

# Object-Oriented Programming

CSCI-UA 0470-001

Class 5

Instructor: Randy Shepherd

# Object-Oriented Analysis and Design

# Analysis, Design, Implementation

- Non-trivial software development requires forethought. Jumping into an editor first thing will get you into trouble.
- The book calls Analysis, Design & Implementation ‘phases’ and treats them as sequential. This is naive and not the way things work in practice.

# Analysis

- Think the problem through, get an understanding in broad strokes of what you will be building.
  - What should the software **do**?
  - Ask ‘Why?’ as many times as is necessary!
- Decompose the problem into *subproblems*.
- Identify the ‘nouns’ and the ‘verbs’.

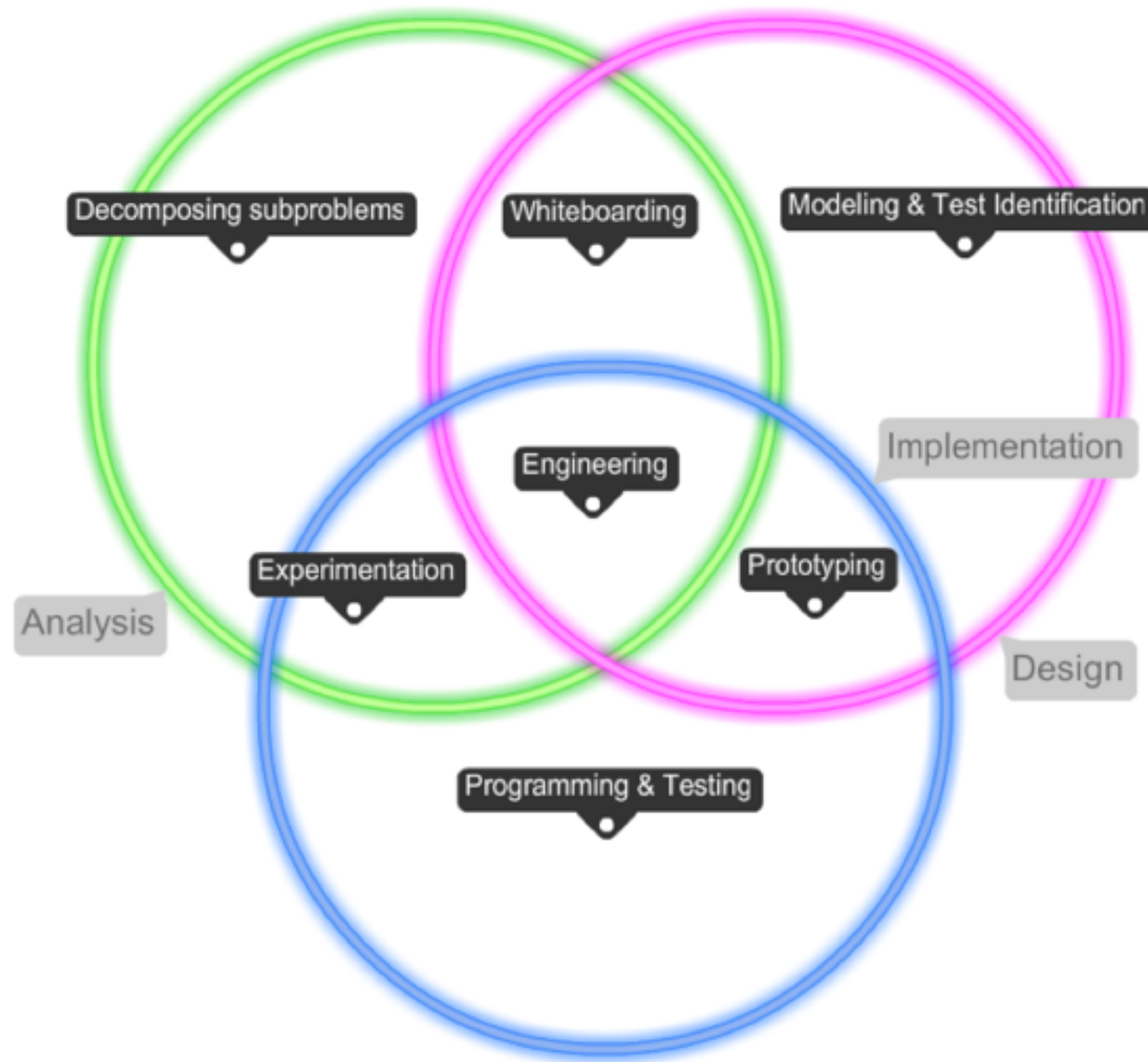
# Design

- Model some classes based on your analysis
  - Identify the responsibilities and relationships between classes.
- Are there design patterns you can use to solve the problem?
- Continue to analyze. Looking for opportunities to add constraints that eliminate complexity.

# Implementation

- Decide how to distribute work.
  - Who works on what subproblem?
- Start coding based on your design.
- Continue to refine your understanding of the problem and the design approach.

# Iterative A/D/I



# Design Patterns



# What are Design Patterns?

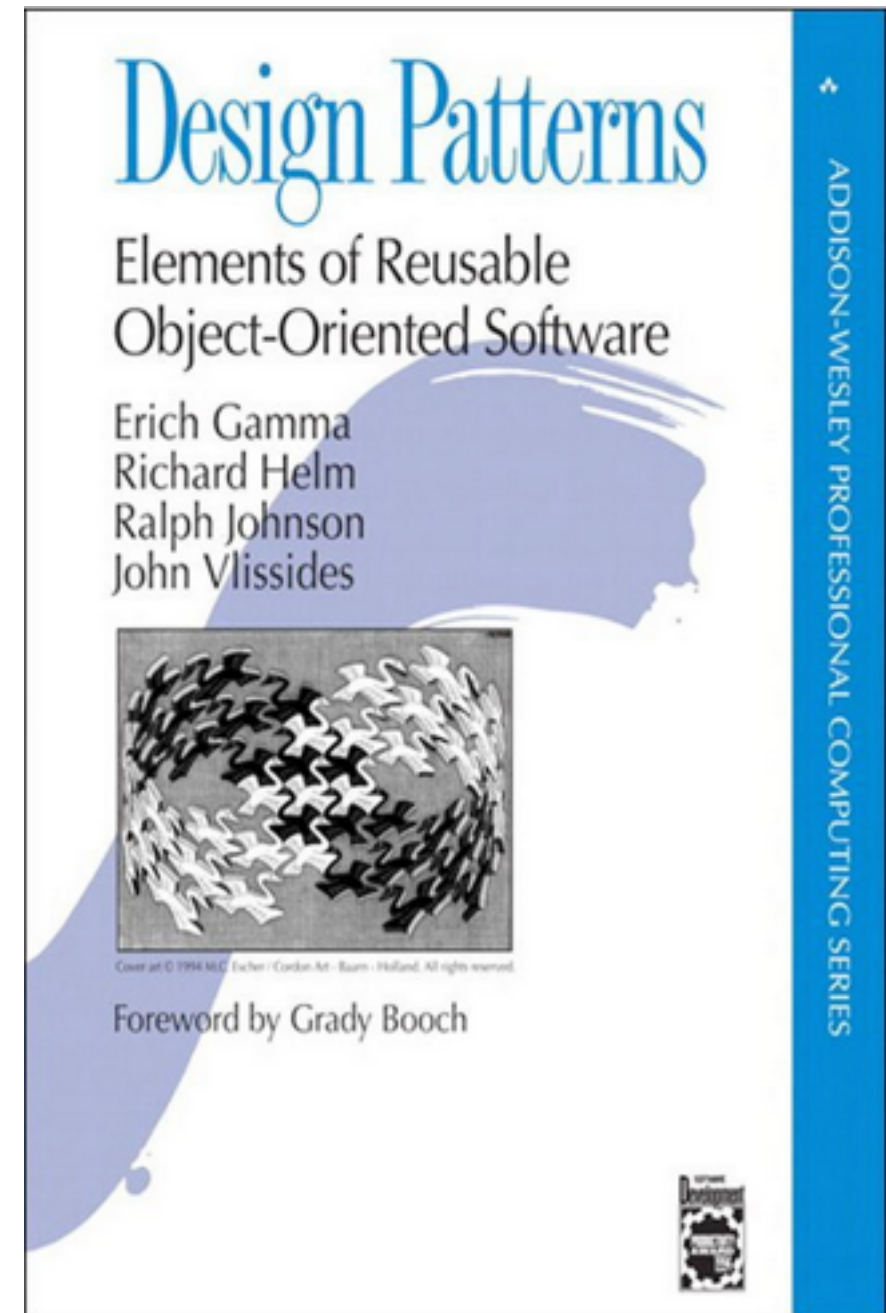
- is a general, reusable solution to a commonly occurring problem
- is abstract from programming languages
- identifies classes and their roles in the solution to a problem
- patterns are not code, only design; must be applied
- ..some would argue are indicative of missing language features!

# Why are Design Patterns?

- patterns are a common design vocabulary
- allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation
- patterns capture design expertise and allow that expertise to be communicated
- promotes design reuse and avoid mistakes
- improve documentation (less is needed) and understandability (patterns are described well once)

# Canonical Text

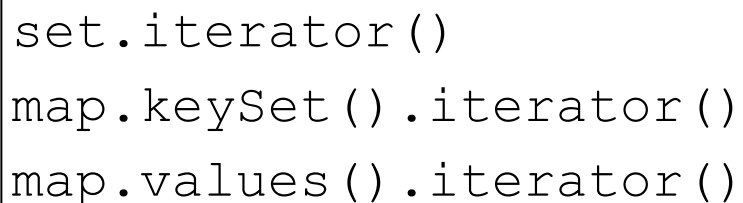
- Design Patterns: Elements of Reusable Object-Oriented Software
- A classic.
- Authors known as "Gang of Four"
- If you are interested in this subject, this is the text to get.
- (Really useful knowledge for interviews!)



# Iterator Pattern

- *iterator*: an object that provides a standard way to examine all elements of any collection
- uniform interface for traversing many different data structures
- supports concurrent iteration and element removal

```
1  for (Iterator<Account> itr = list.iterator(); itr.hasNext(); ) {  
2      Account a = itr.next();  
3      System.out.println(a);  
4  }
```



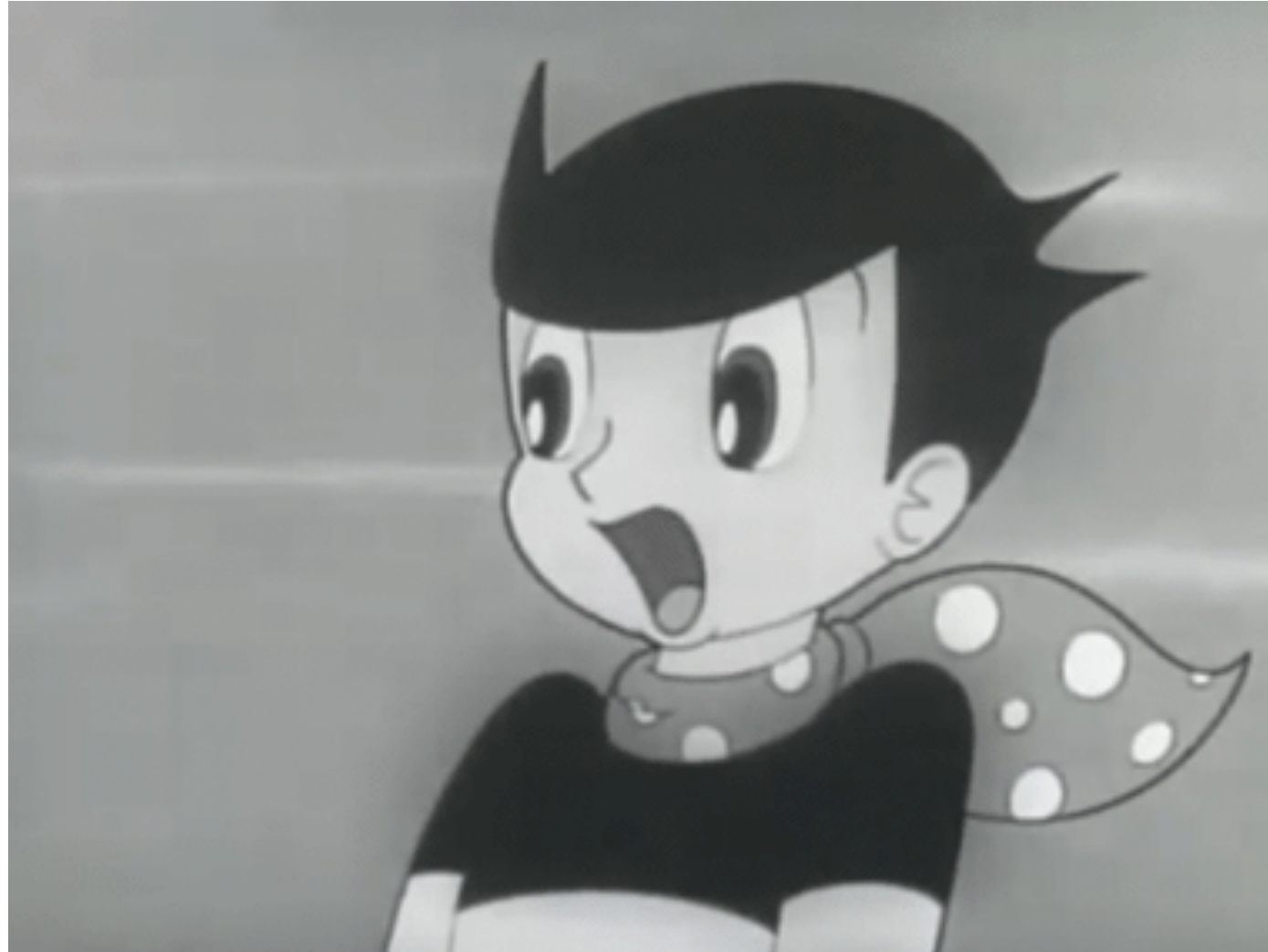
set.iterator()  
map.keySet().iterator()  
map.values().iterator()

# Chain of Responsibility

- A design pattern consisting of a source of 'command' objects and a series of 'processing' objects
- Each processing object contains logic that defines the types of command objects that it can handle;
- A processing 'pipeline', good for programs that pass the same data through a series of 'phases'.

# Chain of Responsibility

- Our code example is as follows:
  - A company needs to have any expenditure approved.
  - Depending on the price, the expenditure needs approval by different levels of management
  - Small priced expenditures require manager approval whereas large priced expenditures require director approval and so on
- **We can apply the CoR as a design pattern to solve this problem!**



Confused? Lets look at some code.

<https://github.com/nyu-oop/chain-of-responsiblity>

Sbt



# What is Sbt (3 things)

- It is an extensible, interactive **build tool**.
- Provides commands to build, test and run your code.
- Those commands are executed in a interactive console.
- Other build tools...
  - Make, Gradle, Stack, Rake...
- Sbt typically associated with Scala projects, but works equally well for Java projects.

# What is Sbt (3 things)

- It is a **dependency manager**
- Most non-trivial software projects have code libraries that they depend on. (Ex. Junit)
- Where do those libraries come from? Bundled with the code? That presents problems (such as version management)
- Sbt fetches your dependencies from the internet

# What is Sbt (3 things)

- It is a **project structure**
  - It has a specific set of conventions for layout of a Java project
- These questions are all answered by Sbt...
  - Where does code go?
  - Where to tests go?
  - How about project configuration?

# Sbt Directory Structure

- Every Sbt project we work with this semester will have the same structure, as follows..
  - **build.sbt** - project configuration. Where dependencies and custom commands are define
  - **lib** - the directory that contains *unmanaged* dependencies
  - **logs** - when using the logging class, output will go here as well as the console.
  - **output** - the directory for target code, will also contain provided C++ library code
  - **project** - sbt configuration, you should not need to touch this
  - **src/main/java** - your code for the project
  - **src/test/java** - your unit tests
  - **src/main/resources** - xtc.properties configure your team name and other details here.
  - **target** - compiled class files

# Sbt Commands

- Commands we care about for this course
  - **compile** - compiles the java code in your project
  - **run** - looks for the main method and runs it, if there is more than one, it gives you a menu
  - **clean** - removes compiled class files
  - **test** - runs all unit tests in your project
  - **test-only** - runs specific tests based on a pattern. Globbing allowed with asteriks. ex. test-only \*MethodVisitorTest\*

# Sbt Extensions

- Commands I have created (defined in **build.sbt**)
  - **format** - formats all code (c++, java tests and main source) using astyle
  - **compilec** - compiles c++ in output directory
  - **execc** - executes c++ in output directory
  - **cpp** - alias for both the previous commands
  - **runxtc** - runs the Boot class which is the main class of the translator.

JUnit

# What is JUnit

- JUnit is a unit testing framework for Java.
- Unit testing refers to the testing of the functionality provided by an individual class.
- Configuration and execution is all managed by Sbt.
- You need to know 3 things...
  1. How to write tests.
  2. How to use assertions.
  3. How to execute tests.





**WARNING**

We are using JUnit v4, which introduced significant changes from v3. *Many* tutorials on the interwebs are using JUnit v3.

# How to write tests

- Create a Java class, put it in `src/test/java/some/` package directory that mirrors the `src/main/java/some/` package directory that the code you want to test is in
- Write some test methods (using *assertions*) and put the `@Test` annotation on that method
- Profit
- There are also the `@Before`, `@After`, `@BeforeClass` and `@AfterClass` annotations (see `ExampleJUnitTest.java` in the `xtc-demo` repo)

# How to use Assertions

- Assertions are functions, provided by JUnit, that test that some expected condition is true.
- This is the way we signal to JUnit if tests pass or fail.
- There are many provided assertion types, but essentially they all can be reduced to the following..
  - `assertEquals("this", "that");`
- However, that can be inconvenient, so JUnit provides many assertion methods (see `ExampleJUnitAssertions.java` in the xtc-demo repo)

# How to execute tests

- You know already! Right....?

# Git & Github

(The 'No Frills' version, adapted from <http://rogerdudler.github.io/git-guide/>)

# Git is a Vcs

- A “version control system”
- Change tracking on files. "Backup" of versions of files, if you so choose.
- Enables multiple people to work on same code without too much headache.
- Git was initially designed and developed by Linus Torvalds for Linux kernel development

# Git is distributed

- Every Git ‘working directory’ is a full-fledged repository with complete history and full version-tracking capabilities.
- A ‘working directory’ is just a copy on disk a ‘repository’
- A ‘repository’ is a code base that you want to collaborate on with others. (sometimes called a ‘repo’ for short)

# Github is Git Hosting Service

- Github has generously donated an 'organization' to us.
- An organization is just a private site for us to share repositories as a group.
- Github will contain repos for each of homeworks, in-class code, each team's translator, etc..
- We will effectively download the code from Git to work on it, then we will upload the code back there so other team members can retrieve it. (We will do this through git commands.)

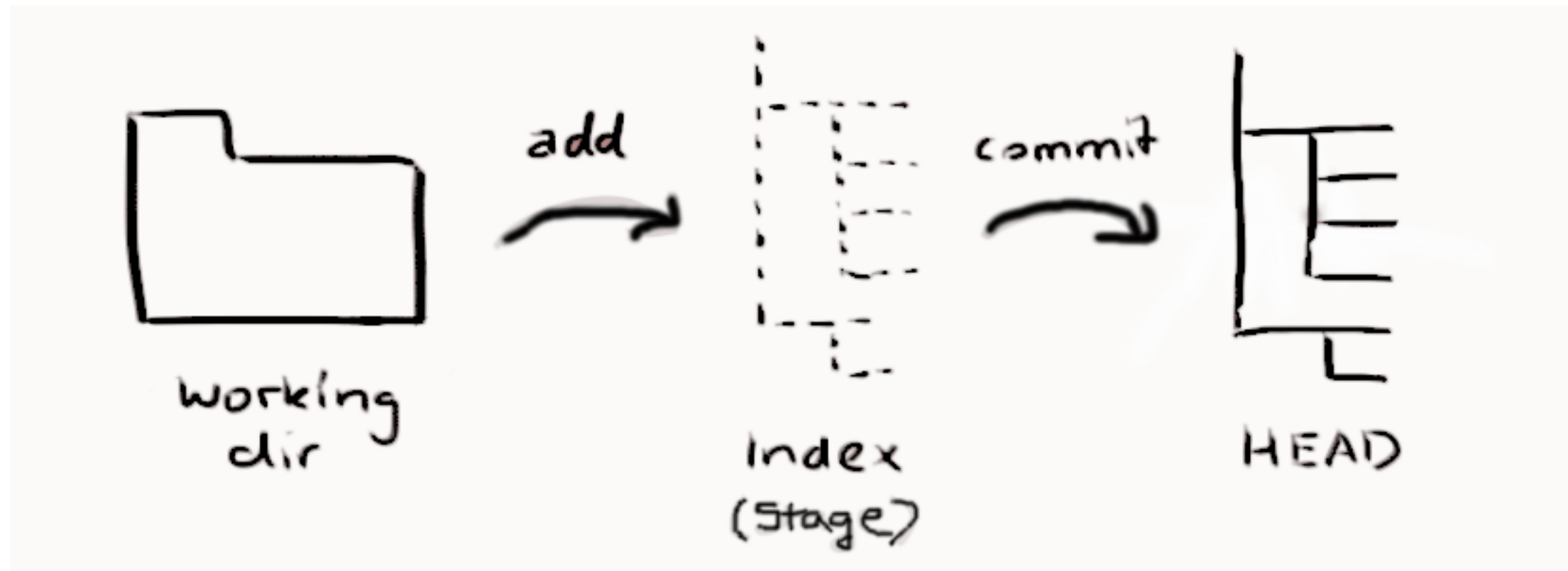


# Basic Git Workflow

1. First, you 'clone' a repository from Github.  
(translation: make a local copy, you do this only once!)
2. Next you 'add' new files and modify existing files.
3. Then you 'commit' those changes and additions.  
(translation: take a backup of that version)
4. Finally, you will 'push' that code to Github

# Steps 1-3

- After you've cloned a repo...
  - your local repository consists of three "trees" maintained by git.
  - the first one is your 'working directory' which holds the actual files.
  - the second one is the Index which acts as a staging area
  - and finally the HEAD which points to the last commit you've made.



# Steps 1-3

- You can propose changes (add it to the Index) using
  - `git add .`
- To actually commit these changes use
  - `git commit -am "Commit message"`
- Now the file is committed to the HEAD, but not in your remote repository yet.
- (Don't break the build!! I.e. do not commit code that is known to be broken)

# Step 4

- Your changes are now in the HEAD of your local working copy. To send those changes to your remote repository, execute
  - `git push origin master`

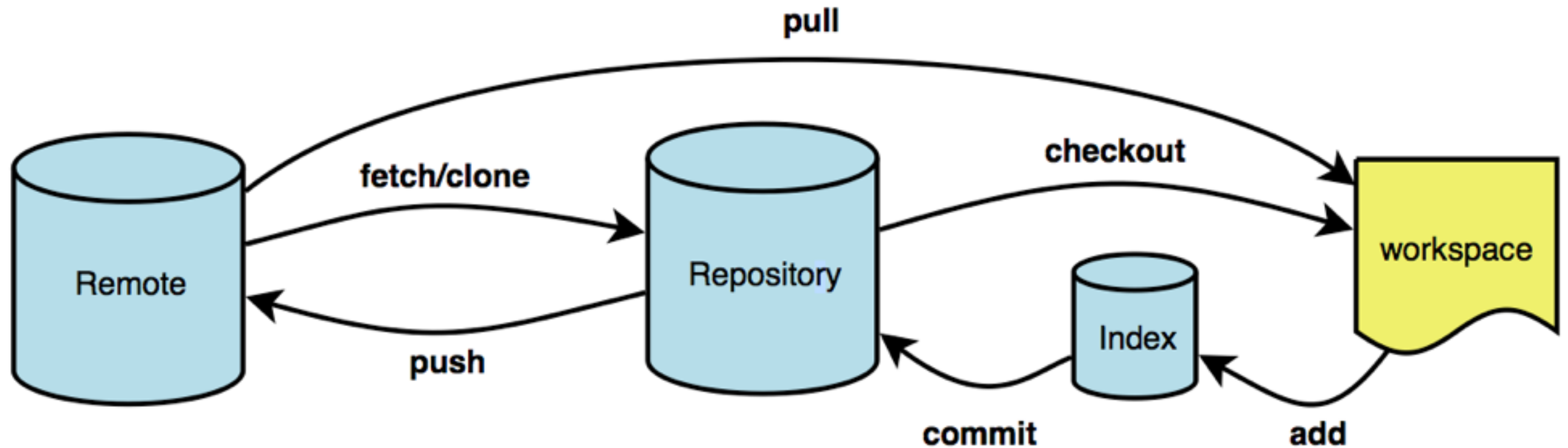
# Step x.5

- Interleaved throughout that process you may want to see if your teammates have pushed anything.
- You can get their code by executing..
  - `git pull origin master`

# Update & Merge

- git tries to auto-merge changes.
- This is not always possible and results in conflicts.
- You are responsible to merge those conflicts manually by editing the files shown by git.
- After changing, you need to mark them as merged with..
  - `git add <filename>`

# Moreover...



# What about branching?!??!?

(We've covered enough for today. You'll need to, but you will work that out with your team.)



# Learning Resources

- Interactive tutorial on Git
  - <https://try.github.io/levels/1/challenges/1>
- Interactive tutorial on Git Branching
  - <http://pcottle.github.io/learnGitBranching/>