

Lecture Notes for Lecture 5 of CS 5500
(Foundations of Software Engineering) for the
Spring 2021 session at the Northeastern
University San Francisco Bay Area Campuses.

Managing Project Content

Philip Gust,
Clinical Instructor
Department of Computer Science

<http://www.ccis.northeastern.edu/home/pgust/classes/cs5500/2021/Spring/index.html>

Managing Project Content

Review of Lecture 4

- In lecture 4, we studied a different type of software development methodology that began to take shape with the advent of the open source software (OSS) movement.
- We learned that the term *open source* refers to a computer software whose source code is available to the general public for use or modification to its original design.
- Open source software development is meant to be a collaborative effort, where programmers improve upon the source code and share the changes within the community.

Managing Project Content

Review of Lecture 4

- We saw that the impetus for the OSS movement a seminal 1997 by Eric S. Raymond entitled, “The Cathedral and the Bazaar” that described two software development models:
 - **Cathedral model:** source code is available with each software release, but code developed between releases is restricted to an exclusive group of software developers (examples: GNU Emacs, GCC).
 - **Bazaar model:** code is developed over the Internet in view of the public. The central thesis is that, “given enough eyeballs, all bug are shallow,” (“Linus’s Law” after Linux creator Linus Torvalds).

Managing Project Content

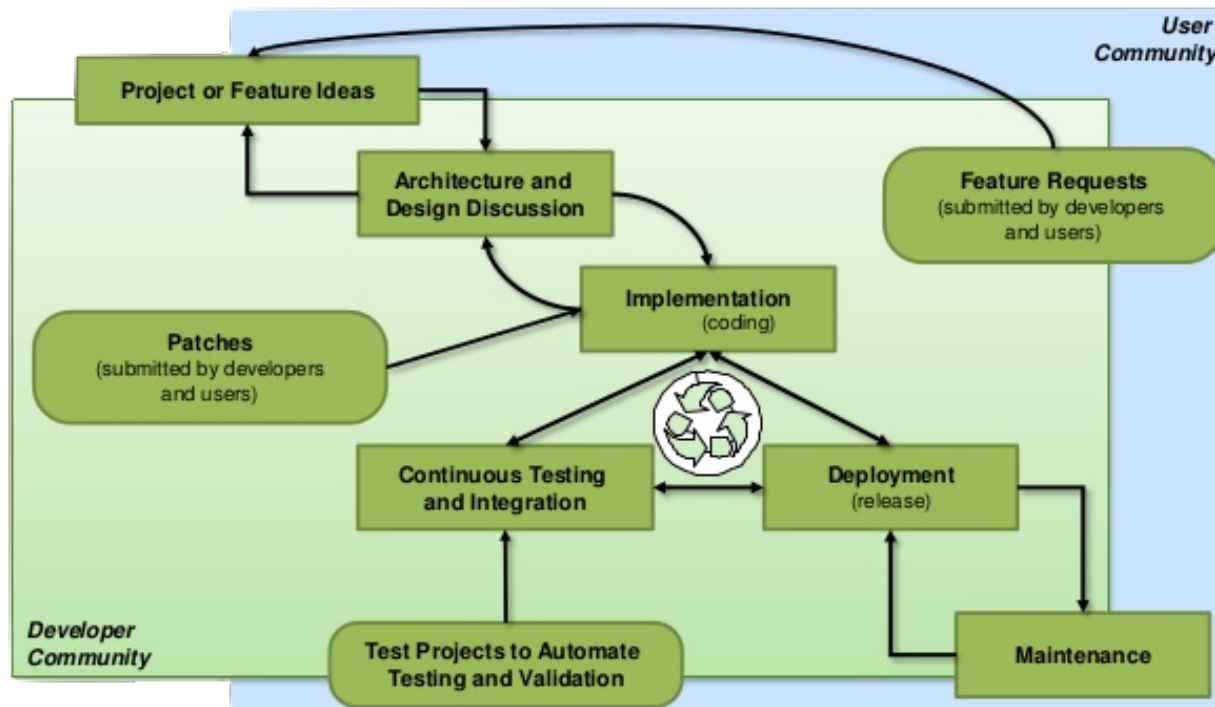
Review of Lecture 4

- The three tenants of the OSS model are:
 - OSS is a decentralized software development model that encourages open collaboration among a community of developers and end-users.
 - Source code and all work products are freely available. Developers can improve upon the source code and freely share changes.
 - The development process and all discussions about the project and its development are transparent and visible to the entire community.

Managing Project Content

Review of Lecture 4

- Under an open-source model the end-user and developer communities work together collaboratively.



Managing Project Content

Introduction

- Before looking at other aspects of a software project, we should first consider how we will organize and manage the content that will be created during the project.
- The content created by the project will need to be stored in a way that is accessible by all members of the project team and can be referred to as needed during the project.
- What content needs to be captured and stored during the project will depend on the kind of project and the software development model being followed.

Managing Project Content

Introduction

- For example projects employing an agile software methodology such as SCRUM or XP will generate little in the way of requirements, functional and design specifications.
- These software methodologies consider everything but the code itself to be ephemeral, and as little effort as possible is spent creating or managing it.
- The kinds of projects undertaken using these methodologies have few external content management requirements, so only a source code repository plays a significant role.

Managing Project Content

Introduction

- On the other hand, projects undertaken in organizations that follow other software methodologies may require external documentation as part of the project deliverables.
- This is especially true for projects undertaken on behalf of
 - Regulated industries (e.g. banks, pharma, hospitals)
 - Financial institutions (e.g. brokerages)
 - Military and military contractors (e.g. Air Force, Boeing, Lockheed)
 - Governmental and inter-governmental bodies (e.g. HHS, NASA)
 - Grant-making organizations (e.g. NSF, DARPA, NIH)
- In these cases, a content management and maintenance plan is a required early deliverable from the project.

Managing Project Content

Introduction

- Over the next several lectures, we will look at the kinds of content that software development teams create during a project, and techniques and platforms for managing it.
- Even for methodologies that minimize the requirement for long-term capture and management of project content, it is useful to develop a plan for how project content is shared.
- We will begin with a type of content that every methodology must manage over the long-term: source code.

Managing Project Content

Introduction to source code management

- Source code is the most quickly-changing content in a software project, because it is frequently accessed and modified by developers.
- The ability to manage source code for a software projects is critical for several reasons.
 - Multiple threads of development often require modification to overlapping sets of source code, so it is critical to coordinate changes, and identify and resolve conflicts.
 - Multiple versions of the software are under development, and changes must be integrated into versions of shared source modules without impacting other versions.
 - When defects occur, it may be necessary to regress to the last known good version of the software to identify the problem.

Managing Project Content

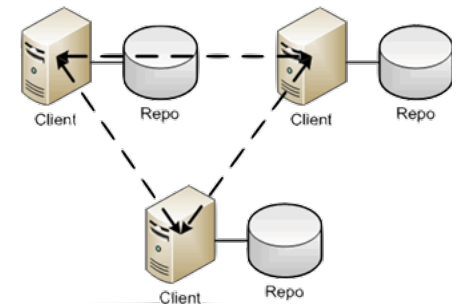
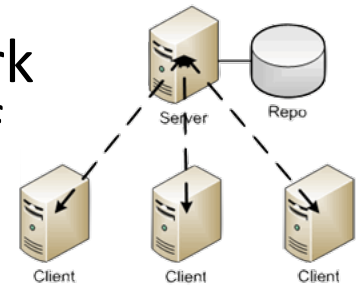
Introduction to source code management

- Traditional and agile software development projects are often co-located, so a centralized or client/server-based source code management system works well in these cases
- On the other hand, open-source development projects are usually highly distributed, with developers from many geographical locations and time zones.

Managing Project Content

Introduction to source code management

- Centralized resource management does not work well for open-source projects because groups of developers are not continuously connected.
- Developers working on a feature or a defect often require access to large portions of the code base locally and commit changes only when work is completed.
- A distributed or peer-to-peer source code management system is a better match for these developers to their working style.



Managing Project Content

Introduction to Git for managing source code

- The Git source code management system has become popular among open-source teams, and increasingly in the software developer community.
- Git is a distributed source code management system for tracking changes in source code during development.
- It was designed for coordinating work among developers, but can track changes in other types of files. Its goals include
 - speed
 - data integrity
 - support for distributed, non-linear workflows.



Managing Project Content

Introduction to Git for managing source code

- Linus Torvalds created Git in 2005 to manage Linux kernel code, with other developers contributing to the initial version.
- One of Torvalds' concerns was scalability, citing one system that took 30 second to apply a patch and update metadata.
- He wanted a system that would scale to the Linux community.
- Synchronizing with fellow maintainers would require 250 such actions at once and should take no more than 3 seconds.



Managing Project Content

Introduction to Git for managing source code

- Torvalds also wanted a system with strong distributed workflow, and strong safeguards against accidental or malicious corruption.
- Coding began on April 3, and GIT became self-hosting on April 7. On April 29 Git achieved 6.7 patches per second, and on June 16 Git managed the entire 2.6.12 release of Linux.

Managing Project Content

Characteristics of Git

- Strong support for non-linear development
 - Supports rapid branching and merging; branches are a light-weight reference to one commit. Given parental commits, the full branch structure can be constructed.
- Compatibility with existent systems and protocols
 - Supports HTTP, FTP, or a Git protocol over either a plain socket, or Secure Shell (ssh). Also has emulation for other popular source code management systems.
- Efficient handling of large projects
 - Very fast and scalable, orders of magnitude faster than popular systems. Fetching local version history 100x faster than from a remote server.

Managing Project Content

Characteristics of Git

- Cryptographic authentication of history
 - History stored so that ID of a commit depends on full development history leading up to it. Changes to old versions can be detected (similar to Merkle tree, with data at nodes, leaves).
- Toolkit-based design
 - Designed as a set of C programs and several wrapper shell scripts that make it easy to chain the components together
- Pluggable merge strategies
 - Well-defined model of an incomplete merge, with multiple algorithms for completing it; manual editing in worst case.

Managing Project Content

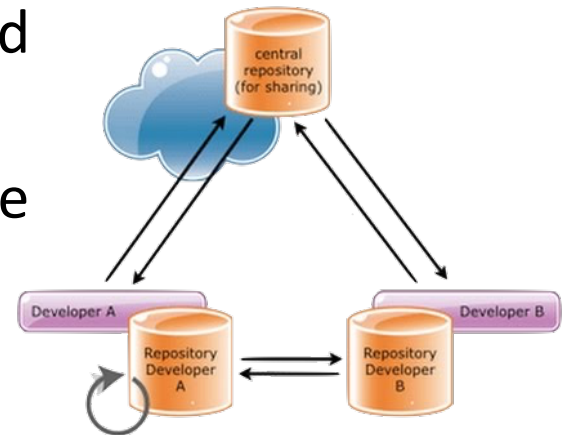
Git implicit revision relationships

- Git snapshots directory tree of files and does not explicitly record file revision relationships at any level below the source-code tree. Consequences include:
 - Change history of project as easy as for single file
 - To obtain a file history, Git walks global history and determines whether each change modified that file. Makes examining history of changes to arbitrary set of file just as efficient.
 - Renames handled implicitly
 - Names do not identify files. Git detects renames while browsing history of snapshot, rather than when making snapshots.
 - Searches for file in previous revision very similar to new one.
 - Does not always work: best to rename and make changes separately to avoid identifying as deletion and addition.

Managing Project Content

Git server

- Git can be used as a server out of the box by using a built-in git daemon command. which starts simple TCP server running on the GIT protocol.
- Dedicated Git HTTP servers can be used to add access control, displaying the contents of a Git repository via the web interfaces, and managing multiple repositories.
- Existing Git repositories can be cloned and shared by others as a centralized repo.
- Git server can also be accessed via remote shell by having the Git software installed and allowing a user to log in. Git servers typically listen on TCP port 9418



Managing Project Content

Git and versioning

- Although Git can be used to manage any file type, it is particularly beneficial for source code created by members of a development teams.
 - Note that there are several Git extensions designed to work with large project content such as MSWord and PowerPoint files, CAD designs, and media files such as TIFF and MPEG.
 - Some of these extensions include *git-media*, *git-annex*, and *git-lfs* (large-file storage). These executable extensions plugin in to Git repositories to enable their functionality.
 - We will talk more about these extensions later.

Managing Project Content

Git and versioning

- As source code files change, GIT keeps track of changes to the files over time. These changes are usually the result of units of work on new product features or resolving defects.
- A group of changes made as a result of this work is known as a version, and these systems are often referred to as version control systems (VCS) in recognition of that important role.

Managing Project Content

Git and versioning

- Version control system have been around in some form since the late 1960s. The earliest VCS model has a single, central repository that everyone works from.
 - Everyone puts changes in and decides when and how to stay current.
 - All work by tracking changes to go from one version to another version of each file.
 - Examples include
 - Source Code Control System – SCCS (1972)
 - Revision Control System – RCS (1982)
 - Concurrent Version System – CVS (1986)
 - Subversion – SVN (2004)

Managing Project Content

Git and versioning

- Git uses distributed version control
 - Each user has his/her own repository, but in the real world you work with others and a central repository is standard and very good way to work.
 - We will see later how this is defined by process and supported by git. This is the model we will use
 - Git emphasizes tracking change sets and not individual versions of a file.
 - If you change three files in a directory, those three changes are marked as a single change set.

Managing Project Content

How Git works conceptually

- Git uses a three-part repository architecture

1. All work is done in the **working** area.
For example, you use an IDE to create
a Java file

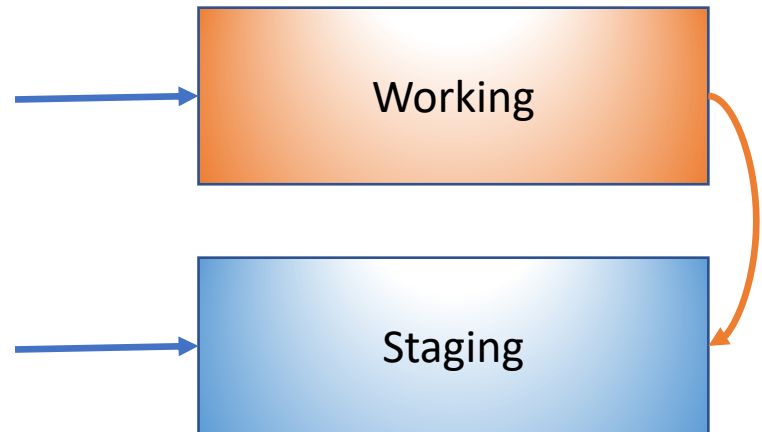


Managing Project Content

How Git works conceptually

- Git uses a three-part repository architecture

1. All work is done in the **working** area.
For example, you use an IDE to create a Java file



2. “Finished” work is added to a **staging** area. For example, a Java module passes the test.

Managing Project Content

How Git works conceptually

- Git uses a three-part repository architecture

1. All work is done in the **working** area.
For example, you use an IDE to create a Java file



2. “Finished” work is added to a **staging** area. For example, a Java module passes the test.



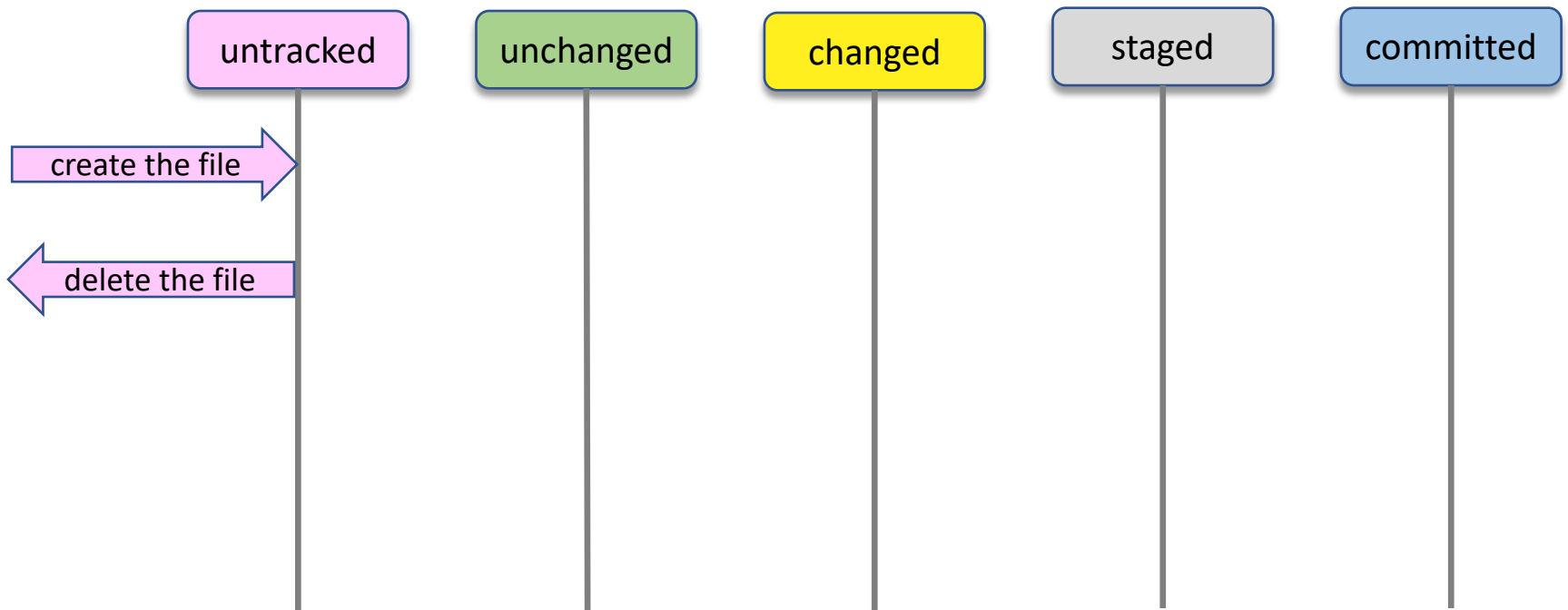
3. When a batch of work is ready to be committed as complete, the entire batch is committed to the **repository**.



Managing Project Content

How Git works with an individual file

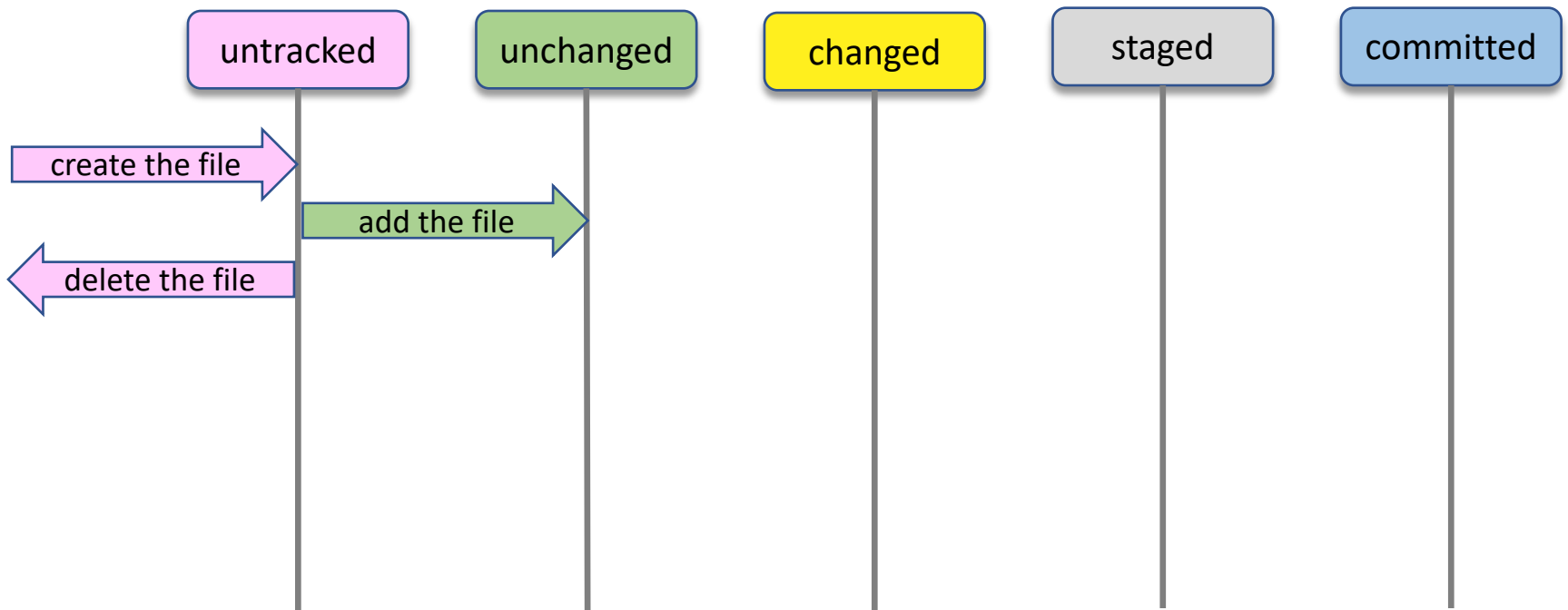
- Creating and deleting an untracked file.



Managing Project Content

How Git works with an individual file

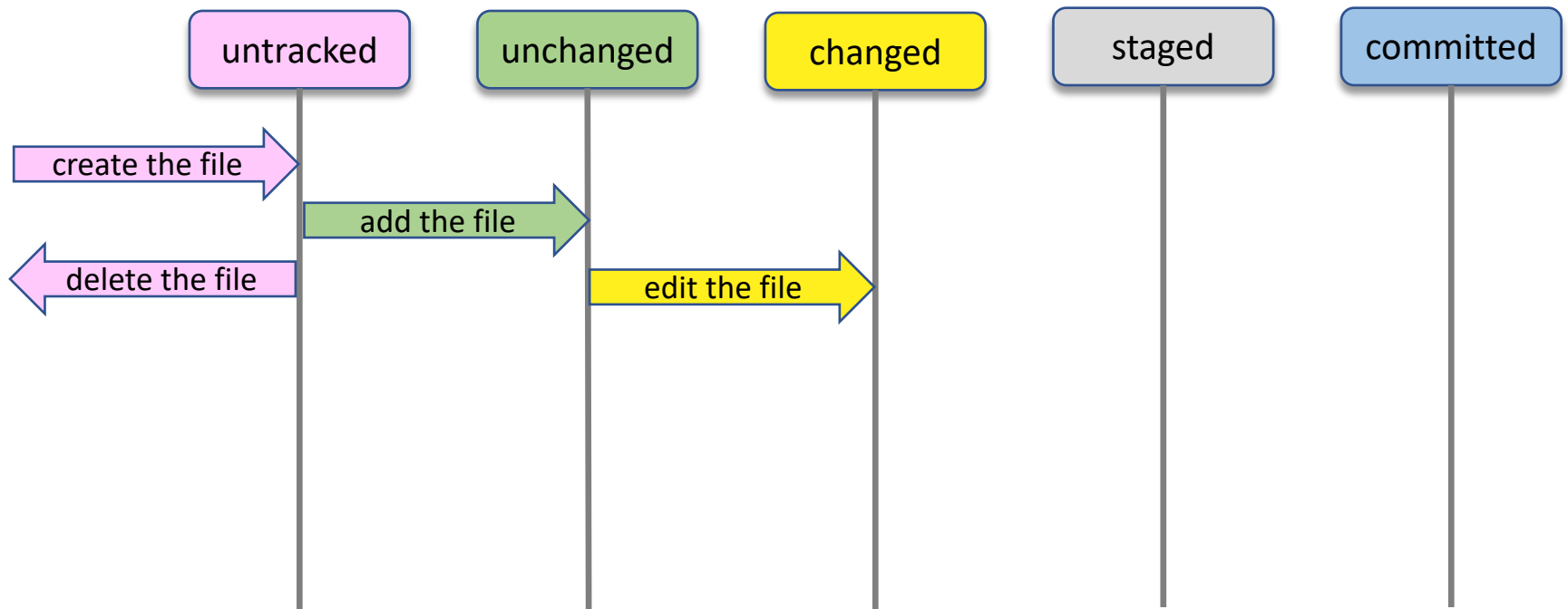
- Adding an untracked file.



Managing Project Content

How Git works with an individual file

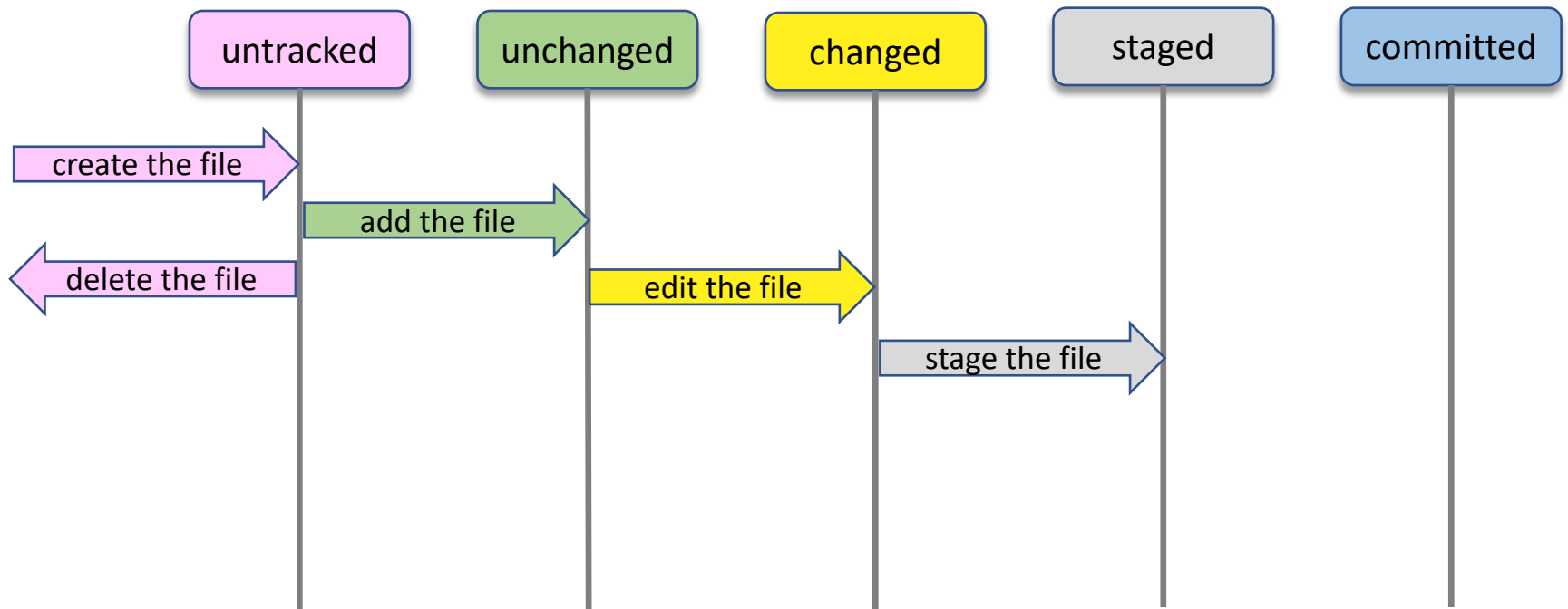
- Editing a tracked file.



Managing Project Content

How Git works with an individual file

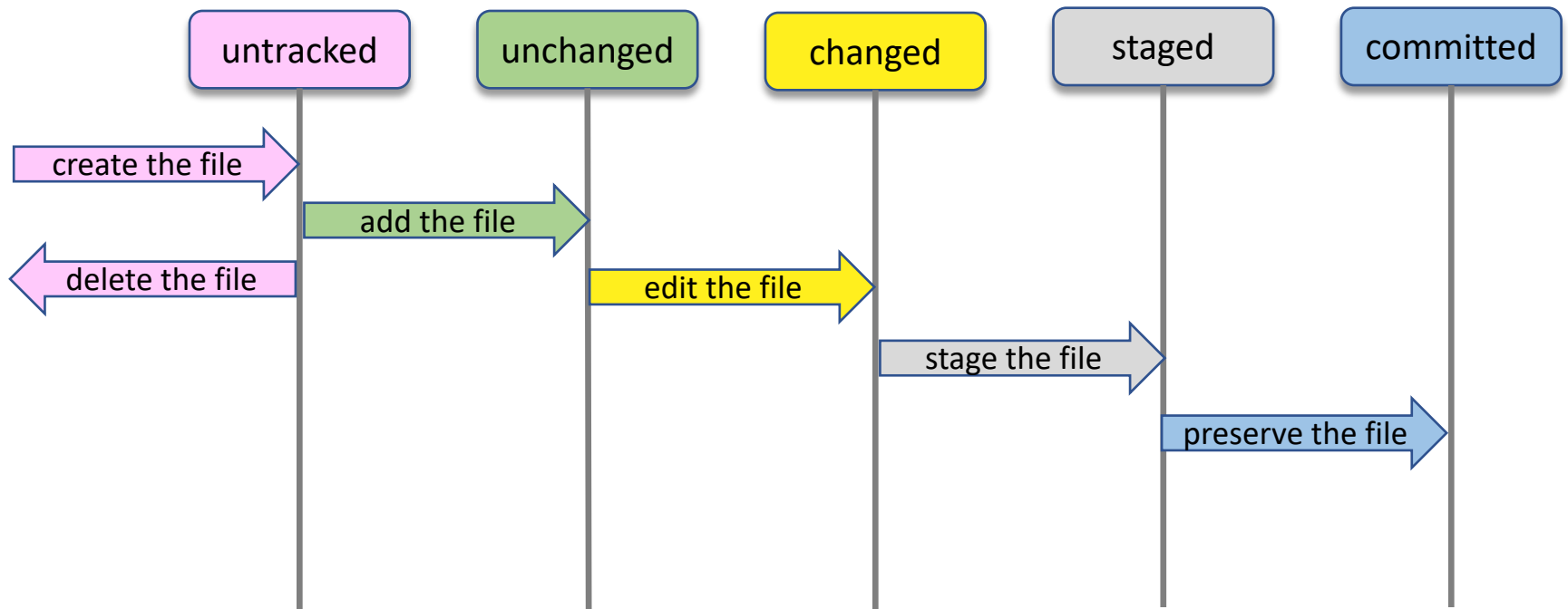
- Staging a changed tracked file.



Managing Project Content

How Git works with an individual file

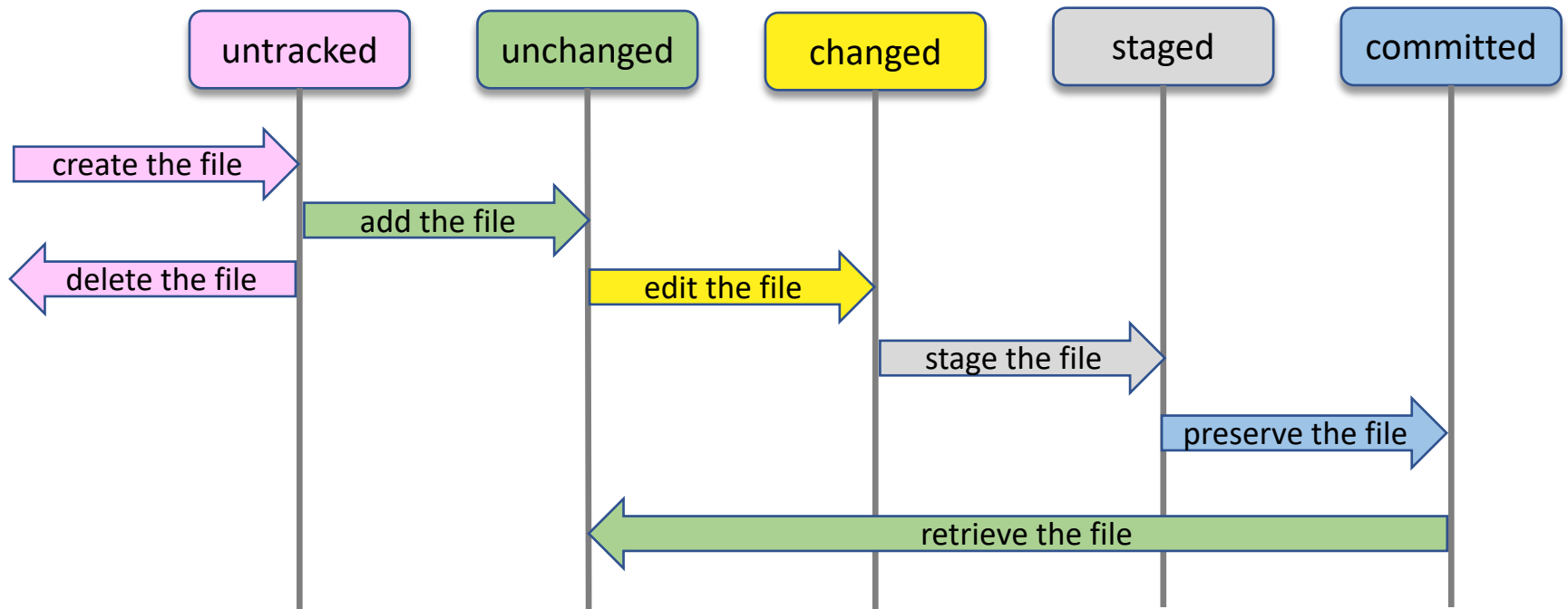
- Preserving a changed staged file.



Managing Project Content

How Git works with an individual file

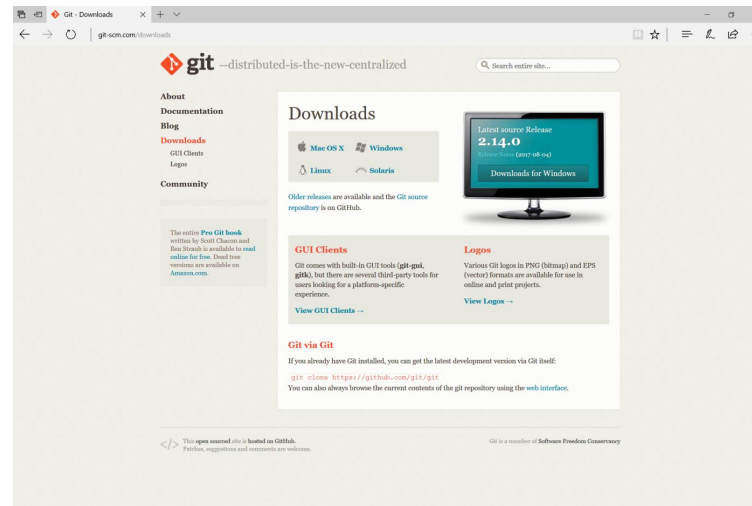
- Retrieving a committed file.



Managing Project Content

Getting Git

- The latest version of Git is available for Linux, MacOS, and MSWindows from Git, Inc: <https://git-scm.com/downloads>.
- A version of Git comes pre-installed on most Linux distros. On MacOS Git is part of the Xcode command line tools. An optional install package for Git is available on CygWin.



Managing Project Content

Interfacing with Git

- Interfaces available to Git include
 - Command line utilities from git-scm.com
 - JGit Java library from [Eclipse.org](https://eclipse.org/jgit/)
 - GitPython Python library (via 'pip' installer)
 - Eclipse IDE integration
 - JetBrains IDE integrations
 - XCode IDE integration
 - GitHub hosted Git server and tools
- In this lecture, we will learn about Git through the command line utilities, then see to use Git through IDE integrations.

Managing Project Content

Introducing the Git command

- The Git command line interface is dominated by a single command:

git *<verb>* *<options>* *<pathspec>*

- *<verb>*: the Git action to be performed
- *<options>*: the options to customize the behavior of the verb
- *<pathspec>*: a verb-specific path specifier

Managing Project Content

Configuring Git

- The Git command reads and writes configuration information:
git config <level> <config>
 - Configuration information is stored at three levels
 - **--system:** system-wide configuration information
Configuration file:
 - “/etc/gitconfig” on Linux and CygWin
 - “/private/etc/gitconfig” on MacOS.
 - **--global:** global-to-user configuration information
Configuration file: (in user home directory)
 - “~/.gitconfig” on Linux , CygWin, and MacOS
 - **--local:** project/repository-specific configuration information
Configuration file (in project directory):
 - “./.git/config” on Linux, CygWin, and MacOS

Managing Project Content

Configuring Git

- Setting basic information at global (user level):
 - user name:
`git config --global user.name "Philip Gust"`
 - user email:
`git config --global user.email "pgust@ccs.neu.edu"`
- Listing global configuration settings:
`git config --global --list`
 - Output:
user.name=Philip Gust
user.email=pgust@ccs.neu.edu
- List git config options:
`git config`

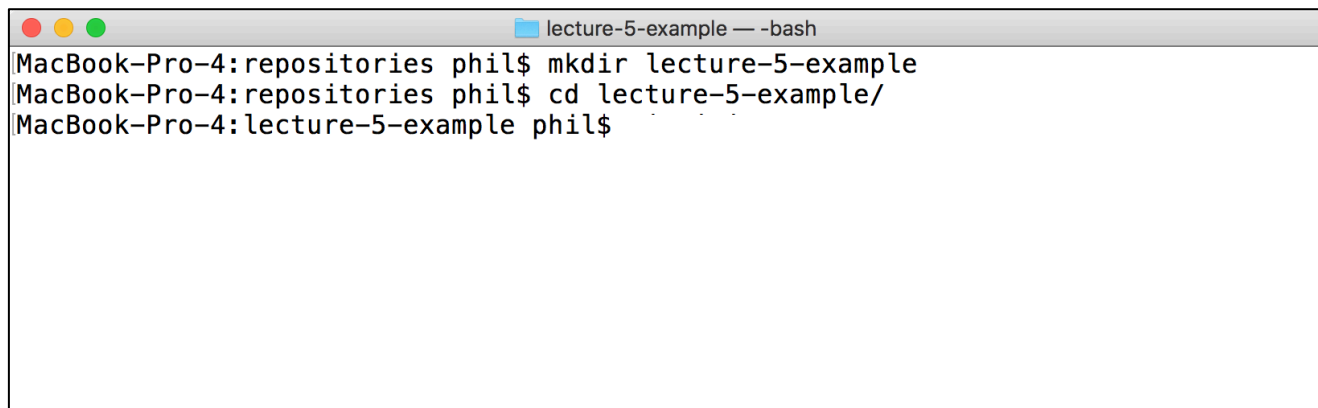
Managing Project Content

Create a Git repository

- The location for project files is called a *repository*. The repository is in a local directory. For this lecture, this will be “lecture-5-example”:

mkdir lecture-5-example

cd lecture-5-example



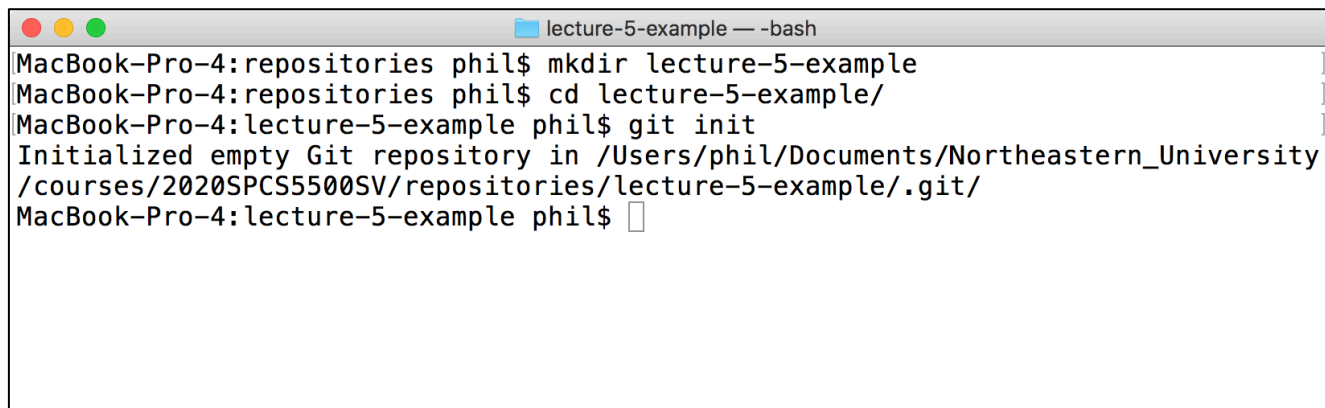
```
lecture-5-example — -bash
MacBook-Pro-4:repositories phil$ mkdir lecture-5-example
MacBook-Pro-4:repositories phil$ cd lecture-5-example/
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Create a Git repository

- Now we will initialize the directory to create a Git repository and to track all changes made inside the directory.

git init

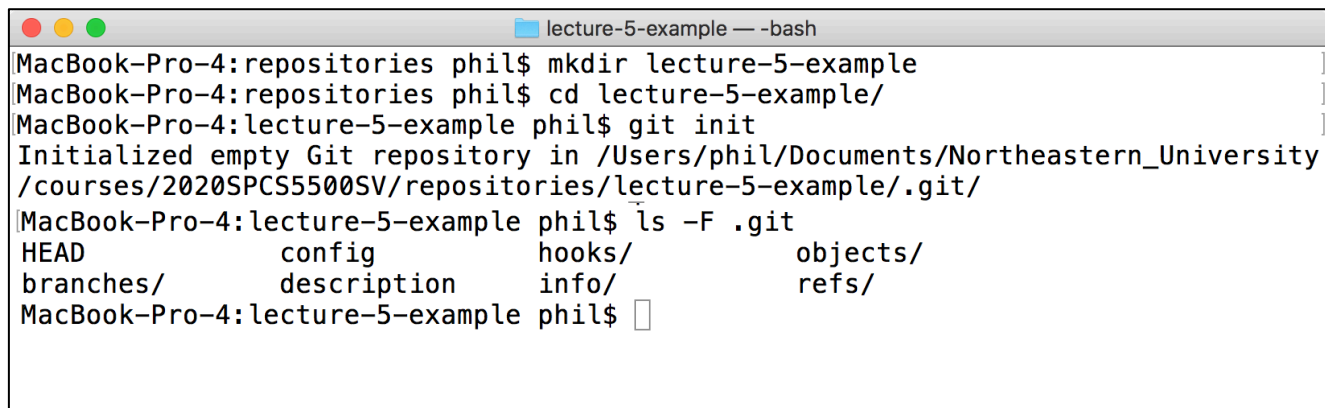
A terminal window titled "lecture-5-example — -bash" with standard macOS window controls (red, yellow, green buttons). The terminal shows the following commands and output:

```
MacBook-Pro-4:repositories phil$ mkdir lecture-5-example
MacBook-Pro-4:repositories phil$ cd lecture-5-example/
MacBook-Pro-4:lecture-5-example phil$ git init
Initialized empty Git repository in /Users/phil/Documents/Northeastern_University/courses/2020SPCS5500SV/repositories/lecture-5-example/.git/
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Create a Git repository

- Everything that Git keeps about the project is in the “.git” directory within the project directory. If you were to “rm -rf .git” then Git would no longer track the project.
- Here is a listing of the “.git” directory showing the repository files and directories

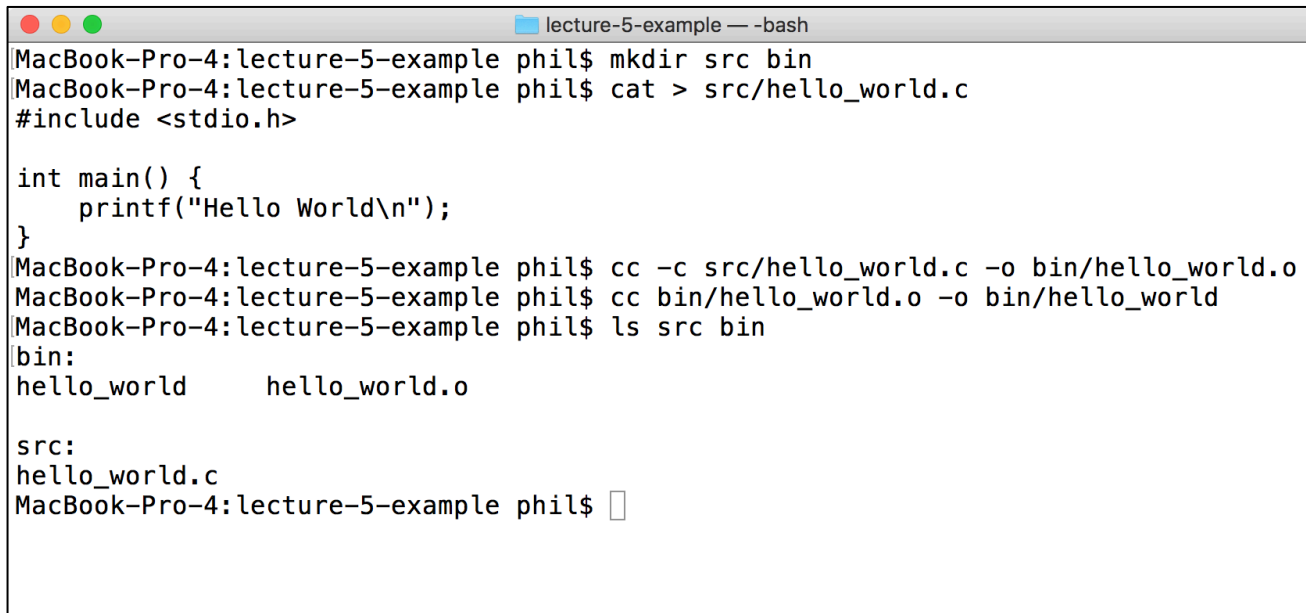
A terminal window titled "lecture-5-example — -bash" showing the steps to create a Git repository. The user runs 'mkdir lecture-5-example', 'cd lecture-5-example/', and 'git init'. The output of 'git init' is "Initialized empty Git repository in /Users/phil/Documents/Northeastern_University/courses/2020SPCS5500SV/repositories/lecture-5-example/.git/". Finally, the user runs 'ls -F .git', which lists the contents of the .git directory: HEAD, config, hooks/, objects/, branches/, description, info/, and refs/.

```
MacBook-Pro-4:repositories phil$ mkdir lecture-5-example
MacBook-Pro-4:repositories phil$ cd lecture-5-example/
MacBook-Pro-4:lecture-5-example phil$ git init
Initialized empty Git repository in /Users/phil/Documents/Northeastern_University/courses/2020SPCS5500SV/repositories/lecture-5-example/.git/
MacBook-Pro-4:lecture-5-example phil$ ls -F .git
HEAD          config        hooks/        objects/
branches/     description   info/         refs/
MacBook-Pro-4:lecture-5-example phil$
```


Managing Project Content

Create Git repository content

- Now we will create some content to preserve by making “src” and “bin” directories, creating a C source file “hello_world.c” in the “src” directory, and compiling and building it in “bin”



```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ mkdir src bin
MacBook-Pro-4:lecture-5-example phil$ cat > src/hello_world.c
#include <stdio.h>

int main() {
    printf("Hello World\n");
}
MacBook-Pro-4:lecture-5-example phil$ cc -c src/hello_world.c -o bin/hello_world.o
MacBook-Pro-4:lecture-5-example phil$ cc bin/hello_world.o -o bin/hello_world
MacBook-Pro-4:lecture-5-example phil$ ls src bin
bin:
hello_world    hello_world.o

src:
hello_world.c
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Create Git repository content

- We will also create a “docs” directory with a “index.html” file in it, and a “README.md” file in the repository root.

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ mkdir docs
MacBook-Pro-4:lecture-5-example phil$ cat >docs/index.html
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World</h1>
</body>
</html>
MacBook-Pro-4:lecture-5-example phil$ cat >README.md
# lecture-5-example
example repository for lecture 5
MacBook-Pro-4:lecture-5-example phil$ ls -F *
README.md

bin:
hello_world*    hello_world.o

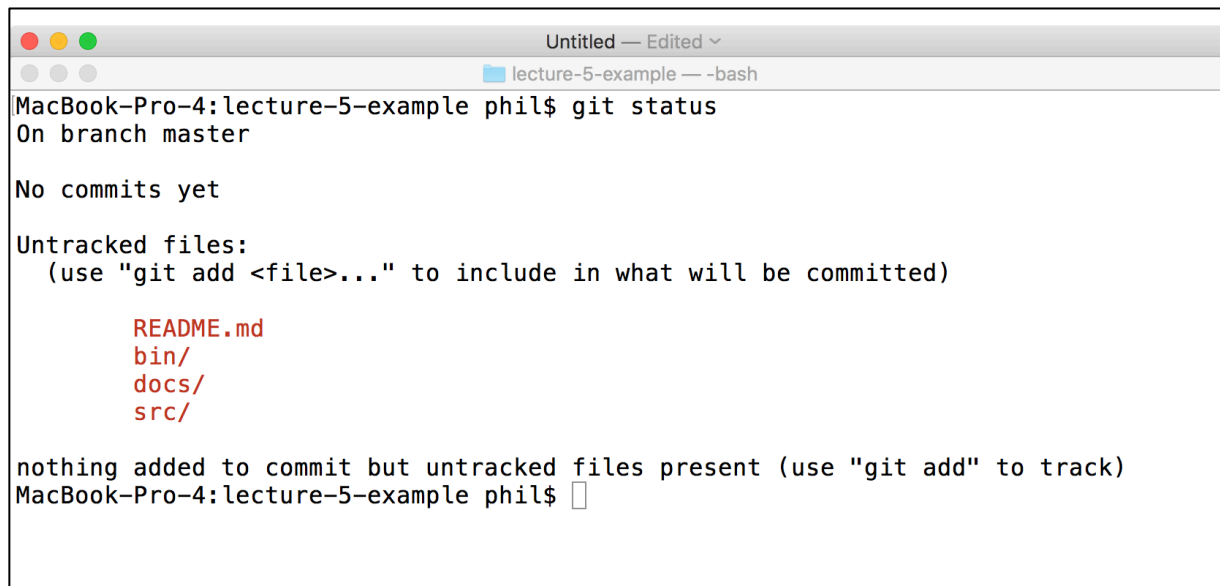
docs:
index.html

src:
hello_world.c
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Get Git repository status

- The command “**git status**” reports differences between tiers:
 - We are on a branch called “master”
 - There is nothing *staged*
 - There are files in directories and the root that are not *tracked*.

A screenshot of a macOS terminal window. The title bar shows 'Untitled — Edited' and a dropdown menu. Below the title bar, the window title is 'lecture-5-example — -bash'. The terminal content shows the command 'git status' being executed in a directory 'MacBook-Pro-4:lecture-5-example'. The output indicates the current branch is 'master', there are no commits yet, and there are untracked files: 'README.md', 'bin/', 'docs/', and 'src/'. A message at the bottom states 'nothing added to commit but untracked files present (use "git add" to track)'.

```
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md
        bin/
        docs/
        src/

nothing added to commit but untracked files present (use "git add" to track)
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Adding files for GitHub to track

- We will take the advice in the status instructions and add all non-hidden files in the repository to be tracked by GitHub. These are the files that are ready to be committed locally.
- Here is the status after calling “**git add ***” to track all files.

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git add *
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
        new file:   bin/hello_world
        new file:   bin/hello_world.o
        new file:   docs/index.html
        new file:   src/hello_world.c

MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Remove files from being tracked

- We do not intend to commit build artifacts like files in “bin” to the repository. We remove them so they are not tracked. Here is the status after “**git rm --cached bin/***”

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git rm --cached bin/*
rm 'bin/hello_world'
rm 'bin/hello_world.o'
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
        new file:   docs/index.html
        new file:   src/hello_world.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        bin/

MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Not everything should be tracked

- You may create all kinds of files in a project, but not every file is important to preserve
 - Files that change constantly
 - e.g. log files, databases
 - Temporary files
 - Files that will be created by builds
 - e.g. object files, compressed files
 - Instance data
 - E.g. user-generated content (should be stored in a database)
- Typically you want to track all source files (code, documents, templates,...) and configuration files.

Managing Project Content

Telling Git to ignore items

- Use “.gitignore” to tell Git to ignore specific files or types of files It is placed in the project’s root directory.
- The community has built up recommendations for what should go into a “.gitignore” based on environment
 - See <https://github.com/github/gitignore>

Managing Project Content

Telling Git to ignore items

- Git ignore rules are usually defined in a “.gitignore” file at the root of your repository. This is the convention and simplest approach
- You may have different “.gitignore” files in different directories in your repository.
 - Each pattern in a particular “.gitignore” file is tested relative to the directory tree containing that “.gitignore”
 - It is best to use “.gitignore” in sub- directories to augment the rules of “.gitignore” files in the parent directories.
 - Typically you should only include patterns in “.gitignore” that will benefit other users of the repository.

Managing Project Content

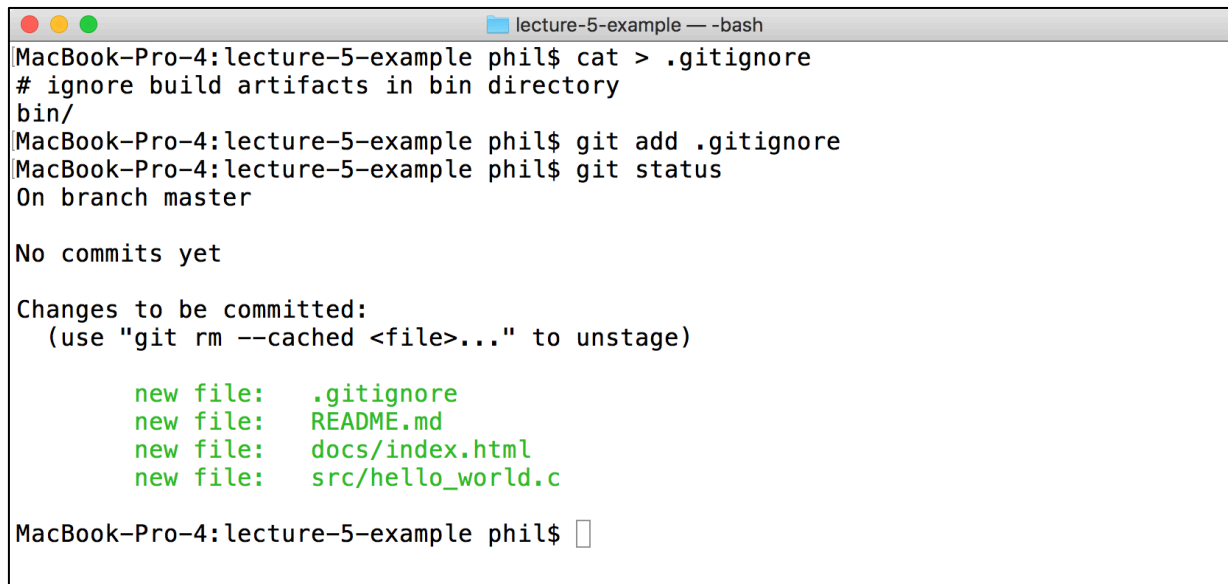
Telling Git to ignore items

- .gitignore lines may identify specific files or may be simple regular expressions
 - ? [char_set] [range] !
- This would ignore all *foo* files but not *myReallyImportant.foo*
 - *.foo
 - ! myReallyImportant.foo
- One may use / to mean files in subdirectories.
- Git does not track empty directories.

Managing Project Content

Telling Git to ignore items

- We want to ignore build artifacts in “bin”, so we create a “.gitignore” file with that path in the repository root, and also add “.gitignore” to be tracked.
- Note that ‘bin’ directory is now not reported as untracked.

A terminal window titled "lecture-5-example — -bash" on a Mac. The user is in the directory "MacBook-Pro-4:lecture-5-example" and runs several git commands. First, they create a .gitignore file with the content "# ignore build artifacts in bin directory" and "bin/". Then they run "git add .gitignore". Finally, they run "git status", which shows they are on the master branch with no commits yet. The changes to be committed are listed as four new files: .gitignore, README.md, docs/index.html, and src/hello_world.c.

```
MacBook-Pro-4:lecture-5-example phil$ cat > .gitignore
# ignore build artifacts in bin directory
bin/
MacBook-Pro-4:lecture-5-example phil$ git add .gitignore
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   README.md
        new file:   docs/index.html
        new file:   src/hello_world.c

MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Moving from staging to the repository

- The command “**git** commit -m ‘*message*’” moves changes from staging to the repository.
- The message describes what is in this change set, so when anyone looks at it in the future, they do not have to look at the actual changes to know what the change set contains.
- Good, descriptive commit messages are vital.
- You are writing these messages for other people to read and use later on, when you may not be available. In the future, you may not remember what the commit entailed either.

Managing Project Content

Moving from staging to the repository

- Here is the result of committing our files and directories to our local repository.
- After the initial commit of the files, the “**git status**” command reports nothing to commit: working tree clean.

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git commit -m "initial commit of example files"
[master (root-commit) 7104a55] initial commit of example files
 4 files changed, 17 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 README.md
 create mode 100644 docs/index.html
 create mode 100644 src/hello_world.c
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
nothing to commit, working tree clean
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Best practices for commit messages

- A good commit message should answer three questions:
 1. Why is it necessary? It may fix a bug, it may add a feature, it may improve performance, reliability, stability, or just be a change for the sake of correctness.
 2. How does it address the issue? For short obvious patches this part can be omitted, but it should be a high level description of what the approach was.
 3. What effects does the patch have? (In addition to the obvious ones, this may include benchmarks, side effects, etc.)

Managing Project Content

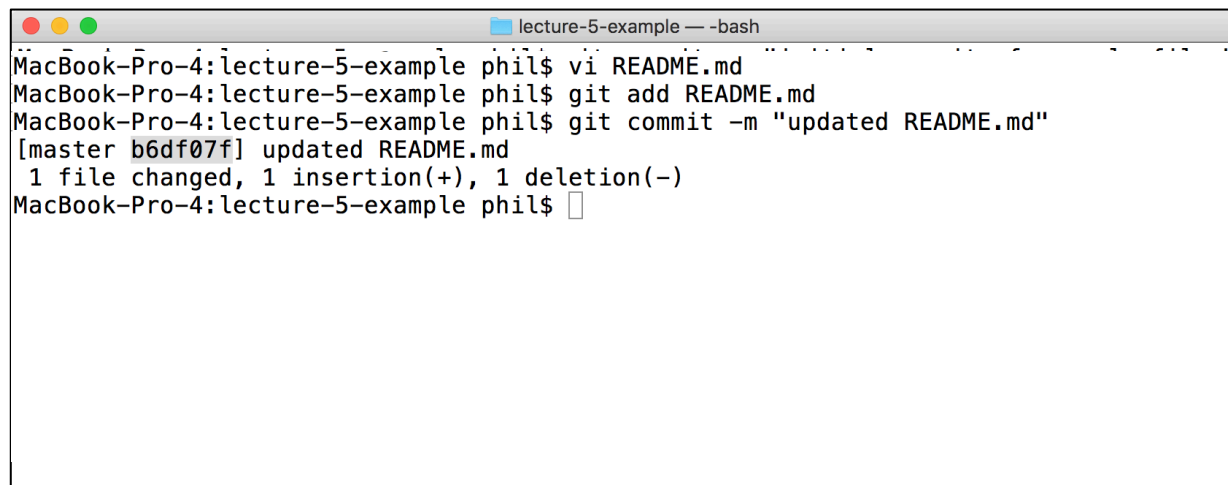
Best practices for commit messages

- If you omit the ‘-m’ parameter, Git will launch a text editor of your choice for you to enter a message and descriptive text. When the editor closes, the commit completes.
 - Make the first 25-50 characters a short summary. Separate the summary from the rest of the message by a blank line.
 - Bullets are OK using hyphen or asterisks.
 - Use imperative statements in subject line (e.g. Fix broken link).
 - Begin subject line with capitalized verb (e.g. “Added...”).
 - Wrap body at 72 characters or less.
 - Mention associated bug or feature request number.
 - Explain how it was before commit, and what has changed.

Managing Project Content

How Git identifies commits

- Git uses a SHA-1 hash against all changes in the change set to create a unique 40 digit hex number identifier.
- Our initial commit has SHA-1: 7104a55... . If we edit "README.md" and commit it again, the new commit has SHA-1: b6df07f... . The first 7 hex digits serve as a short identifier for the change.

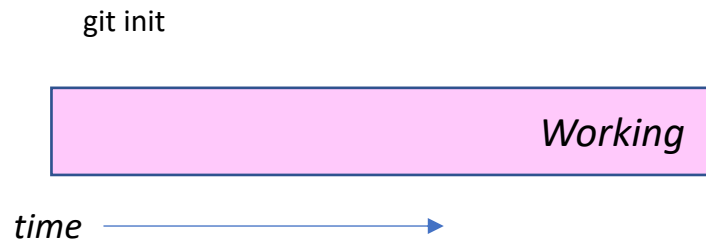


```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ vi README.md
MacBook-Pro-4:lecture-5-example phil$ git add README.md
MacBook-Pro-4:lecture-5-example phil$ git commit -m "updated README.md"
[master b6df07f] updated README.md
 1 file changed, 1 insertion(+), 1 deletion(-)
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

HEAD

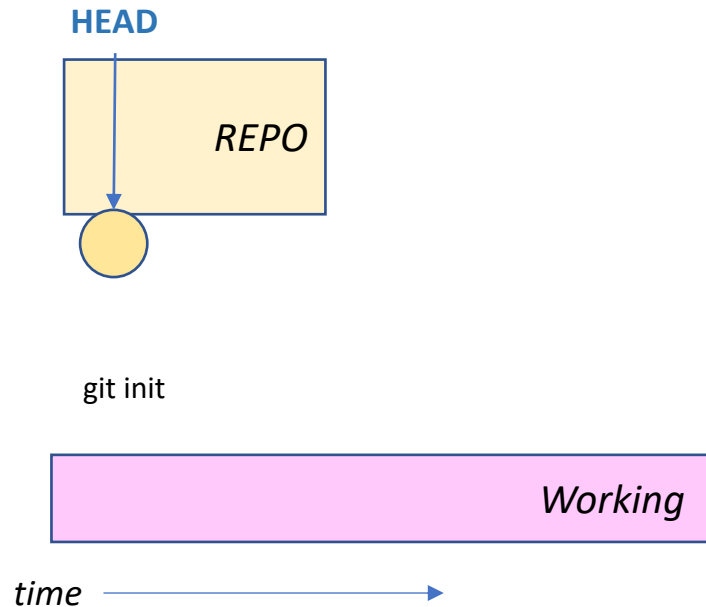
- *HEAD* is a pointer to the most recent commit on a branch. You can reference HEAD directly in Git commands.



Managing Project Content

HEAD

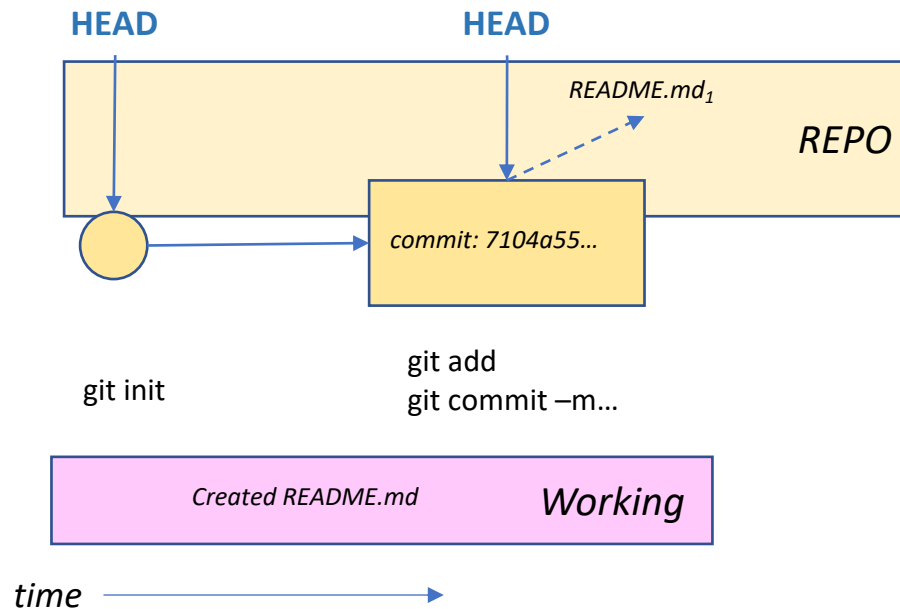
- *HEAD* is a pointer to the most recent commit on a branch. You can reference HEAD directly in Git commands.



Managing Project Content

HEAD

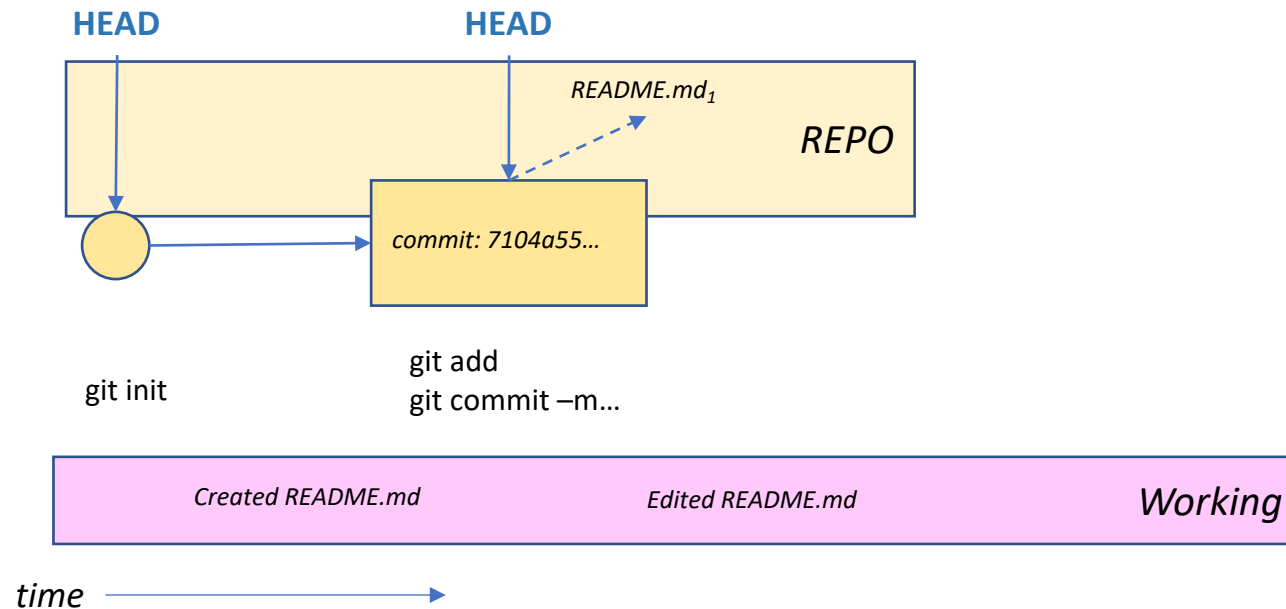
- *HEAD* is a pointer to the most recent commit on a branch. You can reference HEAD directly in Git commands.



Managing Project Content

HEAD

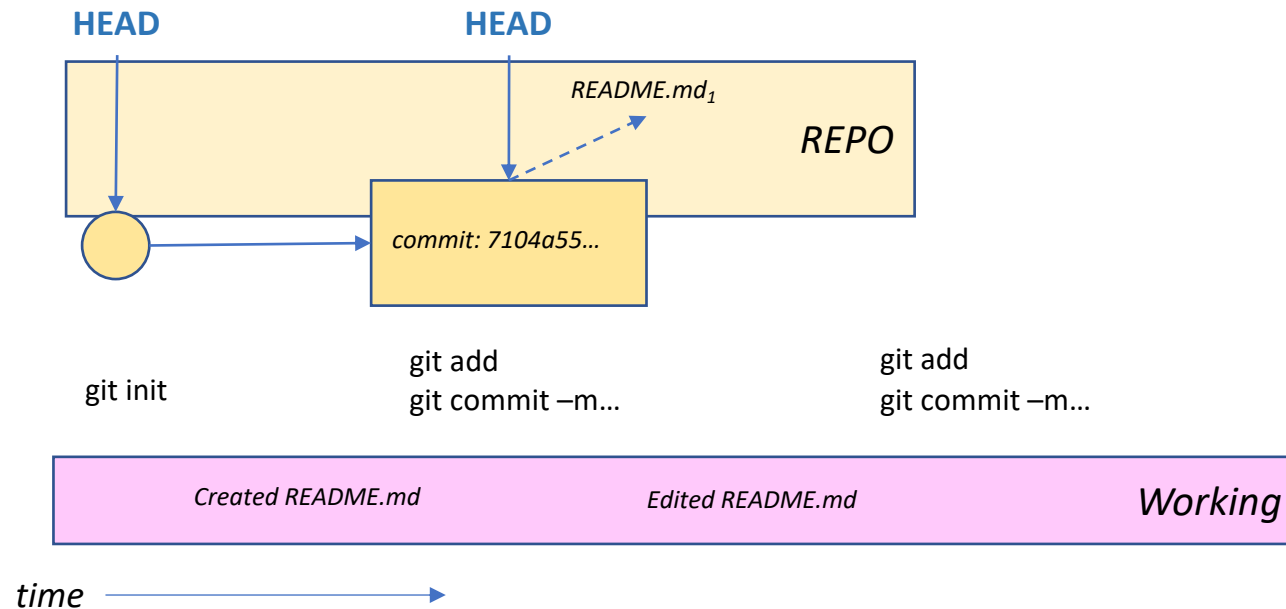
- *HEAD* is a pointer to the most recent commit on a branch. You can reference HEAD directly in Git commands.



Managing Project Content

HEAD

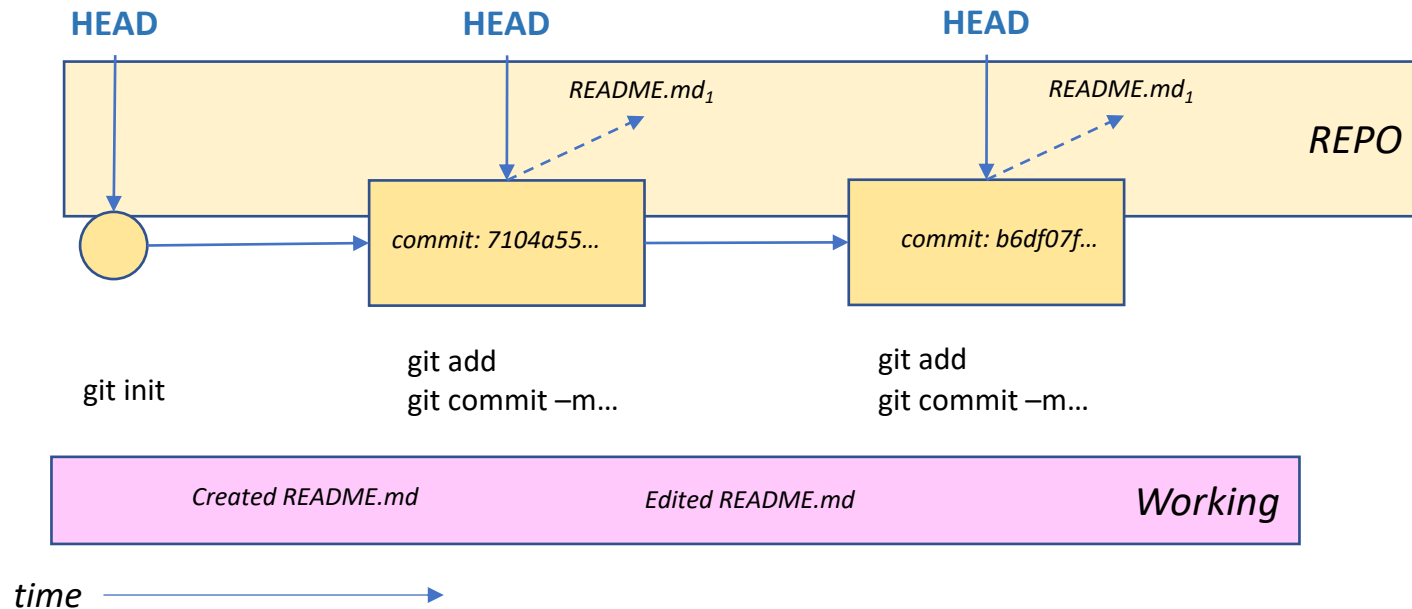
- *HEAD* is a pointer to the most recent commit on a branch. You can reference HEAD directly in Git commands.



Managing Project Content

HEAD

- *HEAD* is a pointer to the most recent commit on a branch. You can reference HEAD directly in Git commands.



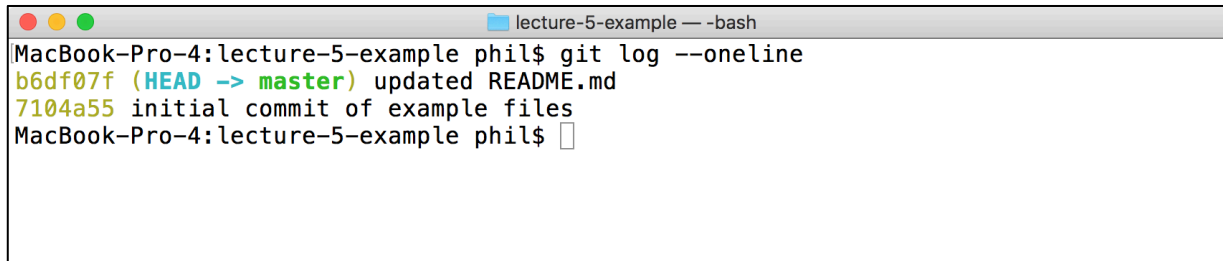
Managing Project Content

Listing commits

- The command “**git log**” lists commits to the repository

Helpful ways to use the log

- `git log --oneline`
- `git log --since “2020-2-04”`
- `git log --until “2020-2-05”`
- `git log --since “one month ago” --until “1 day ago”`
- `git log --since=1.month --until=1.day`
- `git log --author=“maw”`
- `git log --grep=“someKeyWord”`

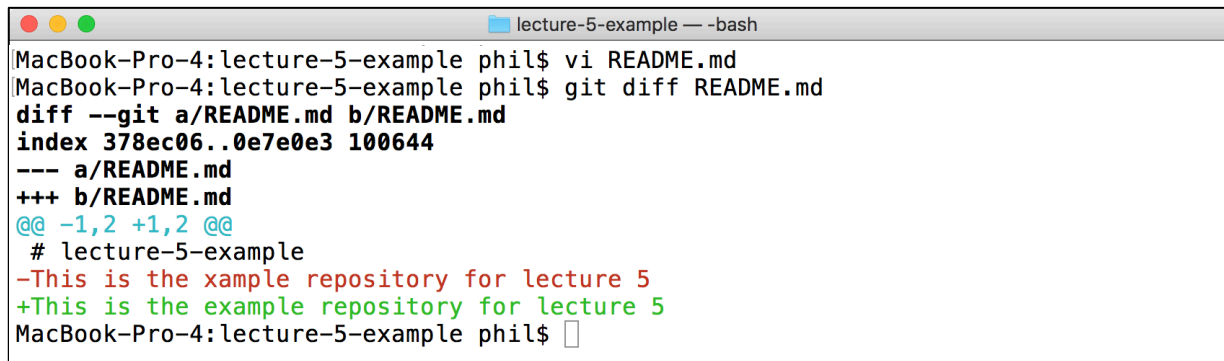
A screenshot of a terminal window titled "lecture-5-example — -bash". The window shows the command "git log --oneline" being executed in a directory "MacBook-Pro-4:lecture-5-example" by a user named "phil". The output displays two commits: "b6df07f (HEAD -> master) updated README.md" and "7104a55 initial commit of example files".

```
MacBook-Pro-4:lecture-5-example phil$ git log --oneline
b6df07f (HEAD -> master) updated README.md
7104a55 initial commit of example files
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Working file differences

- The command “**git diff** <working file>” tells us what is different between file in WORKING and HEAD in the repository.
 - Shows only the lines where something is different. This may include white space.
 - Additions are in green and deletions are in red.
- The command “**git diff -w** <working file>” ignores whitespace
- Edited “README.md” and showed diff with repository

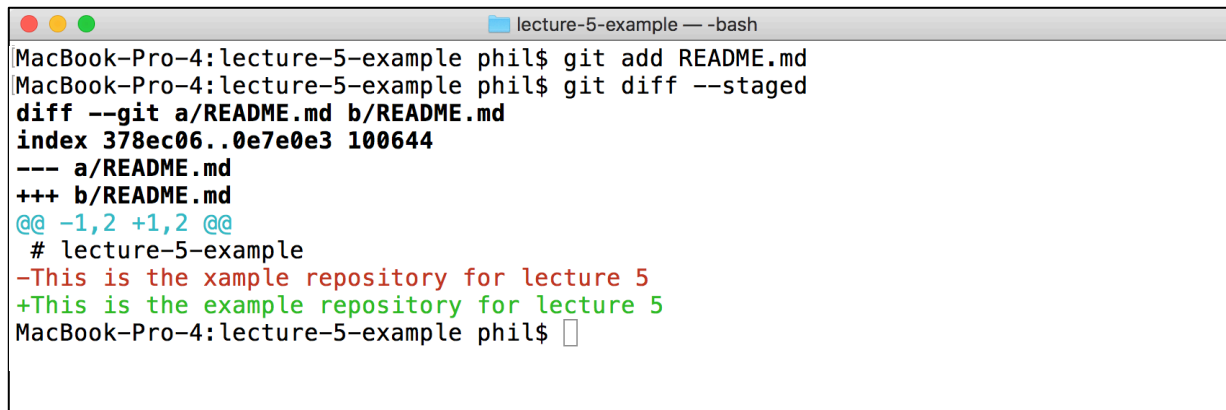
A terminal window titled "lecture-5-example -- -bash" showing the output of the command "git diff README.md". The output displays the differences between the working file and the repository HEAD. It shows a deletion of a line (in red) and an addition of a line (in green).

```
MacBook-Pro-4:lecture-5-example phil$ vi README.md
MacBook-Pro-4:lecture-5-example phil$ git diff README.md
diff --git a/README.md b/README.md
index 378ec06..0e7e0e3 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 # lecture-5-example
-This is the xample repository for lecture 5
+This is the example repository for lecture 5
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Staged file differences

- The command “**git diff --staged**” tells us what differences between what has been STAGED and what is in the HEAD in the repository.
 - Shows us only the lines where something is different. This may include white space.
 - Additions are in green and deletions are in red.
- Staged “README.md” and showed diff with repository

A terminal window titled "lecture-5-example — -bash" on a MacBook-Pro-4. The user has run "git add README.md" and then "git diff --staged". The output shows the diff for README.md, with deletions in red and additions in green. The diff shows two lines being added to the repository.

```
MacBook-Pro-4:lecture-5-example phil$ git add README.md
MacBook-Pro-4:lecture-5-example phil$ git diff --staged
diff --git a/README.md b/README.md
index 378ec06..0e7e0e3 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 # lecture-5-example
-This is the xample repository for lecture 5
+This is the example repository for lecture 5
MacBook-Pro-4:lecture-5-example phil$
```


Managing Project Content

Deleting files

- The command “**git rm** *<rm_file>*” stages the change from removing a tracked file to remove it from the repository.
 - Delete file using ‘rm’ or sending to trash
 - Stage change using “**git rm**” as opposed to “**git add**”
 - deletes file if still exists
 - Git commit the change
 - Git only cares about deleting tracked files; doesn’t care about deleting untracked files.

Managing Project Content

Deleting files

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ cat >temp1.txt
Temporary file.
MacBook-Pro-4:lecture-5-example phil$ git add temp1.txt
MacBook-Pro-4:lecture-5-example phil$ git commit -m "temporary file to delete"
[master aee2e6c] temporary file to delete
1 file changed, 1 insertion(+)
create mode 100644 temp1.txt
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
nothing to commit, working tree clean
MacBook-Pro-4:lecture-5-example phil$ rm temp1.txt
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    temp1.txt

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-4:lecture-5-example phil$ git rm temp1.txt
rm 'temp1.txt'
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    temp1.txt

MacBook-Pro-4:lecture-5-example phil$ git commit -m "commit delete temp file"
[master 98d1d66] commit delete temp file
1 file changed, 1 deletion(-)
delete mode 100644 temp1.txt
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Moving/rename (two steps)

- You can mv files on the command line. Git will notice the change, but treat it as two actions. The Git workflow is:
 1. Stage the changes
 - a) git add the “new” file.
 - b) git rm the “old” file”
 2. Git will then notice the renaming
 3. Commit the change
 - a) git commit -m “two-step rename”

Managing Project Content

Moving/rename (two steps)

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ cat >temp1.txt
Temporary file
MacBook-Pro-4:lecture-5-example phil$ git add temp1.txt
MacBook-Pro-4:lecture-5-example phil$ git commit -m "temporary file to rename"
[master 234c75b] temporary file to rename
 1 file changed, 1 insertion(+)
 create mode 100644 temp1.txt
MacBook-Pro-4:lecture-5-example phil$ mv temp1.txt temp2.txt
MacBook-Pro-4:lecture-5-example phil$ git add temp2.txt
MacBook-Pro-4:lecture-5-example phil$ git rm temp1.txt
rm 'temp1.txt'
MacBook-Pro-4:lecture-5-example phil$ git commit -m "two-step rename"
[master b9d61f2] two-step rename
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename temp1.txt => temp2.txt (100%)
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Moving/renaming (one step)

- The command “**git mv** *<cur-name>* *<new-name>*” moves/renames a file in the repository.

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git mv temp2.txt temp1.txt
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    temp2.txt -> temp1.txt

MacBook-Pro-4:lecture-5-example phil$ git commit -m "move file one step"
[master 44448b4] move file one step
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename temp2.txt => temp1.txt (100%)
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Restore from repository (undo)

- To restore what is in WORKING from the repository, use the command “**git checkout <file-name>**”

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ vi README.md
MacBook-Pro-4:lecture-5-example phil$ cat README.md
# lecture-5-example
This is the example repository for lecture 5
This line not in repository
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
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:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
MacBook-Pro-4:lecture-5-example phil$ git checkout README.md
MacBook-Pro-4:lecture-5-example phil$ !cat
cat README.md
# lecture-5-example
This is the example repository for lecture 5
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
nothing to commit, working tree clean
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Unstaging files

- If you accidentally add a file, or decide changes to a file should not be committed, use the command “**git reset HEAD <file-name>**”

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ echo "this is a mistake" > file1.txt
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
MacBook-Pro-4:lecture-5-example phil$ git add file1.txt
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   file1.txt

MacBook-Pro-4:lecture-5-example phil$ git reset HEAD file1.txt
MacBook-Pro-4:lecture-5-example phil$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

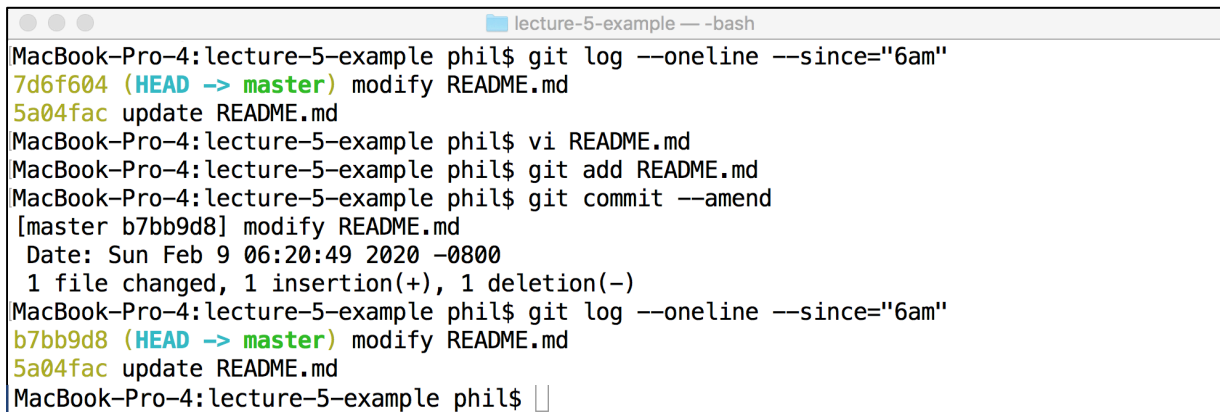
Changing HEAD

- Another form of the *reset* command allows you to reset HEAD to a previous commit. Any new commits will be from that point. Any later commits will be lost.
 - *Be careful using this form of the reset command!*
- There are three reset types:
 - `git reset --soft <commit>`
 - Moves the repo HEAD but leaves WORKING and STAGING alone (in their current states)
 - `git reset --mixed <commit>`
 - Moves the repo HEAD and sets STAGING to match the REPO. WORKING remains in its current state (untouched).
 - `git reset --hard <commit>`
 - Moves the repo HEAD and aligns both WORKING and STAGING to match the repo.

Managing Project Content

Touching commits

- Sometimes you want to make an adjustment to the last commit and you do not want to trigger a new commit. Using the command “**git commit --amend**” adds changes to the recent commit.
 - In this example, we just fixed a minor typo in the previous checkin of “README.md”.
 - The current HEAD is not changed, but the file has been updated in it.

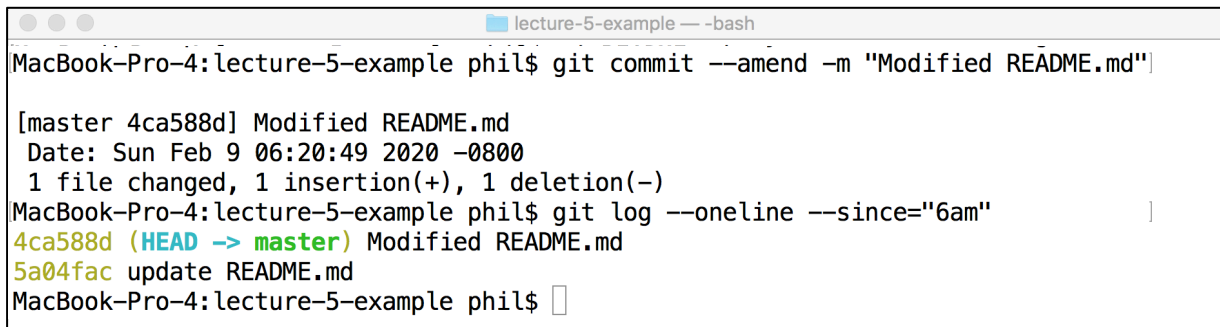


```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git log --oneline --since="6am"
7d6f604 (HEAD -> master) modify README.md
5a04fac update README.md
MacBook-Pro-4:lecture-5-example phil$ vi README.md
MacBook-Pro-4:lecture-5-example phil$ git add README.md
MacBook-Pro-4:lecture-5-example phil$ git commit --amend
[master b7bb9d8] modify README.md
Date: Sun Feb 9 06:20:49 2020 -0800
1 file changed, 1 insertion(+), 1 deletion(-)
MacBook-Pro-4:lecture-5-example phil$ git log --oneline --since="6am"
b7bb9d8 (HEAD -> master) modify README.md
5a04fac update README.md
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Touching commits

- Sometimes you just want to adjust the most recent commit message without triggering a new commit. Using the command “**git commit --amend**” can change the recent commit message.
 - In this example, we want to improve the commit message of the previous checkin, which we have just amended.
 - The current HEAD is not changed, but the commit message has been updated in it.



```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git commit --amend -m "Modified README.md"

[master 4ca588d] Modified README.md
Date: Sun Feb 9 06:20:49 2020 -0800
1 file changed, 1 insertion(+), 1 deletion(-)
MacBook-Pro-4:lecture-5-example phil$ git log --oneline --since="6am"
4ca588d (HEAD -> master) Modified README.md
5a04fac update README.md
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Reverting a previous commit

- The command “**git revert <commit>**” “resets” the working, staging, and repo. to the version indicated by the commit hash code or HEAD if reverting the most recent commit.
 - Git keeps around the changes so people can see the project’s path and the reasons for the change
 - Do not hide mistakes: accept them and move on.

```
lecture-5-example — -bash
MacBook-Pro-4:lecture-5-example phil$ git log --oneline --since="6am"
4ca588d (HEAD -> master) Modified README.md
5a04fac update README.md
MacBook-Pro-4:lecture-5-example phil$ git revert HEAD
[master a47052c] Revert "Modified README.md"
 1 file changed, 1 insertion(+), 1 deletion(-)
MacBook-Pro-4:lecture-5-example phil$ git log --oneline --since="6am"
a47052c (HEAD -> master) Revert "Modified README.md"
4ca588d Modified README.md
5a04fac update README.md
MacBook-Pro-4:lecture-5-example phil$
```

Managing Project Content

Next time in lecture 6

- In lecture 6, we will conclude our discussion of team-oriented aspects of source code and other content management.
 - We will first look at branching, and learn about the role that branches play in collaborative software development.
 - Next, we will learn how to clone remote repositories and to make repositories available for other team members to clone.
 - Then, we will look GitHub features that support distributed software development, and tools that support collaborative development and continuous integration.
 - Finally, we will learn about extension to Git and GitHub that support storing large binary content in a Git repository.