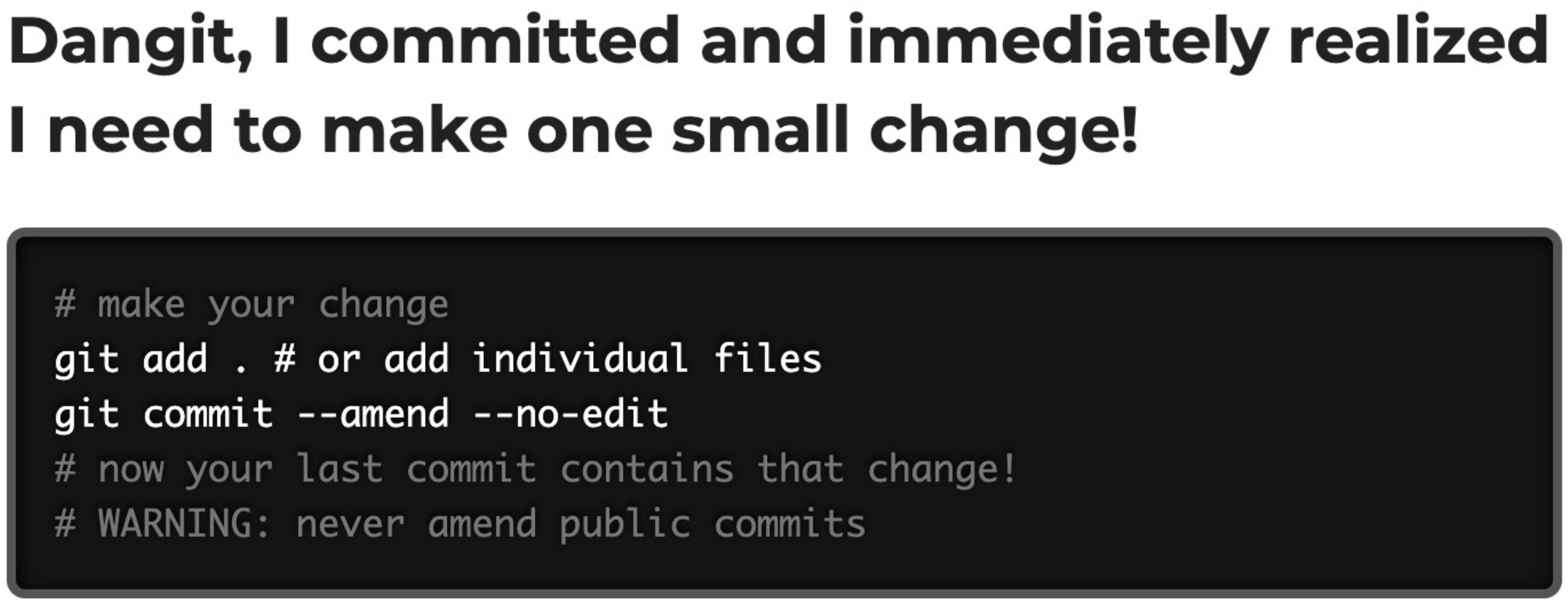
UNDO

Discard all local changes in your working directory
```
$ git reset --hard HEAD
```
Discard local changes in a specific file
```
$ git checkout HEAD <file>
```
Revert a commit (by producing a new commit with contrary changes)
```
$ git revert <commit>
```
Reset your HEAD pointer to a previous commit
...and discard all changes since then
```
$ git reset --hard <commit>
```
...and preserve all changes as unstaged changes
```
$ git reset <commit>
```
...and preserve uncommitted local changes
```
$ git reset --keep <commit>
```

Be careful!

Dangit, Git!

https://dangitgit.com/



Dangit, I committed and immediately realized I need to make one small change!

```
# make your change
git add . # or add individual files
git commit --amend --no-edit
# now your last commit contains that change!
# WARNING: never amend public commits
```

Version Control: Gitignore

Don’t clutter your team’s repo!

A ‘.gitignore’ file is included with the starter code

- /build ignore all that gradle produces in the build folder
- *.iml ignore IntelliJ files

http://gitignore.io/

```
build/
gradle/
.gradle/
.idea/
out/

gradlew
gradlew.bat

*~
*.bak
*.idea
*.iml
```

Version Control: Best Practices

Commit **related** changes

- Fixing two bugs should lead to two commits

Commit often

- ‘Take small steps’, break big into small, one step at a time
- Keeps a safe version to checkout in case of ‘Do Over’

Use **informative commit messages**! You may need to search through the log for a previous commit to return to

Push often to sync your work

- Can re-clone your remote repo if something goes wrong

Version Control: Best Practices

Avoid pushing broken commits!

- Can commit broken builds locally if changing tasks, to preserve work in progress
- Pushed commits should reflect a finished step/feature/bugfix
    - **All tests should pass in pushed code**
- That said, best practice is that commits also have all tests passing

Summary

git status

git add <. or file names>

git commit

git push

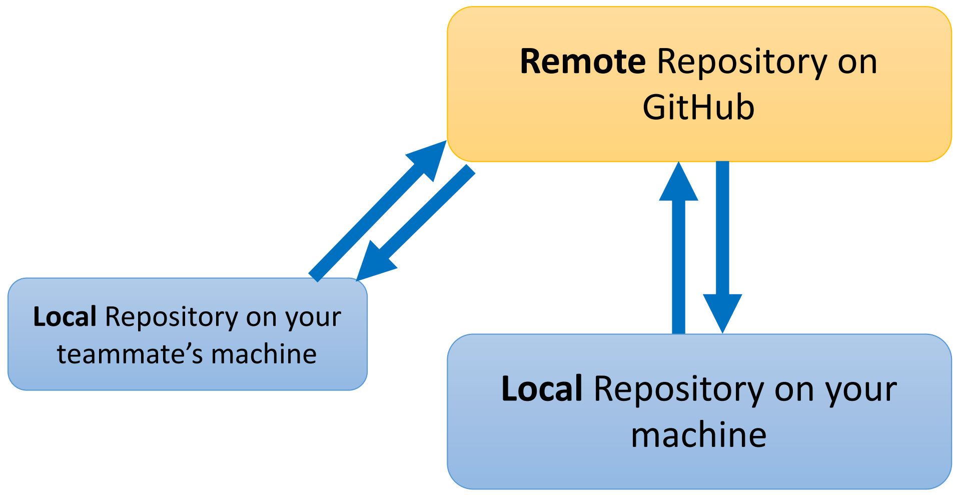
git pull

git tag

git log

git checkout

git diff


Remote Repository on GitHub
Local Repository on your teammate's machine
Local Repository on your machine

Next time:

Branching

Build management