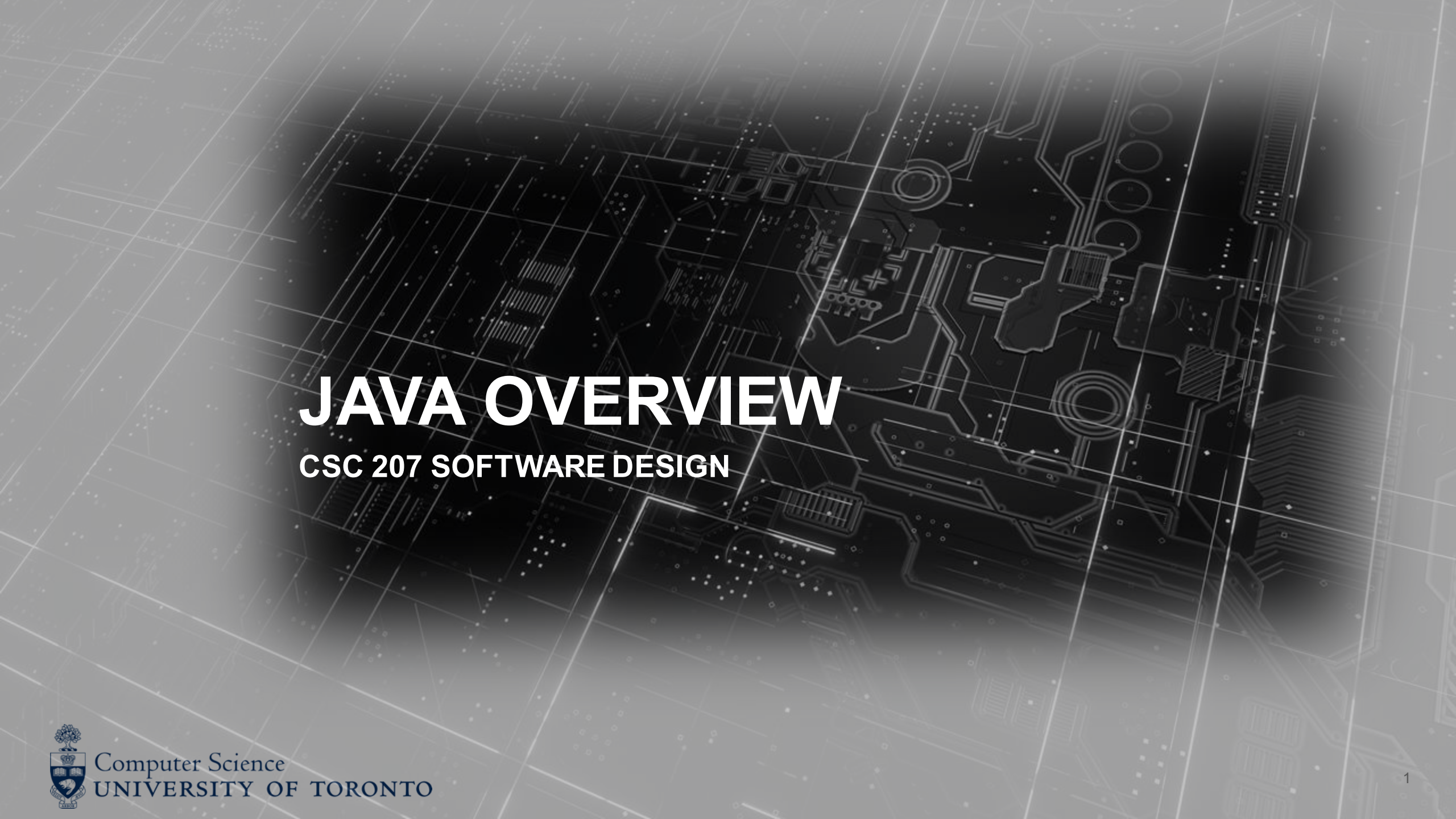
CSC207H1: Software Design



JAVA OVERVIEW

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

LEARNING OUTCOMES

- Understand how Java works at a high level
- Have a working knowledge of Java so you can contribute code to your group project throughout the term.

REFERENCE MATERIAL ON JAVA

- See the Github Classroom Coding exercises for the course and the course notes linked from Quercus.
- The official java tutorials:

http://docs.oracle.com/javase/tutorial/java/TOC.html

- This website does a nice job walking you through Java:

https://www.sololearn.com/Course/Java/

- Share any other resources you find useful on Piazza for others to benefit from.
- Ask on Piazza if you have questions.

RUNNING PROGRAMS

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

INTERPRETED VS. COMPILED

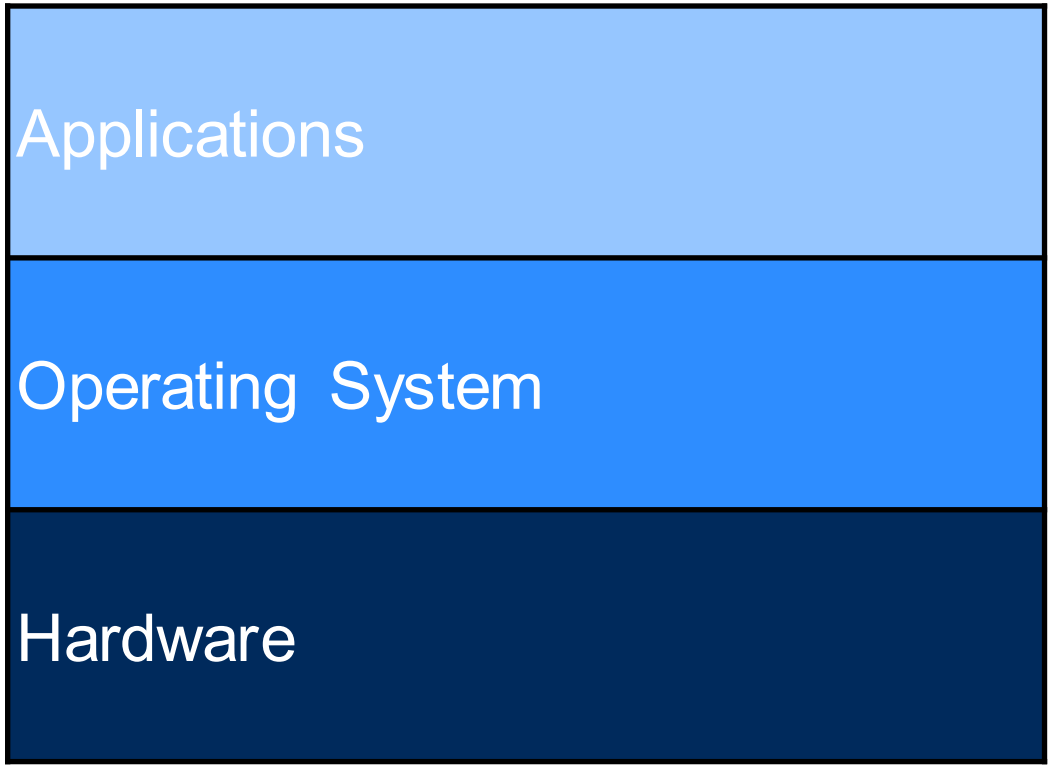
- **Interpreted**:
  - e.g., Python
  - Translate and execute one statement at a time
- **Compiled**:
  - e.g., C
  - Translate the entire program (once), then execute (any number of times)
  - **Hybrid**:
  - e.g., Java
  - Translate to something intermediate (in Java, bytecode)
  - Java Virtual Machine (JVM) runs this intermediate code

COMPILING JAVA

- You need to compile, then run (as described above).
- If using command line, you need to do this manually.
    - First, compile using “javac”:
    - `dianeh@laptop$ javac HelloWorld.java`
    - This produces file “HelloWord.class”:
    - `dianeh@laptop$ ls`
    - `HelloWorld.class  HelloWorld.java`
    - Now, run the program using “java”:
    - `dianeh@laptop$ java HelloWorld`
    - `Hello world!`
- Most modern IDEs (like IntelliJ) do this for you with the help of a build system (e.g. ant or gradle)
- But you should have some idea of what’s happening under the hood!
    - We encourage you to try compiling a simple HelloWorld application at least once.

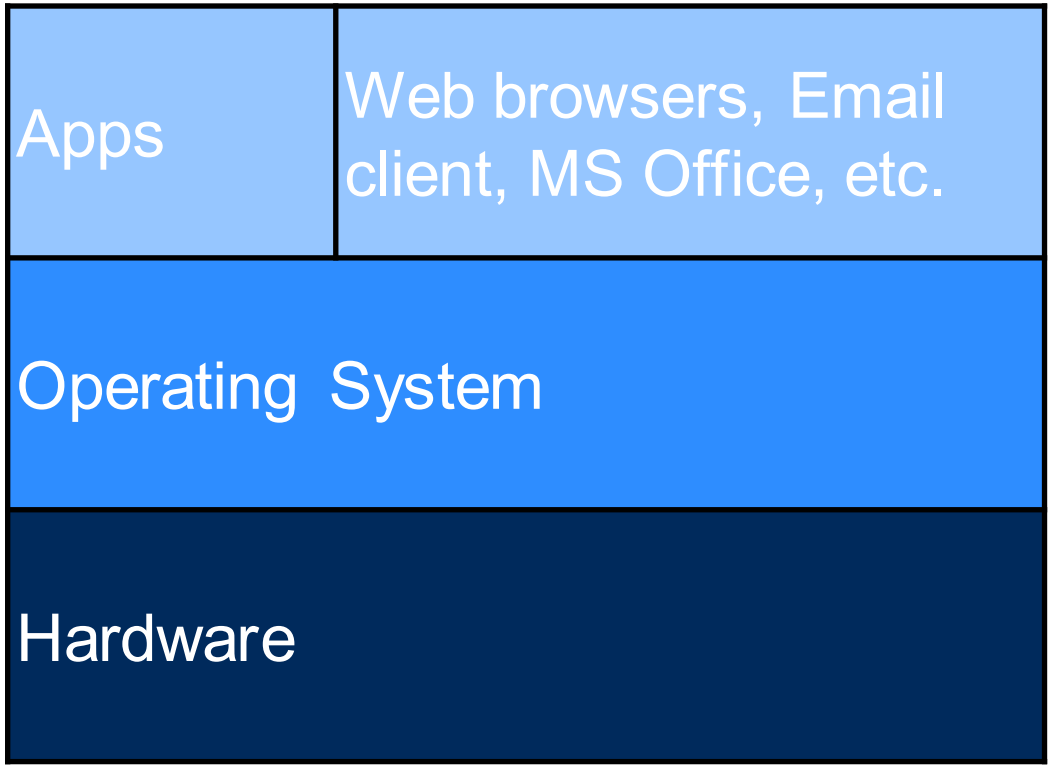
COMPUTER ARCHITECTURE

- 108 and 148: 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).



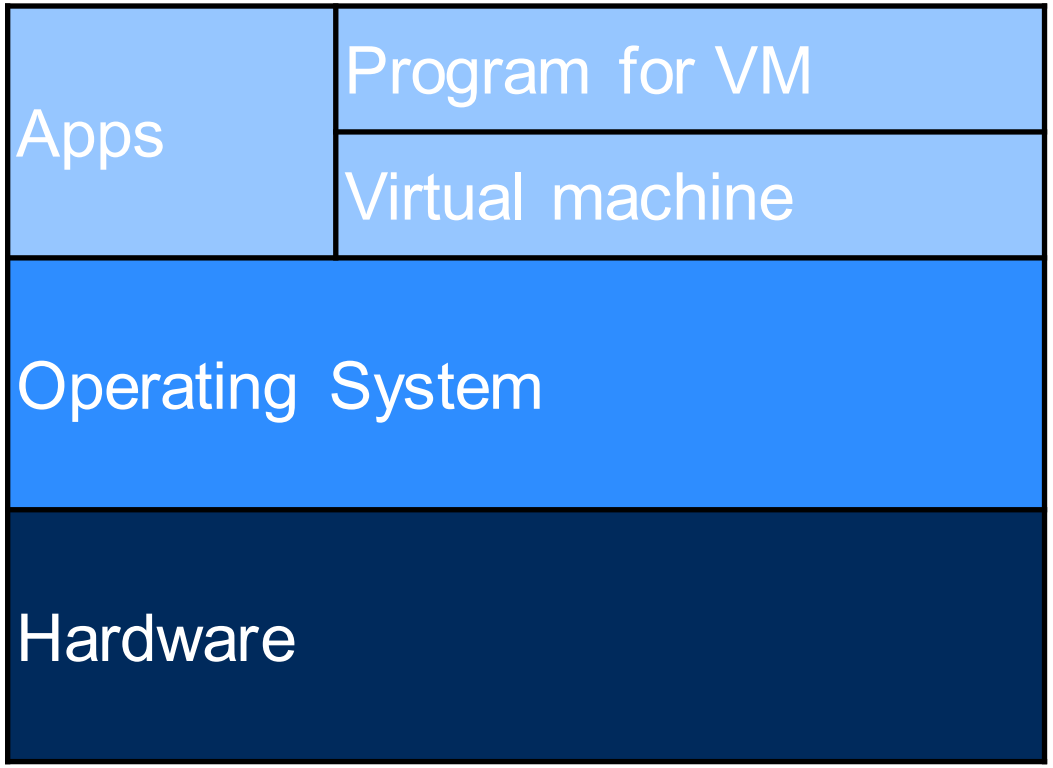
COMPILED APPLICATIONS

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



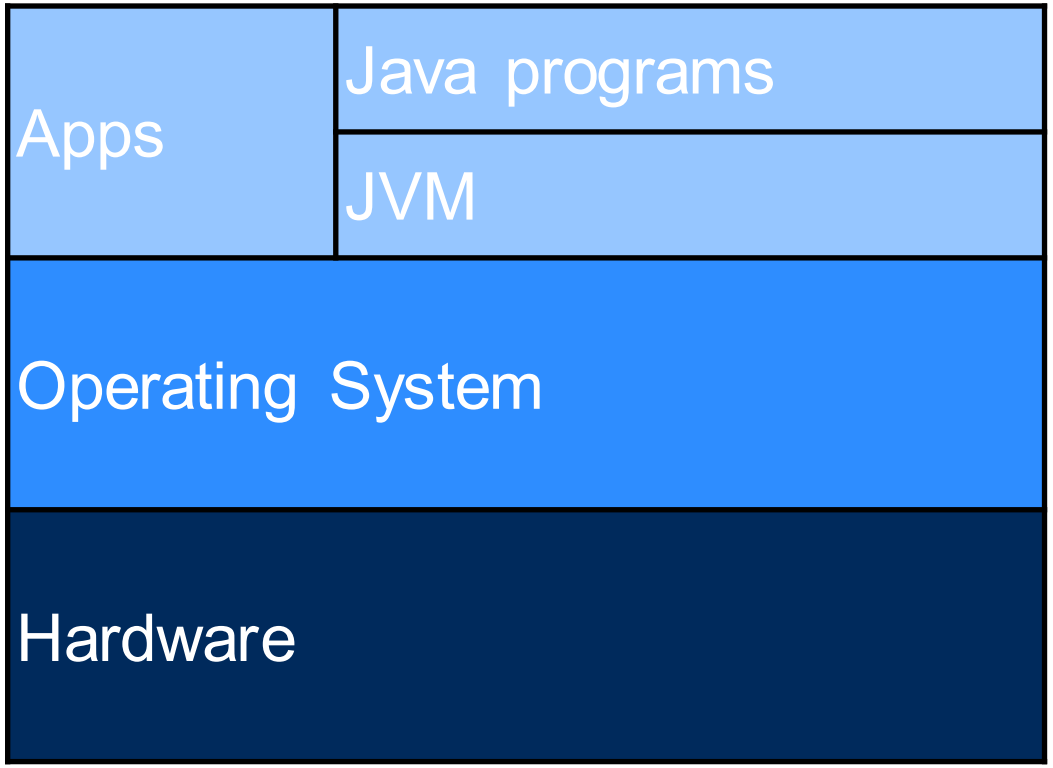
VIRTUAL MACHINE ARCHITECTURE

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

Duck Typing means that an object is defined by what it can do, not by what it is.
This means that we are less concerned with the class/type of an object and more concerned with what methods can be called on it and what operations can be performed on it. We don't care about it's type, we care about what it can do.

TYPES ARE CRITICAL

↑ type of dynamic typing

→ the type or class of an object is less important than the method it defines

- 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!

- 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.

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, as long as their signatures are different.
- If you define no constructors, the compiler supplies one with no parameters and no body.
- If you define any constructor for a class, the compiler will no longer supply the default constructor.

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.

DEFINING METHODS

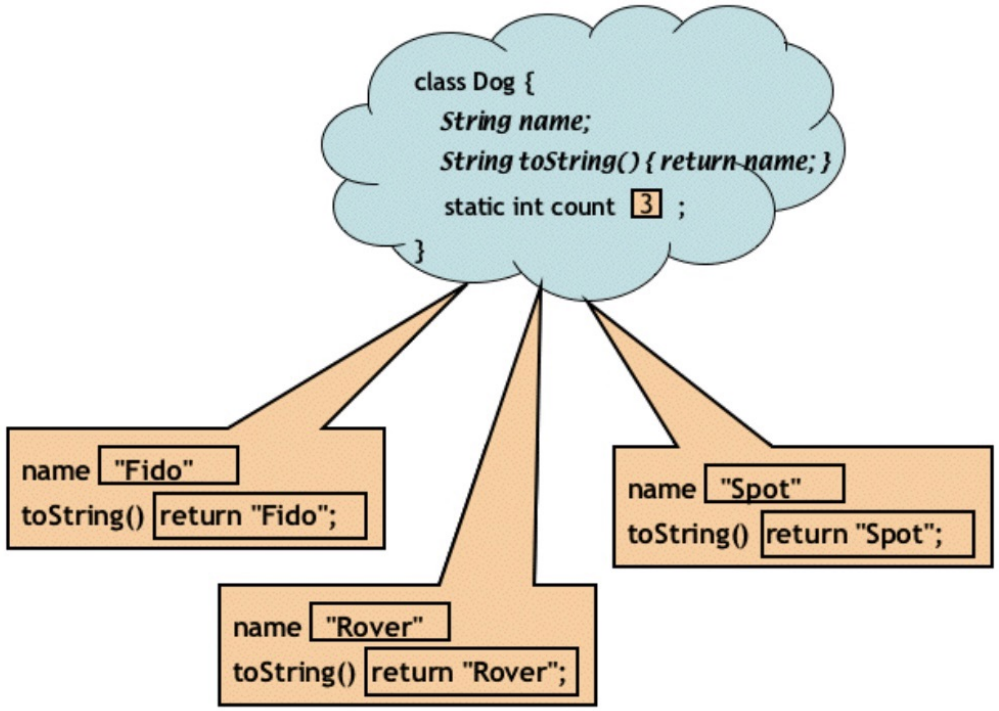
- A method must have a return type declared. Use `void` if nothing is returned.
- The form of a return statement:
  `return` *expression;*

  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.

A class is a ***description*** of objects. Almost all variables declared within a class belong to the objects of that class, not to the class itself; these are called **instance variables**. In the picture below, `name` is an instance variable, and each of the three `Dog` objects has its own copy ("instance") of this variable. The class itself does *not* "have" a `name` variable; it just *describes* that variable, so that when you create `Dog` objects, Java knows what variables to put in them.



Static Variables in Java

Similarly, almost all methods declared in a class belong to the objects of the class; these can be called **instance methods**, though this term is rarely used. In the above picture, `toString` is an instance method that returns the `name` of the object to which it belongs.

If you want the class itself to have a variable, declare it as `static`. A **static variable** (or **class variable**) is one that belongs *to the class itself,* rather than to the objects of the class. In the above example, `count` is a variable that might be used to keep track of how many `Dog` objects have been created.

Because a static variable belongs to the class itself, there is only one of it, not one for each object. If you change the value of this variable, it is changed in the class. Individual `Dog` objects can use static variables just like any other (because every object knows what class it belongs to), but individual dogs don't have a "count." What is Fido's count? That doesn't make sense.

Likewise, a **static method** (no example shown) *belongs to the class itself,* not to individual objects of that class. This means:

- You don't need an object to use the method; you can talk to the class itself. For instance, if you had a method `static int getCount()` to return the value of `count`, you could use it by saying `int c = Dog.getCount();`
- Static methods *cannot use instance variables or instance methods.* If you look at the picture, you can see why; the instance variables and methods are not in the class. This makes static methods more limited than normal methods.

When should you use `static`?

The general rule is: Avoid **`static`**. However,

- The **`main`** method *must* be static. When the program first starts, there are no instances (objects) yet to talk to, so you cannot start from an instance method. However, the classes themselves *have* been created, so static methods can be used.
- When you want to have a variable that describes something about the class itself, not the individual objects of that class, make it **`static`**.
- When you want to have a variable that *always* has the same value for *every* object of the class, forever and ever, make it **`static`**.
- If you have a method that does not use any instance variables or instance methods, you should *probably* make it **`static`**.

A static method cannot use a non-static variable or method (those belong to the actual objects); if you try to do this, you will get the following error message:

`non-static variable` [or **`method`**] ***`Name`*** **`cannot be referenced from a static context`**

The correct way to fix this error is almost always to make the calling method non-static, rather than to make the called method static. You can avoid most such problems by starting your program as follows: Have your **`main`** method create an object of whatever class it is in, and call an instance method of that object to do the actual work. Like this:

```
public class MyClass {

    public static void main(String[] args) {
        MyClass slave = new MyClass();
        slave.doAllTheWork();
    }

    void doAllTheWork() {
        // Start the real work here...
    }
}
```

Final parameters

If you ever see the `final` keyword with a parameter variable, it means that the value of this variable cannot be changed anywhere in the function.

Example:

```
class finalParameter {
    public static void example( final int parameter ) {

      parameter = 4; //attempting to reassign a value to a parameter throws an

    }
}
```

Run

FINAL VARIABLES

If a variable is declared with the final keyword, its value cannot be changed once initialized. Note that the variable does not necessarily have to be initialized at the time of declaration. If it's declared but not yet initialized, it's called a blank final variable.

Final methods

A method, declared with the `final` keyword, cannot be *overridden* or *hidden* by subclasses.

Example:

```
' declaring a final method
lass Base{

   public final void finalMethod(){
       System.out.print("Base");
   }



lass Derived extends Base{
   public final void finalMethod() { //Overriding the final method throws an err
       System.out.print("Derived");
   }

```

Final classes

A class declared as a `final` class, cannot be subclassed

Example:

```
// declaring a final class
final class FinalClass {
  //...
}

class Subclass extends FinalClass{ //attempting to subclass a final class throw
  //...
}
```

Run

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.

INSTANCE VARIABLES AND ACCESSIBILITY

- If an instance variable is private, how can client code use it?
- Why not make everything public — so much easier! NO!!

Instance variables are made private to force the users of those class to use methods to access them. In most cases there are plain getters and setters but other methods might be used as well.

Using methods would allow you, for instance, to restrict access to read only, i.e. a field might be read but not written, if there's no setter. That would not be possible if the field was public.

Additionally, you might add some checks or conversions for the field access, which would not be possible with plain access to a public field. If a field was public and you'd later like to force all access through some method that performs additional checks etc. You'd have to change all usages of that field. If you make it private, you'd just have to change the access methods later on.

If `phone` was private:

Consider this case:

```
class Address {
  private String phone;

  public void setPhone(String phone) {
    this.phone = phone;
  }
}

//access:
Address a = new Address();
a.setPhone("001-555-12345");
```

If we started with the class like this and later it would be required to perform checks on the phoneNumber (e.g. some minimum length, digits only etc.) you'd just have to change the setter:

```
class Address {
  private String phone;

  public void setPhone(String phone) {
    if( !isValid( phone) ) { //the checks are performed in the isVali
     throw new IllegalArgumentException("please set a valid phone numb
    }

    this.phone = phone;
  }
}

//access:
Address a = new Address();
a.setPhone("001-555-12345"); //access is the same
```

If `phone` was public:

Someone could set `phone` like this and you could not do anything about it:

```
Address a = new Address();
a.phone="001-555-12345";
```

If you now want to force the validation checks to be performed you'd have to make it private and whoever wrote the above lines would have to change the second line to this:

```
a.setPhone("001-555-12345");
```

Thus you couldn't just add the checks without breaking other code (it wouldn't compile anymore).

Additionally, if you access all fields/properties of a class through methods you keep access consistent and the user would not have to worry about whether the property is stored (i.e. is a instance field) or calculated (there are just methods and no instance fields).

encapsulation is used to hide the values or state of a structured data object inside a class - preventing unauthorised parties' direct access to them

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*.

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.

- [A *final* variable cannot change value; it is a constant.]

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

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.

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 practise:
    - 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.
- 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`.

- So 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 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!

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 lis = new ArrayList();   // lis = []`
  - `Map map = new HashMap();      // map = {}`
  - `Set set = new HashSet();      // set = set()`

INTRO TO EXCEPTIONS

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 you aren’t going to deal with exceptions (or may throw your own):
  - *accessMod 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.
- `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.

EXAMPLES

- Dealing with exceptions:
- You will call many methods that can generate exceptions if something goes wrong. E.g., I/O methods.
- Example ...
- You can try-catch, or “pass it on”

- Throwing your own exceptions:
- Example ...

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

(BAD) 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.
  - Unlike Python, 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`.

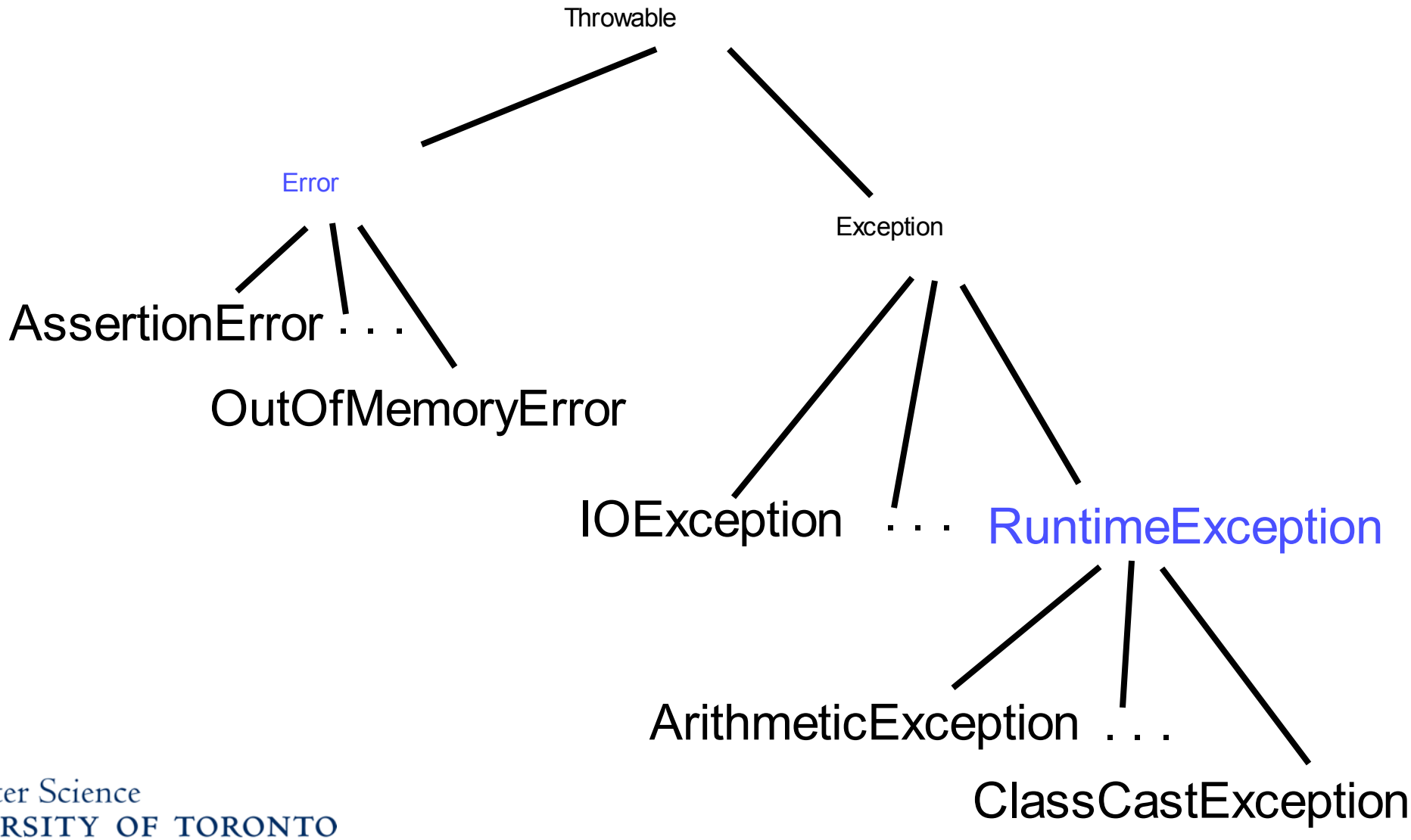
```
try { ... }
catch (ExSubA e) {
  // We do this if an ExSubA is thrown.
}
catch (ExSup e) {
  // We do this if any ExSup that's not an ExSubA is thrown.
}
catch (ExSubB e) {
  // We never do this, even if an ExSubB is thrown.
}
finally {
  // We always do this, even if no exception is thrown.
}
```

EXCEPTIONS – PART 2

RECAP

- If you call code that may throw an exception, you have two choices.
- When you declare that a method “throws” something, you are reserving the right to do so, not guaranteeing that you will.
- 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”.
- Benefits of using exceptions?
- There is more to know . . .

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 it’s “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 the static keyword.
  (It can only be used with classes that are nested.)
    - Cannot access any other members of the enclosing class.
- Inner classes do not use the static keyword.
    - 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!"); ...
}
```

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

EXTEND EXCEPTION OR RUNTIMEEXCEPTION?

- 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`:
  - "The class Exception and its subclasses are a form of Throwable that indicates 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."
  - `RuntimeException` examples:
    - `ArithmeticException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`
- non-`RuntimeException` (checked):
  - non-`RuntimeException` examples:
    - `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**."

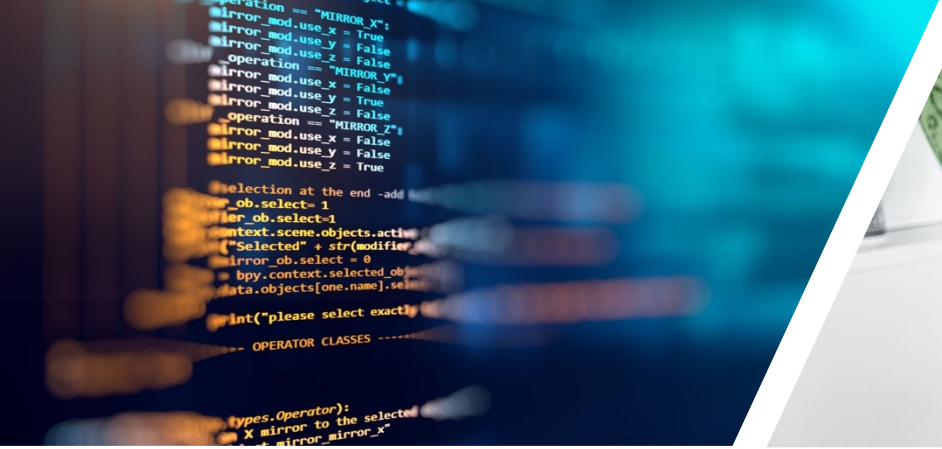
https://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html

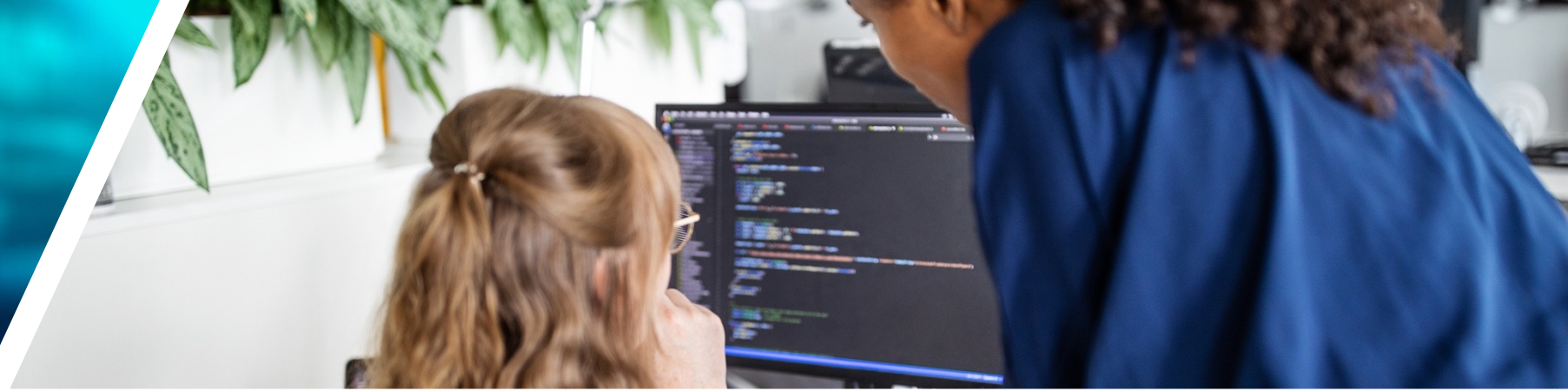
EXAMPLE

- 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 before they call `o.getItem(x)`.
  - So sending an invalid index should not cause a checked exception to be thrown.
- "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.





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?

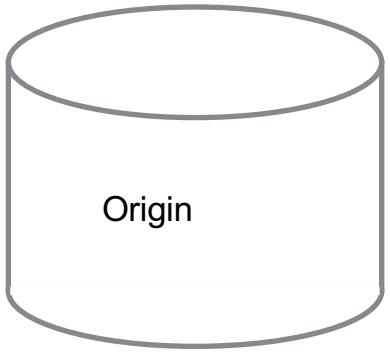
- **Backup and restore** - because accidents happen
- **Synchronization** - multiple people can make changes
- **Short term undo** - that last change made things worse?
- **Long term undo** - find out when a bug was introduced
- **Track changes** - all changes related to a bug fix
- **Sandboxing** -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

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

GIT

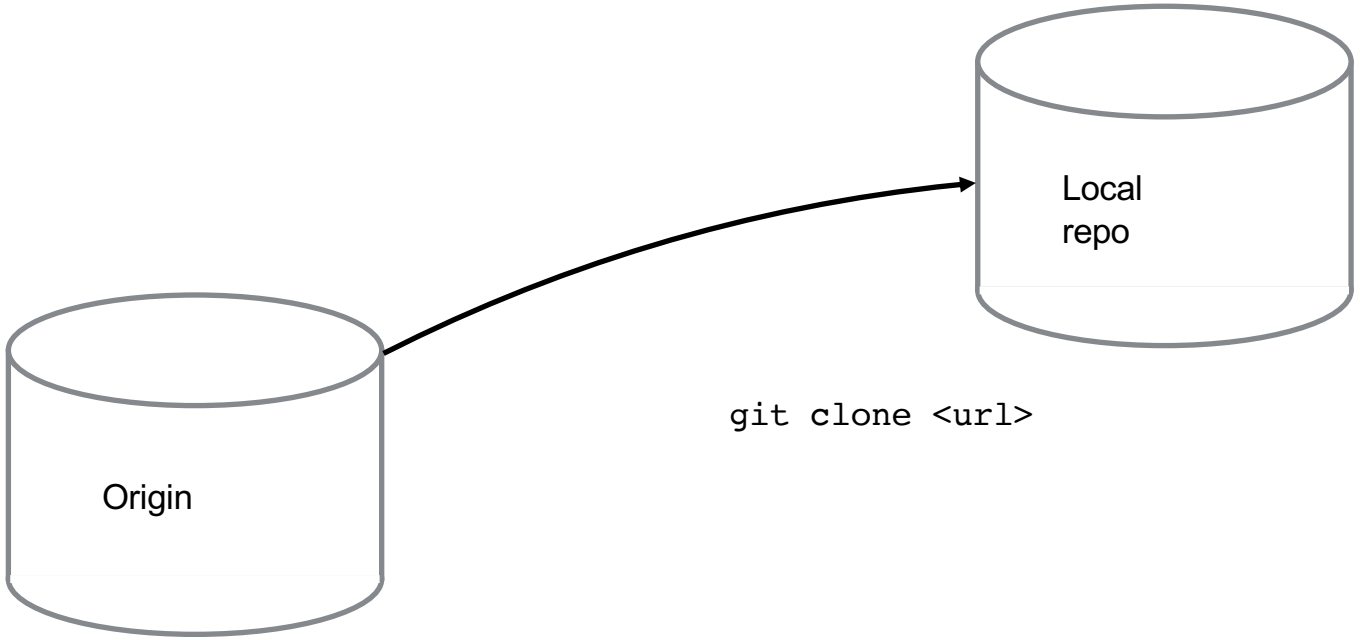
- For CSC207, we will create repositories for you
- These repositories live on GitHub Classroom
- You will **clone** your remote repository, work on files locally, **add** and **commit** changes to your local repository, and **push** changes to the remote repository.
- We'll only teach the basics in this course, so 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 and the system's design.

REMOTE REPOSITORY



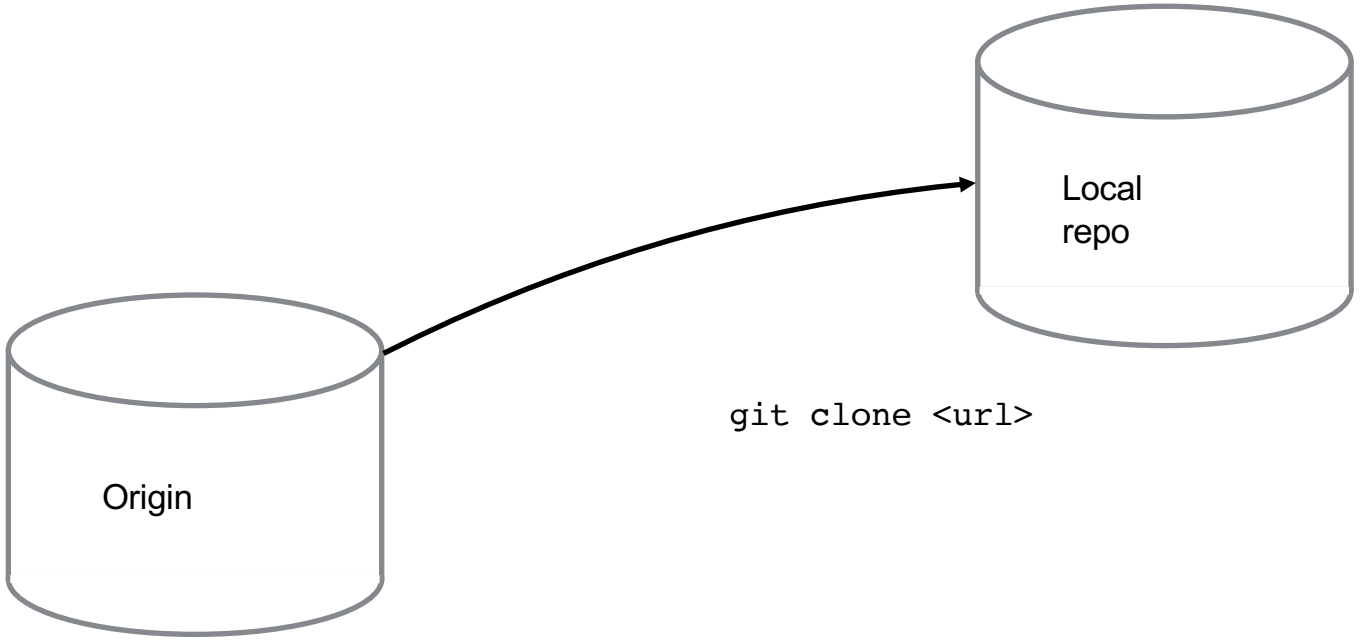
- This is the repo that lives on another server.
- By convention we call it the *origin*
- (In Github terminology, you may have an “upstream” repo and a forked copy of the repo called the “origin”, but we are working with a simpler model)

CLONE TO GET LOCAL REPO



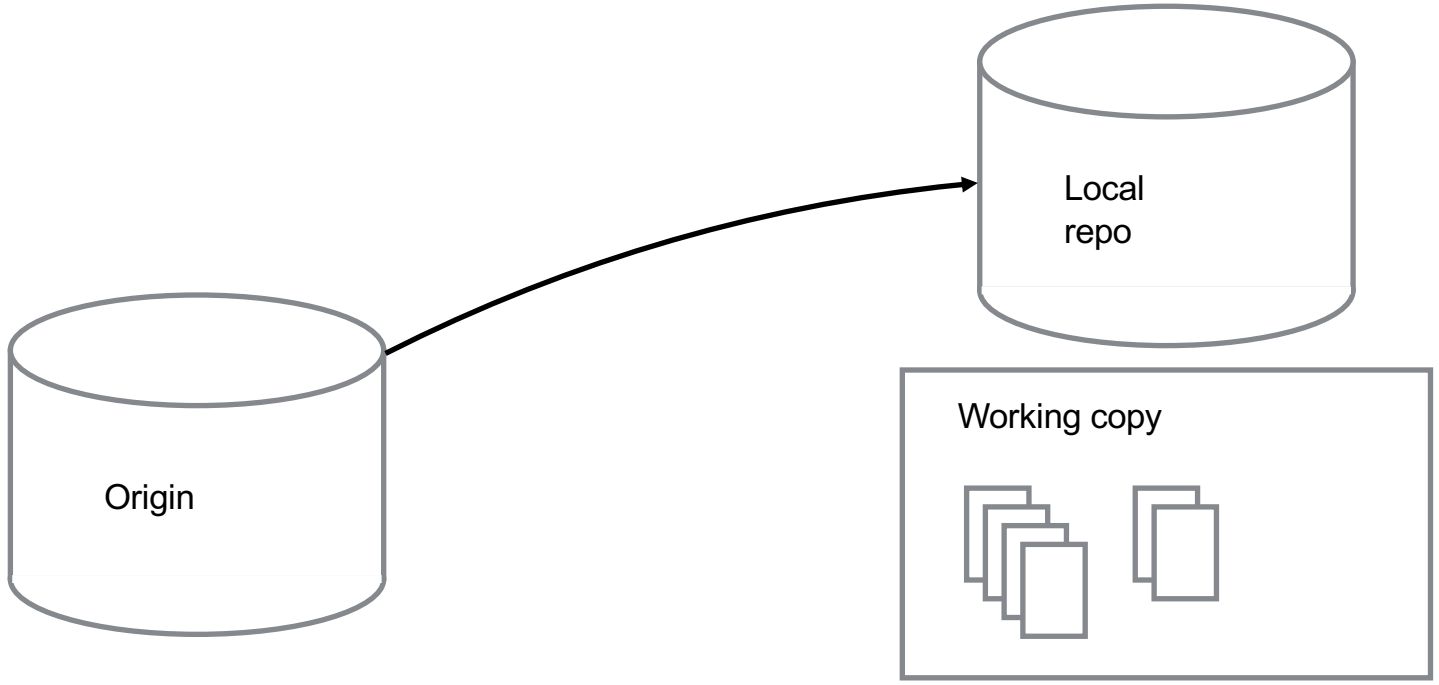
- The “clone” command makes a copy of the remote repository on your local machine.
- Now there are two repositories. (Git is a distributed version control system)

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



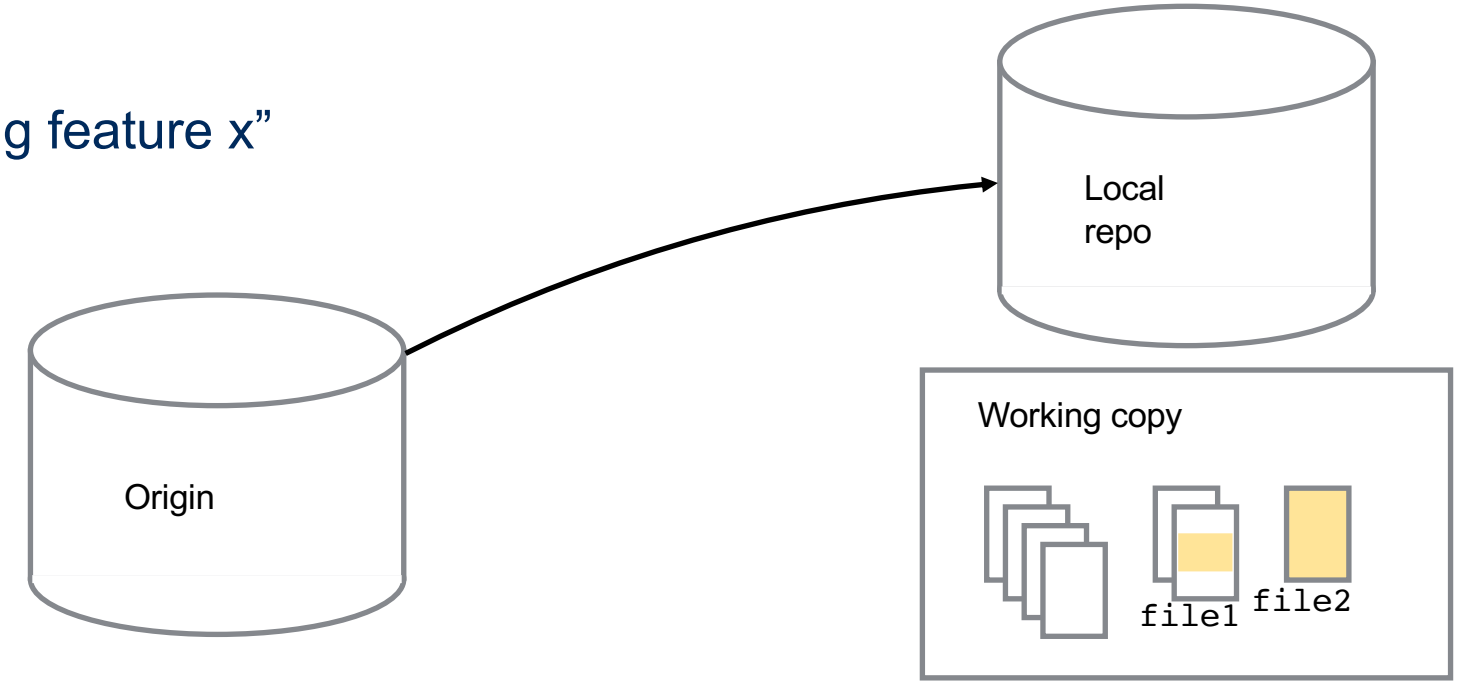
- The “clone” command makes a copy of the remote repository on your local machine.
- Now there are two repositories. (Git is a *distributed* version control system)

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.

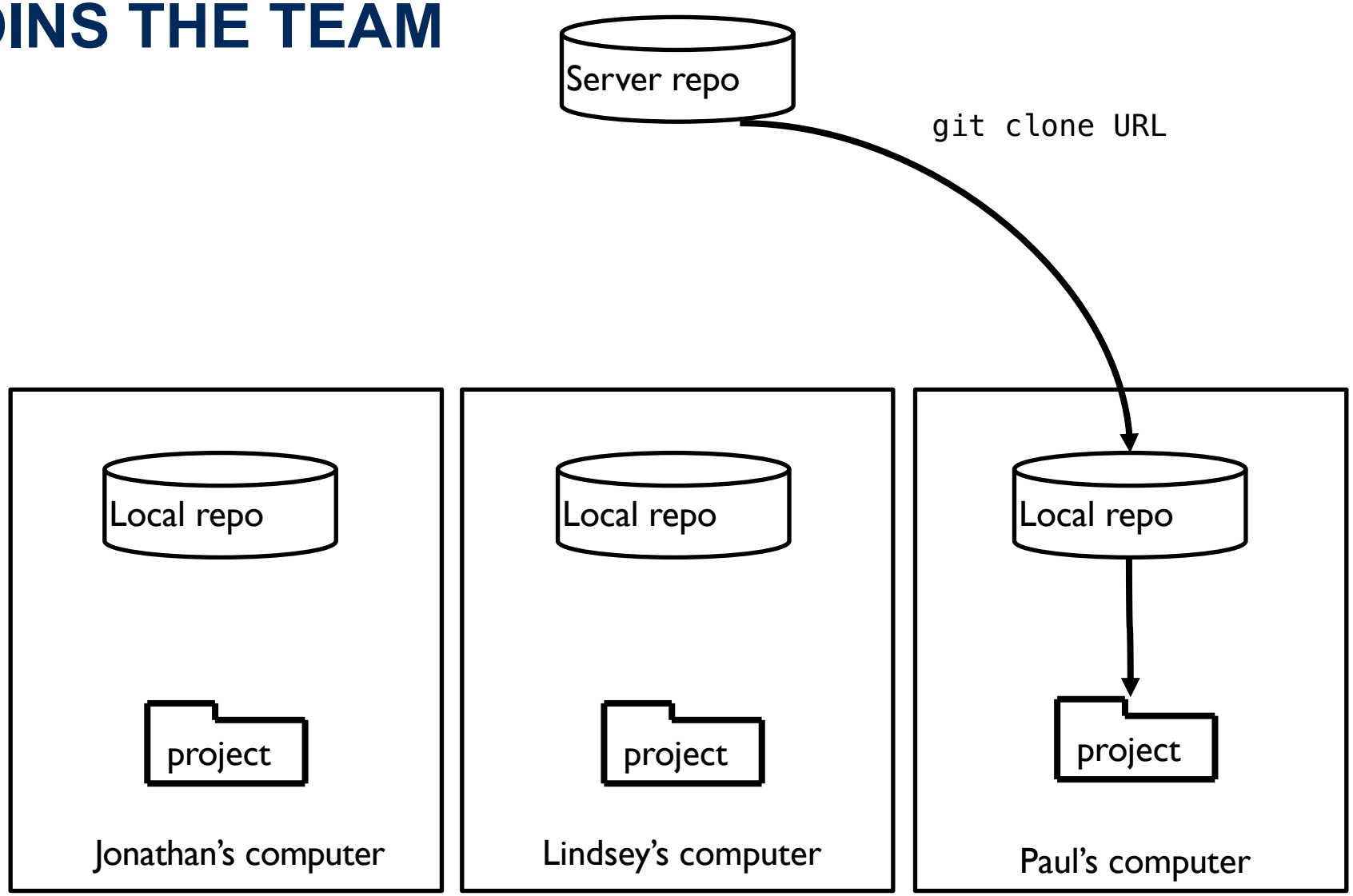
GIT STATUS

- 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)

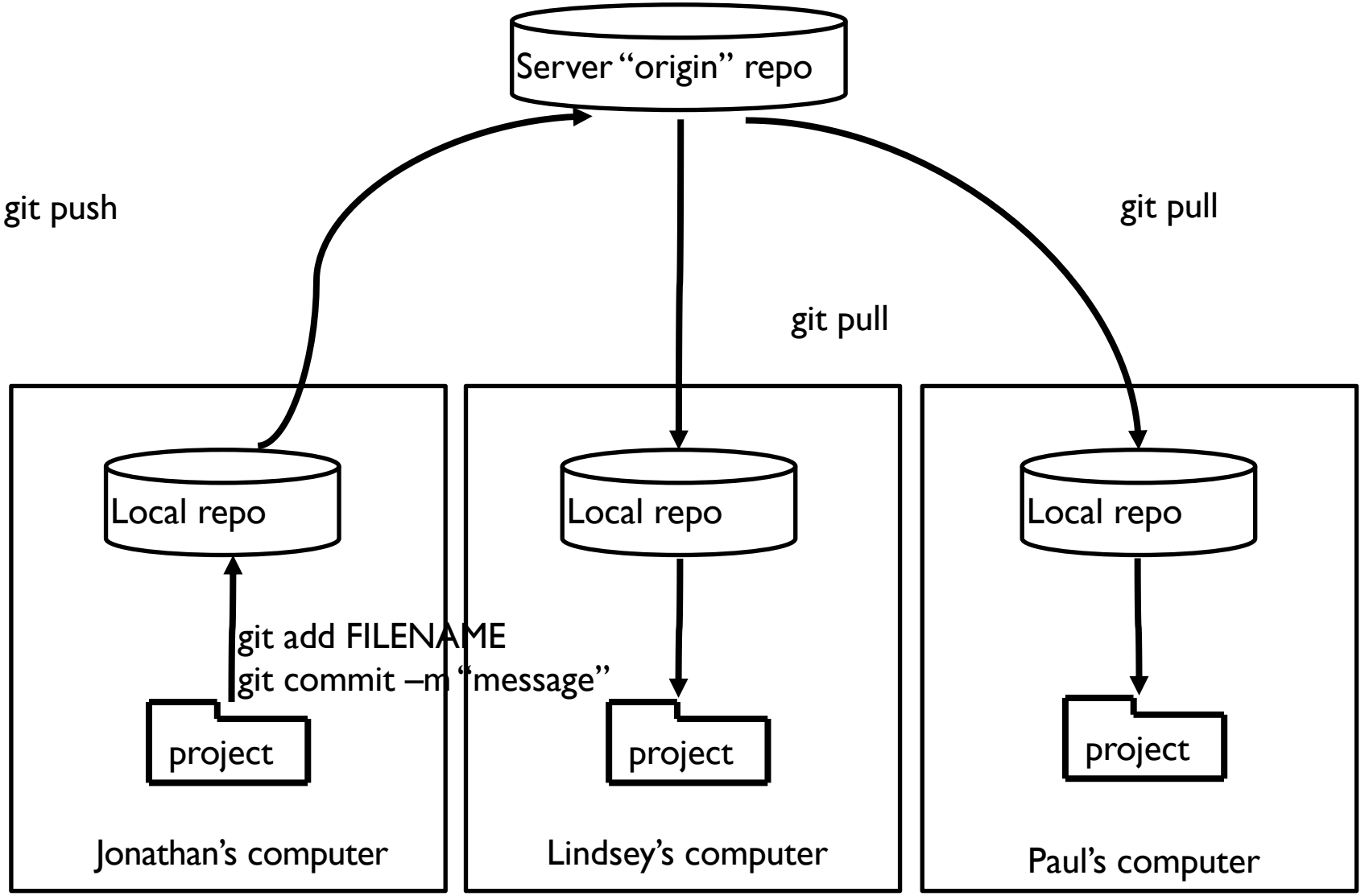
BASIC WORKFLOW

- 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"` ◄····· Save changes to local repo
  - `git push` ◄····· Push changes to the remote repo
- In IntelliJ, you can either type these commands in the Terminal or use the graphical user interface it provides.

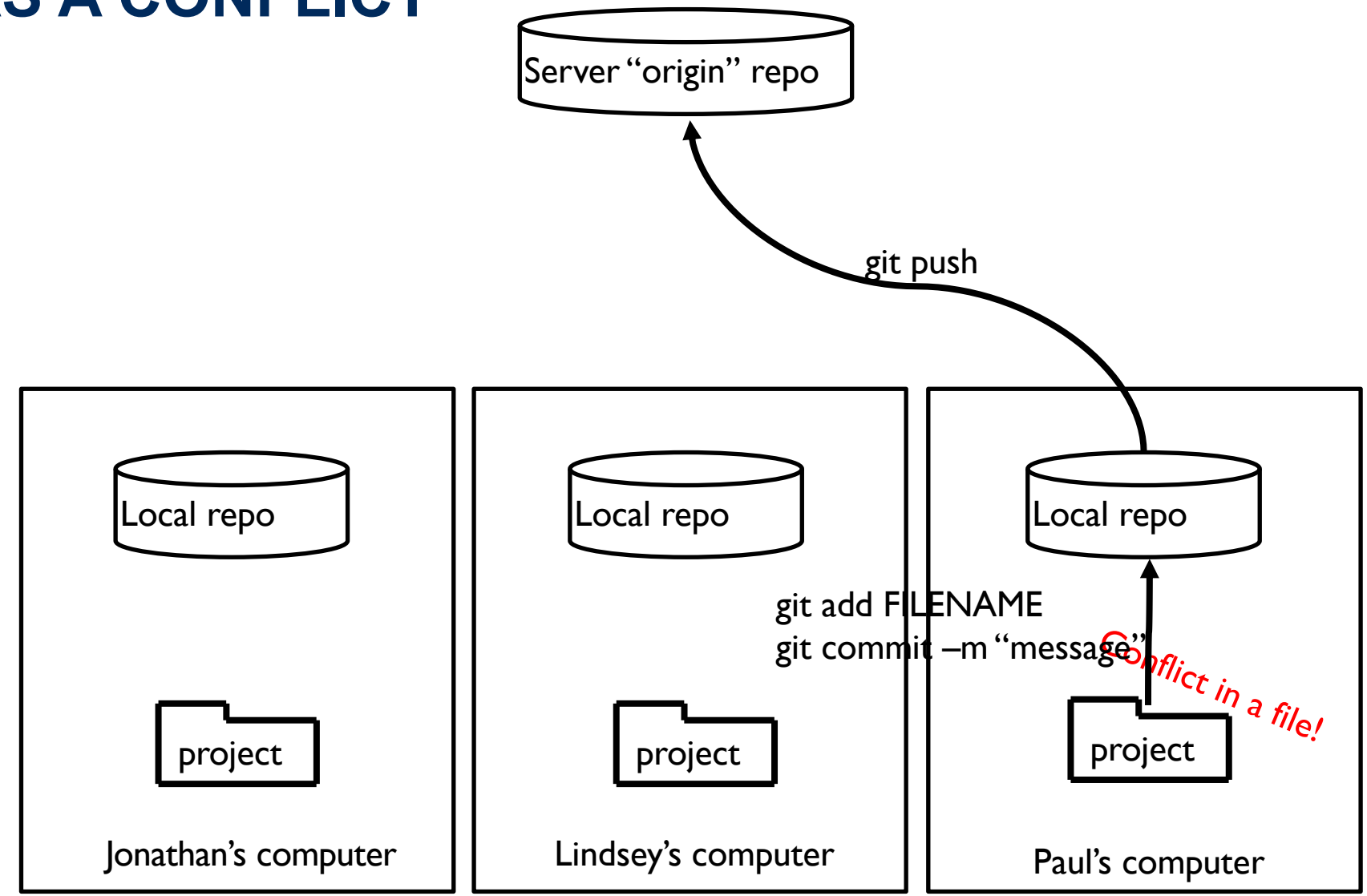
DISTRIBUTED VERSION CONTROL SYSTEMS: PAUL JOINS THE TEAM



DISTRIBUTED VERSION CONTROL SYSTEMS: JUSTIN 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!
- 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

- Complete the weekly questions about version control and git. Ask if you have questions!

CRC CARDS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

LEARNING OUTCOMES

- Know what CRC cards are and how they can be useful.
- Be able to construct CRC cards for a simple specification.
- Have a motivation for why we need design principles.

CRC CARDS

- Part of the Object-Oriented development paradigm.
- Highly interactive and human-intensive.
- Final 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.
- Input from every team member.

WHAT IS A CRC CARD?

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

Class

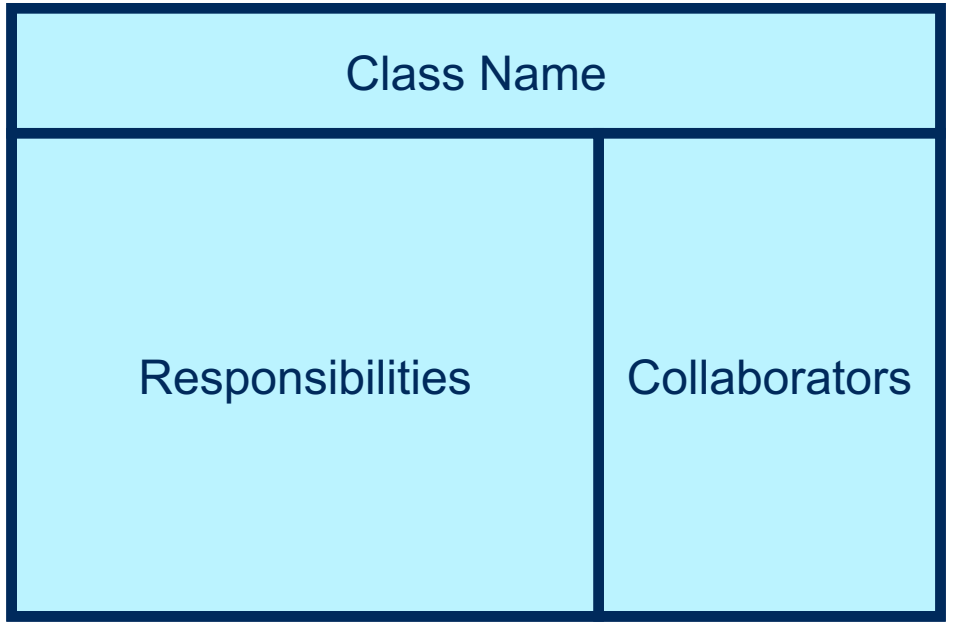
- An object-oriented class name
- Include information about super- and sub-classes

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 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.

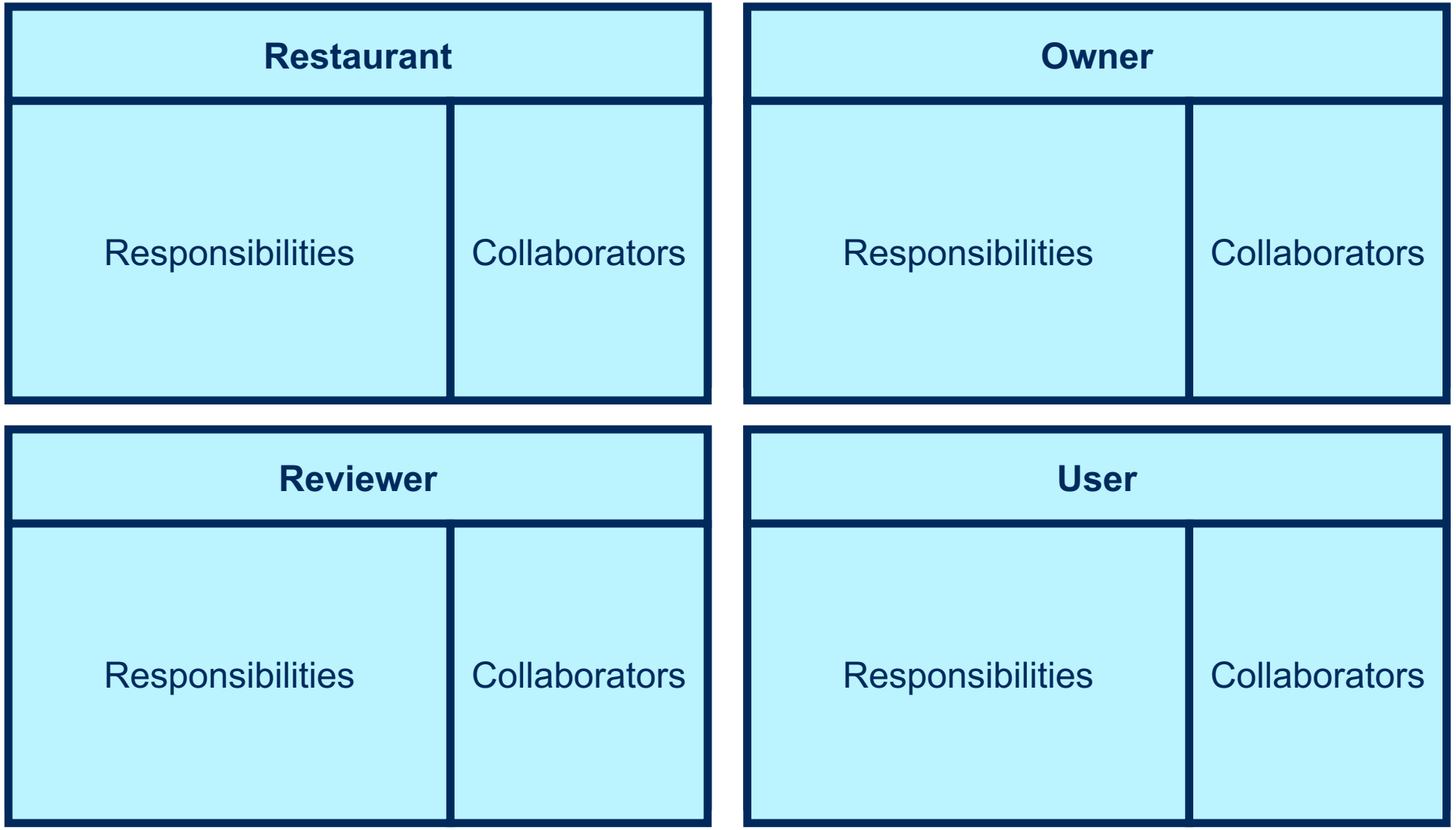
When reviewers leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An owner of a restaurant can respond to a review with a comment. All users of the system log in with their username. Users can choose to be contacted by email ...

CHOOSE POTENTIAL CLASSES

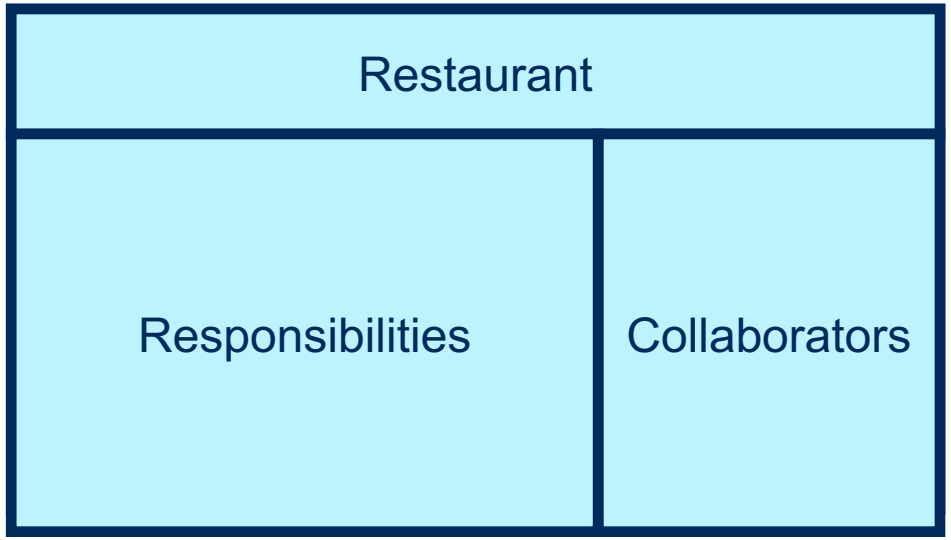
Let's try to narrow down to the most important nouns and start with those as potential classes to make cards for to get started.

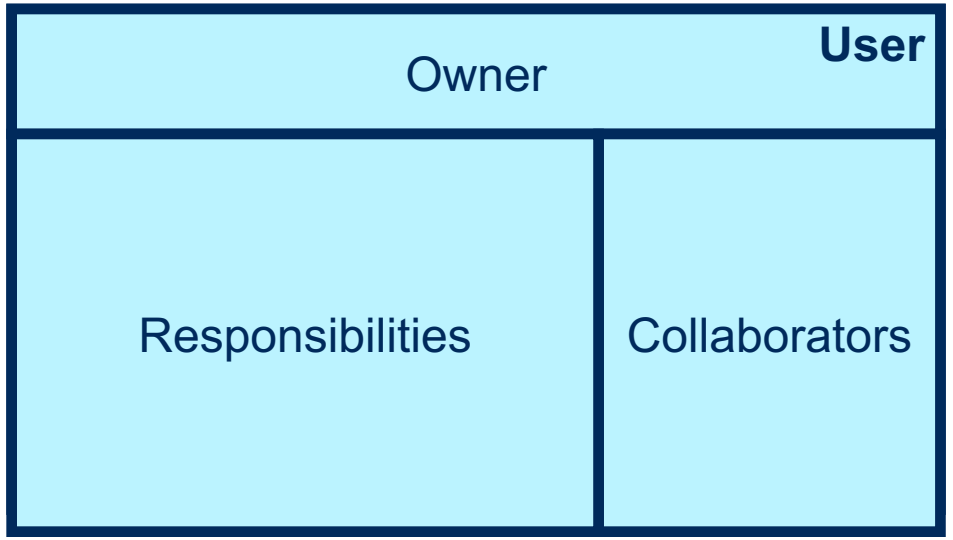
*Each **restaurant** corresponds to a certain price range, neighbourhood, and cuisines it serves. Restaurants that serve alcohol must have a license, which they need to renew every year. The system should also report how long, on average, customers wait for take out in restaurants that offer take-out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

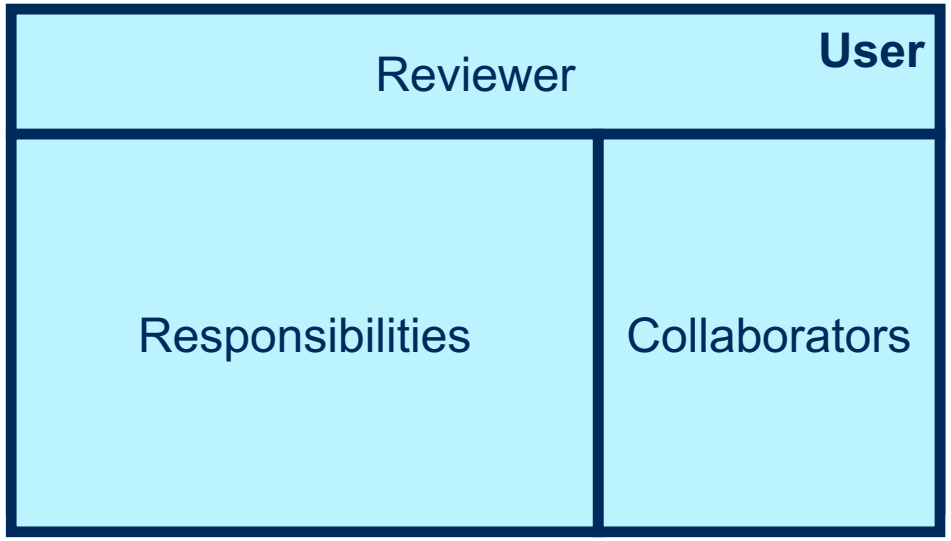
*When **reviewers** leave a review for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An **owner** of a restaurant can respond to a review with a comment. All **users** of the system log in with their username. Users can choose to be contacted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

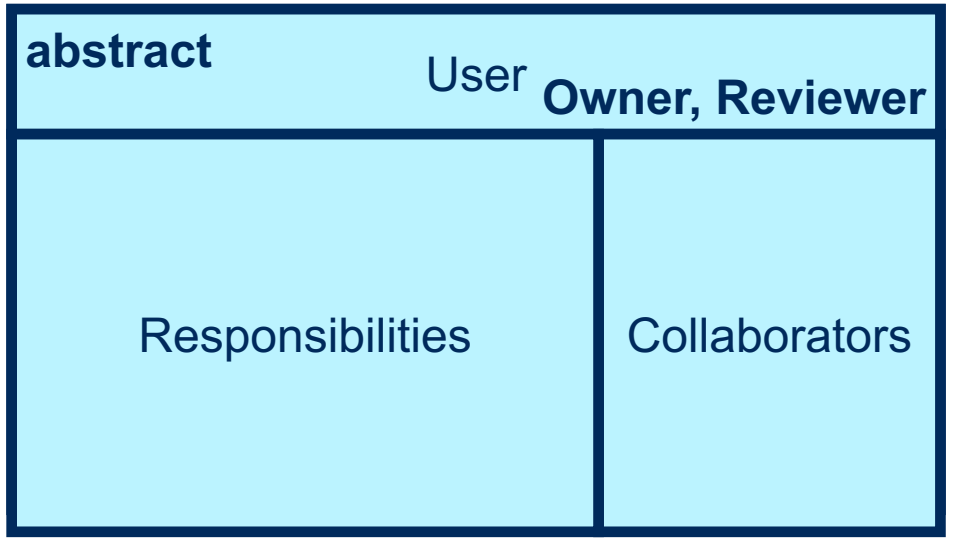

Restaurant
Responsibilities
Collaborators
Owner
Responsibilities
Collaborators
Reviewer
Responsibilities
Collaborators
User
Responsibilities
Collaborators

- 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.









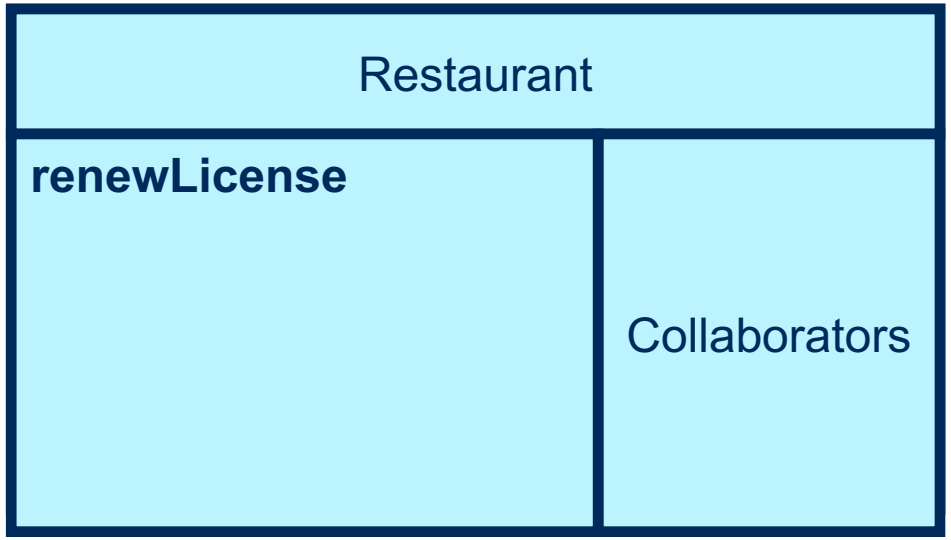
IDENTIFY VERB PHRASES DESCRIBING RESPONSIBILITIES

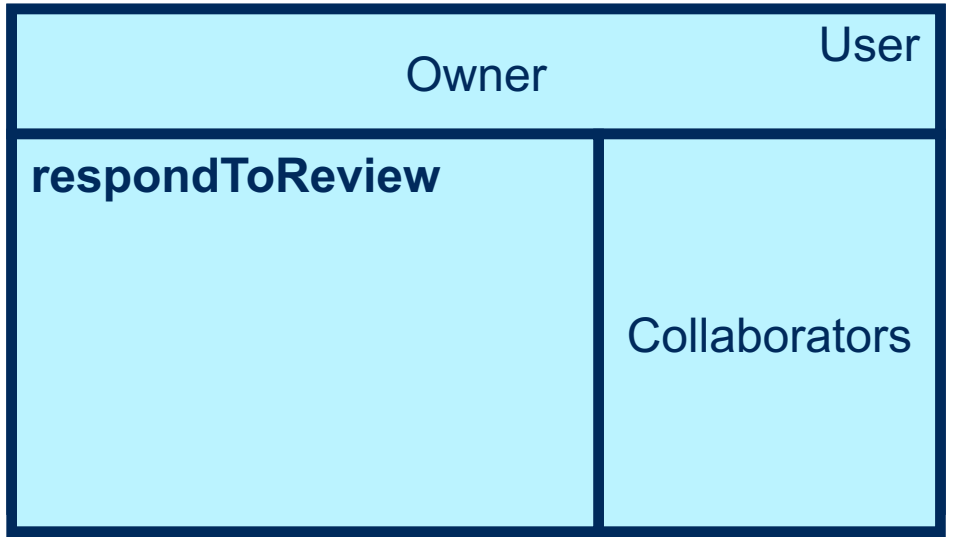
Also keep in mind what class is responsible for doing each of these verb phrases.

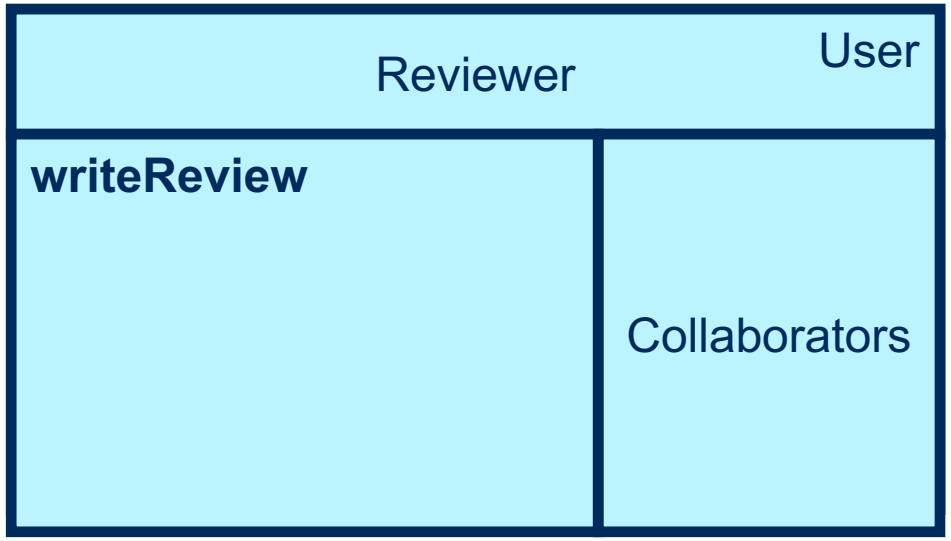
*Each **restaurant** corresponds to a certain price range, neighbourhood, and cuisines it serves. Restaurants that serve alcohol must have a license, which they need to **renew** every year. The system should also **report how long, on average, customers wait for take out** in restaurants that offer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

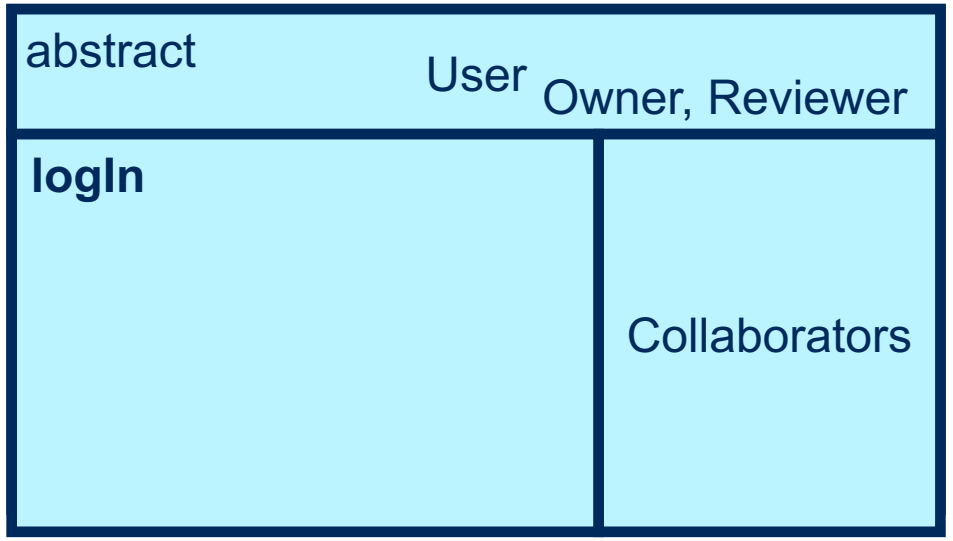
*When **reviewers leave a review** for a restaurant, they must specify a recommendation (Thumbs Up or Thumbs Down) and can also leave a comment. An **owner** of a restaurant can **respond to a review** with a comment. All **users** of the system **log in** with their username. Users can choose to be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

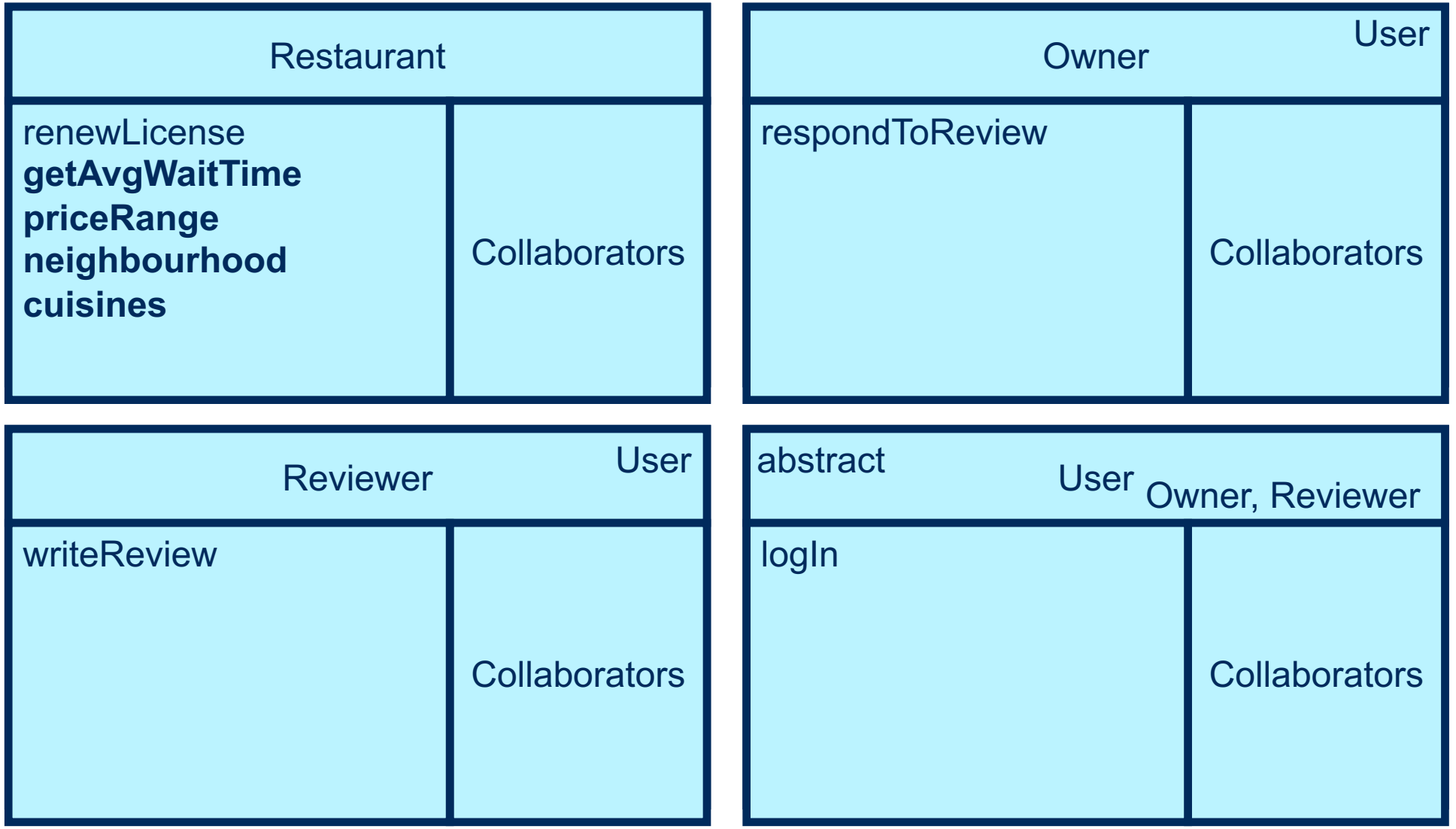






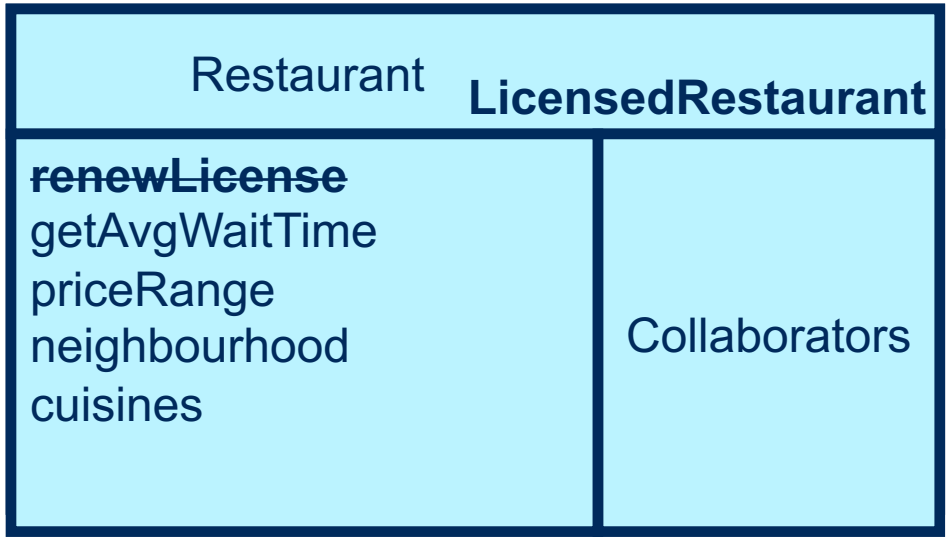


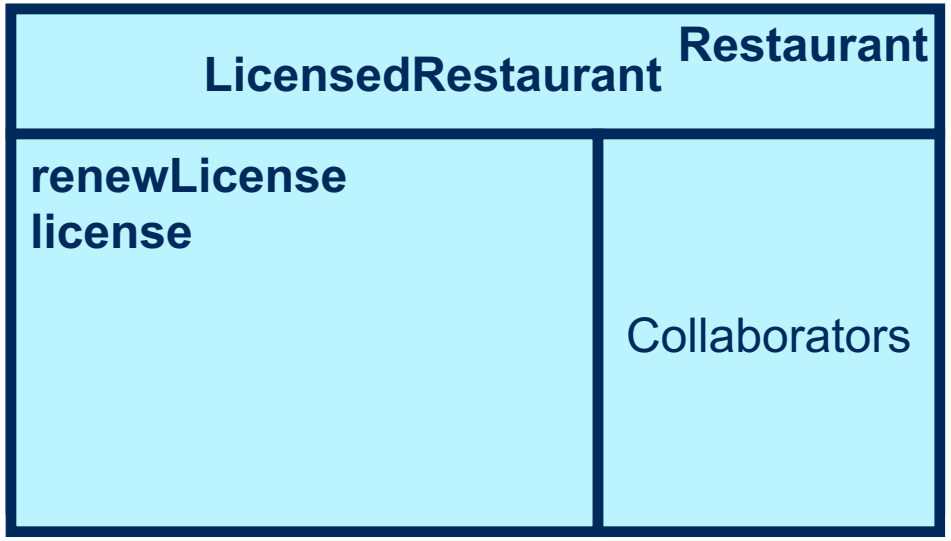
- We can also add some “what they store” responsibilities for Restaurant.



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.





TAKEOUT

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

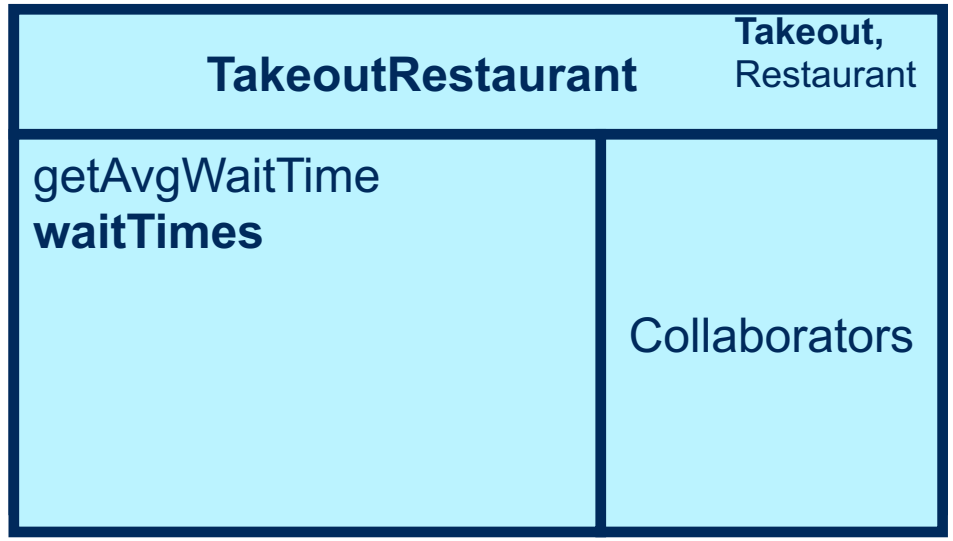
TakeoutRestaurant	**Restaurant**
getAvgWaitTime **waitTimes**	Collaborators

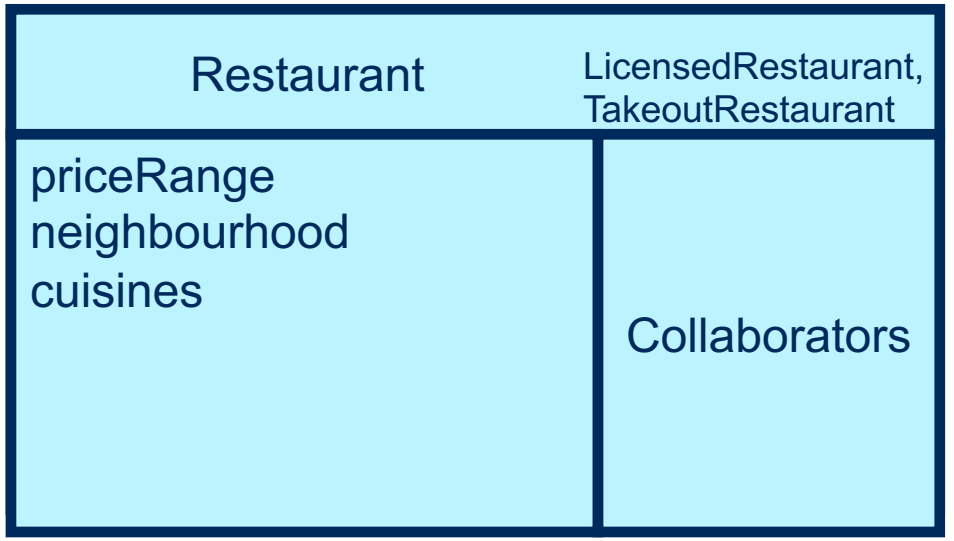
Restaurant	LicensedRestaurant, **TakeoutRestaurant**
~~**getAvgWaitTime**~~ priceRange neighbourhood cuisines	Collaborators

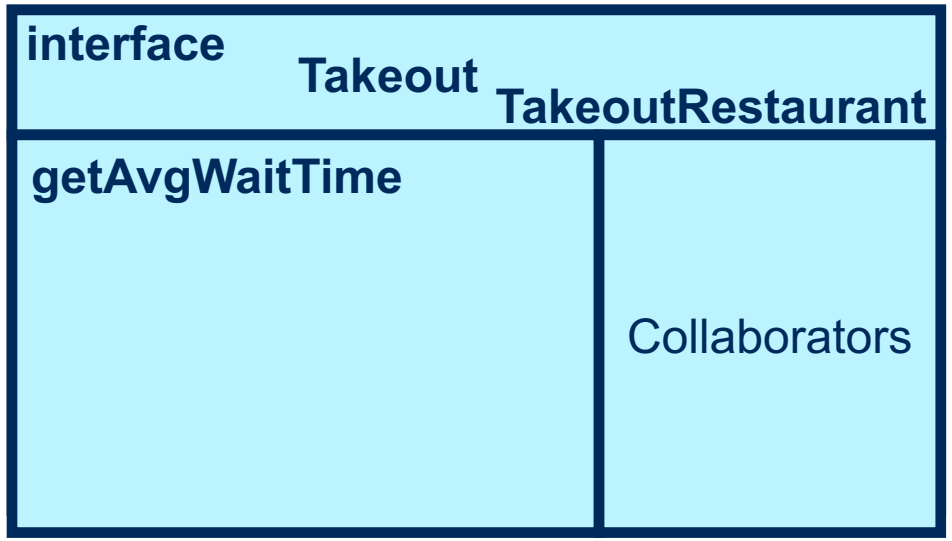
LicensedRestaurant	Restaurant
renewLicense license	Collaborators

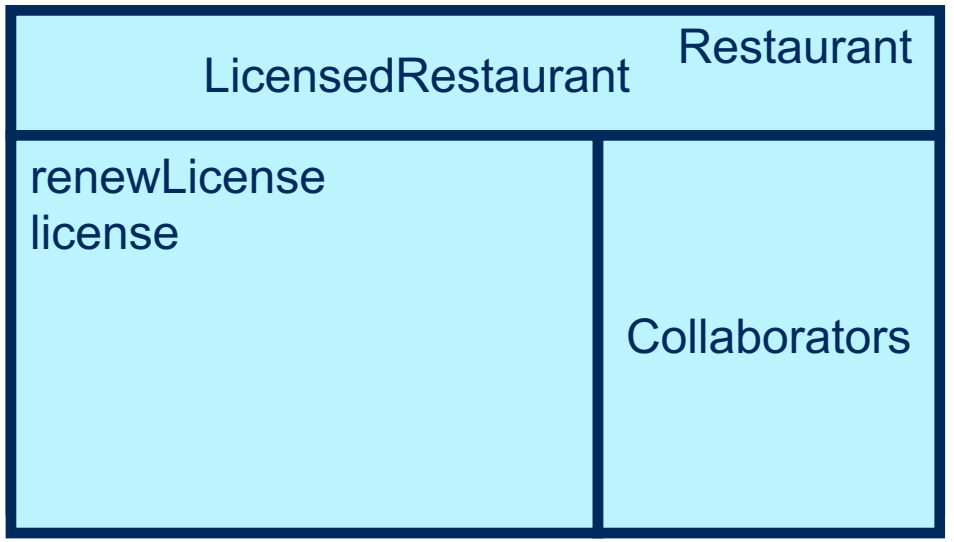
TAKEOUT AND LIQUOR LICENSE?

- 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.









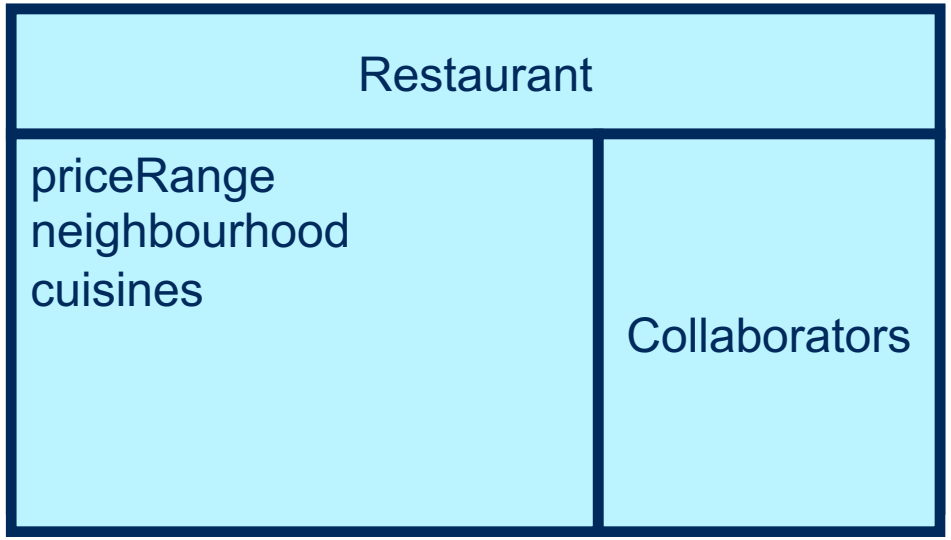
MORE ABOUT REVIEWS

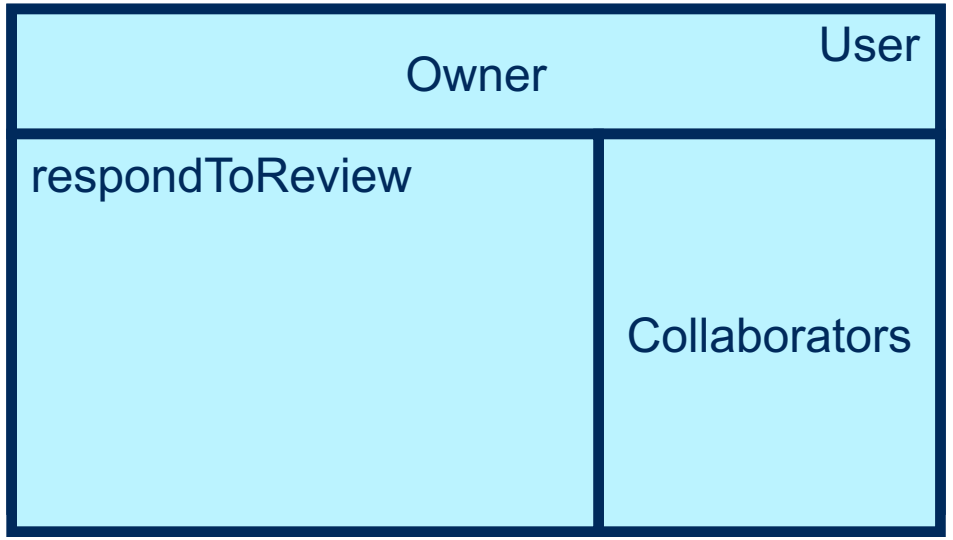
Let's look more closely at where it talks about reviews in our specification.

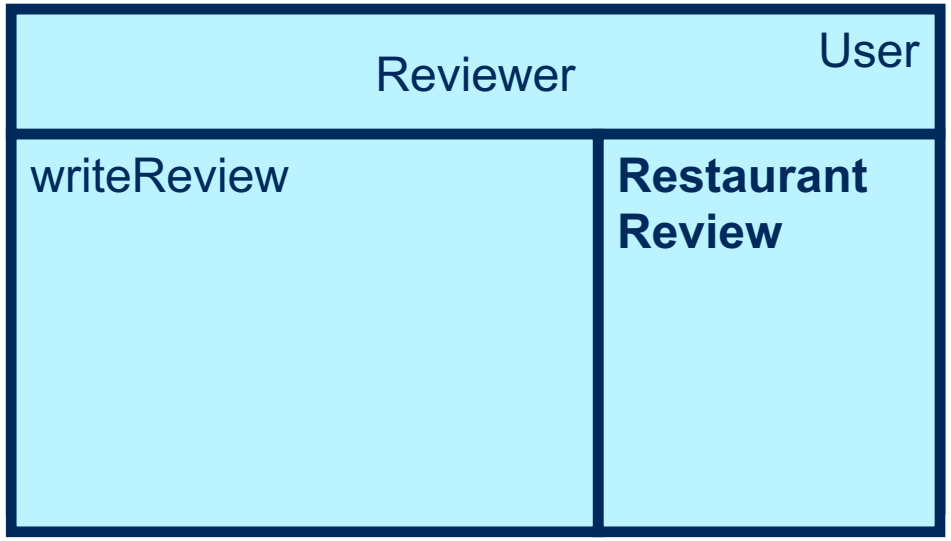
*When **reviewers** **leave a** **review** for a restaurant, they must specify **a** **recommendation** (Thumbs Up or Thumbs Down) and can also leave a comment. An **owner** of a restaurant can **respond to a review** with a comment. All **users** of the system **log in** with their username. Users can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

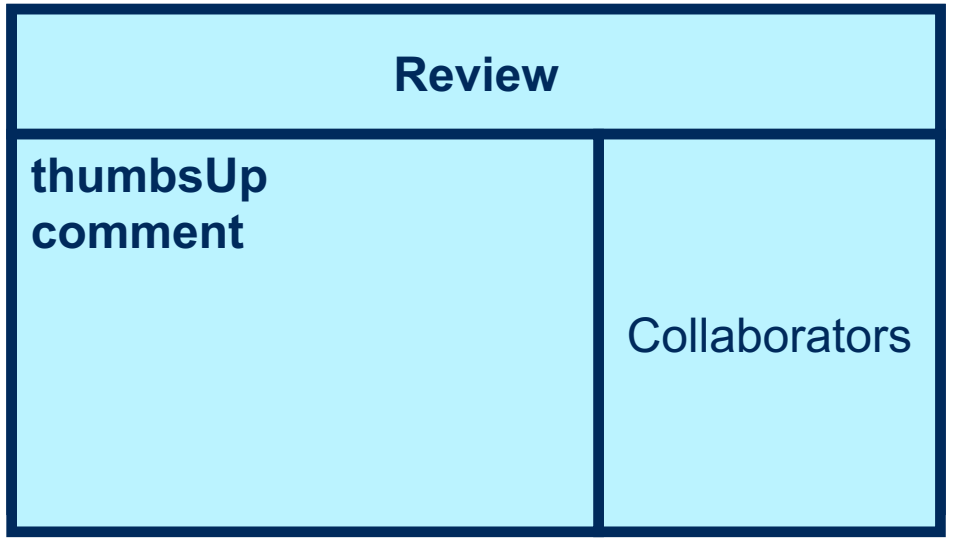
REVIEW CLASS

- 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.









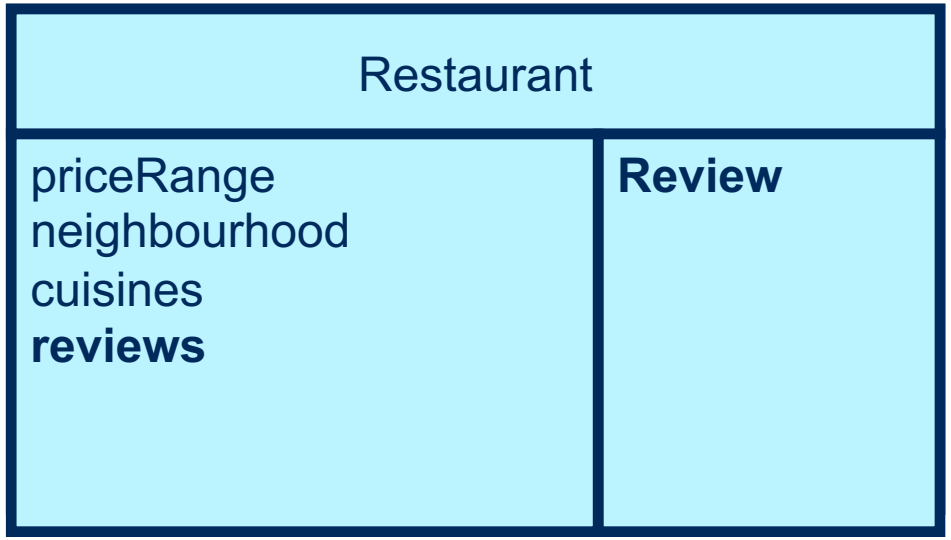
HOW DO REVIEWS WORK?

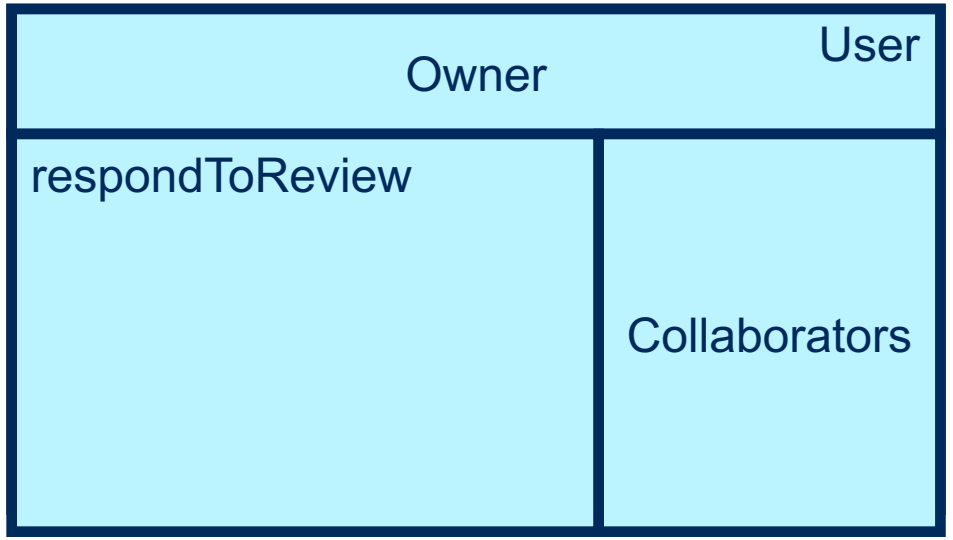
We have some design decisions to make here:

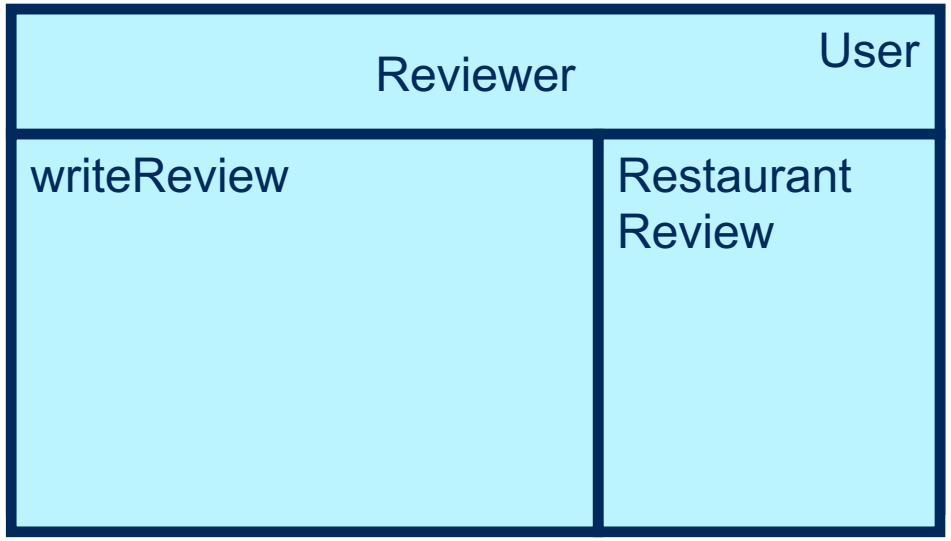
- Does a Review know which Restaurant it is for?
  - i.e. Is a Review responsible for storing which Restaurant it is for?
- Does a Review know who wrote it?
  - i.e. Is a Review responsible for storing which Reviewer wrote it?
- Where do Reviews live? With a Restaurant? With a Reviewer? Somewhere else?

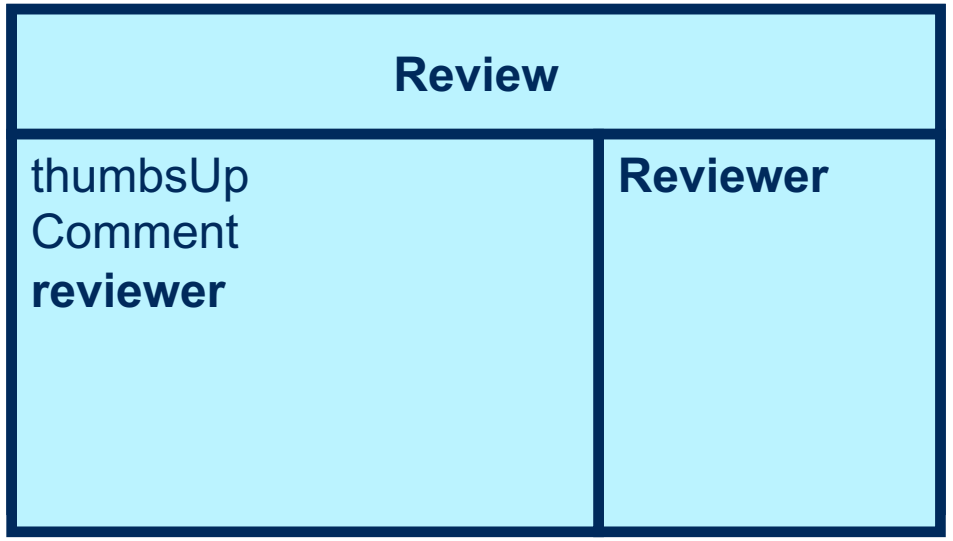
REVIEW CLASS – ONE POSSIBLE DESIGN

- 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.

SCENARIO WALK-THROUGH: RESPOND TO A REVIEW

For extra practice, try completing a walk-through for when “an owner responds to a review”.

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.

Discuss any design decisions with other students, on Piazza, or in office hours.

OUR CRC MODEL SO FAR

- Other classes omitted for space

Restaurant	
priceRange neighbourhood cuisines reviews addReview	Review

Owner	User
respondToReview	

Reviewer	User
writeReview	Restaurant Review

Review	
thumbsUp Comment reviewer	Reviewer

EXTRA PRACTICE

The following are three short specifications taken from the CSC148 course notes on object-oriented programming and designing classes. Try making CRC cards for each of them if you feel you need a bit 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 scenario walkthroughs to convince ourselves it will at least work.
- We could try implementing it and see if it results in working code.
- We can attempt to judge it based on established **design principles** (SOLID).
- We might ask: “How easy is the code to understand, maintain, and extend?”
- We can have an architecture in mind when developing our CRC model (in this course we’ll be using **Clean Architecture**)

HOMEWORK

- Weekly questions about CRC
- You'll get some practice with CRC in the next lab, and you'll work on a CRC model in your group project soon!

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

SRP

- 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?

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

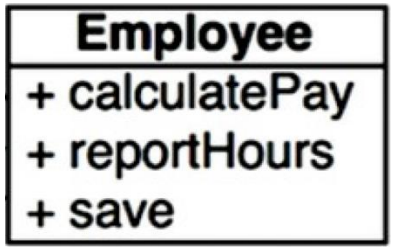
SINGLE RESPONSIBILITY PRINCIPLE

SRP

"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

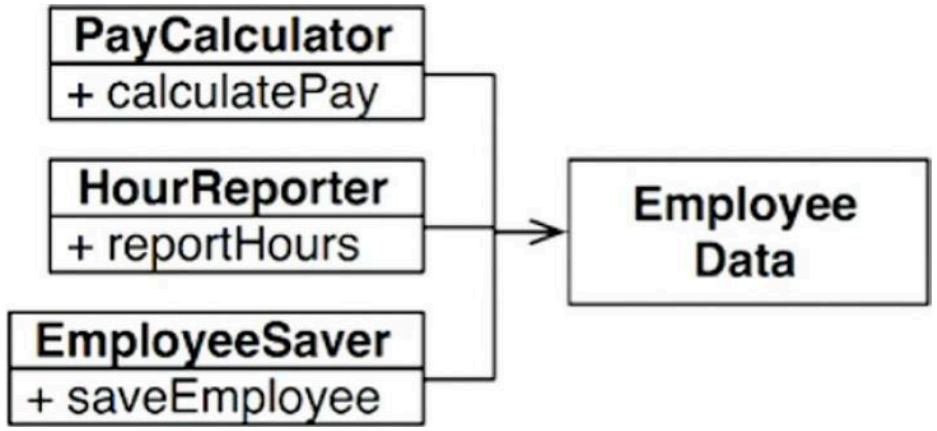
- Domain: an `Employee` class from a payroll application.
  - `calculatePay`: accounting department (CFO)
  - `reportHours`: human resources department (COO)
  - `save`: database administrators (CTO)



- 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?

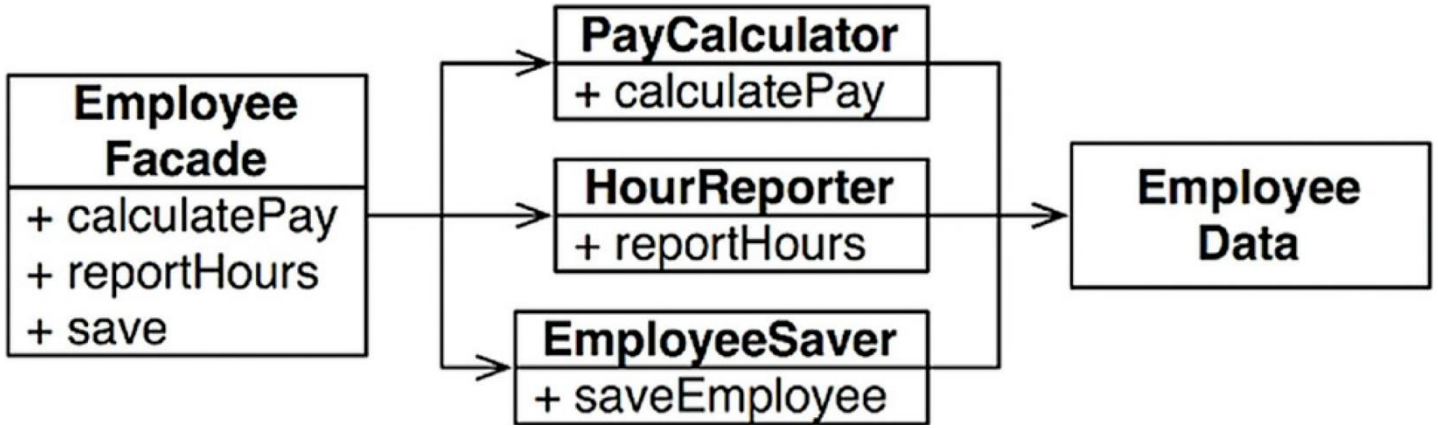
CAUSE OF PROBLEM AND SOLUTION

- 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

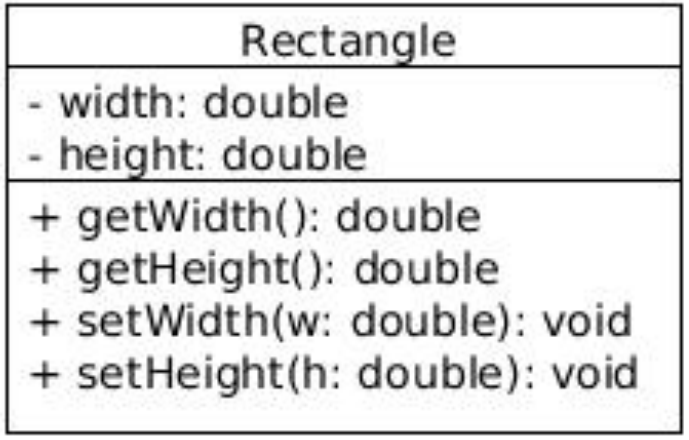
OCP

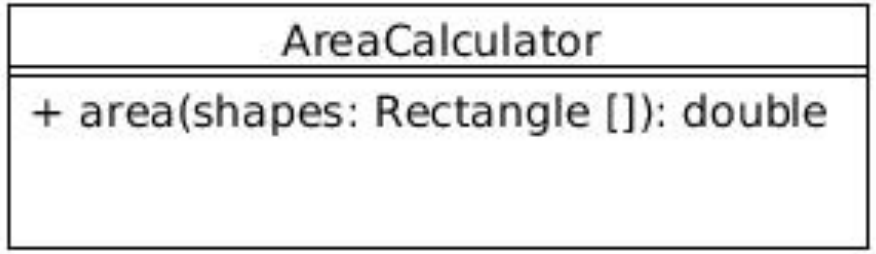
- Software entities (classes, modules, functions, etc.) should be **open for extension, but closed for modification**.
- Add new features not by modifying the original class, but rather by extending it and adding new behaviours, or by adding plugin capabilities.
- “I’ve heard it said that the OCP is 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

OCP

- 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?

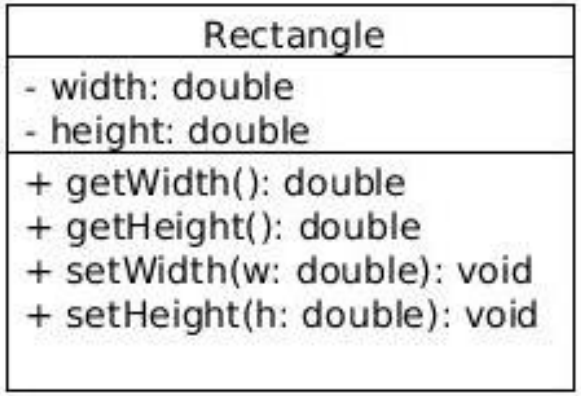


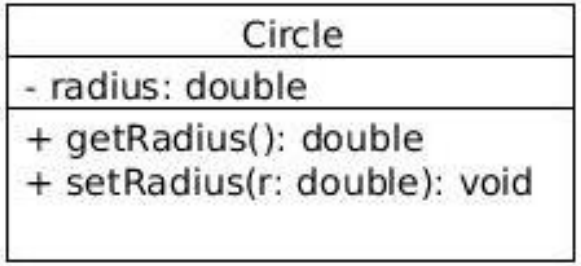


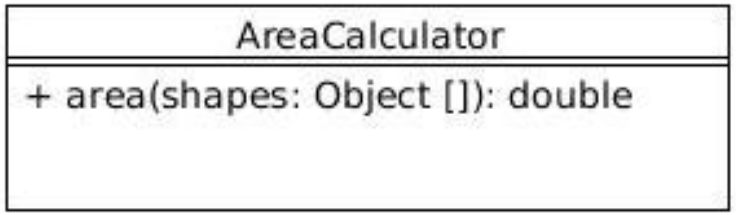
OPEN/CLOSED PRINCIPLE

OCP

- 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).





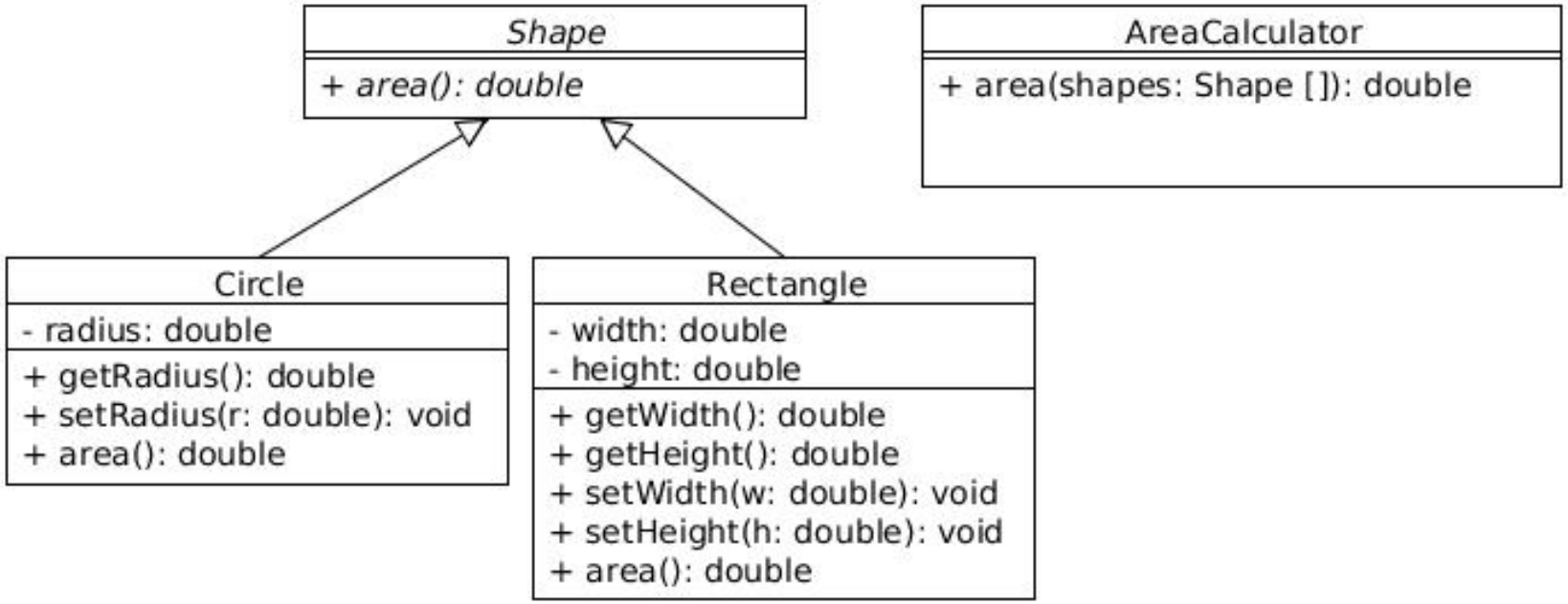


- But what if we need to add even more shapes?

OPEN/CLOSED PRINCIPLE

OCP

- 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`.

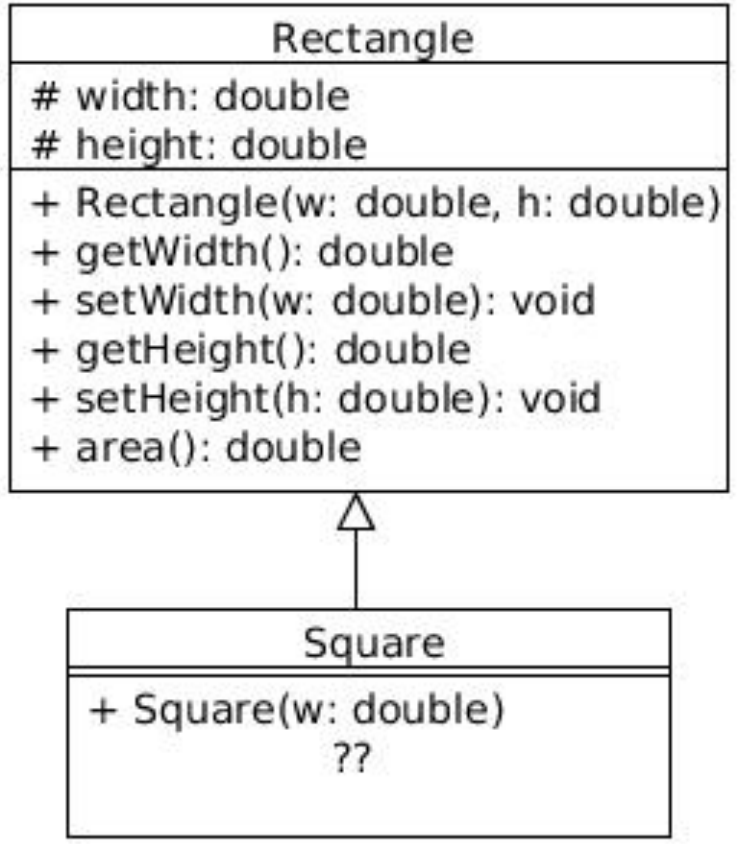
- "A program that uses an interface must not be confused by an implementation of that interface." [Uncle Bob]

LISKOV SUBSTITUTION PRINCIPLE

LSP

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

DIP

- 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

DIP

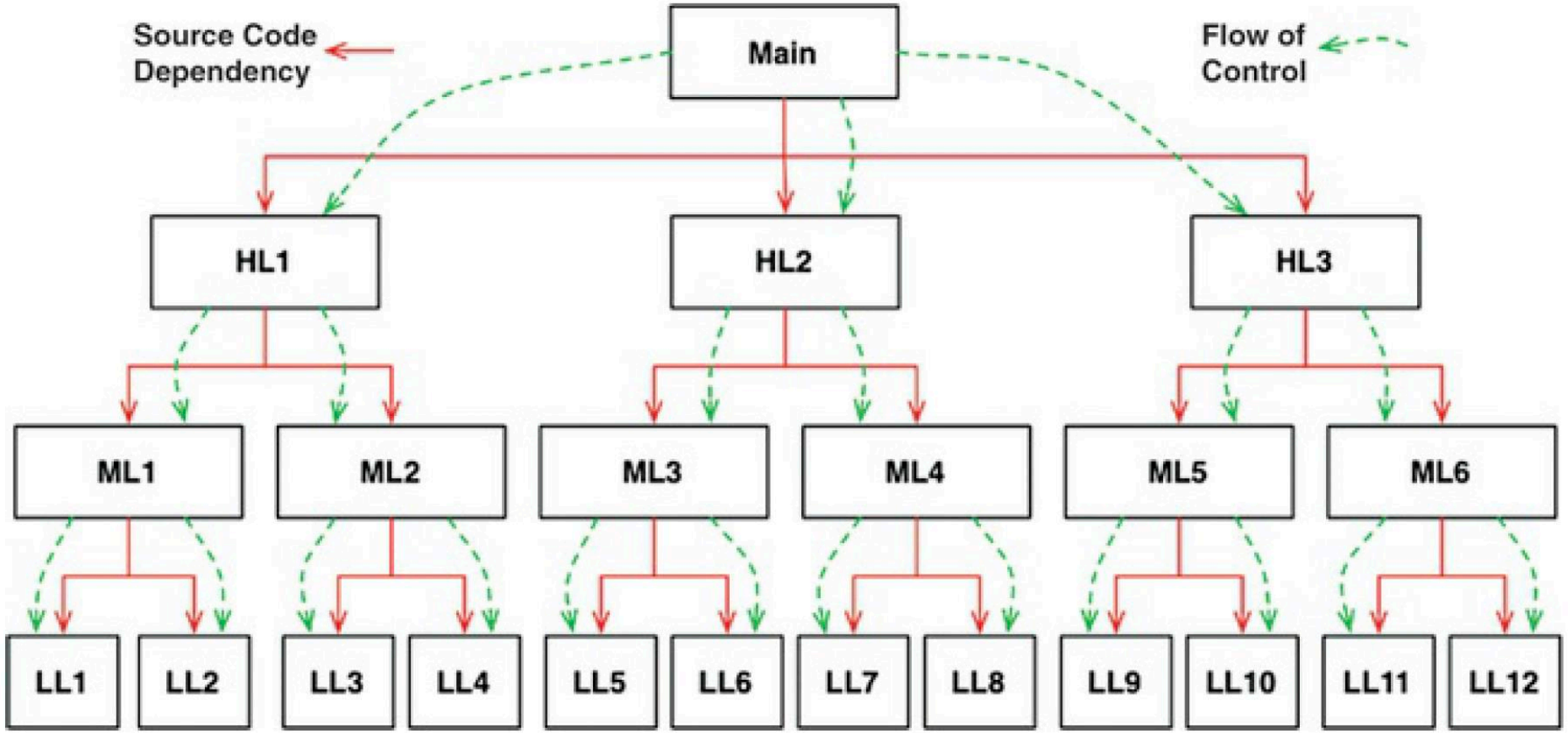


Figure 5.1 Source code dependencies versus flow of control

Clean Architecture, Robert C. Martin

DEPENDENCY INVERSION PRINCIPLE

DIP

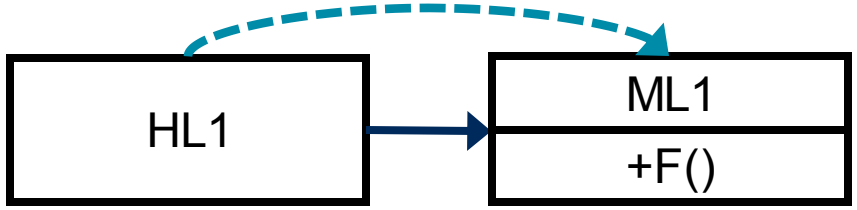
Goal:

- 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

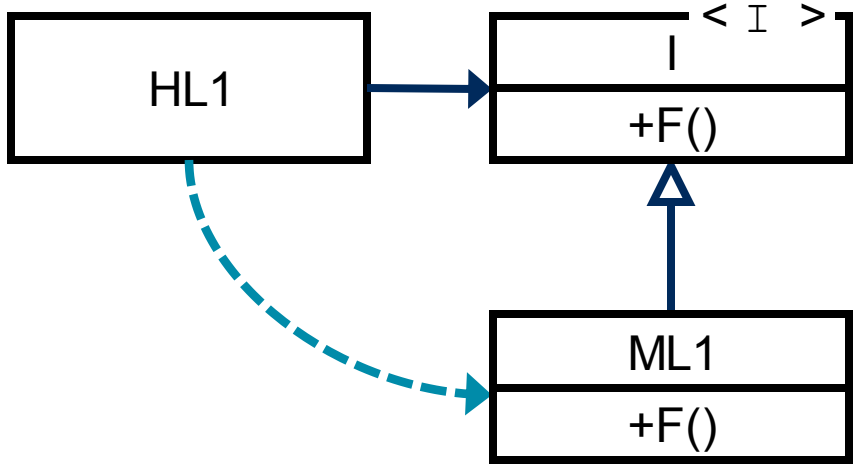
DIP

How do we invert the source code dependency?



We introduce an interface!

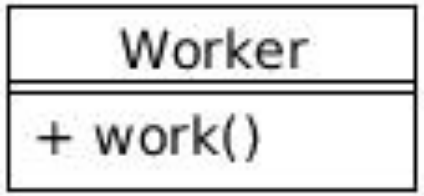
- The flow of control remains the same.
- HL1 depends on the interface and ML1 implements that interface.
- There is no longer a source code dependency between HL1 and ML1!

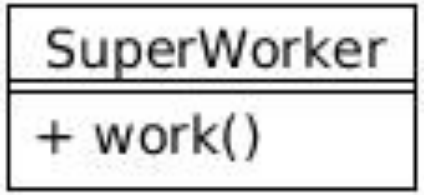


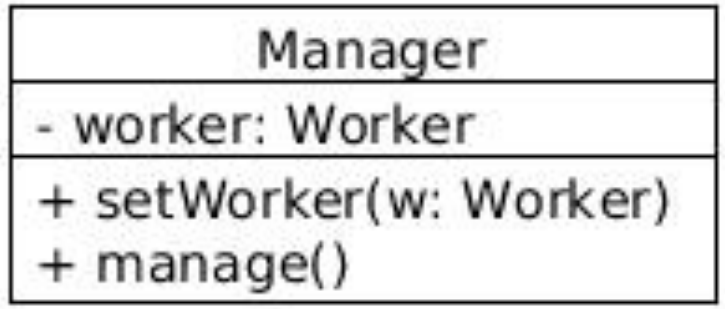
EXAMPLE FROM DEPENDENCY INVERSION PRINCIPLE ON OODESIGN

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".



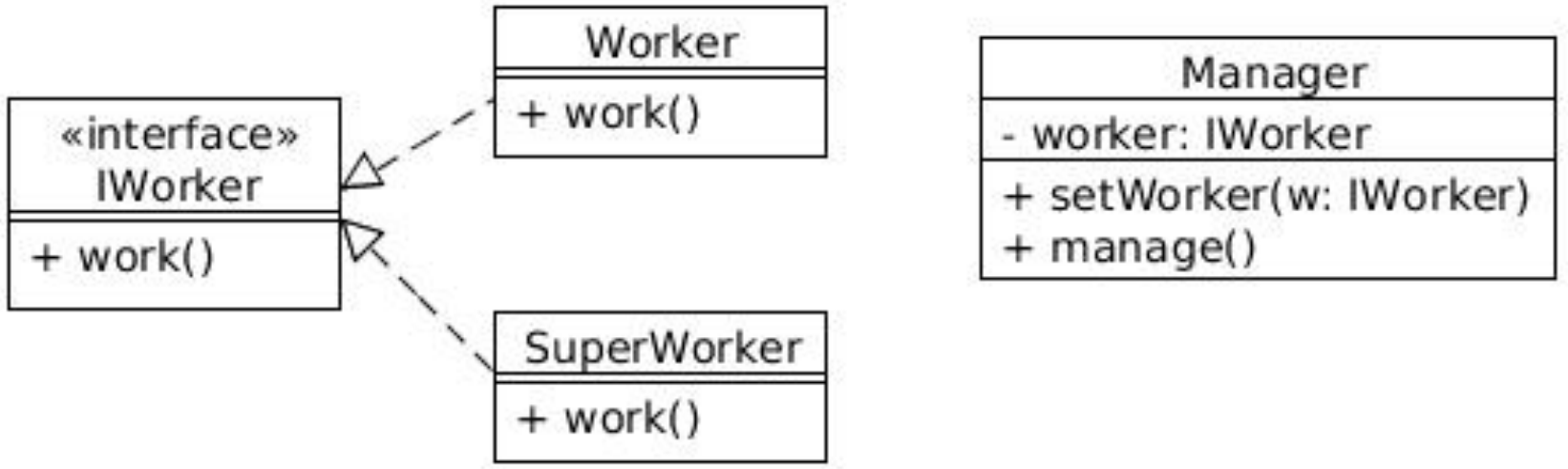




EXAMPLE FROM DEPENDENCY INVERSION PRINCIPLE ON OODESIGN

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

- Weekly questions about SOLID

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.

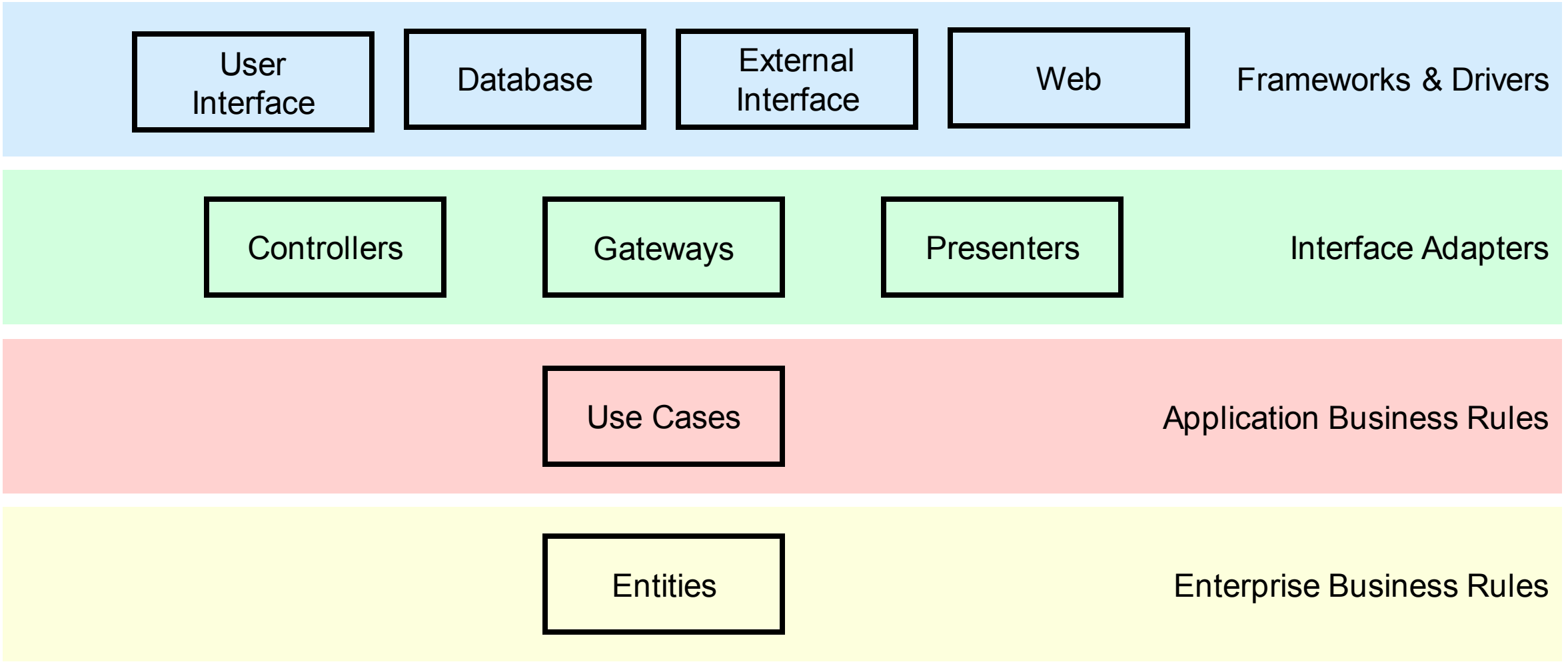
Use cases know nothing about the user interface or the data storage mechanism!

Gather Contact Info for New Loan

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

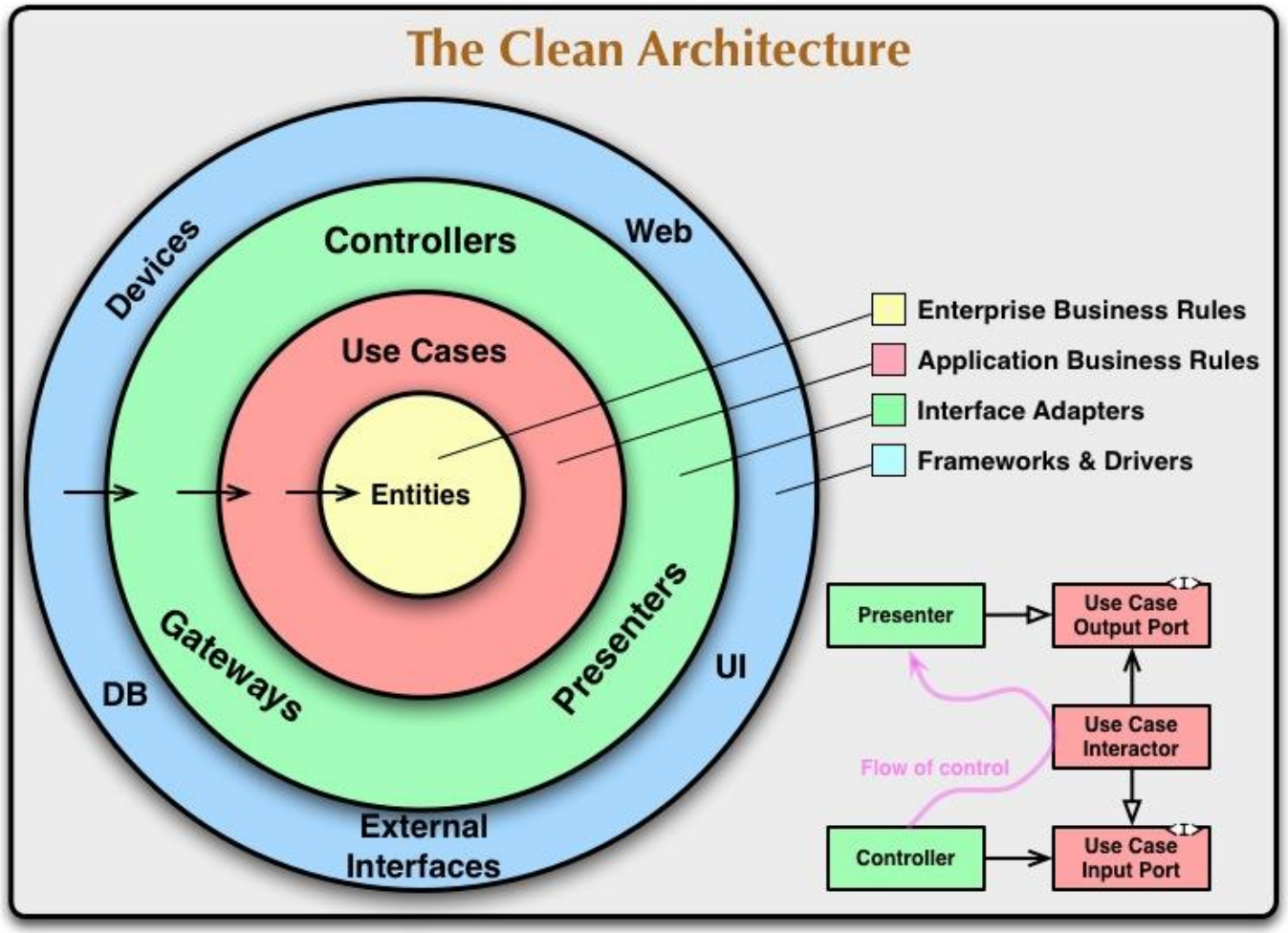
Primary Course:
1. Accept and validate name.
2. Validate address, birthdate, D.L.#, SSN, etc.
3. Get credit score.
4. If credit score is < 500 activate Denial.
5. Else create Customer and activate Loan Estimation.

CLEAN ARCHITECTURE



CLEAN ARCHITECTURE

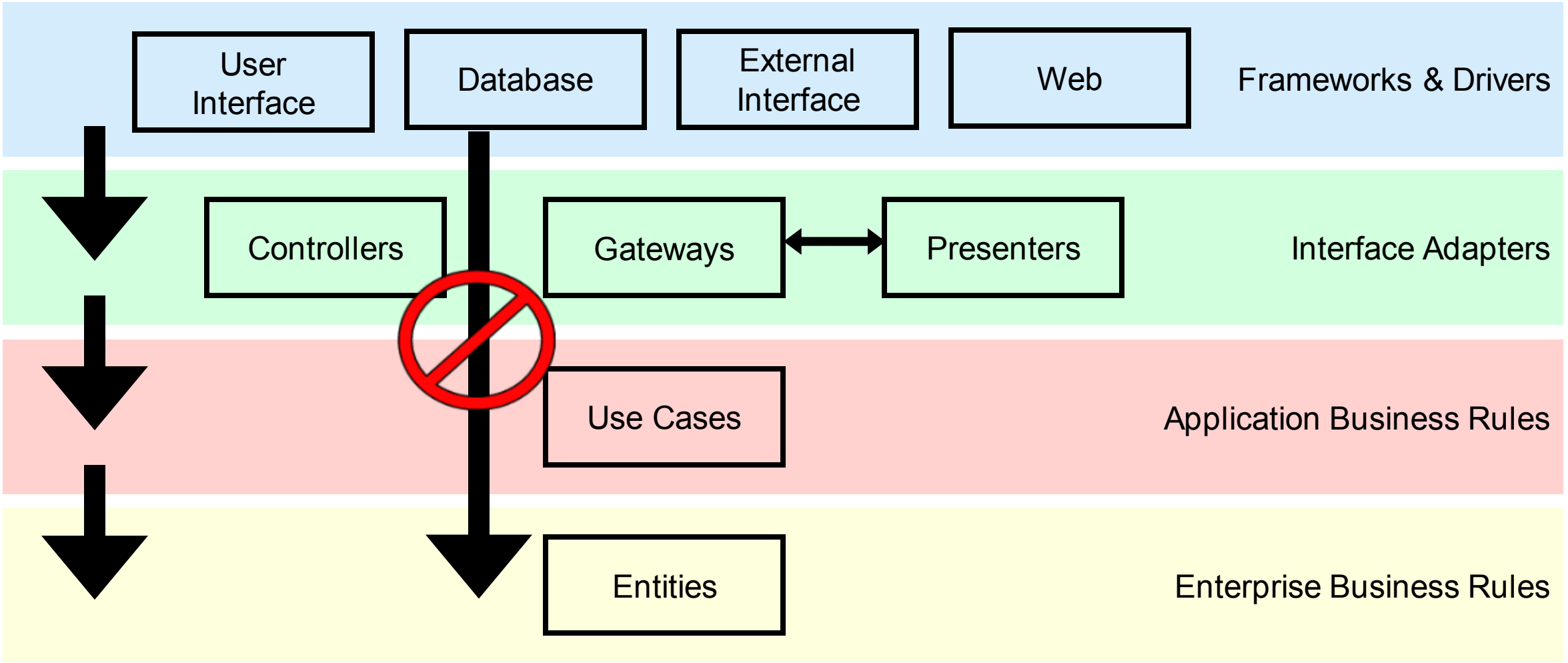
- Often visualize the layers as concentric circles.
- Reminds us that Entities are at the core
- Input and output are both in outer layers



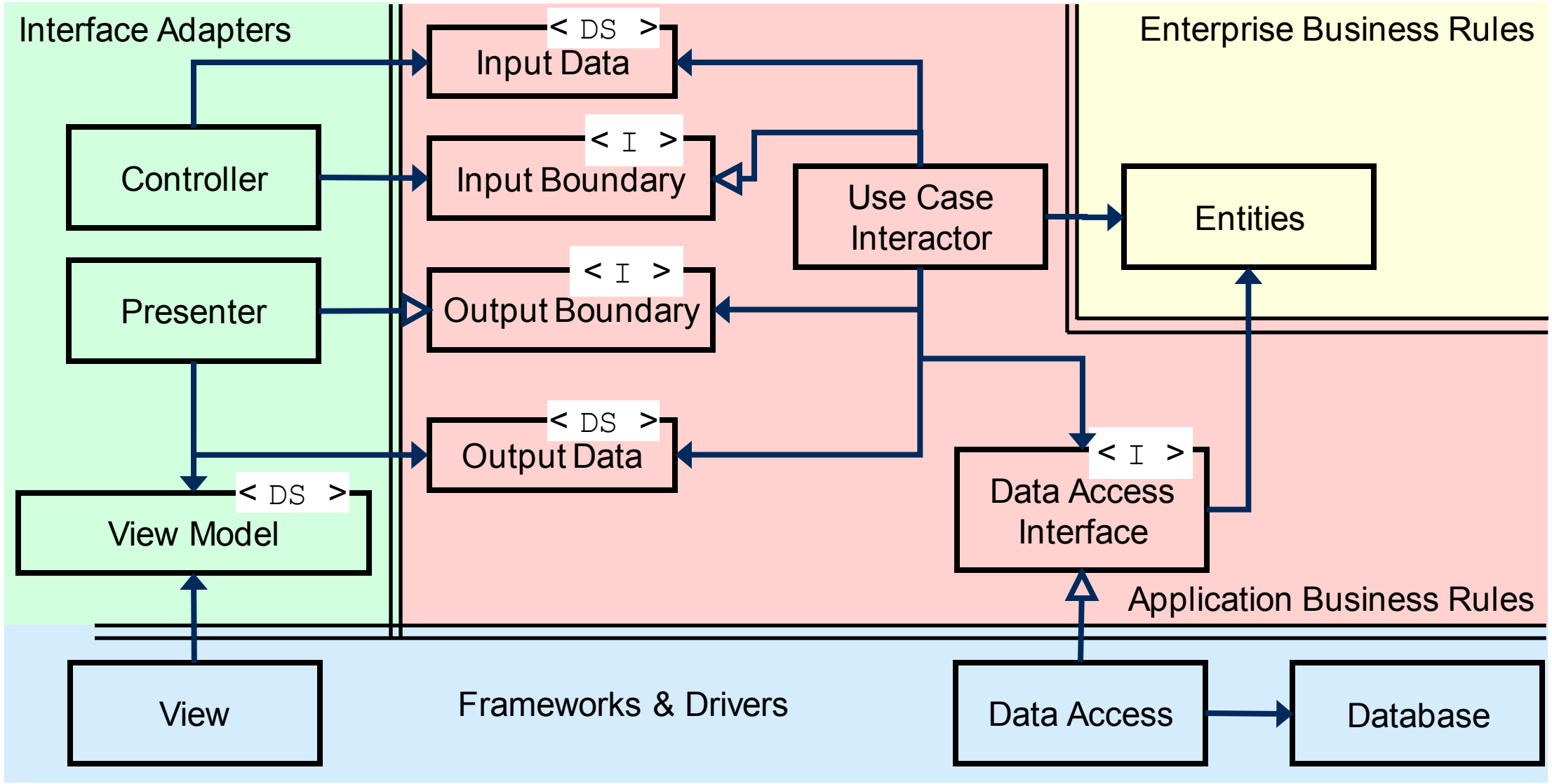
CLEAN ARCHITECTURE – DEPENDENCY RULE

- 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.

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!

HOMEWORK

- Weekly questions about Clean Architecture

UML & DESIGN PATTERNS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

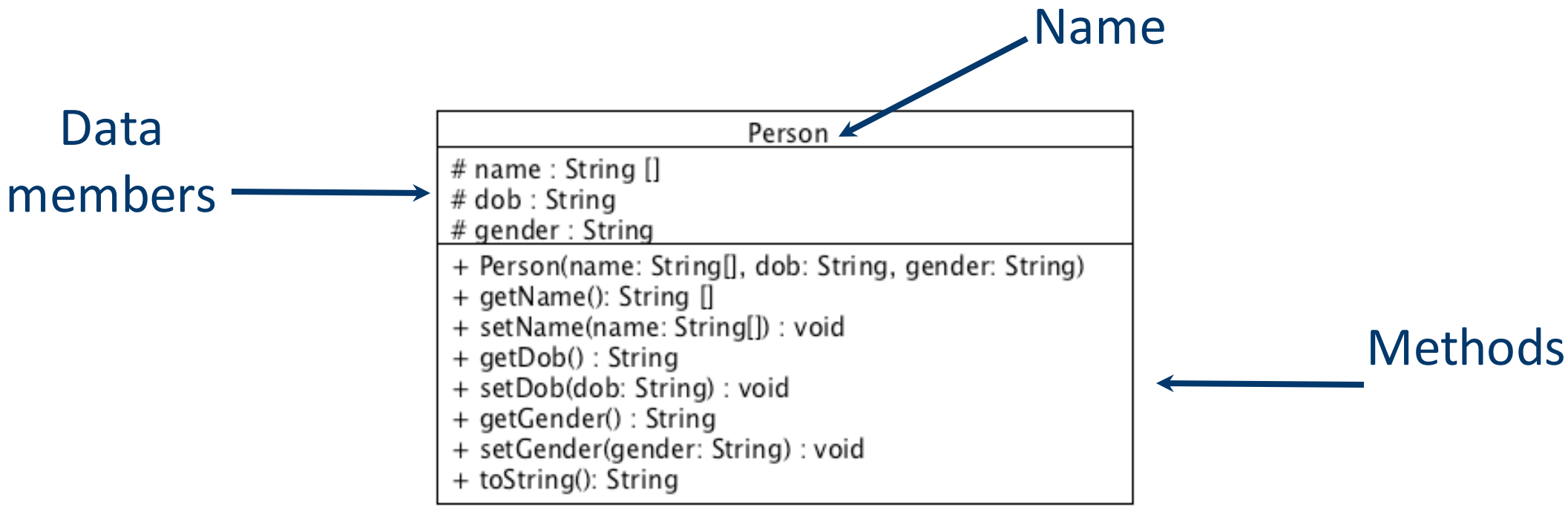
LEARNING OUTCOMES

- 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.

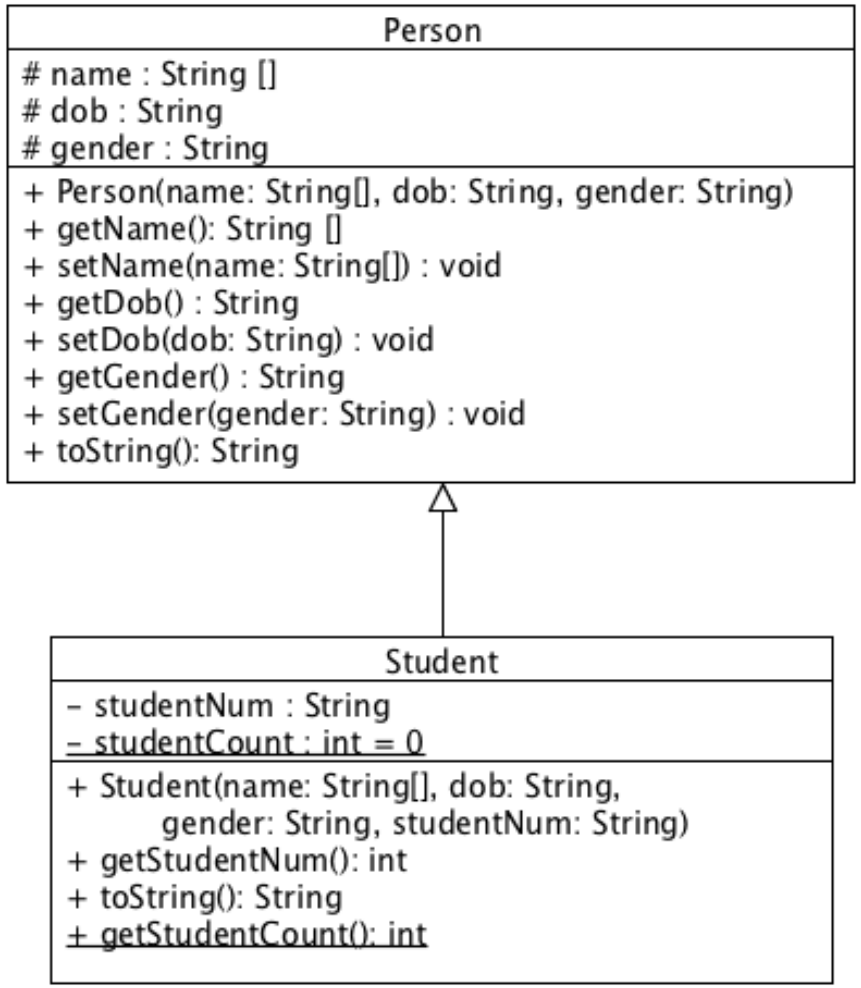
EXAMPLE: CLASS PERSON



NOTATION

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

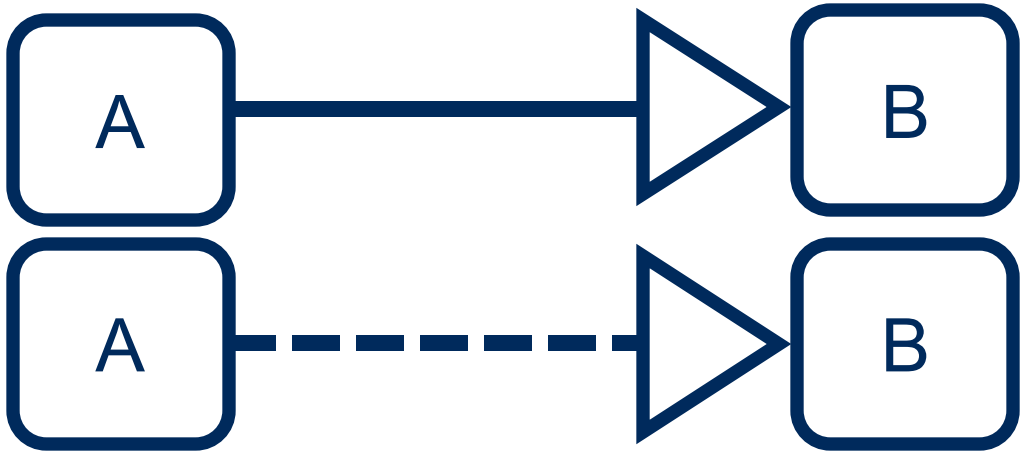
EXAMPLE: INHERITANCE



NOTATION (CONT'D)

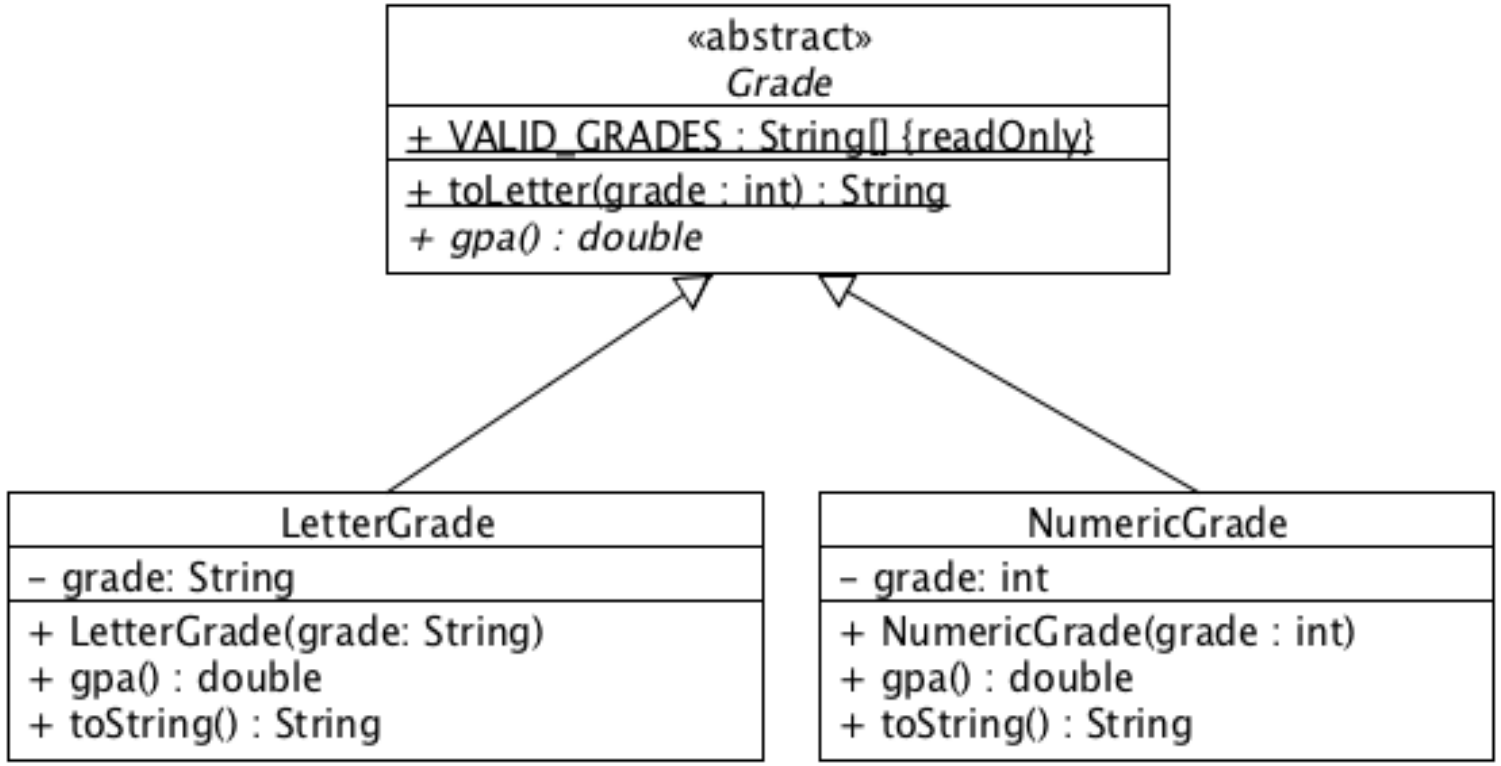
- Abstract method: *italic*
- Abstract class: *italic* or <<abstract>>
- Interface: <<interface>>
- Relationship between classes:

Inheritance
(A inherits from B)



Interface
(A implements B)

EXAMPLE: ABSTRACT CLASS



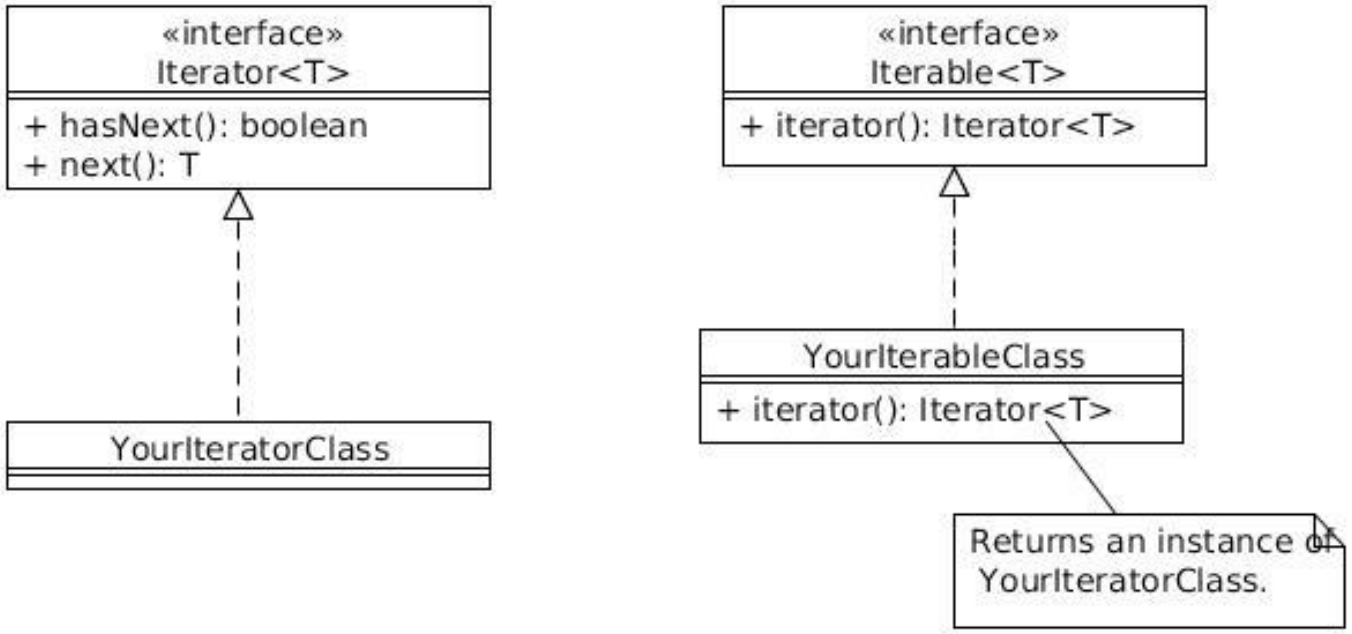
DESIGN PATTERNS

- 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

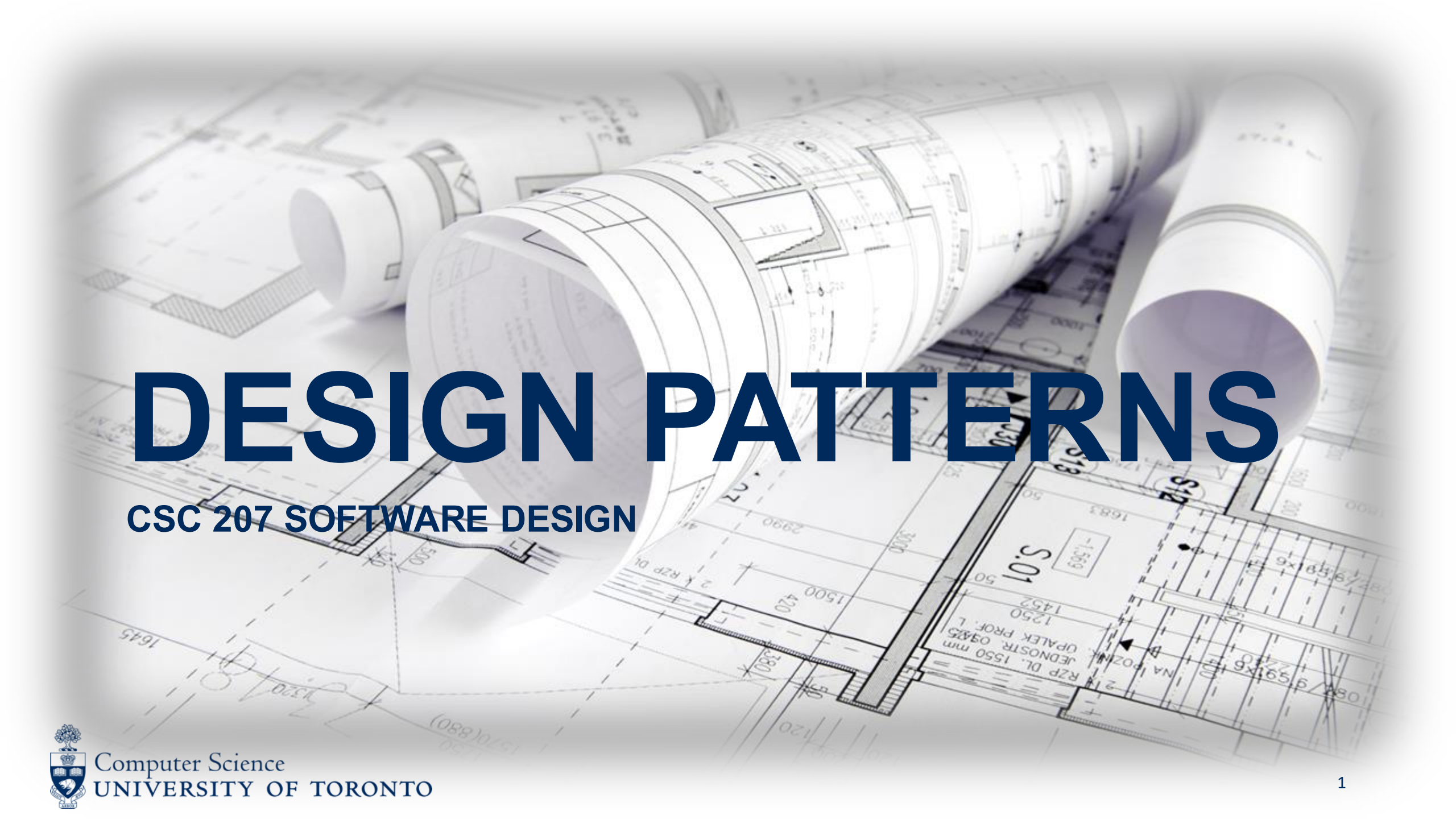


HOMEWORK

- Weekly questions about design patterns
- Start working on Design Question 1

HOMEWORK

- Weekly questions about design patterns
- Start working on Design Question 1 if you have not yet done so



DESIGN PATTERNS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

LEARNING OUTCOMES

- Recognize some common design patterns and how to apply them.

DESIGN PATTERNS

- We'll be covering a few common design patterns this term.
- Iterator, Observer, Strategy, Dependency Injection, Factory Method, Façade, Builder
- 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.

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

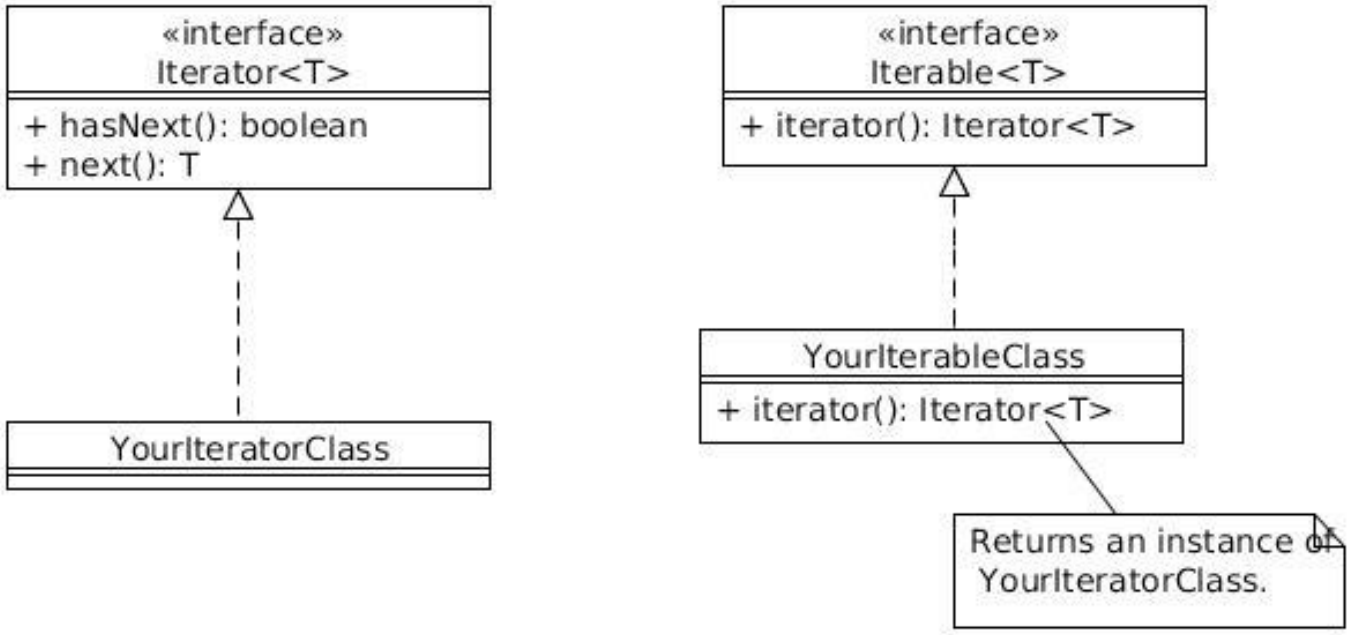
THE BOOK PROVIDES AN OVERVIEW OF:

- **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

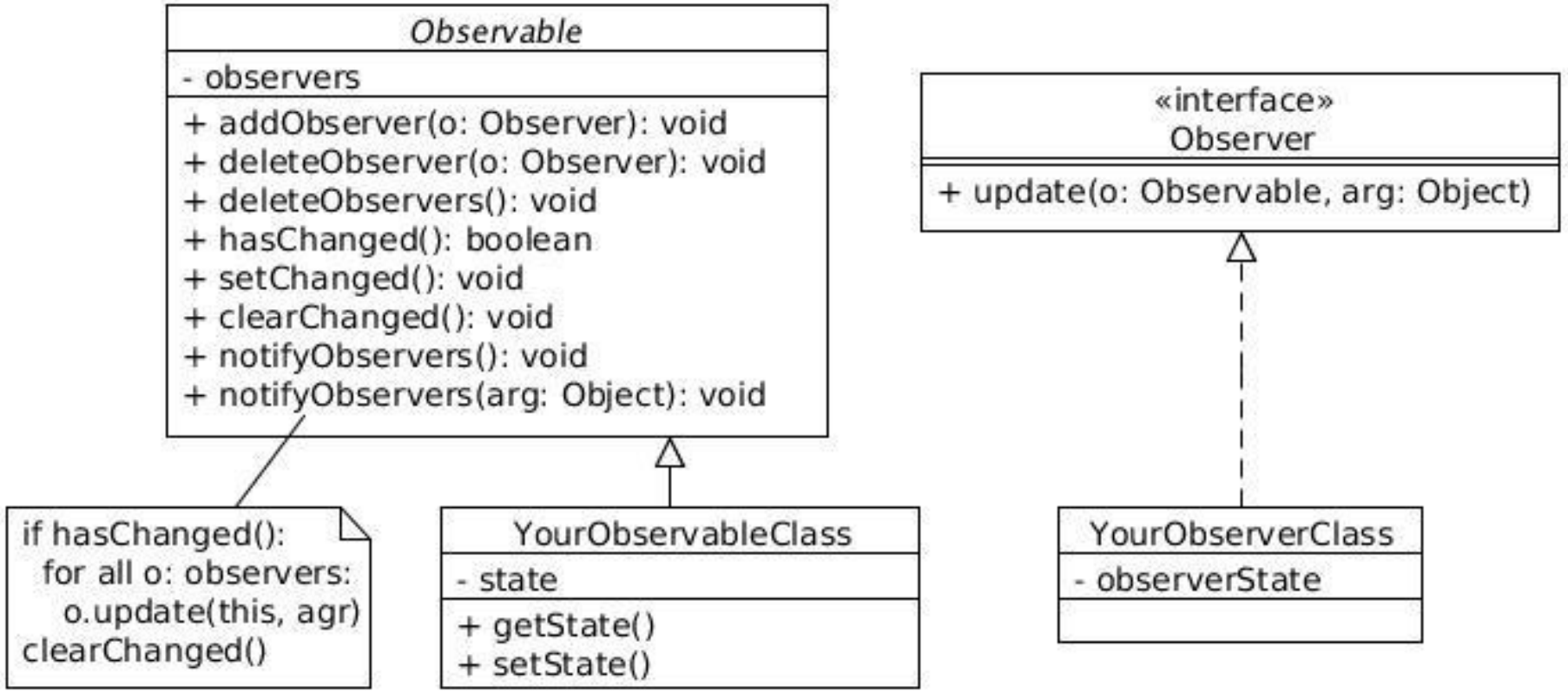


Example: look at the source code for ArrayList and its ArrayList.iterator() method (in IntelliJ or online)!

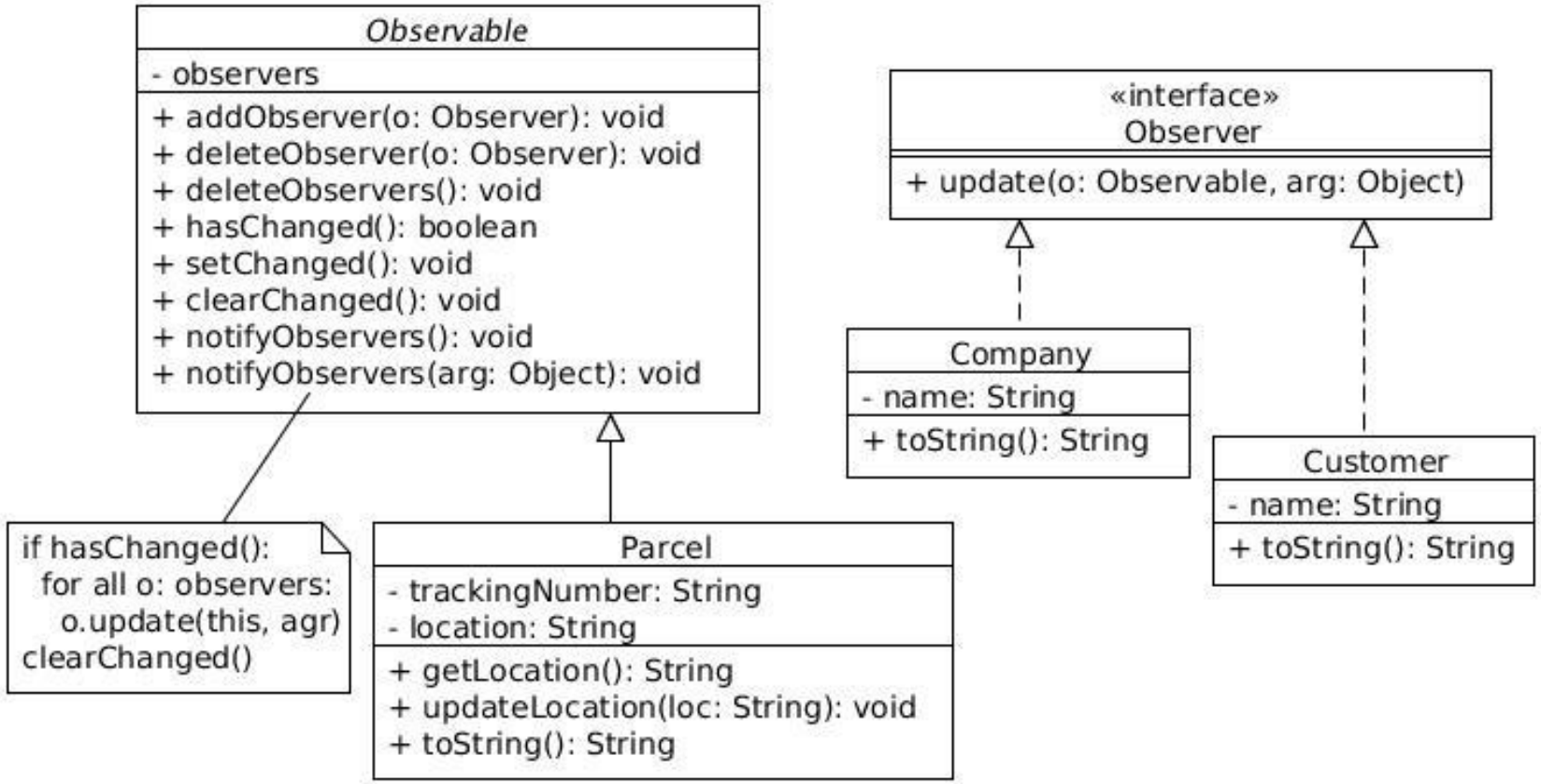
OBSERVER DESIGN PATTERN

- 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



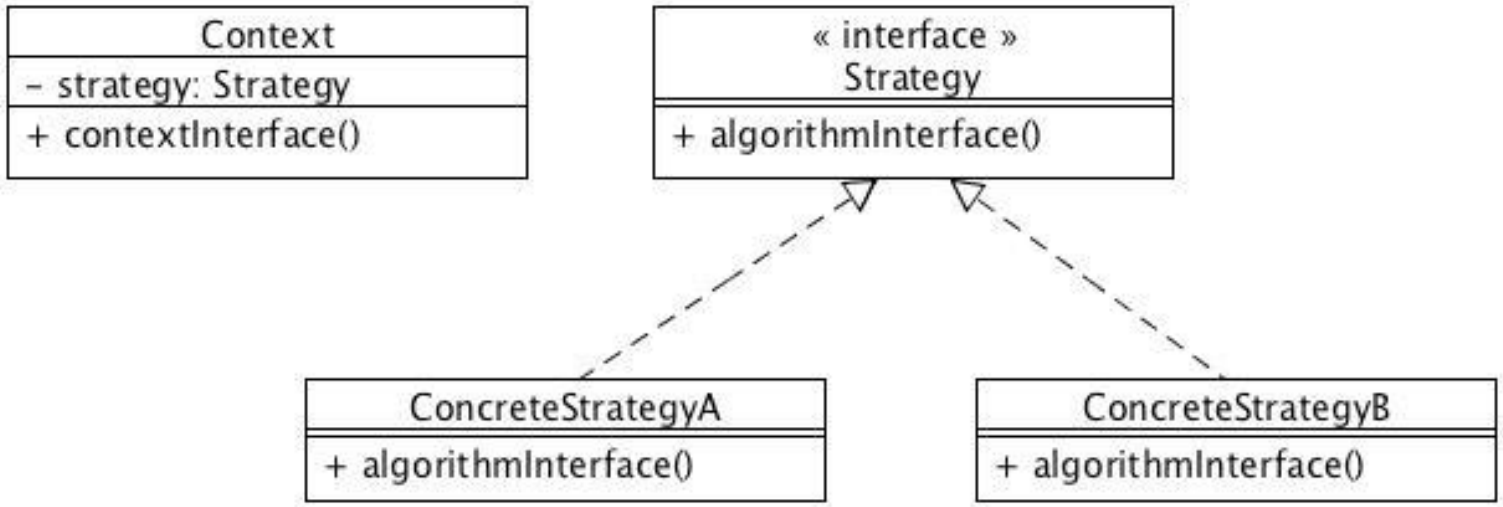
OBSERVER: PARCEL EXAMPLE IN JAVA



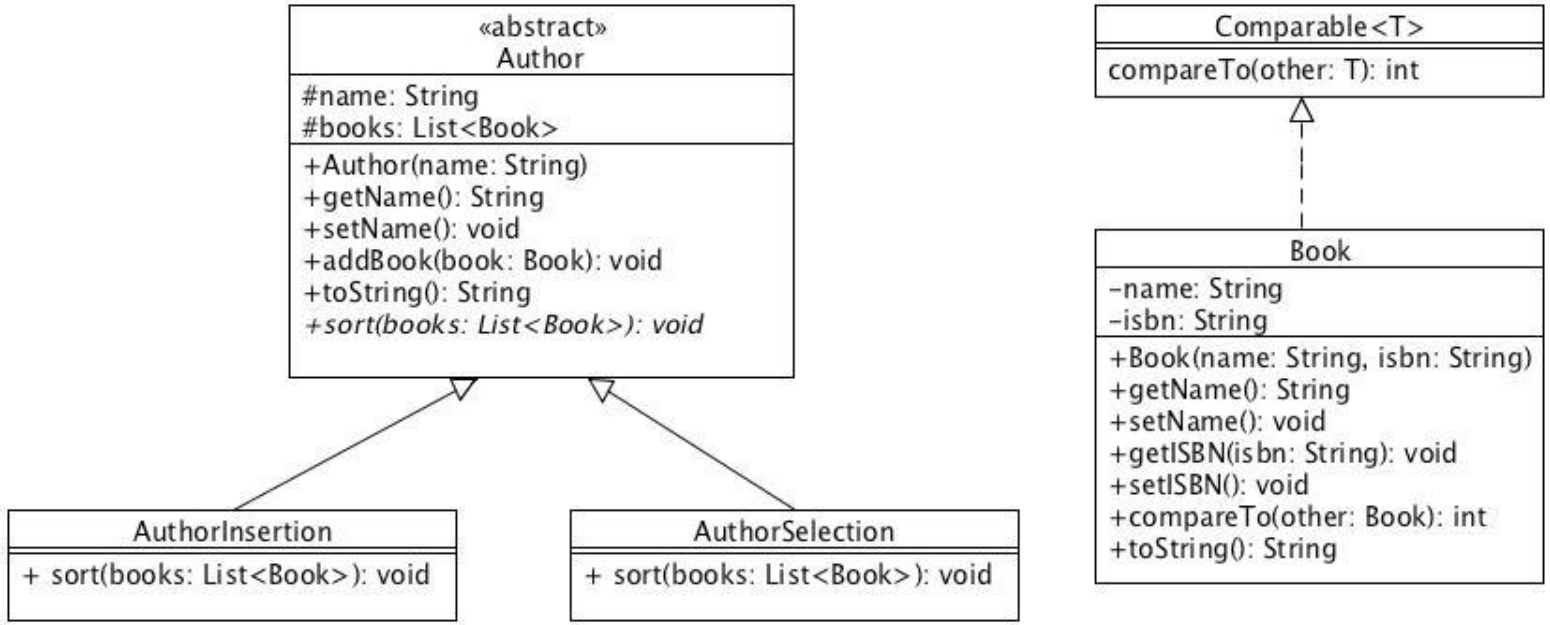
STRATEGY DESIGN PATTERN

- 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

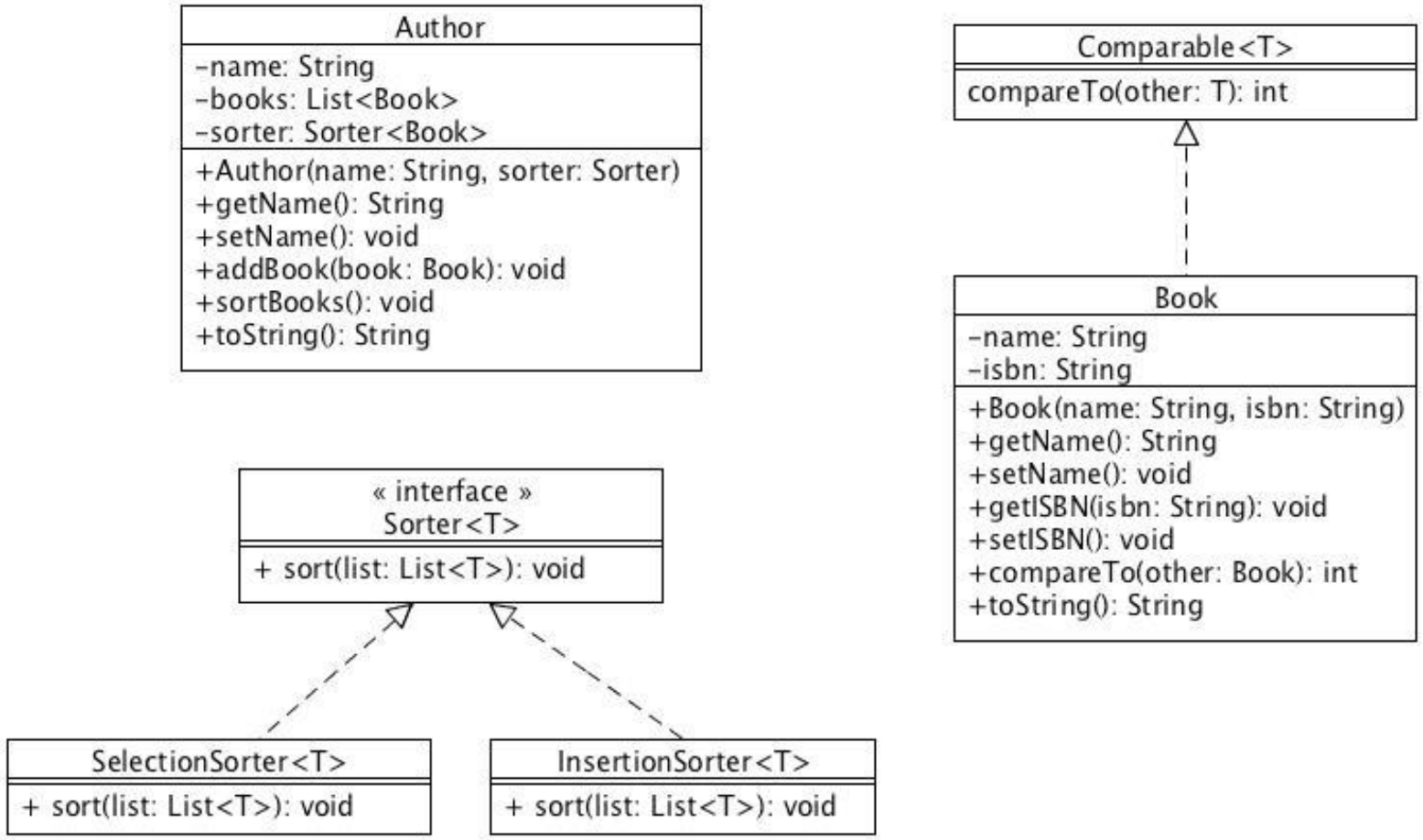
STRATEGY: STANDARD SOLUTION



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);
    }
}
```

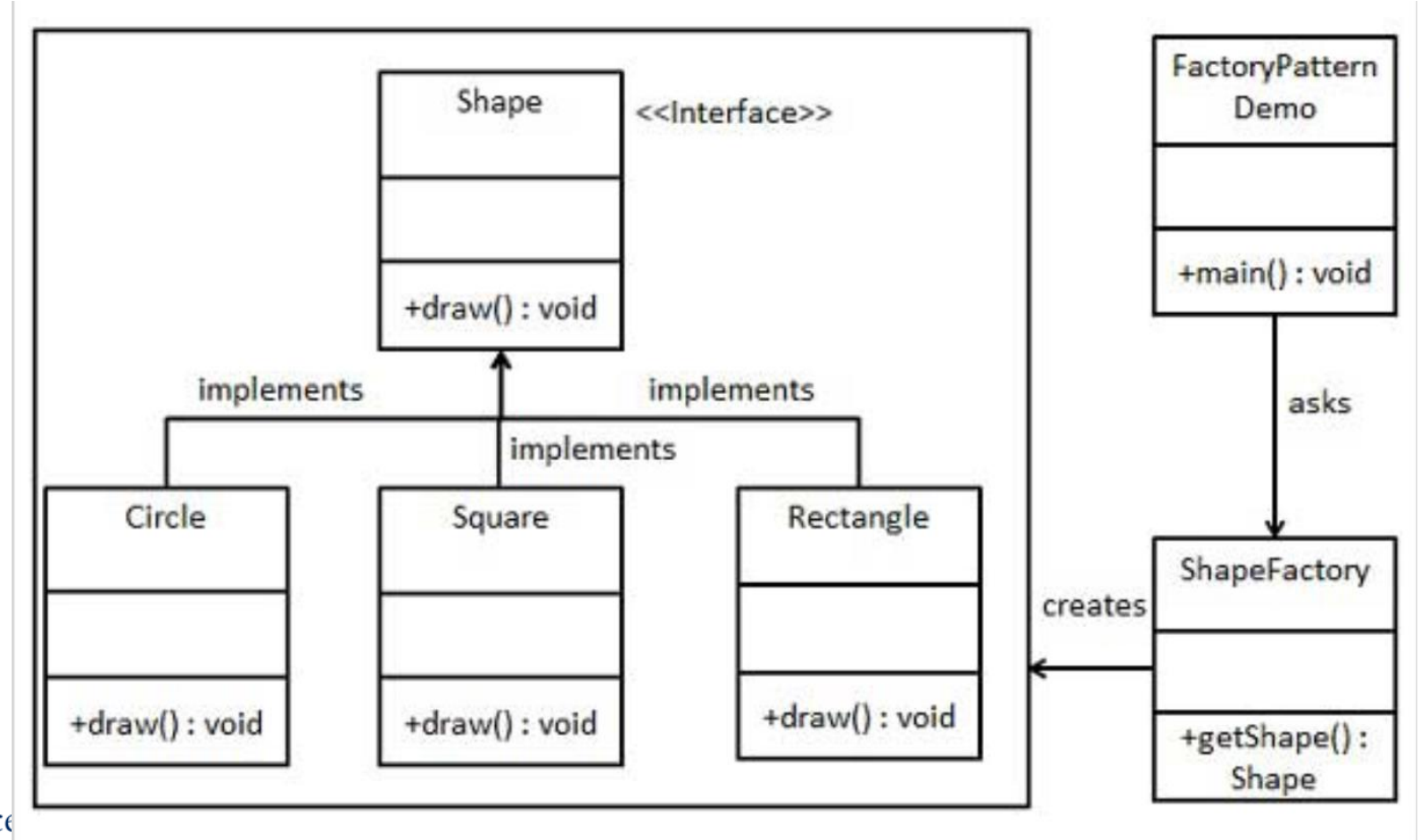
DEPENDENCY INJECTION PATTERN EXERCISE (FOR EXTRA PRACTICE)

- Consider our example of class `Person` and its subclass `Student`.
- Create a class called Email that contains instance variables of type `String` called: `to`, `from`, `cc`, `subject`, and `message`. It should also contain methods `printToScreen` and `printToFile` methods.
- Class `Person` should have the ability to send an `Email` object to another `Person` object. But that will require our first instance of `Person` to obtain the second `Person` object's name or email address.

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

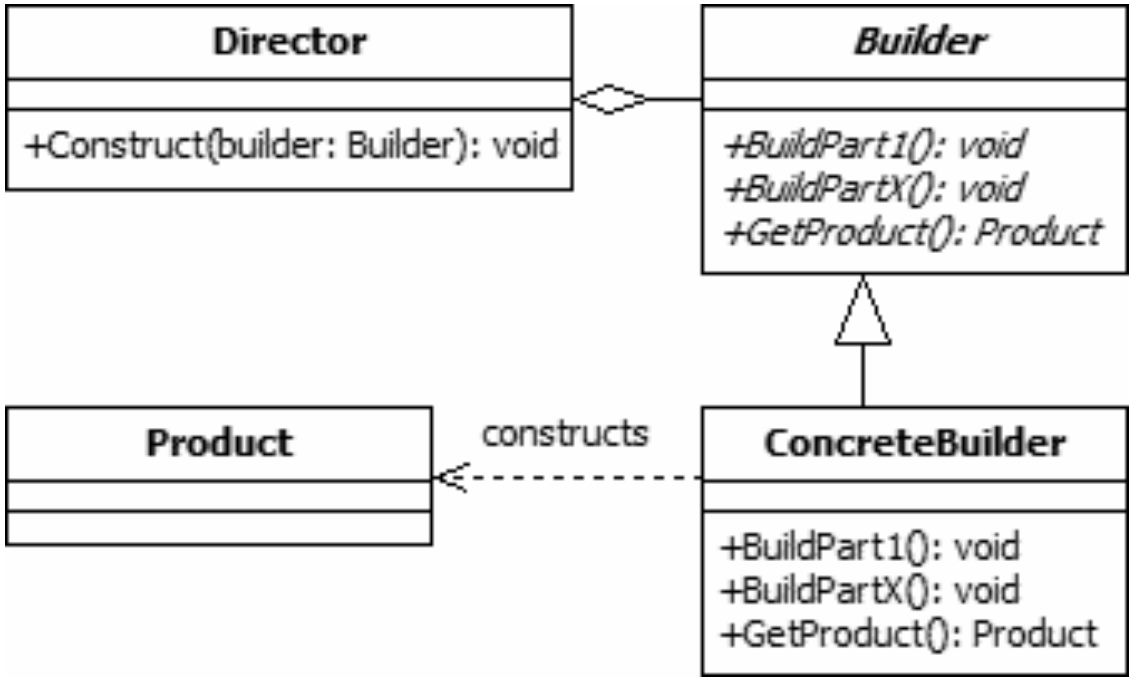
- Factor out an `Order` object that contains the menu items that were ordered.
- Create classes called `BillCalculator`, `BillLogger`, and `BillPrinter` that all use Order.
- Create `BillFacade`, which **delegates** the operations to `BillCalculator`, `BillLogger`, and `BillPrinter`.
- For example, `BillFacade`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

- Weekly questions about design patterns
- Start working on Design Question 1 if you have not yet done so

SERIALIZATION & PERSISTENT DATA

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

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)
- 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)

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

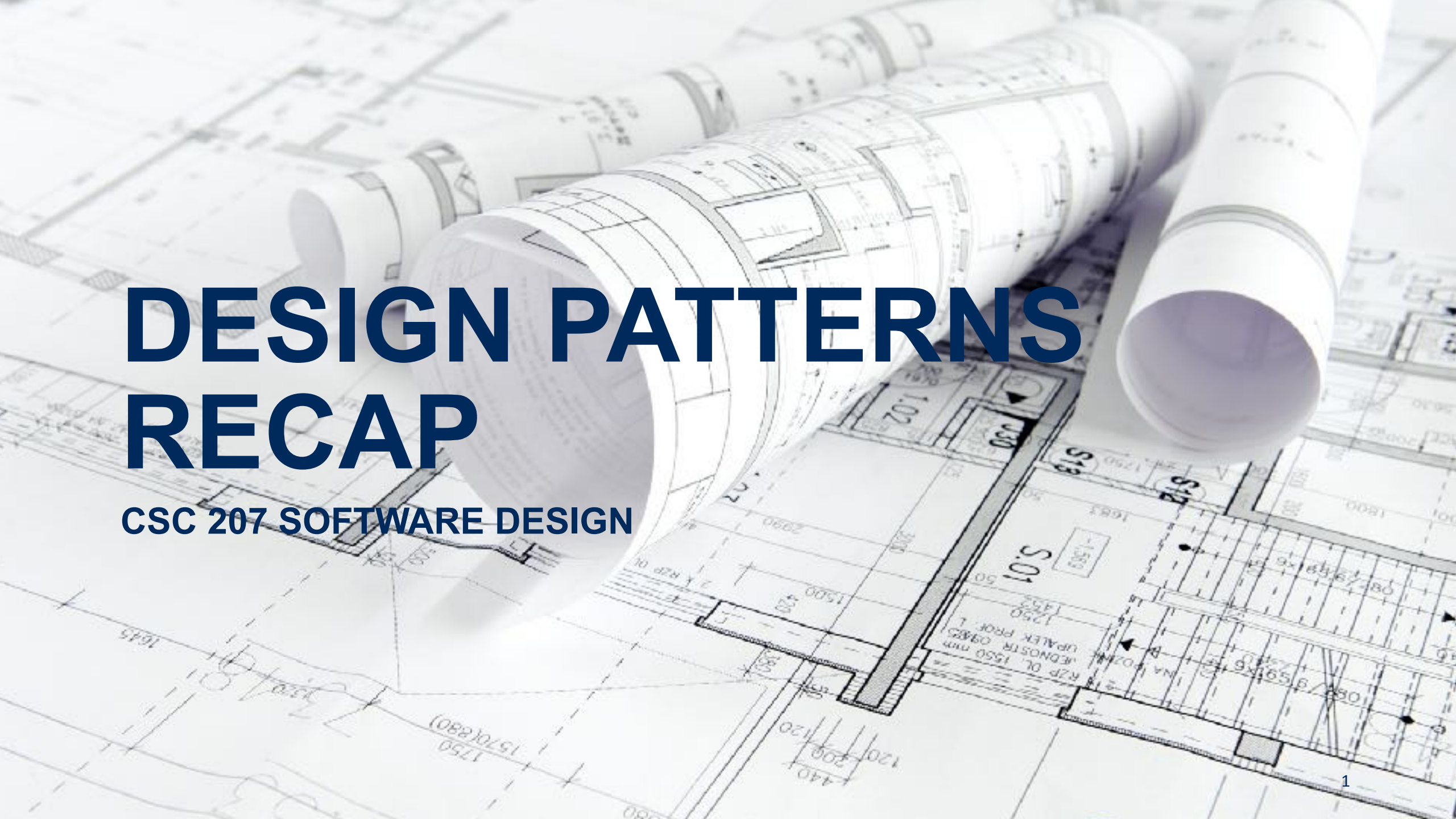
- https://www.baeldung.com/java-serialization

A tutorial about reading / writing files in Java

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

HOMEWORK

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



DESIGN PATTERNS RECAP

CSC 207 SOFTWARE DESIGN

LEARNING OUTCOMES

- Reinforce the design patterns we covered.

THE DESIGN PATTERNS WE COVERED

- Iterator
- Observer
- Strategy
- “Dependency Injection”
- “Factory Method”
- Façade
- Builder
- 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.
- They're a means of **communicating** design ideas.
- **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.
- Examples you've seen in the course?

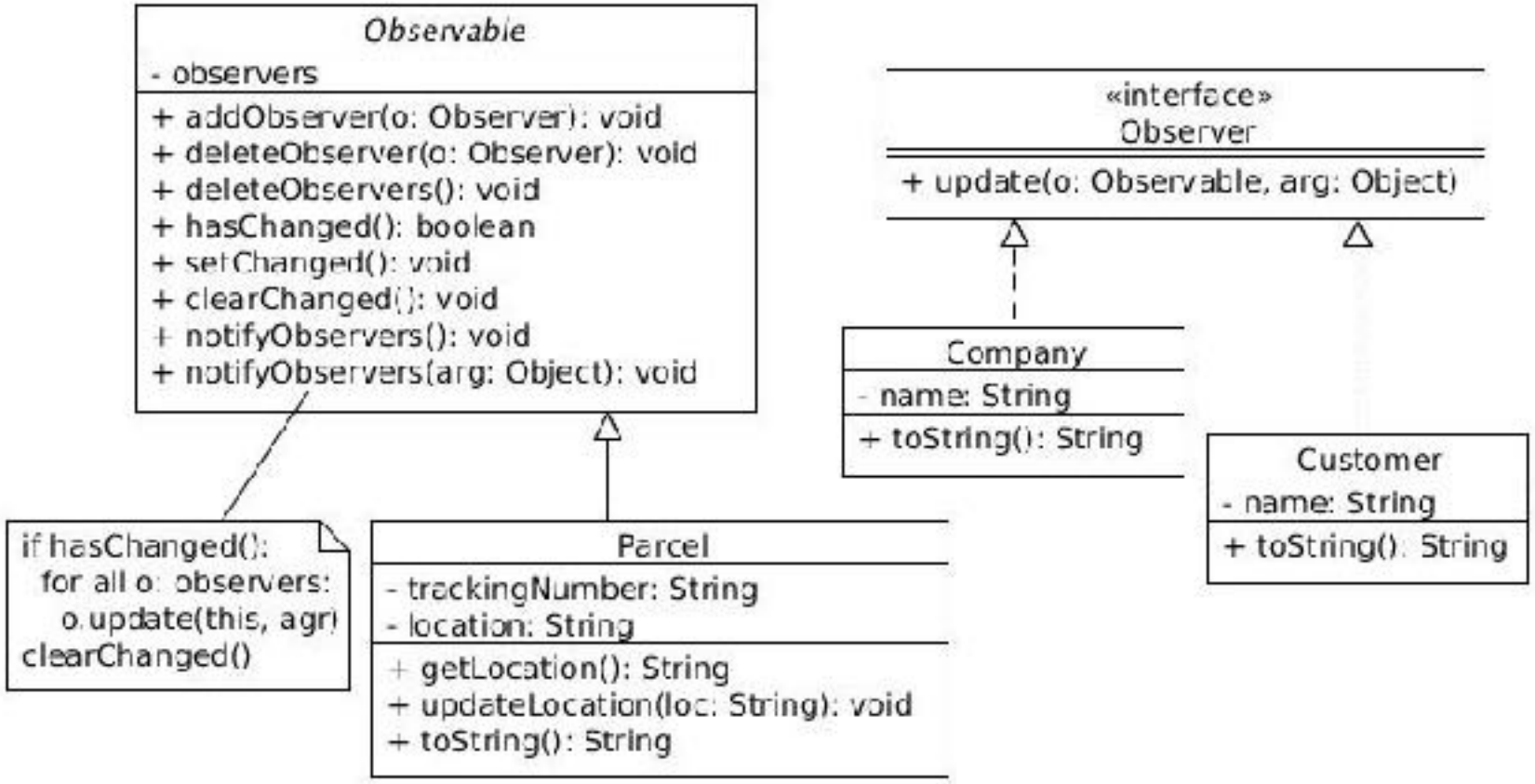
ITERATOR DESIGN PATTERN

- 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.
- Solution
  - To be iterable, a class provides an Iterator class for others to use!
  - The Iterator provides a hasNext() and a next() method to facilitate iteration!

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.
- Solution:
  - Define an interface for what it means to be an Observer (e.g. has an update() method)
  - Allow Observers to watch for changes in a given Observee.
    - When the Observee changes, all Observers have their update() method called.

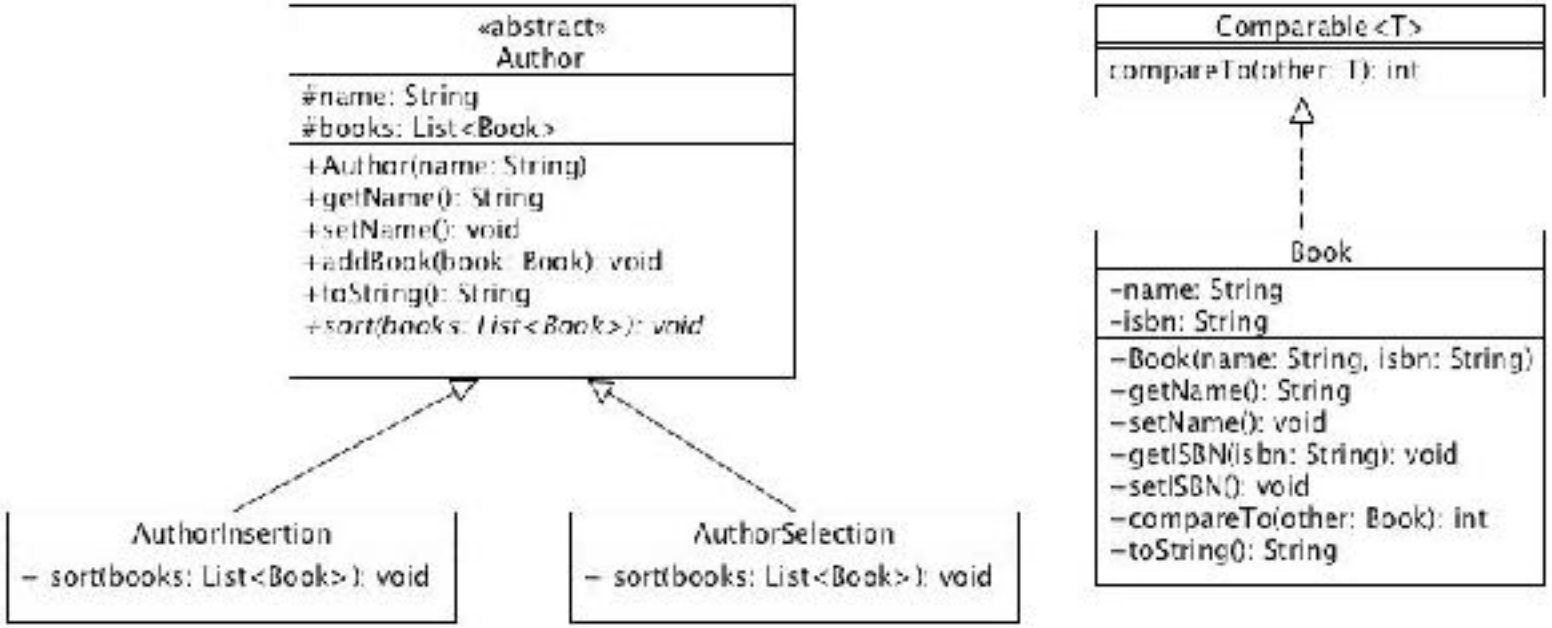
OBSERVER: PARCEL EXAMPLE IN JAVA



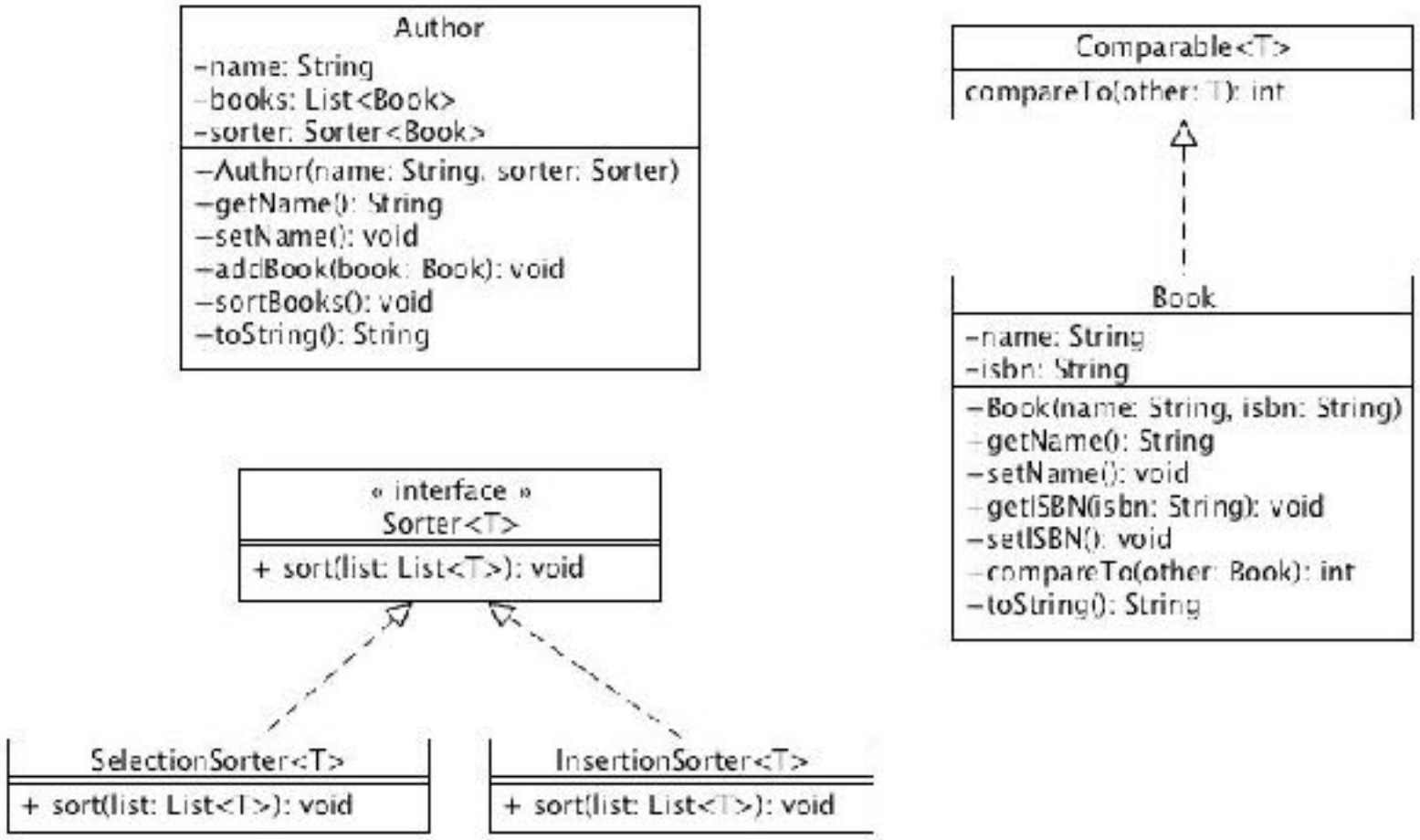
STRATEGY DESIGN PATTERN

- 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
- Solution:
  - Define an interface (Strategy) and pull that out of the class.
  - Pass an **implementation** of the Strategy into the class and **delegate** to it when you need to apply the strategy!
  - Replacing inheritance with delegation!

EXAMPLE: WITHOUT THE STRATEGY PATTERN



EXAMPLE: USING THE STRATEGY PATTERN



“DEPENDENCY INJECTION DESIGN PATTERN”

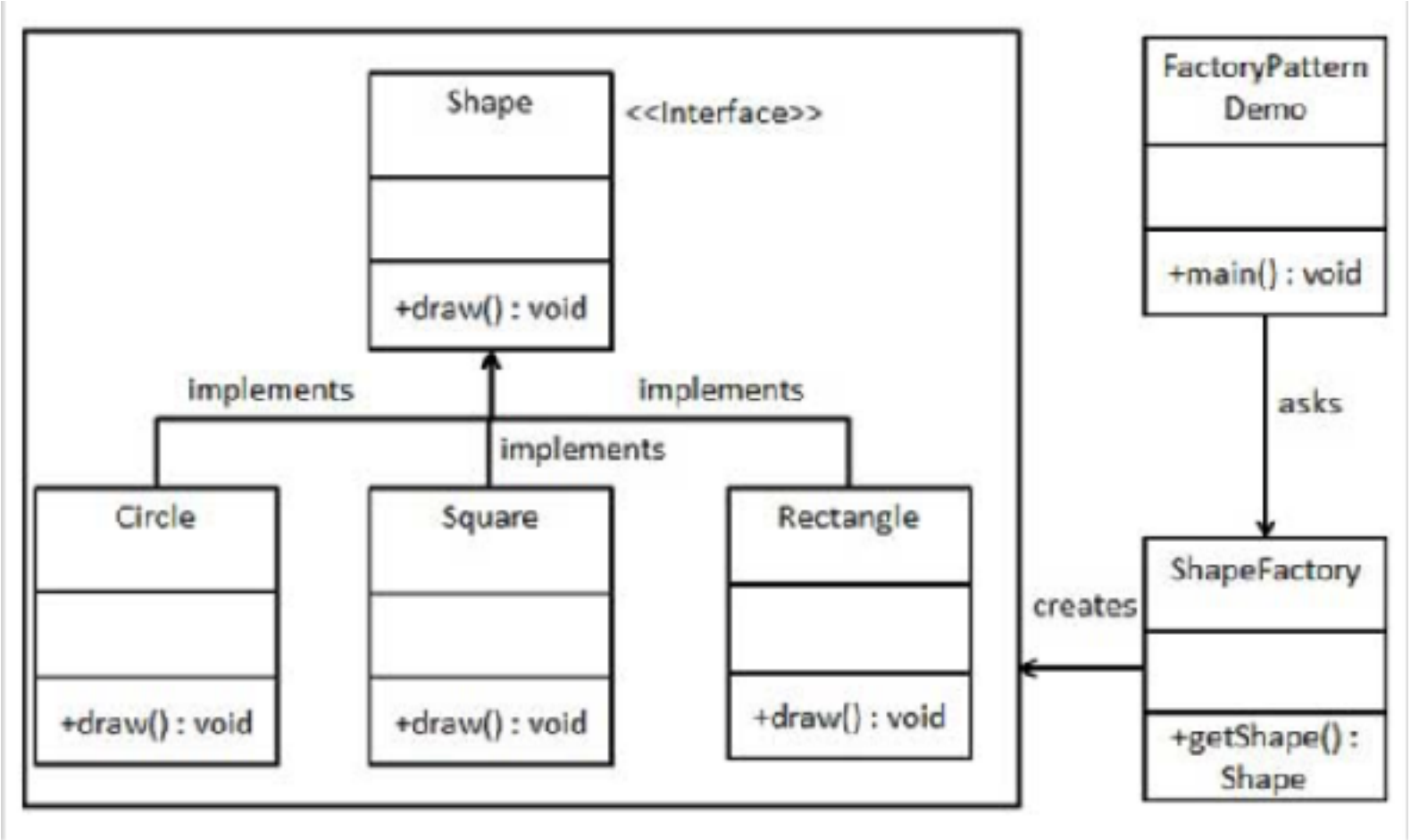
- Problem:
  - we need to assign values to the instance variables of a class
  - we don’t want to hard-code the types of the values.
  - Instead, we want to allow for the use of subclasses.
- Solution:
  - Create the object outside of the class.
  - Inject the dependency into the class.
  - Can inject an instance of the class or any of its subclasses!
- Similar to the Strategy design pattern!
- Used when we invert dependencies in Clean Architecture!
- Exposes class to the outside of the original object.

“SIMPLE FACTORY DESIGN PATTERN”

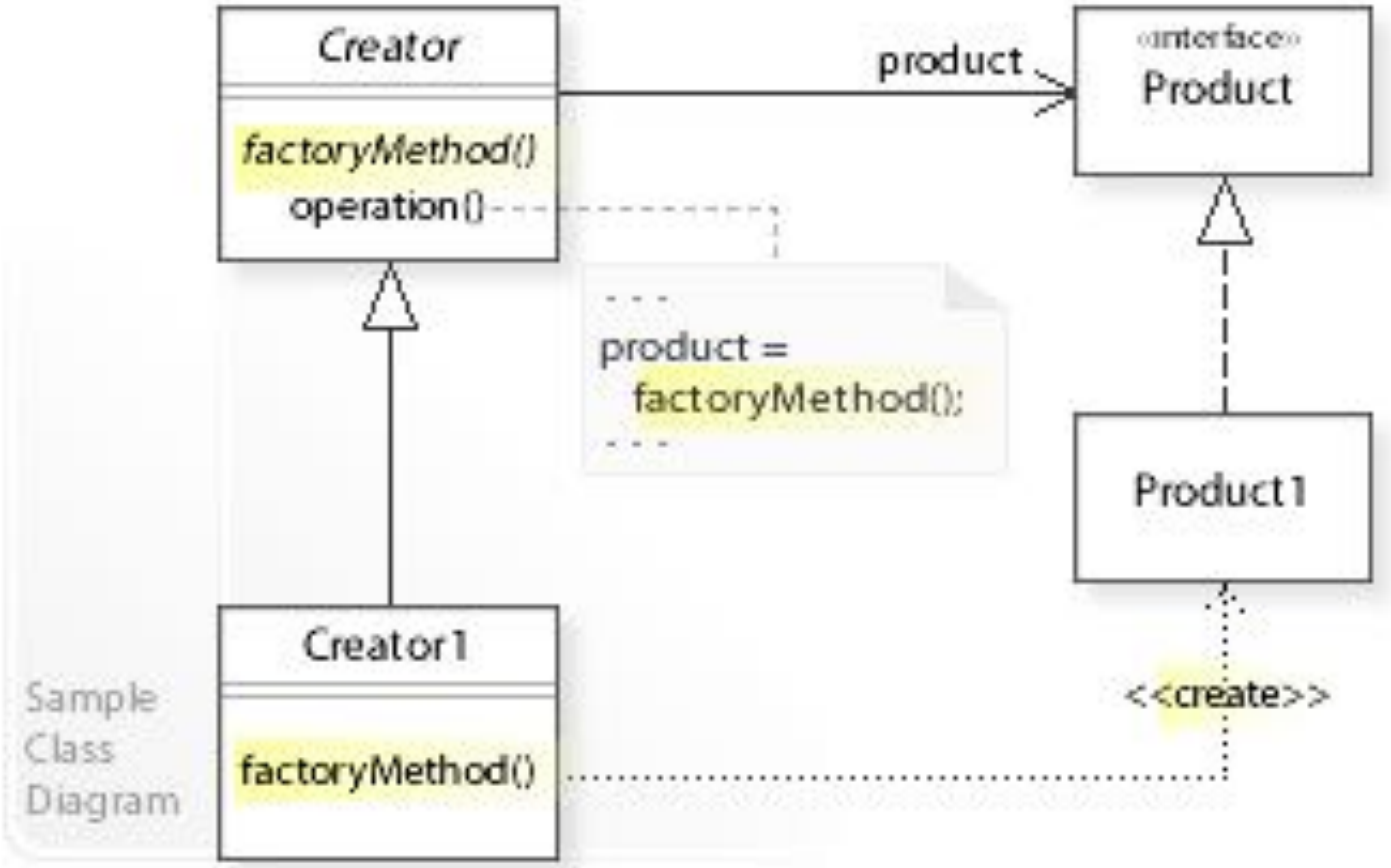
- Problem:
  - One class wants to interact with many possible related objects.
  - We want to obscure the creation process for these related objects.
  - At a later date, we might want to change the types of the objects we are creating.
- Solution:
  - Define a Factory responsible for creating objects.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

- Explanation of the Factory Method design pattern
  - https://en.wikipedia.org/wiki/Factory_method_pattern#Structure
- Another good article discussing the distinction between “Simple Factory” and “Factory Method”
  - https://dzone.com/articles/factory-method-vs-simple-factory-1
- Discussion of using “static factory methods instead of constructors”
  - https://www.drdobbs.com/jvm/creating-and-destroying-java-objects-par/208403883

SIMPLE FACTORY: AN EXAMPLE



FACTORY METHOD : UML



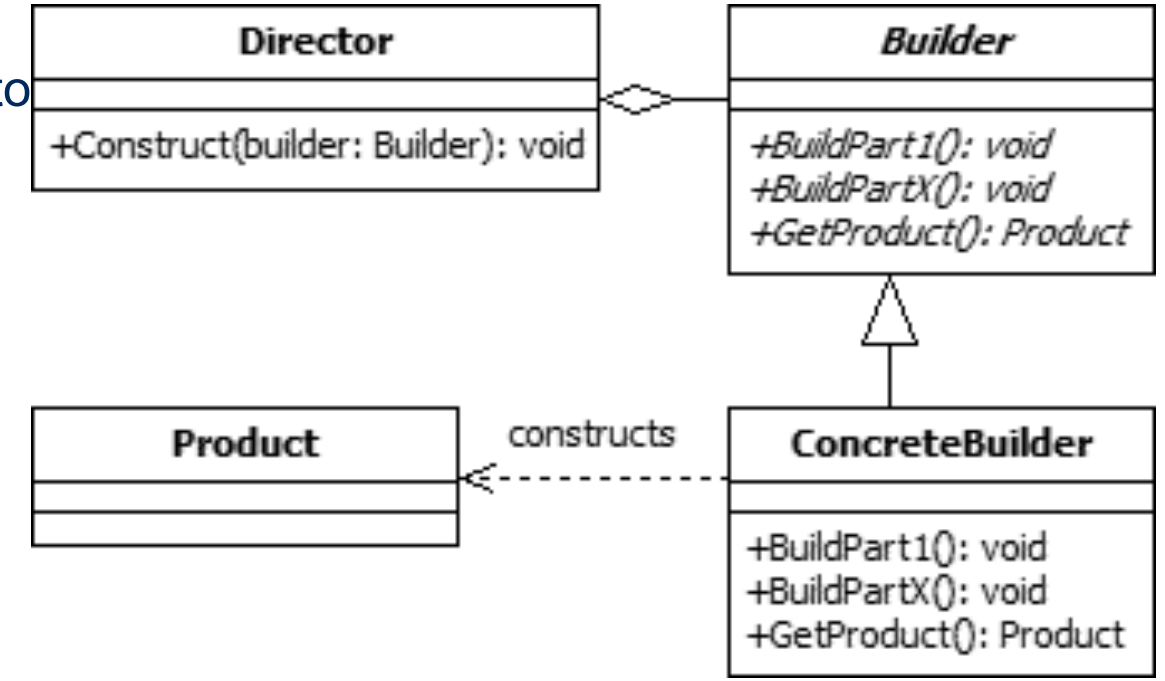
https://commons.wikimedia.org/wiki/File:W3sDesign_Factory_Method_Design_Pattern_UML.jpg

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 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.

BUILDER DESIGN PATTERN

- Problem:
  - Need to create a complex structure of objects in a step-by-step fashion.
- Solution:
  - Define a Builder class that can defines the steps to build the complex structure.
  - Client code or a Director class uses the Builder to apply the required steps.
- If you go to the Wikipdia page for Builder, it links to
  - https://en.wikipedia.org/wiki/Currying



HOMEWORK

- Keep working on Phase 1 — applying what you have been learning about design patterns, code smells, and refactoring!

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 Regex or regular expression
→ string matches the regex, not the other way around.

- 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.

(not the other way around)

^[a-z][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.

^[a-z][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`

uppercase first → number first character

only numbers & alphabets allowed

✓ ✓

↳ can't start with number

WHAT IF THERE ARE NO ANCHORS?

↳ no ^ and $ (carrot) (dollar)

- Here is another regular expression:
  - `[abc]C[a-e][24680]*[A-Z]`

If there is only carrot → substring has to start at the beginning. If you propose a dollar, then match has to happen at the end of the string (whole string matches)

- 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`.

→ contains odd number which isn't in the the regex

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://regexcrossword.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

- Weekly questions about regex
- Try the Regex crossword website

Regex Checker

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexChecker {

    public static void main(String[] args) {

        //match one String against one regex
        System.out.println(Pattern.matches("([.w]){2}f", "..f"));
        System.out.println(Pattern.matches("[k*]a*b", "kaaaaabx"));

        //using a Matcher object to find matching substrings
        Pattern pattern = Pattern.compile("[a-c]\\+{5}\""); // \\
+{5} becomes +{5} in the regex
        Matcher matcher = pattern.matcher("a1bb\\\\c2a4\\\\b+++++
\"6c89\\\\c+++++\"");
        while (matcher.find()) {
            System.out.println(matcher.group());
        }

        //using the String method "matches" to check if a String
matches a regex
        String str = "Hello_World11";
        boolean isSame = str.matches("(Hello)\\w.*(1)\\2");
        System.out.println(isSame);
    }
}
```

REGEX MATCHER

```
public static void main(String[] args) {

    // You can do an individual match in one easy line:
    System.out.println(Pattern.matches("a*b", "aaaaab"));

    // Notice that it automatically anchors
    // That is, it is equivalent to ^a*b$
    System.out.println(Pattern.matches("a*b", "baaaaab"));
    System.out.println();

    // If you never reuse the same pattern, this is fine.
    // As in this method:
```

```
        doMatching();

        // But if you plan to reuse a pattern, it's more efficient
        // to let Java build the matching infrastructure once and
        // reuse it for each match against that pattern.
        Pattern p = Pattern.compile("CSC[0-9][0-9][0-9]H1([FS])");
        Matcher m = p.matcher("CSC207H1S");
        System.out.println("Does CSC207H1S match " + p + " ?");
        System.out.println(m.matches());

        // Here we reuse that (under the hood) infrastructure.
        System.out.println("Does CSC199H1Y match " + p + " ?");
        System.out.println(p.matcher("CSC199H1Y").matches() + "\n");

        // The matcher has other methods that let you find out
        // which substrings matched with which "capturing group"
        // of the pattern. Each capturing groups begins with a
        // left bracket. The capturing groups are numbered from 0,
        // and group 0 is the whole pattern.

        // Here we add more brackets to the pattern.
        // This will allow us to capture the group of characters
        // that is the course number.
        // (Exercise: rewrite the pattern to be more concise)
        p = Pattern.compile("CSC([0-9][0-9][0-9])H1([FS])");
        m = p.matcher("CSC207H1S");
        if (m.matches()) { // need to run the matches method before
calling the group method!
            System.out.println("Captured groups for pattern
CSC([0-9][0-9][0-9])H1([FS]) and input CSC207H1S");
            System.out.println("group 0: " + m.group(0));  // the
entire string
            System.out.println("group 1: " + m.group(1));  // the
first group: 207
            System.out.println("group 2: " + m.group(2));  // the
second group: S
        }
        else {
            System.out.println("group example failed");
        }
```

```
        // Example of using a back reference.
        p = Pattern.compile("(\\d\\d\\d)ABC\\1");
        m = p.matcher("123ABC123");
        System.out.println("Example using a back reference to match
group later");
        System.out.println("123ABC123 matches " + p.pattern() +
" ...");
        System.out.println(m.matches());
        m = p.matcher("123ABC456");
        System.out.println("123ABC456 matches " + p.pattern() +
" ...");
        System.out.println(m.matches());
    }

}
```

Embedded Ethics Module I

4. What are the two models of disability that were discussed? Briefly explain both models.

5. What are the seven principles of universal design?

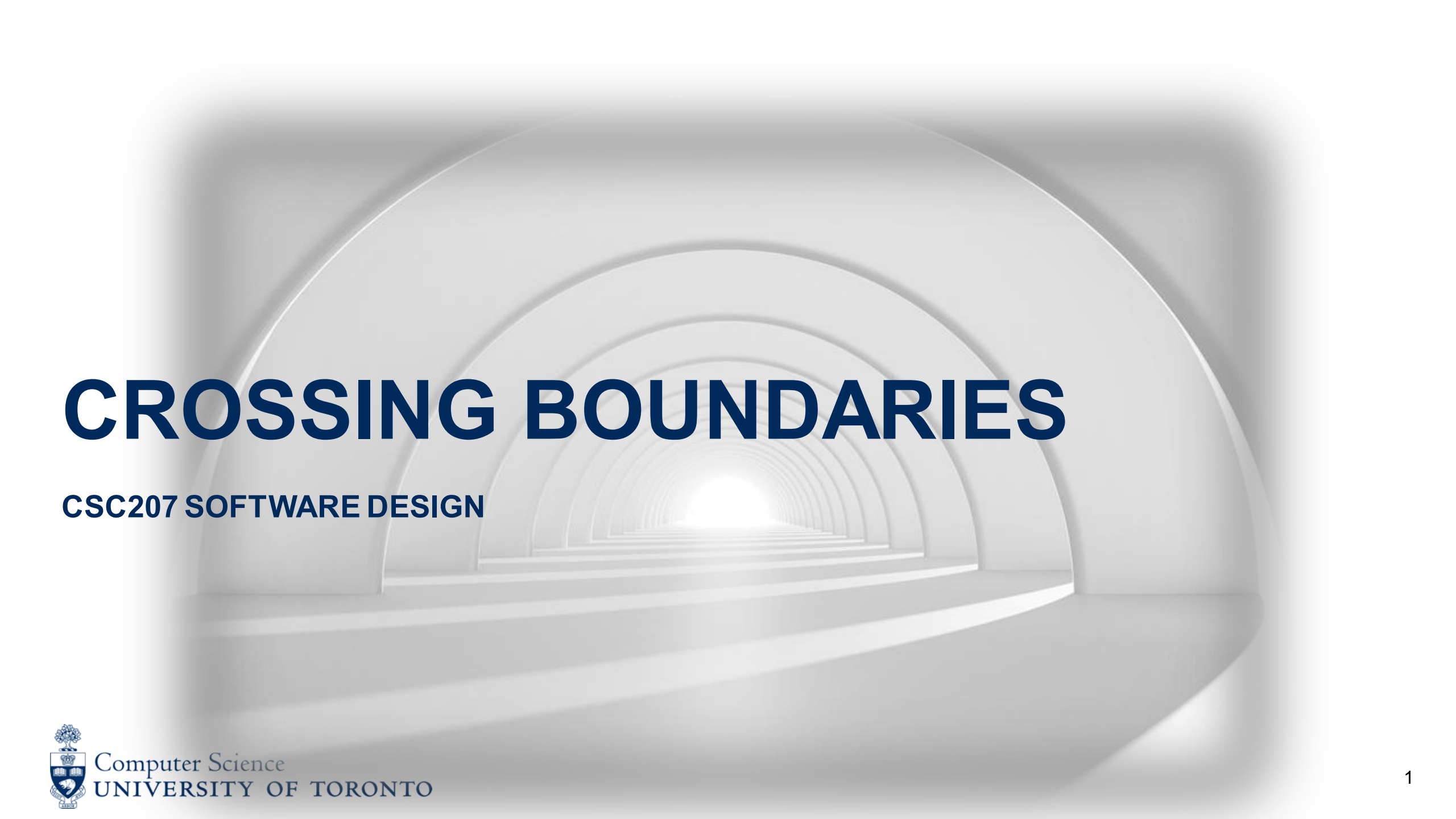
Embedded Ethics Module II

5. What is the explanation for why we should accommodate people with accessibility needs?

6. Briefly, what are Utilitarianism and the Capability Approach?

7. Give an example of a legal requirement related to accessibility.

8. Why might following Clean Architecture make it easier for software developers to implement accessibility features?



CROSSING BOUNDARIES

CSC207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

LEARNING OUTCOMES

- Improve our understanding of architectural boundaries and Clean Architecture.

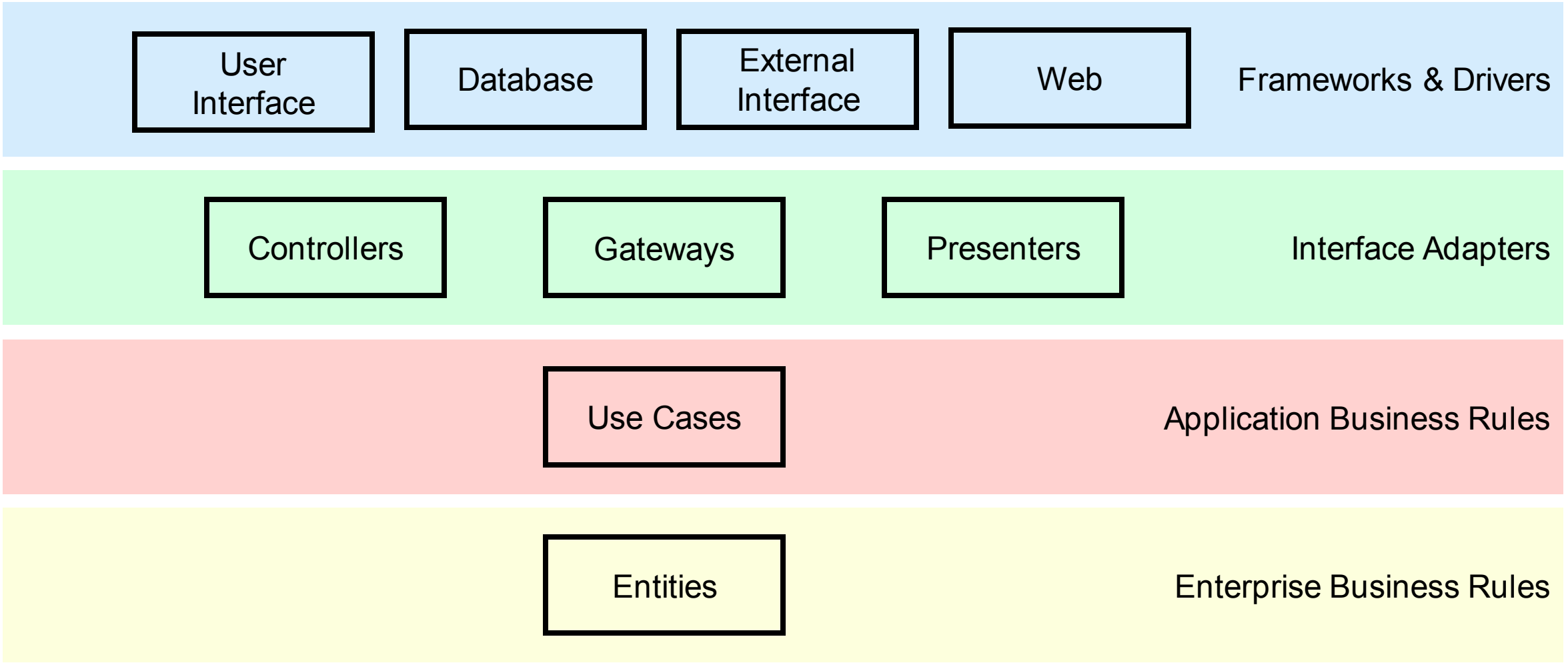
ARCHITECTURE (RECAP)

(Brief Summary of Chapter 15 from Clean Architecture textbook)

Design of the system

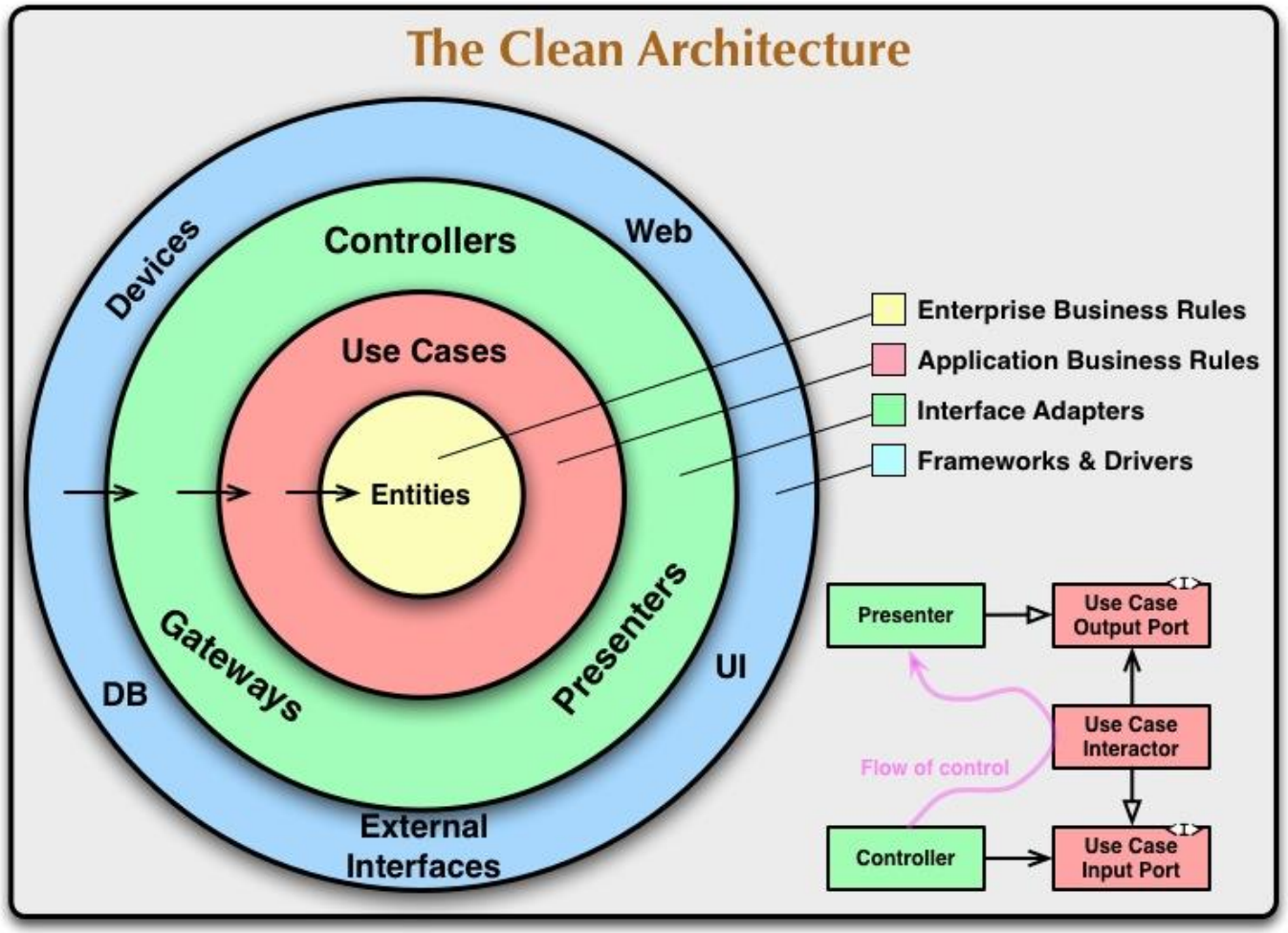
- Dividing it into logical pieces and specifying how those pieces communicate with each other.
- Input and output between layers.
- This amounts to identifying where boundaries should exist in your program and what data should cross those boundaries.

CLEAN ARCHITECTURE (RECAP)



CLEAN ARCHITECTURE — FLOW OF CONTROL

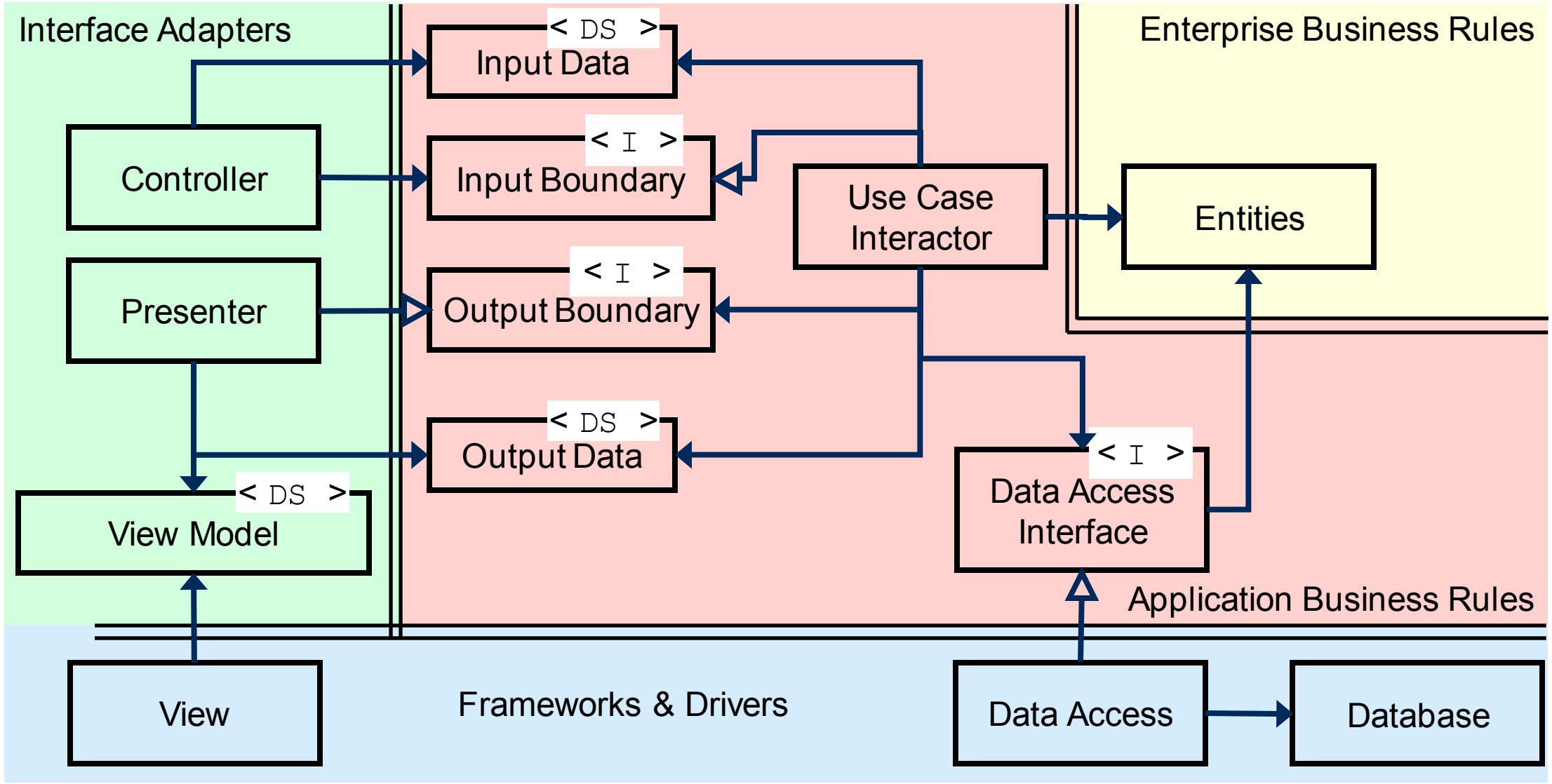
- Note how the flow of control goes from Controller to Use Case to Presenter, with Dependency Inversion used to satisfy the Dependency Rule.
- There is some good discussion of this on piazza and online too, for example.



CLEAN ARCHITECTURE — DEPENDENCY RULE (RECAP)

- 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 PROGRAM WITH CLEAN ARCHITECTURE (RECAP)



BENEFITS OF CLEAN ARCHITECTURE (RECAP)

- 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!

BOUNDARIES

(Summary of Ch 17 Boundaries: Drawing Lines)

- “Software architecture is the art of drawing lines that I call boundaries. Those boundaries separate software elements from one another, and restrict those on one side from knowing about those on the other. Some of those lines are drawn very early in a project’s life—even before any code is written. Others are drawn much later. Those that are drawn early are drawn for the purposes of deferring decisions for as long as possible, and of keeping those decisions from polluting the core business logic.”
- “You draw lines between things that matter and things that don’t. The GUI doesn’t matter to the business rules, so there should be a line between them. The database doesn’t matter to the GUI, so there should be a line between them. The database doesn’t matter to the business rules, so there should be a line between them.”
- This motivated the layers in Clean Architecture!
- “The SRP tells us where to draw our boundaries.”

BOUNDARIES

(Summary of Ch 17 Boundaries: Drawing Lines)

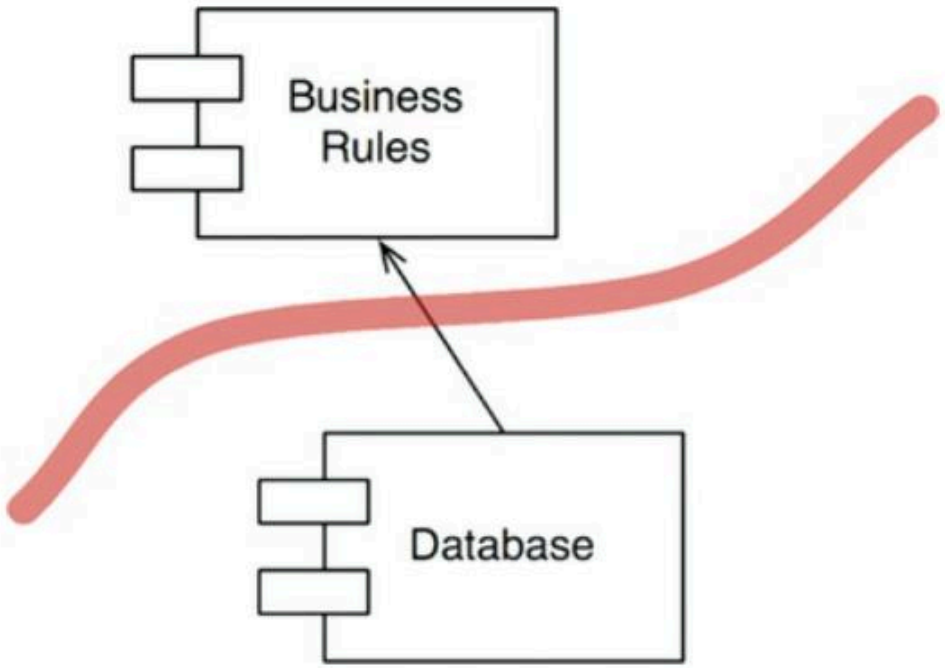


Figure 17.3 The business rules and database components

BOUNDARIES

(Very Brief Summary of Ch 18 Boundary Anatomy)

- “At runtime, a boundary crossing is nothing more than a function on one side of the boundary calling a function on the other side and passing along some data. The trick to creating an appropriate boundary crossing is to manage the source code dependencies.”
- As we’ve discussed, dependency inversion can be used to ensure the source code dependency is in the direction we want.

ADVICE ABOUT BOUNDARIES

(Brief summary of conclusions from Ch. 25 Layers and Boundaries)

- "We, as architects, must be careful to recognize when they are needed. We also have to be aware that such boundaries, when fully implemented, are expensive. At the same time, we have to recognize that when such boundaries are ignored, they are very expensive to add in later—even in the presence of comprehensive test-suites and refactoring discipline."
- "But this is not a one-time decision. You don't simply decide at the start of a project which boundaries to implement and which to ignore. Rather, you watch. You pay attention as the system evolves. You note where boundaries may be required, and then carefully watch for the first inkling of friction because those boundaries don't exist."

WAREHOUSE BOUNDARY ACTIVITY

- See the warehouse boundary activity page on Quercus
- Mentimeter
  - https://www.menti.com/tu9oze29oa
  - (or code 6699 8560 at https://menti.com)

HOMEWORK

- Weekly questions about Clean Architecture and boundaries
- Look at your project code and identify places where you have closely followed Clean Architecture and places where your code needs refactoring!
  - Identifying boundaries may help you gain a new perspective on your design!
- Optionally read the textbook chapters mentioned here or this post summarizing much of what the textbook says about boundaries and architecture

Week 10

Interviewing

1. What are the five steps to the interviewing process discussed in the slides?

2. What are three resources that can help you prepare for technical interviews?

Floating Point

3. What are the various parts of a floating point number? Name each part and briefly describe it.

4. What is the main difference between double and single precision?

5. Of the algorithms for summing n numbers, which do you think is the best to use in most cases? Said another way, which algorithm do you think the sum function would use in a programming language, such as Python? Briefly explain.