

# Lecture 1



# A TOUR OF SOFTWARE DESIGN

CSC 207 SOFTWARE DESIGN

=> = implies  
w/ = with  
b/w = between  
:: = because  
:: = therefore



# LEARNING OUTCOMES

- Understand that (good) industry software is organized into layers
  - User interface and persistence, interface adapters, use cases, core classes
- Explain why each layer has a “public interface”
  - the set of classes and methods that it exposes to the world
  - often called an application programming interface (API)
- Explain which parts of a program need to change when moving to a new platform, and how to structure your program to enable this
- Get a feel for what this course will be like

# CSC108/148/110/111 STUFF YOU KNOW

- value and type; expressions
- naming a value using an assignment statement (assigning a value to a variable)
- control flow: sequence of statements, if, while, for, function call, return statement, call stack, recursion
- ADTs and data structures: string, list, dictionary, linked list, stack, queue, tree
- classes and the objects they describe; composition; inheritance (OOP)
- some variables and methods are private (Python: use a leading `_` underscore)
- computational complexity (big-Oh)
- unit testing, debugging
- function and class design recipes — processes by which to write code



# USE CASES FOR A PROGRAM

Imagine you were asked to write a program that allows users to

- **register a new user account** (with a username and password)
- **log in to a user account**
- **log out of a user account**
- They're planning on having a few different kinds of accounts, but we'll start with just one for now.

Bold words are *use cases*: what will the user want to do?

1. What **data needs to be represented**?
2. What **data structure** might you use while the program is running?
3. What should happen if the user **quits** and **restarts** the program?

# USE CASE: USER REGISTERS NEW ACCOUNT

- The user chooses a username
- The user chooses a password and enters it twice (to help them remember)
- If the username already exists, the system alerts the user
- If the two passwords don't match, the system alerts the user
- If the username exists in the system and the passwords match, then the system shows that the user is logged in

Operationalization  
of use cases

# USE CASE: USER LOGS IN

- The user enters a username and password
- If the username exists in the system and the passwords match, then the system shows that the user is logged in
- If there is no such username, the system alerts the user
- If the password doesn't match the one in the system, the system alerts the user

# USE CASE: USER LOGS OUT

- The system logs the user out and informs the user

# USE CASE: WHEN LOGGED OUT, CHOOSE USE CASE

- The user chooses between the *user registers a new account* use case and the *user logs in* use case

# BURNING QUESTIONS

- What is the **user interface**?
  - A webpage?
  - A Java application on your computer?
  - A Python command-line program?
  - A mobile app?
- How to do **data persistence**?
  - A text file?
  - A database?
  - Google Drive/OneDrive/etc.?
- How can you design your program so that it's **easy to move to a new UI**?  
*Easy UI migration*
- How can you design your program so that it's easy to save data to a different kind of storage?  
*Easy Data migration*
- How can you **design** your program so that **as much code as possible** stays the same when you do these things?

# DESIGN CONUNDRUMS

- How can we design the use cases so that they **do not directly depend** on the UI and persistence choices?
  - Then we can test all the use cases thoroughly!
- What are the **use case APIs**?
  - What is the interface to each use case?
  - What public methods do we want to provide to call the use cases from the UI?
- What persistence methods will we need in any storage?
  - Saving
  - Finding a user by username
  - Etc.

# TERMINOLOGY (OH DEAR ME!)

- **Entity**: a basic bit of data that we're storing in our program (like a user with a username and password)
- **Factory**: an object that knows how to make a class instance (typically by calling a constructor)
- **Use case**: something a user wants to do with the program
- **Input boundary**: the public interface for calling the use case
- **Interactor**: the class that runs the use case (a subclass of the input boundary)
- **Repository**: the persistence mechanism (a subclass of the Gateway)
- **Gateway**: the methods the repository needs to implement for the Interactor to do its job
  - this is designed with the use case
- **Controller**: the object that the UI asks to run a use case
- **Presenter**: the object that tells the UI what to do when a use case finishes
- **Model**: a temporary object to pass data between layers

# USE CASE APIs

A class for each use case

UserRegisterInteractor

- **Public API:** a method called `create` (perhaps with the username and password as parameters, or maybe they're wrapped up in a data class)
- Needs to tell the UI to prepare a **success view** or a **failure view**
- Needs to know **how to make a new user**
- Needs to **ask the persistence layer whether a username exists**
- Needs to **tell the persistence layer to save a new user**

Only thing that can  
be accessed outside  
of class

} Private /  
behind the  
scenes.



# REGISTER NEW USER USE CASE SKETCH

```
class UserFactory:  
    def create(name: str, password: str) -> User
```

```
class UserRegisterInteractor:
```

Factory Design  
Pattern

```
_user_factory # a UserFactory ← obj of factory  
_user_presenter # controls the UI ← obj of presenter
```

```
_user_register_gateway # a persistence object Allows access to database
```

```
create(name: str, password: str) -> UserRegisterResultModel  
# On success, this method returns an object with the  
# username and creation time, but not the password (why?)
```

Is this a  
model?

# THE USER INTERFACE

would this come under  
user presenter?

```
# The window for user registration
```

```
class UserRegisterScreen:
```

```
    # a username text field
```

```
    # a password text field
```

```
    # a "repeat password" text field
```

```
    # a Register button that, when clicked, asks the controller to
```

```
    # run the "register new account" use case
```

Thought question: could we do this with a command-line interface?

Yes you could do this with a CLI w/ modification



# LEARNING OUTCOMES

- Understand that (good) industry software is organized into layers
  - User interface and persistence, interface adapters, use cases, core classes
- Explain why each layer has a “public interface”
  - the set of classes and methods that it exposes to the world
  - often called an application programming interface (API)
- Explain which parts of a program need to change when moving to a new platform, and how to structure your program to enable this
- Get a feel for what this course will be like

# Lecture 2

## JAVA OVERVIEW

CSC 207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

- Understand how Java works at a high level.
- Be aware of what concepts you will be learning through the Quercus quizzes and readings.
- Be familiar with what resources are available to help you learn the fundamentals of Java.
- After completing the Quercus quizzes, readings, and exercises, you should have a working knowledge of Java so that you can contribute code to your group project throughout the term (or know where to look if you need to learn more about a specific language feature).



# REFERENCE MATERIAL ON JAVA

- See the Github Classroom Coding exercises for the course and the course notes linked from Quercus.
- The official java tutorials are a great place to learn (specific links provided throughout these slides):  
<http://docs.oracle.com/javase/tutorial/java/TOC.html>
- This website also does a nice job walking you through Java:  
<https://www.sololearn.com/Course/Java/>
- Please share any other resources you find useful on Piazza for others to benefit from.
- Ask on Piazza if you have questions or visit office hours. Everyone is here to help.

# VERSION OF JAVA IN THIS COURSE

- In most materials we focus on the core language features and point to the official Java tutorials, which were written for Java 8.
- There are some nice language features in later versions of Java, but we largely won't be emphasizing them in this course.
- For example, the var keyword introduced in Java 10 isn't something that we'll be discussing, and we recommend you initially don't make use of it while learning the basics of Java.
- For those interested, <https://developer.oracle.com/java/jdk-10-local-variable-type-inference.html> has a rather nice discussion of the var keyword and type inference.
- We have suggested you work with Java 11, but once you start working on the project, your team can agree on which version you want to use.

# RUNNING PROGRAMS

- What is a program?
- What does it mean to “run” a program?
- To run a program, it must be translated from its **high-level programming language** to a **low-level machine language** whose instructions can be executed.
- Roughly, there are two flavours of translation:
  - Interpretation
  - Compilation

# INTERPRETED VS. COMPILED

- **Interpreted:**

- e.g., Python
- Translate and execute one statement at a time

- **Compiled:**

- e.g., C
- Translate the entire program (once), then execute (any number of times)

- **Hybrid:**

- e.g., Java
- Translate to something intermediate (in Java, bytecode)
- Java Virtual Machine (JVM) runs this intermediate code

# COMPILING JAVA

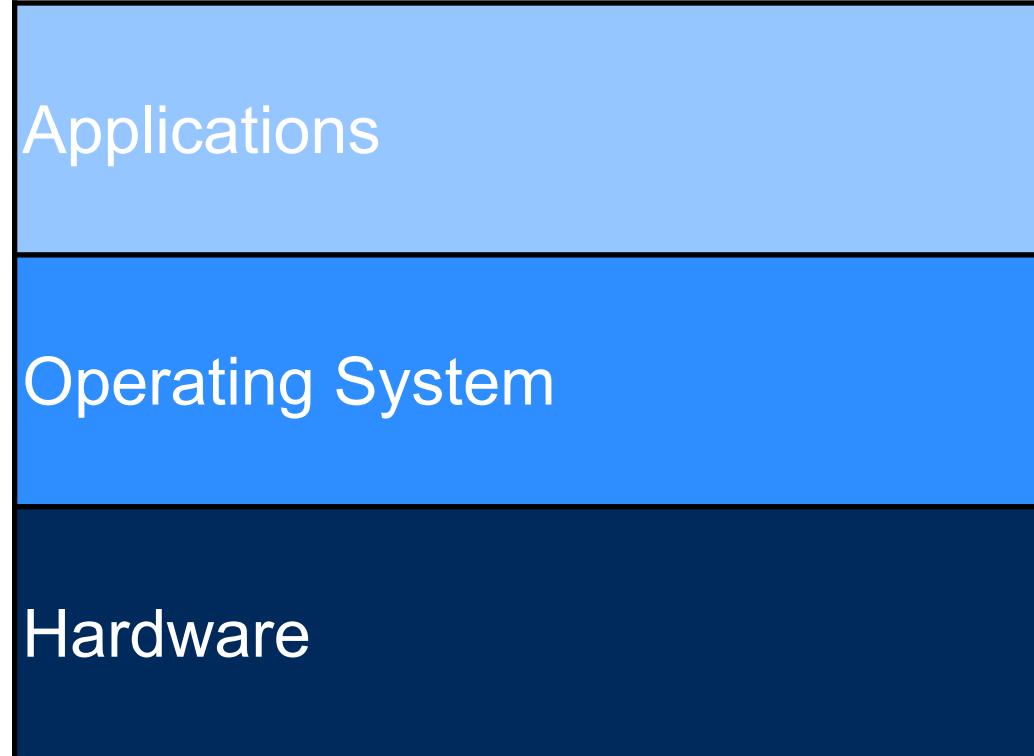
- You need to compile, then run (as described above).
- If using the command line, you need to do this *manually*. For example:

```
First, compile using "javac":  
diane@laptop$ javac HelloWorld.java  
This produces file "HelloWorld.class":  
diane@laptop$ ls  
HelloWorld.class  HelloWorld.java  
Now, run the program using "java":  
diane@laptop$ java HelloWorld  
Hello world!
```

- Modern IDEs (like IntelliJ) do this for you with the help of a build system (e.g. maven or gradle)
  - Build systems take care of the build process, so that you can focus on the code.
- But you should have some idea of what's happening under the hood!
  - If you go on to take CSC209, you will learn C and get much more practice with compiling code.

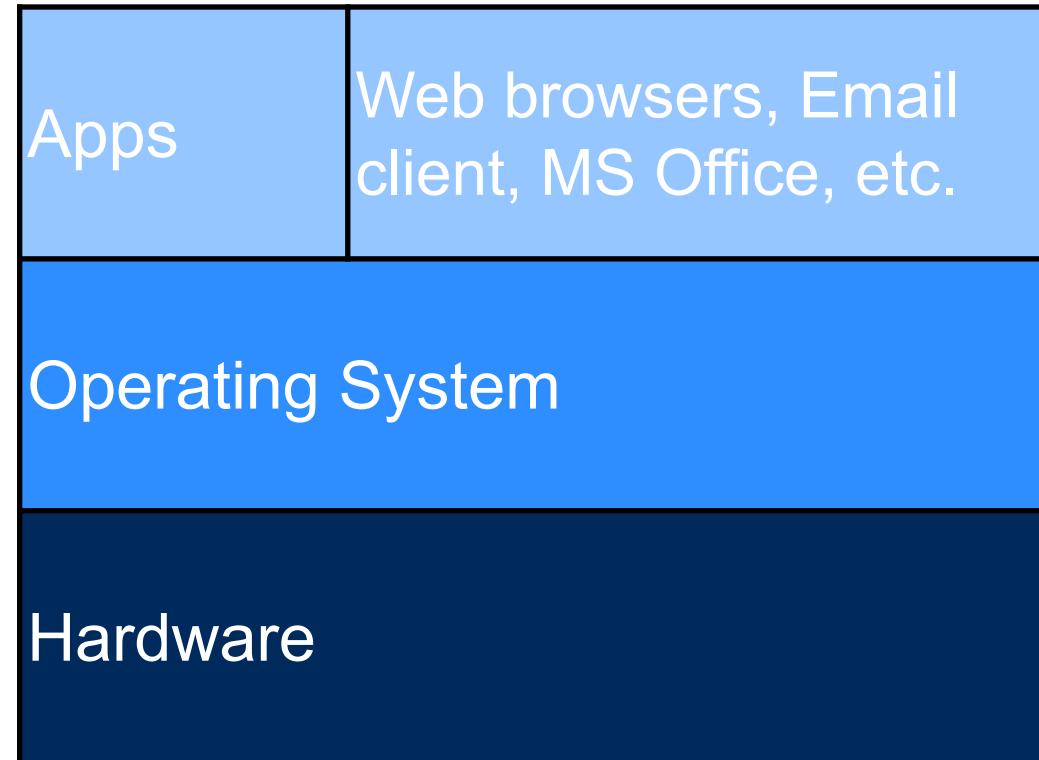
# COMPUTER ARCHITECTURE

- 108/148 and 110/111: we focused on applications.
- The operating system (OS) manages the various running applications and helps them interact with the hardware (**CSC209H**, **CSC369H**).
- The OS works directly with the hardware (**CSC258H**, **CSC369H**).



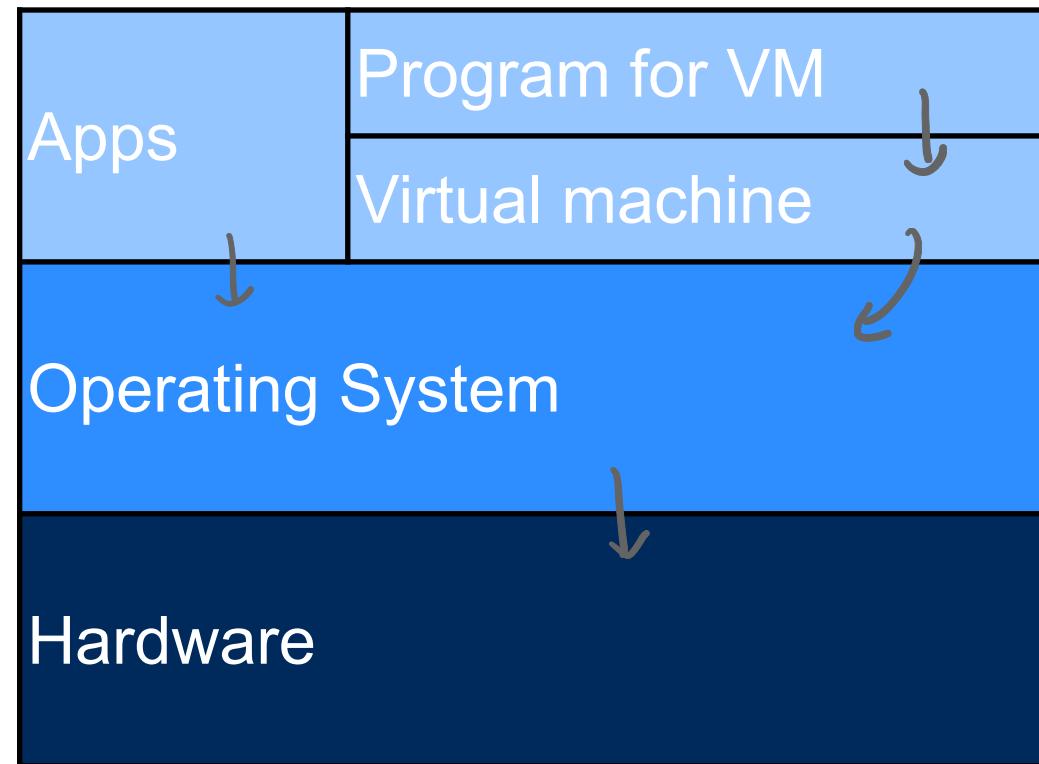
# COMPILED APPLICATIONS

- Programs written in languages like C, Objective C, C++, Pascal, and Fortran are **compiled** into OS-specific applications.
- Compilation involves turning human-readable programs into OS-specific machine-readable code (CSC258H, CSC369H, CSC488H).



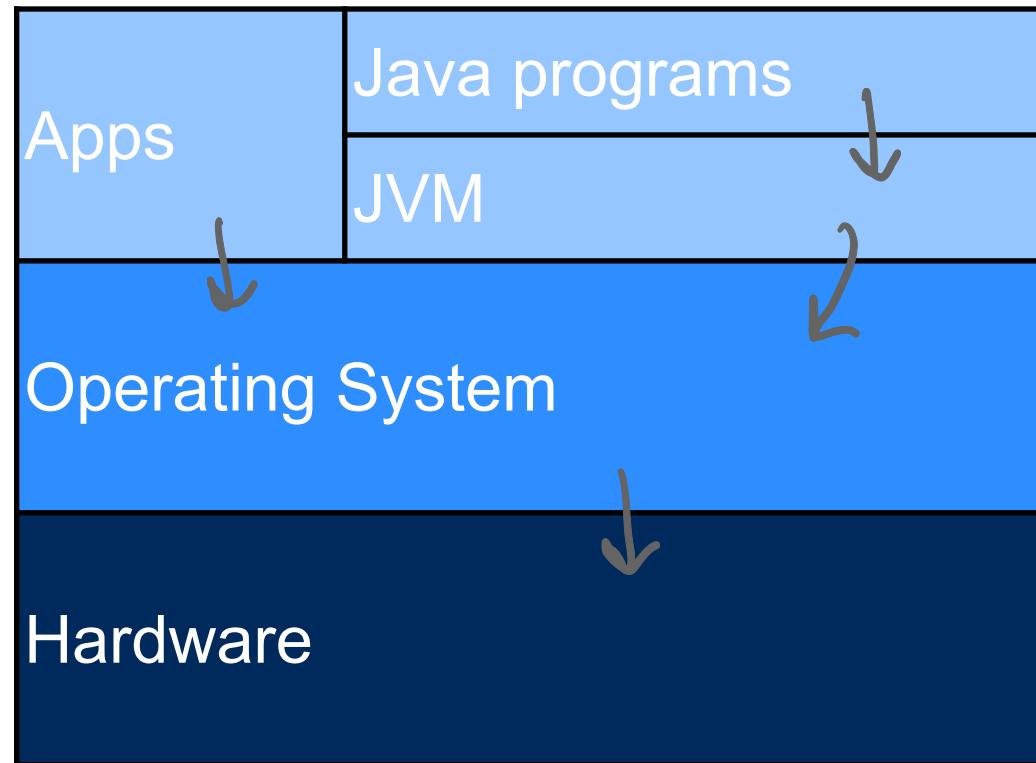
# VIRTUAL MACHINE ARCHITECTURE

- **Virtual machine (VM)**: an application that pretends to be a computer.
- For example, each of Java, Python, and Scheme have their own VM.
- The VM application is written for each OS so that programs are portable across operating systems.



# JAVA ARCHITECTURE

- **Java Virtual Machine (JVM)**: an application that is something like an operating system
- Java programs are compiled to an intermediate state called **byte code**: machine code for the JVM
- The same compiled Java program can run in any JVM on any OS!
- JVMs optimize the byte code as it runs (CSC324H, CSC488H) and can even be as fast as code written in C!



# **TYPES**



# TYPES ARE CRITICAL

- Python uses “Duck Typing”
  - Types are checked at runtime, and any object with the appropriate capabilities (methods) is legal.
  - Can lead to hard-to-find bugs if we aren’t careful, which is why we encourage the use of "type annotations"
- Java tries to catch errors early, so it checks types **before** the program is run.
  - This helps catch a lot of silly mistakes / typos.

# WHY CHECK TYPES?

- We often think in terms of types: “this value is a ...” or “this value can ...”
    - Java tells us when our assumptions are incorrect.
  - Types don’t always have to match exactly.
    - We can replace a parent class with one of its children, for example.
- Note all types are classes
- ↑ Super types can be replaced with sub types



# **DEFINING CLASSES IN JAVA**

# INSTANCE VARIABLES

- public class Circle {
  - private String radius;
  - }
- 
- radius is an instance variable. Each instance of the circle class has its own radius variable.

# CONSTRUCTORS



- A constructor has:

- the same name as the class
- no return type (not even void)

- A class can have multiple constructors, but their signatures must be different.
- If you define no constructors, the compiler supplies one with no parameters and no body.
- But if you define any constructor for a class, the compiler will no longer supply the default constructor.

```
class test {  
    public test ( int a ) {  
        this.a = a  
    }  
}
```

class test {  
 public test () {}



# THIS

- **this** is an instance variable that you get without declaring it.
- It's like **self** in Python.
- Its value is the address of the object whose method has been called.
- It can be useful to disambiguate between an instance variable and a parameter with the same name.

# DEFINING METHODS

- A method must have a return type declared. Use `void` if nothing is returned.
- The form of a return statement:  
`return expression;`

If the expression is omitted or if the end of the method is reached without executing a return statement, nothing is returned.

- Must specify the accessibility. For now:  
`public` – callable from anywhere  
`private` – callable only from this class
- Variables declared in a method are local to that method.

# PARAMETERS

- When passing an argument to a method, you pass what's in the variable's box:
  - For class types, you are passing a reference.  
(Like in Python.)
  - For primitive types, you are passing a value.  
(Python can't do anything like this.)
- This has important implications!
- You must be aware of whether you are passing a primitive or an object.
  - This is a similar idea to mutable and immutable objects in Python.
- You may hear people talk about "pass by reference" and "pass by value" as you learn more programming languages.

# INSTANCE VARIABLES AND STATIC VARIABLES

- class Sneetch {  
•     private String name;  
•     private boolean starBellied;  
•     private static int howMany = 0;  
•     public static final String SAYING =  
•   "Best on the  
•     Beeches.";  
• }
- name and starBellied are instance variables.
- howMany is a static or class variable.
- SAYING is (static and) final so it's OK to make it public.
  - ... if there's also a good reason to do that of course.
- You can mix instance variable and class variable declarations with method definitions in any order you like, but organize things in a way that makes sense.

Good Practice to capitalize  
final & static  
public variables.



# INSTANCE VARIABLES AND ACCESS

- If an **instance variable** is private, how can client code use it?
- Why not make everything public — so much easier!

↑ ∵ clients are dumb and  
stupid and bad

[Encapsulation]



# ENCAPSULATION

- Think of your **class** as **providing** an abstraction, or a **service**.
  - We provide access to information through a **well-defined interface**: the public methods of the class.
  - We **hide** the implementation details.
- What is the advantage of this “encapsulation”?
  - We can change the implementation — to improve speed, reliability, or readability — and **no other code has to change**.

Interface is the same  
∴ already implemented  
code still works.



# CONVENTIONS

- Make all non-final instance variables either:
  - *private*: accessible only within the class, or
  - *protected*: accessible only within the package.
- When desired, give outside access using “getter” and “setter” methods, rather than making instance variables public.
- Note: A *final* variable cannot change value; it is a constant. That doesn't mean that you can't mutate the object that it refers to though — you just can't assign it to refer to a *different* object.

# ACCESS MODIFIERS

- Classes can be declared public or package-private.
- Members of classes can be declared public, protected, package-protected, or private.

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default (package private)	Yes	Yes	No	No
private	Yes	No	No	No

Global  
(accessible everywhere)



# INHERITANCE IN JAVA



# INHERITANCE HIERARCHY

- All classes form a tree called the inheritance hierarchy, with Object at the root.
- Class Object does not have a parent. All other Java classes have one parent.
- If a class has no parent declared, it is a child of class Object.
- A parent class can have multiple child classes.
- Class Object guarantees that every class inherits methods `toString`, `equals`, and others.

<https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>

# INHERITANCE

- Inheritance allows one class to inherit the data and methods of another class.
- In a subclass, `super` refers to the part of the object defined by the parent class.
  - Use `super.«attribute»` to refer to an attribute (data member or method) in the parent class.
  - Use `super( «arguments» )` to call a constructor defined in the parent class.

# CONSTRUCTORS AND INHERITANCE

- If the first step of a constructor is `super( «arguments» )`, the appropriate constructor in the parent class is called.
  - Otherwise, the no-argument constructor in the parent is called.
- Net effect on order if, say, A is parent of B is parent of C?
- Which constructor should do what? Good practice:
  - Initialize your own variables.
  - Count on ancestors to take care of theirs.

# MULTI-PART OBJECTS

- Suppose class `Child` extends class `Parent`.
- An instance of `Child` has
  - a `Child` part, with all the data members and methods of `Child`
  - a `Parent` part, with all the data members and methods of `Parent`
  - a `Grandparent` part, ... etc., all the way up to `Object`.
- An instance of `Child` can be used anywhere that a `Parent` is legal.
  - But not the other way around.



# NAME LOOKUP

- A subclass *can* reuse a name already used for an inherited data member or method, but it can be confusing to do so.
- Example: class Person could have a data member motto and so could class Student. Or they could both have a method with the signature sing( ).
- When we construct

```
x = new Student();
```

the object has a Student part and a Person part.
- If we say `x.motto` or `x.sing()`, we need to know which one we'll get!
- In other words, we need to know how Java will look up the name motto or sing inside a Student object.



# NAME LOOKUP RULES

- For a method call: expression.method(arguments)
  - Java looks for the method in the most specific, or bottom-most part of the object referred to by expression.
  - If it's not defined there, Java looks "upward" until it's found (else it's an error).
- For a reference to an instance variable: expression.variable
  - Java determines the type of expression, and looks in that box.
  - If it's not defined there, Java looks "upward" until it's found (else it's an error).



# SHADOWING AND OVERRIDING

- Suppose class A and its subclass AChild each have an instance variable x and an instance method m.
- A's m is **overridden** by AChild's m.
  - This is often a good idea. We often want to specialize behaviour in a subclass.
- A's x is **shadowed** by AChild's x.
  - This is confusing and rarely a good idea.
  - Avoid instance variables with the same name in a parent and child class.
- If a method must not be overridden in a descendant, declare it final.

# CASTING FOR THE COMPILER

- If we could run this code, Java would find the `charAt` method in `o`, since it refers to a `String` object:

```
Object o = new String("hello");
char c = o.charAt(1);
```

- But the code won't compile because the compiler cannot be sure it will find the `charAt` method in `o`.

Remember: the compiler doesn't run the code – it can only look at the **type** of `o`.

- We need to cast `o` as a `String`:

```
char c = ((String) o).charAt(1);
```

# A FEW QUICK POINTS ABOUT INHERITANCE

- **super** refers to the part of the object that is defined by its parent class.
- You can call a constructor in the parent class by calling `super(arguments)`. If you don't do this explicitly, it will happen implicitly (and with no arguments) as the very first thing.
- A subclass can add new data members or methods to those it inherits.
- It can also reuse a name already in use for an inherited data member or method. There are subtleties though.
- Child classes don't have access to private members of their parent. You can read more about access modifiers at:

<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

# JAVADOC

- Like a Python docstring but placed above the method.

```
/**  
 * Replace a square wheel of diagonal diag with a round wheel of  
 * diameter diam. If either dimension is negative, use a wooden tire.  
 * @param diag Size of the square wheel.  
 * @param diam Size of the round wheel.  
 * @throws PiException If pi is not 22/7 today.  
 */  
public void squareToRound(double diag, double diam) { ... }
```

- Javadoc is written for classes, member variables, and member methods.
- This is where the Java API documentation comes from!
- <https://www.oracle.com/ca-en/technical-resources/articles/java/javadoc-tool.html> provides an in-depth discussion of how to write documentation for Java code.



# JAVA NAMING CONVENTIONS

- The Java Language Specification recommends these conventions
- Generally: Use camelCase not pothole\_case.
- Class name: A noun phrase starting with a capital.
- Method name: A verb phrase starting with lower case.
- Instance variable: A noun phrase starting with lower case.
- Local variable or parameter: ditto, but acronyms and abbreviations are more okay.
- Constant: all uppercase, pothole.
  - E.g., MAX\_ENROLMENT

# DIRECT INITIALIZATION OF INSTANCE VARIABLES

- You can initialize instance variables inside constructor(s).
- An alternative: initialize in the same statement where they are declared.
- Limitations:
  - Can only refer to variables that have been initialized in previous lines.
  - Can only use a single expression to compute the initial value.

# WHAT HAPPENS WHEN WE CREATE AN OBJECT?

1. Allocate memory for the new object.
2. Initialize the instance variables to their default values:
  - 0 for ints, `false` for booleans, etc., and `null` for class types.
3. Call the appropriate constructor in the parent class.
  - The one called on the first line, if the first line is `super(arguments)`, else the no-arg constructor.
4. Execute any direct initializations in the order in which they occur.
5. Execute the rest of the constructor.





# ABSTRACT CLASSES AND INTERFACES

- A class may define methods without giving a body. In that case:
  - Each of those methods must be declared abstract.
  - The class must be declared abstract too.
  - The class can't be instantiated.
- A child class may implement some or all of the inherited abstract methods.
  - If not all, it must be declared abstract.
  - If all, it's not abstract and so can be instantiated.
- If a class is completely abstract, we may choose instead to declare it to be an interface.



# INTERFACES

- An interface is (usually) a class with no implementation.
  - It has just the method signatures and return types.
  - It guarantees capabilities.
- Example: `java.util.List`
  - "To be a List, here are the methods you must support."
- A class can be declared to `implement` an interface.
  - This means it defines a body for every method.
  - A class can implement 0, 1 or many interfaces, but a class may extend only 0 or 1 classes.
- An interface may extend another interface.



# THE PROGRAMMING INTERFACE

- The "user" for almost all code is a programmer. That user wants to know:
  - ... what kinds of object your class represents
  - ... what actions it can take (methods)
  - ... what properties your object has (getter methods)
  - ... what guarantees your methods and objects require and offer
    - ... how they fail and react to failure
    - ... what is returned and what errors are raised



# INTERFACE LEFT, CLASS RIGHT

- `List ls = new List(); // Fails`
- `List ls = new ArrayList();`
- We can choose a different implementation of `List` at any point.
- The compiler guarantees that we don't call any methods that are in `ArrayList` but not in `List`...
  - ... unless we say so:

```
((ArrayList) ls).ensureCapacity(1000); // a cast
```

- As a rule, avoid this kind of explicit casting whenever you can.

# NAME RESOLUTION ORDER

1. Search for a local variable within the enclosing code block (loop, catch, etc.).
2. Within a method, search for a matching parameter.
3. Extend the search to a class or interface member. Start with the class itself, then search its ancestors.
4. Search within explicitly named imported types.
5. Search for other types declared in the same package.
6. Search the implicitly named imported types (the packages Java imports automatically).

# **GENERICS**

# GENERICs (FANCIER TYPE PARAMETERS)

- “`class Foo<T>`” introduces a class with a type parameter T.
- “`<T extends Bar>`” introduces a type parameter that is required to be a descendant of the class Bar — with Bar itself a possibility.  
In a type parameter, “extends” is also used to mean “implements”.
- “`<? extends Bar>`” is a type parameter that can be any class that extends Bar.  
We’ll never refer to this type, so we don’t give it a name.
- “`<? super Bar>`” is a parameter that can be any ancestor of Bar.

<https://docs.oracle.com/javase/tutorial/java/generics/types.html>

# AN INTERFACE WITH GENERICS

- ```
public interface Comparable<T> {  
    /**  
     * Compare this object with o for order.  
     * Return a negative integer, zero, or a  
     * positive integer as this object is less  
     * than, equal to, or greater than o.  
     */  
    int compareTo(T o); // No body at all.  
}
```
- ```
public class Student implements Comparable<Student> {  
    . . .  
    public int compareTo(Student other) {  
        // Here we need to provide a body for the method.  
    }  
}
```

# GENERICs: NAMING CONVENTIONS

- The Java Language Specification recommends these conventions for the names of type variables:
  - very short, preferably a single character
  - but evocative
  - all uppercase to distinguish them from class and interface names
- Specific suggestions:  
Maps: K, V  
Exceptions: X  
Nothing particular: T (or S, T, U or T1, T2, T3 for several)

# COLLECTIONS

- Equivalents to Python lists, dictionaries, and sets are in the standard “Collections” library.

```
import java.util.*;  
...  
List list = new ArrayList();    // list = []  
Map map = new HashMap();        // map = {}  
Set set = new HashSet();         // set = set()
```

<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html> for more information (the Design Goals section at the end is highly relevant to the design aspect of the course too!)

# HOMEWORK

- Work through the assigned Java exercises and course readings available through Quercus.
- Bookmark the Java documentation pages and tutorial links that you find helpful so that you can easily find them again.
- Set aside some time each week to practice writing Java code.



# Lecture 3



# VERSION CONTROL

CSC 207 SOFTWARE DESIGN



# LEARNING OUTCOMES

- Understand what version control is and why it is useful
- Understand the basics of how to use git
- Be able to follow a simple branch and merge workflow

# WHAT'S VERSION CONTROL?

- A master repository of files exists on a server.
- People “clone” the repo to get their own local copy.
- As significant progress is made, people “push” their changes to the master repo, and “pull” other people’s changes from the master repo.
- The repo keeps track of every change, and people can revert to older versions.

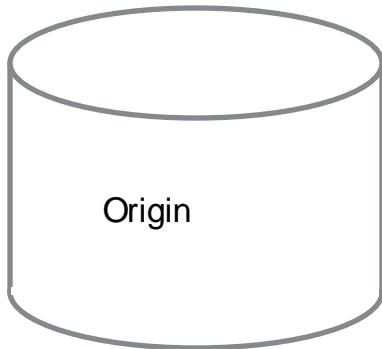
# WHY VERSION CONTROL?

- **Backup and restore** - because accidents happen
- **Synchronization** - multiple people can make changes
- **Short term undo** - that last change made things worse?
- **Long term undo** - find out when a bug was introduced
- **Track changes** - all changes related to a bug fix
- **Sandboxing** - try something out without messing up the main code
- **Branching and merging** - (better defined sandboxes)

# GIT

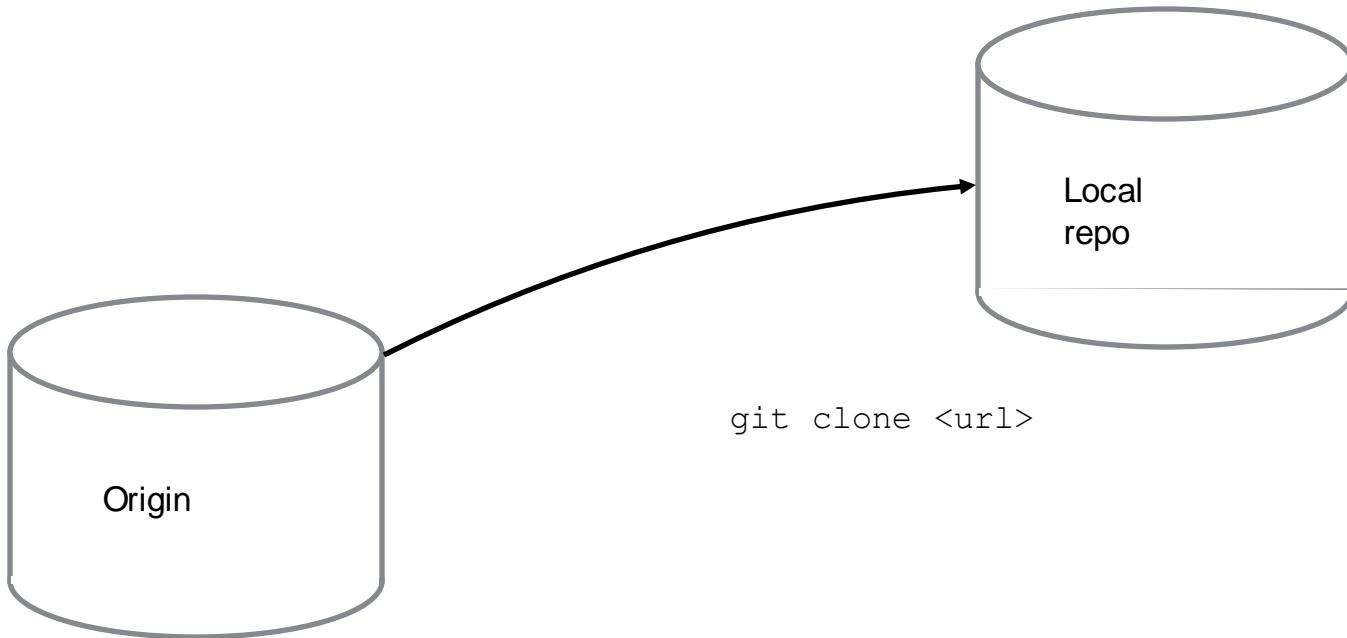
- For CSC207, we will usually create repositories for you.
- These repositories live on GitHub Classroom (or possibly MarkUs)
- You will **clone** your remote repository, work on files locally, **add** and **commit** changes to your local repository, and **push** changes to the remote repository.
- We'll only teach the basics in this course, so that you know enough to use it in your group project.
- We encourage you to read <https://en.wikipedia.org/wiki/Git>, which discusses the motivation for why Linus Torvalds decided to create git, as well as the system's design.

# REMOTE REPOSITORY



- This is the repo that lives on another server.
- By convention we call it the *origin*
- (In Github terminology, you may have an “upstream” repo and a forked copy of the repo called the “origin”, but we are working with a simpler model)

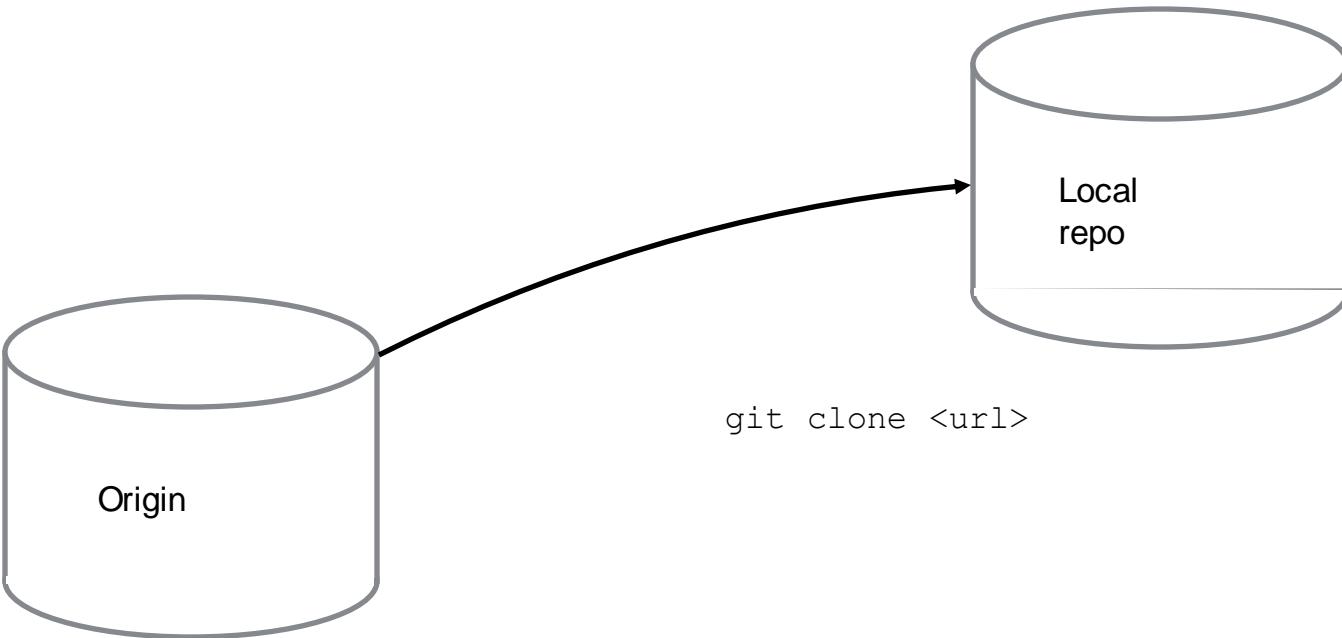
# CLONE TO GET LOCAL REPO



- The “clone” command makes a copy of the remote repository on your local machine.
- Now there are two repositories. (Git is a distributed version control system)



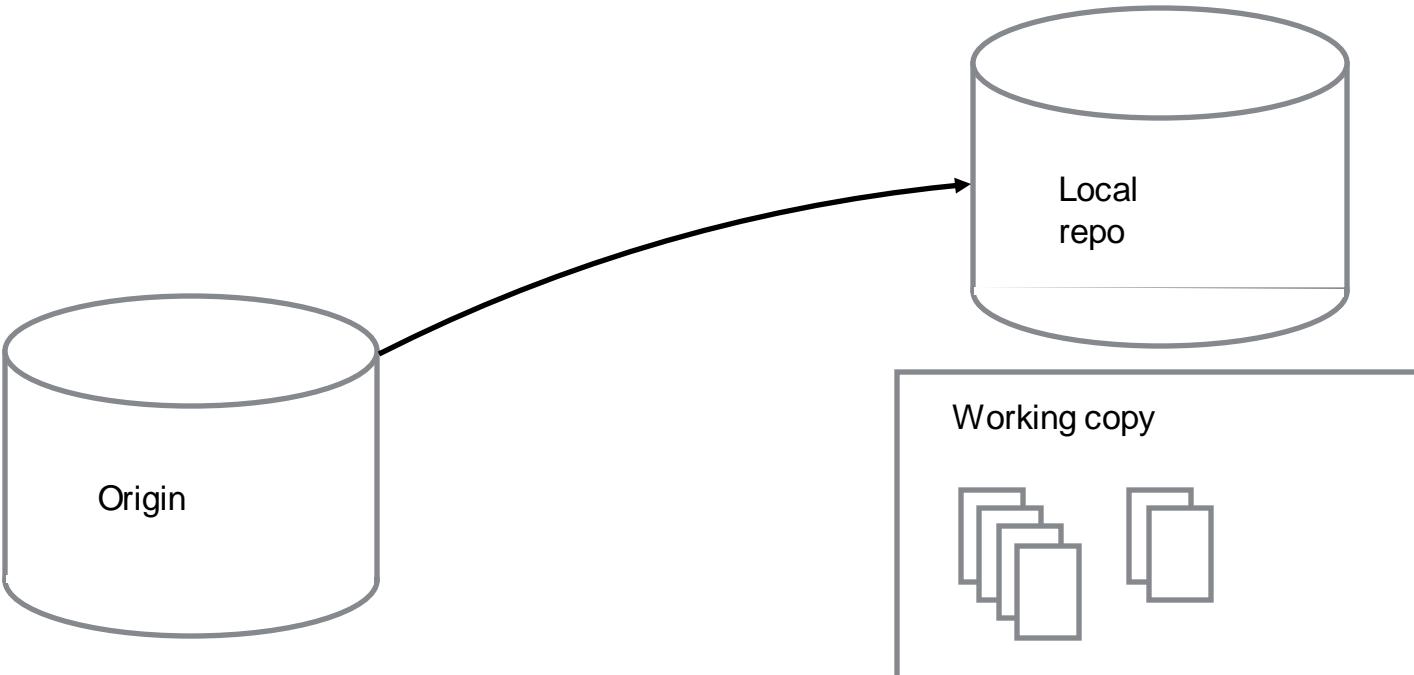
# CLONE TO GET LOCAL REPO



- The “clone” command makes a copy of the remote repository on your local machine.
- Now there are two repositories. (Git is a distributed version control system)



# CLONE TO GET LOCAL REPO



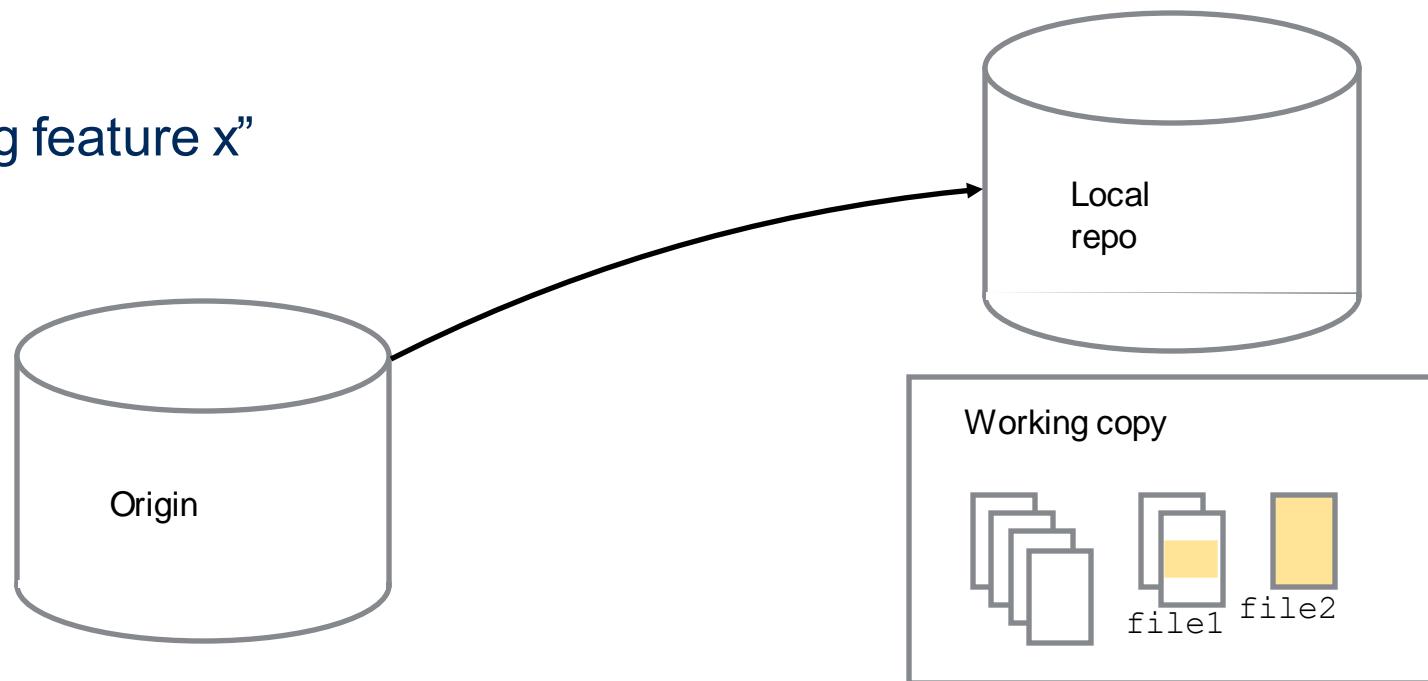
- The “clone” command makes a copy of the remote repository on your local machine.
- Now there are two repositories. (Git is a *distributed* version control system)

# LOCAL REPOSITORY

- The actual repository is hidden. What you see is the working copy of the files from the repository.
- Now you can create new files, make changes to files, and delete files.
- When you want to “commit” a change to the local repository, you need to first “stage” the changes

# HOW TO GET WORK DONE

- Make changes to files, add new files. When you are ready to commit your work to your local repo, you need to tell git which files have changes that you want to add this time.
- `git add file1 file2`
- `git commit -m "adding feature x"`



# STAGING CHANGES

- `git add` doesn't add files to your repo. Instead, it marks a file as being part of the current change.
- This means that when you make some changes to a file and then add and commit them, the next time you make some changes to a file you will still have to run `git add` to add the changes to the next commit.

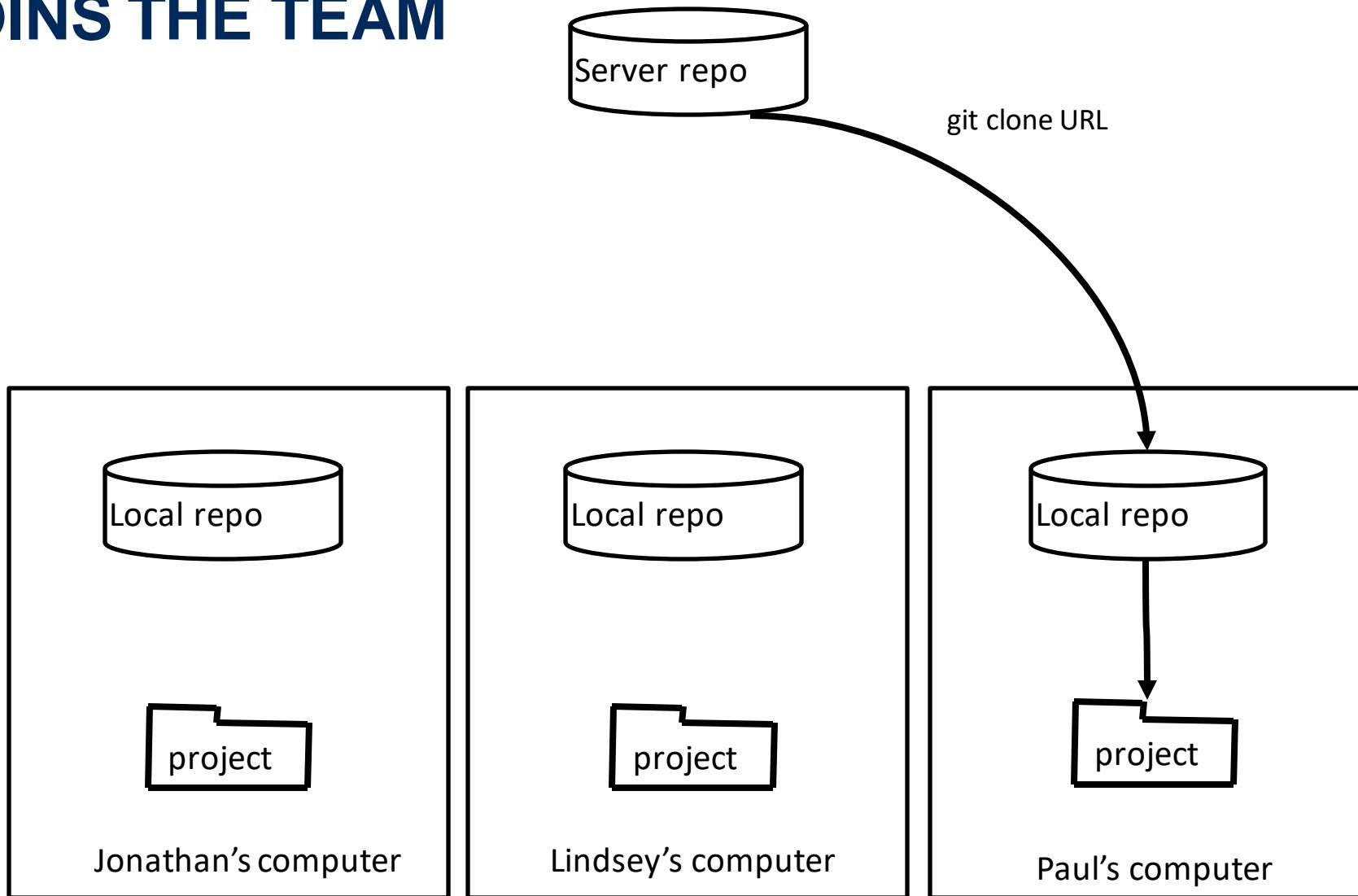
# GIT STATUS

- A file can be in one of 4 states:
  - **untracked** – you have never run a git command on the file
  - **tracked** – committed
  - **staged** – `git add` has been used to add changes in this file to the next commit
  - **dirty/modified** – the file has changes that haven't been staged
- TIP: Use `git status` *regularly*. It helps you make sure the changes you have made really make it into the repo! (IntelliJ uses colours and symbols to help you easily see the status of files)

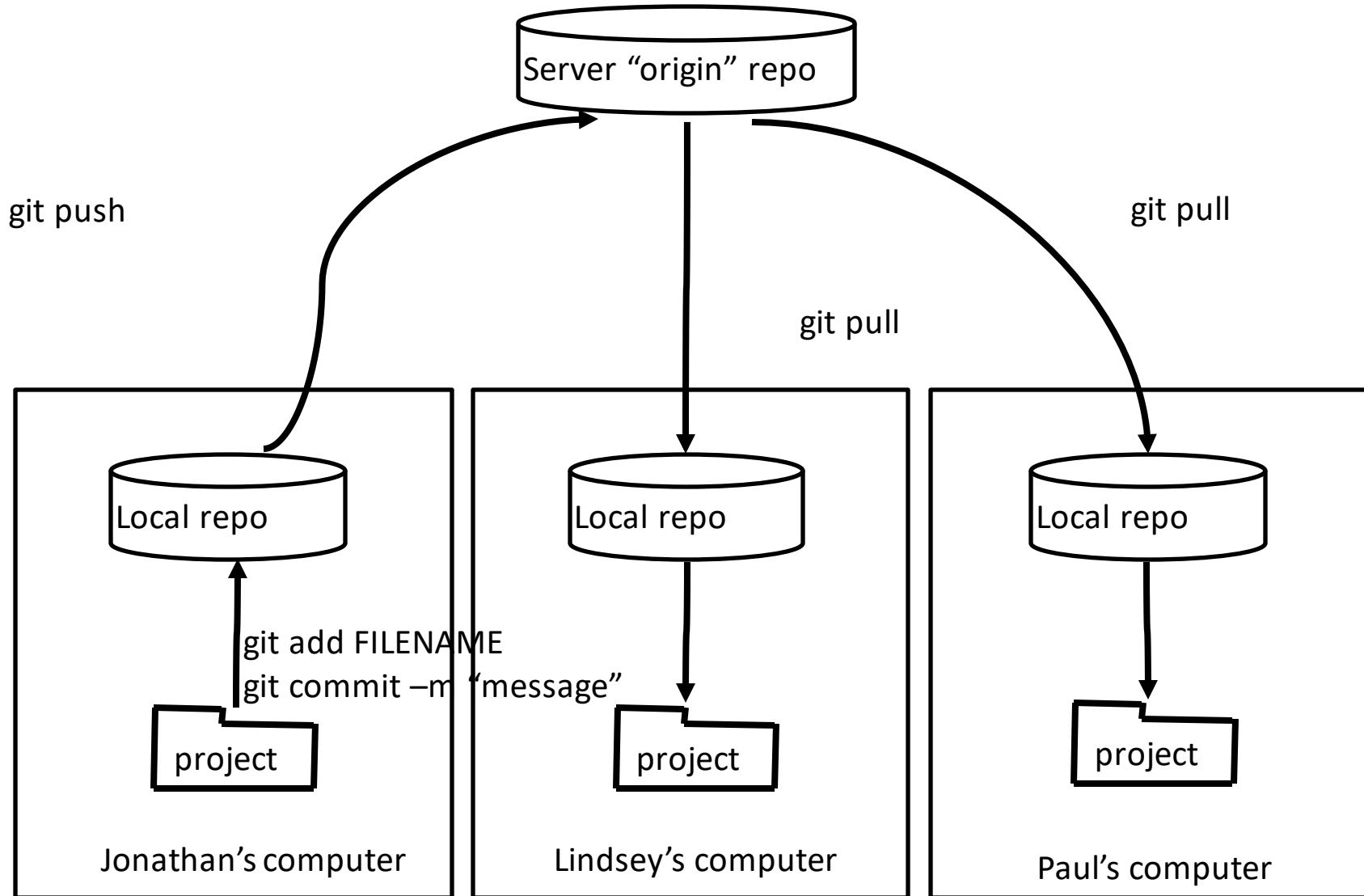
# BASIC WORKFLOW (NO BRANCHING)

- Starting a project:
    - git clone <url>
  - Normal work:
    - After you have made some changes
    - git status (see what has really changed)
    - git add file1 file2 file3
    - git commit -m "meaningful commit message"
    - git push
  - In IntelliJ, you can either type these commands in the Terminal or use the graphical user interface it provides.
-

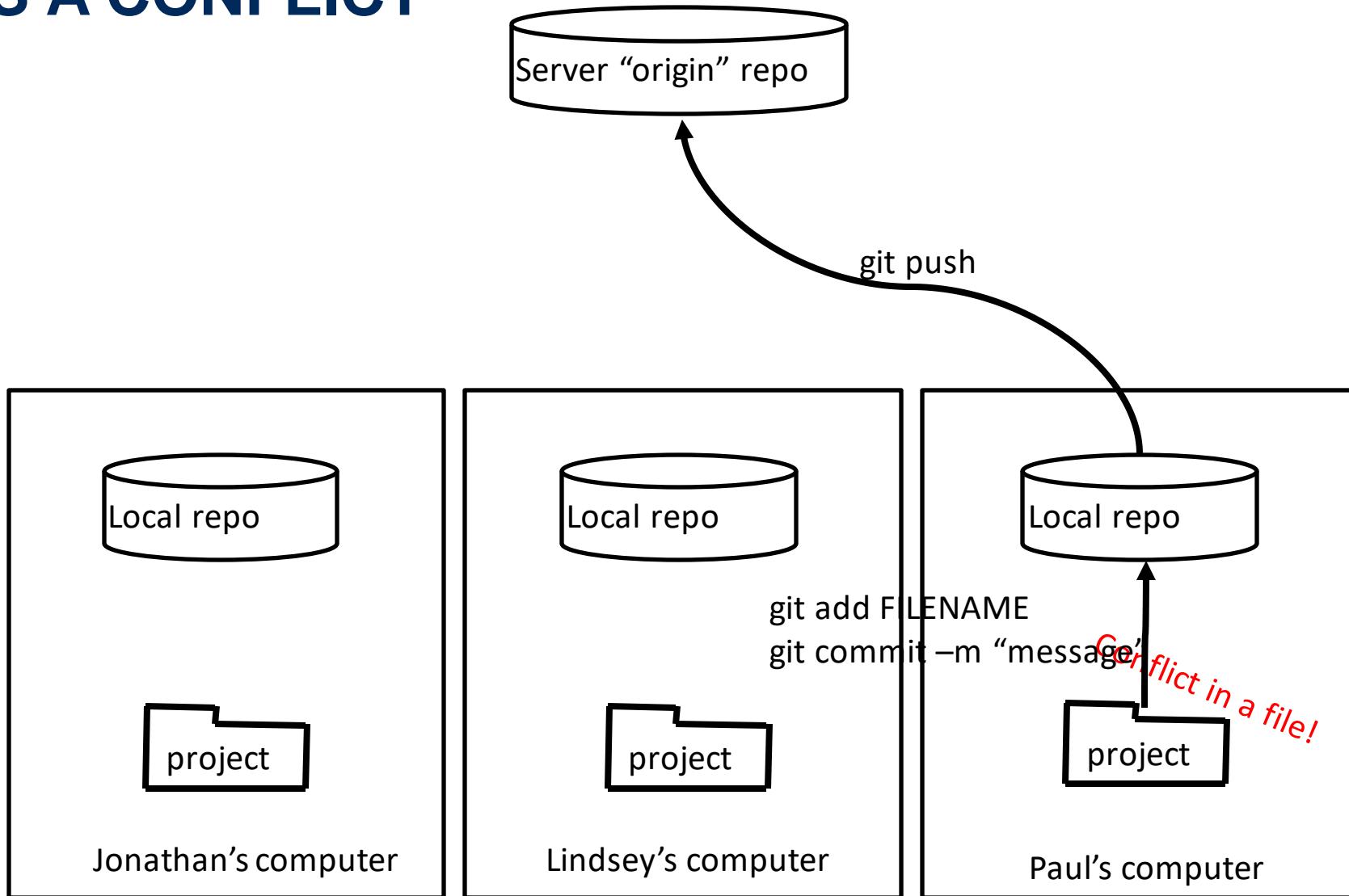
# DISTRIBUTED VERSION CONTROL SYSTEMS: PAUL JOINS THE TEAM



# DISTRIBUTED VERSION CONTROL SYSTEMS: JONATHAN WORKS ON HIS CURRENT FEATURE



# DISTRIBUTED VERSION CONTROL SYSTEMS: PAUL HAS A CONFLICT



# BRANCH AND MERGE WORKFLOW

- To avoid having to constantly resolve conflicts, developers often create a **branch**, where they work on a new feature, then submit a **pull request**.
- One or more members of the team review the pull request, resolve any conflicts (as before), and the branch is merged back in.
- You'll get lots of practice with this in your project this term and this workflow was described in Lab 1!
- If you are interested, you can also read about other workflows here:

<https://www.atlassian.com/git/tutorials/comparing-workflows>

# EXTRA GIT RESOURCES (SUPPLEMENTAL READING)

What is version control and overview of commands:

<https://betterexplained.com/articles/a-visual-guide-to-version-control/>

A quick guide to getting started using git:

<https://towardsdatascience.com/a-quick-primer-to-version-control-using-git-3fbdbb123262>

# HOMEWORK

- Continue working on the Java quizzes and exercises.
- If you are having trouble with any concepts or software, write down your issue and plan to ask about it on piazza, in office hours, or during lab.
- If you feel like you need to try out the branch and merge workflow a few more times to get comfortable with the steps, make a repo on GitHub and go through the steps a few times.





# CLEAN ARCHITECTURE

CSC207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

- Understand clean architecture and its dependency rule

# ARCHITECTURE

(Brief Summary of Chapter 15 from Clean Architecture textbook)

Design of the system

- Dividing it into logical pieces and specifying how those pieces communicate with each other.
- Input and output between layers.

“Goal is to facilitate the **development, deployment, operation, and maintenance** of the software system”

- *“The strategy behind this facilitation is to leave as many options open as possible, for as long as possible.”*

Good architecture strives to maximize programmer productivity!

# POLICY AND LEVEL

Brief Summary of Chapter 19 from Clean Architecture textbook

- “A computer program is a detailed description of the **policy** by which inputs are transformed into outputs.”
- Software design seeks to separate policies and group them as appropriate. (Ideally form a directed acyclic dependency graph between components)
- A policy has an associated **level**. (e.g. “high level policy”)

# POLICY AND LEVEL

- Level: “the distance from the inputs and outputs”
  - higher level policies are farther from the inputs and outputs.
  - lowest level are those managing inputs and outputs.
- In Clean Architecture, **entities** are the highest-level policies (core of the program).

# BUSINESS RULES

(Brief Summary of Chapter 20 from Clean Architecture textbook)

- **Entity:** “An object within our computer system that embodies a small set of Critical Business Rules operating on Critical Business Data.”
- **Use Case:** “A description of the way that an automated system is used. It specifies the input to be provided by the user, the output to be returned to the user, and the processing steps involved in producing that output. A use case describes *application-specific* business rules as opposed to the Critical Business Rules within the Entities.”

# ENTITIES

- Objects that represent critical business data (variables) and critical business rules (methods).
- Some examples
  - a Loan in a bank
  - a Player in a game
  - a set of high scores
  - an item in an inventory system
  - a part in an assembly line

# USE CASE

- A description of the way that an automated system affects Entities
- Use cases manipulate Entities
- Specifies the input to be provided by the user, the output to be returned to the user, and the processing steps involved in producing that output.

Use cases know nothing  
about the user interface or  
the data storage  
mechanism!

## Gather Contact Info for New Loan

**Input:** Name, Address, Birthdate, D.L. #, SSN, etc.  
**Output:** Same info for readback + credit score.

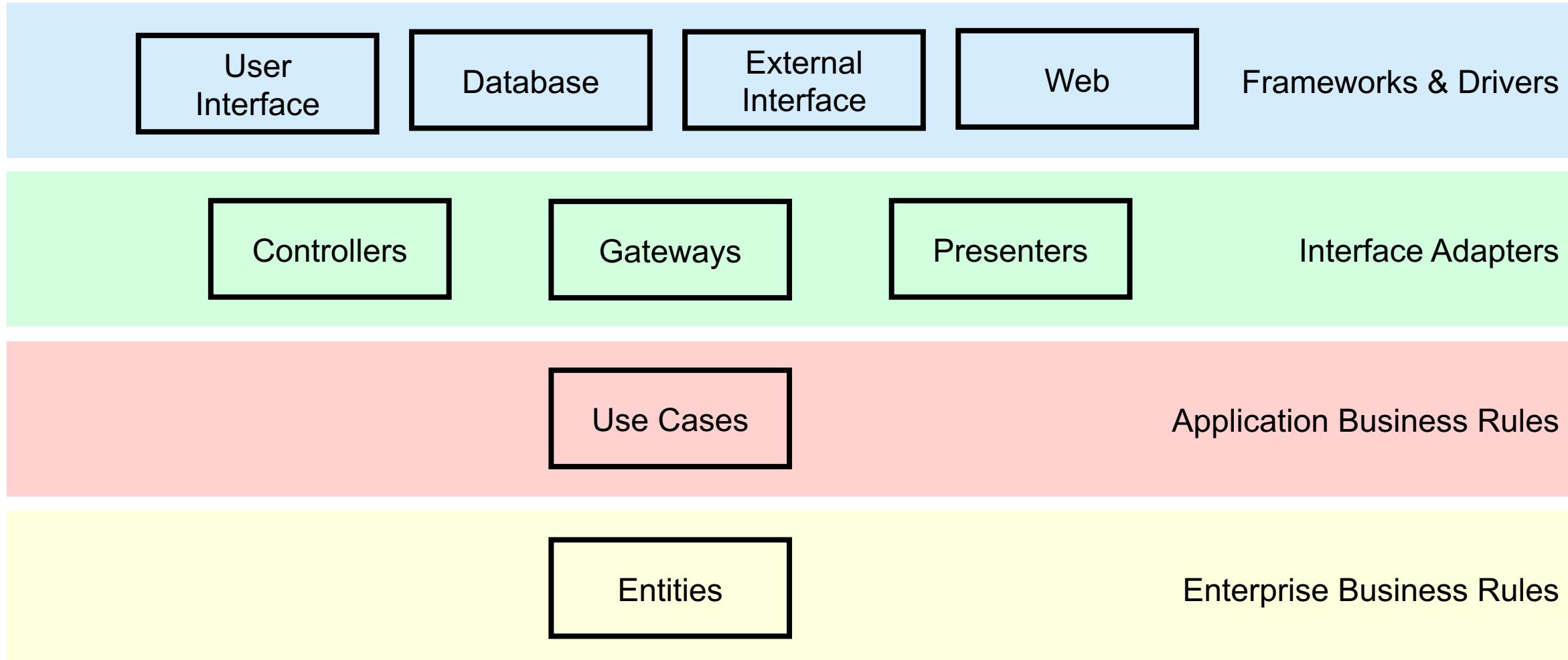
### Primary Course:

1. Accept and validate name.
2. Validate address, birthdate, D.L.#, SSN, etc.
3. Get credit score.
4. If credit score is < 500 activate Denial.
5. Else create Customer and activate Loan Estimation.



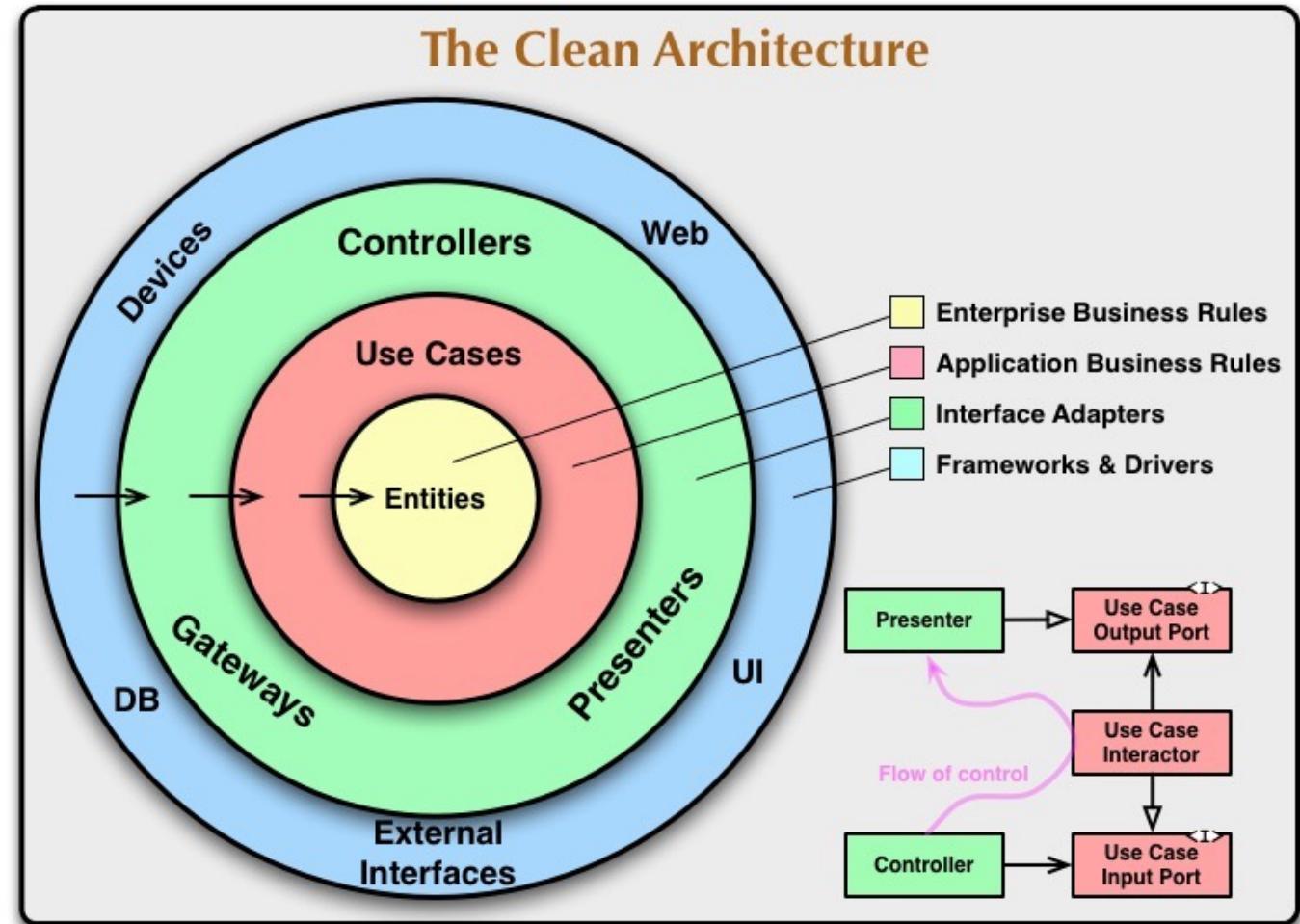


# CLEAN ARCHITECTURE



# CLEAN ARCHITECTURE

- Often visualize the layers as concentric circles.
- Reminds us that Entities are at the core
- Input and output are both in outer layers

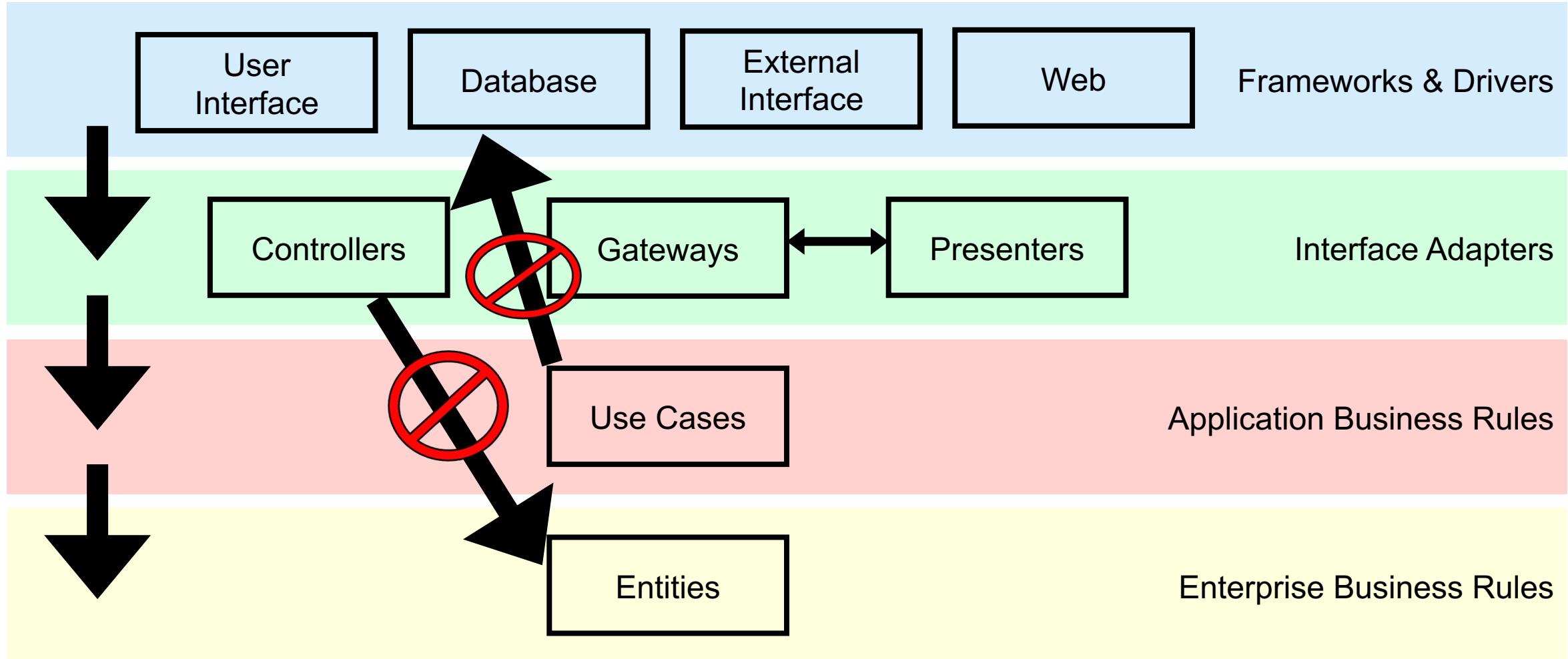




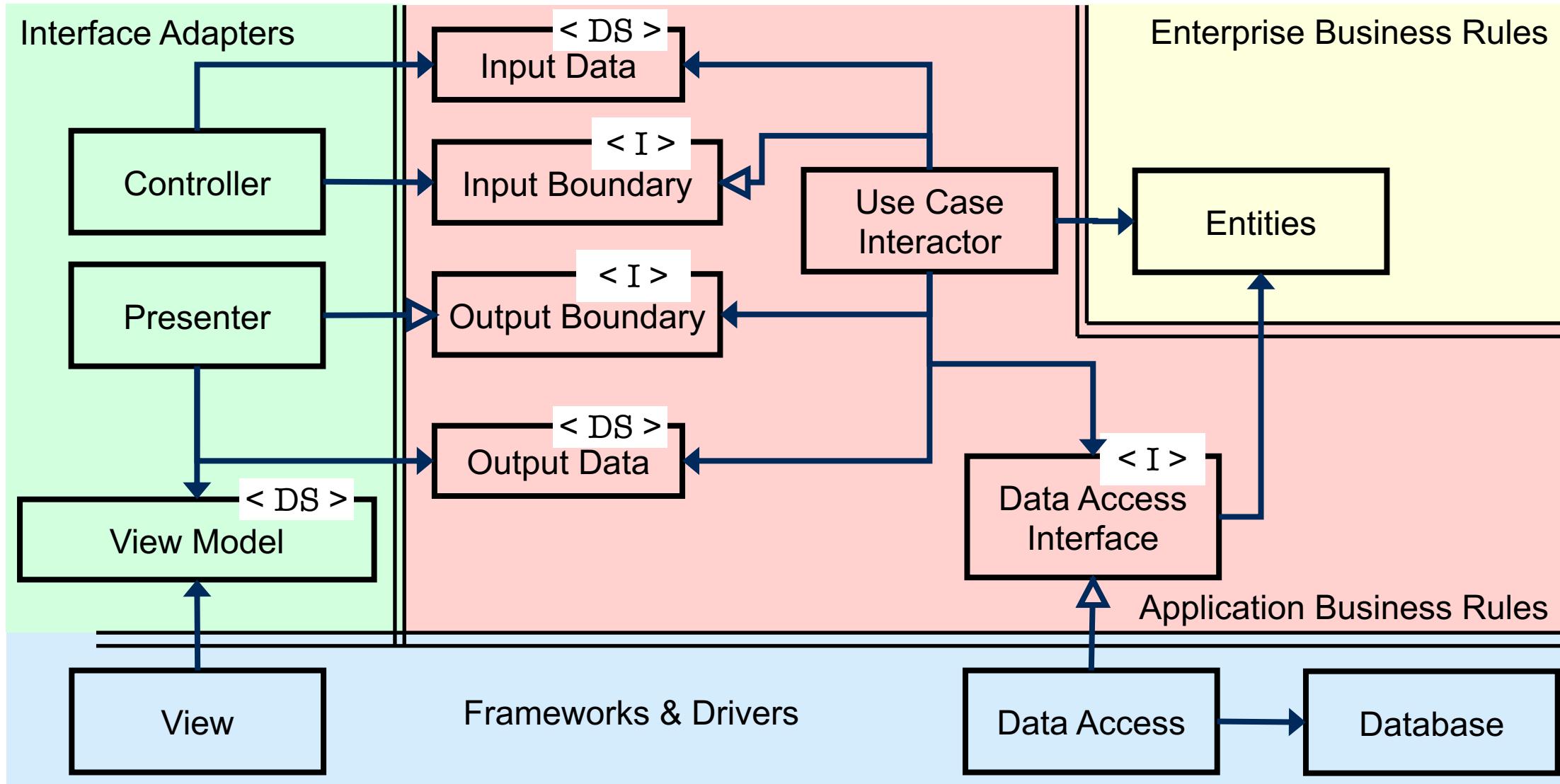
# CLEAN ARCHITECTURE – DEPENDENCY RULE

- All dependencies must point inward.
- Dependence within the same layer is allowed (but try to minimize coupling)
- On occasion you might find it unnecessary to explicitly have all four layers, but you'll almost certainly want at least three layers and sometimes even more than four.
- **The name of something (functions, classes, variables) declared in an outer layer must not be mentioned by the code in an inner layer.**
- How do we go from the inside to the outside then?
  - Dependency Inversion!
  - An inner layer can depend on an interface, which the outer layer implements.
  - We'll talk about dependency inversion in more detail later.

# CLEAN ARCHITECTURE – DEPENDENCY RULE



# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



# IDENTIFYING VIOLATIONS OF CLEAN ARCHITECTURE

- Look at the imports at the top of each source file in your project.
- For example, if you see that you are importing a Controller class from inside an Entity class, that is a violation of clean architecture! Likewise, if you see a Controller referencing an Entity that is also likely a violation.
- You can also achieve this visually by having IntelliJ generate a dependency graph for your project.



# BENEFITS OF CLEAN ARCHITECTURE

- All the “details” (frameworks, UI, Database, etc.) live in the outermost layer.
- The business rules can be easily tested without worrying about those details in the outer layers!
- Any changes in the outer layers don’t affect the business rules!

# USER LOGIN EXAMPLE

- Recall the specification for a user login system we talked about during the first week.
- We had started to develop a design for it in terms of use cases.
- Hopefully, much of what we talked about today felt like it generalized some of the ideas that arose during our previous discussion.
- You can find a partial implementation at  
<https://github.com/paulgries/UserLoginCleanArchitecture>
- We'll refer to this code throughout the term as we learn about new concepts.

# HOMEWORK

---

- Explore the repo from the last slide and take note of how the various classes map onto the layers of Clean Architecture.
- Don't worry if this felt like a lot of new material — throughout the term you will be seeing Clean Architecture a lot in the context of your project and the goal is for you to be comfortable with these concepts by the end of the term.





# CLEAN ARCHITECTURE USER LOGIN EXAMPLE

CSC207 SOFTWARE DESIGN



# LEARNING OUTCOMES

- Understand how the Clean Architecture typical program diagram maps onto Java code by looking at a concrete example.

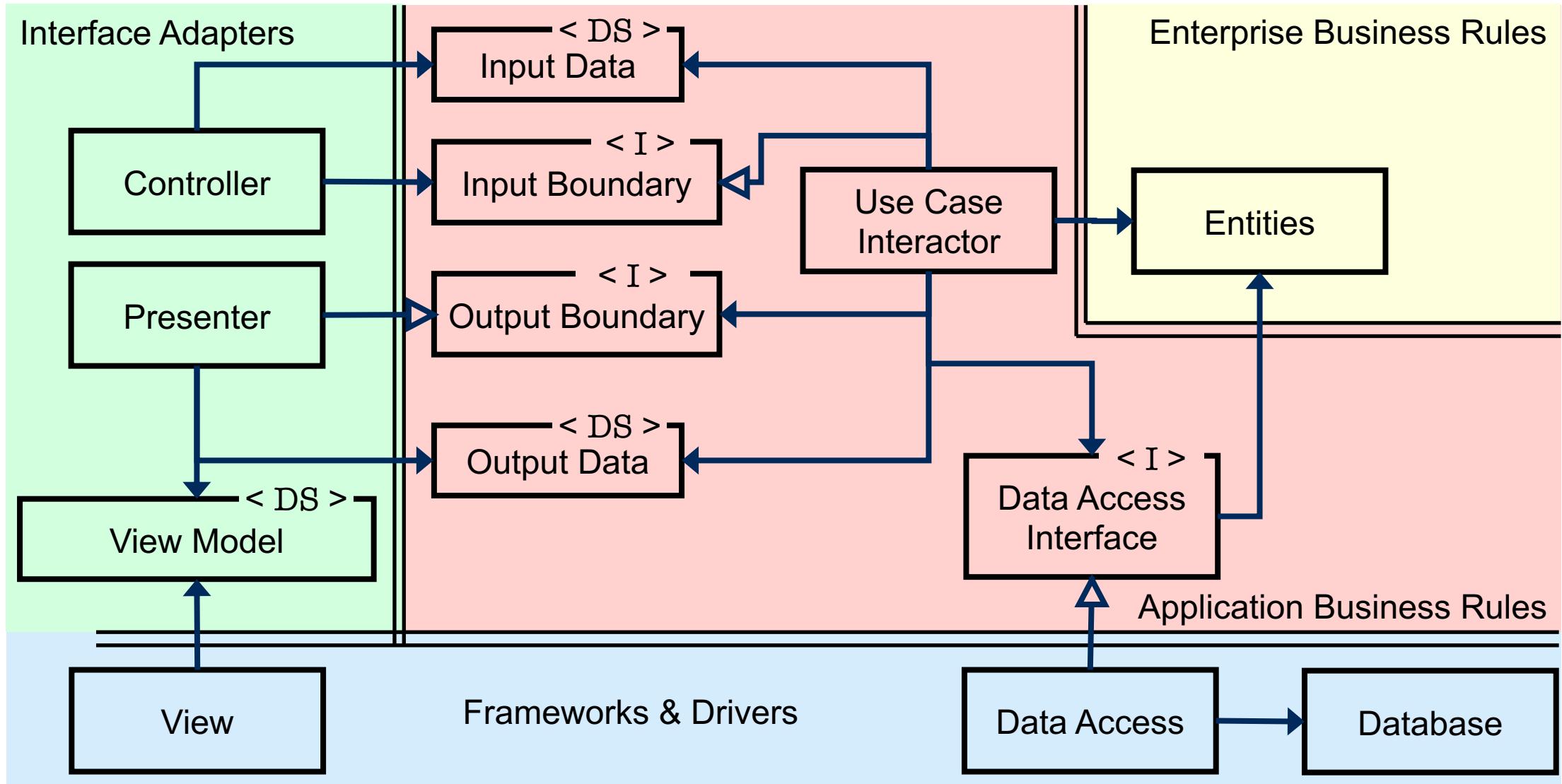
# RECALL: USER REGISTERS NEW ACCOUNT USE CASE

- The user chooses a username
- The user chooses a password and enters it twice (to help them remember)
- If the username already exists, the system alerts the user
- If the two passwords don't match, the system alerts the user
- If the username doesn't exist in the system and the passwords match, then the system creates the user but does not log them in

Link to the repo:

<https://github.com/paulgries/UserLoginCleanArchitecture>

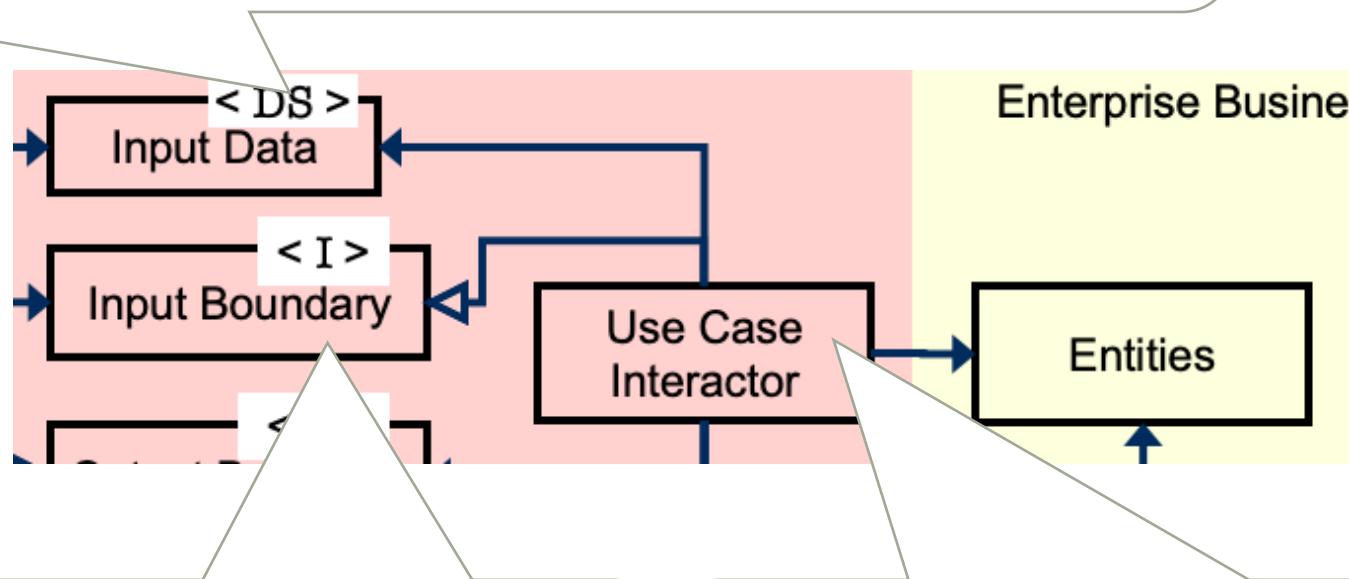
# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE

What info does the Interactor need?

- A class with basic instance variables: the raw data (Strings and numbers) from the user interface.
- Transient: it's created, passed in, unpacked by the use case, and discarded.



How will the Controller start the use case?

- It's an interface with a method.
- One of the parameters is the Input Data.

- Implements the Input Boundary interface
- Its primary method is specified by that interface
- Unpacks the Input Data and manipulates Entities

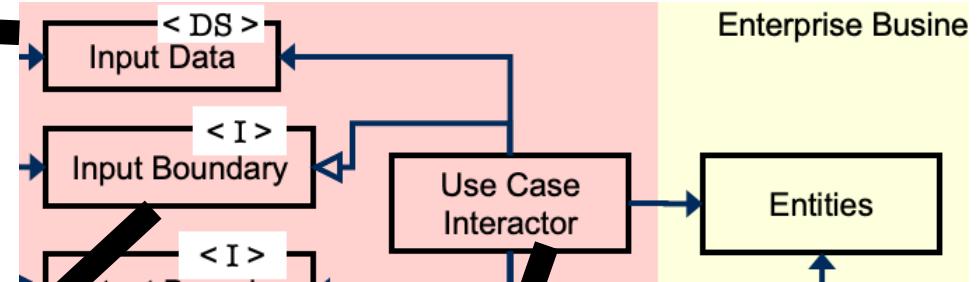


# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE

```
public class UserRequestModel {  
    private String name, password, repeatPassword;  
  
    public UserRequestModel(  
        String name, String password, String repeatPassword) {  
        this.name = name;  
        this.password = password;  
        this.repeatPassword = repeatPassword;  
    }  
    ...  
}
```

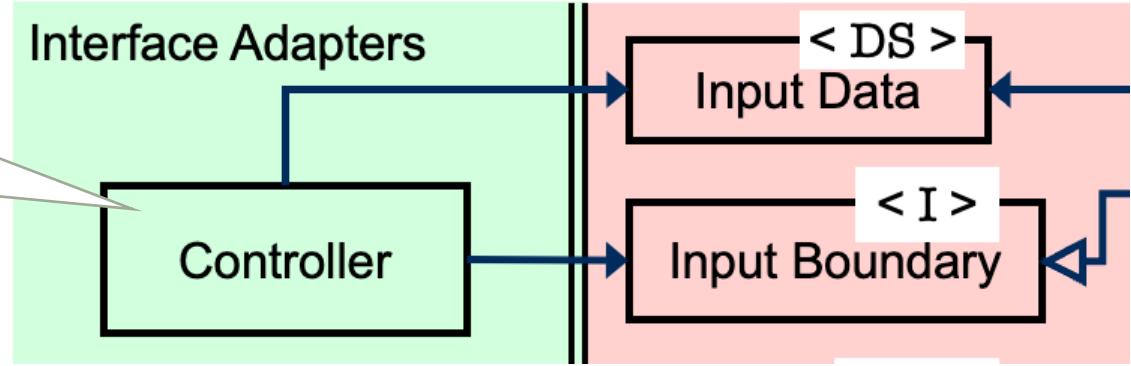
```
    public interface UserInputBoundary {  
        UserResponseModel create(UserRequestModel  
requestModel);  
    }
```

```
    public class UserRegisterInteractor implements UserInputBoundary {  
        public UserResponseModel create(UserRequestModel requestModel) {  
            ...  
        }  
    }
```



# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE

Knows that it is using an Input Boundary,  
but not which Interactor class.



```
public class UserRegisterController {  
  
    private final UserInputBoundary userInput;  
  
    public UserRegisterController(UserInputBoundary accountGateway) {  
        this.userInput = accountGateway;  
    }  
  
    UserResponseModel create(UserRequestModel requestModel) {  
        return userInput.create(requestModel);  
    }  
}
```

Called by the UI.

# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE

```
UserRequestModel model = new UserRequestModel(  
    username.getText(),  
    String.valueOf(password.getPassword()),  
    String.valueOf(repeatPassword.getPassword()));
```

```
try {  
    userRegisterController.create(model);  
    JOptionPane.showMessageDialog(this, "%s created.".formatted(username.getText()));  
} catch (Exception e) {  
    JOptionPane.showMessageDialog(this, e.getMessage());  
}
```

In the user interface, in reaction to a button click

- `username`, `password`, and `repeatPassword` are fields of the UI

```
UserInputBoundary interactor = new UserRegisterInteractor(user, presenter, userFactory);  
UserRegisterController userRegisterController = new UserRegisterController(interactor);  
RegisterScreen registerScreen = new RegisterScreen(userRegisterController);
```

In method `main`

# CRC CARDS

CSC 207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

- Know what CRC cards are and how they can be used to design programs.
- Be able to construct CRC cards for a simple specification.

# CRC CARDS

- Part of the Object-Oriented development paradigm.
- Highly interactive and human-intensive.
- Result: initial set of classes and their relationships.
- *What* rather than *How*.

Benefits:

- Cheap and quick: all you need is a set of index cards.
- Simple, easy methodology.
- Forces you to be concise and clear.
- Allows for input from every team member.

# WHAT IS A CRC CARD?

CRC stands for **C**lass, **R**esponsibility and **C**ollaboration.

## Class

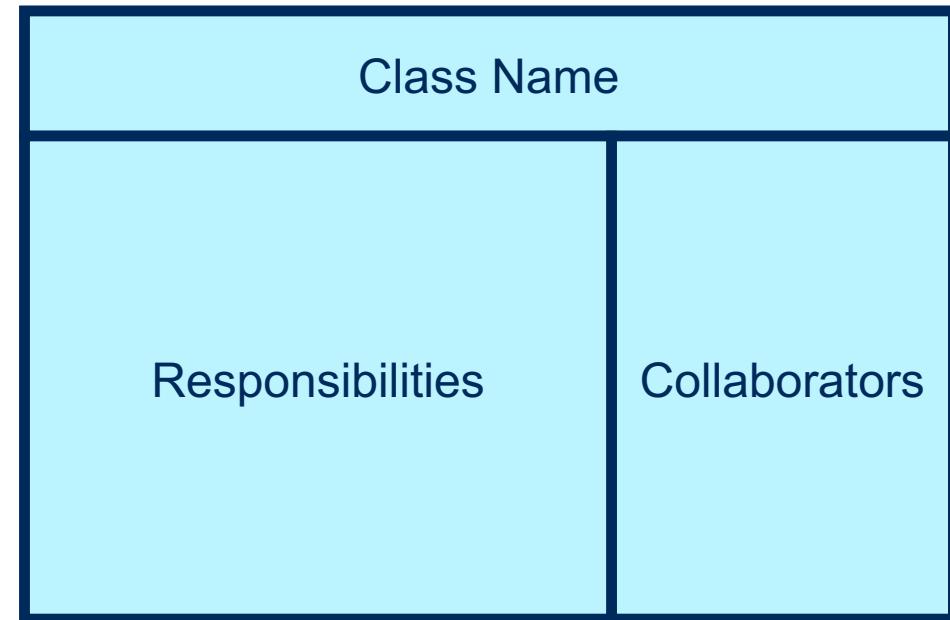
- An object-oriented class name
- Include information about super- and sub-classes

## Responsibility

- What information this class stores
- What this class does
- The behaviour for which an object is accountable

## Collaboration

- Relationship to other classes
- Which other classes this class uses



# CRC MODEL

- A collection of CRC cards specifying the Object-Oriented Design of a software system.

## How do we create a CRC Model?

- Typically, you are given a specification (a written description of the requirements) for the software system.
- You work in a team.
- Ideally, you all gather around a table.
- You need a set of index cards and something to write with.
- Coffee / other beverages and snacks are optional.



# HOW TO CREATE A CRC MODEL

- Read the specification. Again. And again.
- Identify core **classes** (simplistic advice: look for nouns).
- Create a card per class (begin with class names only).
- Add **responsibilities** (simplistic advice: look for verbs).
- Identify other classes that this class needs to talk to in order to fulfil its responsibilities
  - These are its **collaborators**.
- Add more classes as you discover them.
- Move classes to the side if they become unnecessary. (But don't tear them up yet!)
- Refine by identifying abstract classes, inheritance, etc.
- Keep adding/refining until everyone on the team is sufficiently satisfied.



# HOW CAN WE TELL IF OUR CRC MODEL WORKS?

- A neat technique: a **Scenario Walk-through**.
- Select a scenario and choose a plausible set of inputs for it.
- Manually “execute” the scenario
  - Start with the initial input for the scenario and find a class that has responsibility for responding to that input.
  - Trace through the collaborations of each class that participates in satisfying that responsibility.
  - Adjust, as necessary.
  - Repeat until the scenario has “stabilized” (that is, no further adjustments are necessary).

# EXAMPLE: RESTAURANT REVIEW SYSTEM

Consider this description of a software system that you are developing to facilitate restaurant reviews

*Each restaurant corresponds to a certain price range, neighbourhood, and cuisines it serves. Restaurants that serve alcohol must have a license, which they need to renew every year. The system should also report how long, on average, customers wait for take out in restaurants that offer take-out service.*

*When reviewers leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An owner of a restaurant can respond to a review with a comment. All users of the system log in with their username. Users can choose to be contacted by email ...*

# RECALL OUR KEY STEPS

A key part of developing the model involves careful analysis of the problem specification. We must:

## **Identify important nouns.**

- Underline nouns that may make sensible classes or that describe information a class could be responsible for storing.

## **Choose potential classes.**

- From the nouns identified, write down the ones that are potential classes.

## **Identify verbs that describe responsibilities.**

- In the problem description, circle verbs that describe tasks that a class may be responsible for doing.

# IDENTIFYING IMPORTANT NOUNS

*Each restaurant corresponds to a certain price range, neighbourhood, and cuisines it serves. Restaurants that serve alcohol must have a license, which they need to renew every year. The system should also report how long, on average, customers wait for take out in restaurants that offer take-out service.*

*When reviewers leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An owner of a restaurant can respond to a review with a comment. All users of the system log in with their username. Users can choose to be contacted by email ...*

# CHOOSE POTENTIAL CLASSES

Let's try to narrow down to the most important nouns and start with those as potential classes to make cards for to get started.

*Each restaurant corresponds to a certain price range, neighbourhood, and cuisines it serves. Restaurants that serve alcohol must have a license, which they need to renew every year. The system should also report how long, on average, customers wait for take out in restaurants that offer take-out service.*

*When reviewers leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An owner of a restaurant can respond to a review with a comment. All users of the system log in with their username. Users can choose to be contacted by email ...*

<b>Restaurant</b>	
Responsibilities	Collaborators

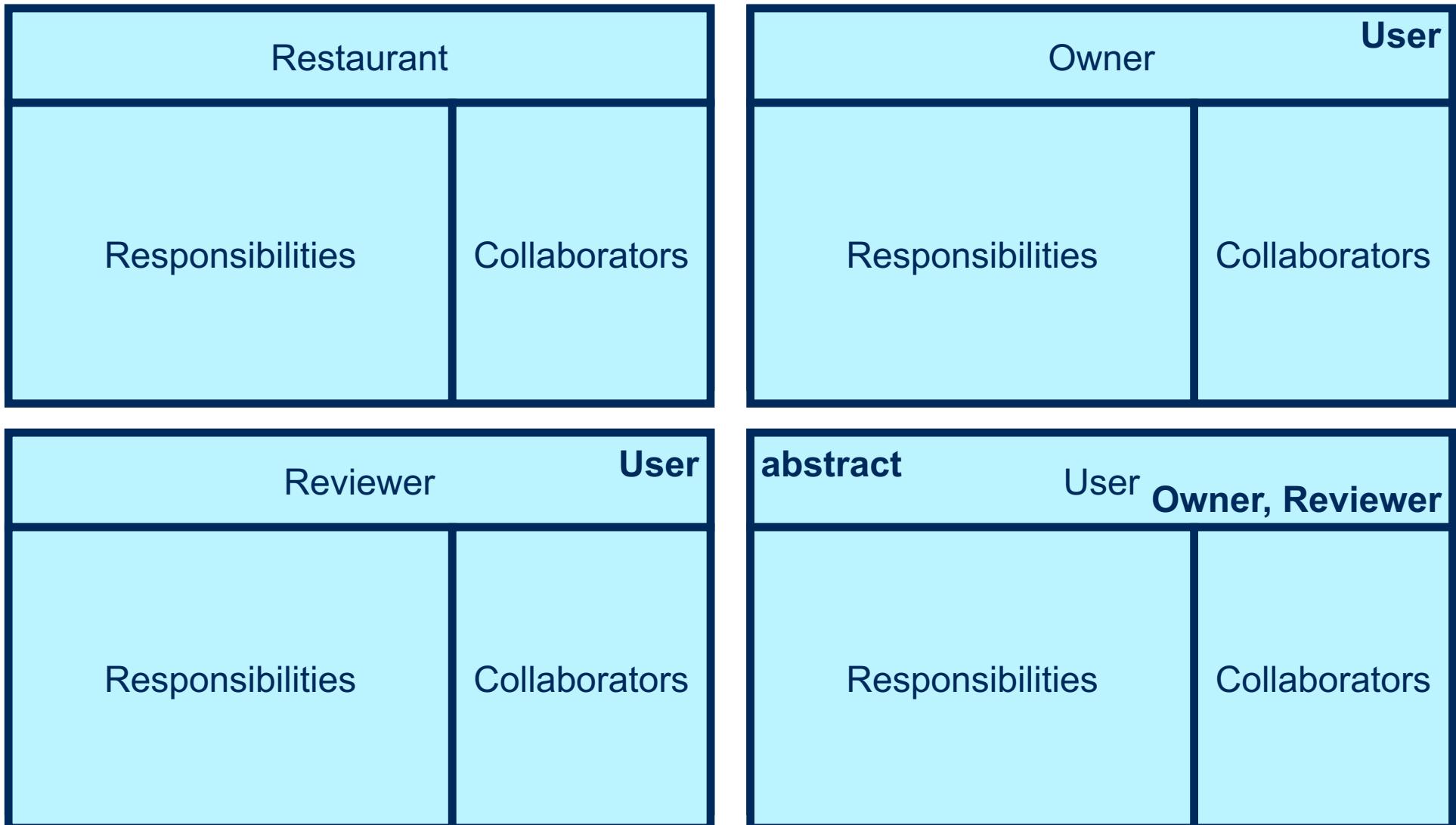
<b>Owner</b>	
Responsibilities	Collaborators

<b>Reviewer</b>	
Responsibilities	Collaborators

<b>User</b>	
Responsibilities	Collaborators



- We can continue by identifying inheritance relationships – placing the parent class in the upper right corner, any child classes in the lower right corner, and indicating if a class is abstract in the upper left corner.



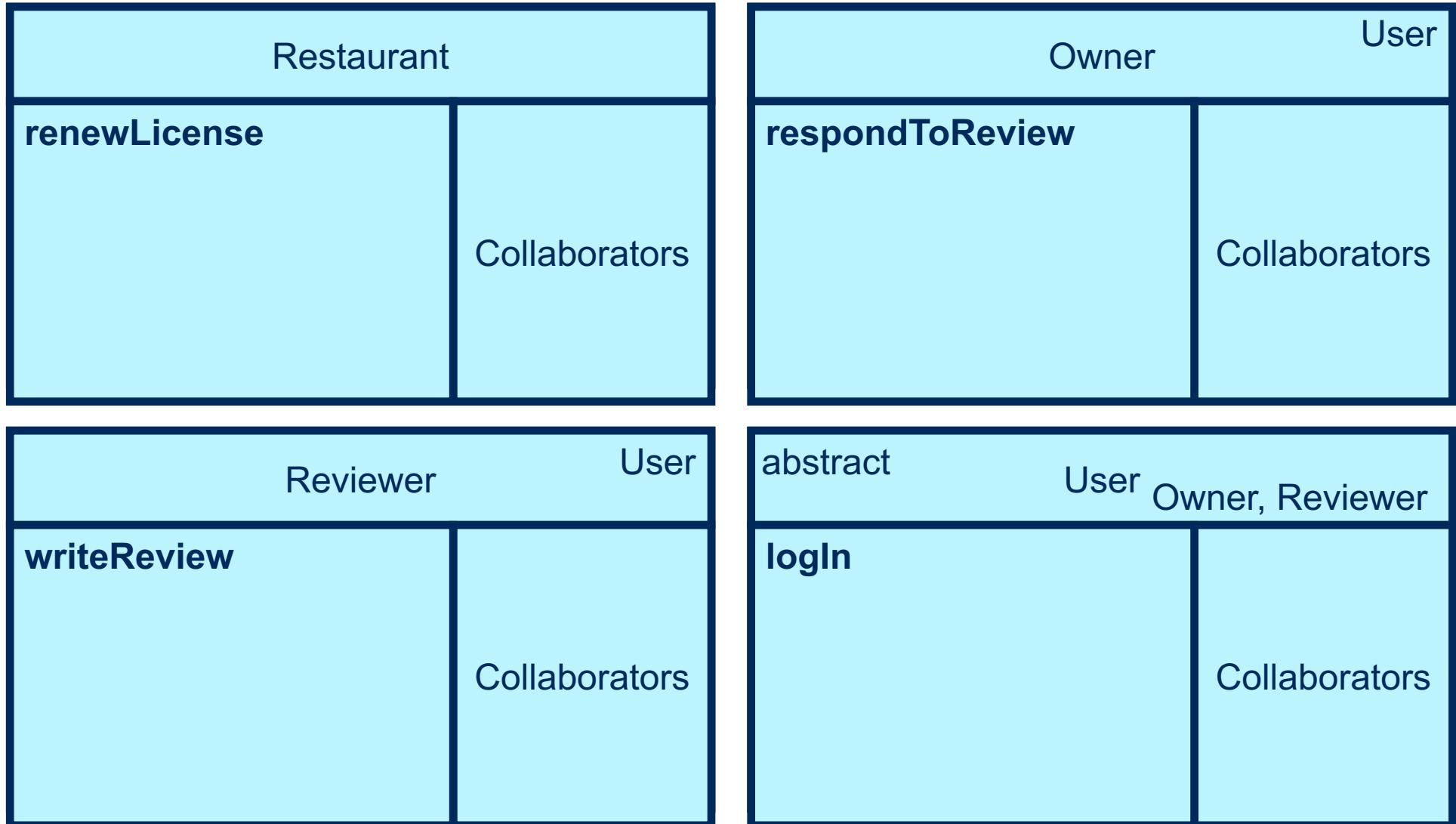
# IDENTIFY VERB PHRASES DESCRIBING RESPONSIBILITIES

Also keep in mind what class is responsible for doing each of these verb phrases.

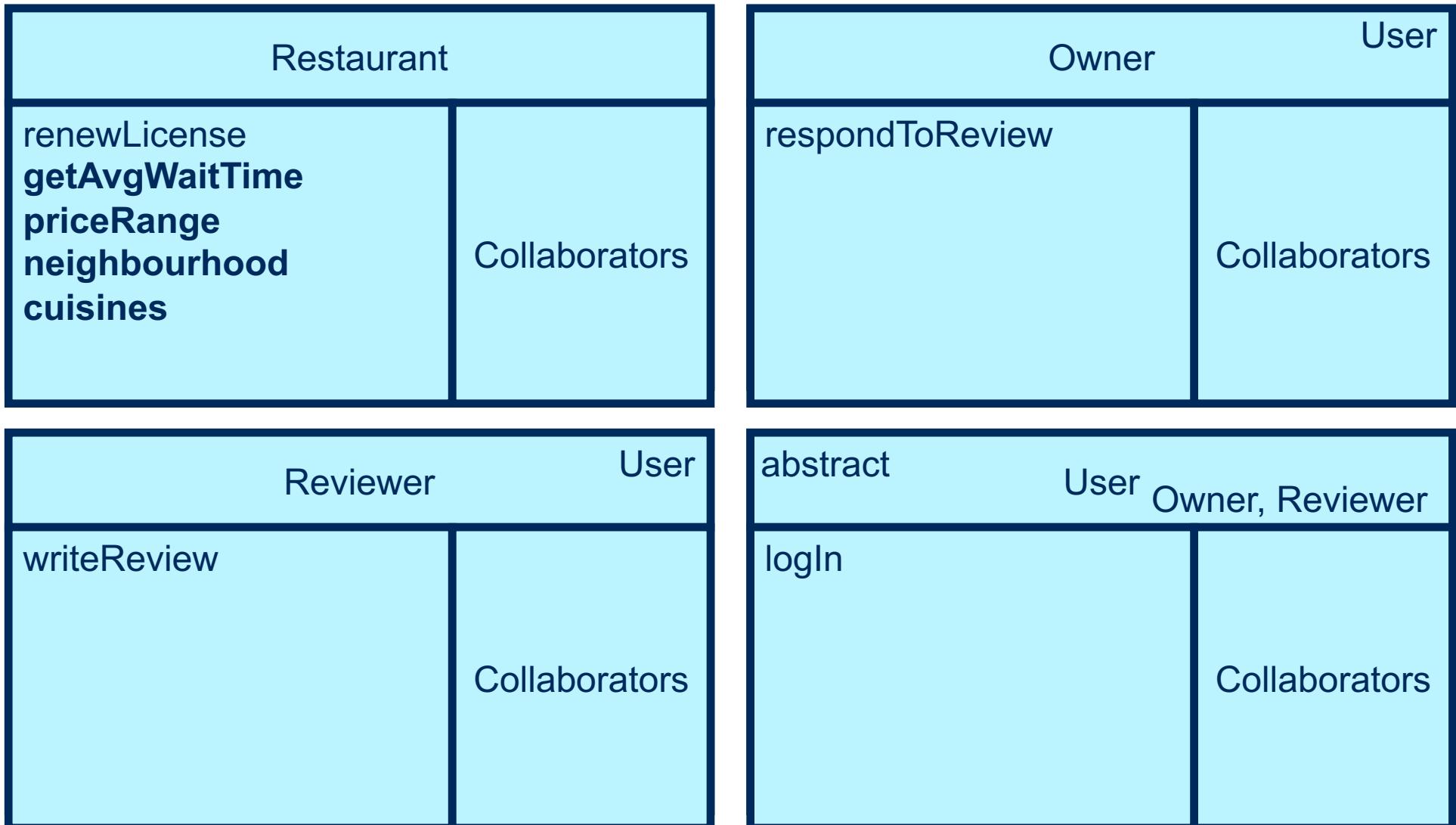
*Each **restaurant** corresponds to a certain price range, neighbourhood, and cuisines it serves. Restaurants that serve alcohol must have a license, which they need to renew every year. The system should also **report how long, on average**, customers wait for take out in restaurants that offer take-out service.*

*When reviewers leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An owner of a restaurant can **respond to a review** with a comment. All users of the system **log in** with their username. Users can choose to be contacted by email ...*

- We can add some of these responsibilities to our cards.
- We'll just write a function name, but you'll want to find a balance between being concise and being overly wordy.
- Make sure you write enough so the responsibility is clear to your whole team.



- We can also add some “what they store” responsibilities for Restaurant.



# LIQUOR LICENSE

- What about the responsibility of storing licenses? Not all restaurants have licenses!
- Solution: We could introduce a new type of Restaurant and move the renewLicense responsibility to this subclass.
- Note that we don't repeat responsibilities that are handled by Restaurant on the CRC card for its subclass.

Restaurant	LicensedRestaurant
<code>renewLicense</code> <code>getAvgWaitTime</code> <code>priceRange</code> <code>neighbourhood</code> <code>cuisines</code>	Collaborators

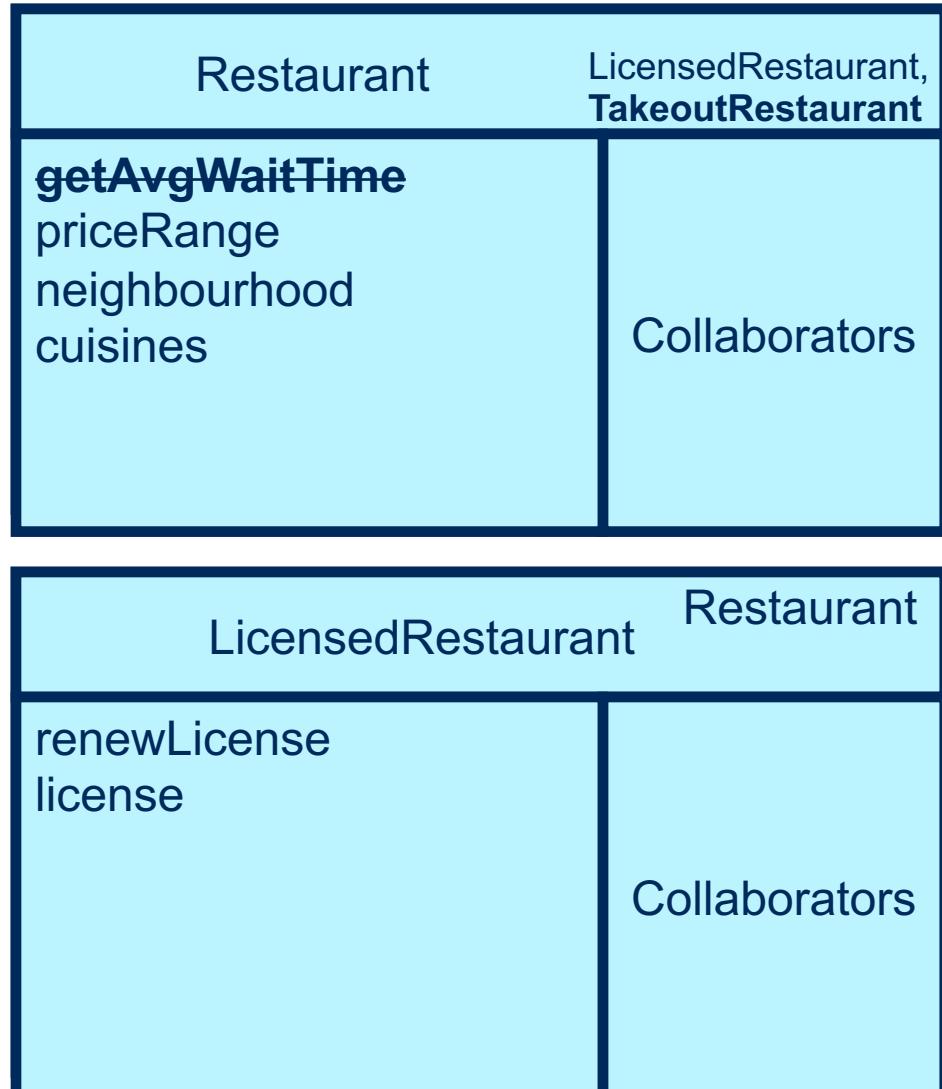
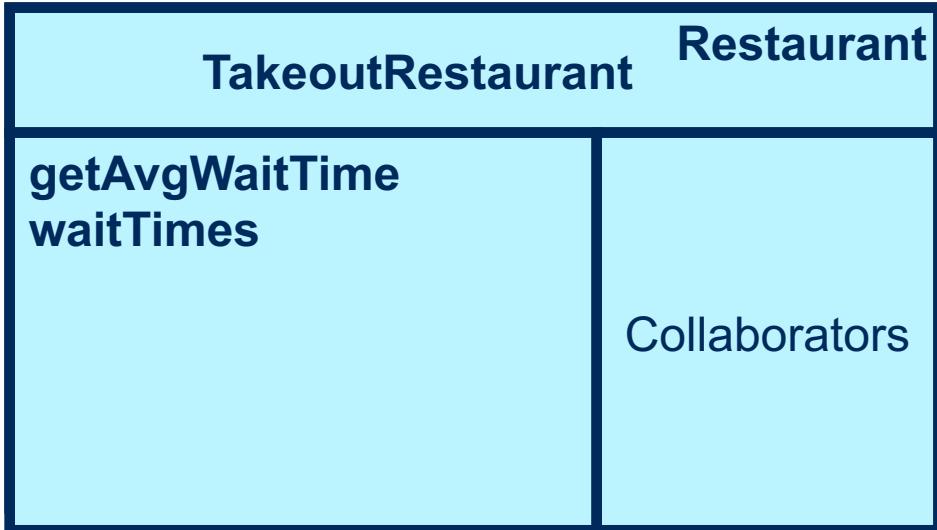
  

LicensedRestaurant	Restaurant
<code>renewLicense</code> <code>license</code>	Collaborators



# TAKEOUT

- What about the responsibility of tracking wait times? Not all restaurants offer takeout!
- Solution: We could introduce another subclass of Restaurant.



# TAKEOUT AND LIQUOR LICENSE?

- But what if a restaurant has a license **and** offers takeout?
- Solution: Introduce an interface.
- As with inheritance, we can use the corners to indicate interfaces and classes that implement them.

<b>TakeoutRestaurant</b>	<b>Takeout, Restaurant</b>
getAvgWaitTime <b>waitTimes</b>	Collaborators
<b>Restaurant</b>	<b>LicensedRestaurant, TakeoutRestaurant</b>
priceRange neighbourhood cuisines	Collaborators
<b>interface</b>	
<b>Takeout</b>	<b>TakeoutRestaurant</b>
getAvgWaitTime	Collaborators
<b>LicensedRestaurant</b>	<b>Restaurant</b>
renewLicense license	Collaborators



# MORE ABOUT REVIEWS

Let's look more closely at where it talks about reviews in our specification.

*When reviewers leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An owner of a restaurant can respond to a review with a comment. All users of the system log in with their username. Users can choose to be contacted by email ...*

# REVIEW CLASS

- Let's add a review class to our model.
- When a Reviewer writes a review, that will somehow involve the Restaurant and Review classes, so they are added as collaborators.



# HOW DO REVIEWS WORK?

We have some design decisions to make here:

- Does a Review know which Restaurant it is for?
  - i.e. Is a Review responsible for storing which Restaurant it is for?
- Does a Review know who wrote it?
  - i.e. Is a Review responsible for storing which Reviewer wrote it?
- Where do Reviews live? With a Restaurant? With a Reviewer? Somewhere else?

# REVIEW CLASS – ONE POSSIBLE DESIGN

- Here's one way we might design this, with Restaurants storing reviews and each Review storing who wrote it.
- As a team, you would want to discuss pros and cons of potential designs.



# SCENARIO WALK-THROUGH: WRITE A REVIEW

Let's see if this works...

To write a review, a Reviewer needs to:

- create a Review,
- provide it to the Restaurant, and
- the Restaurant needs to store it

Our current model isn't specifying how this last responsibility is being handled.

- We can fix this by adding an “addReview” responsibility to the Restaurant CRC card.



# USER STORIES AND USE CASES

- So far, our CRC model appears to consist of Entities for our system.
- It would be great to add some cards for use case classes too.
- Depending on how we state our scenario walkthrough, we can get at different aspects of the program. The previous one was still framed around our Entities, but we could frame it around the user of the program.
- User Story
  - "A user story is an informal, general explanation of a software feature written from the perspective of the end user or customer."
    - <https://www.atlassian.com/agile/project-management/user-stories>
  - "Write a review" might become something like "A customer writes a review for a given restaurant and submits it to the system to record"
  - This should naturally introduce a use case, something like a "ReviewRecorderInteractor"

# ADDING OUR USE CASE TO OUR CRC MODEL

recordReview will be the API for our use case. What inputs does it require from the user? What should it return or do upon completion?

Tentatively, this design has our new use case class storing restaurants and reviewers, but is that the best place?

ReviewRecorderInteractor		
restaurants	Restaurant	
reviewers	Reviewer	
recordReview	Review	

Restaurant	
priceRange neighbourhood cuisines reviews addReview	Review

Reviewer	User
writeReview	Restaurant Review

Review	
thumbsUp Comment reviewer	Reviewer



# SCENARIO WALK-THROUGH: WRITE A REVIEW (USE CASE VERSION)

Let's see if this works...

To write a review:

- The user of the program enters the required information for the review, corresponding to the API of our use case class. (Note: we'll need a CRC card for a controller class too! What are its class responsibilities and collaborators?)
- The use case class:
  - constructs a Review object using the information provided,
  - finds the reviewer in the collection of reviewers and calls that reviewer's writeReview method,
  - finds the restaurant in the collection of restaurants and calls that restaurant's addReview method, and
  - reports the success of recording the review (Note: this sounds like we need a presenter class too!)



# ADDITIONAL DETAILS

Recall that we defined additional infrastructure around the use case classes in Clean Architecture. I.e. the InputBoundary and OutputBoundary interfaces + their associated Input and Output Data.

These can of course also be added to our model to further specify our design.

You'll likely find that if you don't take the time to really critique and refine your CRC model, then you'll end up needing to add more classes and interfaces as you attempt to implement your design.

# SCENARIO WALK-THROUGH: RESPOND TO A REVIEW

For extra practice, try completing a walk-through for when “an owner responds to a review”. (start with a simple walk-through, then try a “user story” walk-through — adding CRC cards as needed)

If any responsibilities are missing, add them to the cards so that you are convinced your model works.

Refer to the specification and keep adding functionality to the model if you need more practice.

# OUR CRC MODEL SO FAR (WITHOUT USE CASES)

- Other classes omitted for space



# EXTRA PRACTICE

The following are three short specifications taken from the CSC148 course notes on object-oriented programming and designing classes. You can try making CRC cards for each of them if you feel you need more practice.

## *People*

- We'd like to create a simple model of a person. A person normally can be identified by their name, but might commonly be asked about her age (in years). We want to be able to keep track of a person's mood throughout the day: happy, sad, tired, etc. Every person also has a favourite food: when she eats that food, her mood becomes 'ecstatic'. And though people are capable of almost anything, we'll only model a few other actions that people can take: changing their name and greeting another person with the phrase 'Hi \_\_\_\_\_, it's nice to meet you! I'm \_\_\_\_\_'.

## *Rational numbers*

- A rational number consists of a numerator and denominator; the denominator cannot be 0. Rational numbers are written like  $7/8$ . Typical operations include determining whether the rational is positive, adding two rationals, multiplying two rationals, comparing two rationals, and converting a rational to a string.

## *Restaurant recommendation*

- We want to build an app which makes restaurant recommendations for a group of friends going out for a meal. Each person has a name, current location, dietary restrictions, and some ratings and comments for existing restaurants. Each restaurant has a name, a menu from which one can determine what dishes accommodate what dietary restrictions, and a location. The recommendation system, in addition to making recommendations, should be able to report statistics like the number of times a certain person has used the system, the number of times it has recommended each restaurant, and the last recommendation made for a given group of people.

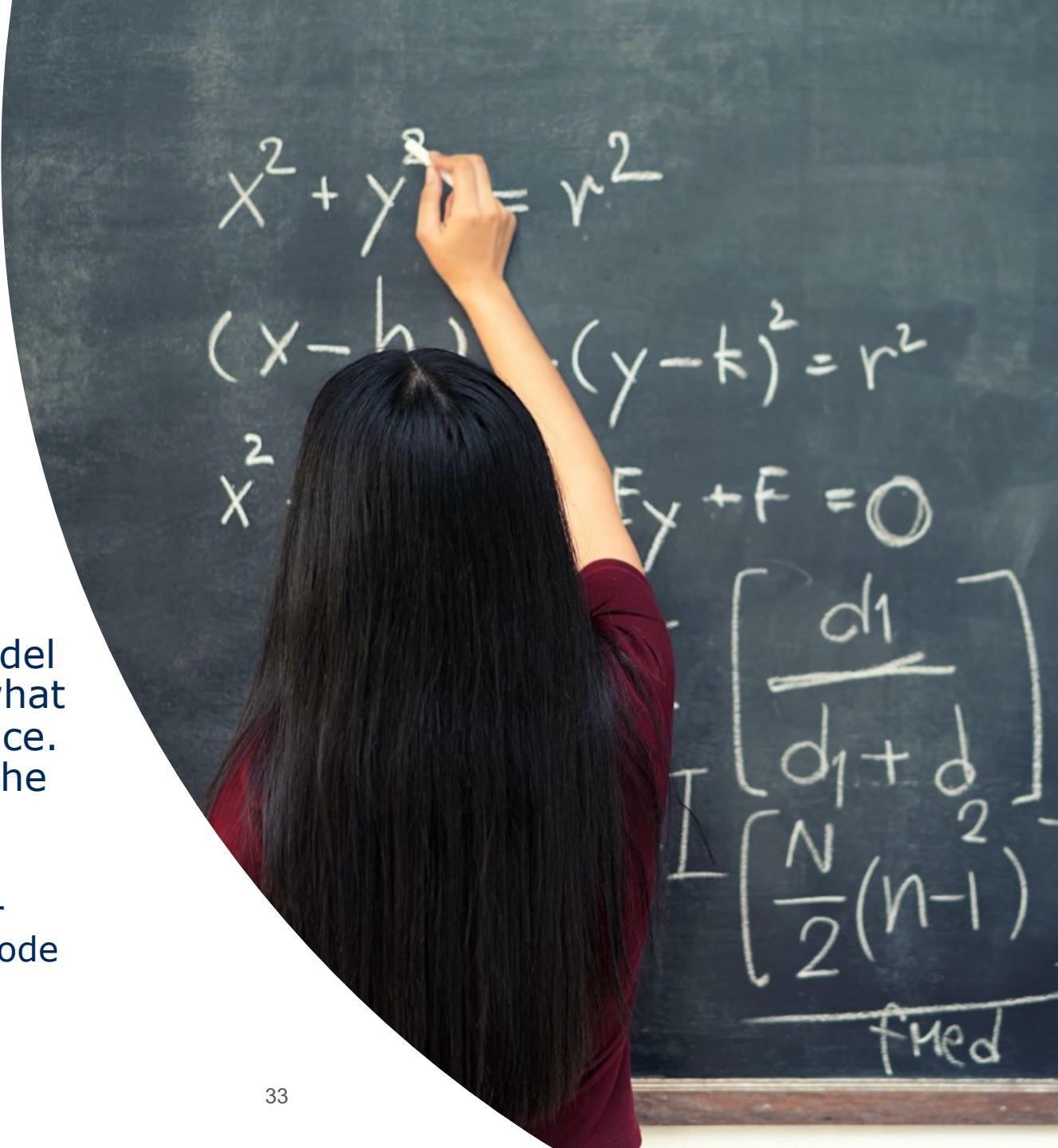
# DESIGN PRINCIPLES AND CLEAN ARCHITECTURE

CRC gives us a way to brainstorm and design our system, but how do we know if our design is any good?

- As we saw, we can try some scenario walkthroughs to convince ourselves it will at least work, and we can design with **Clean Architecture** in mind.
- We could try implementing it and see if it results in working code.
- We might ask: “How easy is the code to understand, maintain, and extend?”
- We can attempt to judge it based on established **design principles** (SOLID).

# HOMEWORK

- For extra practice, try developing a CRC model for the User Login system we've looked at previously or continue with the specification we considered here.
- Eventually, one must go from the CRC model to code. You can try translating some of what we did here into Java code for extra practice. This will also help you spot places where the CRC model may still be leaving out information crucial to implementing it!
  - This could also be a great exercise for your project team before you dive into writing code for your project!



# SOLID DESIGN PRINCIPLES

CSC207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

- Know what the five SOLID design principles are.

# FUNDAMENTAL OOD PRINCIPLES

SOLID: five basic principles of object-oriented design

(Developed by [Robert C. Martin](#), affectionately known as “Uncle Bob”.)

Single responsibility principle (SRP)

Open/closed principle (OCP)

Liskov substitution principle (LSP)

Interface segregation principle (ISP)

Dependency inversion principle (DIP)

# SINGLE RESPONSIBILITY PRINCIPLE

- Every class should have a single responsibility.
- Another way to view this is that **a class should only have one reason to change.**
- But who causes the change? An actor.

Actor: a user of the program or a stakeholder, or a group of such people.

# SINGLE RESPONSIBILITY PRINCIPLE

“This principle is about people. ... When you write a software module, you want to make sure that when changes are requested, **those changes can only originate from a single person, or rather, a single tightly coupled group of people representing a single narrowly defined business function.** You want to **isolate your modules from the complexities of the organization as a whole**, and design your systems such that **each module is responsible (responds to) the needs of just that one business function.**” [Uncle Bob, [The Single Responsibility Principle](#)]

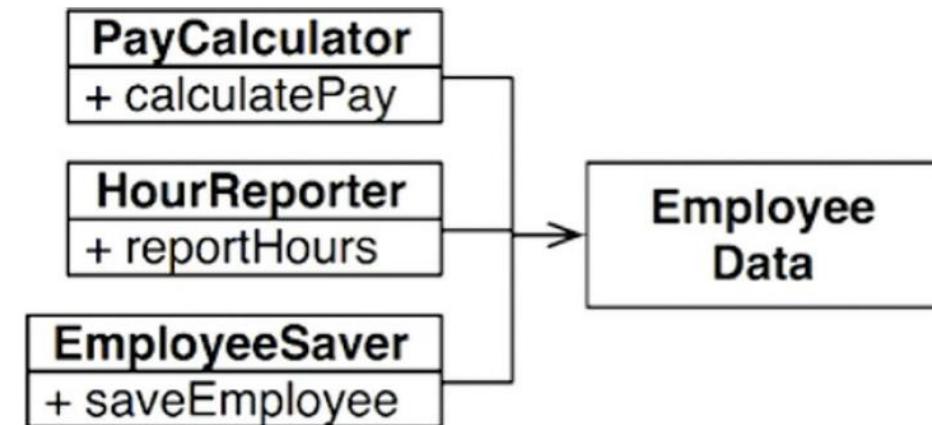
# A STORY OF THREE ACTORS

- Domain: an Employee class from a payroll application.
  - calculatePay: accounting department (CFO)
  - reportHours: human resources department (COO)
  - save: database administrators (CTO)
- Suppose methods calculatePay and reportHours share a helper method to calculate regularHours (and avoid duplicate code).
- CFO decides to change how non-overtime hours are calculated and a developer makes the change.
- The COO doesn't know about this. What happens?

Employee
+ calculatePay
+ reportHours
+ save

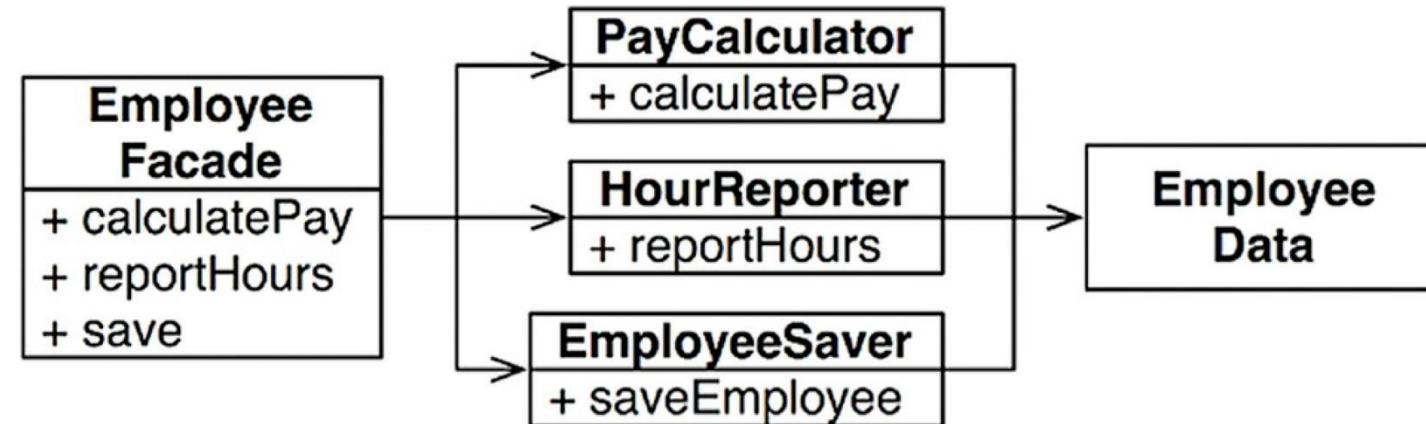
# CAUSE OF PROBLEM AND SOLUTION

- Cause of Problem: code is “owned” by more than one actor
- Solution: adhere to the Single Responsibility Principle
  - Factor out the data storage into an EmployeeData class.
  - Create three separate classes, one for each actor.



# FAÇADE DESIGN PATTERN

- Downside of solution: need to keep track of three objects, not one.
- Solution: create a façade (“the front of a building”).
  - Very little code in the façade. Delegates to the three classes.



- We'll talk about the Façade design pattern and many more throughout the term.

# OPEN/CLOSED PRINCIPLE

- Software entities (classes, modules, functions, etc.) should be **open for extension, but closed for modification.**
- Add new features not by modifying the original class, but rather by extending it and adding new behaviours, or by adding plugin capabilities.
- “I’ve heard it said that the OCP is wrong, unworkable, impractical, and not for real programmers with real work to do. The rise of plugin architectures makes it plain that these views are utter nonsense. On the contrary, a strong plugin architecture is likely to be the most important aspect of future software systems.” [Uncle Bob, [The Open Closed Principle](#)]

# OPEN/CLOSED PRINCIPLE

- An example using inheritance
- The area method calculates the area of all Rectangles in the given array.
- What if we need to add more shapes?

Rectangle
- width: double - height: double
+ getWidth(): double + getHeight(): double + setWidth(w: double): void + setHeight(h: double): void

AreaCalculator
+ area(shapes: Rectangle[]): double

# OPEN/CLOSED PRINCIPLE

- We might make it work for circles too.
- We could implement a **Circle** class and **rewrite** the area method to take in an **array of Objects** (using `isinstanceof` to determine if each Object is a **Rectangle** or a **Circle** so it can be cast appropriately).

Rectangle
- width: double
- height: double
+ getWidth(): double
+ getHeight(): double
+ setWidth(w: double): void
+ setHeight(h: double): void

Circle
- radius: double
+ getRadius(): double
+ setRadius(r: double): void

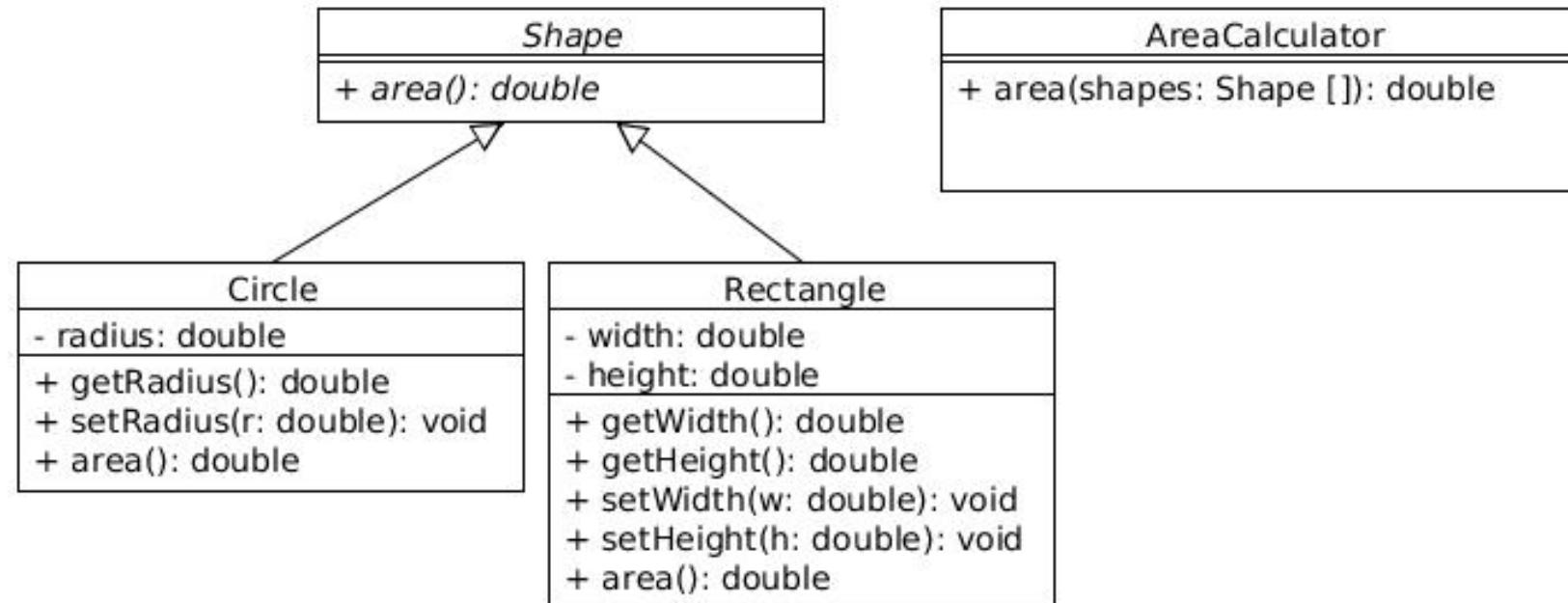
AreaCalculator
+ area(shapes: Object []): double

- But what if we need to add even more shapes?



# OPEN/CLOSED PRINCIPLE

- With this design, we can add any number of shapes (open for extension) and we don't need to re-write the AreaCalculator class (closed for modification).



# LISKOV SUBSTITUTION PRINCIPLE

LSP

- If  $S$  is a subtype of  $T$ , then objects of type  $S$  may be substituted for objects of type  $T$ , without altering any of the desired properties of the program.
- “ $S$  is a subtype of  $T$ ”?

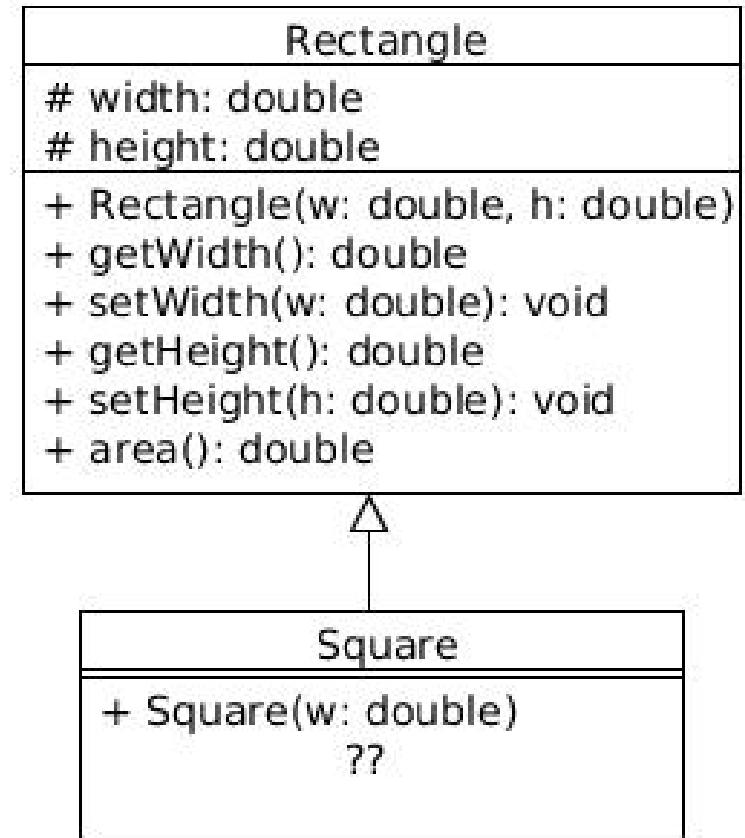
In Java, this means that  $S$  is a child class of  $T$ , or  $S$  *implements* interface  $T$ .

- “A program that uses an interface must not be confused by an implementation of that interface.” [Uncle Bob]

# LISKOV SUBSTITUTION PRINCIPLE

## Example

- Mathematically, a square “is a” rectangle.
- In object-oriented design, it is not the case that a Square “is a” Rectangle!
- This is because a Rectangle has *more* behaviours than a Square, not less.
- The LSP is related to the Open/Closed principle: the subclasses should only extend (add behaviours), not modify or remove them.



# INTERFACE SEGREGATION PRINCIPLE

ISP

- Here, interface means the public methods of a class. (In Java, these are often specified by defining an interface, which other classes then implement.)
- Context: a class that provides a service for other “client” programmers usually requires that the clients write code that has a particular set of features. The service provider says “your code needs to have this interface”.
- No client should be forced to implement irrelevant methods of an interface. Better to have lots of small, specific interfaces than fewer larger ones: easier to extend and modify the design.
- (Uh oh: “The interface keyword is harmful.” [Uncle Bob, ['Interface' Considered Harmful](#)] — we encourage you to read this and discuss with others. Does the fact that Java supports “default methods” for interfaces change anything?)

# INTERFACE SEGREGATION PRINCIPLE

ISP

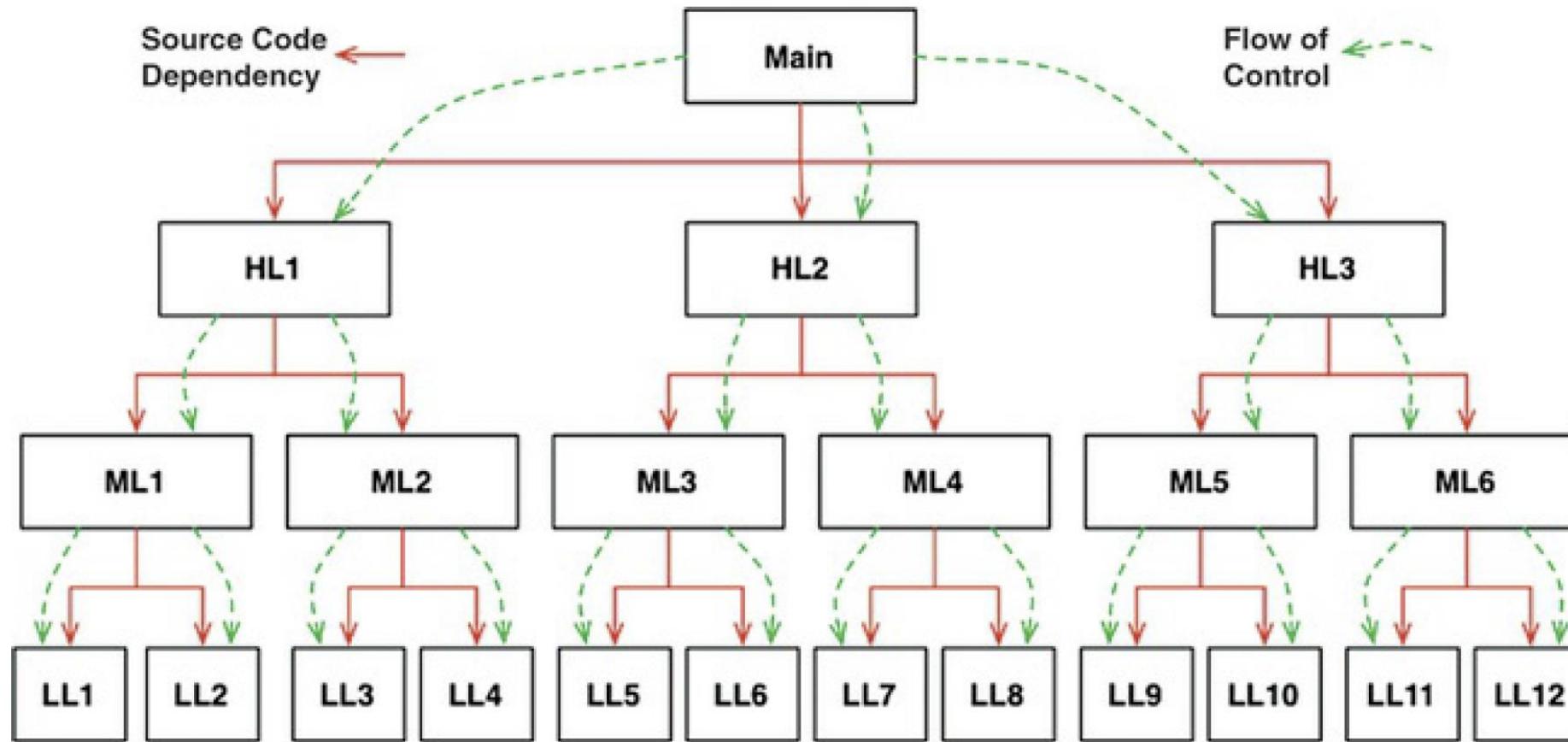
**“Keep interfaces small so that users don’t end up depending on things they don’t need.”**

We still work with compiled languages. We still depend upon modification dates to determine which modules should be recompiled and redeployed. So long as this is true we will have to face the problem that when module A depends on module B at compile time, but not at run time, then changes to module B will force recompilation and redeployment of module A.” [ Uncle Bob, [SOLID Relevance](#) ]

# DEPENDENCY INVERSION PRINCIPLE

- When building a complex system, programmers are often tempted to define “low-level” classes first and then build “higher-level” classes that use the low-level classes directly.
- But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced — an indication of high coupling.
- To avoid such problems, we introduce an **abstraction layer** between low-level classes and high-level classes.

# DEPENDENCY INVERSION PRINCIPLE



**Figure 5.1** Source code dependencies versus flow of control

Clean Architecture, Robert C. Martin



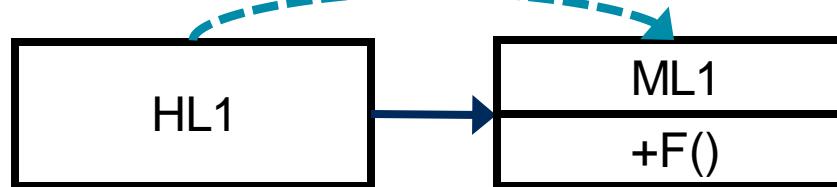
# DEPENDENCY INVERSION PRINCIPLE

Goal:

- You want to decouple your system so that you can change individual pieces without having to change anything more than the individual piece.
- Two aspects to the dependency inversion principle:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.

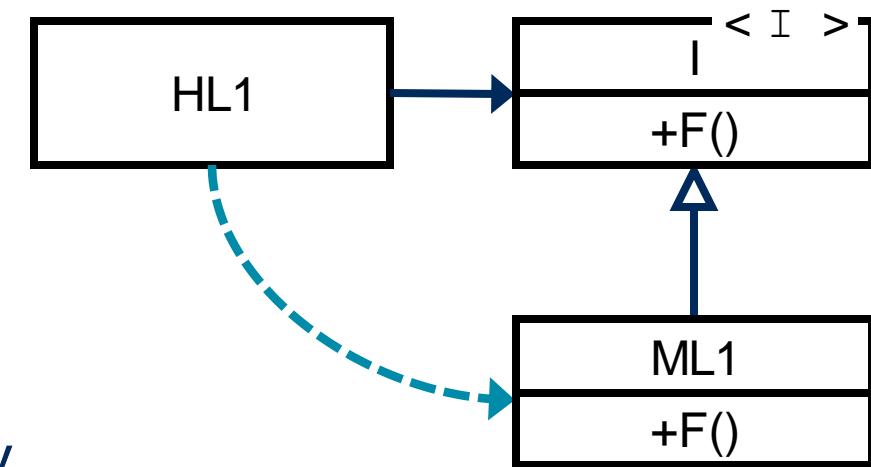
# DEPENDENCY INVERSION PRINCIPLE

How do we invert the source code dependency?



We introduce an interface!

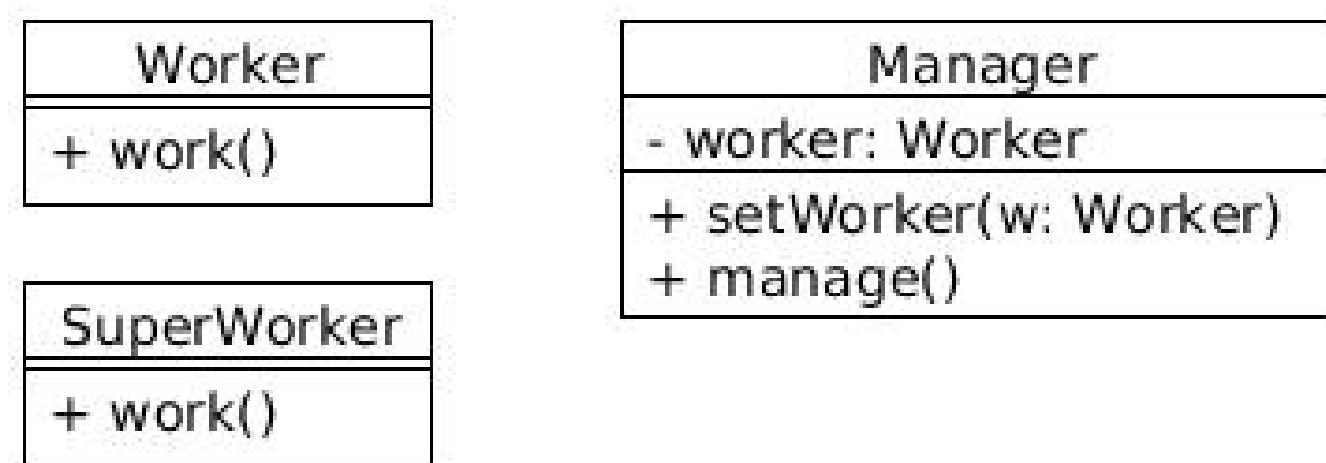
- The flow of control remains the same.
- HL1 depends on the interface and ML1 implements that interface.
- There is no longer a source code dependency between HL1 and ML1!



# EXAMPLE FROM DEPENDENCY INVERSION PRINCIPLE ON OODESIGN

DIP

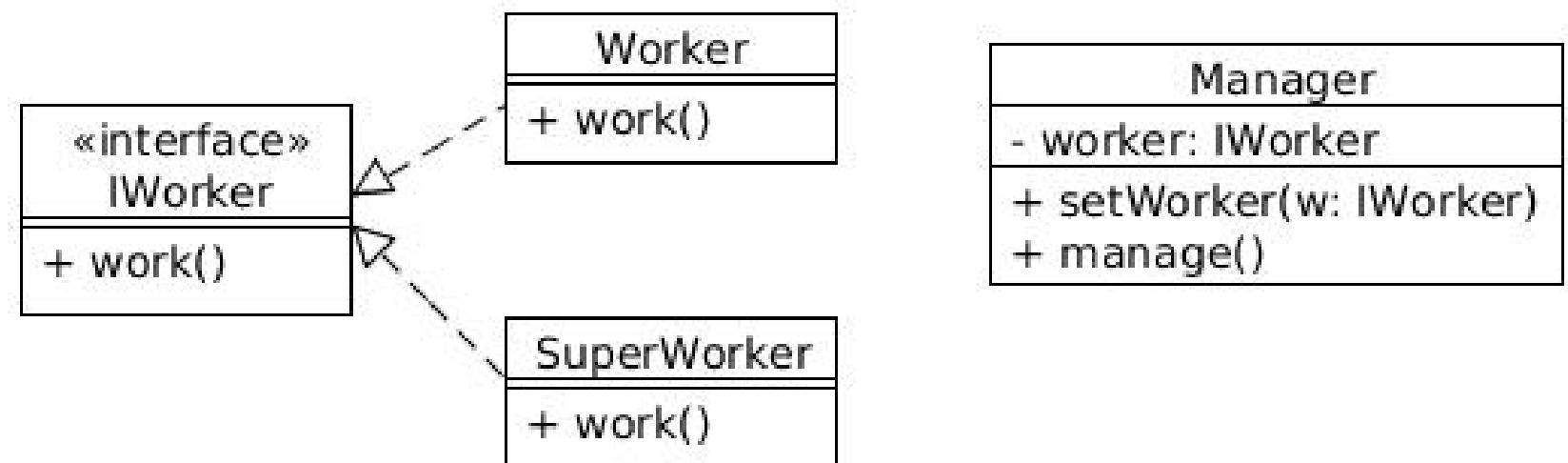
- A company is structured with managers and workers. The code representing the company's structure has Managers that manage Workers. Let's say that the company is restructuring and introducing new kinds of workers. They want the code updated to reflect this change.
- Your code currently has a Manager class and a Worker class. The Manager class has one or more methods that take Worker instances as parameters.
- Now there's a new kind of worker called SuperWorker whose behaviour and features are separate from regular Workers, but they both have some notion of "doing work".



# EXAMPLE FROM DEPENDENCY INVERSION PRINCIPLE ON OODESIGN

DIP

- To make Manager work with SuperWorker, we would need to rewrite the code in Manager (e.g. add another attribute to store a SuperWorker instance, add another setter, and update the body of manage())
- Solution: create an IWorker interface and have Manager depend on it instead of directly depending on the Worker and SuperWorker classes.
- In this design, Manager does not know anything about Worker, nor about SuperWorker. The code will work with any class implementing the IWorker interface and the code in Manager does not need to be rewritten.



# HOMEWORK

- We talked about Clean Architecture previously — consider each SOLID principle and how it relates to Clean Architecture.
- As you read and write Java code, start asking yourself whether it adheres to the SOLID principles.



# USER LOGIN: CRC EXAMPLE

CSC 207 SOFTWARE DESIGN

# LEARNING OUTCOMES

- Become more comfortable with Clean Architecture and CRC through our user login example

# USE CASES FOR A PROGRAM

Imagine you were asked to write a program that allows users to

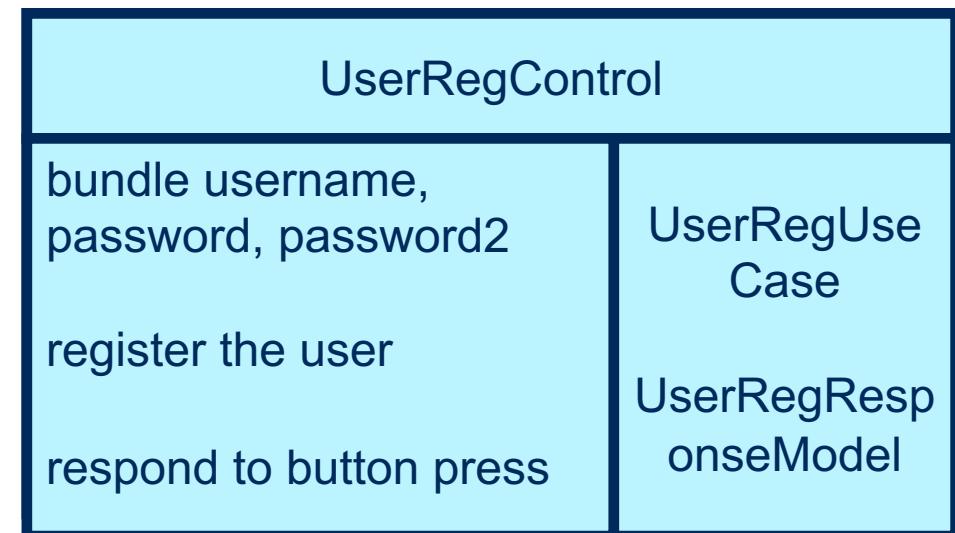
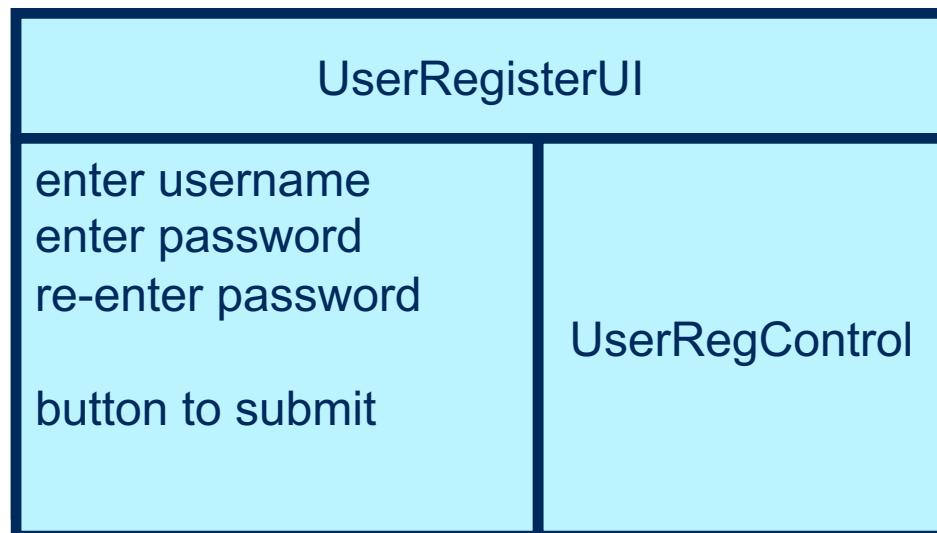
- **register a new user account** (with a username and password)
- log in to a user account
- log out of a user account

# USE CASE: USER REGISTERS NEW ACCOUNT

- The user chooses a username
- The user chooses a password and enters it twice (to help them remember)
- If the username already exists, the system alerts the user
- If the two passwords don't match, the system alerts the user
- The system creates a new user.
- If the password is not valid, then the system alerts the user, and the account is not saved
- If the password is valid, the account is saved, and the user is alerted.

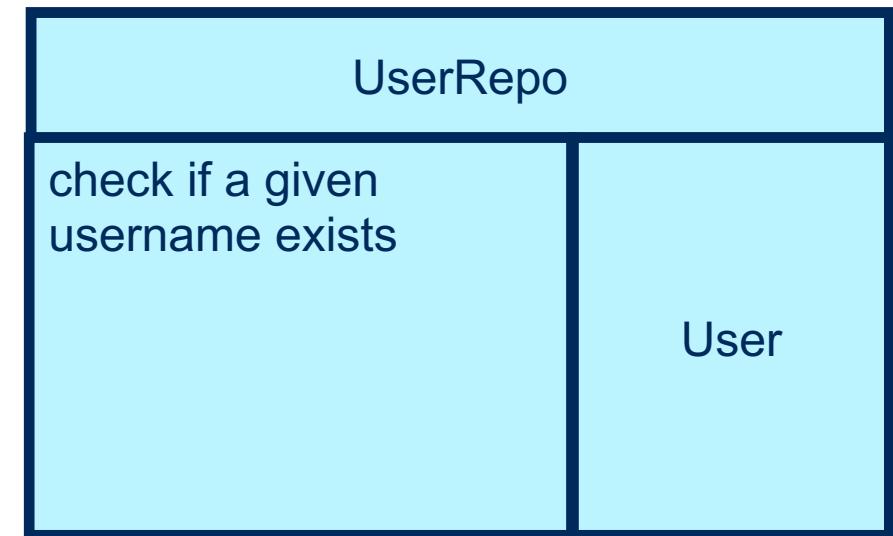
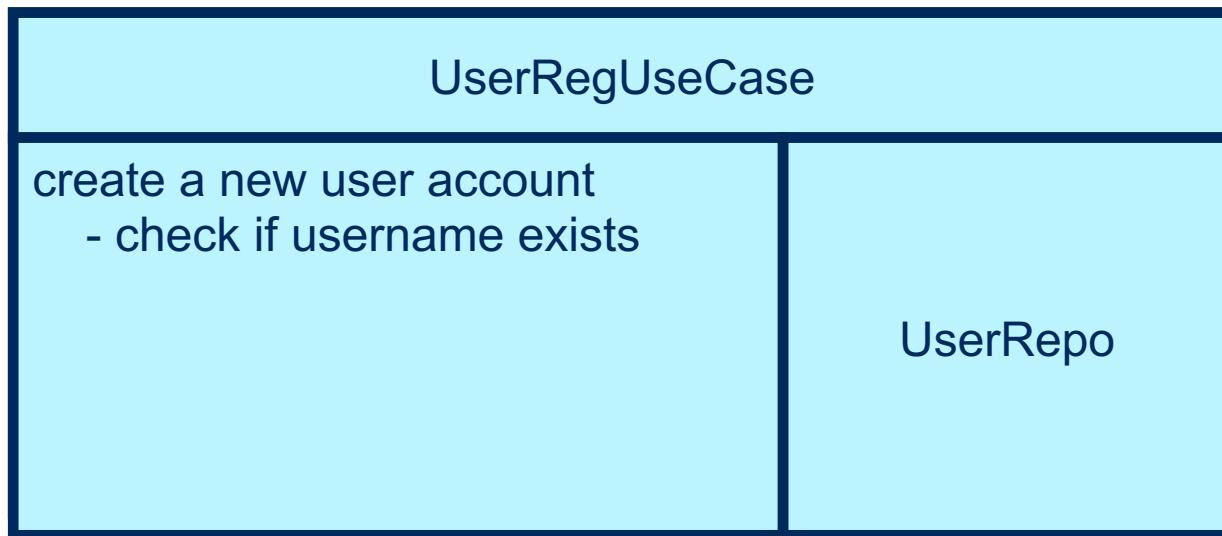
# GETTING THE INPUT

- The user chooses a username
- The user chooses a password and enters it twice (to help them remember)



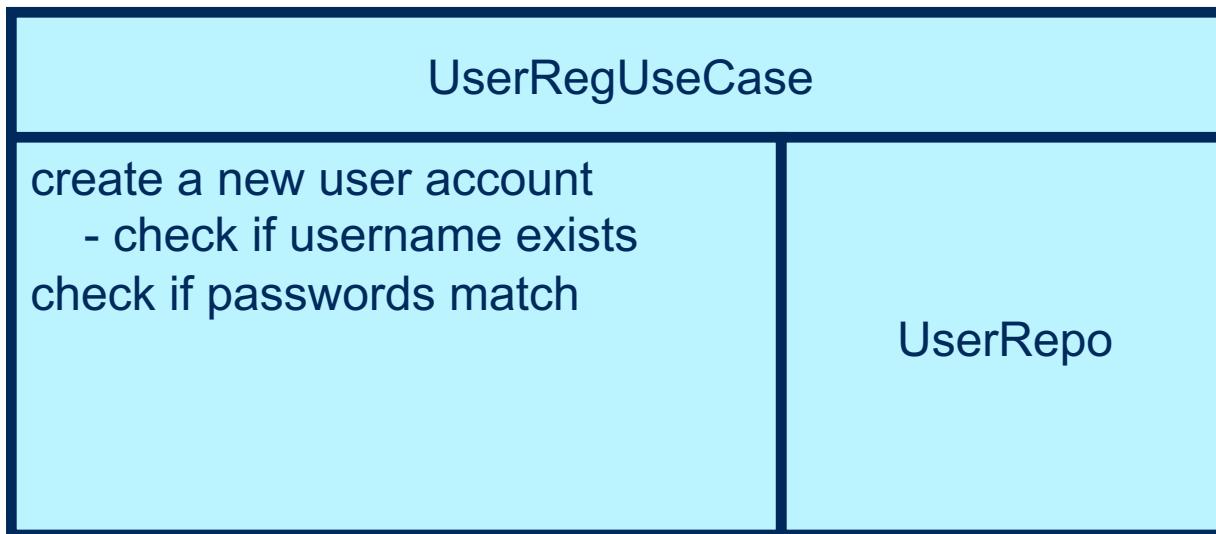
# THE USE CASE CLASS

- If the username already exists, the system alerts the user



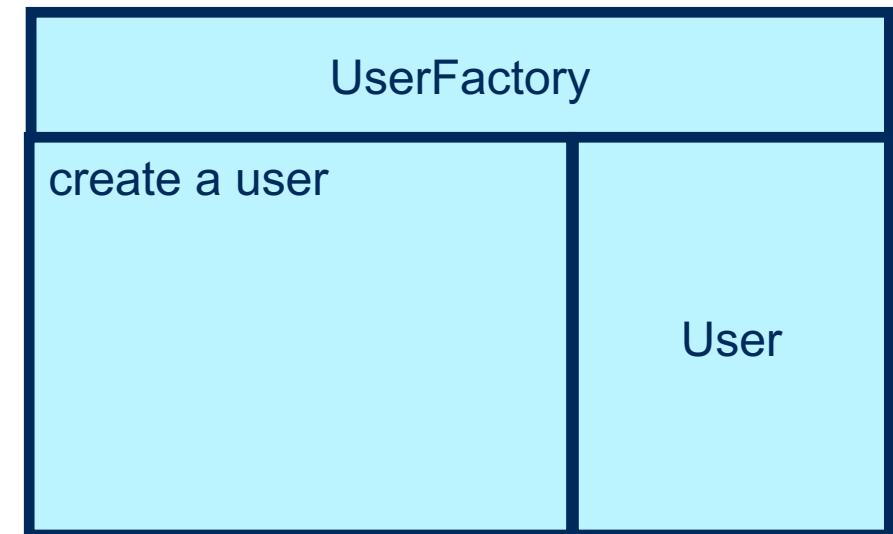
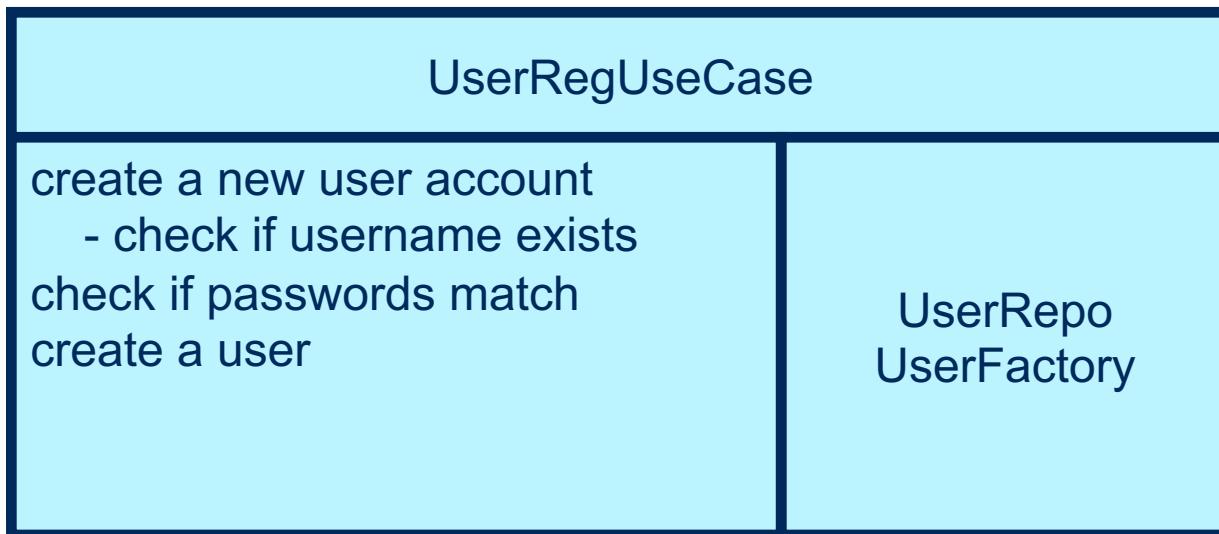
# THE USE CASE CLASS

- **If the two passwords don't match, the system alerts the user**



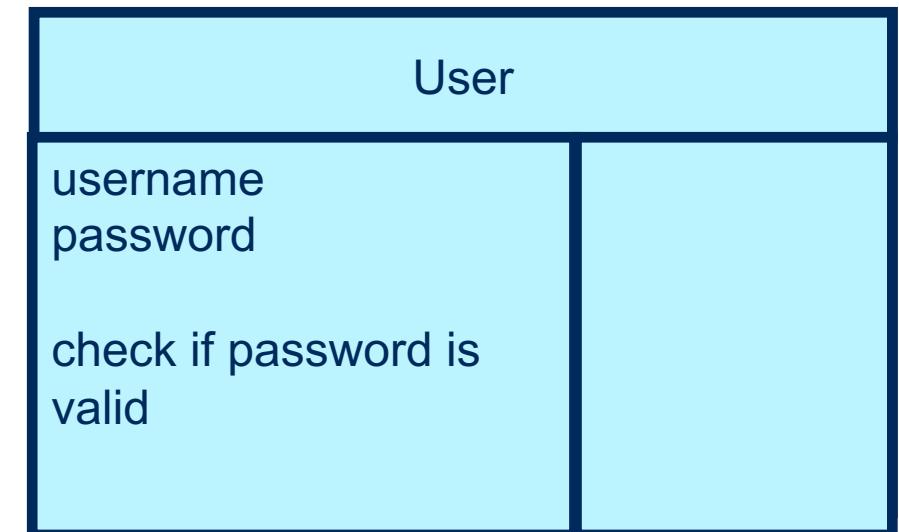
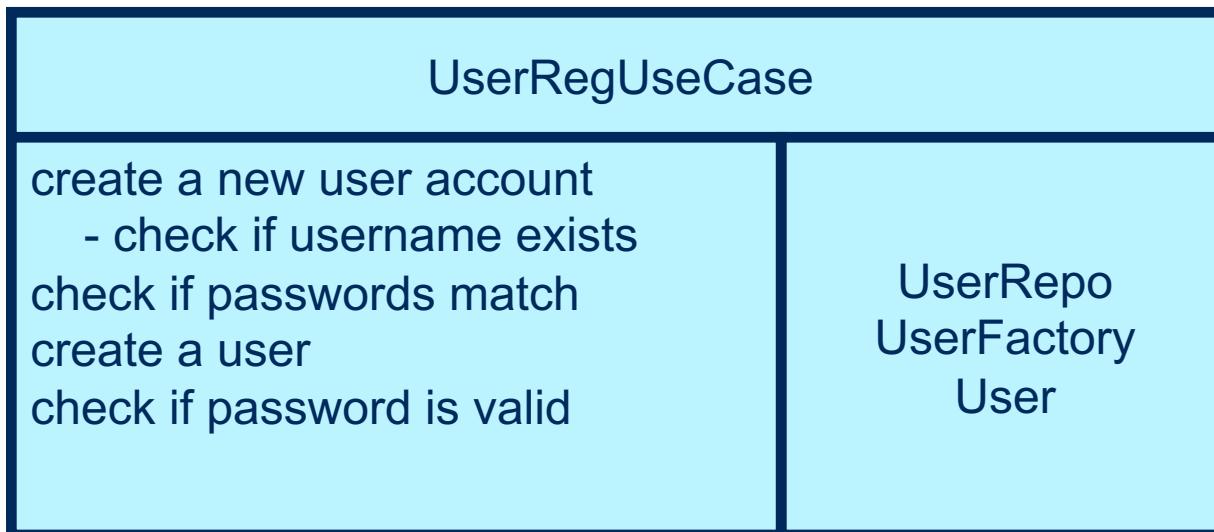
# THE USE CASE CLASS

- the system creates a new user.



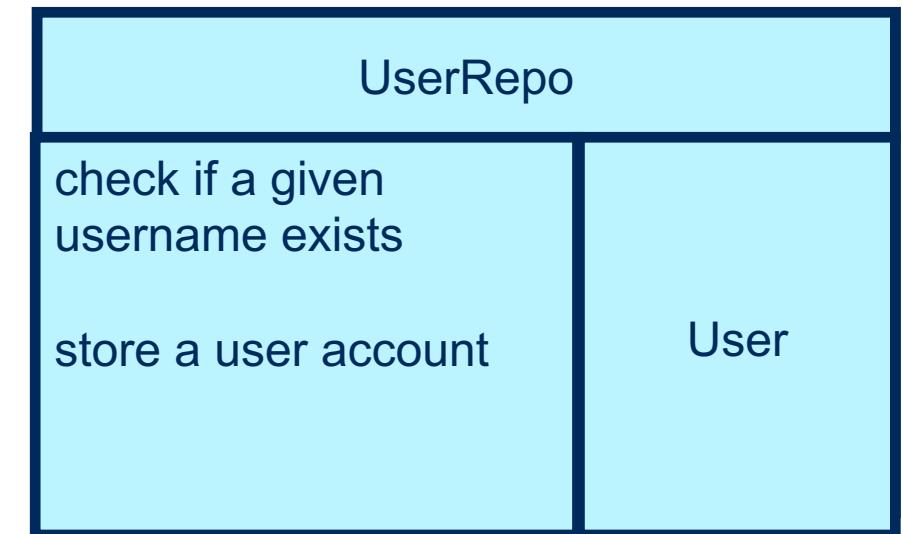
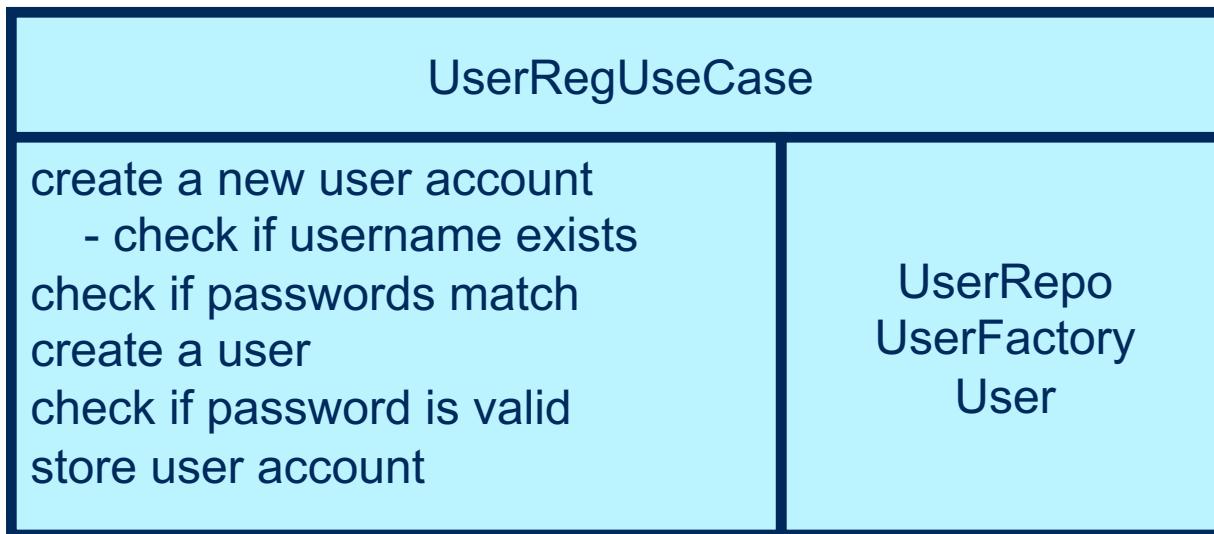
# THE USE CASE CLASS

- **If the password is not valid, then the system alerts the user, and the account is not saved**



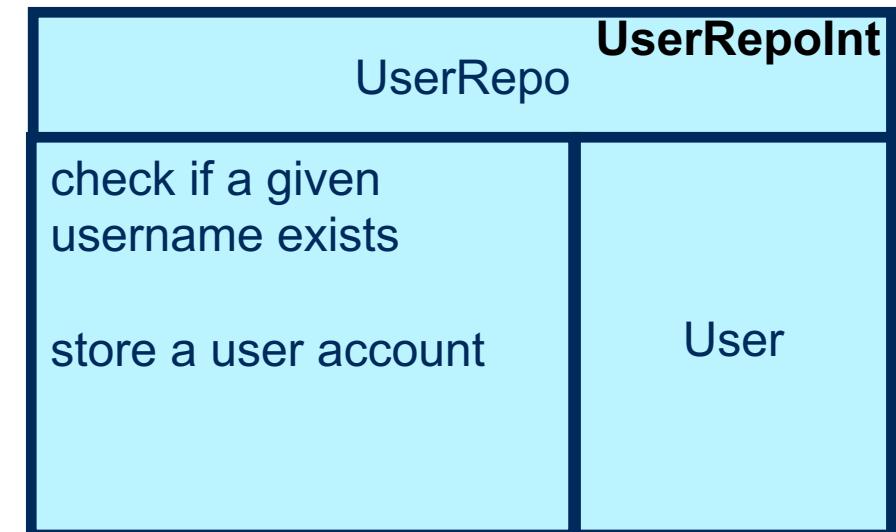
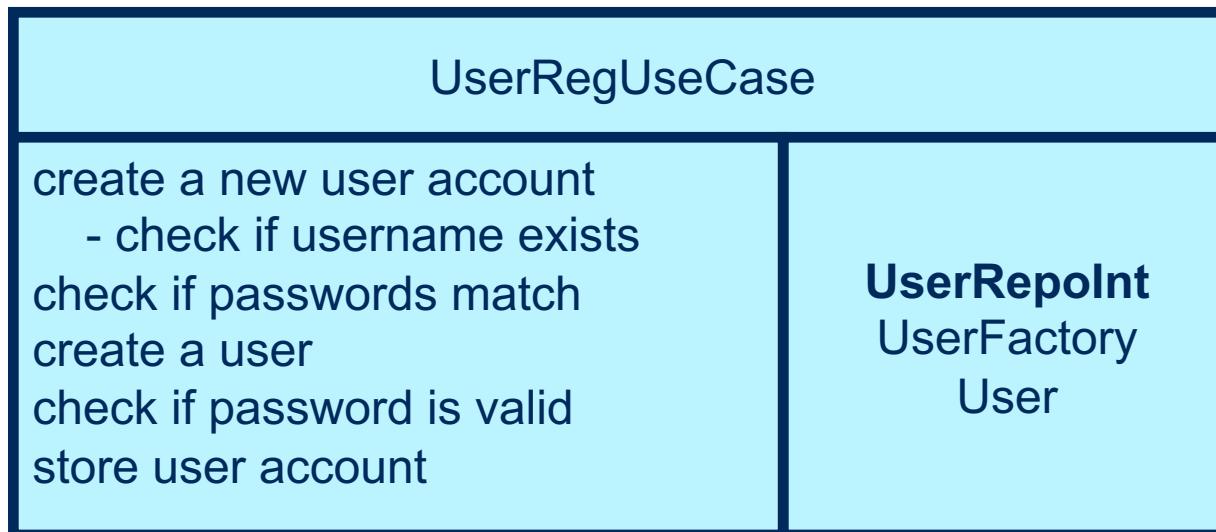
# THE USE CASE CLASS

- If the password is valid, **the account is saved**, and the user is alerted.



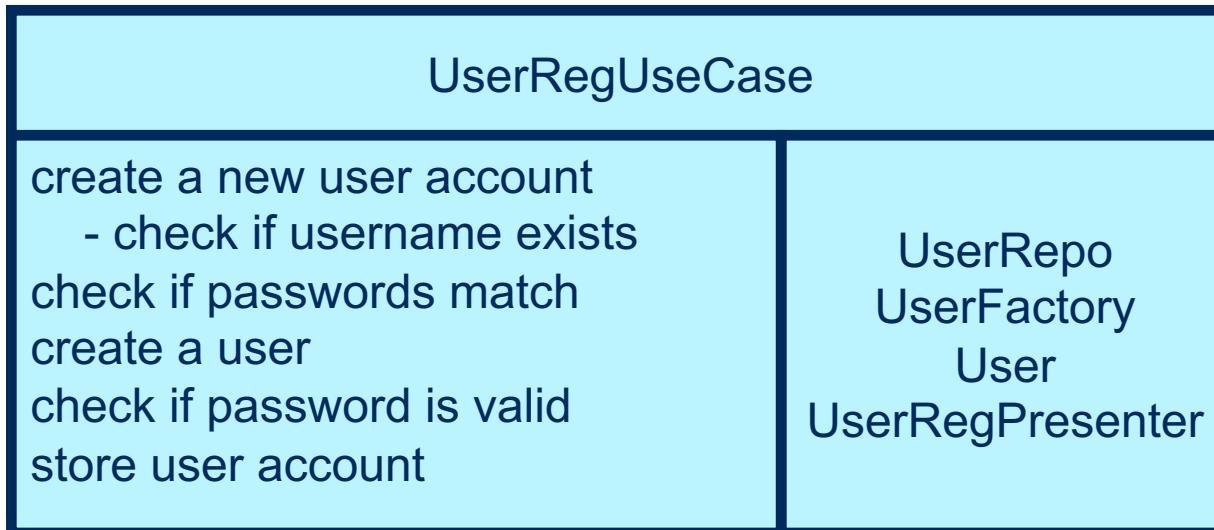
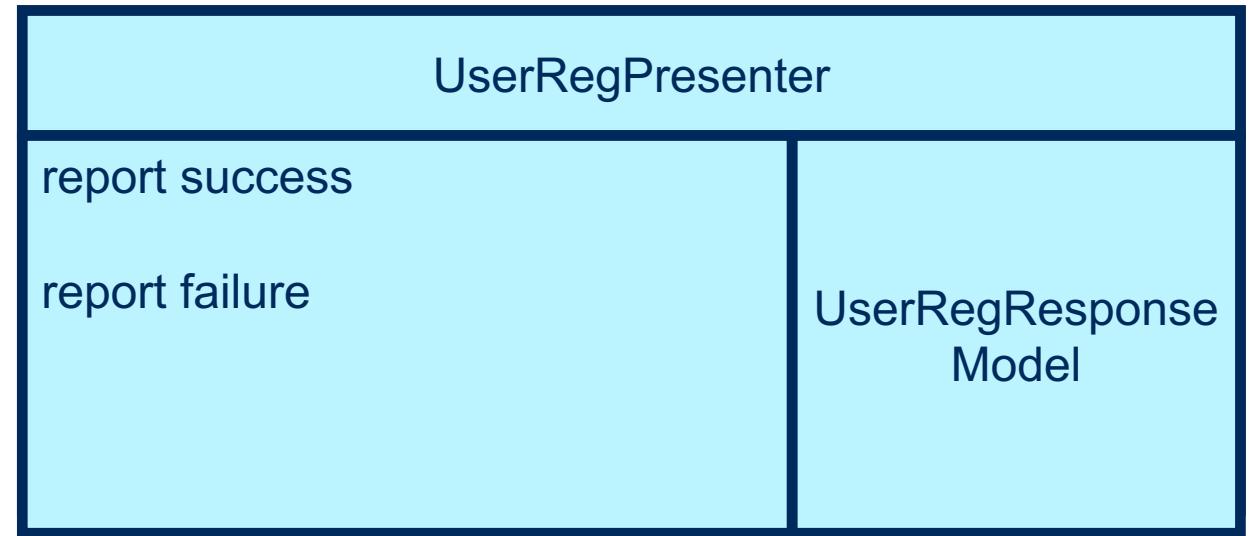
# DEPENDENCY INVERSION

- But wait, if our `UserRepo` class is saving data, then our use case class shouldn't depend on it directly — we need an interface!



# THE USE CASE CLASS

- the user is alerted



# USE CASE APIs (SUMMARY)

A class for each use case

UserRegisterInteractor

- Public API: a method called `create` (perhaps with the username and password as parameters, or maybe they're wrapped up in a data class)
- Needs to tell the UI to prepare a success view or a failure view
- Needs to know how to make a new user
- Needs to ask the persistence layer whether a username exists
- Needs to tell the persistence layer to save a new user

# REGISTER NEW USER USE CASE SKETCH

```
class UserFactory:  
    def create(name: str, password: str) -> User  
  
class UserRegisterInteractor:  
    _user_factory # a UserFactory  
    _user_presenter # controls the UI  
    _user_register_gateway # a persistence object  
    create(name: str, password: str) -> UserRegisterResponseModel
```

# TOURING THE CODE

<https://github.com/paulgries/UserLoginCleanArchitecture>

Some questions to consider

- How does the system initially get started?
- How does our use case initialize its `_user_factory`, `_user_presenter`, and `_user_register_gateway` instance variables?
- What happens when we click the “register” button?
- Can we identify each of our CRC cards in the implementation? Did we miss any of the important classes when making our CRC model?

# HOMEWORK

---

- Continue exploring the code and ask questions if you aren't sure about any parts of it.
- Take one of the other use cases for the system and repeat the exercise we went through here.



# JAVA RECAP + LOOSE ENDS

## CSC 207 SOFTWARE DESIGN

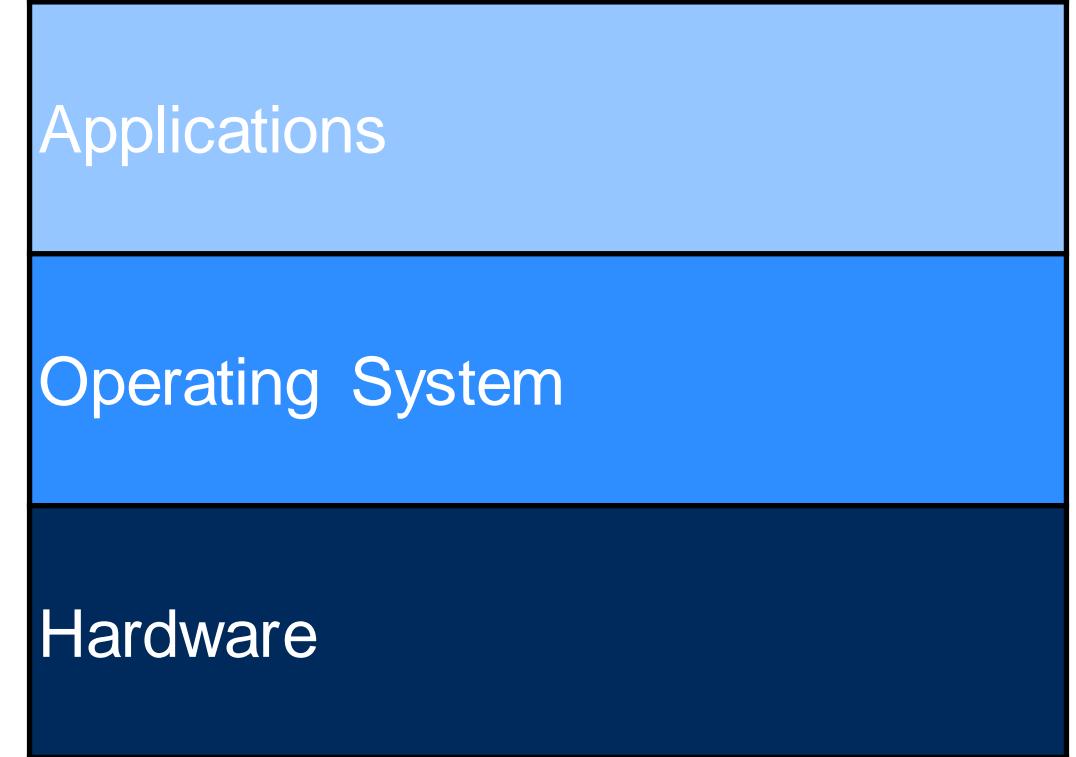


# LEARNING OUTCOMES

- Know enough Java to be able to contribute to your team project this term.

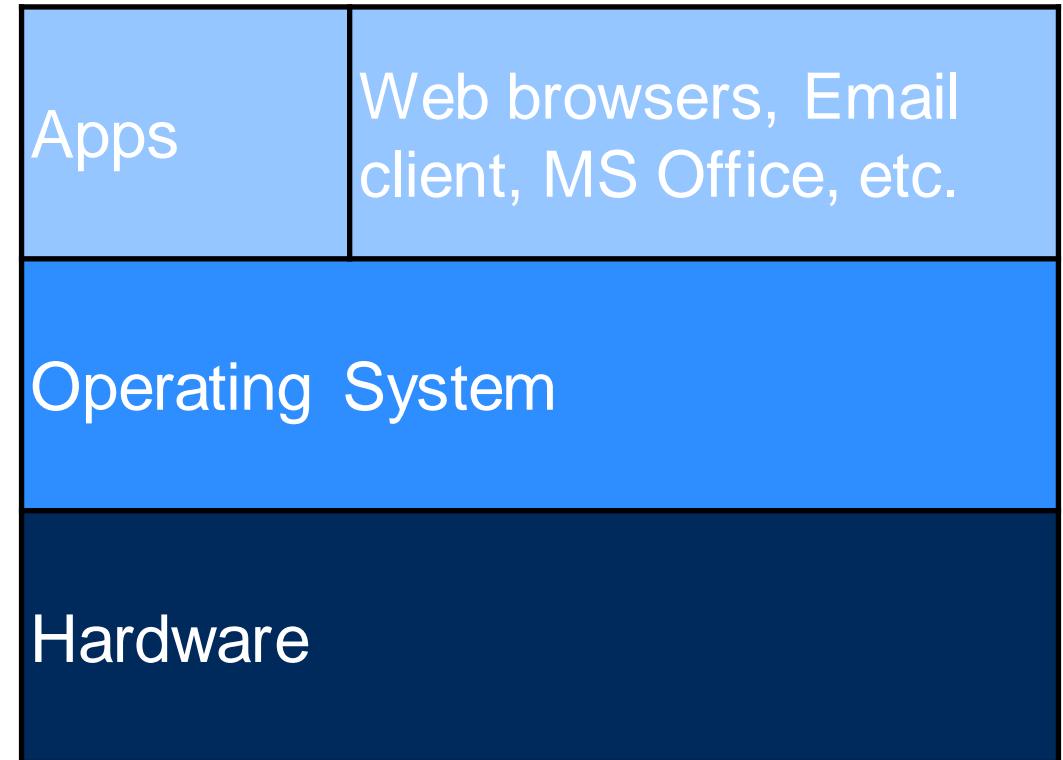
# COMPUTER ARCHITECTURE

- 108 and 148: we focused on writing applications that “Python” ran. But what is Python?
- The operating system (OS)
  - Works directly with the hardware (CSC258H, CSC369H) and
  - Manages the various running applications (CSC209H, CSC369H), helping them interact with the hardware.
- Notice the layers!



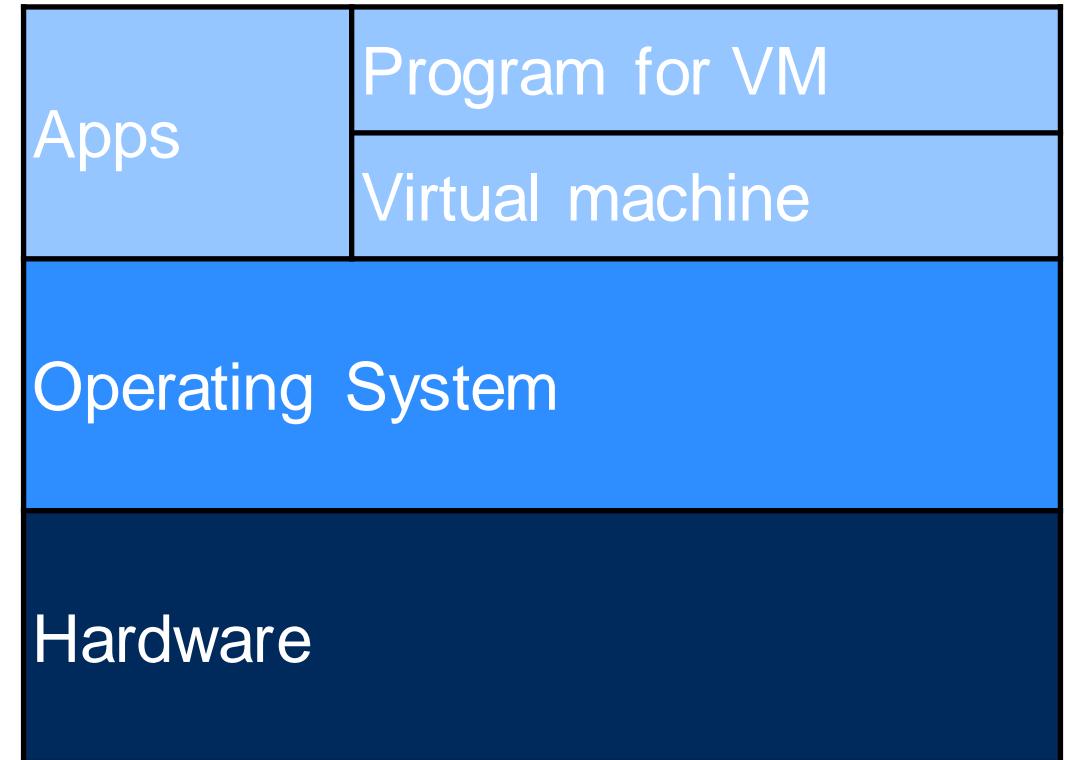
# COMPILED APPLICATIONS

- Programs written in languages like C, Swift, C++, Rust, and Fortran are **compiled** into “native” applications.
- Compilation involves turning **human-readable** programs into **OS-specific machine-readable** code (CSC258H, CSC369H, CSC488H).



# VIRTUAL MACHINE ARCHITECTURE

- **Virtual machine (VM)**: an application that pretends to be a computer.
- Java, Python, and Scheme have their own VM.
  - The same compiled Java program can run in any JVM on any OS
  - The JVM optimizes the byte code as it runs (CSC324H, CSC488H) and can even be as fast as code written in C!
- Each VM application is compiled for each OS so programs in the languages are **portable** across operating systems.
- [Further reading on StackOverflow](#)



# JAVA CONVENTIONS

# JAVA CONVENTIONS TO FOLLOW FOR YOUR PROJECT

- Make all your instance variables `private` and create getters and setters, as necessary.
- `AllClassNamesAreCamelCase` with first letter capitalized
- `packages_use_lowercase_pothole`
- `variablesAreCamelCase` with first letter lowercase
- `methodsAreCamelCase` with first letter lowercase
- `FINAL_VARIABLES_ARE_ALL_CAPS` with underscores separating the words

# ALWAYS USE BRACES

```
if (something) {  
    // even if this is only one line, use the braces.  
}  
else {  
    // here too!  
}  
  
for (item in thingy) {  
    // even if this is only one line, use the braces!  
}
```

# JAVA MEMORY MODEL

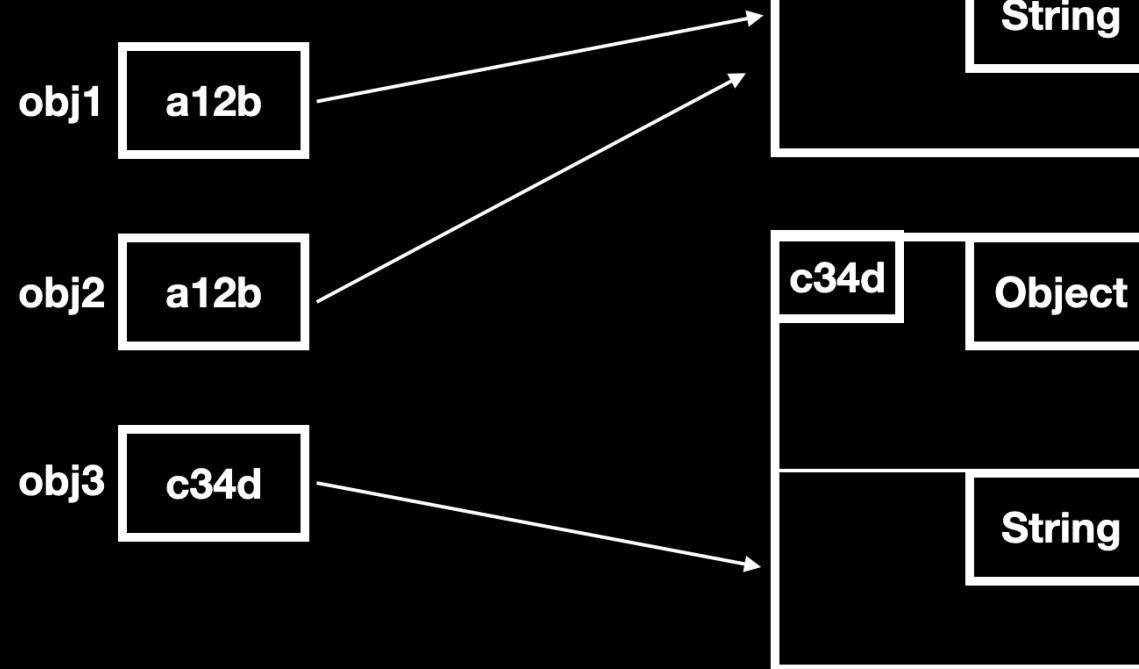


# Reference Types

```
String obj1 = "hello";
```

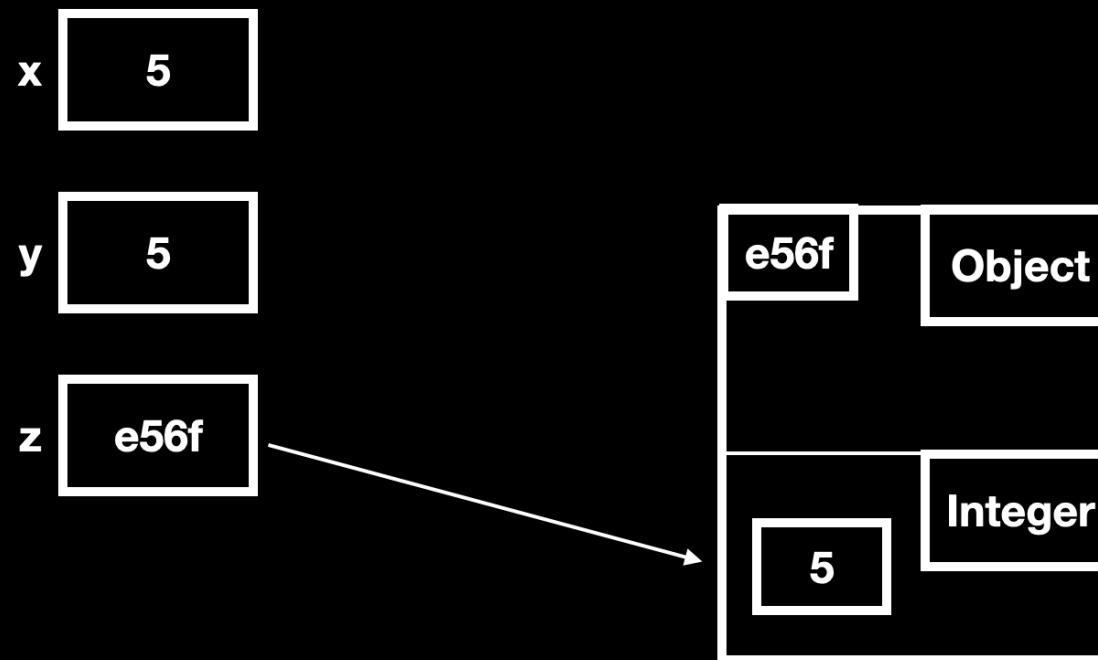
```
String obj2 = obj1;
```

```
String obj3 = "hi";
```



```
int x = 5;  
int y = x;  
Integer z = new Integer(5);
```

## Primitive Types



# Casting

```
Person s1 = new Student("A B");
```

```
Student s2 = (Student) s1;
```

s1

x89y

s2

x89y

x89y  
Object

equals

equals

Person

equals

Student

getStudentNum

Note: s1.getStudentNum() won't work  
but s2.getStudentNum() will work.



# INSTANCE VARIABLES AND ACCESSIBILITY

- If an instance variable is private, how can client code use it?
- Why not make everything public — so much easier!
- More about access modifiers at  
<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

# ENCAPSULATION

- Think of your class as providing an abstraction, or a service.
  - We provide access to information through a well-defined interface: the public methods of the class.
  - We hide the implementation details.
- What is the advantage of this “encapsulation”?
  - We can change the implementation — to improve speed, reliability, or readability — and *no other code must change!*

# INHERITANCE IN JAVA

# INHERITANCE HIERARCHY

- All classes form a tree called the inheritance hierarchy, with `Object` at the root.
- Class `Object` does not have a parent. All other Java classes have one parent.
- If a class has no parent declared, it is a child of class `Object`.
- A parent class can have multiple child classes.
- Class `Object` guarantees that every class inherits methods `toString`, `equals`, and others.

# MULTI-PART OBJECTS

- Suppose class `Child` extends class `Parent`.
- An instance of `Child` has
  - a `Child` part, with all the data members and methods of `Child`
  - a `Parent` part, with all the data members and methods of `Parent`
  - a `Grandparent` part, ... etc., all the way up to `Object`.
- An instance of `Child` can be used anywhere that a `Parent` is legal.
  - But not the other way around. (`Child` may have methods not in the `Parent` class!)

# SHADOWING AND OVERRIDING

- Suppose class `A` and its subclass `AChild` each have an instance variable `x` and an instance method `m`.
- `A`'s `m` is **overridden** by `AChild`'s `m`.
  - **This is often a good idea. We often want to specialize behaviour in a subclass.**
- `A`'s `x` is **shadowed** by `AChild`'s `x`.
  - **This is confusing and rarely a good idea.**
  - **Avoid instance variables with the same name in a parent and child class.**
- If a method must not be overridden in a descendant, declare it `final`.

# CASTING FOR THE COMPILER

- If we could run this code, Java would find the `charAt` method in `o`, since it refers to a `String` object:

```
Object o = new String("hello");
char c = o.charAt(1);
```

- But the code won't compile because the compiler cannot be sure it will find the `charAt` method in `o`.

**Remember: the compiler doesn't run the code – it can only look at the type of `o`.**

- So we need to cast `o` as a `String`:

```
char c = ((String) o).charAt(1);
```

# JAVADOC

- Like a Python docstring, but more structured, and placed above the method.

```
/**  
 * Replace a square wheel of diagonal diag with a round wheel of  
 * diameter diam. If either dimension is negative, use a wooden tire.  
 * @param diag Size of the square wheel.  
 * @param diam Size of the round wheel.  
 * @throws PiException If pi is not 22/7 today.  
 */  
public void squareToRound(double diag, double diam) { ... }
```

- Javadoc is written for classes, member variables, and member methods.
- This is where the Java API documentation comes from!

# WHAT HAPPENS WHEN WE CREATE AN OBJECT?

1. Allocate memory for the new object.
2. Initialize the instance variables to their default values:
  - 0 for ints, `false` for booleans, etc., and `null` for class types.
3. Call the appropriate constructor in the parent class.
  - The one called on the first line, if the first line is `super(arguments)`, else the no-arg constructor.
4. Execute any direct initializations in the order in which they occur.
5. Execute the rest of the constructor.

# THE PROGRAMMING INTERFACE

- The "user" for almost all code is a programmer. That user wants to know:
  - ... what kinds of object your class represents
  - ... what actions it can take (methods)
  - ... what properties your object has (getter methods)
  - ... what guarantees your methods and objects require and offer
    - ... how they fail and react to failure
    - ... what is returned and what errors are raised
- Document your classes!

# INTERFACE LEFT, CLASS RIGHT

- `List<String> ls = new List<>(); // Fails`
- `List<String> ls = new ArrayList<>();`
- **We can choose a different implementation of List at any point.**

# **GENERICs**



# GENERICs (FANCIER TYPE PARAMETERS)

- “`class Foo<T>`” introduces a class with a type parameter `T`.
- “`<T extends Bar>`” introduces a type parameter that is required to be a descendant of the class `Bar` — with `Bar` itself a possibility.
  - In a type parameter, “`extends`” is also used to mean “`implements`”.
- “`<? extends Bar>`” is a type parameter that can be any class that extends `Bar`. We’ll never refer to this type later, so we don’t give it a name.
- “`<? super Bar>`” is a parameter that can be any ancestor of `Bar`.

<https://docs.oracle.com/javase/tutorial/java/generics/types.html>

# AN INTERFACE WITH GENERICS

- ```
public interface Comparable<T> {  
    /**  
     * Compare this object with o for order.  
     * Return a negative integer, zero, or a  
     * positive integer as this object is less  
     * than, equal to, or greater than o.  
     */  
    int compareTo(T o); // No body at all.  
}
```
- ```
public class Student implements Comparable<Student> {  
    . . .  
    public int compareTo(Student other) {  
        // Here we need to provide a body for the method.  
    }  
}
```

# GENERICs: NAMING CONVENTIONS

- The Java Language Specification recommends these conventions for the names of type variables:
  - very short, preferably a single character
  - but evocative
  - all uppercase to distinguish them from class and interface names
- Specific suggestions:
  - Maps: K, V
  - Exceptions: X
  - Nothing particular: T (or S, T, U or T1, T2, T3 for several)

# COLLECTIONS

- Equivalents to Python lists, dictionaries, and sets are in the standard “Collections” library.

- `import java.util.*;`
- `...`
- `List list = new ArrayList(); // list = []`
- `Map map = new HashMap(); // map = {}`
- `Set set = new HashSet(); // set = set()`

# HOMEWORK

Study for the Java test!

Continue working on your projects.



# MODEL VIEW

CSC207 SOFTWARE DESIGN

---

# CONTROLLER PRESENTER VIEWMODEL



Computer Science  
UNIVERSITY OF TORONTO

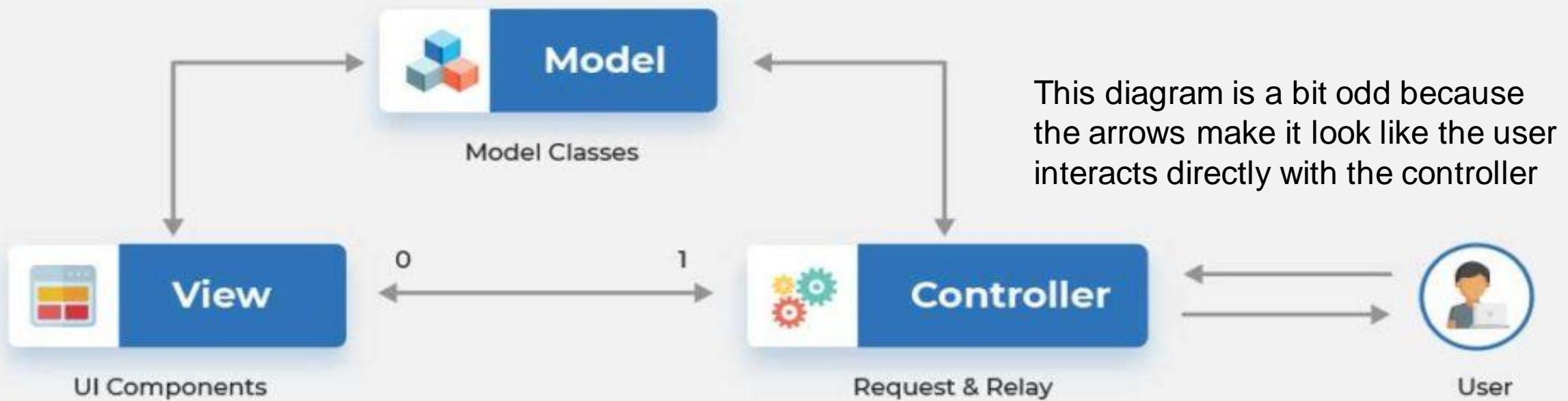
# LEARNING OUTCOMES

- Understand the basics of three related patterns
  - Model View Controller (MVC)
  - Model View Presenter (MVP)
  - Model View ViewModel (MVVM)
- Relate them to Clean Architecture



## MVC Pattern

Source: <https://www.bacancytechnology.com/blog/mvc-vs-mvp-vs-mvvm>

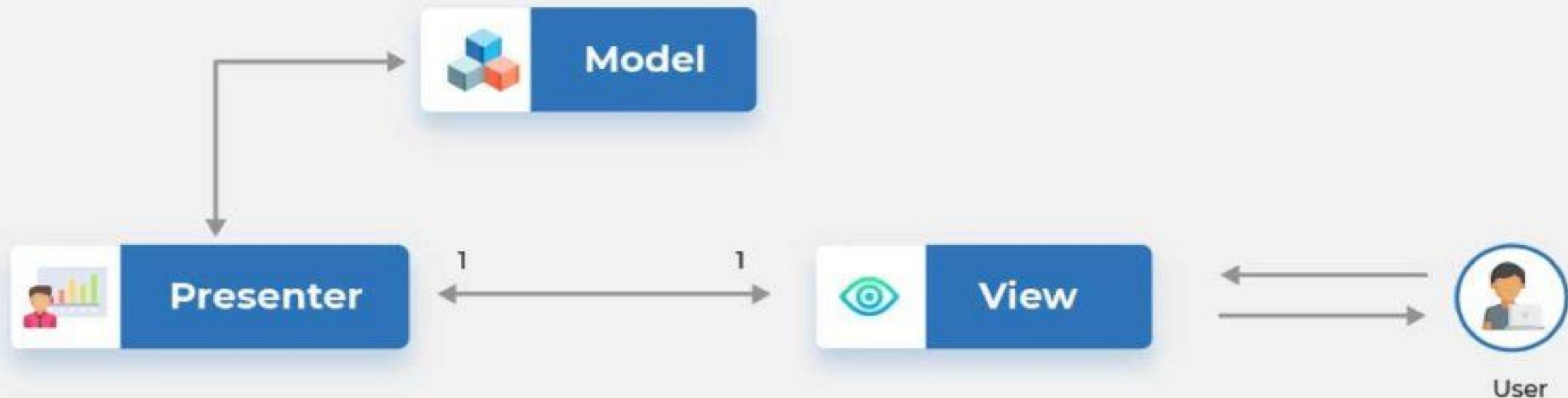


1. The user sees the View
2. The user interacts with the View, which immediately asks the Controller to take over
3. The Controller manipulates the Model
4. The Model updates the View





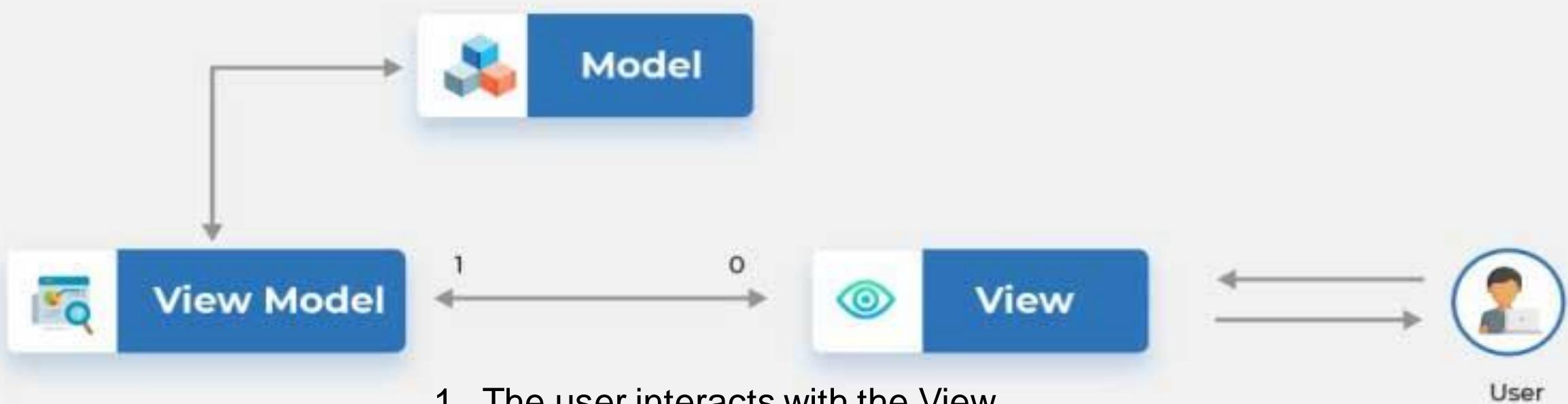
## MVP Pattern



1. The user interacts with the View, but the View immediately asks the Presenter to take over
2. The Presenter manipulates the model
3. The Presenter updates the View



## Model View ViewModel (MVVM) Pattern



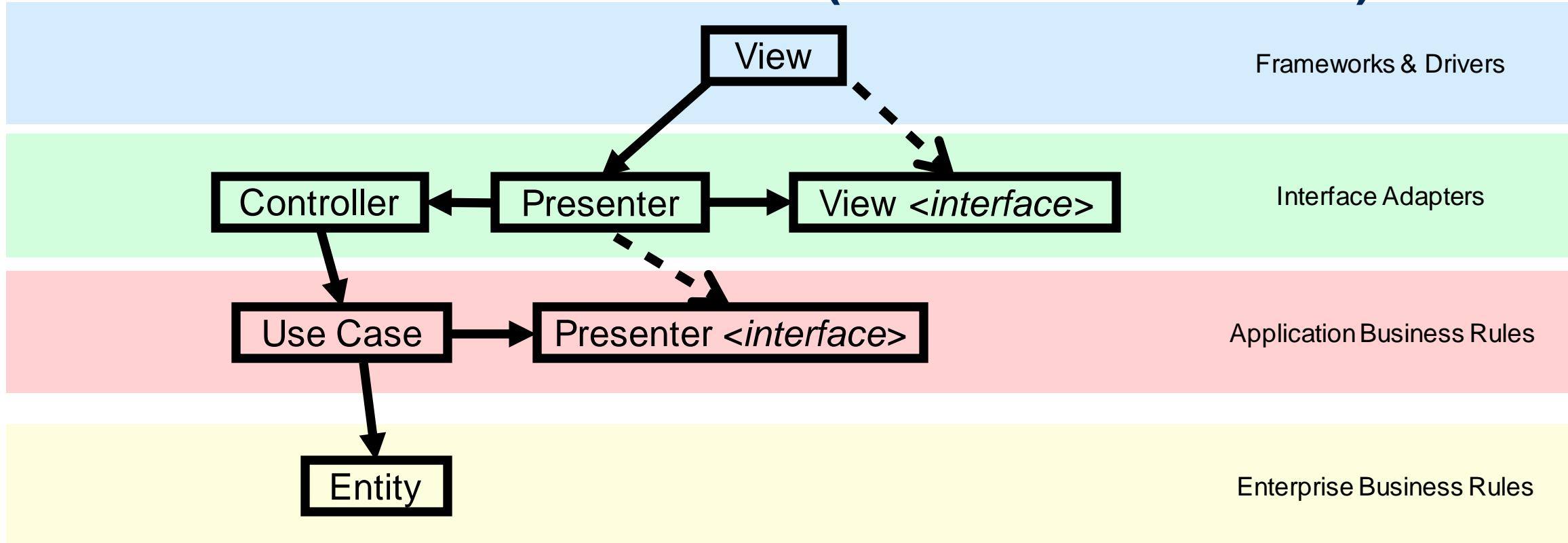
1. The user interacts with the View
2. The View updates information in the View Model
3. The View Model represents the state of the View — the View is very thin, and is bound to the View Model
4. The View Model passes control to the Model
5. The Model updates the View Model, which is observed by the View



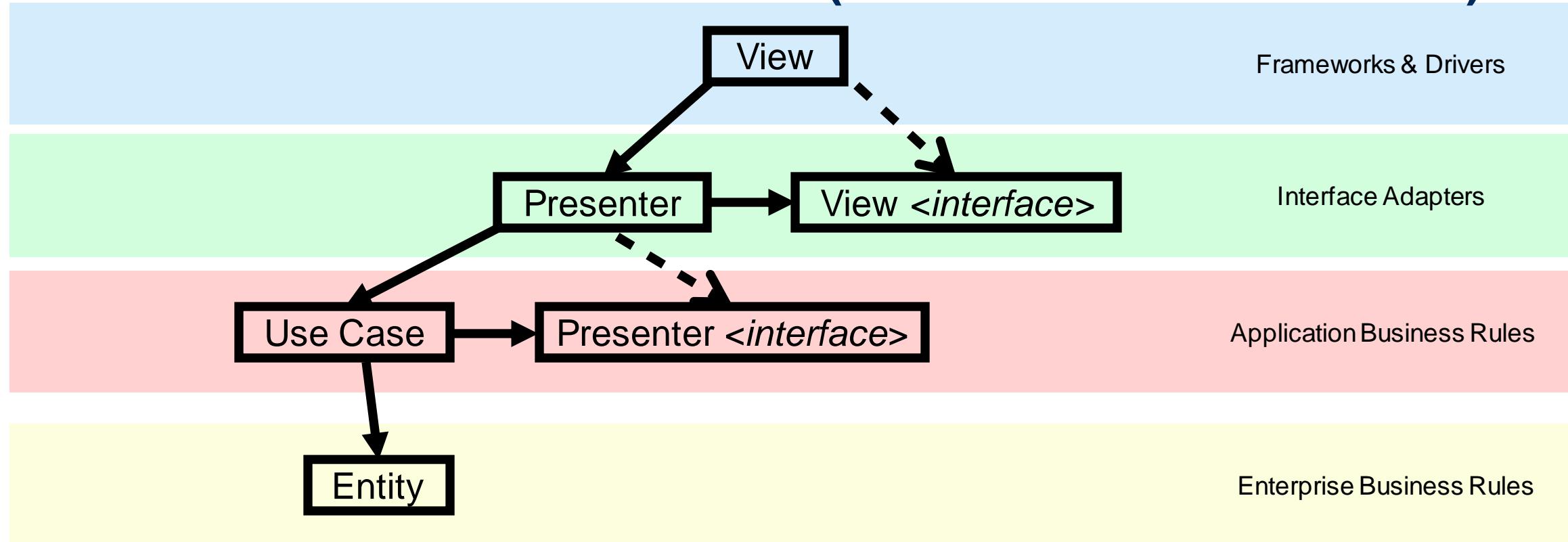
# MVP VS MVVM VS MVC

- MVP: the Presenter has a reference to a View and directly instructs it what to do.
  - All presentation logic is in the Presenter.
  - The “Presenter” also manipulates the Model, so it is functioning as the Controller in Clean Architecture!
- MVVM: no such link
  - View updates are managed through *binding* to the ViewModel, observing the changes.
  - This is usually cleaner, and allow UI/UX designers to focus purely on the GUI.
  - Programmers manage and update the state of the ViewModel, which is directly reflected in the View.
- MVC: the Controller doesn't directly update the View. The Model takes care of that.
- References
  - <https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/>
  - <https://github.com/husaynhakeem/TicTacToe-MVVM>

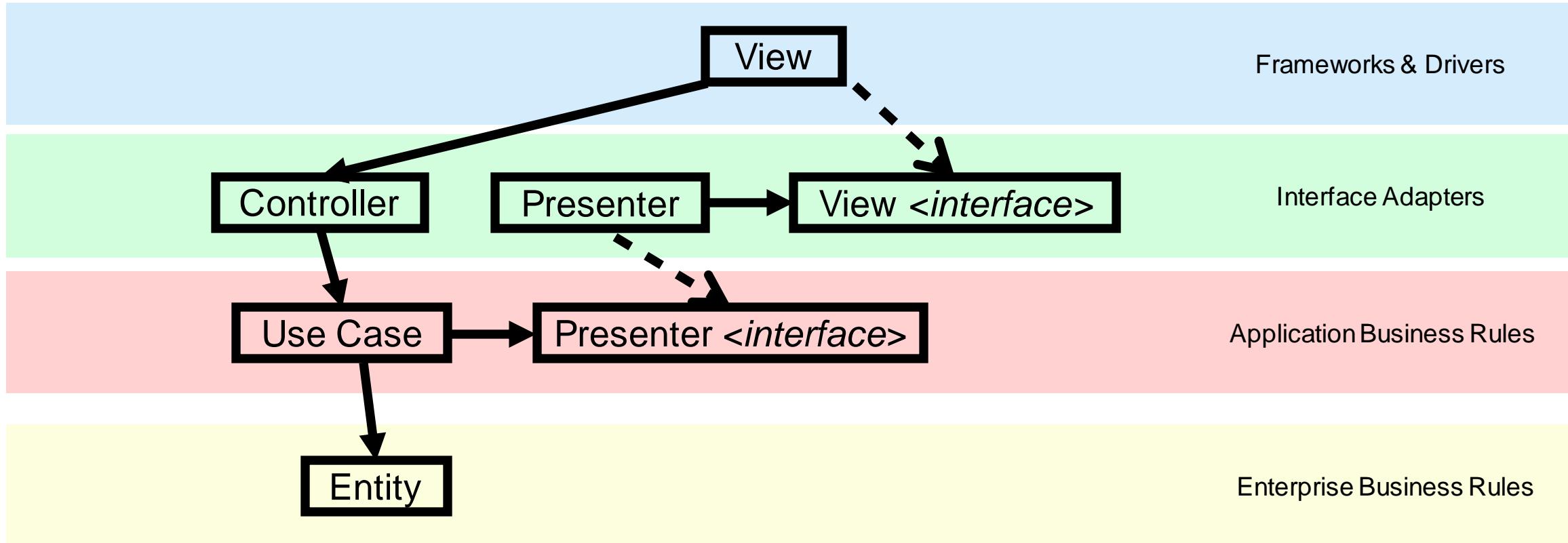
# MVP IN CLEAN ARCHITECTURE (WITH CONTROLLER)



# MVP IN CLEAN ARCHITECTURE (WITHOUT CONTROLLER)



# MVC IN CLEAN ARCHITECTURE



# SOME RESOURCES ABOUT MVP

- Telligent MVP paper:
  - <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>
- Detailed discussion of different UI architectures (including MVP):
  - <https://www.martinfowler.com/eaaDev/uiArchs.html>
- Some general discussion of the Model-View-Presenter (MVP) architecture:
  - <https://en.wikipedia.org/wiki/Model–view–presenter>
  - <https://softwareengineering.stackexchange.com/questions/60774/model-view-presenter-implementation-thoughts>
- Small code examples:
  - Java swing: <https://riptutorial.com/swing/example/14137/simple-mvp-example>
  - Android: <https://medium.com/cr8resume/make-you-hand-dirty-with-mvp-model-view-presenter-eab5b5c16e42>

# MVP EXAMPLE IN JAVA

- [link to repo](#) based on code in <https://riptutorial.com/swing/example/14137/simple-mvp-example>
- [MVC branch](#) of the above repo.

# EVENT DRIVEN PROGRAMMING

- Some programs run without user interaction, typically processing data
  - Most first-year assignments
- Some programs are *event-driven*: once the program starts, it waits for external events
  - User events: button clicks, keystrokes, etc.
  - Events from other programs: notifications, etc.
  - Most CSC207 projects are event-driven
- An event triggers a method call, like in a use case interactor's interface
- If we debug event-driven code and stop it when an event is being handled, we can look at what is on the stack to get an idea of how events work.

# EXCEPTIONS IN JAVA

CSC 207 SOFTWARE DESIGN



# LEARNING OUTCOMES

- Understand how Exceptions work in Java
- Be familiar with the Throwable hierarchy
- Know the difference between checked and unchecked exceptions — and when to use each of them

# WHAT ARE EXCEPTIONS?

- Exceptions report **exceptional conditions**: unusual, strange, unexpected.
- These conditions deserve exceptional treatment: not the usual go-to-the-next-step, plod-onwards approach.
- Therefore, understanding exceptions requires thinking about a different model of program execution.

# EXCEPTIONS IN JAVA

- To “throw an exception”:

```
throw Throwable();
```

- To “catch an exception” and deal with it:

```
try {  
    statements  
} catch (Throwable parameter) { // The catch belongs to the try.  
    statements  
}
```

- To say a method isn’t going to deal with exceptions (or may throw its own):

```
accessModifier returnType methodname(parameters) throws Throwable {  
... }
```

# ANALOGY

- throw
  - I'm in trouble, so I throw a rock through a window, with a message tied to it. It's like a return statement (exits the current method) but bad.
- try
  - Someone in the following block of code might throw rocks of various kinds. All you catchers line up ready for them.
- catch
  - If a rock of my kind comes by, I'll catch it and deal with it.
- throws
  - I'm warning you: if there's trouble, I may throw a rock.
- Even though there are only two new statement types, this changes the whole picture of how a program runs.

# WHY USE EXCEPTIONS?

- Less programmer time spent on handling errors
- Cleaner program structure:
  - isolates exceptional situations rather than sprinkling them through the code
- Separation of concerns:
  - Pay local attention to the algorithm being implemented and global attention to errors that are raised

# (AWKWARD) EXAMPLE

```
int i = 0;
int sum = 0;
try {
    while (true) {
        sum += i++;
        if (i >= 10) {
            // we're done
            throw new Exception("i at limit");
        }
    }
} catch (Exception e) {
    System.out.println("sum to 10 = " + sum);
}
```



# WHY WAS THAT CODE BAD?

- The situation that the exception reports is not exceptional.
  - It's obvious that `i` will eventually be 10. It's expected.
  - In Java, exceptions are reserved for exceptional situations.
- It's uncharacteristic. Real uses of exceptions aren't local.
  - `throw` and `catch` aren't generally in the same block of code.

# WE CAN HAVE CASCADING CATCHES

- Much like an `if` with a series of `else if` clauses, a `try` can have a series of `catch` clauses.
- After the last `catch` clause, you can have a clause:
  - `finally { ... }`
- But `finally` is not like a last `else` on an `if` statement:  
The `finally` clause is always executed, whether an exception was thrown or not, and whether or not the thrown exception was caught.
  - Example of a good use for this: close open files as a clean-up step.

# AN EXAMPLE OF MULTIPLE CATCHES

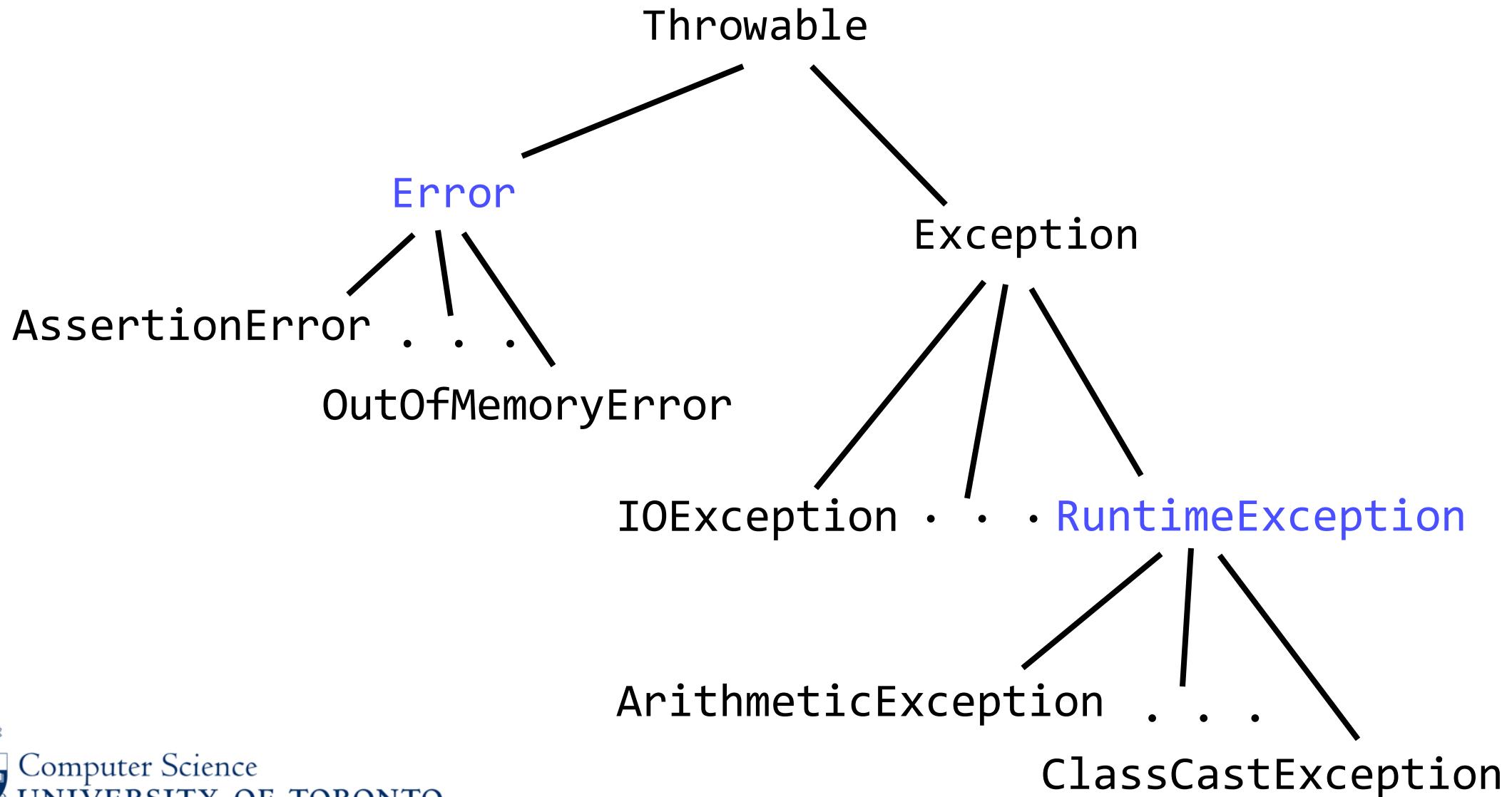
- Suppose ExSup is the parent of ExSubA and ExSubB.

```
try { ... }
catch (ExSubA e) {
    // We do this if an ExSubA is thrown.
}
catch (ExSup e) {
    // We do this if any ExSup that's not an ExSubA is thrown.
}
catch (ExSubB e) {
    // We never do this, even if an ExSubB is thrown.
}
finally {
    // We always do this, even if no exception is thrown.
}
```

# RECAP

- If you call code that may throw an exception, you have two choices:
  - you can wrap the code in a try-catch, or
  - you can use “throws” to declare that your code may throw an exception.
- Exceptions don’t follow the normal control flow.
- Some guidelines on using exceptions well:
  - Use exceptions for exceptional circumstances.
  - Throwing and catching should not be in the same method.  
“Throw low, catch high”.

# WHERE EXCEPTION FITS IN



# “THROWABLE” HAS USEFUL METHODS

- Constructors:
  - `Throwable()`, `Throwable(String message)`
- Other useful methods:
  - `getMessage()`
  - `printStackTrace()`
  - `getStackTrace()`
- You can also record (and look up) within a `Throwable` its “cause”: another `Throwable` that caused it to be thrown. Through this, you can record (and look up) a chain of exceptions.

# YOU DON'T HAVE TO HANDLE ERRORS OR RUNTIMEEXCEPTIONS

- Error:
  - “Indicates serious problems that a reasonable application should not try to catch.”
  - Do not have to handle these errors because they “are abnormal conditions that should never occur.”
- RuntimeException:
  - These are called “unchecked” because you do not have to handle them.
  - A good thing, because so many methods throw them it would be cumbersome to check them all.

# SOME THINGS NOT TO CATCH

- Don't catch Error: You can't be expected to handle these.
- Don't catch Throwable or Exception: Catch something more specific.
- (You can certainly do so when you're experimenting with exceptions. Just don't do it in real code without a good reason.)

# WHAT SHOULD YOU THROW?

- You can throw an instance of Throwable or any subclass of it (whether an already defined subclass, or a subclass you define).
- Don't throw an instance of Error or any subclass of it: These are for unrecoverable circumstances.
- Don't throw an instance of Exception: Throw something more specific.
- It's okay to throw instances of:
  - specific subclasses of Exception that are already defined, e.g., UnsupportedOperationException
  - specific subclasses of Exception that you define.

# DON'T USE THROWABLE OR ERROR

- Throwable itself, and Error and its descendants are probably not suitable for subclassing in an ordinary program.
- Throwable isn't specific enough.
- Error and its descendants describe serious, unrecoverable conditions.
  - e.g. OutOfMemoryError (a child of VirtualMachineError, which is a child of Error)

# EXTENDING EXCEPTION: VERSION 1

Example: a method `m()` that throws your own exception  
MyException, a subclass of Exception:

```
class MyException extends Exception {...}

class MyClass {
    public void m() throws MyException { ...
        if (...) throw new MyException("oops!"); ...
    }
}
```

# EXTENDING EXCEPTION: VERSION 2

Example: a method `m()` that throws your own exception `MyException`, a subclass of `Exception`:

```
class MyClass {  
    public static class MyException extends Exception {...}  
  
    public void m() throws MyException { ...  
        if (...) throw new MyException("oops!"); ...  
    }  
}
```

# ASIDE: CLASSES INSIDE OTHER CLASSES

- You can define a class inside another class. There are two kinds.
- Static nested classes use keyword `static`.  
(It can only be used with classes that are nested.)
  - Cannot access any other members of the enclosing class.
- Inner classes do not use keyword `static`.
  - Can access all members of the enclosing class  
(even private ones).
- Nested classes increase encapsulation. They make sense if you won't need to use the class outside its enclosing class.
- Reference:  
<http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>

# DOCUMENTING EXCEPTIONS

```
/**  
 * Return the mness of this object up to mlimit.  
 * @param mlimit The max mity to be checked.  
 * @return int The mness up to mlimit.  
 * @throws MyException If the local alphabet has no m.  
 */  
public void m(int mlimit) throws MyException { ...  
    if (...) throw new MyException ("oops!"); ...  
}
```

- the Javadoc comment is for human readers, and
- Keyword `throws` is for the compiler.
- Both the reader and the compiler are checking that caller and callee have consistent interfaces.

# EXTEND EXCEPTION OR RUNTIME EXCEPTION?

Recall that RuntimeExceptions are not checked. Example:

```
class MyClass {  
    public static class MyException extends RuntimeException {...}  
    public void m() /* No "throws", yet it compiles! */ { ...  
        if (...) throw new MyException("oops!"); ...  
    }  
}
```

- How do you choose whether to extend Exception or RuntimeException?
- Perhaps you should always extend Exception to benefit from the compiler's exception checking?

# WHAT DOES THE JAVA API SAY?

- Exception
  - “Class `Exception` and its subclasses are a form of `Throwable` that indicate conditions that a reasonable application might want to catch.”
- `RuntimeException` (not checked)
  - “`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.”
  - Example subclasses: `ArithmetricException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`
- non-`RuntimeException` (checked)
  - Example subclasses: `IOException`, `NoSuchMethodException`

# WHAT DOES THE JAVA LANGUAGE SPECIFICATION SAY?

“The runtime exception classes (`RuntimeException` and its subclasses) are exempted from compile-time checking because, in the judgment of the designers of the Java programming language, having to declare such exceptions **would not aid significantly in establishing the correctness** of programs.”

“The information available to a compiler, and the level of analysis the compiler performs, are usually not sufficient to establish that such run-time exceptions cannot occur, even though this may be obvious to the programmer. Requiring such exception classes to be declared **would simply be an irritation to programmers.**”

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html>

# EXAMPLE

- Imagine code that implements a circular linked list.
- Suppose that, by construction, this structure can never involve null references.
- The programmer can then be certain that a `NullPointerException` cannot occur.
- But it would be difficult for a compiler to prove it!
- So if `NullPointerException` were checked, the programmer would have to handle them (catch or declare that one might be thrown) ***everywhere*** a `NullPointerException` might occur.

# GOOD ADVICE FROM JOSHUA BLOCH

- "Use checked exceptions for conditions from which the caller can reasonably be expected to recover."
- "Use run-time exceptions to indicate programming errors. The great majority of run-time exceptions indicate precondition violations."
  - I.e., if the programmer could have predicted the exception, don't make it checked.
  - Example: Suppose method `getItem(int i)` returns an item at a particular index in a collection and requires that `i` be in some valid range.
  - The programmer can check that range *before* they call `o.getItem(x)`.
  - Passing an invalid index should not cause a checked exception to be thrown in this case.
- "Avoid unnecessary use of checked exceptions."
  - If the user didn't use the API properly or if there is nothing to be done, then make it a `RuntimeException`.

# TESTING IN JAVA

CSC 207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO



# LEARNING OUTCOMES

Know the basics of how unit testing works in Java using JUnit

Understand the benefits of test-driven development

# UNIT TESTING

- The goal is to fully test each unit — a single behavioural concept.
- In Java, a unit is (often) a method. Call each method at least once. If the behaviour of the method varies with different circumstances, then testing each circumstance is necessary.
- It might make sense to sometimes think of a unit at a higher level, such as a **use case** being the unit we want to test.

# ASSERTION

- **Single-Outcome Assertions:**

- `fail`; OR `fail(msg)`;

- **Stated Outcome Assertions:**

- `assertNotNull(object)`; OR `assertNotNull(msg, object)`;  
`assertTrue(booleanEx)`; OR `assertTrue(msg, booleanEx)`;

- **Equality Assertions**

- `assertEquals(exp, act)`; OR **`assertEqual(msg, exp, act)`**;

- **Fuzzy Equality Assertions**

- `assertEquals(msg, expected, actual, tolerance)`;

# POSSIBLE RESULTS

- **pass**: test produced the expected outcome
- **fail**: test ran but produced an incorrect outcome
- **error**: test ran but produced an incorrect behaviour (i.e., it threw an exception that was unexpected)

# UNIT TESTING

- Unit testing follows a pattern
  - Lots of small, **independent** tests
  - Reports passes, failures, and errors
  - Some optional setup and teardown shared across tests
  - Aggregation (combine tests into test suites)
- We could accomplish all of this “by hand”, but this common structure inspired the development of JUnit:
  - When you see a pattern, build a framework
  - Write shared code once
  - Make it easy for people to do things the right way

# SETUP AND TEARDOWN

- There are three steps in running a test: **setup**, **run**, and **teardown**
- The **setup** phase is in a single method annotated with `@Before`
- The **teardown** phase is in a single method annotated with `@After`
- These run before and after every test method.
- The methods annotated with `@BeforeClass` run once before all test methods in that test class are executed, and those methods annotated with `@AfterClass` run once after.
- The setup and teardown methods are used to avoid repetition. For example, to create/destroy data structures required for more than one test method.

# USING JUNIT IN INTELLIJ

- Define the method signatures for the class to be tested.
- Select the class.
- Have IntelliJ create JUnit tests.
- Replace the dummy method bodies with real ones.
- Add more test cases.
- (Now, write your code.)
- **IntelliJ can produce a report of the test coverage — showing how much of the program is tested and which lines aren't covered!**

# SELECTING TEST CASES

- Test for success
  - General cases, well-formatted input, boundary cases
  - Classics:
    - 0, 1, more
    - odd, even
    - beginning, middle, end
- Check for data structure consistency (representation invariants)
- Test for atypical behaviour
  - Does it handle invalid input (if required)?
  - Does it throw the exceptions it is supposed to?

# TESTING GUIDELINES

- Have at least one test class per class being tested.
- Have at least one test method per method being tested.
- More only if there are multiple test cases.
  - Name your test methods `testMethodNameDescription`
  - Use annotations (e.g., `@Test`, `@Before`, `@After`, ...).
  - Document your test cases.
  - Avoid duplicate test cases.

# DESIGN FOR TESTABILITY

- When you are writing code, think about what you need to test and how you can test it.
  - Write methods that do a single task.
  - **Separate input, computation, and output when possible.**
  - Modularity, modularity, modularity.
- Don't delay writing tests! Write tests before you write code as part of the requirements stage and update those tests as or after you write code.

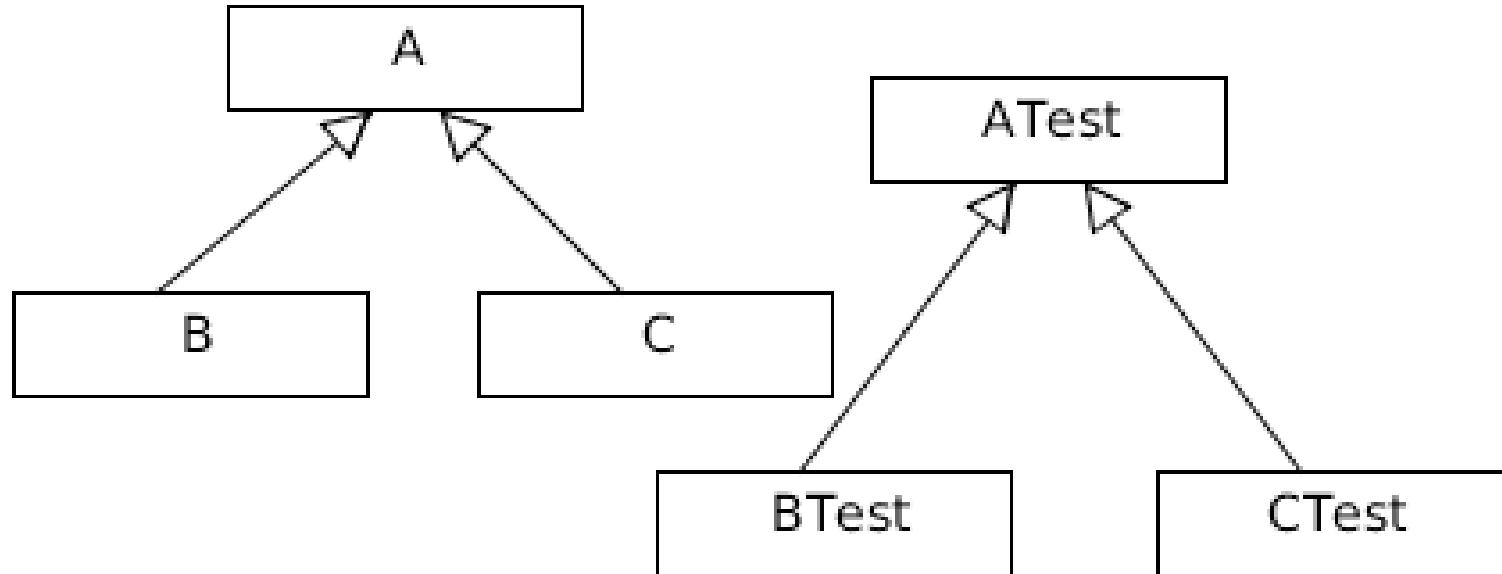
# TESTING CODE WITH EXCEPTIONS

```
@Test(expected=IndexOutOfBoundsException.class)  
public void testIndexOutOfBoundsException() {  
    ArrayList emptyList = new ArrayList();  
    Object o = emptyList.get(0);  
}
```

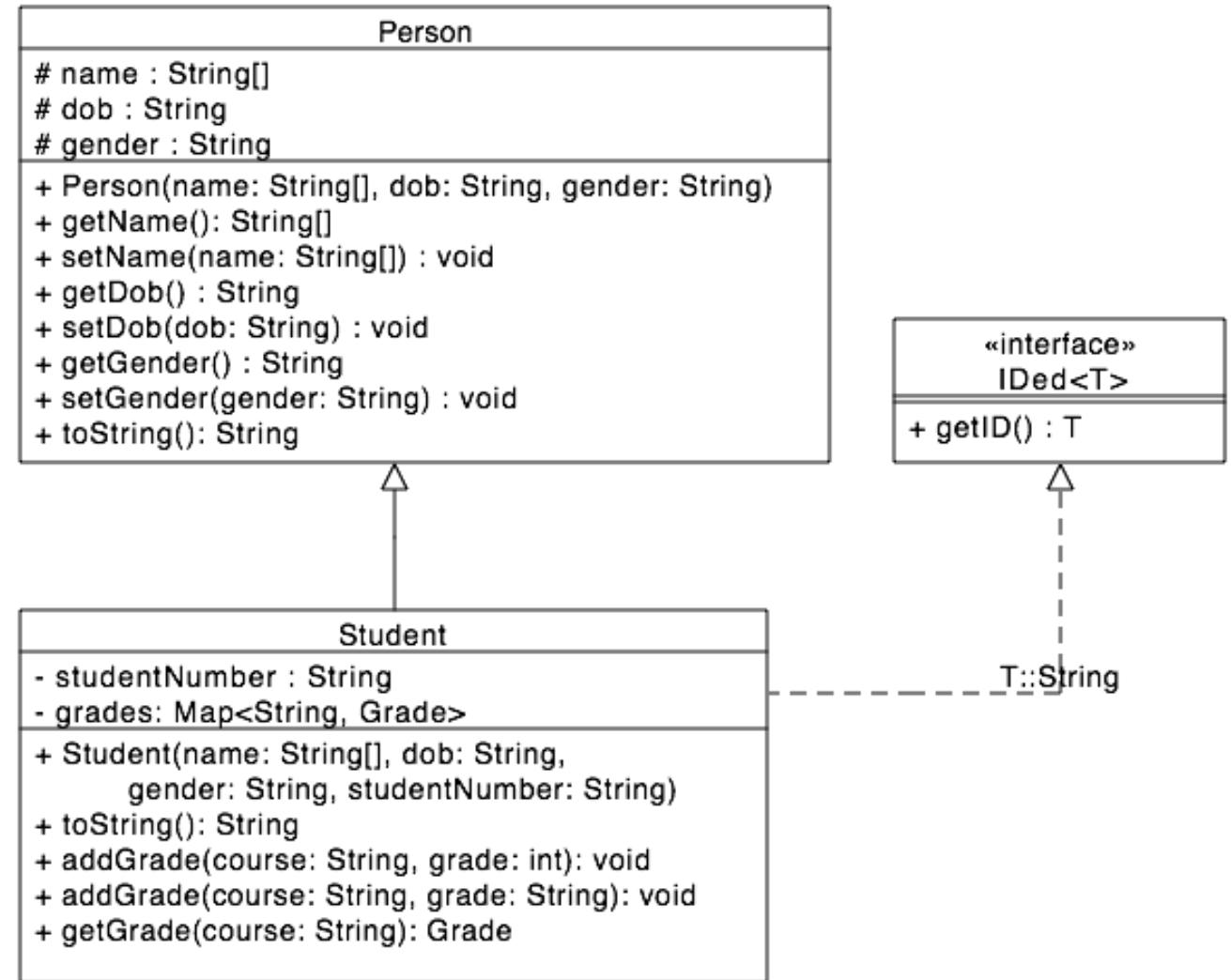
# TESTING CODE WITH EXCEPTIONS (ALTERNATIVE)

```
@Test  
  
public void testIndexOutOfBoundsException() {  
    ArrayList emptyList = new ArrayList();  
    try {  
        Object o = emptyList.get(0);  
        fail("IndexOutOfBoundsException not thrown when " +  
             "trying to access the contents of an empty list.");  
    } catch (IndexOutOfBoundsException e) {  
    }  
}
```

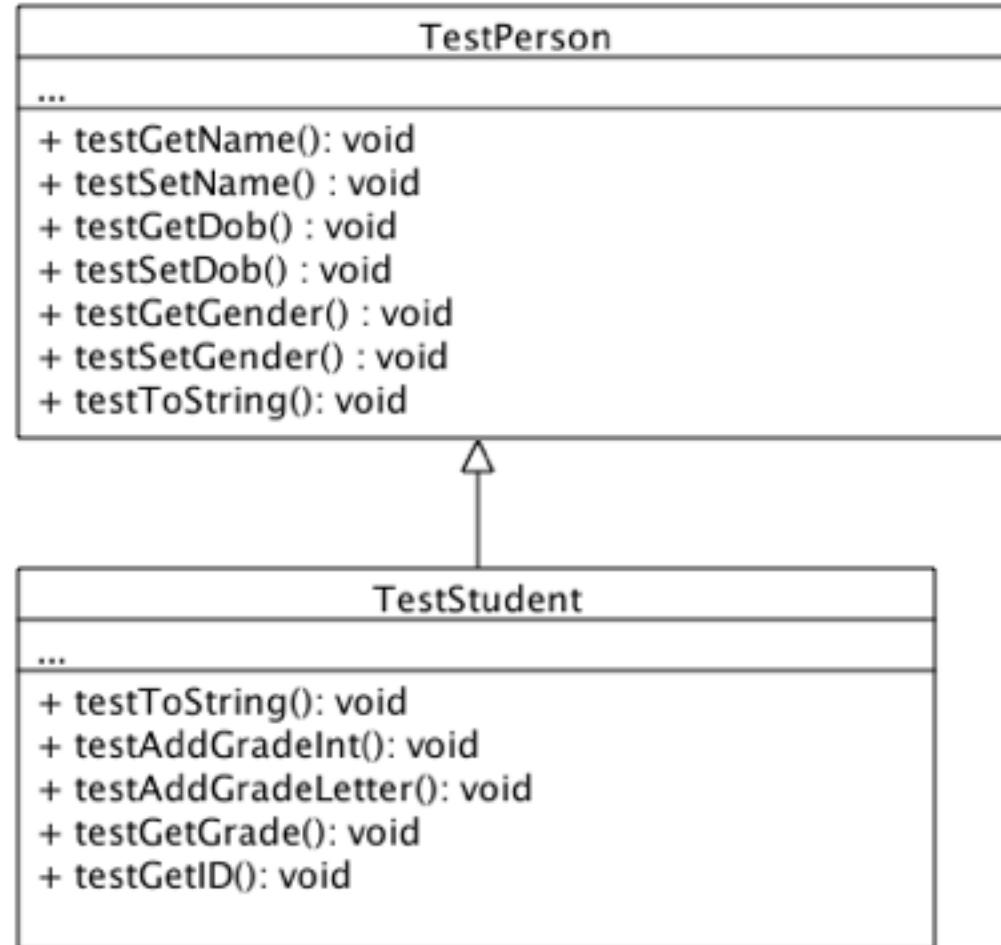
# TESTING CODE WITH INHERITANCE



# EXAMPLE: INHERITANCE



# EXAMPLE: TESTING



# TEST-DRIVEN DEVELOPMENT

- Try writing your tests first!
- Then your tests:
  - are based on requirements rather than code.
  - determine the code you need to write.
- Later, if you think of a situation that your code doesn't handle, add a test for it!
- This approach aids in the definition of requirements.
- It provides tangible evidence of progress.

# AN EXAMPLE

When testing your project code, it can help to think of testing at the use case level.

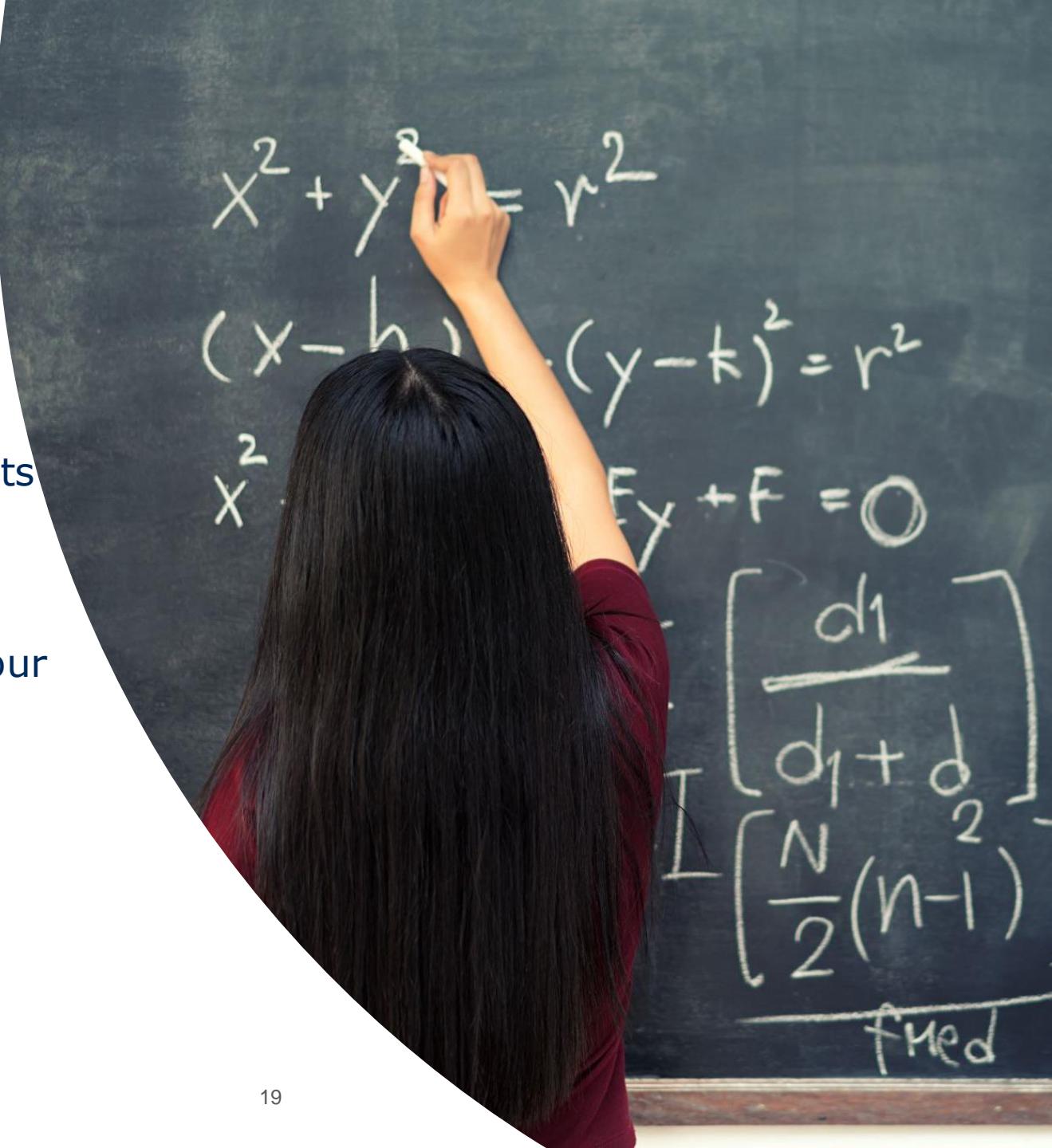
How a test might look for testing our "user registers a new account" use case:

[https://github.com/paulgries/UserLoginCleanArchitecture/blob/main/test/user\\_register\\_use\\_case/UserRegisterInteractorTest.java](https://github.com/paulgries/UserLoginCleanArchitecture/blob/main/test/user_register_use_case/UserRegisterInteractorTest.java)

# HOMEWORK

---

- Go back and look at some of the junit tests we provided in your Java exercises if you didn't look at them before.
- Start thinking about what kind of tests your team will need to write for your project!
- Explore <https://github.com/junit-team/junit4> and the documentation on junit4 linked from there to learn more as you start writing tests.



# PACKAGES

CSC 207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

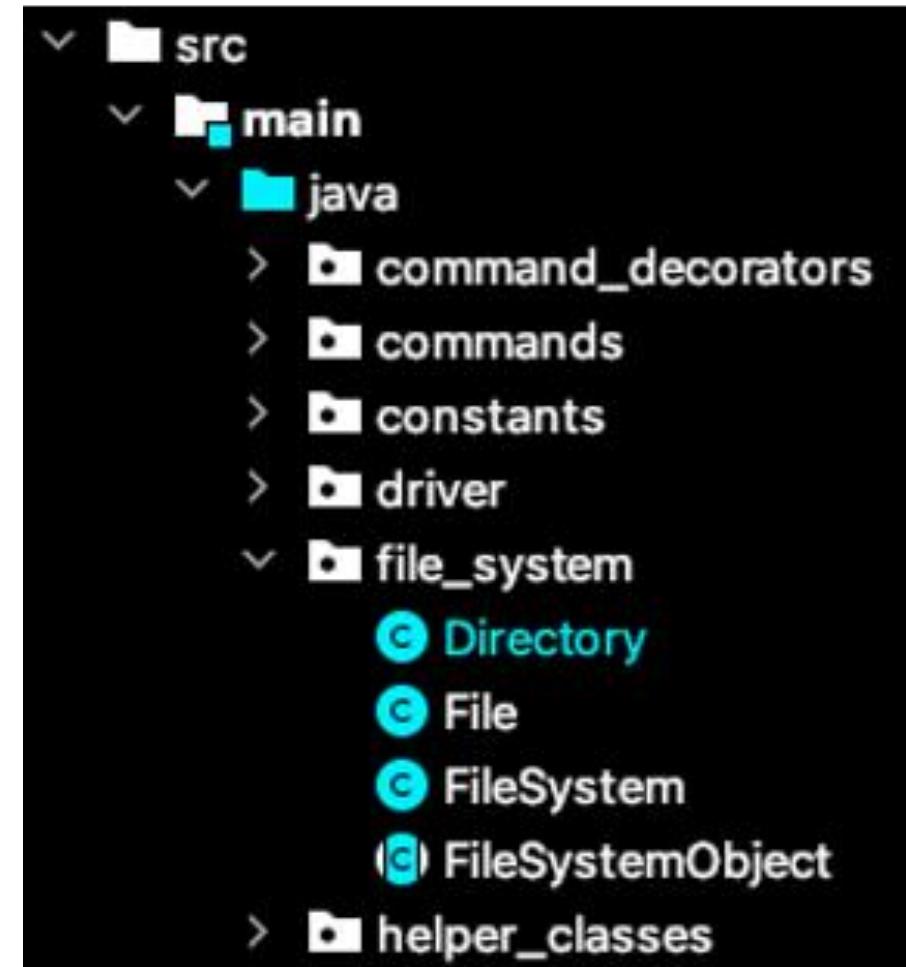
# LEARNING OUTCOMES

- Understand various ways you can organize your code in packages

# PACKAGES IN JAVA

- Package: a folder
  - Contains related classes and packages
  - The full name of a class includes the package name(s): java.util.ArrayList
- When you get to a dozen or more classes, package organization is essential!

## JShell packages

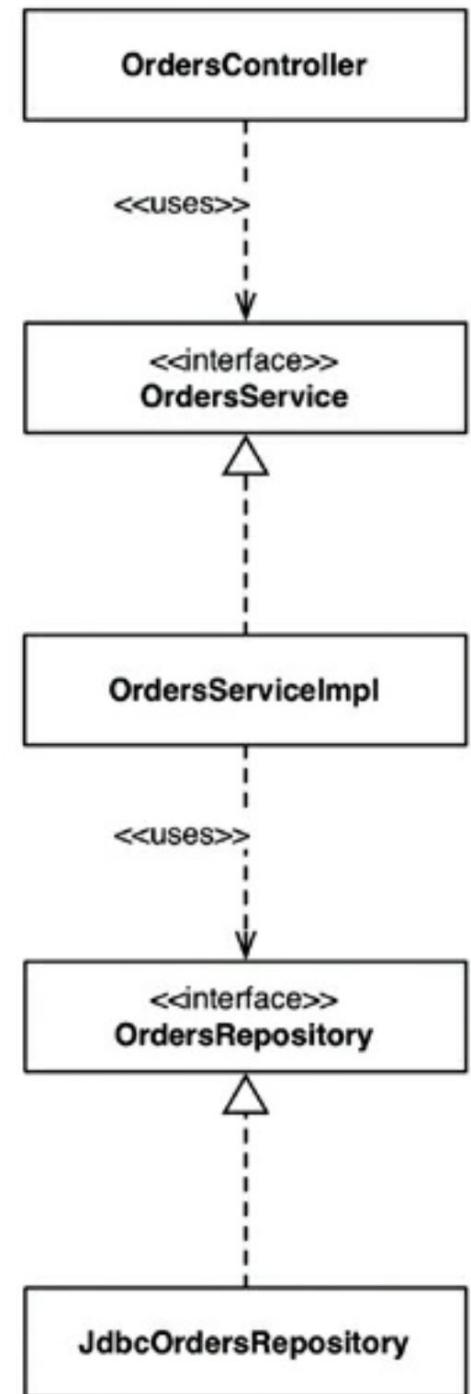


# EXAMPLE: ONLINE BOOK STORE

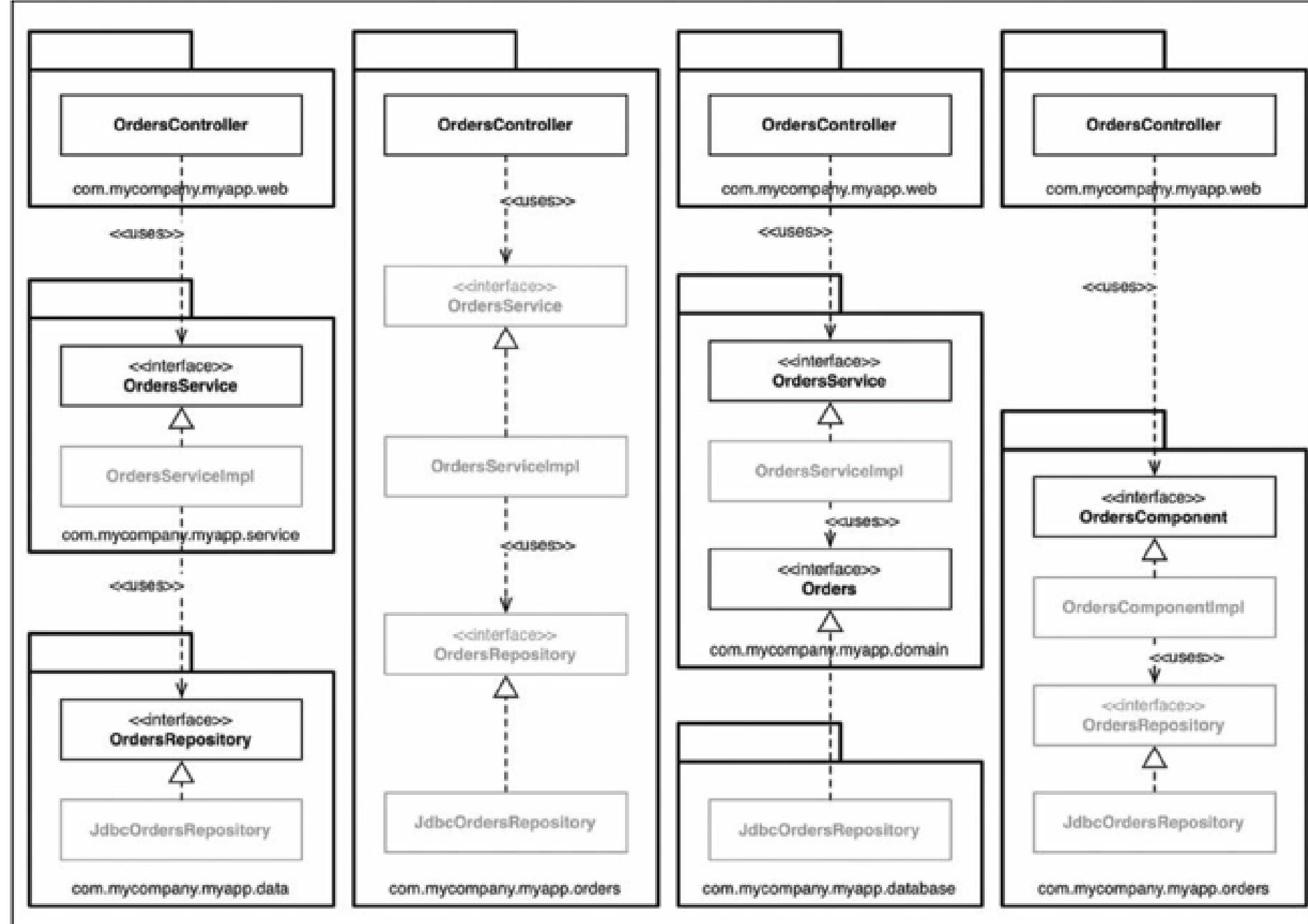
- When building an online book store, one of the use cases we've been asked to implement is about customers being able to view the status of their orders.

# HOW COULD YOU PACKAGE THIS CODE?

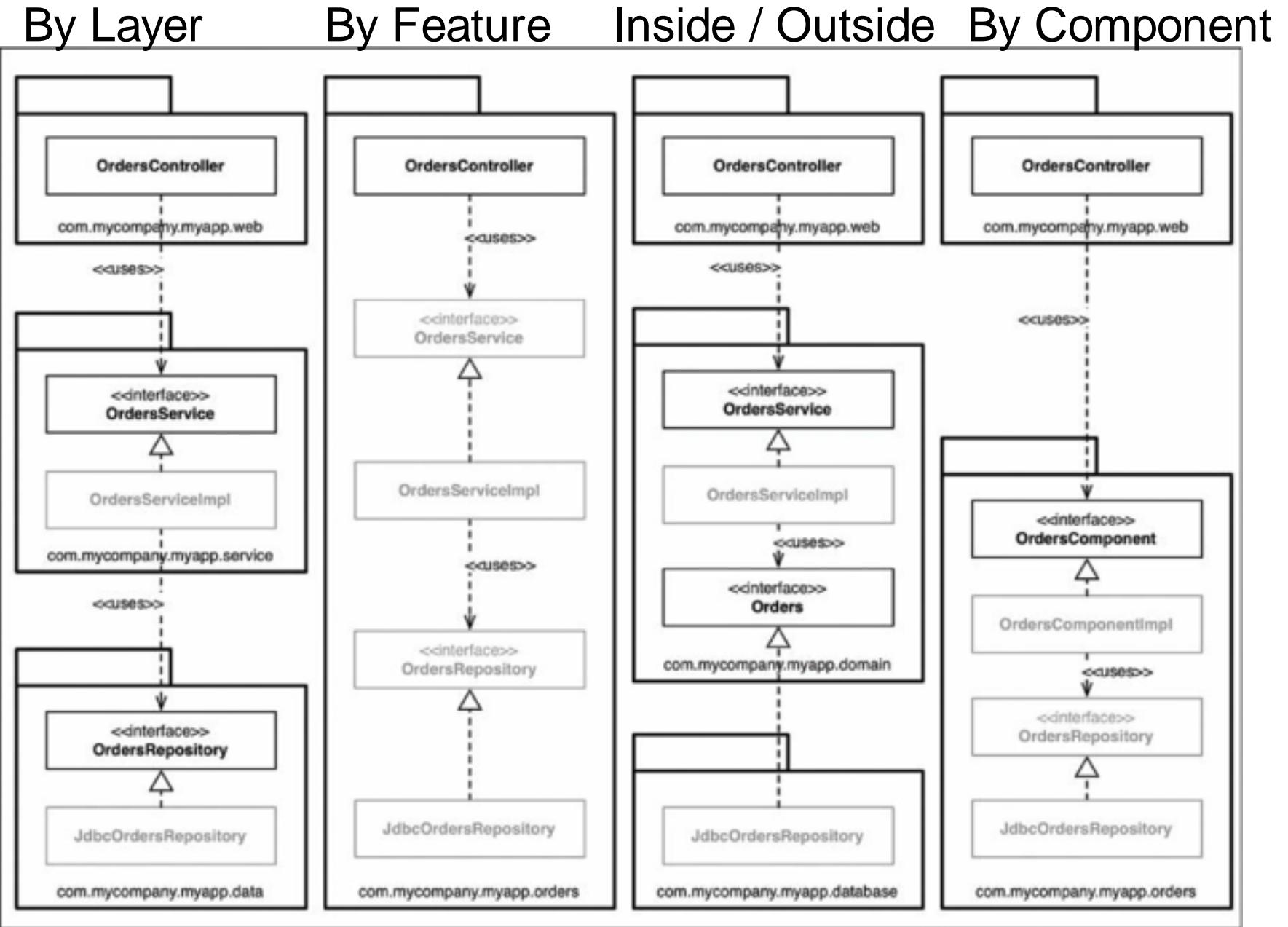
- **OrdersController**: A web controller, something that handles requests from the web
- **OrdersService**: An interface that defines the “business logic” related to orders.
- **OrdersServiceImpl**: The implementation of the orders service.
- **OrdersRepository**: An interface that defines how we get access to persistent order information.
- **JdbcOrdersRepository**: An implementation of the repository interface that saves information in a database.



**Figure 34.8**  
**Chapter 34**  
**Clean Architecture**



**Figure 34.8**  
**Chapter 34**  
**Clean Architecture**



# WHAT'S A COMPONENT?

- An individual software component is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data).
  - [https://en.wikipedia.org/wiki/Component-based\\_software\\_engineering#Definition\\_and\\_characteristics\\_of\\_components](https://en.wikipedia.org/wiki/Component-based_software_engineering#Definition_and_characteristics_of_components)
- Orders is a component in the current example. What might the components be in your course project?

# CONCLUSIONS FROM CHAPTER 34 FROM CLEAN ARCHITECTURE TEXTBOOK

“Think about how to map your desired design onto code structures, how to organize that code, and which decoupling modes to apply”

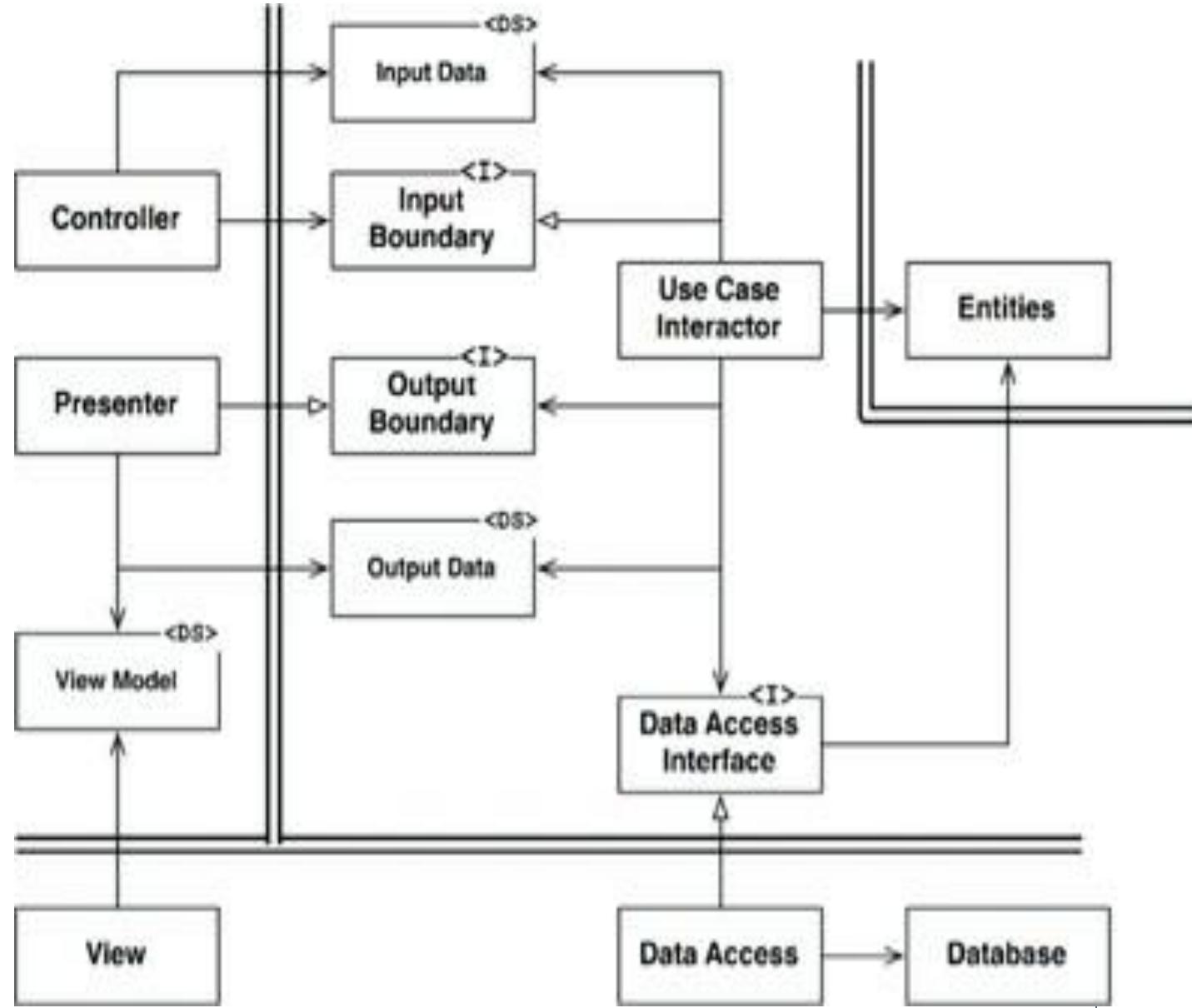
Packages can potentially provide both organization and encapsulation (using access modifiers)

For the project, your team needs to choose a package structure ... and that takes practice!

- IntelliJ will automatically support you as you move files between packages!
- Choose your access modifiers to reflect your design, exposing only the necessary methods.
- Recommendation: "package by use case" (see following slides)

# A TYPICAL SCENARIO

- UI gets input data, hands to the Controller.
- Controller packages that data into a plain old Java object, passes it through the InputBoundary to the UseCaseInteractor.
- UseCaseInteractor interprets that data, uses it to control the dance of the Entities.
- When complete, UseCaseInteractor gathers data from the Entities, constructs the OutputData as another plain old Java object.
- OutputData is passed through the OutputBoundary interface to the Presenter.



# PROJECT: USE CASES

- Each use case should be independent from the other use cases
  - No shared code in the use case layer
  - Entities may be shared
- Easy way to enforce this: Inside / Outside packaging
  - A use case package includes the interactor, its input and output boundaries, the input and output data, and gateway
  - Things outside may be shared (for example, entities, and also the frameworks & drivers layer)
- If you use this packaging strategy on your project, then each person "owns" their use case package
- This is what the login example uses

# HOMEWORK

---

- Start thinking about what packages your project code might have!



# SERIALIZATION & PERSISTENT DATA

CSC 207 SOFTWARE DESIGN

# LEARNING OUTCOMES

Understand what serialization is and some of the ways it can be implemented in Java.

# WHAT IS A SERIALIZATION?

<https://stackoverflow.com/questions/633402/what-is-serialization>

Converting an object into a format in which it can be:

- stored,
- transferred, and
- reconstructed (deserialized).

Can allow for data to persist between runs of a program.

More specifically, you may see it refer to conversion of the object to a sequence of bytes.

# SERIALIZATION IN JAVA

Java provides the Serializable interface

Any class implementing it can be serialized!

Easy to use

Can save to a file (often use a .ser file extension, by convention)

Each attribute of the object must implement Serializable too

Make attributes transient to avoid serializing them

A decent discussion of pros and cons of using Serializable in Java

<https://softwareengineering.stackexchange.com/questions/191269/java-serialization-advantages-and-disadvantages-use-or-avoid>

# SERIALIZATION IN JAVA

Alternatives:

Save to a...

txt file (custom format to represent your object)

csv file (standard format, but may not be appropriate for your objects)

json file (standard format, more expressive than csv)

xml file (another standard, expressive format)

database (e.g. you might save serialized objects or attributes of the object)

# HOW DATA PERSISTS IS A DETAIL

- The specifics of *how* we save and load data for our program should be in the outer layer of our design.
- Design an interface for what it means to save/load, then we are free to implement it using any of the approaches mentioned.

# RESOURCES

Minimal example of serializing an object

- <https://github.com/CSC207-UofT/CleanArchLoginSample>

J-Shell commit where Paul switched to using Serializable

- <https://github.com/CSC207-UofT/Java-Shell/commit/3fdaf05c2a2529a8f20ed4497a595ca9e774d356>

A tutorial covering the basics of Serializable

- <https://www.baeldung.com/java-serialization>

A tutorial about reading / writing files in Java

- <https://docs.oracle.com/javase/tutorial/essential/io/file.html>

# HOMEWORK

---

- Start thinking about how you can serialize your project code and what data needs to be persistent.



A photograph of a refrigerator filled with food, including plastic containers of various colors (blue, red, pink, yellow) and bags of vegetables like lettuce and cabbage. The word "CODE SMELLS" is overlaid in large white capital letters across the middle of the image.

# CODE SMELLS

CSC 207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

Understand what a code smell is and some common categories of code smells.

Understand common ways to address code smells through refactoring.



# WHAT IS A CODE SMELL?

These slides are based on

<https://refactoring.guru/refactoring/smells>



# WHAT IS A CODE SMELL?

- When the refrigerator smells, the food might not have gone bad yet. But it will soon!
- When there is a code smell, the program isn't bad yet. But the more code you add, the more it will have to accommodate the "smelly" code, causing your program to be more and more difficult to handle.
- Some problems can be fixed any time:
  - an inconvenient variable name can be changed using "search and replace"
  - the file path leading to a csv file where you store data can be changed at any time
- Code smells should be fixed as soon as you see them, to avoid extra work later.



# CATEGORIES OF CODE SMELLS

- Bloaters – too much code
- Object Orientation Abusers – can be improved by changing use of inheritance, composition, or by redistributing responsibilities
- Change Preventers – features that make it difficult to extend or update the code
- Dispensables – things that can be deleted or combined
- Couplers – the opposite of "lessening dependencies"
- etc

# EXAMPLES OF A PROBLEM AND ITS SOLUTION

- <https://refactoring.guru/smells/long-method>
- Long Method
- Signs and Symptoms: "A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions."
- Reasons for the Problem: "...Mentally, it's often harder to create a new method than to add to an existing one: "But it's just two lines, there's no use in creating a whole method just for that..." Which means that another line is added and then yet another, giving birth to a tangle of spaghetti code..."
- Possible Fixes:
  - [Extract Method](#) to make the original method shorter
  - [Replace Temp with Query](#), [Introduce Parameter Object](#) or [Preserve Whole Object](#), if variables or parameters make it difficult to extract method.
  - [Replace Method with Method Object](#) to encapsulate the method, if you cannot make it smaller.
  - If an if statement prevents such a move, use [Decompose Conditional](#). If a loop prevents the move, use [Extract Method](#).

# NOTES ABOUT SMELLS – PART 1

- Switch statements are only a problem if they are complicated or use "instanceof"
- "Large class" can mean a class that is inconveniently long to read and work with. But it can also mean a class that breaks the Single Responsibility Principle.
- "Alternative Class with Different Interfaces" comes up when different people solve the same problem independently in different parts of the code. This can be prevented or fixed quickly if you present your code to each other and compare others' code to your own.

# NOTES ABOUT SMELLS – PART 2

- "Divergent Change" and "Shotgun Surgery" sound similar, but are not the same problem
  - Divergent Change means...
  - Shotgun Surgery means...
- "Parallel Inheritance Hierarchies" are not just the existence of two inheritance hierarchies that look similar. To have this code smell, an addition of one class in one hierarchy forces a new class be added to the other inheritance hierarchy. Example: each view class corresponds to a different presenter class
- Isn't an Input Data Object a Data Class?

# NOTES ABOUT SMELLS – PART 3

- How do I know if a class is "lazy"?
- Isn't "Speculative Generality" the same as "I haven't finished my program yet"?
- "Duplicate code" can easily be avoided by not copying and pasting code (copy-pasta).
- Aren't "Message Chains" just methods calling other methods?
  - No.
  - Example: obj1.method1().method2().method3().method4();  
Which object contains method4? What if you want to refactor it?

# REFACTORING

- <https://refactoring.guru/refactoring>
- "Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design."
- <https://refactoring.guru/refactoring/what-is-refactoring>
- Clean Code is well formatted, well documented, contains good variable names, does not contain duplicate code, has nice small methods and reasonably sized classes, no magic numbers, and passes all its tests.

# REFACTORING AND THE OPEN/CLOSED PRINCIPLE

- The Open/Closed Principle says that we should write code in a way that lets us change as few things as possible when adding new features.
- Refactoring amounts to making changes to the code. Doesn't that go against the principle?
- We use refactoring now to get the code to a point where we can avoid making too many changes when we add new features later.

# WHEN TO REFACTOR?

- Whenever you see a place where your code could be improved, or when you identify a code smell.
- When you find yourself doing the same thing for the third time.
- When trying to understand someone else's code, but it is confusing.
- When trying to fix a bug, but the code is difficult to follow.
- During or immediately after a code review, when conducted with at least one of the authors of the code.
- After receiving feedback from your TA.

# ADDING NEW FEATURES ≠ REFACTORING

- When you refactor, the functionality of your code should not change.
- Refactoring should change the readability and understandability of your code.
- If you refactor before adding new features, it should be easier to add the new features.
- Working "smart" now can save you hard work later.
- The goal is to make it easier to grow your program and save time for you and your team.

# CATEGORIES OF CODE SMELL FIXES

The solution to a code smell is almost always "refactor".

<https://refactoring.guru/refactoring/techniques>

- Composing Methods
- Moving Features Between Objects
- Organizing Data
- Simplifying Conditional Expressions
- Simplifying Method Calls
- Dealing With Generalization

# COMPOSING METHODS EXAMPLE

- "Replace Temp With Query"
- <https://refactoring.guru/replace-temp-with-query>
- You can encapsulate a temporary variable assignment by putting it in a separate method and querying that method.

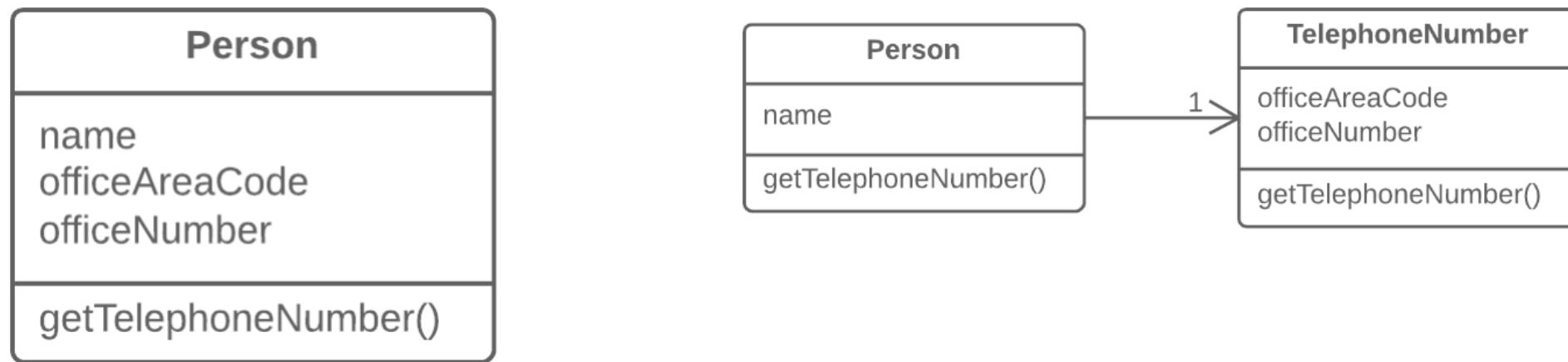
```
double calculateTotal() {  
    double basePrice = quantity * itemPrice;  
    if (basePrice > 1000) {  
        return basePrice * 0.95;  
    }  
    else {  
        return basePrice * 0.98;  
    }  
}
```

```
double calculateTotal() {  
    if (basePrice() > 1000) {  
        return basePrice() * 0.95;  
    }  
    else {  
        return basePrice() * 0.98;  
    }  
}  
double basePrice() {  
    return quantity * itemPrice;  
}
```



# MOVING FEATURES BETWEEN OBJECTS EXAMPLE

- "Extract Class"
- <https://refactoring.guru/extract-class>
- When a class does too many things, you can take a subset of the functionality that belongs together to create a new class. The new class can become a variable inside the original.





# ORGANIZING DATA EXAMPLE

- Replace Magic Number with Symbolic Constant
- <https://refactoring.guru/replace-magic-number-with-symbolic-constant>
- Try not to hardcode values in your methods. It is easier to find and change their values later, if they occur as constants.

```
double potentialEnergy(double mass, double height) {  
    return mass * height * 9.81;  
}
```

```
static final double GRAVITATIONAL_CONSTANT = 9.81;  
  
double potentialEnergy(double mass, double height) {  
    return mass * height * GRAVITATIONAL_CONSTANT;  
}
```

# SIMPLIFYING CONDITIONAL EXPRESSIONS

## EXAMPLE

- Replace Conditional with Polymorphism
- <https://refactoring.guru/replace-conditional-with-polymorphism>
- If statements or switch statements based on "instanceof" make extension difficult.

We saw an example of this in the solid.pdf slides for the Open/Closed principle, with the AreaCalculator class requiring a list of Rectangles. We created a Shape super class and relocated the "area" method inside each subclass of Shape.

```
class Bird {  
    // ...  
    double getSpeed() {  
        switch (type) {  
            case EUROPEAN:  
                return getBaseSpeed();  
            case AFRICAN:  
                return getBaseSpeed() - getLoadFactor() * number_of_coconuts;  
            case NORWEGIAN_BLUE:  
                return (isNailed) ? 0 : getBaseSpeed(voltage);  
        }  
        throw new RuntimeException("Should be unreachable");  
    }  
}
```

```
abstract class Bird {  
    // ...  
    abstract double getSpeed();  
}  
  
class European extends Bird {  
    double getSpeed() {  
        return getBaseSpeed();  
    }  
}  
class African extends Bird {  
    double getSpeed() {  
        return getBaseSpeed() - getLoadFactor() * number_of_coconuts;  
    }  
}  
class NorwegianBlue extends Bird {  
    double getSpeed() {  
        return (isNailed) ? 0 : getBaseSpeed(voltage);  
    }  
}  
  
// Somewhere in client code  
speed = bird.getSpeed();
```

# SIMPLIFYING METHOD CALLS EXAMPLE

- Preserve Whole Object
- <https://refactoring.guru/preserve-whole-object>
- If you must call multiple getters from an object and then pass those values on as arguments, don't. Just pass the original object.
- This sometimes conflicts with Clean Architecture. For this reason, we can have Input Data Objects and Output Data Objects.

```
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```
boolean withinPlan = plan.withinRange(daysTempRange);
```

# DEALING WITH GENERALIZATION EXAMPLES

- Replace Inheritance with Delegation
  - <https://refactoring.guru/replace-inheritance-with-delegation>
- Replace Delegation with Inheritance
  - <https://refactoring.guru/replace-delegation-with-inheritance>
- If a subclass S does not use all of the inherited methods from its super class T, then S can contain a variable of type T instead.
- If a class S has nothing but simple methods that call methods from variable T, then you can consider making T a superclass of S.
- Do not do the previous refactoring if it makes more sense to have a Façade, or if there is no "is-a" relationship between S and T.

# HOMEWORK

---

- As you write code for your course project and beyond this course, keep an eye out for code smells!
- Complete the Quercus quiz about Code Smells.





# LEARNING TO LISTEN FOR DESIGN

CSC 207 SOFTWARE DESIGN

Based on the paper of the same name by:  
Elisa Baniassad, Ivan Beschastnikh, Reid Holmes, Gregor Kiczales,  
and Meghan Allen

<https://dl.acm.org/doi/10.1145/3359591.3359738>



# LEARNING OUTCOMES

Gain an understanding of the connection between code smells, refactoring, and design patterns.

Understand what it means to “listen to code”.

# LEARNING TO LISTEN FOR DESIGN

## ABSTRACT

- In his essay, *Designed as Designer*, Richard Gabriel suggests that artifacts are agents of their own design. Building on Gabriel's position, this essay makes three observations  
**(1)** Code “speaks” to the programmer through code smells, and it talks about the shape it wants to take by signaling design principle violations. By “listening” to code, even a novice programmer can let the code itself signal its own emergent natural structure. **(2)** Seasoned programmers listen for code smells, but they hear in the language of design principles **(3)** Design patterns are emergent structures that naturally arise from designers listening to what the code is signaling and then responding to these signals through refactoring transformations. Rather than seeing design patterns as an educational destination, we see them as a vehicle for teaching the skill of listening. By showing novices the stories of listening to code and unfolding design patterns (starting from code smells, through refactorings, to arrive at principled structure), we can open up the possibility of listening for emergent design.

# TERMINOLOGY

- Code Smell
- Design Pattern
- Emergent Structure / Emergent Design
- Design Principle
- Refactoring

# "LISTENING" TO THE CODE

(1) Code “speaks” to the programmer through code smells, and it talks about the shape it wants to take by signalling design principle violations. By “listening” to code, even a novice programmer can let the code itself signal its own emergent natural structure.

- What does it mean to "listen to the code"?
- What do we mean by "emergent natural structure"?
- How does this relate to the anti-pattern of a design pattern?

# USING SOLID TO FIX CODE SMELLS

(2) Seasoned programmers listen for code smells, but they hear in the language of design principles

- How do we choose which fix to use when eliminating a code smell?
- How do the SOLID principles help prevent code smells?
- We can see a code smell as an inconvenience that must be eliminated. We can also see a code smell as a SOLID violation. What are the pros and cons (if there are any) of these two views?

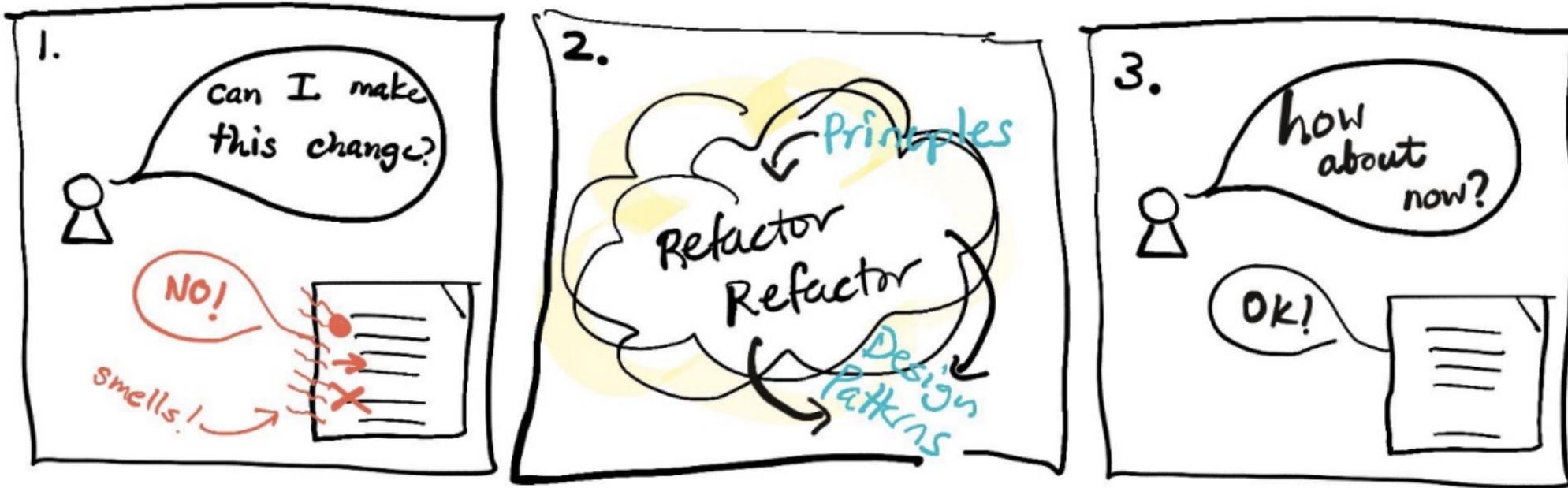
# DISCOVERING DESIGN PATTERNS

(3) Design patterns are emergent structures that naturally arise from designers listening to what the code is signaling and then responding to these signals through refactoring transformations.

- Compare and contrast this definition with our previous definition of design patterns: "generally accepted solutions to common problems in OOD (Object Oriented Design)".
- Why do we need to learn the "anti-pattern" to each design pattern?
- How do anti-patterns relate to "listening to what the code is signaling"?

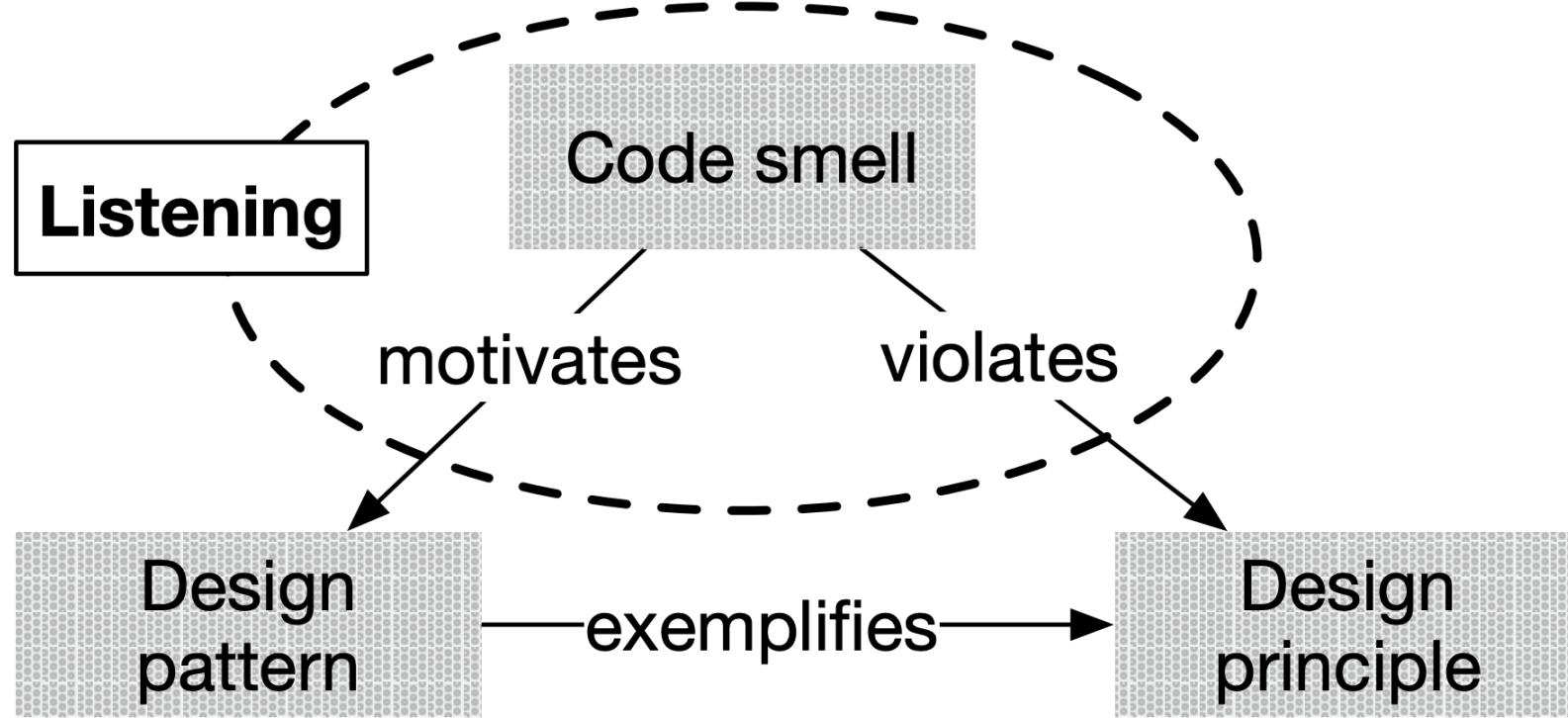
# INTUITION

- It would be helpful to have intuition about what features of the code should be refactored. How can we build this intuition?
- In this course we talk about:
  - the Java naming conventions,
  - packaging,
  - identifying code smells,
  - design patterns and their anti-patterns,
- Read and write lots of code! Make mistakes and participate in code reviews!
- What else can we use to build our intuition and "listen" when the code "speaks"?



**Figure 1.** Seasoned programmer's conversation with code.





**Figure 2.** Integration of code smells, design patterns, and design principles into a single conceptual structure.



# LISTENING AS A TEAM

- Group projects in software engineering courses teach students to respect what others on their team may be hearing the code say, and to work collaboratively.
- Different programmers will hear different smells and will bring individualised perspective and wisdom, rooted in principled understanding, to their interpretation of what the code is telling them.
- By listening in concert, a team can hear the code more clearly.
- As Gabriel writes, “Conceptual integrity arises not (simply) from one mind or from a small number of agreeing resonant minds, but from sometimes hidden co-authors and the thing designed itself.”

# HOMEWORK

---

- Listen to your team's code as you begin writing code for your project.





# **UML & DESIGN PATTERNS**

CSC 207 SOFTWARE DESIGN



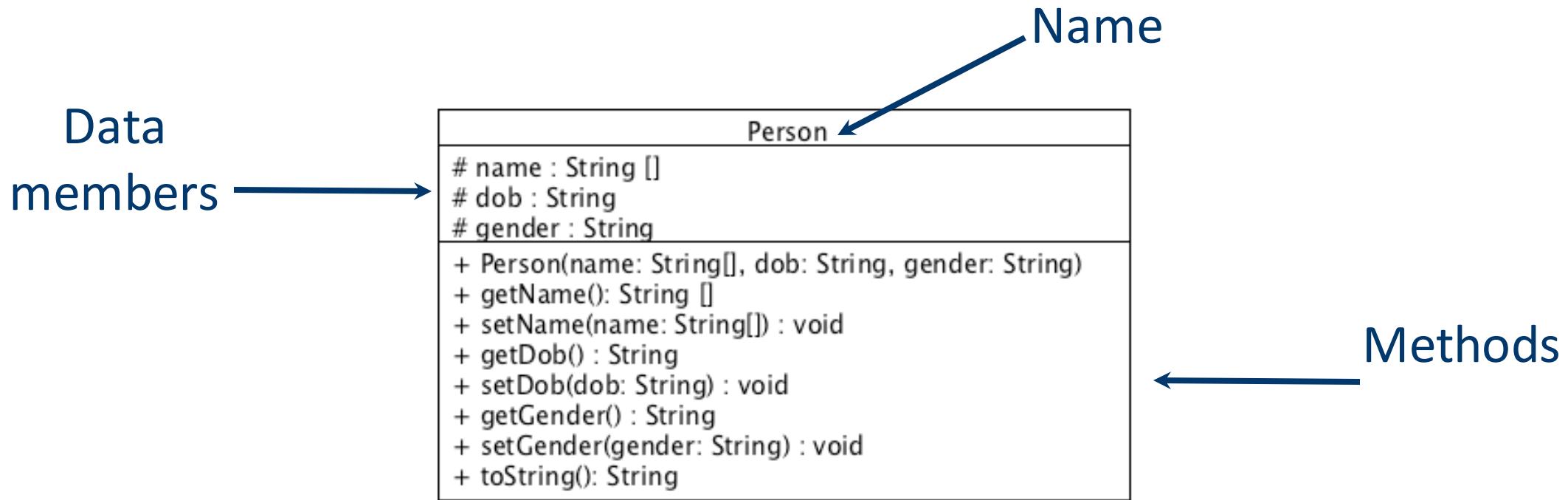
# LEARNING OUTCOMES

- Understand the basics of UML and appreciate why it is useful for conveying design patterns
- Know what a design pattern is

# UML

- Unified Modeling Language (UML)
- A way to draw information about software, including how parts of a program interact.
- We'll use only a small part of the language, [Class Diagrams](#), to represent basic OO design.

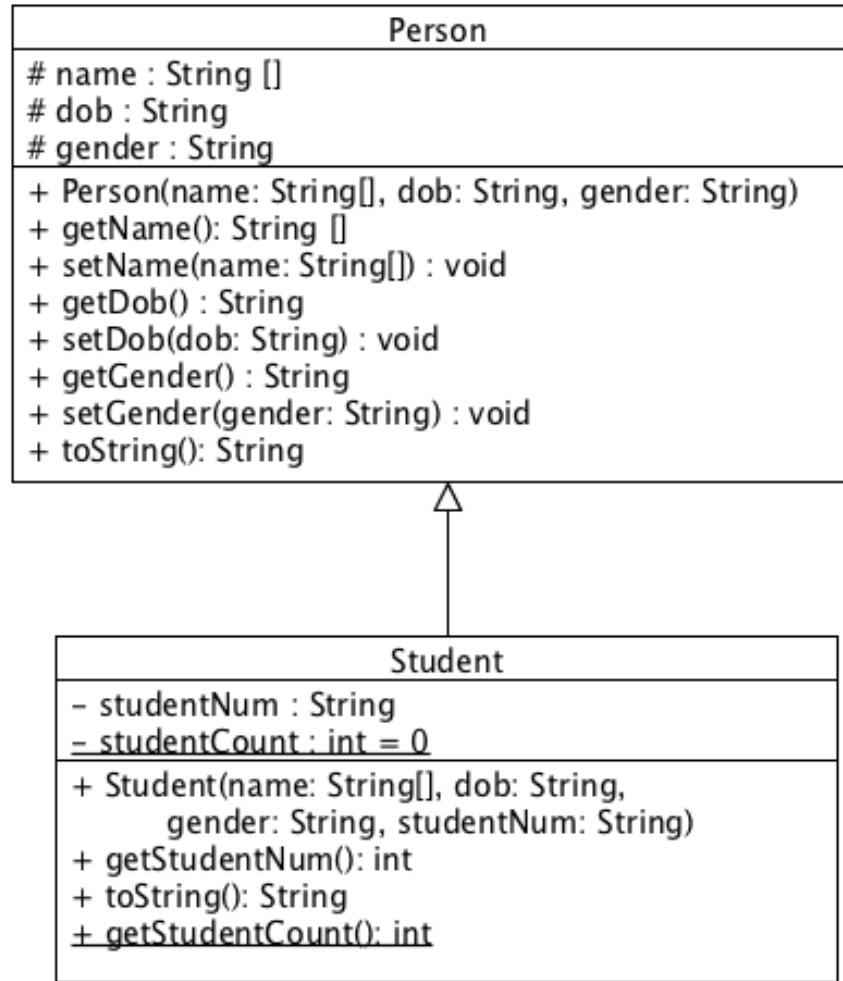
# EXAMPLE: CLASS PERSON



# NOTATION

- Data members:
  - name : type
- Methods:
  - methodName (param1 : type1, param2 : type2, . . . ) : returnType
- Visibility:
  - – private
  - + public
  - # protected
  - ~ package
- Static: underline

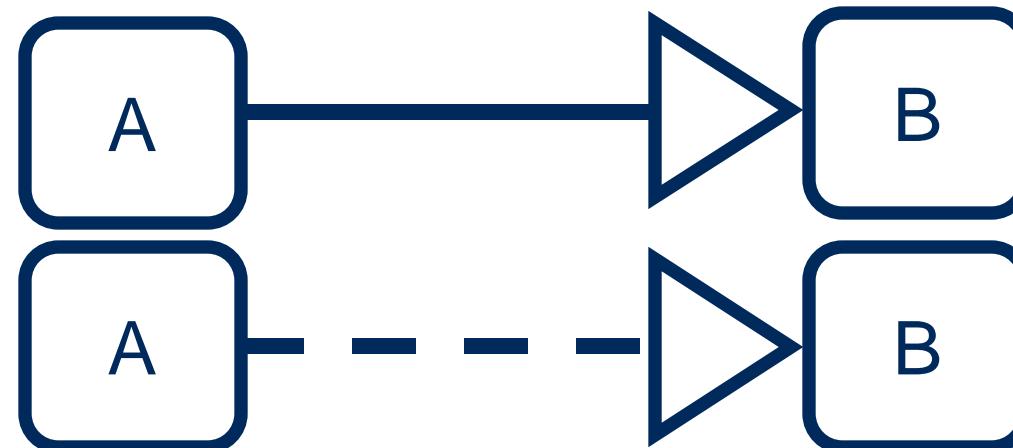
# EXAMPLE: INHERITANCE



# NOTATION (CONT'D)

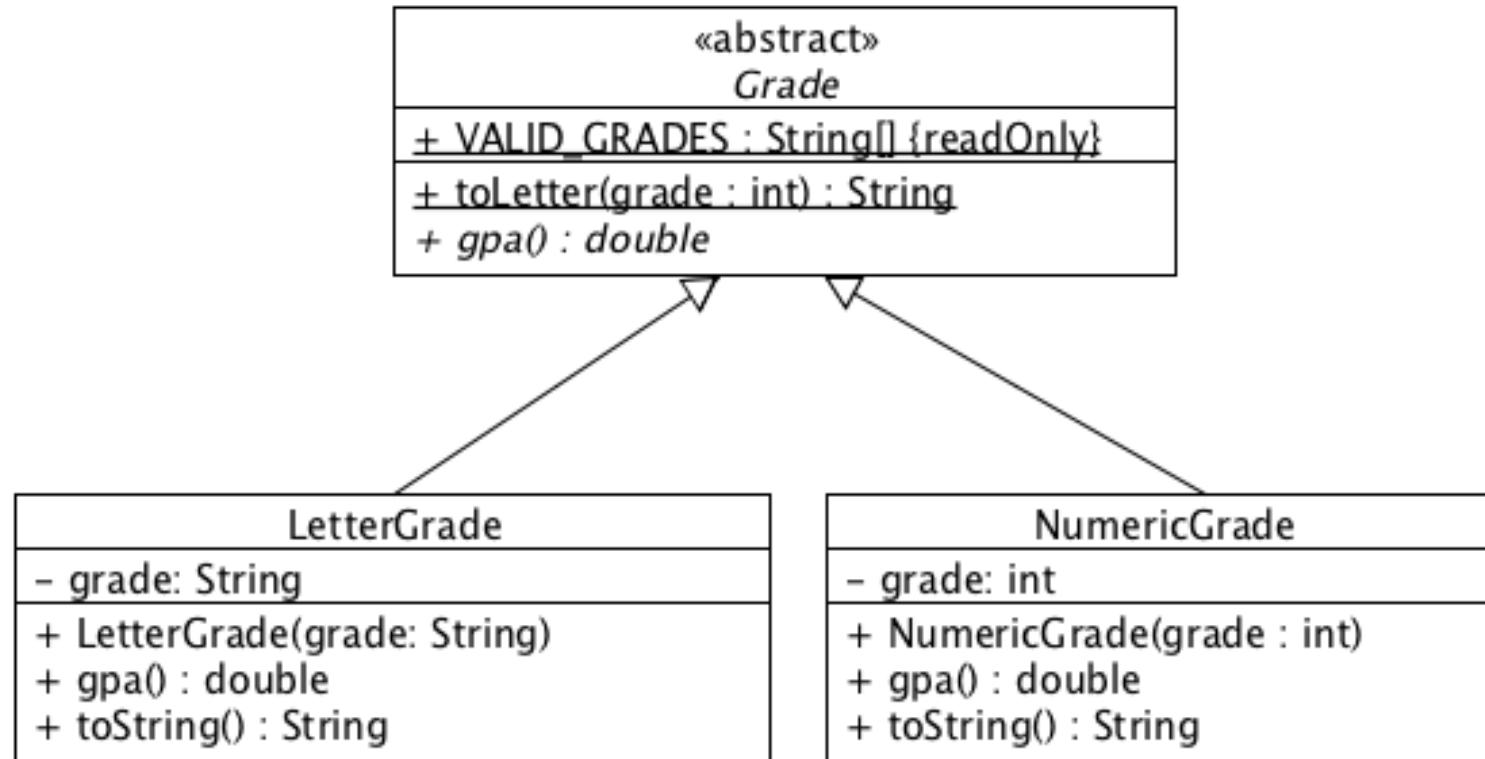
- Abstract method: *italic*
- Abstract class: *italic* or <<abstract>>
- Interface: <<interface>>
- Relationships between classes:

Inheritance  
(A inherits from B)



Interface  
(A implements B)

# EXAMPLE: ABSTRACT CLASS



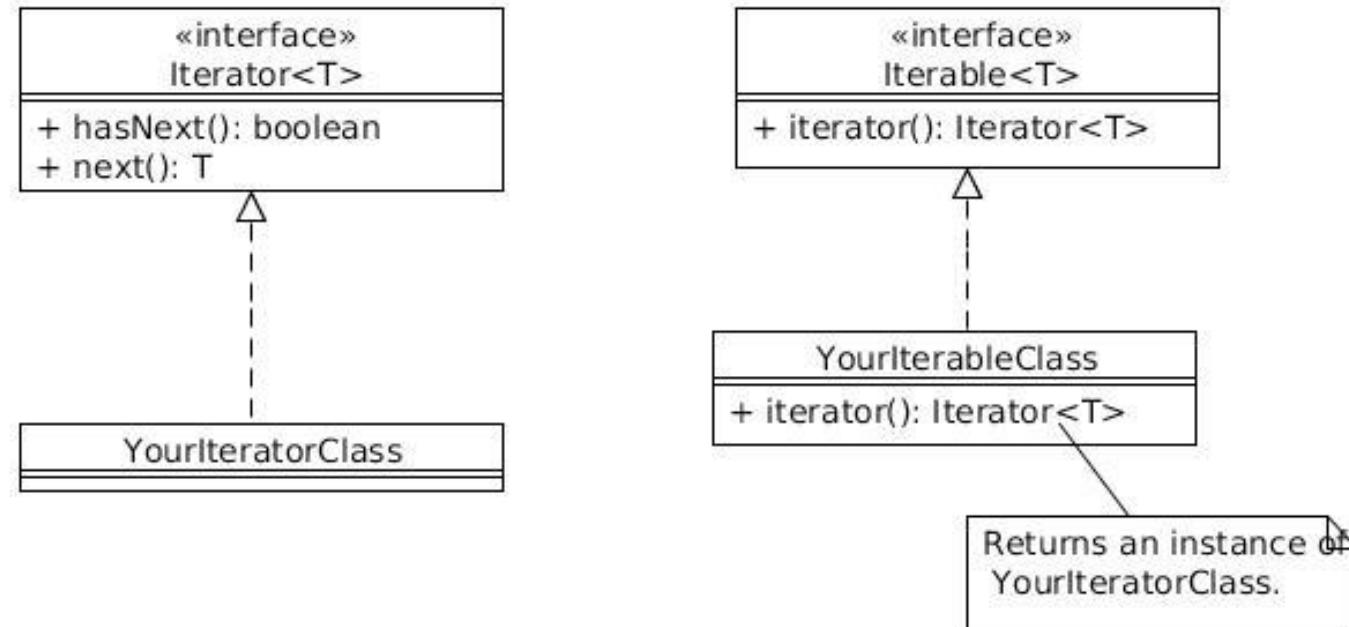
# DESIGN PATTERNS

- A **design pattern** is a general description of the solution to a well-established problem using an arrangement of classes and objects.
- Patterns describe the shape of the code rather than the details, so UML is an effective way to express them.
- They are a means of communicating design ideas.
- They are not specific to a programming language.
- You'll learn about many more patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).

# ITERATOR DESIGN PATTERN

- Context
  - A container/collection object.
- Problem
  - Want a way to iterate over the elements of the container.
  - Want to have multiple, independent iterators over the elements of the container.
  - Do not want to expose the underlying representation: should not reveal *how* the elements are stored.

# ITERATOR DESIGN PATTERN: JAVA



# HOMEWORK

---

- Start thinking about what design patterns might be useful for your project!



# DESIGN PATTERNS

CSC 207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

- Know what a design pattern is.
- Recognize some common design patterns and understand how to apply them.

# DESIGN PATTERNS

- We'll be covering a few common design patterns here.
- Iterator, Observer, Strategy, Dependency Injection, Simple Factory, Façade, Builder
- [https://sourcemaking.com/design\\_patterns/](https://sourcemaking.com/design_patterns/) has detailed explanations of these and many more design patterns, which you may find useful for your project and beyond this course.



# DESIGN PATTERNS (REVIEW)

- A **design pattern** is a general description of the solution to a well-established problem.
- Patterns describe the shape of the code rather than the details.
- They're a means of communicating design ideas.
- They are not specific to any one programming language.
- You'll learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).



# LOOSE COUPLING, HIGH COHESION

- These are two goals of object-oriented design.
- **Coupling:** the interdependencies between objects. The fewer couplings the better, because that way we can test and modify each piece independently.
- **Cohesion:** how strongly related the parts are inside a class. High cohesion means that a class does one job, and does it well. If a class has low cohesion, then an object has parts that don't relate to each other.
- **Design patterns are often applied to decrease coupling and increase cohesion.**

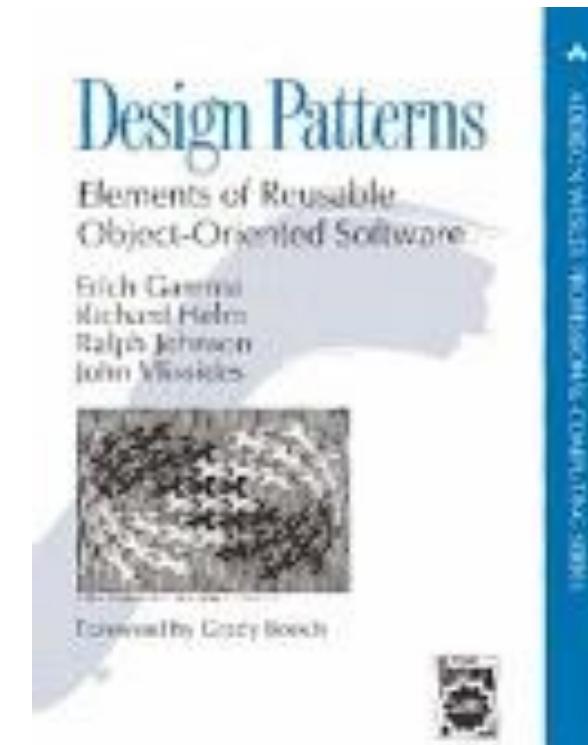
# REMINDER: SOLID

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Keep these in mind as we discuss each design pattern!

# GANG OF FOUR

- First codified by the Gang of Four in 1995
  - - Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- Original Gang of Four book described 23 patterns
  - - More have been added
  - - Other authors have written books



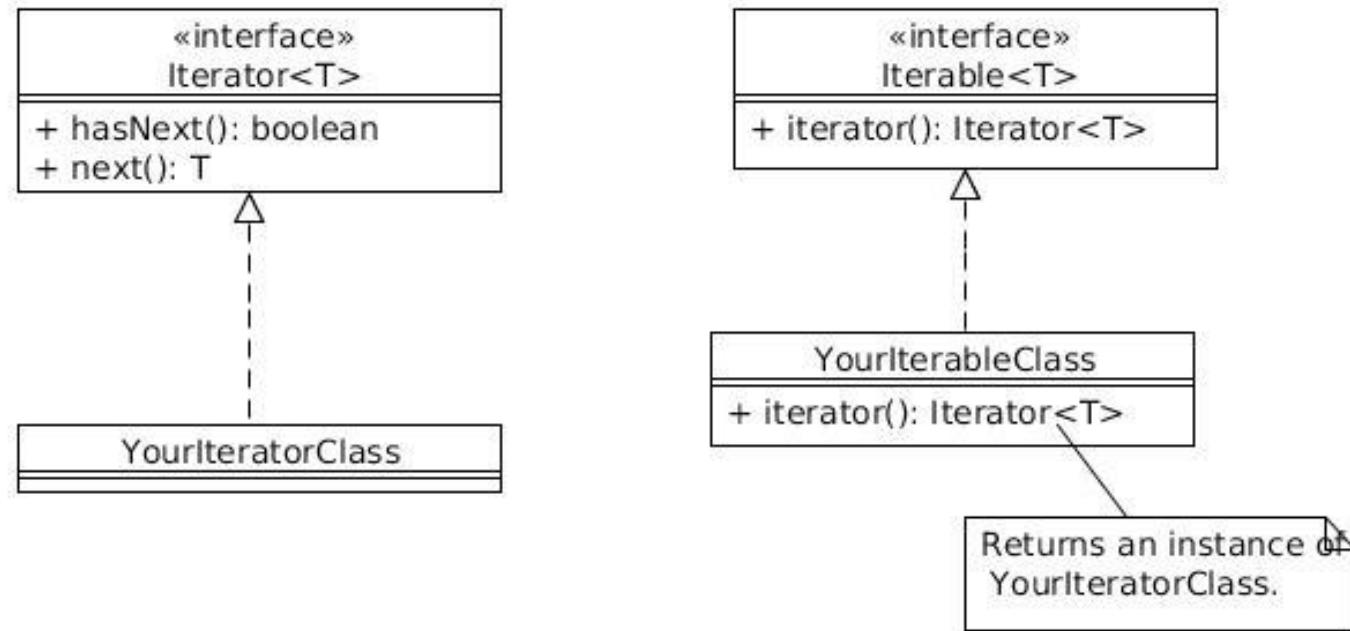
# THE BOOK PROVIDES AN OVERVIEW OF:

- **Design Pattern Name**
- **Problem**
  - when to use the pattern
  - motivation: sample application scenario
  - applicability: guidelines for when your code needs this pattern
- **Solution**
  - structure: UML Class Diagram of generic solution
  - participants: description of the basic classes involved in generic solution
  - collaborations: describes the relationships and collaborations among the generic solution participants
  - sample code
- **Consequences, Known Uses, Related Patterns, Anti-patterns**
  - Anti-patterns: what the code might look like before applying the pattern

# ITERATOR DESIGN PATTERN (REMINDER)

- Problem
  - Want a way to iterate over the elements of the container.
  - Want to have multiple, independent iterators over the elements of the container.
  - Do not want to expose the underlying representation: should not reveal how the elements are stored.

# ITERATOR DESIGN PATTERN

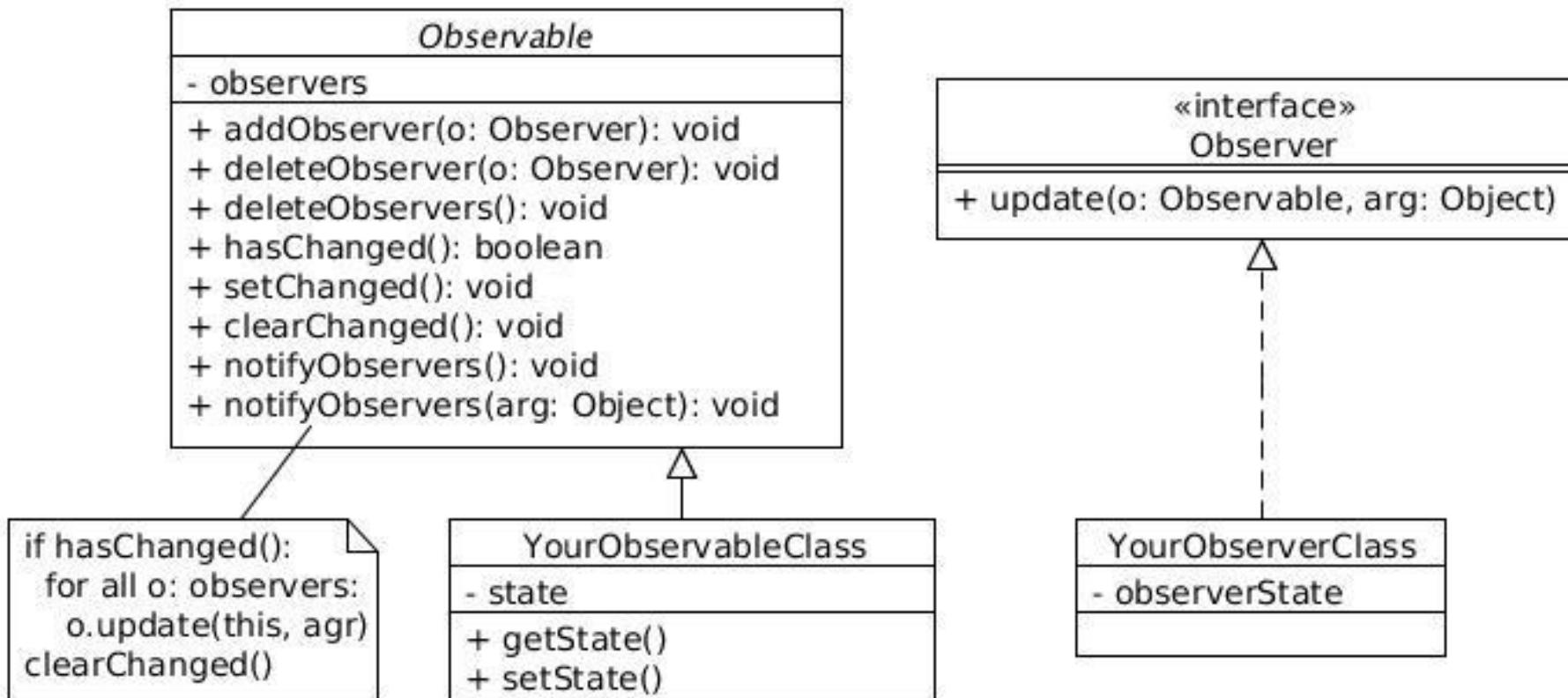


Example: look at the source code for `ArrayList` and its `ArrayList.iterator()` method (in IntelliJ or [online](#))!

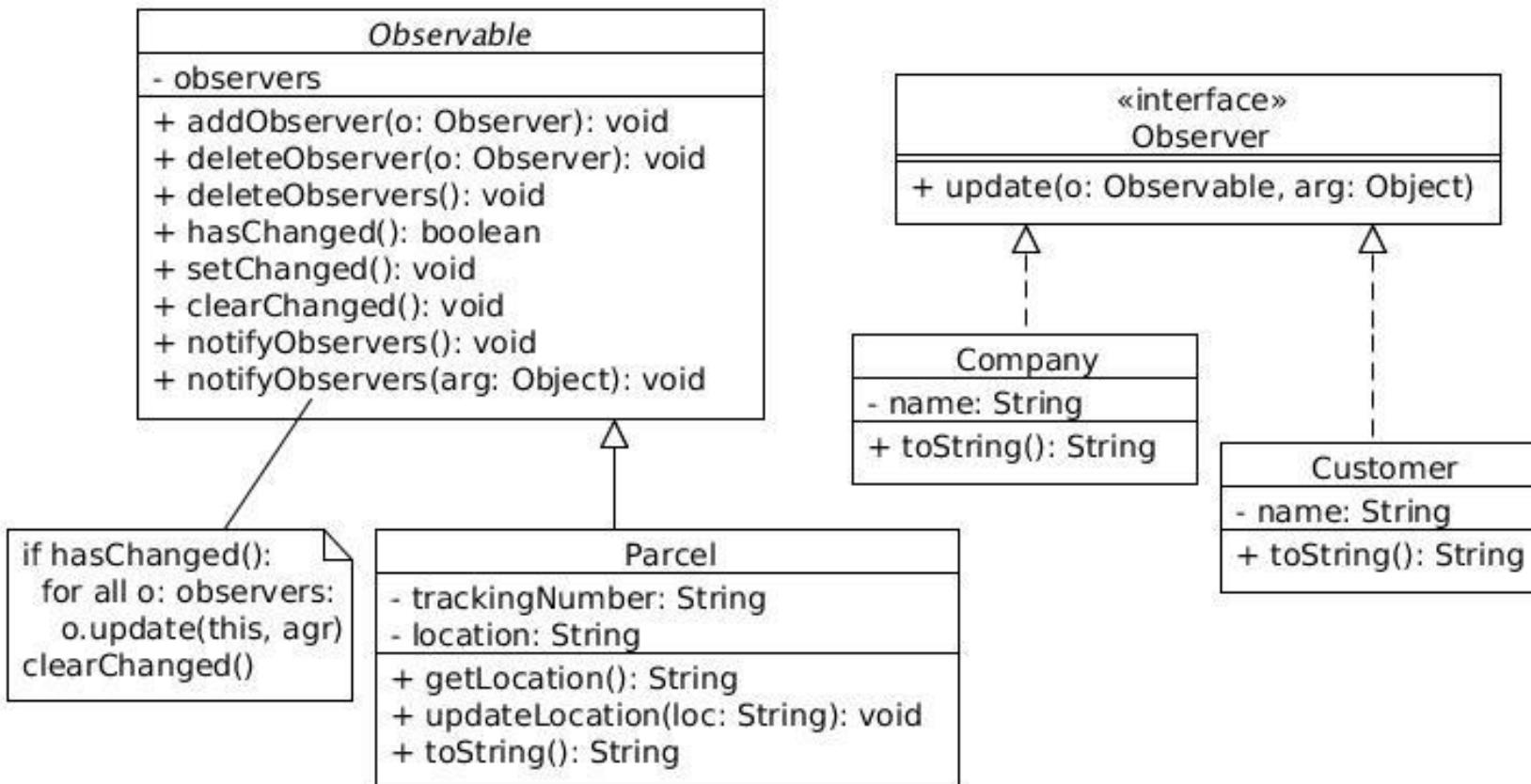
# OBSERVER DESIGN PATTERN

- Problem:
  - Need to maintain consistency between related objects.
  - Two aspects, one dependent on the other.
  - An object should be able to notify other objects without making assumptions about who these objects are.

# OBSERVER: JAVA IMPLEMENTATION



# OBSERVER: PARCEL EXAMPLE IN JAVA

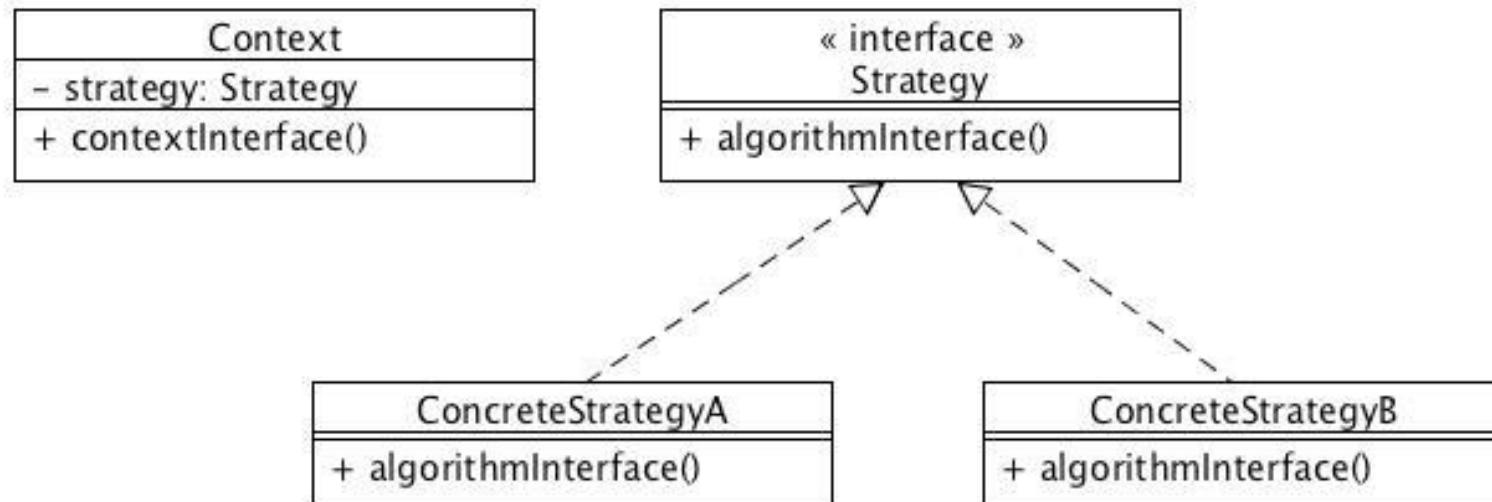


# STRATEGY DESIGN PATTERN

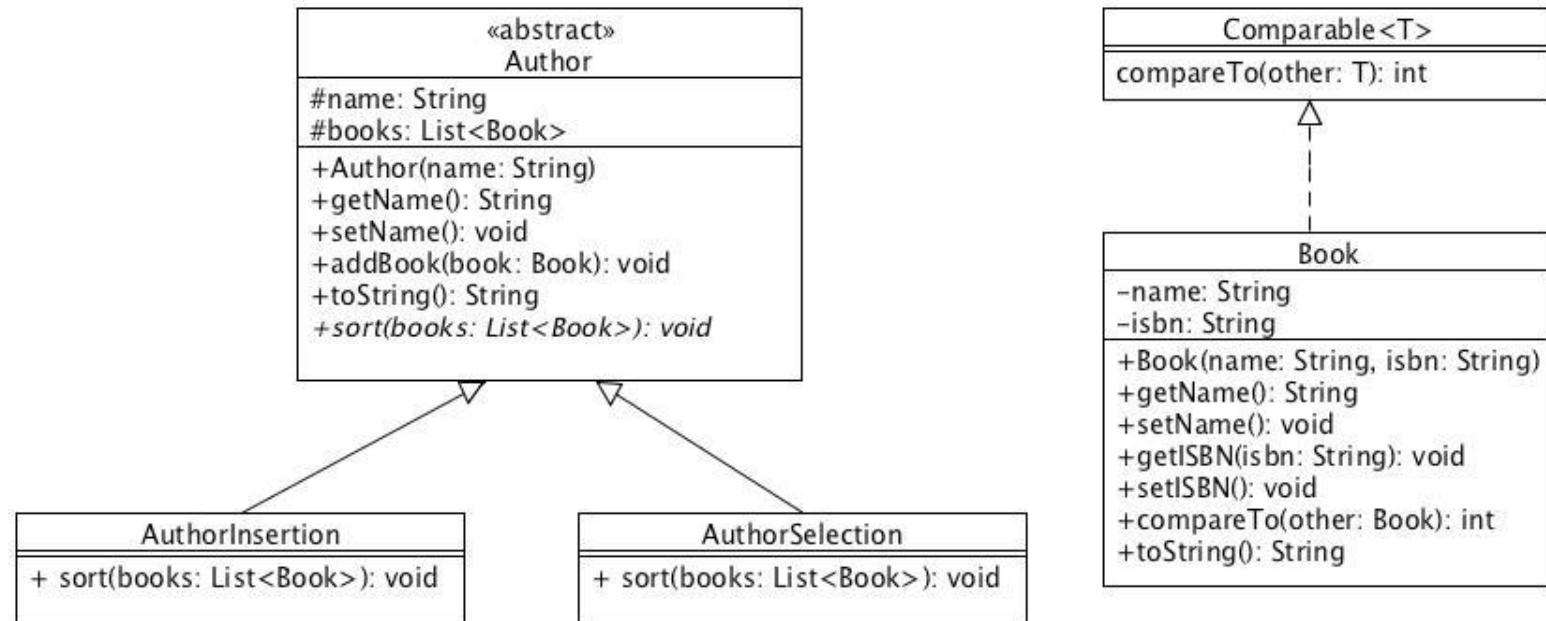
- Problem:
  - multiple classes that differ only in their behaviour (for example, use different versions of an algorithm)
  - but the various algorithms should not be implemented within the class
  - want the implementation of the class to be independent of a particular implementation of an algorithm
  - the algorithms could be used by other classes, in a different context
  - want to **decouple** — separate — the implementation of the class from the implementation of the algorithms



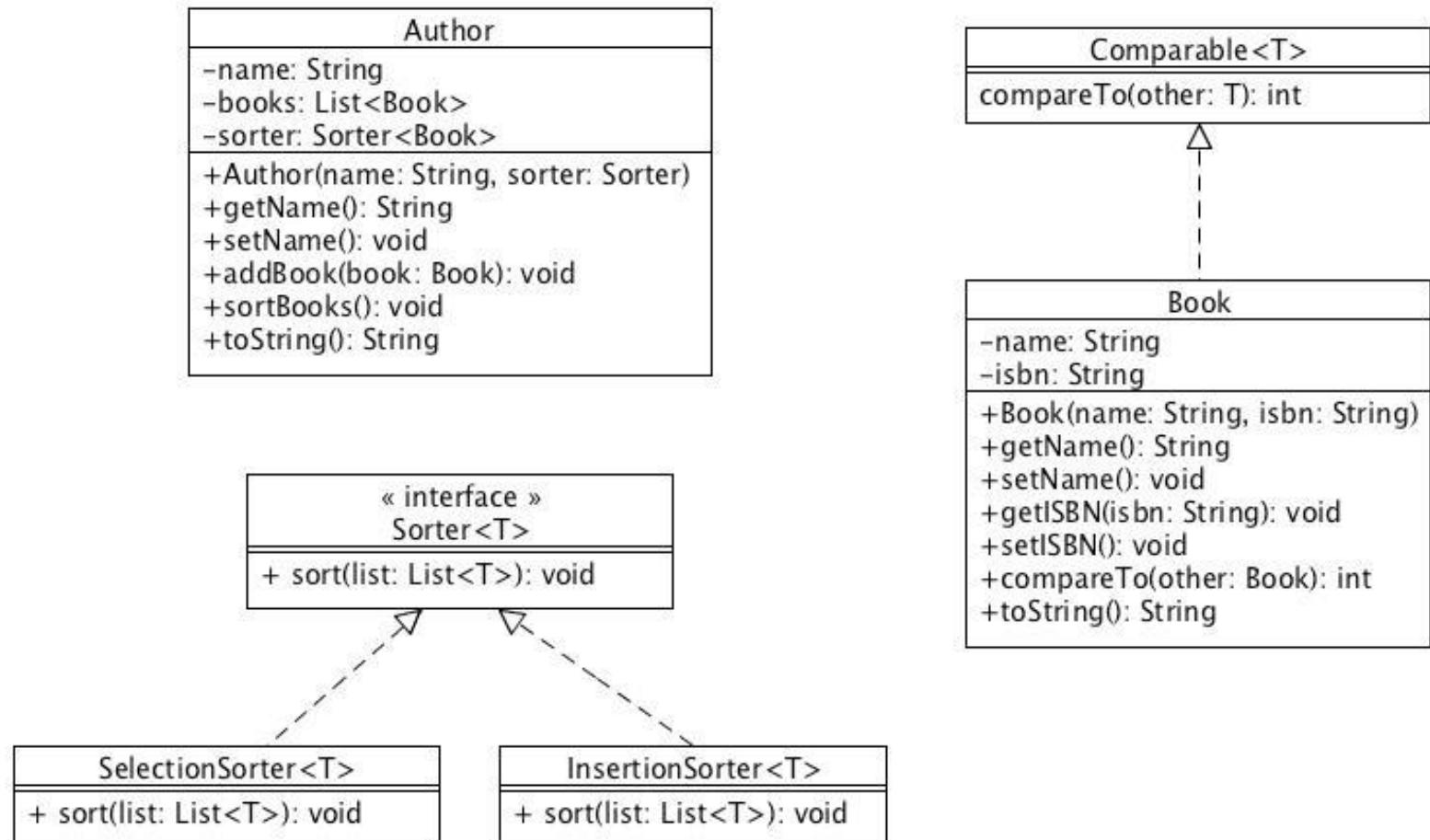
# STRATEGY: STANDARD SOLUTION



# EXAMPLE: WITHOUT THE STRATEGY PATTERN



# EXAMPLE: USING THE STRATEGY PATTERN



# DEPENDENCY IN OBJECT ORIENTED PROGRAMMING

- A “dependency” relationship between two classes (also called a “using” relationship) means that any change to the second class will change the functionality of the first.
- For example: class AddressBook depends on class Contact because AddressBook contains instances of Contact.
- Some other examples of dependencies: loggers, handlers, listeners

# DEPENDENCY INJECTION DESIGN PATTERN

- Problem:
  - We are writing a class, and we need to assign values to the instance variables, but we don't want to hard-code the types of the values. Instead, we want to allow subclasses as well.

# DEPENDENCY INJECTION EXAMPLE: BEFORE

- Using operator new inside the first class can create an instance of a second class that cannot be used nor tested independently. This is called a “**hard dependency**”.
- For example, this code creates a hard dependency from Course to Student

```
public class Course {  
    private List<Student> students = new ArrayList<>();  
  
    public Course(List<String> studentNames) {  
        for (String name : studentNames) {  
            Student student = new Student(name);  
            students.add(student);  
        }  
    }  
}
```



# DEPENDENCY INJECTION EXAMPLE: AFTER

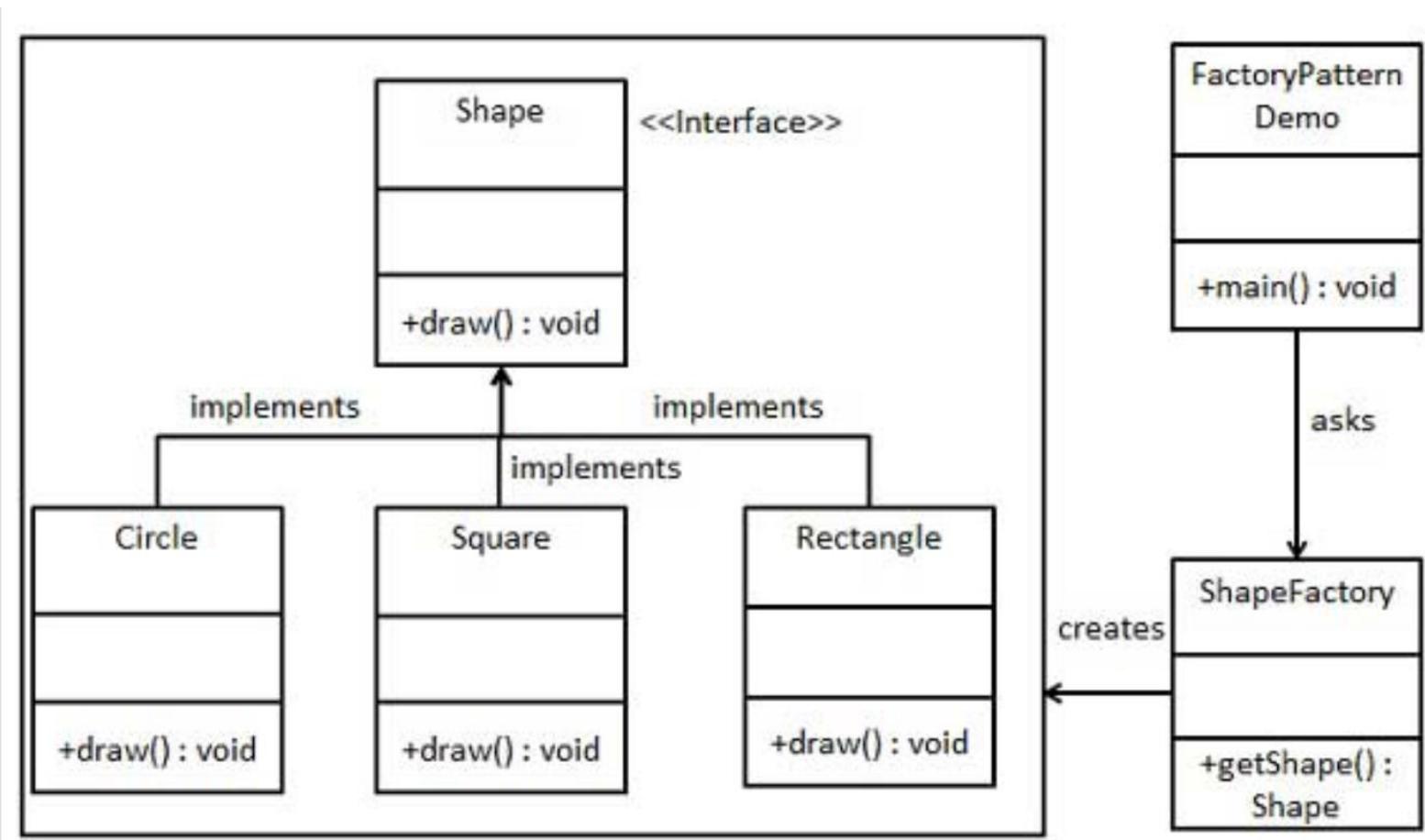
- The dependency injection solution is to create the Student objects outside and pass each Student (or perhaps a list of Students) into Course.
- This allows us to avoid the hard dependency and therefore we can even inject subclasses of Student into Course!

```
public class Course {  
    private List<Student> students = new ArrayList<>();  
  
    // Student objects are created outside the Course class and injected here.  
    public add(Student s) {  
        this.students.add(s);  
    }  
    // We might also inject all of them at once.  
    public addAll(List<Student> studentsToAdd) {  
        this.students.addAll(studentsToAdd);  
    }  
}
```

# SIMPLE FACTORY DESIGN PATTERN

- Problem:
  - One class wants to interact with many possible related objects.
  - We want to obscure the creation process for these related objects.
  - At a later date, we might want to change the types of the objects we are creating.

# FACTORY : AN EXAMPLE



# FAÇADE DESIGN PATTERN

- Problem:
  - A single class is responsible to multiple “actors”.
  - We want to encapsulate the code that interacts with individual actors.
  - We want a simplified interface to a more complex subsystem.
- Solution:
  - Create individual classes that each interact with only one actor.
  - Create a Façade class that has (roughly) the same responsibilities as the original class.
  - Delegate each responsibility to the individual classes.
    - This means a Façade object contains references to each individual class.

# FAÇADE DESIGN PATTERN: BEFORE

- In some restaurant software, we have a class called Bill. It is responsible for:
  1. Calculating the total based on a frequently-changing set of discount rates. (“10% off before 11am”)
    - Interacts with a discount system that contains a list of rates.
  2. Logging the amount paid and updating the accounting subsystem.
    - Interacts with the accounting system.
  3. Printing a nicely-formatted bill to give to the customer.
    - Interacts with the print device.

# FAÇADE DESIGN PATTERN: AFTER

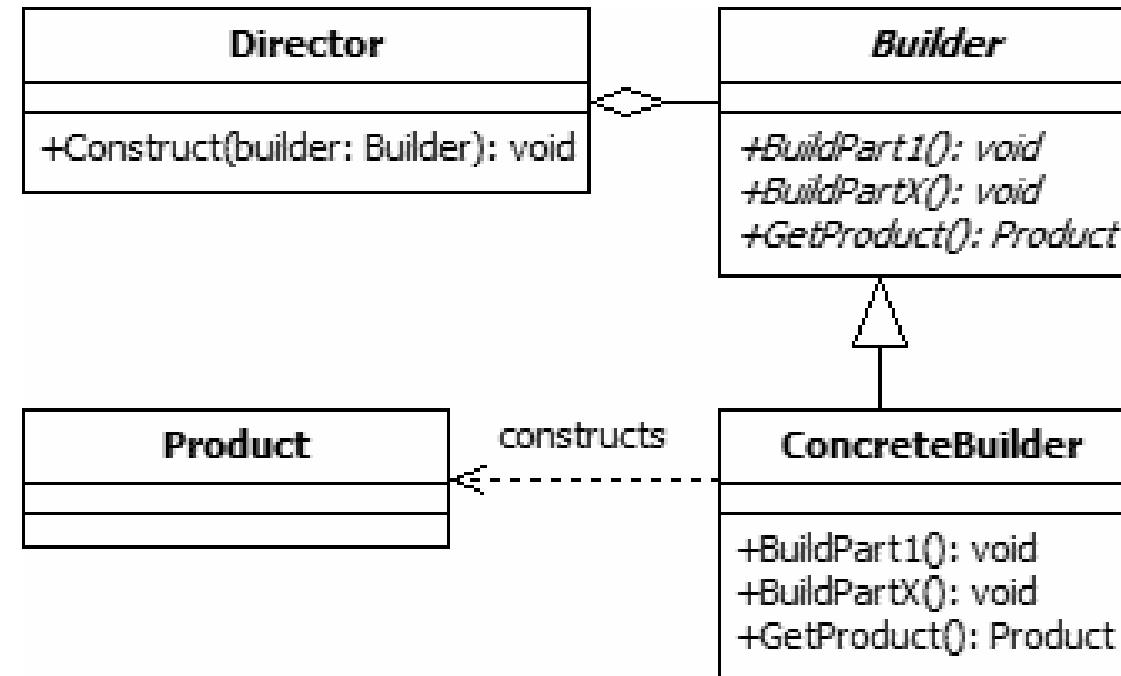
- Factor out an Order object that contains the menu items that were ordered.
- Create classes called BillCalculator, BillLogger, and BillPrinter that all use Order.
- Create BillFacade, which **delegates** the operations to BillCalculator, BillLogger, and BillPrinter.
- For example, BillFacade might contain this instance variable and method:

```
BillCalculator calculator = new BillCalculator(order);

public calculateTotal() {
    calculator.calculateTotal();
}
```

# BUILDER DESIGN PATTERN

- Problem:
  - Need to create a complex structure of objects in a step-by-step fashion.
- Solution:
  - Create a Builder object that creates the complex structure.



# BUILDER DESIGN PATTERN: AFTER

- With the Bill example, we might have the BillFacade create the various sub-objects. Instead, we will create a Builder object that does this work.

```
class OrderManager {  
    void construct(Builder builder, List<MenuItem> orders) {  
        builder.buildOrder(orders);  
        builder.buildCalculator();  
        builder.buildLogger();  
        builder.buildPrinter();  
        builder.buildFacade();  
    }  
}
```

# MORE BUILDER EXAMPLES

- A repo that extensively uses builder (see [SpotifyApi.java](#) and many other classes in it)
  - <https://github.com/spotify-web-api-java/spotify-web-api-java#General-Usage>
- IntelliJ refactoring to replace a constructor with a builder
  - <https://www.jetbrains.com/help/idea/replace-constructor-with-builder.html>

# HOMEWORK

- Complete the Quercus quizzes about design patterns.
- Continue to think about design patterns that would make sense to incorporate into your project!





# Embedded Ethics Disability and Software Accessibility

## Module 1

# Target Demographics of Your Course Project

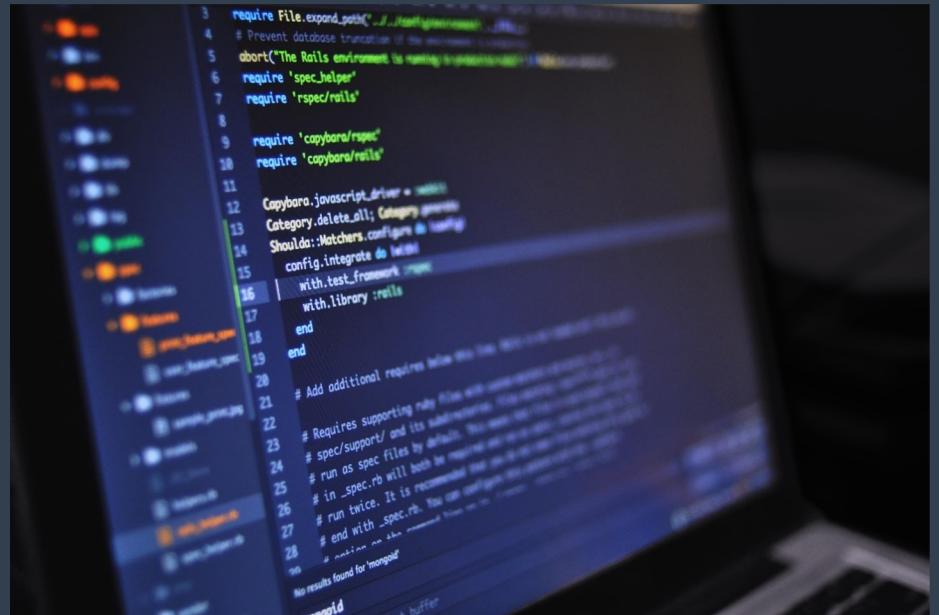
- 1) When you imagine your average user, whom do you think of?
- 2) Who else could benefit from your program?
- 3) Who might find it difficult to use your program?

# Case Study: Night mode

By giving the user the option to use Night Mode or Dark Theme, does this eliminate any barriers for users?

If so, which barriers? Who benefits from this?

If not, why else might it be offered?



```
3 require File.expand_path('../config/environment', __FILE__)
4 # Prevent database truncation if the environment is :test.
5 abort("The Rails environment is running in production.") if Rails.env == "production"
6 require 'spec_helper'
7 require 'rspec/rails'
8
9 require 'capybara/rspec'
10 require 'capybara/malli'
11
12 Capybara.javascript_driver = :webkit
13 Category.delete_all; Category.create!(name: "All")
14 Shoulda::Matchers.configure do |config|
15   config.integrate do |with|
16     with.test_framework :rspec
17     with.library :rails
18   end
19 end
20
21 # Add additional requires below this line to include more frameworks or libraries.
22
23 # Requires supporting files within the same directory as this file if you want to load
24 # them from there. The 'spec/support/' directory is excluded from the search path.
25 # run as spec files by default. This can be changed in .spec.opts
26 # in _spec.rb will both be recognized as test files.
27 # run twice. It is recommended that you do not name them both _spec and spec.
28 # end with _spec.rb. You can control this behavior with the --tag option
29 # or via an environment variable named RSpec::OPTIONS.
30
31 # This file is not part of the standard distribution.
32 # It is located here so that it can be shared across applications without
33 # having to copy it into every application.
34
35 # This file is not part of the standard distribution.
36 # It is located here so that it can be shared across applications without
37 # having to copy it into every application.
```

How we imagine our users affects  
how we choose which features to  
implement in our programs.

One crucial set of users whose needs may differ from the average user are those with disabilities.

The WHO estimates that 15% of users have disabilities

# Welcome to Embedded Ethics!

- 1) As usual, feel free to contribute, ask questions, etc.
- 2) Our goal is not to tell you *what* to think about ethical problems, but *how* to think about them.

The concepts we use influence how  
we talk, act and design.

## Examples of disabilities

Paraplegia (paralysis of lower limbs)

Deafness

Blindness

Mental illness

Speech impairment

# What do Disabilities have in Common with Each Other?

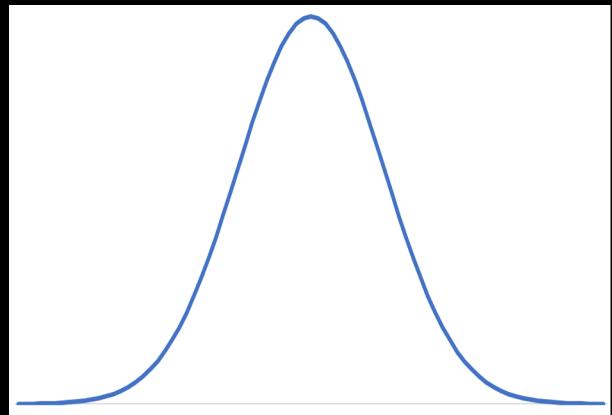
Wasserman *et al* 2006: a disability is a *physical or mental impairment* that is associated with a *personal or social limitation* on the activities one can perform.

Physical or mental impairment

Personal / social limitation

Question: why does disability involve both of these?

## Physical or mental impairment



Biological (based on a theory of human function)



Statistical (based on deviation from some defined average)



Personal / social  
limitation

A **limitation** is a limit on the set of activities that a person can perform:

Basic movements

Complex actions

Social Activities



Lightning storm

causes



The forest fire



Availability of  
oxygen for the fire

Normally we think of the “**cause**” of an event as another specific event that occurred before it.

Other factors are simply “**background conditions**” - required for the event to happen, but not part of the “cause”

## Poll 1

Event: Billiard ball #1 hits billiard ball #2, and billiard ball #2 starts moving.

What do you think is best described as the “cause” of billiard ball #2 starting to move?

- 1) Billiard ball #1 hitting billiard ball #2
- 2) The lack of glue on the table

Insert Link to Poll (mentimeter, google forms, etc)



## Poll 2

What do you think is best described as the “cause” and what is the “background condition”?

Event: The forest fire.

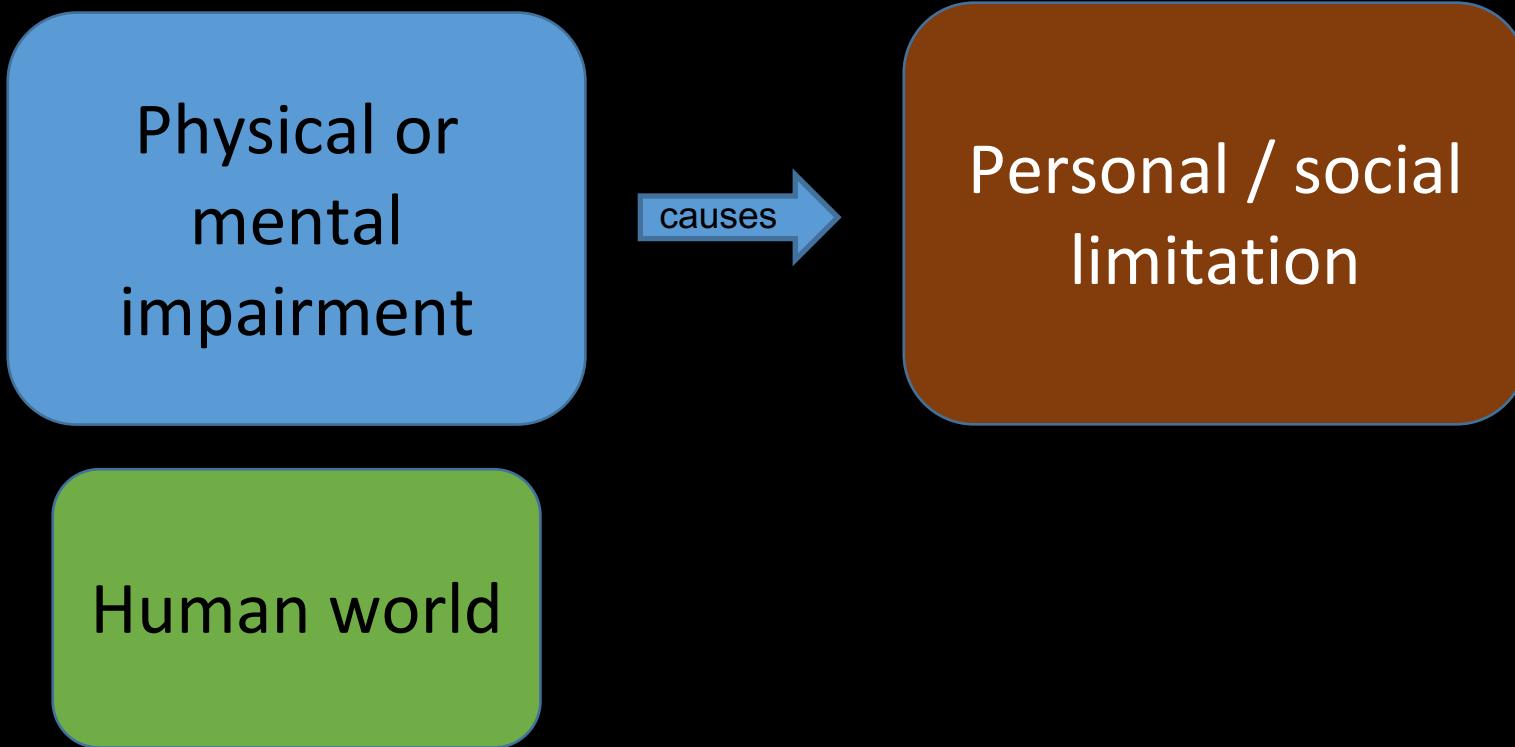
What is the cause?

- 1) The routine thunderstorm
- 2) The exceptionally dry summer

Insert Link to Poll (mentimeter, google forms, etc)



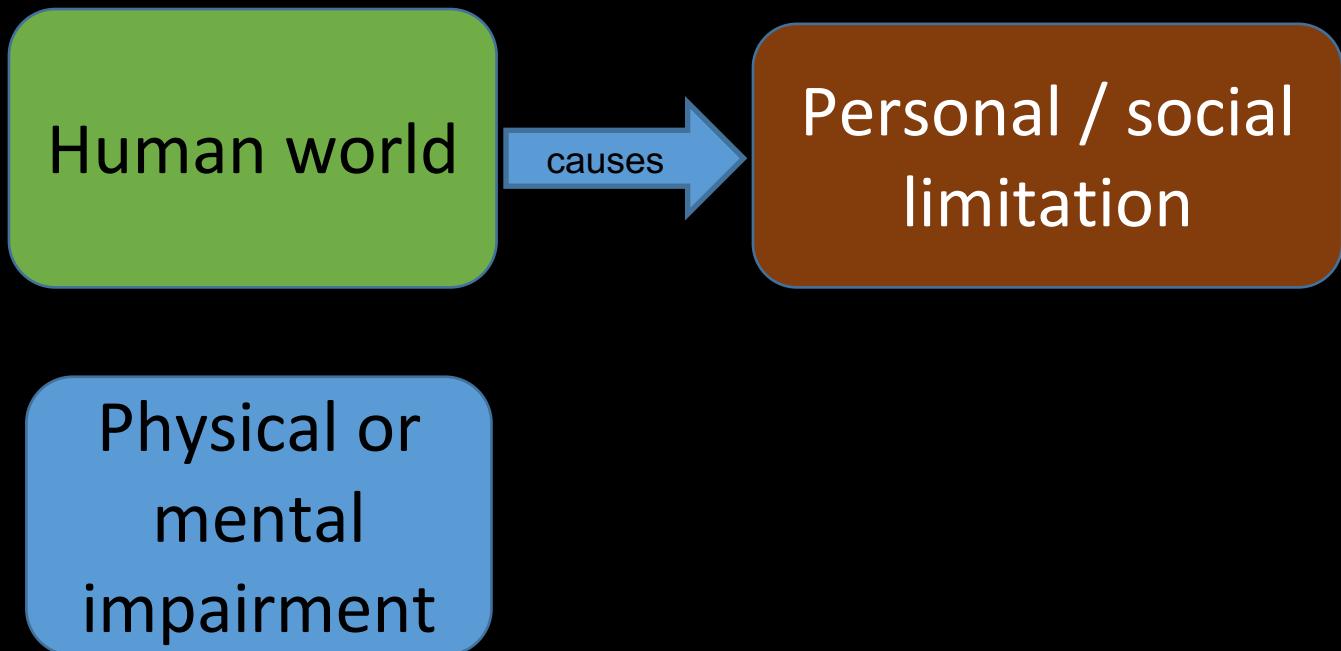
# The Medical Model of Disability



In some disabilities, the limitations are **caused by the physical or mental impairment**.

The **human world** is a background condition.

# The Social Model of Disability



In some disabilities, the limitations are caused by **the human world** – whether through stigma or environmental design. The **impairment** is a background condition.

It matters how we conceptualize disability.

If we conceptualize disability through the **medical model**, we tend to address disability by removing the trait that causes it.

If we conceptualize disability through the **social model**, we tend to address disability by modifying the environment around it.





“Well, I was born with a rare visual condition called achromatopsia, which is total color blindness, so I've never seen color, and I don't know what color looks like, because I come from a grayscale world...

...But, since the age of 21, instead of seeing color, I can hear color... it's a color sensor that detects the color frequency in front of me — (Frequency sounds) — and sends this frequency to a chip installed at the back of my head, and I hear the color in front of me through the bone, through bone conduction.”

Neil Harbisson, “I Listen to Color”  
[\(TED Talk\)](#)

“Hearing people assume that the Deaf live in a perpetual state of wanting to hear, because they can't imagine any other way. But I've never once wished to be hearing. I just wanted to be part of a community like me.”

Rebecca Krill, “How Technology has Changed What it's Like to be Deaf”  
[\(TED Talk\)](#)





For some impairments, and some limitations, the **medical model** will seem more appropriate.

For some impairments, and some limitations, the **social model** will seem more appropriate.



In the following, what would you say causes the limitation? Is it the impairment, or the human world?

- Impairment: Paraplegia (paralysis of the lower limbs)
  - Limitation: difficulty accessing public spaces.
  - Impairment: Attention deficit disorder (ADD)
  - Limitation: difficulty concentrating on schoolwork.
1. Take five minutes: Discuss your answer with your neighbours. Do they have the same opinion as you?

Insert Link to Poll (mentimeter, google forms, etc)

# Comparing the Medical Model and Social Model

2) How would the medical and social model differ in their prescriptions for each disability?

	<b>How would the medical model address this disability?</b>	<b>How would the social model address this disability?</b>
Paraplegia		
Attention Deficit Disorder		

# Summary

The concepts we use to think about disability (and programming!) matter.

According to the medical model of disability, a physical or mental impediment causes a personal or social limitation.

According to the social model of disability, the human world causes a personal or social limitation.

# Back to Software Design

- When creating a program, we already have to consider the technical requirements for the user's software and hardware
  - (example: Which operating systems will run it? How much memory is required?)
- From the medical model perspective, we can consider personal requirements as well
  - (example: How precise does their coordination have to be?)
- Through the social model, we can ask ourselves: Who might need this software? What barriers to usage might they face?

# Software and the Medical Model

Here are some examples of software that are specifically designed to be accessible to people with various diagnoses:

- Braille Translation Software (example: Index BrailleApp) converts text to a document that can be printed by a Braille printer
- Software that runs a simplified phone interface for people with Down Syndrome (example: Simple Smartphone)
- Programs that can learn from users' past mistakes to predict future text and suggest corrections for spelling and grammatical errors, aimed at people with dyslexia (example: Ghotit)

# Software and the Medical Model

Here are some more examples of software that are specifically designed to be accessible to people with various diagnoses:

- Sip-and-puff systems that allow a person to input information into a computer using a joystick that can be manipulated in any direction by their mouth, like a joystick, and sipping or puffing into the controller replacing clicking a mouse (example: Jouse3 for drawing or computer games, Origin instrument products for controlling a mouse, joystick, or keyboard)
- Domain specific voice recognition software that allows people to create text by speaking instead of typing (example for math symbols: MathTalk)

# Software and the Social Model

Here are some examples of software features that are aimed to allow programs to reach a wider audience regardless of whether or not users are identified as disabled:

- Automatic Captioning for anyone who might understand better if they see the words being spoken (example: Zoom feature)
- Giving the user the ability to zoom in to see the UI better, or otherwise change the size of icons and text (example: all operating systems)
- Allowing "unlocking" a device using biometrics like fingerprint instead of a pass code (example: Windows 10, Mac OS)
- Text-to-speech features that allow anyone to listen to a text (example: Speechify)
- Dark Theme (example: IntelliJ)

# Links to Companies

- Index BrailleApp -- <https://www.indexbraille.com/>
- Simple Smartphone -- <http://simplesmartphone.com>
- Ghotit -- <https://www.ghotit.com/>
- Jouse3 -- <https://www.enablingtech.ca/products/jouse-3>
- MathTalk -- <https://mathtalk.com/>
- Zoom -- <https://zoom.us>
- Biometrics -- Linux, Mac, Window
  - <https://itsfoss.com/fingerprint-login-ubuntu/>
  - <https://support.apple.com/en-ca/guide/mac-help/mchl16fbf90a/mac>
  - <https://www.computerworld.com/article/3244347/what-is-windows-hello-microsofts-biometrics-security-system-explained.html>
- IntelliJ -- <https://www.jetbrains.com/idea/>

# Designing for Accessibility

- World Wide Web Consortium on Accessibility in Web Design
- Google accessibility pages for Android and web design
- General Google Accessibility Guidelines
- Apple Accessibility Guidelines

# Principles of Universal Design

Principle 1: Equitable Use

Principle 2: Flexibility in Use

Principle 3: Simple and Intuitive Use

Principle 4: Perceptible Information

Principle 5: Tolerance for Error

Principle 6: Low Physical Effort

Principle 7: Size and Space for Approach and Use

**Discussion Question:** For each principle, does the Dark Theme option follow the principle? If so, in what way?

# Acknowledgements

This module was created as part of an Embedded Ethics Education Initiative (E3I), a joint project between the Department of Computer Science<sup>1</sup> and the Schwartz Reisman Institute for Technology and Society<sup>2</sup>, University of Toronto.

## **Instructional Team:**

Philosophy: Steven Coyne, Emma McClure, Julia Lei

Computer Science: Lindsey Shorser, Paul Gries, Jonathan Calver

## **Faculty Advisors:**

Diane Horton<sup>1</sup>, David Liu<sup>1</sup>, and Sheila McIlraith<sup>1,2</sup>

**Department of Computer Science**

**Schwartz Reisman Institute for Technology and Society**

**University of Toronto**



Computer Science  
**UNIVERSITY OF TORONTO**





# References

- Anderson-Chavarria, Melissa. 2021. "The autism predicament: models of autism and their impact on autistic identity", *Disability & Society*, DOI: 10.1080/09687599.2021.1877117
- Wasserman, David, Adrienne Asch, Jeffrey Blustein, and Daniel Putnam, "Disability: Definitions, Models, Experience", *The Stanford Encyclopedia of Philosophy* (Summer 2016 Edition), Edward N. Zalta (ed.), URL =  
<https://plato.stanford.edu/archives/sum2016/entries/disability/>

# CLEAN ARCHITECTURE REVIEW

CSC 207 SOFTWARE DESIGN

Frameworks & Drivers

Interface Adapters

Application Business Rules

Enterprise Business Rules



# LEARNING OUTCOMES

- Solidify your understanding of Clean Architecture.

# ARCHITECTURE

(Brief Summary of Chapter 15 from Clean Architecture textbook)

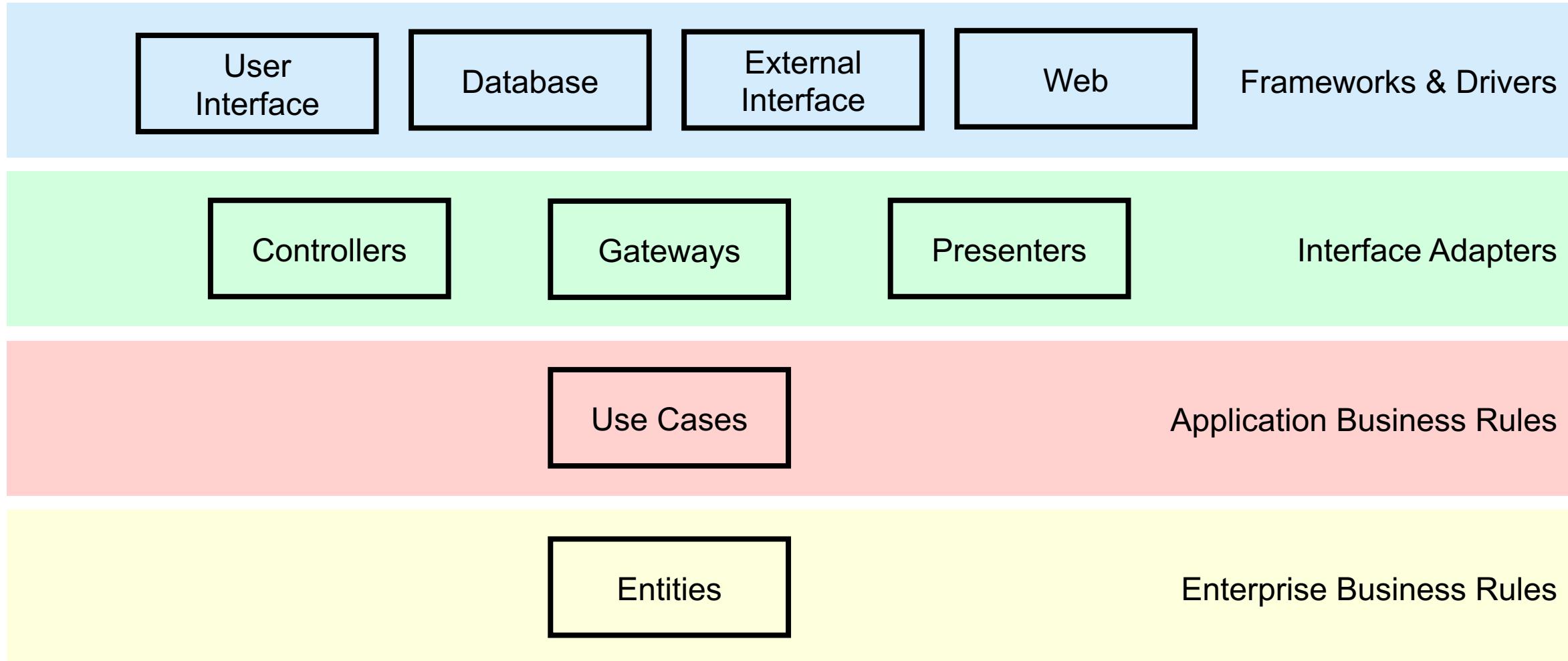
Design of the system

- Dividing it into logical pieces and specifying how those pieces communicate with each other.
- Input and output between layers.

“Goal is to facilitate the **development, deployment, operation, and maintenance** of the software system”

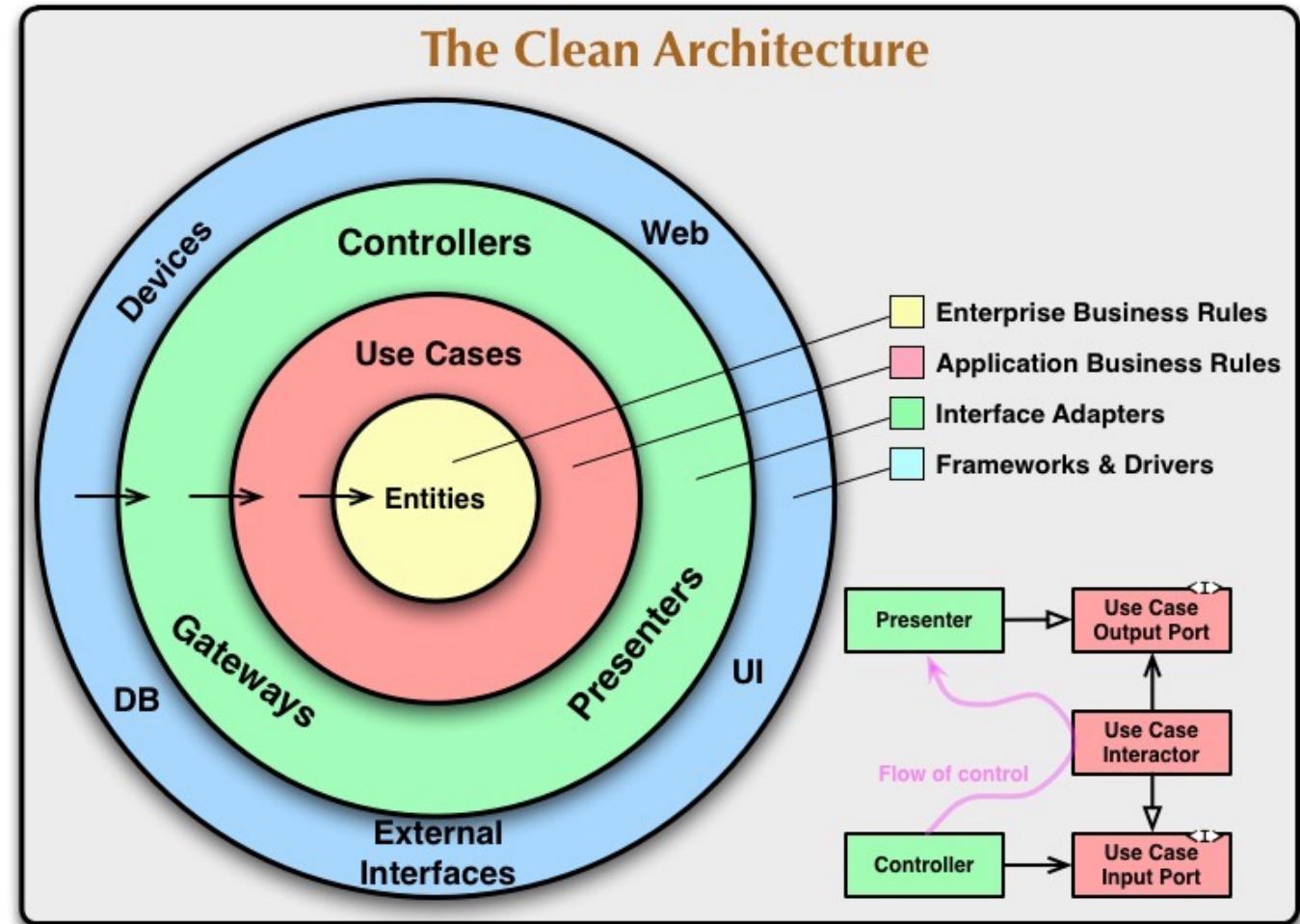
- ***“The strategy behind this facilitation is to leave as many options open as possible, for as long as possible.”***

# CLEAN ARCHITECTURE



# CLEAN ARCHITECTURE

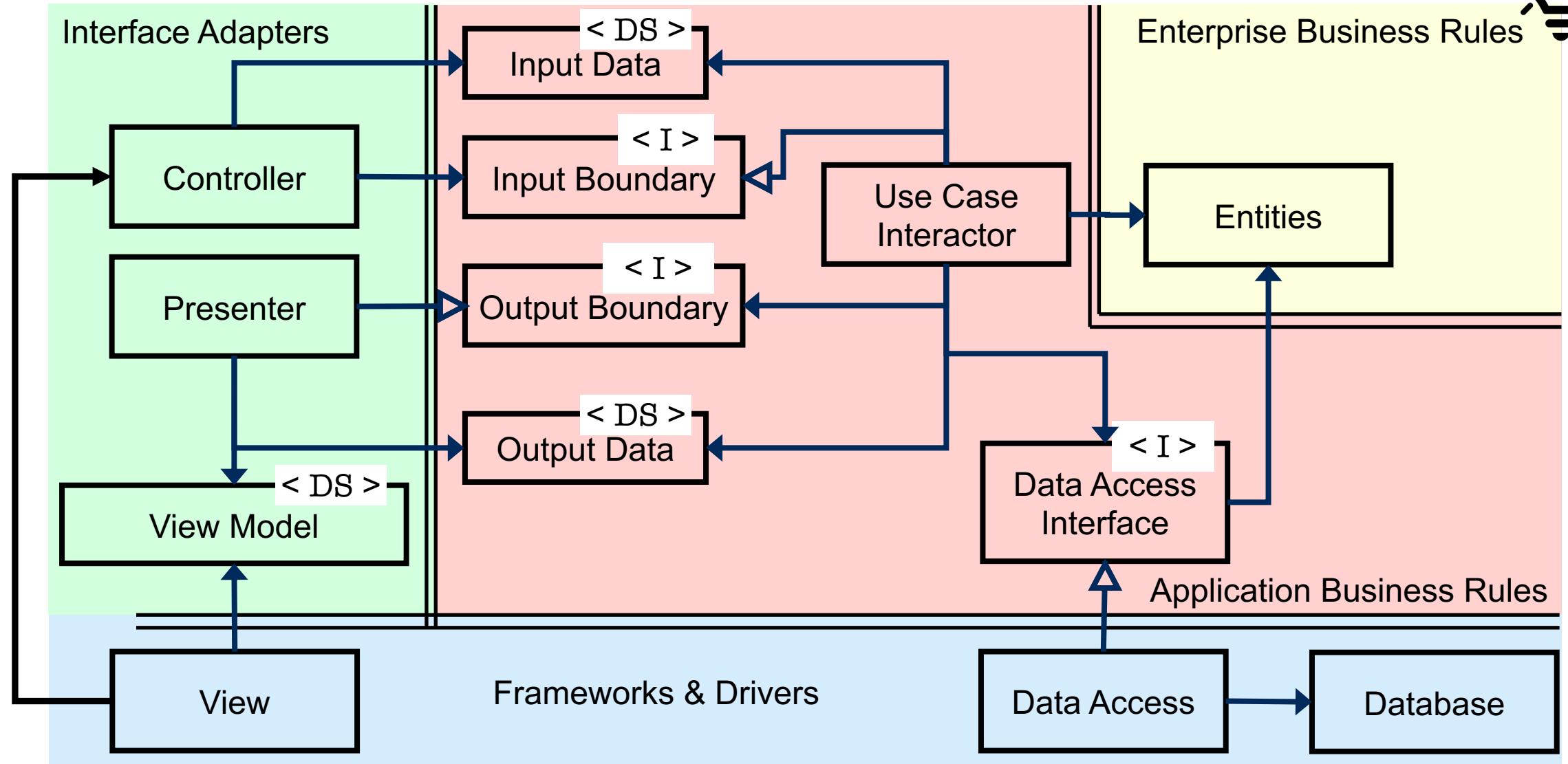
- Often visualize the layers as concentric circles.
- Reminds us that Entities are at the core
- Input and output are both in outer layers



# CLEAN ARCHITECTURE – DEPENDENCY RULE

- Dependence on adjacent layer — from outer to inner
- Dependence within the same layer is allowed (but try to minimize coupling)
- On occasion you might find it unnecessary to explicitly have all four layers, but you'll almost certainly want at least three layers and sometimes even more than four.
- **The name of something (functions, classes, variables) declared in an outer layer must not be mentioned by the code in an inner layer.**
- How do we go from the inside to the outside then?
  - Dependency Inversion!
  - An inner layer can depend on an interface, which the outer layer implements.

# EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



# BENEFITS OF CLEAN ARCHITECTURE

- All the “details” (frameworks, UI, Database, etc.) live in the outermost layer.
- The business rules can be easily tested without worrying about those details in the outer layers!
- Any changes in the outer layers don’t affect the business rules!

# MAIN

(Highlights of Chapter 26 from Clean Architecture textbook)

**Main is the entry point for the program.**

"The point is that Main is a dirty low-level module in the outermost circle of the clean architecture."

"Think of Main as a plugin to the application—a plugin that sets up the initial conditions and configurations, gathers all the outside resources, and then hands control over to the high-level policy of the application. Since it is a plugin, it is possible to have many Main components, one for each configuration of your application."

For example, you could have a Main plugin for Dev, another for Test, and yet another for Production. You could also have a Main plugin for each country you deploy to, or each jurisdiction, or each customer."

Example: <https://github.com/paulgries/UserLoginCleanArchitecture/blob/main/src/Main.java>



# DEPENDENCY INVERSION

This is a branch of the mvp-simple repo which inverts the dependency on the View class for the MVC version of the simple example.

<https://github.com/CSC207-2022F-UofT/mvp-simple/tree/mvc-version-cleaner>

Summary:

- Introduce an `IView` interface (abstraction of what a view is — here, anything with a `showNewCount` method!)
- Update any reference types to be `IView` instead of `View`.
- Have the `View` class implement the new `IView` interface.

# ADDITIONAL CLEAN ARCHITECTURE RESOURCES

We've compiled some supplemental links and suggested readings on Quercus:

- <https://q.utoronto.ca/courses/278453/pages/clean-architecture-resources>

# PROJECT TIPS

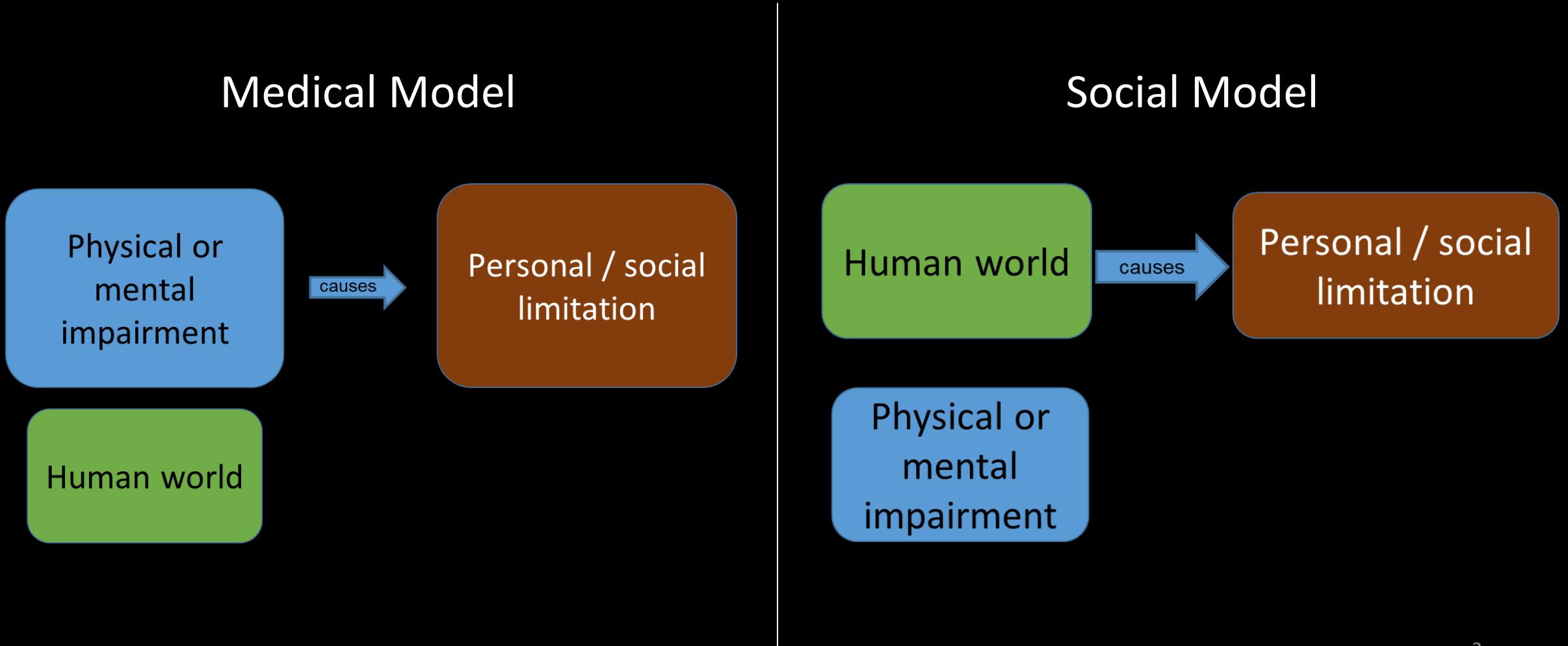
- Communicate with your team when making design decisions.
- Not sure if your code follows Clean Architecture?
  - Talk through your design with your team.
  - Generate the UML diagram for your code and ensure there are no unintended dependencies between classes.
  - Step through your code using the debugger to see the flow of control — keeping an eye on the call stack.
- Look at the reference project and sample grading for it if you have not yet done so.
- Have a plan and set deadlines to ensure you are making progress as a team!



# Embedded Ethics Disability and Software Accessibility Module 2



# A Quick Refresher of Module 1





Most people would say that there is an obligation to offer **reasonable accommodation** to people with disabilities.

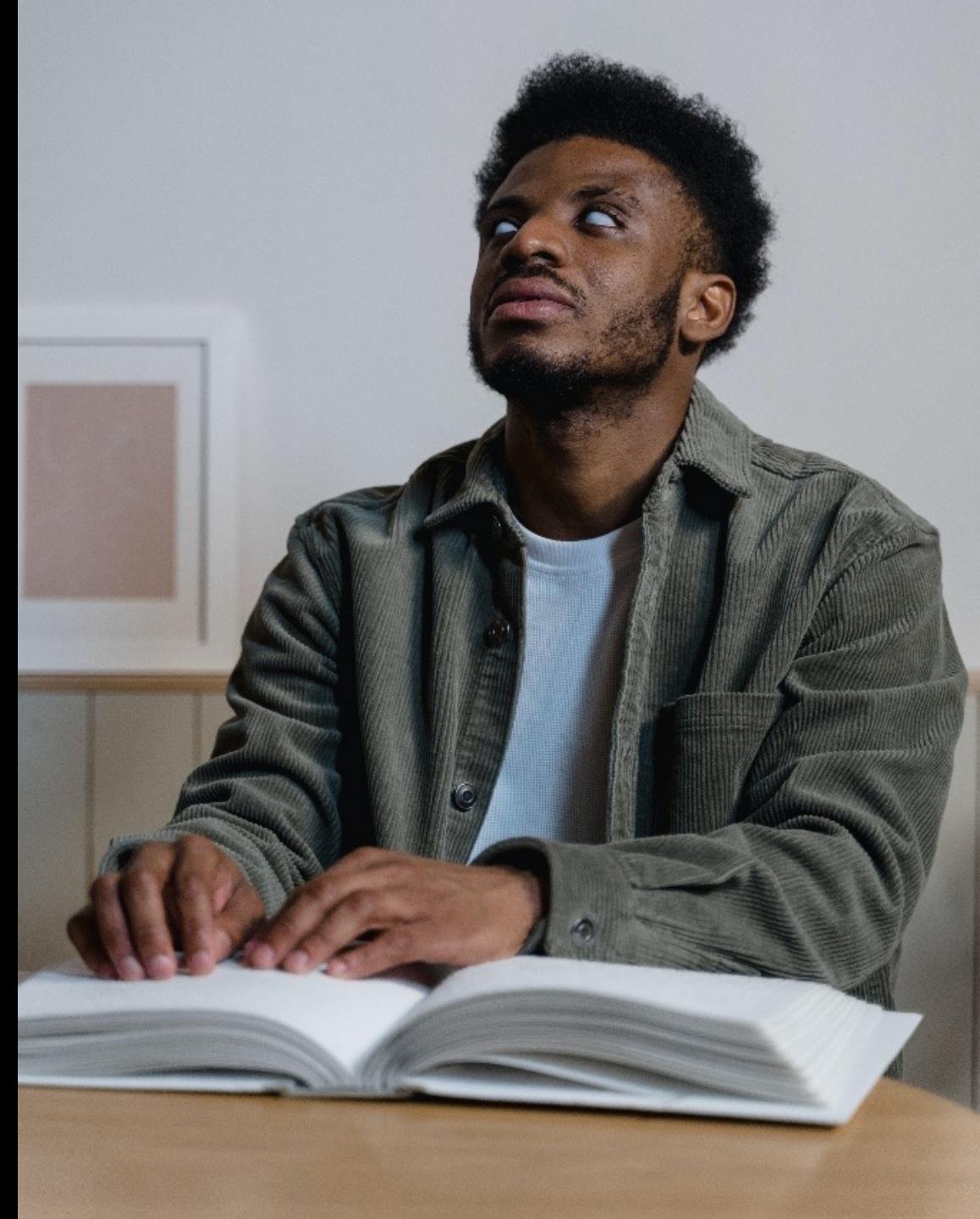


On the medical model,  
this will mean removing  
the traits that cause the  
limitations.

- Prosthetic limbs
- Cognitive Therapy
- Braille translation software

On the social model, this will mean redesigning the physical and social environment to make the trait no longer result in limitations.

- Wheelchair ramps
- Widespread knowledge of sign language
- Text-to-speech features
- Changing social conventions / attitudes to remove stigma around disability



# Question for Discussion

Putting it into words: Why do you think that it is important to accommodate people with disabilities?



# Part 1: The Morality of Accommodation



It is important to think about the impacts your applications will have on other people (and society as a whole).

One impact: Your application will **create** some things, and **redistribute** some things, both to your users and to other stakeholders.

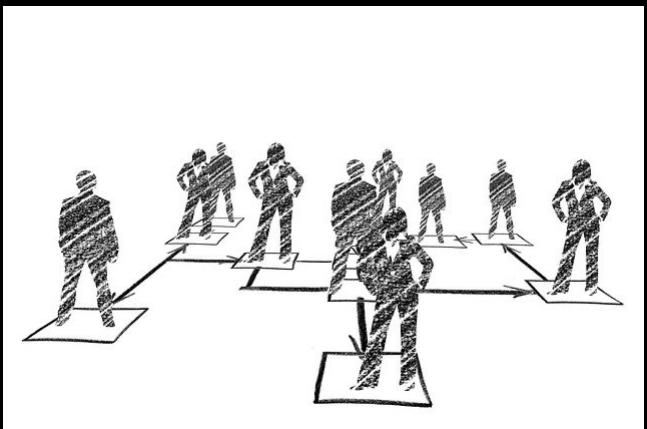
# Some things created and redistributed by apps



Happiness



Resources



Status



Capabilities

# Question for Discussion

What does Uber create and redistribute to its users (and other stakeholders)?

[Oral participation or  
Mentimeter word cloud]



# Question for Discussion

What does Instagram create  
and redistribute to its users  
(and other stakeholders)?

[Oral participation or  
Mentimeter word cloud]





Now I'm going to give you another example from my own life. In the '90s, people around me started talking about the Internet and web browsing. I remember the first time I went on the web. I was astonished. I could access newspapers at any time and every day. I could even search for any information by myself. I desperately wanted to help the blind people have access to the Internet, and I found ways to render the web into synthesized voice, which dramatically simplified the user interface.

This led me to develop the Home Page Reader in 1997, first in Japanese and later, translated into 11 languages.

- Chieko Asakawa, computer scientist



Any time we distribute things, this raises moral considerations of **distributive justice**.

# (Some) elements of distributive justice



**Efficiency:** making sure resources are allocated to where they are used most effectively.



**Equality/equity:** making sure everyone gets the same amount of resources/capabilities

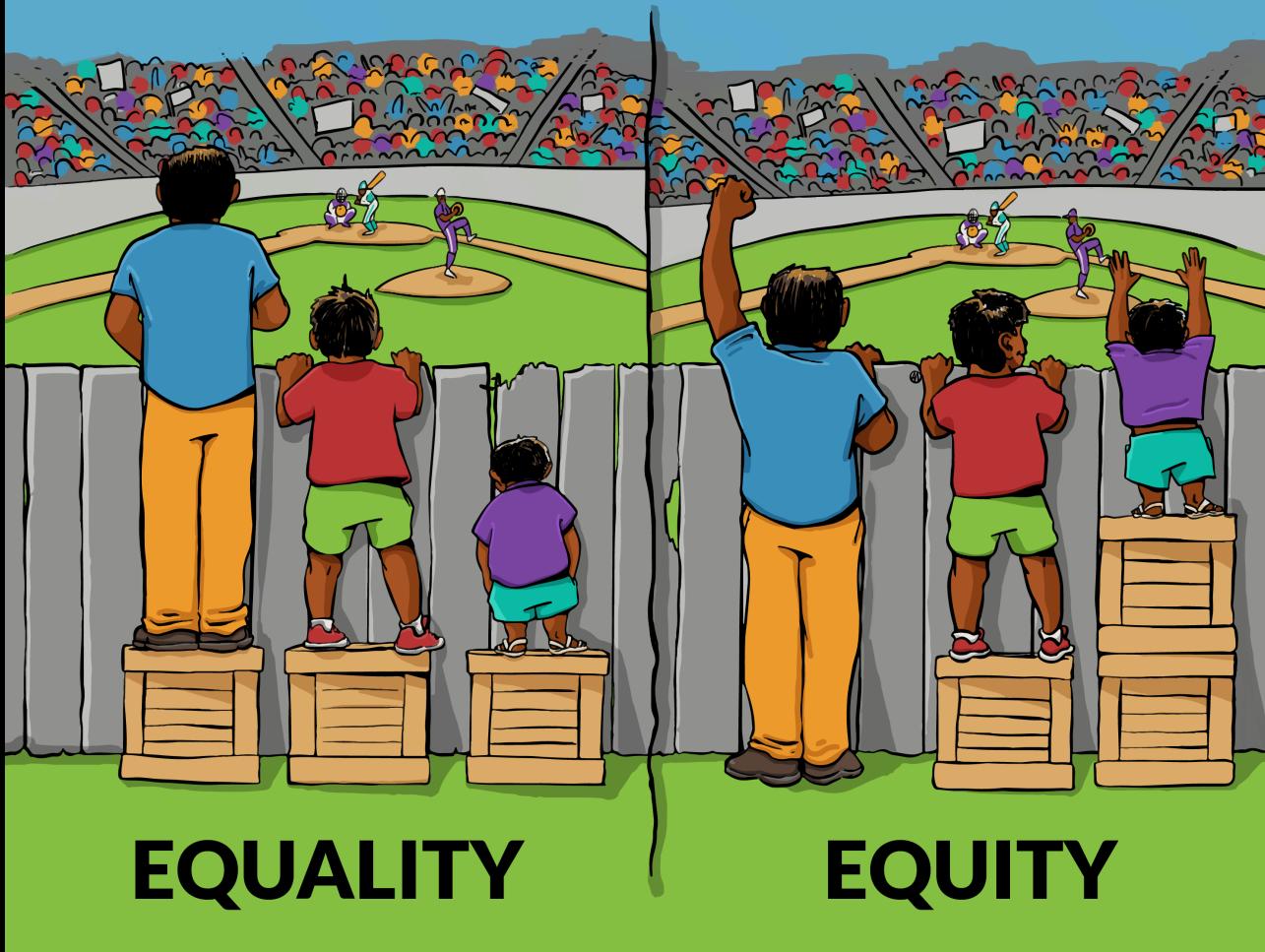


**Rights to a minimum:** making sure everyone has a bare minimum of something



**Luck egalitarianism:** making sure that no one is deprived of resources for reasons they are not responsible for.

# Equality or Equity



# Question for Discussion

In the following cases, which factor above (or none of them) do you think is most important for how we should distribute it?

- 1) The right to vote
- 2) Shelter
- 3) Sports cars

[poll]



# Question for Discussion

Out of {efficiency, rights to a minimum, equality, luck egalitarianism}, which best explains why you personally think it is important to put resources towards improving accessibility?

[poll, verbal discussion of it]



# Question for Discussion

We want to explore the question: Is morality enough to guarantee the production of accessible products?

What pressures might designers face, leading them to settle for a less accessible product?

[Oral participation or  
Mentimeter word cloud]





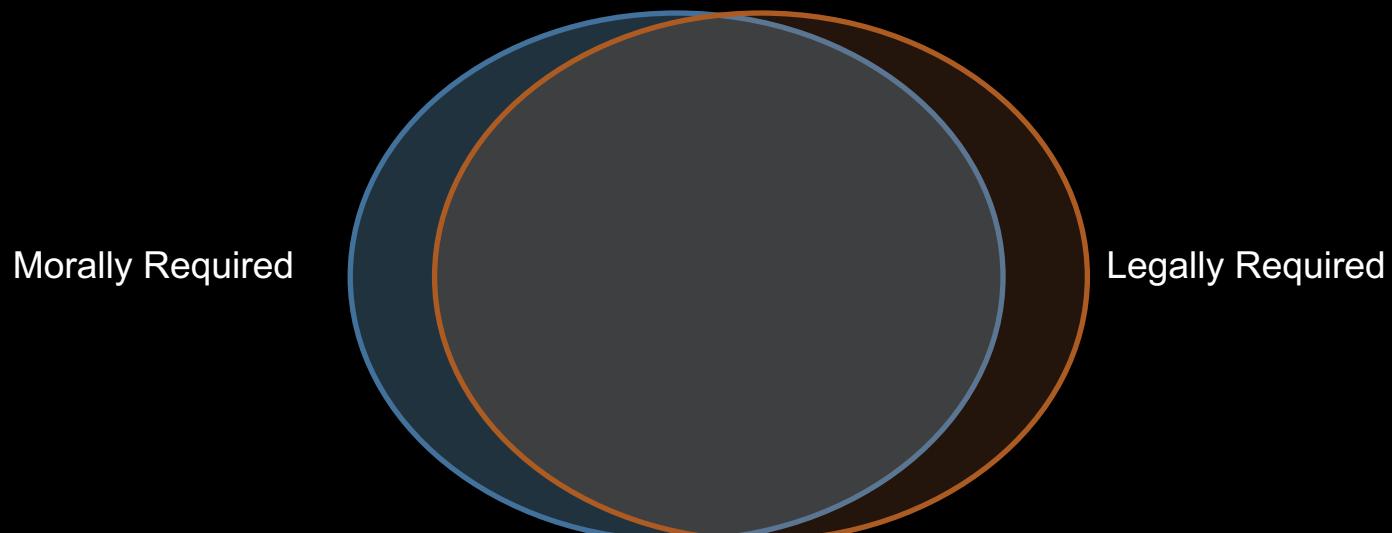
Morality is important, but it has limits.

- Morality is not necessarily enforced.
- Not everyone follows their moral reasons.
- There is substantial disagreement about morality.

# Part 2: The Law of Accommodation

There are **legal** obligations to accommodate people with disabilities.

- Legal requirements are backed by some kind of legal penalty (e.g. a fine).
- Legal requirements are usually contained in a legal code.
- Morality and law overlap, but they are not identical:





## Canadian Charter of Rights and Freedoms

15. (1) Every individual is equal before and under the law and has the right to the equal protection and equal benefit of the law without discrimination and, in particular, without discrimination based on race, national or ethnic origin, colour, religion, sex, age or mental or physical disability.

However, the Charter (mostly) only protects against discrimination by the state or government.



## Accessibility for Ontarians with Disabilities Act (AODA)

- In 2021, new regulations came into effect for websites.
- Most public and private organizations in Ontario are expected to make their websites compliant with Web Content Accessibility Guidelines 2.0 to the “AA” level (which includes the lower “A” level, too). (<https://www.ontario.ca/page/how-make-websites-accessible>)
- Corporations can be fined up to \$100,000 a day for non-compliance.
- Individuals can be fined up to \$50,000 a day for non-compliance.



# AODA

AODA website:

“Accessibility is good for both the economy and the community. The population of Ontarians with disabilities is steadily growing. Accessible information and employment make it possible for this growing group of people to contribute to the economy and society.... Similarly, accessible customer service allows people to exercise their spending power.” (<https://aoda.ca/what-is-the-aoda/>)



# AODA

AODA website:

“Finally, the standards of the AODA give all people **an equal footing** as they work, play, learn, teach, buy, sell, and use their diverse talents to benefit their communities and their province.”  
[\(https://aoda.ca/what-is-the-aoda/\)](https://aoda.ca/what-is-the-aoda/)

# Question for Discussion

What moral values do you see reflected in the government's arguments for AODA – efficiency, right to a minimum, strict equality, luck egalitarianism, or others?



# Summary

There are many reasons to make products accessible:

- Moral reasons (distributive justice, others...)
- Legal reasons (the Charter of Rights and Freedoms, AODA)

(And more...)

## 1.4.1 Use of Color — Level A

Color is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element.

*Note 1:* This success criterion addresses color perception specifically. Other forms of perception are covered in Guideline 1.3 including programmatic access to color and other visual presentation coding.

[Show techniques and failures for 1.4.1](#)

[Understanding 1.4.1](#)

[SHARE](#) | [BACK TO TOP](#)

## 2.3.1 Three Flashes or Below Threshold — Level A

Web pages do not contain anything that flashes more than three times in any one second period, or the flash is below the general flash and red flash thresholds.

*Note 1:* Since any content that does not meet this success criterion can interfere with a user's ability to use the whole page, all content on the Web page (whether it is used to meet other success criteria or not) must meet this success criterion. See Conformance Requirement 5: Non-Interference.

[Show techniques and failures for 2.3.1](#)

[Understanding 2.3.1](#)

[SHARE](#) | [BACK TO TOP](#)

## 1.4.2 Audio Control — Level A

If any audio on a Web page plays automatically for more than 3 seconds, either a mechanism is available to pause or stop the audio, or a mechanism is available to control audio volume independently from the overall system volume level.

*Note 1:* Since any content that does not meet this success criterion can interfere with a user's ability to use the whole page, all content on the Web page (whether or not it is used to meet other success criteria) must meet this success criterion. See Conformance Requirement 5: Non-Interference.

[Show techniques and failures for 1.4.2](#)

[Understanding 1.4.2](#)

[SHARE](#) | [BACK TO TOP](#)

# Some of Our Legal Responsibilities

<https://www.ontario.ca/page/how-make-websites-accessible>

These guidelines are based on  
<https://www.w3.org/WAI/standards-guidelines/wcag/>

# The Professional Answer (Code of Ethics)

- I. To uphold the highest standards of integrity, responsible behavior, and ethical conduct in professional activities.
  - 1. to hold paramount the safety, health, and welfare of the public, to strive to comply with ethical design and sustainable development practices, to protect the privacy of others, and to disclose promptly factors that might endanger the public or the environment;

Institute of Electrical and Electronics Engineers

<https://www.ieee.org/about/corporate/governance/p7-8.html>

# The Professional Answer (Code of Ethics)

II. To treat all persons fairly and with respect, to not engage in harassment or discrimination, and to avoid injuring others.

7. to treat all persons fairly and with respect, and to not engage in discrimination based on characteristics such as race, religion, gender, disability, age, national origin, sexual orientation, gender identity, or gender expression;

Institute of Electrical and Electronics Engineers

<https://www.ieee.org/about/corporate/governance/p7-8.html>

The University and other organizations also have similar policies (codes of conduct):

<https://governingcouncil.utoronto.ca/secretariat/policies/code-student-conduct-december-13-2019>

# Clean Architecture and Accessibility

Clean Architecture (or other layered architectures) makes it easier to update the accessibility features of your program.

Why? Because they should mostly be located in the outer layer.

For a more in depth discussion of the User Interface, consider taking courses in HCI (Human Computer Interaction) and UX (User Experience).

The Design of Interactive Computational Media

<https://artsci.calendar.utoronto.ca/course/csc318h1>

Human-Computer Interaction

<https://artsci.calendar.utoronto.ca/course/csc428h1>

Interactive Graph of CS courses

<https://courseography.cdf.toronto.edu/graph>

# Designing Accessible Software

You are not expected to figure this out on your own.

1. Accessibility checkers
2. Companies often hire consultants and/or beta testers. An excellent way to learn about someone's experience of using your software is to ask them. This is especially true if they have a disability that you have not experienced.
  - Example of a consulting company that specializes in testing, including for accessibility purposes: <https://www.ibeta.com/accessibility-testing/>

# Acknowledgements

This module was created as part of an Embedded Ethics Education Initiative (E3I), a joint project between the Department of Computer Science<sup>1</sup> and the Schwartz Reisman Institute for Technology and Society<sup>2</sup>, University of Toronto.

## Instructional Team:

Philosophy: Steven Coyne, Emma McClure

Computer Science: Lindsey Shorser, Paul Gries, Jonathan Calver

## Faculty Advisors:

Diane Horton<sup>1</sup>, David Liu<sup>1</sup>, and Sheila McIlraith<sup>1,2</sup>

**Department of Computer Science**

**Schwartz Reisman Institute for Technology and Society**

**University of Toronto**



Computer Science  
**UNIVERSITY OF TORONTO**





# References

- Canadian Charter of Rights and Freedoms, URL= <https://www.canada.ca/en/canadian-heritage/services/how-rights-protected/guide-canadian-charter-rights-freedoms.html#a2f>.
- Putnam, Daniel, David Wasserman, Jeffrey Blustein, and Adrienne Asch, "Disability and Justice", The Stanford Encyclopedia of Philosophy (Fall 2019 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/fall2019/entries/disability-justice/>.
- Robeyns, Ingrid and Morten Fibieger Byskov, "The Capability Approach", The Stanford Encyclopedia of Philosophy (Fall 2021 Edition), Edward N. Zalta (ed.), URL = <https://plato.stanford.edu/archives/fall2021/entries/capability-approach/>.
- Thomson, Greg. "What is the AODA?" URL= <https://aoda.ca/what-is-the-aoda/>

# REGULAR EXPRESSIONS

CSC207 SOFTWARE DESIGN



Computer Science  
UNIVERSITY OF TORONTO

# LEARNING OUTCOMES

- Understand the basics of regular expressions (regex)
- Be able to develop simple regular expressions in Java.

# DESCRIBING A SET OF CHARACTERS

- IDEs like IntelliJ might describe the Java naming conventions for variables this way:

$^ [ a-z ] [ a-zA-Z0-9 ] ^*$ \$

- This is a *regular expression* (or *regex*)
- This describes a pattern that appears in a set of strings. We say that any such string *matches* or *satisfies* the regular expression.

`^[a-zA-Z0-9]*$`

- The `^` character means that the pattern must start at the beginning of the string. This is called an *anchor*.
- Square brackets `[ ]` tell you to choose one of the characters listed inside.
  - In the leftmost set of brackets, we are given all lowercase English letters to choose from.
  - The second character will come from the second set of square brackets. It can be any lowercase letter, uppercase letter, or digit.
- The `*` means zero or more of whatever immediately precedes it.
- The `$` signifies the end of the string. This is another anchor.

## $^{\text{[a-z]}}[\text{a-zA-Z0-9}]*\$$ continued...

- So, our entire string must consist of letters and numbers, with the first character being a lower-case letter.
- Here are some examples:
  - x, numStudents, obj1
- These do not satisfy the regular expression. (Why not?)
  - Alphabet, 2ab, next\_value
- Do any of these strings match?
  - z3333, aBcB041, 78a

# WHAT IF THERE ARE NO ANCHORS?

- Here is another regular expression:
  - $[abc]C[a-e][24680]^*[A-Z]$
- When this regex is applied to a string, it will find the substrings that match.
  - cCaA matches the entire expression
  - ABCcCcCa1A23 contains substrings that match. For example, cCcc matches but not cCa1A.

# SPECIAL SYMBOLS

- A period . matches any character.
- Whitespace characters (the backslash is the escape character)
  - \s is a single space
  - \t is a tab character
  - \n is a new line character
- Just inside a square bracket, ^ has another meaning: it matches any character *except* the contents of the square brackets.
  - For example, [ ^aeiouAEIOU ] matches anything that isn't a vowel.

# CHARACTER CLASSES (MORE ESCAPES)

- You can make your own character classes by using square brackets like [q-z], [AEIOU], and [ ^1-3a-c ], or you can use a predefined class.

Construct	Description
.	any character
\d	a digit [0-9]
\D	a non-digit [^0-9]
\s	a whitespace char [ \t\n\x0B\f\r]
\S	a non-whitespace char [^\s]
\w	a word char [a-zA-Z_0-9]
\W	a non-word char [^\w]

# QUANTIFIERS

- \* means zero or more, + means one or more, and ? means zero or one.
- We append {2} to a pattern for exactly two copies of the same pattern, {2, } for two or more copies of the same pattern, and {2, 4} for two, three, or four copies of the same pattern.

Pattern	Matches	Explanation
a*	" 'a' 'aa'	zero or more
b+	'b' 'bb'	one or more
ab?c	'ac' 'abc'	zero or one
[abc]	'a' 'b' 'c'	one from a set
[a-c]	'a' 'b' 'c'	one from a range
[abc]*	" 'acbccb'	combination



# ESCAPING A SYMBOL

- Sometimes we want symbols to show up in the string that otherwise have meanings in regular expressions. To “escape” the meaning of the symbol, we write a backslash \ in front of it.
- A period . means any character. To have a period show up in the string, we write \>.
- Ex 1: abc123 matches the regex [ a-e ] [ a-e ] . +
- Ex 2: 1 . 4 matches the regex [ 0-9 ] \ . [ 0-9 ]



# REPETITION OF A PATTERN VS. A SPECIFIC CHOICE OF CHARACTER

- Here is a pattern that describes all phone numbers on the same continent:  
`\((\d\d\d)\) \d\d\d-\d\d\d\d`
- We could also write this as  
`\((\d{3})\) \d{3}-\d{4}`
- Here we want to repeat the pattern, but not necessarily the digit.  
`(123) 456-7890` matches the pattern.



# REPETITION OF EXACT CHARACTERS

- To repeat the same character twice, we use **groups** which are denoted by round brackets. Then we escape the number of the group we want to repeat:
- The string 124124a124 matches the regular expression:
- `(\d\d\d)\1a\1`
- Groups are assigned the number of open brackets that precede them.
- For example, `^(( [ab] )c)\2\1$` will repeat both groups. The strings that match are: acaac and bcbbc

# LOGICAL OPERATORS

- | means “or”
- && means the intersection of the range before the ampersands and the range that appears after. For example [ a-t&&[ r-z ] ] would only include the letters r, s, and t.

Construct	Description
[abc]	a, b, or c (simple class)
[^abc]	any char except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z inclusive (range)
[a-d[m-p]]	a through d or m through p (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z except for b and c (subtraction)
[a-z&&[^m-p]]	a through z and not m through p (subtraction)



# ANCHORS EXAMPLE

Pattern	Text	Result
$b^+$	abbc	Matches
$^b^+$	abbc	Fails (no b at start)
$^a^* \$$	aabaa	Fails (not all a's)



# REGEX IN JAVA

- The String class
  - `split`, `matches`, `replaceAll`, and `replaceFirst`
- The Pattern class
  - <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>
- The Matcher class
  - <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html>
- See `RegexMatcher.java` and `RegexChecker.java` on Quercus for small demos using these classes and some of their methods.

# OTHER RESOURCES

- Quick reference
  - <http://www.rexegg.com/regex-quickstart.html>
- Tutorial specific to Java
  - <http://tutorials.jenkov.com/java-regex/index.html>
- Regex crossword
  - <https://regexecrossword.com>
- Another short tutorial about escaping special characters in Java regex
  - <https://www.baeldung.com/java-regexp-escape-char>
- One of many online interfaces to experiment with regex
  - <https://regex101.com>

# HOMEWORK

- Try some puzzles  
on the regex  
crossword website



# FLOATING POINT

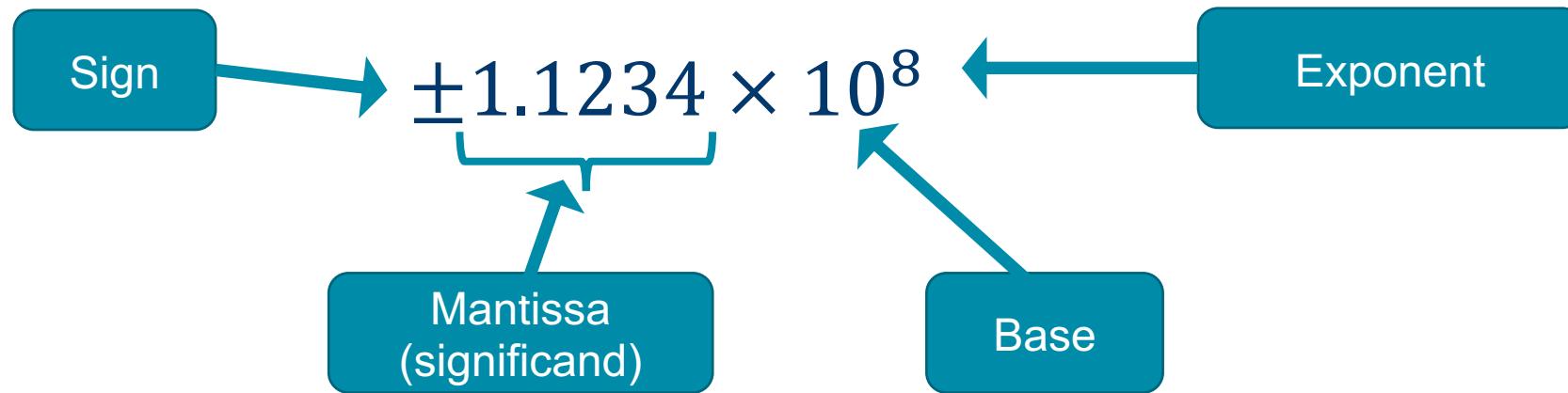
CSC 207 SOFTWARE DESIGN

# LEARNING OUTCOMES

- Understand the basics of how real numbers are represented on digital computers
- Gain an appreciation for the benefits of defining technical standards, through a discussion of the IEEE-754 floating point standard
- Have an idea of what a course like CSC336 has to offer, through a numerical experiment comparing algorithms for summing N numbers

# WHAT IS FLOATING POINT?

- A way to represent and work with real numbers on a digital computer.
- Just think scientific notation.



- Real numbers are continuous (infinite), but since digital computers are discrete (finite), floating point only **approximates** the real numbers.

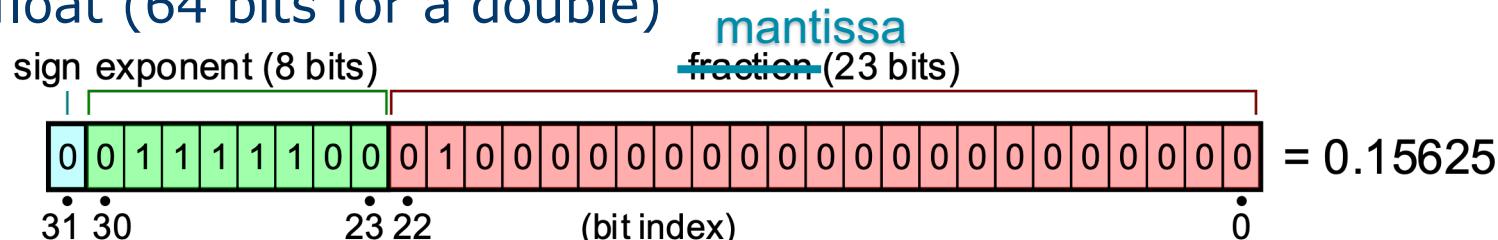
# HISTORICAL NOTE

- 30+ years ago, computer manufacturers each had their own standard for floating point.
- Problem? Impossible to write portable software! Different results on different hardware.
- In the late 1980s, the IEEE produced the standard that now virtually everyone follows.
- William Kahan spearheaded the effort and won the 1989 Turing Award for his work on IEEE-754.

# IEEE-754 FLOATING POINT STANDARD

- A **technical standard** established in 1985, with subsequent revisions in 2008 and most recently [2019](#)

- 32 bits for a float (64 bits for a double)



- For those interested:
  - <https://www.h-schmidt.net/FloatConverter/IEEE754.html>
  - <https://docs.oracle.com/javase/8/docs/api/java/lang/Float.html#floatToIntBits-float->
- Like most programming languages and hardware today, Java and the JVM conform to this standard ([Java SE 15 docs.oracle.com](https://docs.oracle.com/javase/15/docs/api/java/lang/Float.html#floatToIntBits-float-))

# **IEEE-754-2019 FLOATING POINT STANDARD**

## **Scope:**

This standard specifies **formats** and **operations** for floating-point arithmetic in computer systems.

**Exception conditions** are defined and handling of these conditions is specified.

# IEEE-754-2019 FLOATING POINT STANDARD

## Purpose:

This standard provides a method for computation with floating-point numbers that will **yield the same result** whether the processing is done in hardware, software, or a combination of the two.

**The results of the computation will be identical, independent of implementation, given the same input data.**

Errors, and error conditions, in the mathematical processing will be reported in a consistent manner **regardless of implementation**.

# NUMERIC TYPES IN JAVA

- float and double primitives for real numbers (and corresponding classes)
- byte, short, int, and long primitives for integers (and corresponding classes)
- <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>
- **We generally use double and int, but the others may be useful in certain situations.**
- Java also provides BigDecimal and BigInteger classes, which are slower but provide much more precision.
- The above classes are subclasses of class Number.
  - <https://docs.oracle.com/javase/8/docs/api/java/lang/Number.html>

# AN EXAMPLE – WHILE LOOP I



- How many times does the body of the loop run?
- What is the last value of i printed?

```
double i = 0;  
while (i < 1) {  
    i += 0.1;  
    System.out.println(i);  
}  
  
System.out.println(i == 1);
```

# AN EXAMPLE – WHILE LOOP I

- How many times does the body of the loop run?
- What is the last value of i printed?

```
double i = 0;  
while (i < 1) {  
    i += 0.1;  
    System.out.println(i);  
}  
  
System.out.println(i == 1);
```

## OUTPUT

```
0.1  
0.2  
0.3000000000000004  
0.4  
0.5  
0.6  
0.7  
0.7999999999999999  
0.8999999999999999  
0.9999999999999999  
1.0999999999999999  
false
```

In this case, rounding leads to values that are *slightly less* than expected, so we get an extra loop iteration beyond what we would have likely intended!



# AN EXAMPLE – WHILE LOOP II

- What if we use float instead of double?

```
float i = 0;  
while (i < 1) {  
    i += 0.1;  
    System.out.println(i);  
}  
  
System.out.println(i == 1);
```

# AN EXAMPLE – WHILE LOOP II

- What if we use float instead of double?

	OUTPUT
float i = 0;	0.1
while (i < 1) {	0.2
i += 0.1;	0.3
System.out.println(i);	0.4
}	0.5
	0.6
System.out.println(i == 1);	0.70000005
	0.8000001
	0.9000001
	1.0000001
	false

In this case, rounding leads to the value being *slightly larger*, so we happen to get 10 iterations. With every calculation, rounding occurs and you may lose correct digits.



# AN EXAMPLE – WHILE LOOP III

- Generally best to formulate loop conditions using integers where it is natural to do so:

```
int i = 0;
while (i < 10) {
    i += 1;
    System.out.println(i * 0.1);
}
```

```
System.out.println(i * 0.1 == 1);
```

# AN EXAMPLE – WHILE LOOP III

- Generally best to formulate loop conditions using integers where it is natural to do so

```
int i = 0;  
while (i < 10) {  
    i += 1;  
    System.out.println(i * 0.1);  
}
```

```
System.out.println(i * 0.1 == 1);
```

## OUTPUT

0.1	
0.2	
0.3000000000000004	
0.4	
0.5	
0.6000000000000001	
0.7000000000000001	
0.8	
0.9	
1.0	
true	

Remember that floating point isn't exact, so sometimes equality isn't as expected. That is why you will often see code that checks if the difference between two floating point numbers is sufficiently small rather than checking for strict equality.

# AN EXAMPLE – ADDING NUMBERS I

- We will consider the problem of summing  $N$  real numbers (stored as floats) and several algorithms for solving this problem.
  1. Sum the numbers, as given, using an accumulator of type float.
  2. Sort the numbers smallest to largest, then sum them as in algorithm 1.
- We are interested in the time each algorithm takes and how accurately they compute the sum.
- See code posted on Quercus (results shown on next slide).

# AN EXAMPLE – ADDING NUMBERS II

OUTPUT	$O(N)$	$O(N)$	$O(N) ?$	$O(N \log N)$
--------	--------	--------	----------	---------------

N	FloatAcc	FloatAccSorted		
10000	error 0.00000894	time ~ 1ms	error -0.00000021	6ms
20000	0.00002433	~ 0ms	0.00000098	6ms
40000	0.00004332	~ 1ms	-0.00000079	9ms
80000	0.00008664	~ 0ms	0.00000036	15ms
160000	0.00017586	~ 0ms	0.00000056	31ms
320000	0.00034582	~ 0ms	-0.00000004	53ms
640000	0.00072457	~ 1ms	-0.00000131	93ms
1280000	0.00142268	~ 2ms	-0.00000549	111ms

Experiment with sum calculated using a double precision accumulator as the reference solution. Each of the N numbers,  $x[i]$ , randomly drawn using:

$$x \in [10^{-6}, 1] \quad x[i] = \text{Math.pow}(10, -6 + \text{rand.nextDouble()}\ast 6)$$

# AN EXAMPLE – ADDING NUMBERS III

## SUMMARY

- Algorithm 1 had error growing proportional to  $N$  and time also proportional to  $N$
- Algorithm 2 was more accurate (adding numbers of more similar scale) but was slower since we had to sort the numbers too ( $\sim N \log N$ )
- It turns out Kahan found a better algorithm - [Kahan summation algorithm \(compensated summation\)](#) – with error independent of  $N$  and time proportional to  $4N$  (a constant amount of extra work for each number added to compensate for error).
- Also, remember that we used `float` in the experiment – type `double` gives us twice as many digits in our mantissa, so it would require summing many numbers of varying scales for this to manifest as a significant error.

Thought Questions: For a given application, which algorithm should you use?  
What algorithm does the `sum` function in Python use?

# AN EXAMPLE – ADDING NUMBERS IV

## SOME DOCUMENTATION

### DoubleAdder class in Java

- “The order of accumulation within or across threads is not guaranteed. Thus, **this class may not be applicable if numerical stability is required, especially when combining values of substantially different orders of magnitude.**”
- <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/DoubleAdder.html>

### fsum function in Python

- **“Avoids loss of precision by tracking multiple intermediate partial sums”**
- <https://docs.python.org/3/library/math.html#math.fsum>

# DESIGN OF THE EXPERIMENT CODE

- In our experiment, we were:
  - Running several algorithms
  - Reporting two quantities of interest: error – relative to a reference algorithm and time taken
  - Varying the input size ( $N$ )
  - Outputting the results to the console

Thought Questions about the provided experiment code:

How easy would it be to extend the code in certain ways?

E.g. Varying  $M$  inputs instead of just one,  
reporting a different number of quantities of interest than just two,  
specifying how many runs of each algorithm to average over,  
or outputting the results in different ways.

What parts of the design strike you as good / bad?

# CSC336: NUMERICAL METHODS (AKA SCIENTIFIC COMPUTING / APPLIED MATH)

- Explore the implications of performing numerical computation using approximations of continuous quantities.
- The study of computational methods for solving problems in linear algebra, non-linear equations, and approximation.
- The aim is to give students a basic understanding of both floating-point arithmetic and the implementation of algorithms used to solve numerical problems, as well as a familiarity with current numerical computing environments.

# 3 FAMOUS EXAMPLES OF NUMERICAL MISHAPS

In 1991, an American patriot missile failed to track and destroy an incoming missile; it hit a US Army barracks, killing 28 and injuring ~100 more.

The system tracked time in tenths of seconds. The error in approximating 0.1 with 24 bits was magnified in its calculations. The error was proportional to how long the computer had been running (100 hours).

At the time of the accident, the error corresponded to 0.34 seconds. A Patriot missile travels about half a km in that time!

# 3 FAMOUS EXAMPLES OF NUMERICAL MISHAPS

In 1996, the European Space Agency's Ariane 5 rocket exploded 40 seconds after launch.

**During conversion of a 64-bit to a 16-bit format, overflow occurred: the number was too big to store in 16 bits.**

This hadn't been expected since the readings (acceleration reported by the sensors) had never been this large before. But this new rocket was faster than its predecessor.

\$7 billion of R&D had been invested in this rocket.

# 3 FAMOUS EXAMPLES OF NUMERICAL MISHAPS

In 1992, the Sleipner A oil and gas platform sank in the North Sea near Norway

Numerical issues in modelling the structure caused shear stresses to be underestimated by 47%.

Cost: \$700 million

3

# SOME ADDITIONAL RESOURCES

Format strings for neatly displaying floats and other data types:

<https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax>

A light guide to floating point for programmers:

<https://floating-point-gui.de> and <https://floating-point-gui.de/languages/java>

A less light paper about floating point for programmers:

[https://docs.oracle.com/cd/E19957-01/806-3568/ncg\\_goldberg.html](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html)

odeToJava – a java project from usask for solving ODEs (somewhat like our simplified Experiment code, but with many more classes)

Code (can run in IntelliJ): <https://code.google.com/archive/p/odetojava/downloads>

Paper: <https://dl.acm.org/doi/10.1145/2641563>

# AN ASIDE

## STRICTFP

In Java SE 1.2 introduced the `strictfp` keyword.  
(<https://en.wikipedia.org/wiki/Strictfp>)

If `strictfp` is not specified, the JVM *may* give you different results for floating-point calculations — depending on the hardware it is running on, as it may choose to make use of extended precision.

It appears that at least recently in openjdk-17, they are going back to always strict.

<https://bugs.openjdk.java.net/browse/JDK-8175916>

Just like Python, Java is still evolving!

# HOMEWORK

---

- No homework!



# THE INTERNSHIP PROCESS: APPLICATION TO OFFER

CSC 207 SOFTWARE DESIGN



# LEARNING OUTCOMES

- Understand how you can prepare for job and internship interviews.

# STEP 0: PREPARATION

- Network
  - Workshops & Events for technology & industry you are interested in
  - Online platforms
    - LinkedIn, [10KC](#), Mentorship programs
  - Meetups & Social events
    - Email the organizers and ask questions!
    - HackerNest, TechTO, DevTO, etc.
    - <https://www.meetup.com>
    - <https://www.eventbrite.ca/d/canada--toronto/tech-meetup/>
  - Newsletters
    - MaRS, U of T Entrepreneurship
    - DCS Undergrad office
- Experience
  - Side projects
  - Open source contributions

# STEP 1: APPLICATION

- Find interesting companies & teams
  - Direct Search by technology/industry
  - Via a mutual connection/friend
  - Research
  - Ideally, you know someone there to reach out to for a referral or an “informational interview”
  - Job websites (see list on last slide)
- Apply to the position
  - 1-page resume
  - Contact the hiring manager directly
- Large number of generic applications vs. several targeted applications
  - Some companies will toss applications whose cover letters are not customized!
- Optimize for learning, mentorship, and impact (and maybe brand)
  - Optimizing for salary is almost always a mistake



# STEP 2: FIRST SCREEN

- Phone screen:
  - Basic questions (interests, goals, culture fit, legal)
  - Review of resume and prior experience
  - A few small tech questions (or coding ones for companies like Google)
- Tech challenge:
  - Small coding challenge to do anywhere between a few hours to one week
  - Be ready to write code
  - Write clean code (conventions, tests, naming, design patterns, good — not excessive — comments)

# STEP 3: ONSITE INTERVIEWS

- Usually 1–2 onsite interviews
  - Technical and behavioral
  - Cracking the coding interview is a go-to book for this
  - They want to hear how you think
  - Ask questions to understand and clarify
  - Restate the problem, the goal, and the important criteria
  - Explicitly talk about your assumptions and decisions
  - Communicate, communicate, communicate: **think out loud**
  - BEFORE the interview, look up the team (LinkedIn, company website)
    - Always have questions about the team, company, product, what you'll learn
- Questions they want answers to:
  - What is it like to work with you?
  - Can you learn?
  - Do you take initiative?

# STEP 4: OFFER

- Understand your offer:
  - Compensation
  - Team, direct supervisor
  - Vacation, benefits, etc.
  - Always ask for things you want. Don't hold back or be shy.
    - But be nice
    - Do your research
- Feel free to negotiate AFTER you receive a written offer but before you have accepted
  - <https://capd.mit.edu/blog/2021/06/11/how-to-ask-for-a-raise-or-negotiate-your-salary/>
- Know the rough salary levels!
  - <https://www.levels.fyi/?compare=Amazon,Microsoft,Google&track=Software%20Engineer>

# RESOURCES

- How to write a cover letter
  - <https://www.youtube.com/watch?v=P8ci7tnlhEo>
- How to use your social media to get a job
  - <https://www.youtube.com/watch?v=xT6rlbHXC8E>
- Internship websites
  - <https://ca.indeed.com>, <https://talentegg.ca>, <https://www.workopolis.com>,  
<https://www.careeredge.ca>
    - Do a web search for more! There are dozens.
  - <https://hbr.org/2019/07/3-questions-hiring-managers-want-you-to-answer>

# LEETCODE

## HTTPS://LEETCODE.COM

Repository of coding problems that emphasize data structures and algorithms and invariant thinking

<https://leetcode.com/explore/interview/card/top-interview-questions-easy/>

Information about specific coding questions companies have used in the past

<https://leetcode.com/discuss/interview-question/352460/Google-Online-Assessment-Questions>

Practice coding interviews in a timed environment

<https://leetcode.com/assessment/>

# SOME SELECT LEETCODE PROGRAMMING PROBLEMS

- <https://leetcode.com/problems/binary-search-tree-iterator/>
- <https://leetcode.com/problems/smallest-number-in-infinite-set/>
- <https://leetcode.com/problems/valid-palindrome-ii/>
- <https://leetcode.com/problems/valid-palindrome/>
- <https://leetcode.com/problems/integer-to-roman/>
- <https://leetcode.com/problems/path-sum/>
  - <https://leetcode.com/problems/path-sum-ii/>
  - <https://leetcode.com/problems/path-sum-iii/>
- <https://leetcode.com/problems/sum-of-left-leaves/>
- <https://leetcode.com/problems/range-sum-of-bst/>
- <https://leetcode.com/problems/even-odd-tree/>

# GOOGLE CODE JAM

- Another great source for interesting coding problems. They include explanations too!
- <https://codingcompetitions.withgoogle.com/codejam/archive>
- A Couple examples:
  - <https://codingcompetitions.withgoogle.com/codejam/round/00000000000000cb/0000000007966#problem>
  - <https://codingcompetitions.withgoogle.com/codejam/round/0000000000432f3a/000000043324b#problem>

# OTHER TIPS

- Know time and space complexity
  - Exponential = fired, polynomial = try again,  $n \log n$ ,  $n$ ,  $C$  = good job
- Try problems (15-20 min.), then study the solution
- Study common patterns (e.g. <https://hackernoon.com/14-patterns-to-ace-any-coding-interview-question-c5bb3357f6ed>, <https://medium.com/leetcode-patterns>)
- “Memoization” - easy problems can often be solved via this technique
- Work with other people!
- <https://igotanoffer.com/blogs/tech/coding-interview-questions>
- <https://igotanoffer.com/blogs/tech/coding-interview-prep>

# HOMEWORK

- Try out some LeetCode or Code Jam problems if you haven't before!





# COURSE SUMMARY

CSC 207 SOFTWARE DESIGN



# LEARNING OUTCOMES

- Identify what you know and what you feel you need to review before the exam!

# COURSE EVALUATIONS

- Please do them!
  - Constructive criticism is appreciated.
- We use the feedback to improve the course.
  - You can also email the course address to provide feedback — this is best for overall thoughts on how the course is structured, as the evals are by lecture section, but 207 has coordinated sections.
- Course evals are also used for promotion and end-of-year evaluation.
  - Equivalent of the grades you get in your courses as a student!

# FINAL EXAM ADVICE I

It is your chance to show us what you know. You can do this by:

- Carefully reading each question and looking at how many marks things are worth.
- **Using terminology we've introduced in the course and clearly expressing yourself.**
- Writing something down for every question — we can't give you any credit for a blank page!
- **Being explicit in your answers — we can only give you credit for what you wrote down and clearly explained.** Use examples when you can.

# FINAL EXAM ADVICE II

## Example Question

- Explain why having a Square be a subclass of a Rectangle is an example of a Liskov violation. [2 marks]

## Minimal Answer

- Because a square is more constrained than a rectangle. (**We need you to elaborate though – this does not look like a 2 mark answer!**)

But...

*what is a Liskov violation?*

*How is it more constrained?*

*Would a picture or small code example help?*

**We want you to convince us that you understand the concepts!** (Think of the question as a prompt — but make sure to stay on topic! Be clear and concise in your answers!)



Computer Science  
UNIVERSITY OF TORONTO

# COMMUNICATION

- Communication is important, when working with other people. Similarly, it is how we know to give you marks.
- For the exam, you should be able to communicate design ideas using:
  - Code
  - Words
  - CRC
  - UML diagrams
- You should also be able to read and interpret design ideas from other people in each of the above formats. (example: write code based on a CRC model)
- There is rarely one correct answer when solving a design problem. But there are better answers and worse ones. You should be able to communicate the good and bad aspects of a design decision using terminology from the course.

# PROFESSIONAL SKILLS

- A large part of this course aimed to give you experience that will be helpful in your professional life.
  - Working as part of a group
  - Designing a program that is more complex
  - Using version control
  - Reading documentation and being self-sufficient, but asking for help as needed.
- **Reflecting on what worked —and what didn't — will help you make better design decisions in the future!**
- If your code is easy to read, navigate, extend, and work on at the same time as other people work on it, then it probably follows SOLID, is well encapsulated, and takes advantage of the features of Object-Oriented Design!



# FEATURES OF OBJECT-ORIENTED PROGRAMMING LANGUAGES

- *Abstraction* — the process of distilling a concept to a set of essential characteristics.
- *Encapsulation* — the process of binding together data with methods that manipulate the data and hiding the internal representation from the user.
  - The result of applying abstraction and encapsulation is (often) a class with instance variables and methods that together model a concept from the real world. (Further reading: what's the difference between Abstraction, Encapsulation, and Information Hiding?)
- *Inheritance* — the concept that when a subclass is defined in terms of another class, the features of that other class are inherited by the subclass (enabling reuse of common code).
- *Polymorphism* (“many forms”) — the ability of an expression (such as a method call) to be applied to objects of different types.

# SOLID DESIGN PRINCIPLES



# FUNDAMENTAL OOD PRINCIPLES

SOLID: five basic principles of object-oriented design

(Developed by [Robert C. Martin](#), affectionately known as “Uncle Bob”.)

Single responsibility principle (SRP)

Open/closed principle (OCP)

Liskov substitution principle (LSP)

Interface segregation principle (ISP)

Dependency inversion principle (DIP)



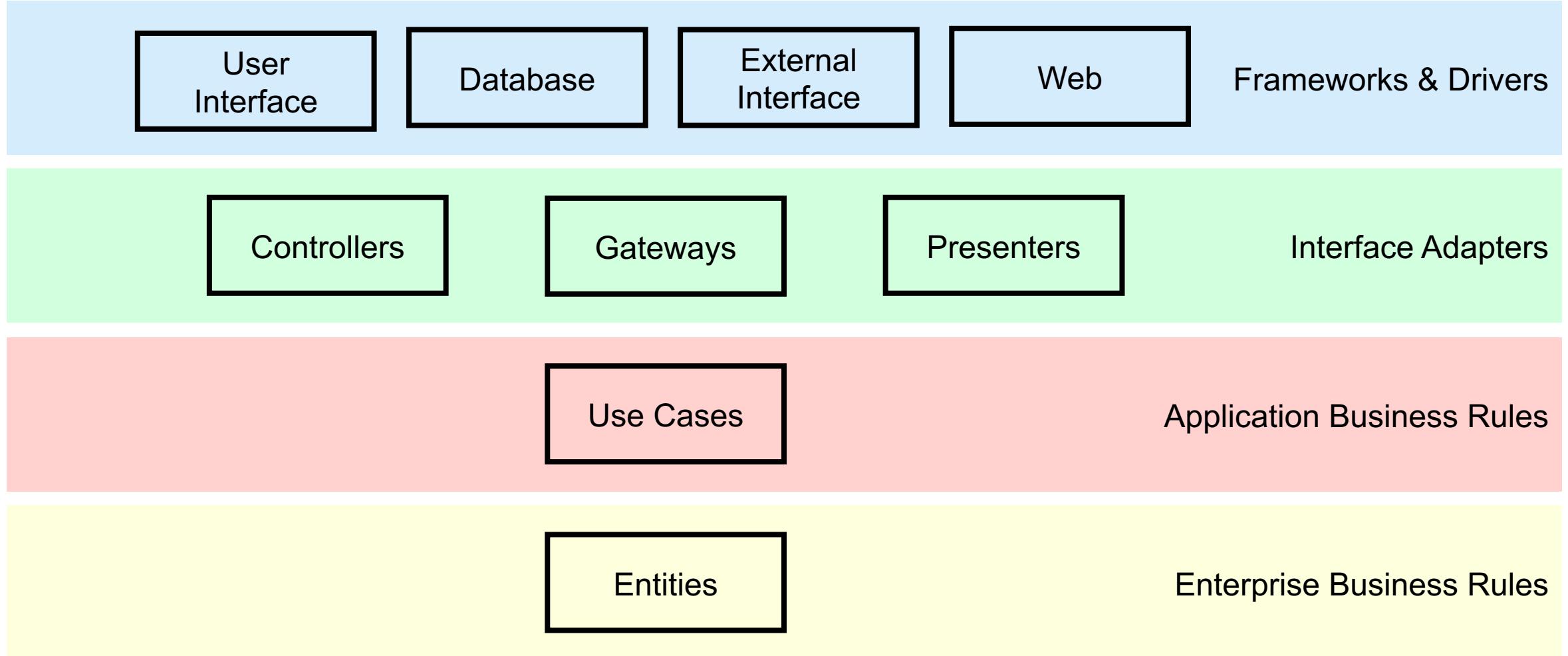
# CLEAN ARCHITECTURE

CSC207 SOFTWARE DESIGN



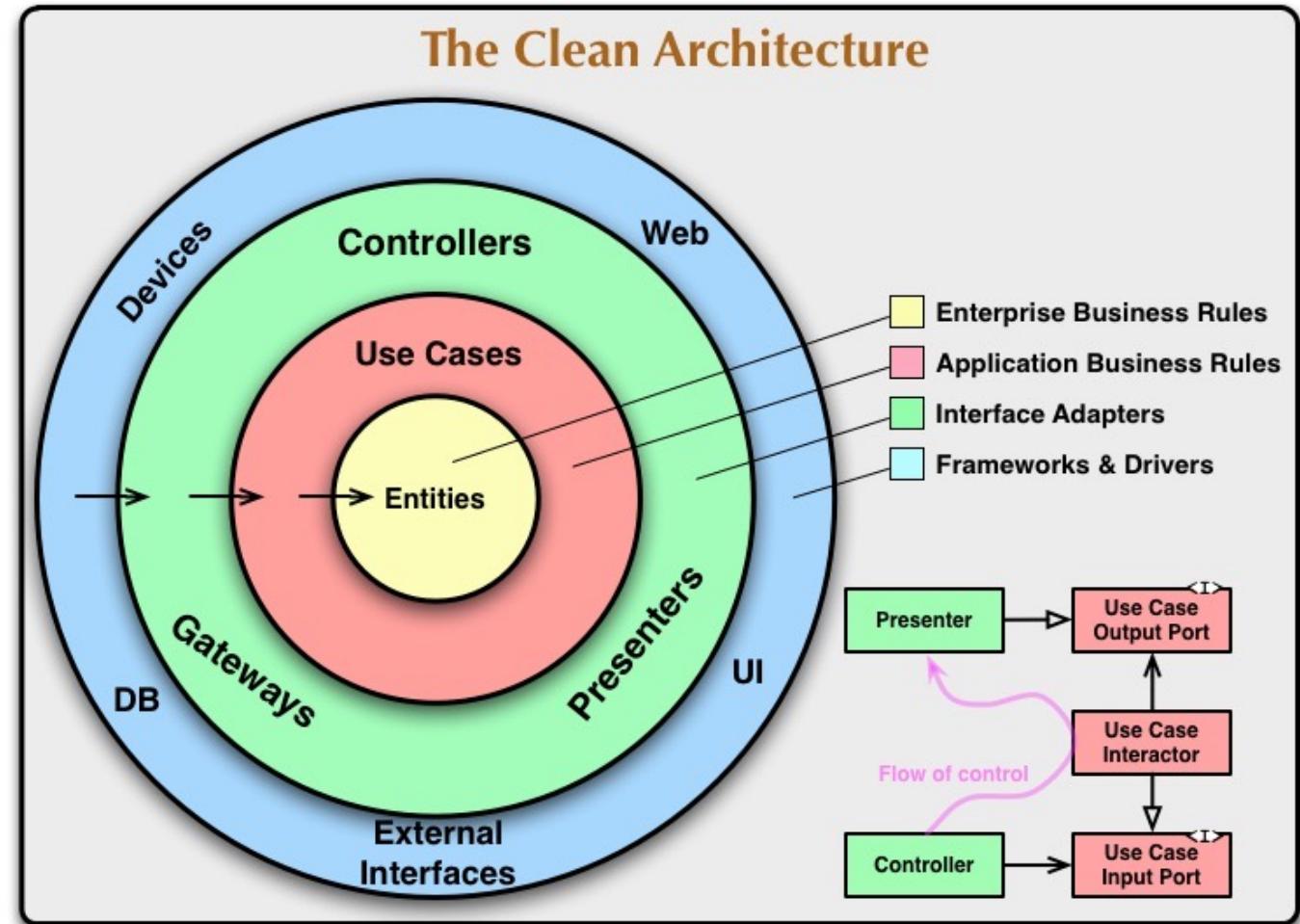
Computer Science  
UNIVERSITY OF TORONTO

# CLEAN ARCHITECTURE



# CLEAN ARCHITECTURE

- Often visualize the layers as concentric circles.
- Reminds us that Entities are at the core
- Input and output are both in outer layers



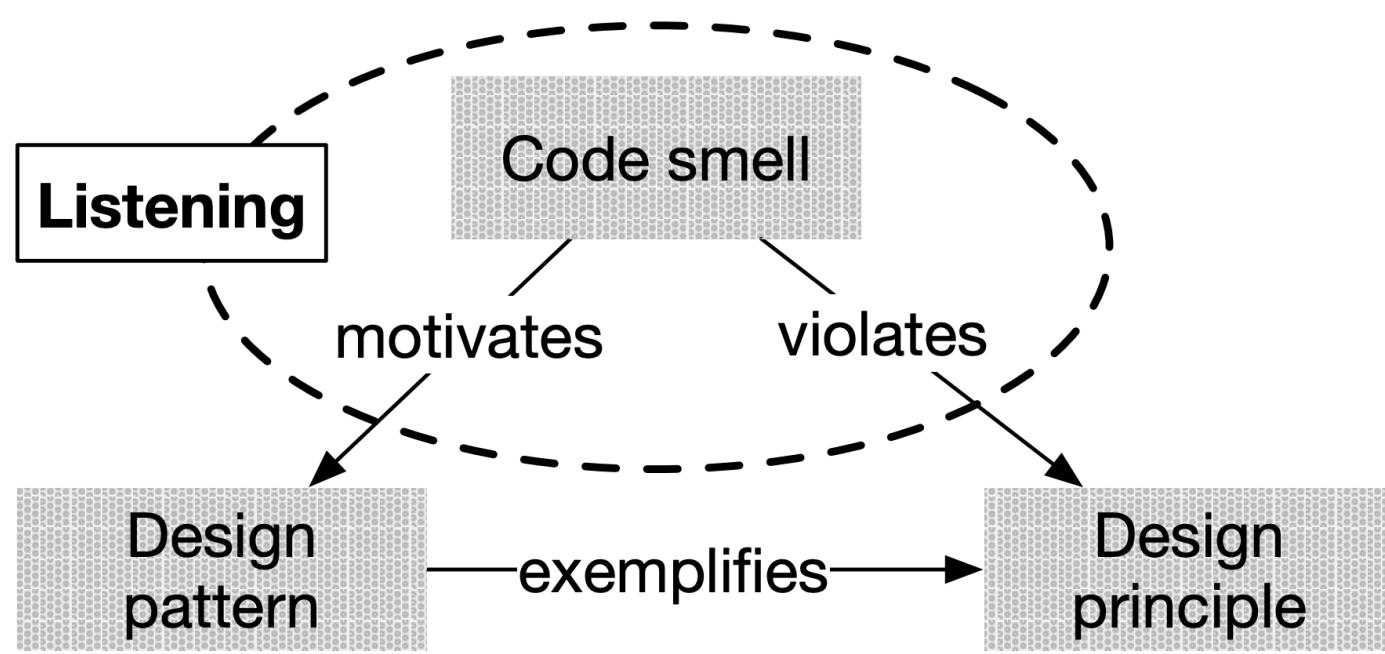
# CLEAN ARCHITECTURE – DEPENDENCY RULE

- Dependence on adjacent layer — from outer to inner
- Dependence within the same layer is allowed (but try to minimize coupling)
- On occasion you might find it unnecessary to explicitly have all four layers, but you'll almost certainly want at least three layers and sometimes even more than four.
- **The name of something (functions, classes, variables) declared in an outer layer must not be mentioned by the code in an inner layer.**
- How do we go from the inside to the outside then?
  - Dependency Inversion!
  - An inner layer can depend on an interface, which the outer layer implements.

# BENEFITS OF CLEAN ARCHITECTURE

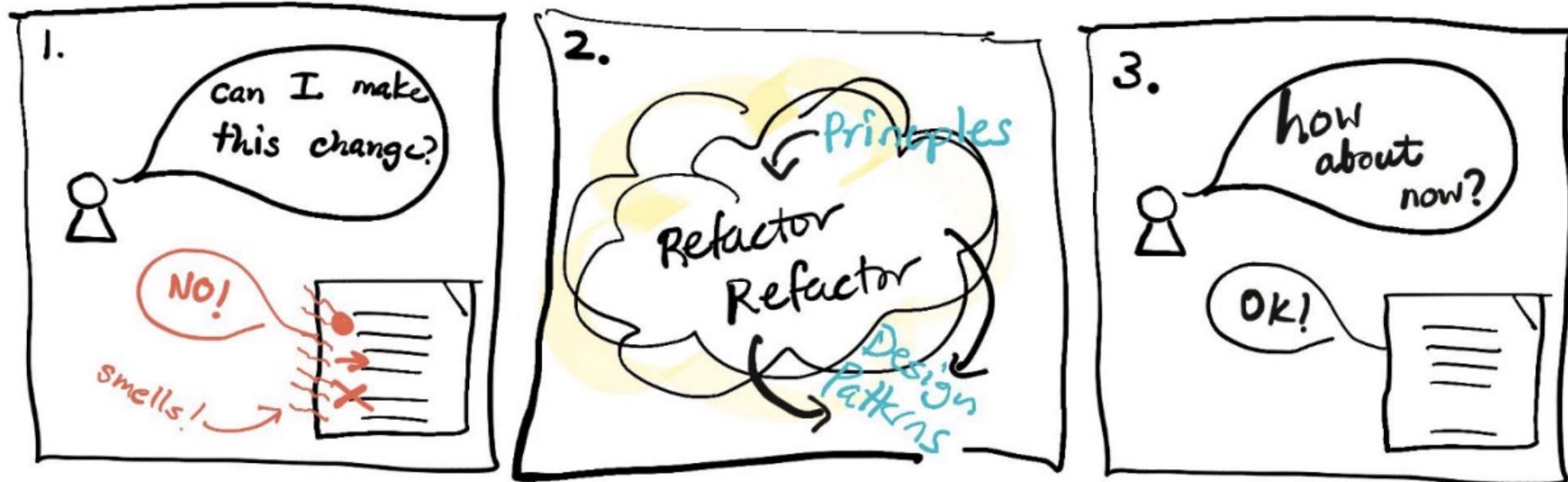
- All the “details” (frameworks, UI, Database, etc.) live in the outermost layer.
- The business rules can be easily tested without worrying about those details in the outer layers!
- Any changes in the outer layers don’t affect the business rules!

# LEARNING TO LISTEN FOR DESIGN



**Figure 2.** Integration of code smells, design patterns, and design principles into a single conceptual structure.

# LEARNING TO LISTEN FOR DESIGN



**Figure 1.** Seasoned programmer's conversation with code.

Having a strong understanding of design principles and familiarity with common design patterns will help you write better code and fix problematic code!

# CATEGORIES OF CODE SMELLS

- Bloaters – too much code
- Object Orientation Abusers – can be improved by changing use of inheritance, composition, or by redistributing responsibilities
- Change Preventers – features that make it difficult to extend or update the code
- Dispensables – things that can be deleted or combined
- Couplers – the opposite of "lessening dependencies"
- etc

# CATEGORIES OF CODE SMELL FIXES

The solution to a code smell is almost always "refactor"

<https://refactoring.guru/refactoring/techniques>

- Composing Methods
- Moving Features Between Objects
- Organizing Data
- Simplifying Conditional Expressions
- Simplifying Method Calls
- Dealing With Generalization

# DESIGN PATTERNS



# DESIGN PATTERNS

- Iterator, Observer, Strategy, Dependency Injection, Simple Factory, Façade, Builder
- [https://sourcemaking.com/design\\_patterns/](https://sourcemaking.com/design_patterns/) has detailed explanations of these and many more design patterns.

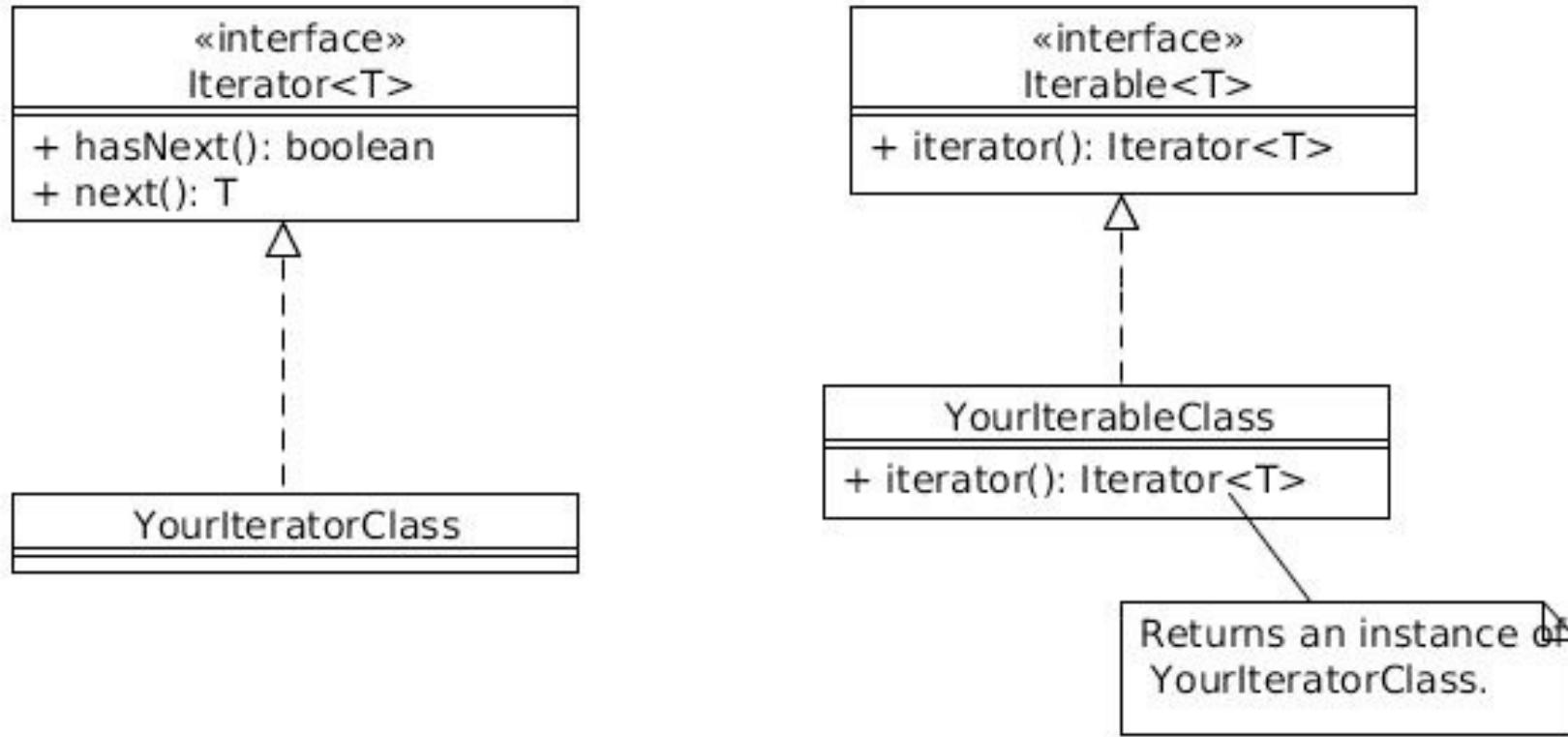
# DESIGN PATTERNS (REVIEW)

- A **design pattern** is a general description of the solution to a well-established problem.
- Patterns describe the shape of the code rather than the details.
- They're a means of communicating design ideas.
- They are not specific to any one programming language.
- You'll learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).

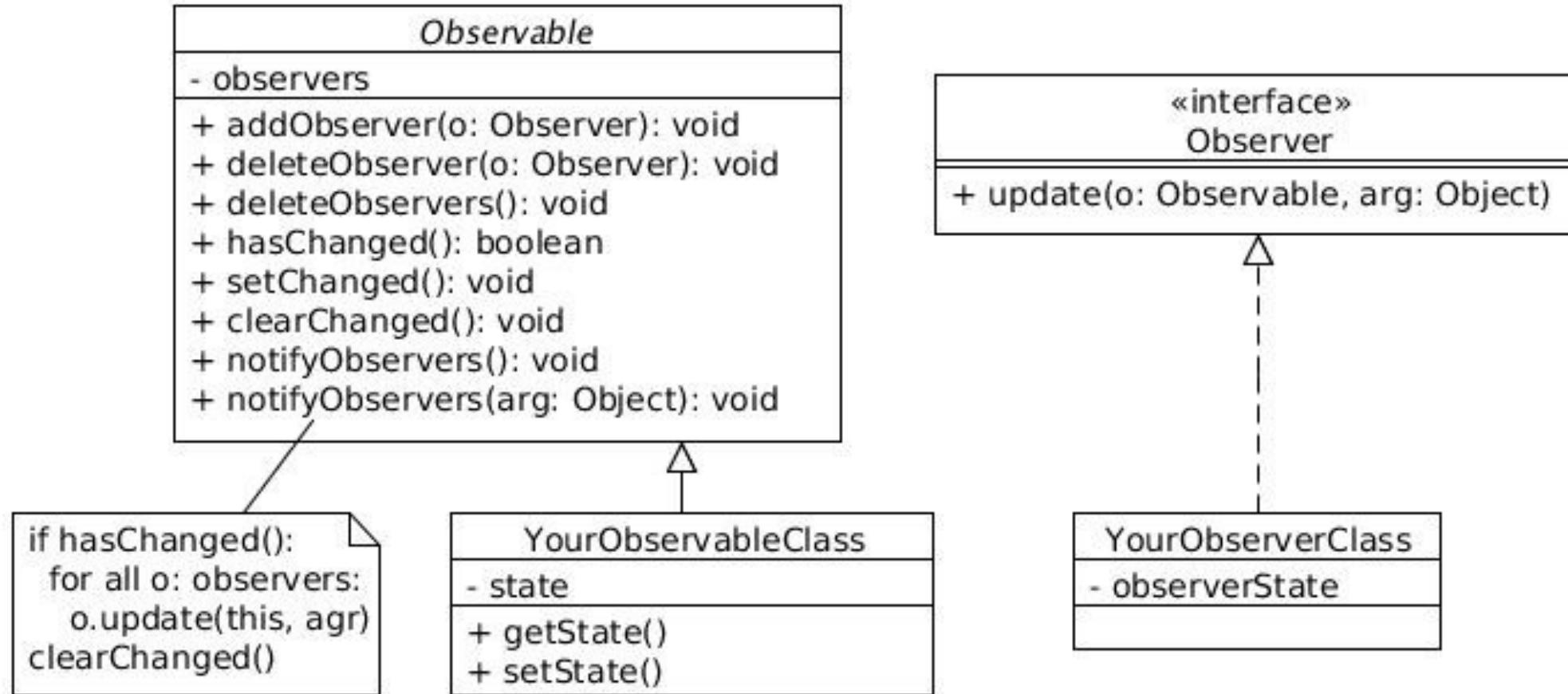
# REMINDER: LOOSE COUPLING, HIGH COHESION

- These are two goals of object-oriented design.
- **Coupling:** the interdependencies between objects. The fewer couplings the better, because that way we can test and modify each piece independently.
- **Cohesion:** how strongly related the parts are inside a class. High cohesion means that a class does one job, and does it well. If a class has low cohesion, then an object has parts that don't relate to each other.
- Design patterns are often applied to decrease coupling and increase cohesion.

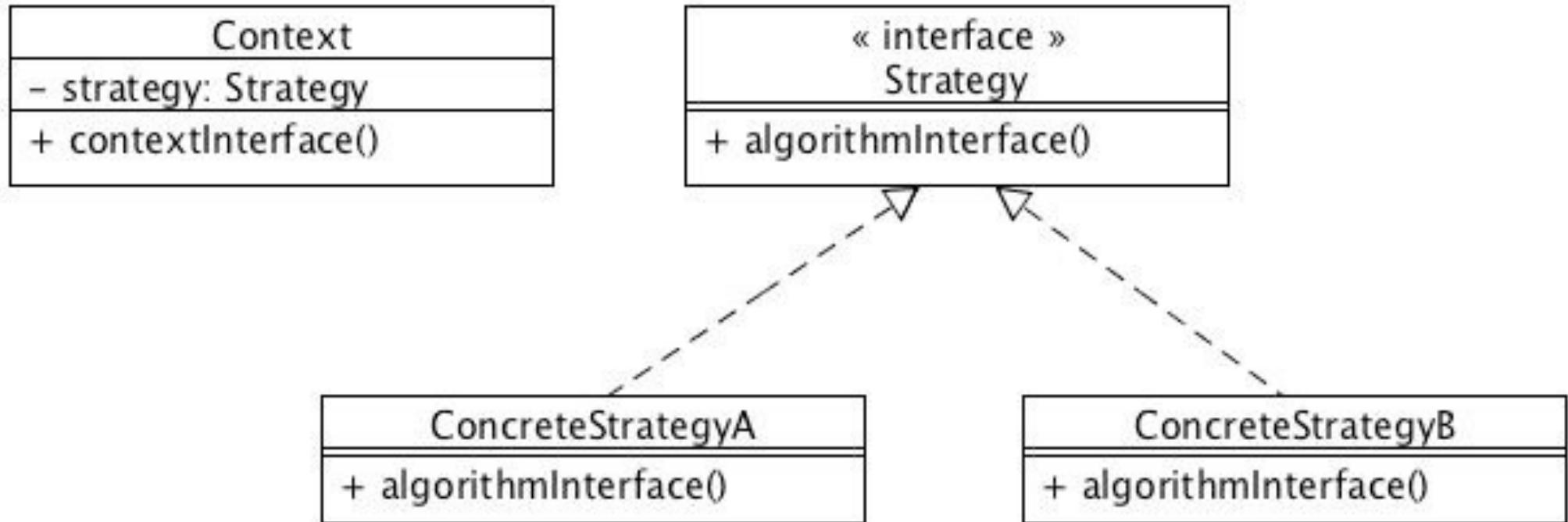
# ITERATOR DESIGN PATTERN



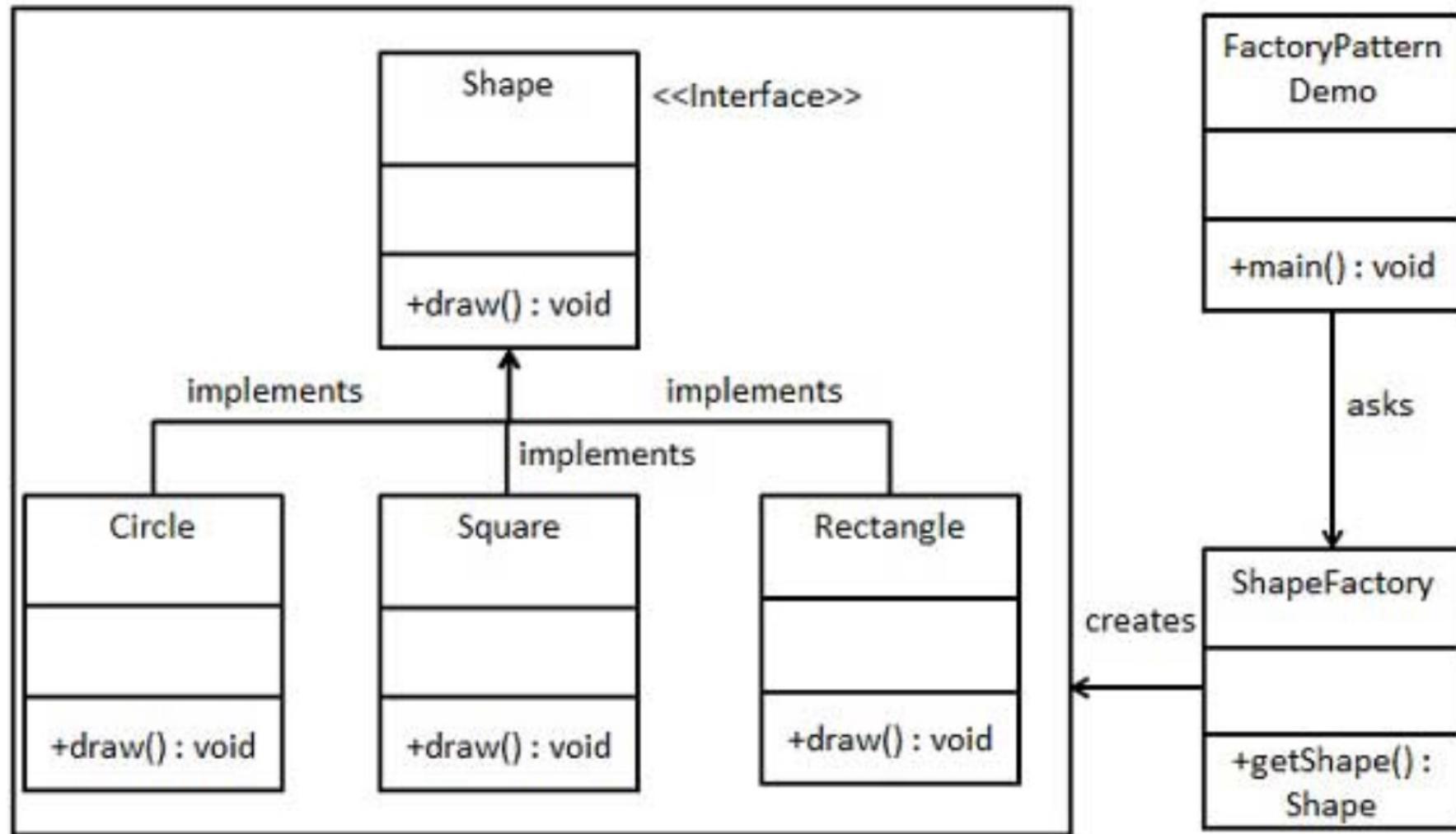
# OBSERVER: JAVA IMPLEMENTATION



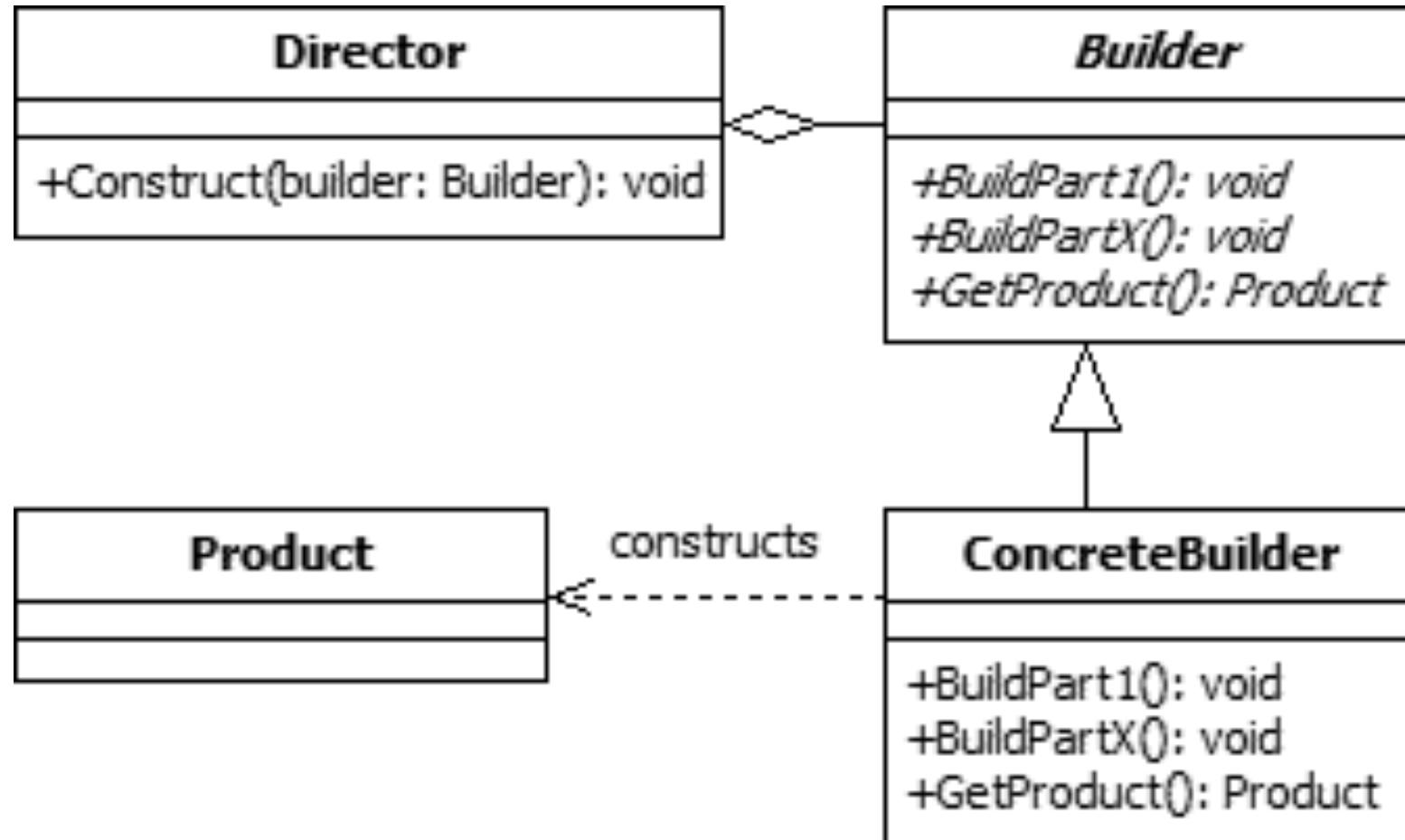
# STRATEGY: STANDARD SOLUTION



# FACTORY : AN EXAMPLE



# BUILDER DESIGN PATTERN

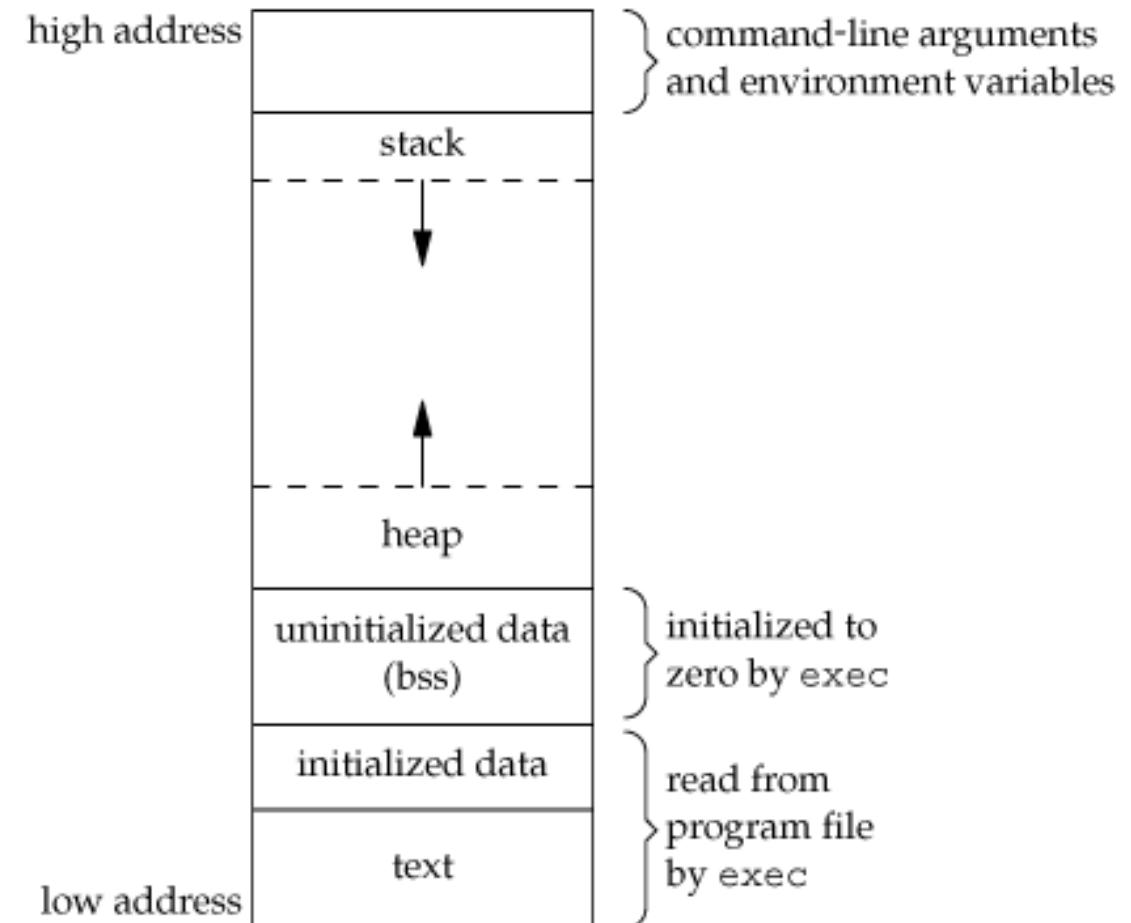


# OTHER TOPICS

- Embedded Ethics
  - models of disability
  - accessibility
- Regex
  - the basics
  - Java code for regex
- Floating Point
  - approximation of real numbers
  - numerical error

# NEXT COURSE: CSC209

- C and shell programming
- More about build systems / how compilation works
- Git from the command line
- Low-level programming
- Memory management (Java does this for you)
- Even more about low-level in CSC369: Operating System!



New view of computer memory

<https://medium.com/@vikasv210/memory-layout-in-c-fe4dffdaeed6>

# OTHER COURSES

Graph of how the various CS courses connect

- <https://courseography.cdf.toronto.edu/graph>

Collection of CS course syllabi

- <https://web.cs.toronto.edu/undergraduate/courses/outlines>

Many topics!

- **Software Engineering**
- **Compilers**
- **Programming languages**
- **Networks**
- **Operating Systems**
- Algorithms
- Theory of Computation
- Computer Vision
- Machine Learning
- Numerical Computation
- Natural Language
- Processing
- Artificial Intelligence
- **Human Computer Interaction**
- **Web Programming**
- **Computers and Society**





# THANK YOU FOR AN AWESOME SEMESTER!



**UNIVERSITY OF TORONTO  
Faculty of Arts & Science**

# **DECEMBER 2021 EXAMINATIONS**

CSC 207 H1F

## **Duration: 2 Hours**

### **Aids Allowed: None**

*Do not turn this page until  
you have received the signal to start.*

In the meantime, write your name, student number, and UTORid below (please do this now!) and *carefully* read *all* the information on the rest of this page.

**First (Given) Name(s):**

**Last (Family) Name(s):**

\_\_\_\_\_

**10-Digit Student Number:**

UTORid (e.g., **pitfra12**):

--	--	--	--	--	--	--	--	--

## MARKING GUIDE

- This final examination consists of 8 questions. The last question consists of 8 multiple choice questions, whose answers must be entered on the bubble sheet on the last page of the exam. *When you receive the signal to start, please make sure that your copy of the examination is complete.*
  - Answer each question directly on the examination paper, in the space provided, and use a “blank” page for rough work. If you need more space for one of your solutions, use one of the “blank” pages and *indicate clearly the part of your work that should be marked.*
  - *Remember that, in order to pass the course, you must achieve a grade of at least 40% on this final examination.*
  - As a student, you help create a fair and inclusive writing environment. If you possess an unauthorized aid during an exam, you may be charged with an academic offence.

Nº 1: \_\_\_\_\_ / 8

Nº 2: / 8

Nº 3: / 8

Nº 4: / 8

Nº 5: / 8

Nº 6 / 6

Nº 7: / 6

Nº 8. / 8

TOTAL: /60

**Question 1.** Clean Architecture [8 MARKS]

**Part (a)** [3 MARKS] List the layers of the Clean Architecture in order from the innermost to the outermost layer and include a one-sentence description of each layer.

**Part (b)** [3 MARKS] What is dependency inversion and how can it be used in Clean Architecture?

DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

Duration: **2 Hours**

**Part (c) [2 MARKS]** Briefly explain one benefit that Clean Architecture brings to a large program.

---

You probably don't need the space below, but use it if you want.

**Question 2.** Contact Tracing Specification [8 MARKS]

Consider a contact tracing application that keeps track of users, the locations they have visited, and the time they visited each location. When a person visits a physical location, they check into that location in the program. Upon login, the user is notified whether or not they have been in the same location at around the same time as someone who has tested positive for COVID-19 in the past two weeks. If a healthcare worker logs into the program, they can enter the name of any user who has been diagnosed with COVID-19. The system can also keep track of outbreaks associated with specific locations.

**Part (a)** [2 MARKS] List the names of 2 (TWO) entity classes.

**Part (b)** [1 MARK] State a use case related to one or both of those entities.

**Part (c)** [5 MARKS] Choose a design pattern that you think could be useful to implement in such a program. Briefly explain how the pattern works and explain why you think it will be useful.

DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

Duration: **2 Hours**

**Question 3.** Project [8 MARKS]

Choose a major design decision that your group made during your project. Using terminology from the course where appropriate, explain what your options were, what you ended up doing, and why. **Be specific and provide enough detail so that someone unfamiliar with your project can understand.**

**Question 4.** Round Robin Iterator [8 MARKS]

Consider the following class:

```
class RoundRobinCollection<T> implements Iterable<T>
    // Implementation details omitted
```

RoundRobinCollection keeps track of a collection of other Iterable objects in the order that they are added to the collection. When iterating through a RoundRobinCollection, the items are traversed in "round-robin order"<sup>1</sup>. Here is an example:

Note: List.of simply converts a given array into a list in the example code.

```
// A few Iterables.
List<String> list1 = List.of(new String[]{"1A", "1B", "1C"});
Set<String> set2 = new HashSet<>();
set2.add("2A");
List<String> list3 = List.of(new String[]{"3A", "3B", "3C", "3D"});

// Put the iterables into a RoundRobinCollection
RoundRobinCollection<String> roundRobinCollection =
    new RoundRobinCollection<>();
roundRobinCollection.add(list1);
roundRobinCollection.add(set2);
roundRobinCollection.add(list3);

for (String s : roundRobinCollection) {
    System.out.println(s);
}
```

This prints these values in this order:

```
"1A"  "2A"  "3A"  "1B"  "3B"  "1C"  "3C"  "3D"
```

In English, describe how you would implement RoundRobinCollection. Provide enough detail that we would be able to implement your idea. Your answer should address at least the following issues (although not necessarily in this order):

- What data structure would you use to store the Iterables?
- Which classes would you create?
- When are the Iterators for the collection of Iterables instantiated, and what data structure would you use to store them?
- How would the RoundRobinCollection's iterator select the next item to return?
- When is the iterator done?

**Use this page for rough work, but write your answer on the next page.**  
**You might want to take some notes for an outline of what you want to say, for example.**

---

<sup>1</sup>By "round-robin order" we mean getting one item at a time from each collection, repeatedly. In the example code, the three collections contain the values (1A, 1B, 1C), (2A), and (3A, 3B, 3C, 3D). When iterating through, we first get 1A from the first collection, then 2A from the second collection, then 3A from the third collection, and then we get 1B from the first collection, then we get 3B from the third collection (since the second collection has no more elements), and so forth. Said another way, in the example we first see all of the A's, then B's, then C's, and then D's.

DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

Duration: **2 Hours**

**Write your RoundRobinCollection description here  
Please use the last couple pages of the exam if you need more space.**

**Question 5.** Agent Based Modeling Application [8 MARKS]

Consider a program used by epidemiologists to investigate the effectiveness of vaccination policies using agent-based modeling. In an agent-based model, there are a set of agents (people) whose actions are simulated.

In this program, class `Simulation` contains a `simulate` method responsible for the core logic of performing a simulation. Other parts of the code would run a series of simulations and report the results back to the user (similar to the main method shown below, but more substantial).

```
import java.util.ArrayList;
import java.util.List;

public class Simulation {

    final private List<Agent> agents = new ArrayList<>();

    public Simulation(List<Agent> agents) {
        this.agents.addAll(agents);
    }

    public void simulate() {
        new VaccinePolicy().apply(agents);

        for (Agent a : agents) {
            a.act();
            for (Agent b : agents) {
                if (!a.equals(b)) {
                    a.interact(b);
                }
            }
        }
    }

    // other methods and attributes related to a simulation would be here...

    public static final int NUM_SIMULATION_STEPS = 100;

    public static void main(String[] args) {
        List<Agent> agents = new ArrayList<>();
        for (int i = 0; i < 42; i++) {
            agents.add(new BasicAgent(i));
        }
        Simulation s = new Simulation(agents);
        for (int j = 0; j < NUM_SIMULATION_STEPS; j++) {
            s.simulate();
        }
        // other code that reports on the results of the simulation...
    }
}
```

DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

Duration: **2 Hours**

**Part (a)** [4 MARKS] Explain how you might modify the `Simulation` class so that you could apply the simulation to different vaccination policies. Be specific.

**Part (b)** [2 MARKS] Identify a code smell in the `Simulation` class and explain how you might fix it.

**Part (c)** [2 MARKS] Do you think that `Agent` should be an abstract class or an interface? Briefly justify your answer.

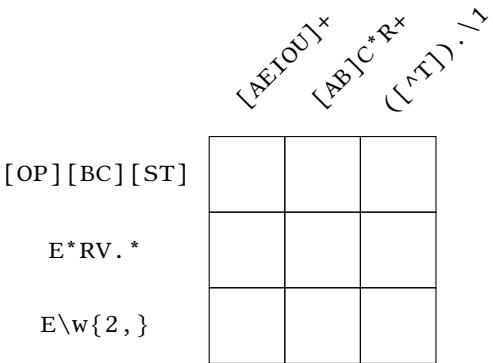
**Question 6.** Ethics and Regex [6 MARKS]

**Part (a) [1 MARK]** In the second ethics lecture, we discussed why we should make software accessible. Name two of those reasons. You do not need to explain them.

**Part (b) [1 MARK]** Name 2 of the Principles of Universal Design that you feel are most relevant to software design. You do not have to justify your choices.

**Part (c) [1 MARK]** Why might following Clean Architecture make it easier for software developers to implement accessibility features?

**Part (d) [2 MARKS]** Complete the following regex crossword.



**Part (e) [1 MARK]**

- i) Write a regex that will match phone numbers of the format `(###)-###-####`. You don't need to include any anchors. Your regex should be quite simple and you should make use of quantifiers where appropriate.
  
  - ii) Rewrite your regex so that we can conveniently find the area code (i.e. the first three digits).

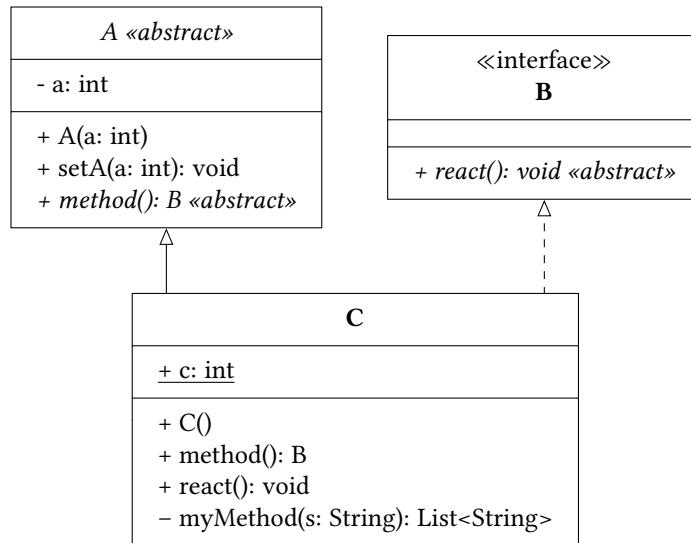
DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

Duration: **2 Hours**

**Question 7.** Code Writing [6 MARKS]

Consider the following UML.



Write Java code for (only) class C that is consistent with the UML class diagram. You do NOT need to include any javadoc or import statements. For any method that returns something or requires a concrete value, you can choose that value.

**Question 8.** Multiple Choice [8 MARKS]

**DO NOT put your answers on this page.** You must fill in your answers on the **last page** of this booklet.

**Select 1 (ONE) answer** for each question.

1. Which class is at the top of the inheritance hierarchy of all Error and Exception classes in Java?
  - (A) class RuntimeException
  - (B) class Throwable
  - (C) class Exception
  - (D) class Error
2. Suppose that you want to reuse part of an old program containing many classes that interact with each other. You want to do this in such a way that the old classes do not directly interact with the new program, except possibly for one class. Which of the following design patterns will encapsulate the old program in this way?
  - (A) Iterator
  - (B) Factory
  - (C) Builder
  - (D) Strategy
3. By replacing a switch statement with polymorphism, which SOLID principle do we more closely follow?
  - (A) Single Responsibility Principle
  - (B) Open Closed Principle
  - (C) Liskov Substitution Principle
  - (D) Interface Segregation Principle
  - (E) Dependency Inversion Principle
4. Which of the following will compile?
  - (i) `ArrayList<String> w = new List<>();`
  - (ii) `List<String> w = new List<>();`
  - (iii) `List<String> w = new ArrayList<>();`
  - (A) all of them
  - (B) none of them
  - (C) (ii) and (iii)
  - (D) only (iii)
5. What part of a floating point number is NOT stored as part of the 32 bits in a float?
  - (A) sign
  - (B) mantissa
  - (C) base
  - (D) exponent

Fill in your answers on the **last page** of this booklet.

- In the following 3 questions, identify the design pattern being described.
6. When the state of an object changes, other objects interested in that object's state are alerted about the change.
    - (A) Builder
    - (B) Façade
    - (C) Observer
  7. A class provides a simplified interface to a complex set of classes.
    - (A) Builder
    - (B) Façade
    - (C) Dependency Injection
  8. A group of related classes together define a family of algorithms. Which one is used is selected at runtime.
    - (A) Abstract Factory
    - (B) Factory Method
    - (C) Strategy

DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

**Duration: 2 Hours**

*Use the space on this “blank” page for scratch work, or for any solution that did not fit elsewhere.*

***Clearly label each such solution with the appropriate question and part number – and leave a note on the page of the original question, so that we can find your answer here.***

DECEMBER 2021 EXAMINATIONS

CSC 207 H1F

Duration: **2 Hours**

*Use the space on this “blank” page for scratch work, or for any solution that did not fit elsewhere.*

***Clearly label each such solution with the appropriate question and part number – and leave a note on the page of the original question, so that we can find your answer here.***