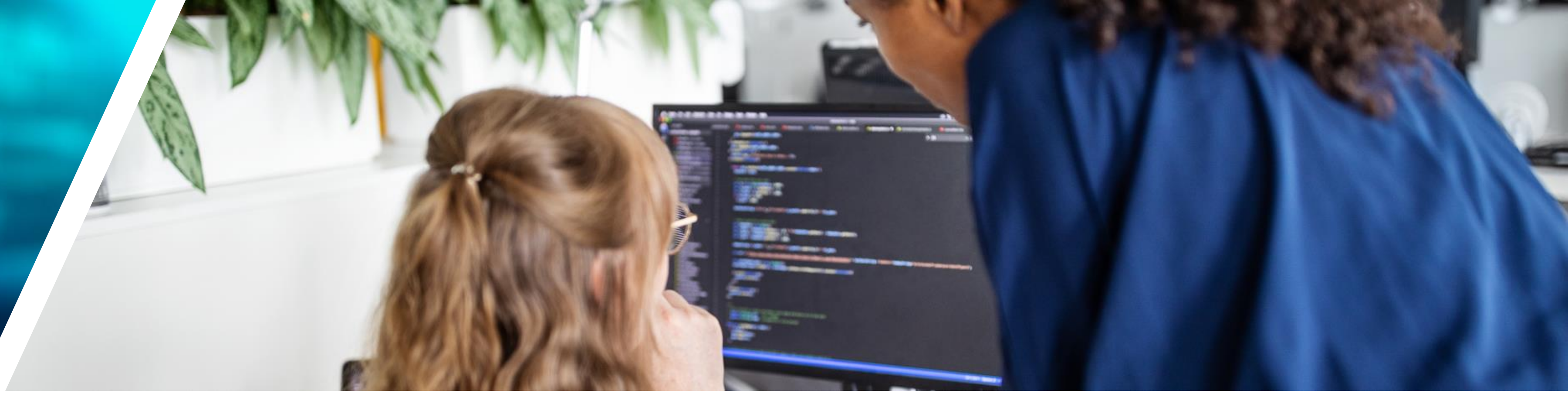


STUDY QUESTIONS

CSC 207 SOFTWARE DESIGN

Java

JAVA

TO STUDY: BASIC FEATURES OF A JAVA CLASS, CONTENTS OF THE QUIZZES, LABS

1. What does a constructor do?
2. Do you have to include a constructor in every class?
3. Does the constructor have to call `super`?
4. What are the "java lookup rules"?
5. What makes a class generic? Abstract? When should you use generics? When should you use abstract classes and methods?
6. How can you choose between interfaces and abstract classes.

NOTABLE ASPECTS OF JAVA

- types
- primitives
- casting
- generics
- exceptions
- Interfaces
- overloading
- memory model
- junit
- javadoc
- constructors
- access modifiers
- packages

GIT

TO STUDY: BASIC COMMANDS, RESOLVING CONFLICTS, BRANCH, MERGE

1. What are some benefits of using version control?
2. What are some best practices that you have noticed when using git?
3. What makes a commit message "good"?
4. What is a branch and how can you use it to benefit your group's project?

Testing

TESTING

TO STUDY: TESTING-RELATED DEFINITIONS, BASICS OF JUNIT

1. What is the difference between "end to end", "integration", and "unit" tests?
2. What does it mean to "Mock" class `A` when testing class `B`?

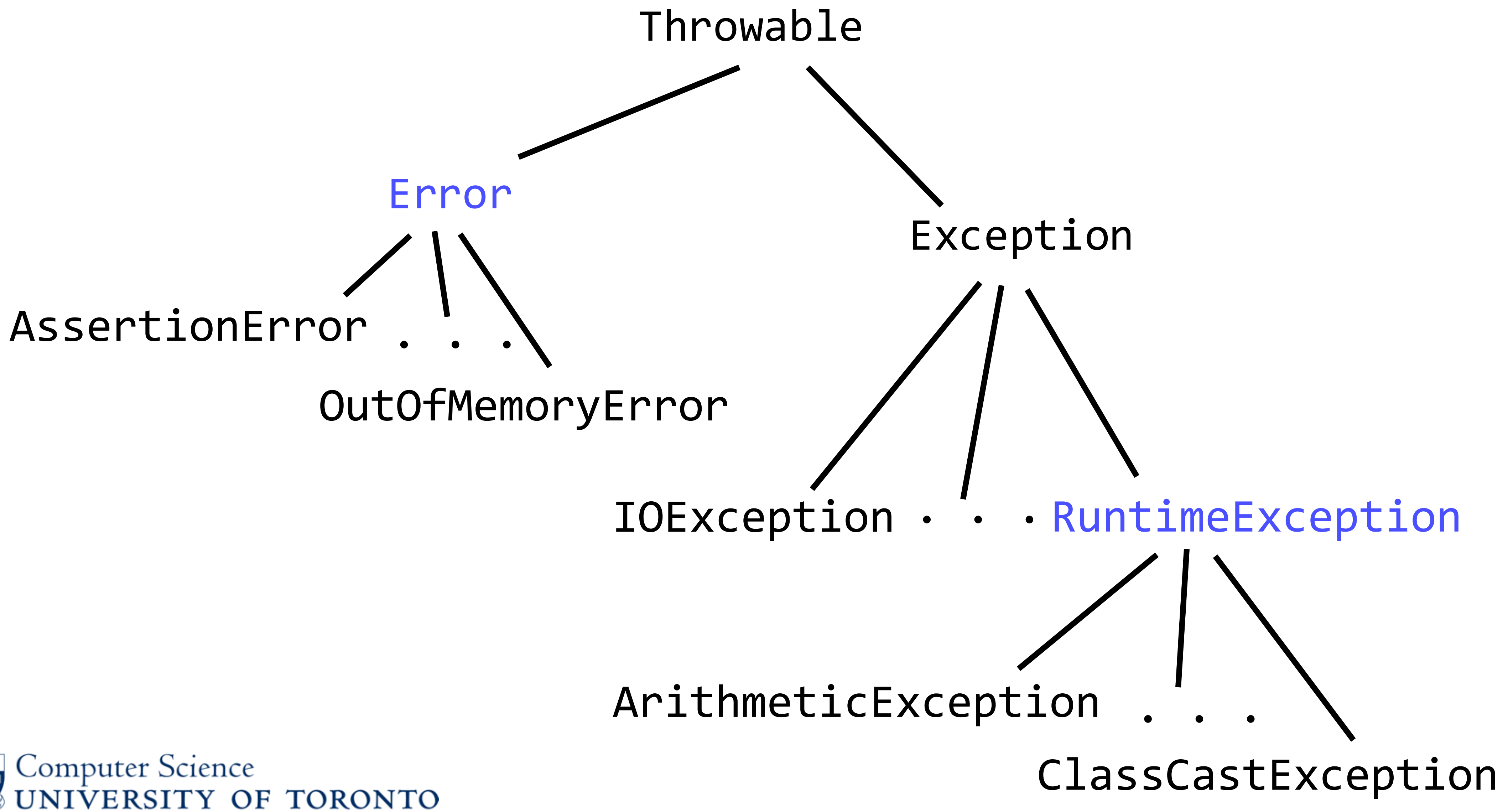
Exceptions

EXCEPTIONS

TO STUDY: WHEN TO THROW, SYNTAX, INHERITANCE HIERARCHY, CHECKED VS UNCHECKED

1. What syntax makes an Exception checked? Unchecked?
2. What is a RuntimeException and why do we not catch it?
3. Are the try/catch blocks and the word "throws" generally located in the same class or different classes?
4. Why do we not catch an object of type "Exception"?

THE THROWABLE HIERARCHY



EXCEPTIONS IN JAVA

- To “throw an exception”:

```
throw new Throwable(message);
```

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

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

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

  accessModifier returnType methodName(*parameters*) **throws *Throwable*** { ... }

WE CAN HAVE CASCADING 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 is 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.
}
```

Computer Science
UNIVERSITY OF TORONTO

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!"); ...
  }
}
```

CHECKED VS UNCHECKED EXCEPTIONS

- Checked Exceptions:
    - A method **must** include "throws ExceptionClassName" in its declaration
    - The compiler will check to make sure that the Exception is caught by the method that calls it, or the method that calls that method, or...
- Unchecked Exception:
    - The program will throw these without checking to see if it is eventually caught
    - Will crash the program and print the stack trace to the terminal
    - You can then see where the Exception was thrown and fix the code so that does not happen again (helpful when debugging!)

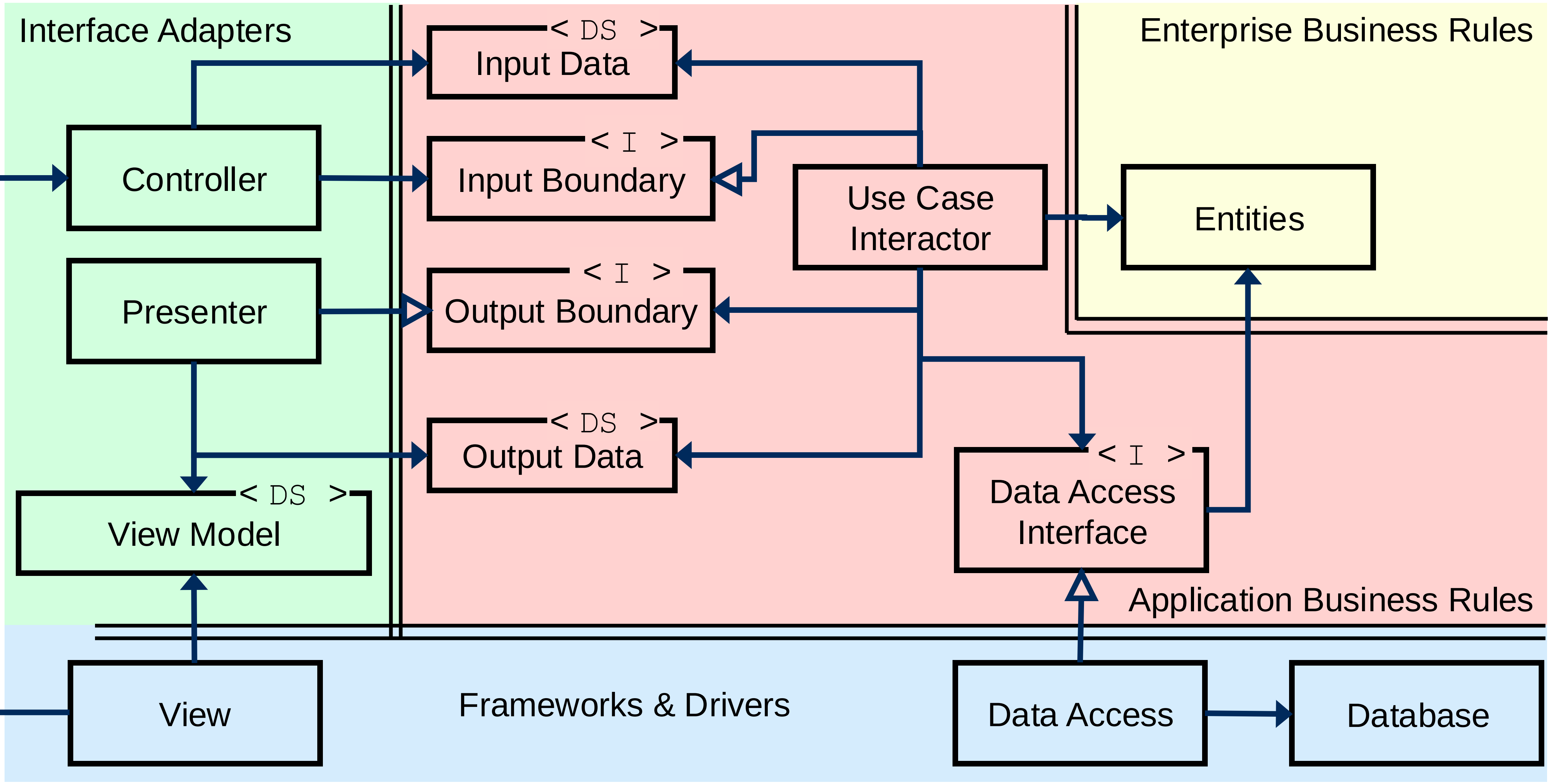
Clean Architecture

CLEAN ARCHITECTURE

TO STUDY: DEPENDENCY RULE, ROLES OF EACH CLASS, 3 DIAGRAMS

1. What does a Presenter do?
2. What is the role of the View Model?
3. What is the Dependency Rules in Clean Architecture?
4. Which of the interfaces is not technically required by the Dependency Rule?
5. What does it mean to invert a dependency? Why do would we do that?
6. What is the benefit of deciding on your interfaces before each group member starts writing their own code.

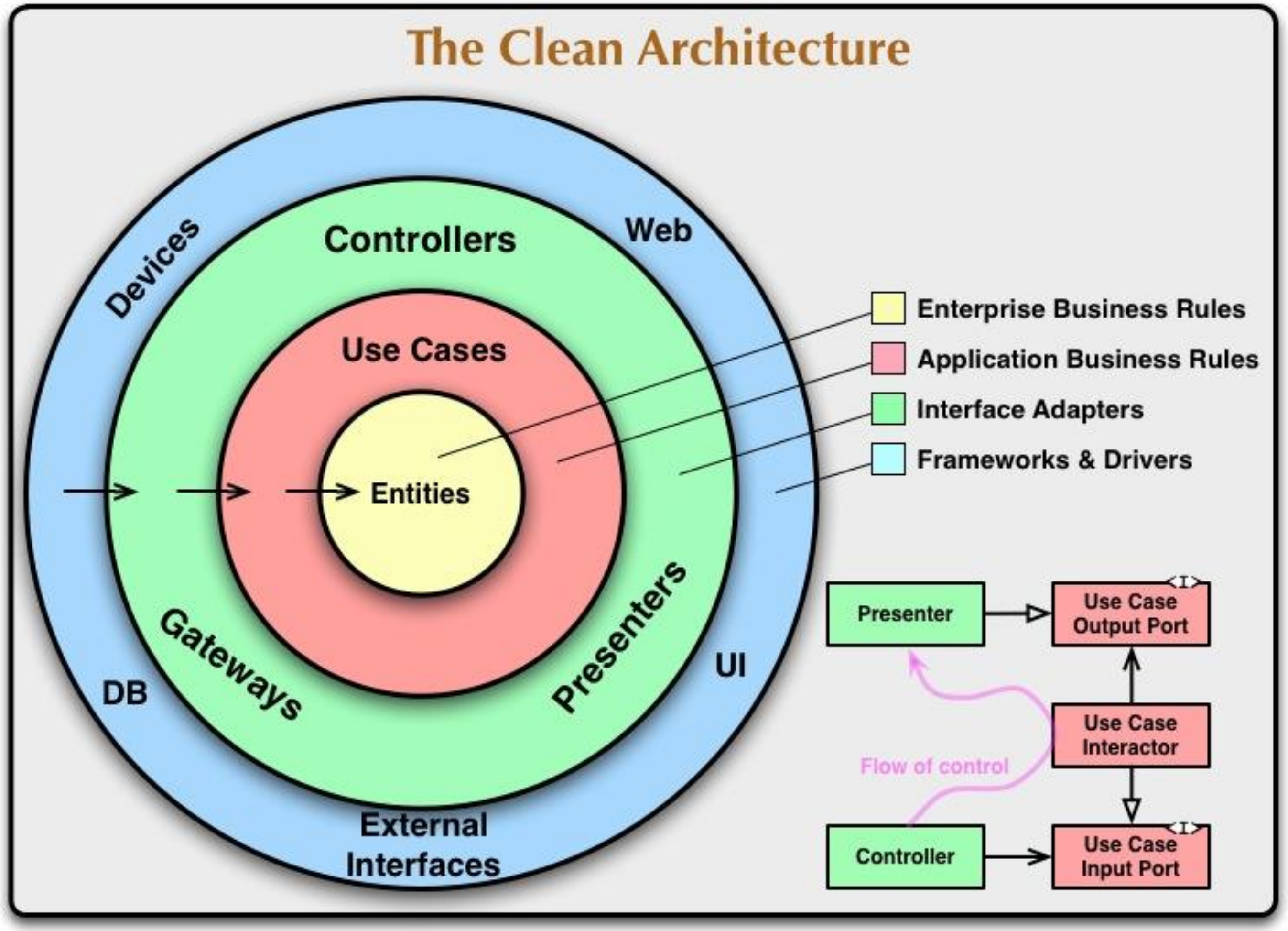
EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Computer Science
UNIVERSITY OF TORONTO

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



SOLID

SOLID

TO STUDY: MEANING AND APPLICATION OF EACH PRINCIPLE

1. Name and summarize each SOLID principle in 2 sentences or fewer.
2. What do we look for when checking for SPR violations, Liskov violations, etc.?
3. Is the Dependency Inversion Principle Related to the Dependency Rule in Clean Architecture? If so, how?

SOLID (PRINCIPLES OF CLASS DESIGN)

SRP	The Single Responsibility Principle	A class should have one, and only one, reason to change.
OCP	The Open Closed Principle	You should be able to extend the behaviour of a class without modifying the class.
LSP	The Liskov Substitution Principle	Derived classes must be substitutable for their base classes.
ISP	The Interface Segregation Principle	Make fine grained interfaces that are client specific.
DIP	The Dependency Inversion Principle	Depend on abstractions, not on concretions.

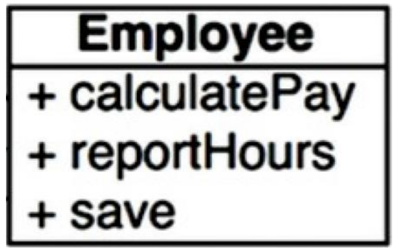
Computer Science
UNIVERSITY OF TORONTO

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

SRP

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)

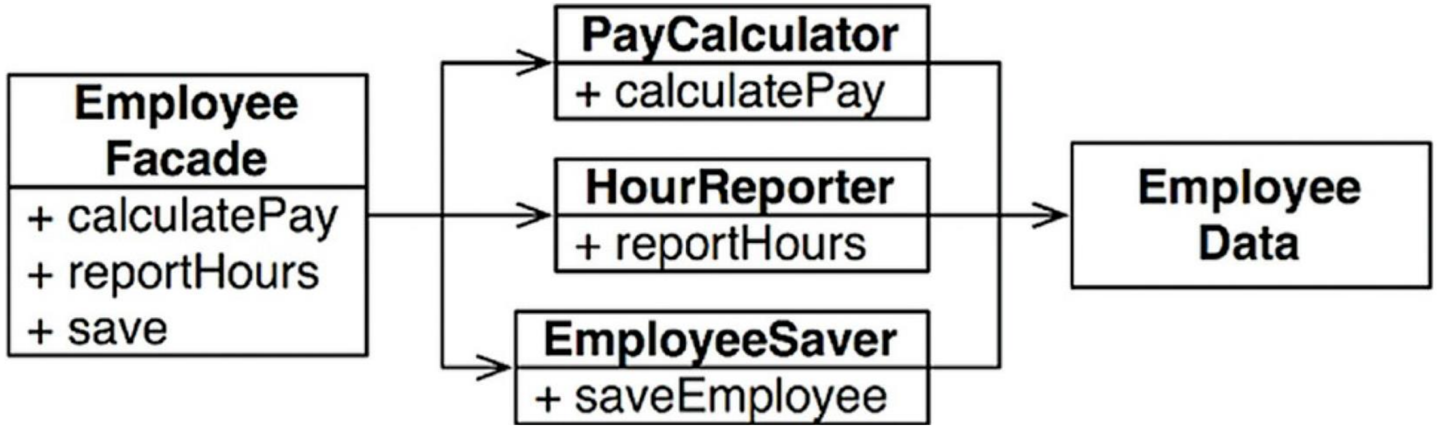


- Suppose methods `calculatePay` and `reportHours` share a helper method to calculate `regularHours` (and avoid duplicate code).
- CFO decides to change how non-overtime hours are calculated and a developer makes the change.
- The COO doesn't know about this. What happens?

Computer Science
UNIVERSITY OF TORONTO

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.

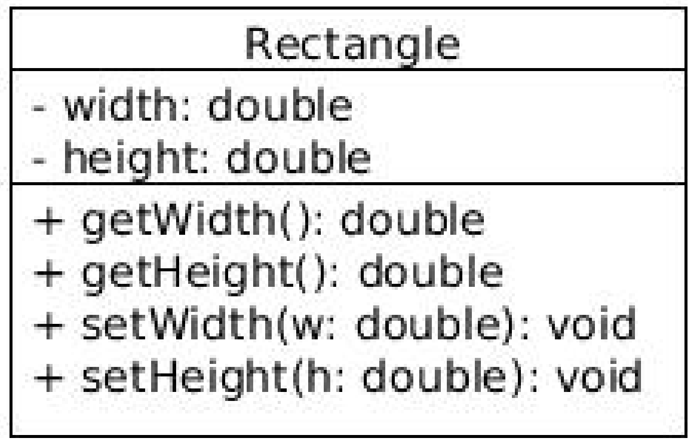


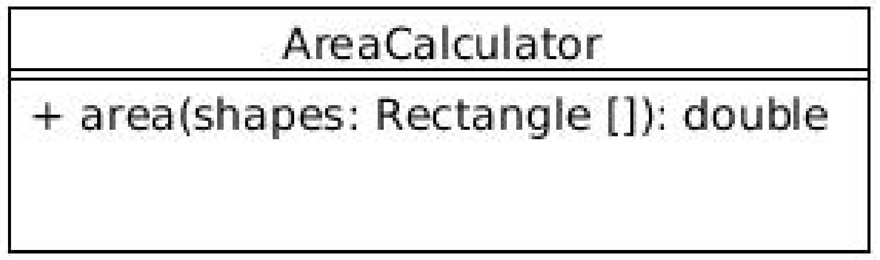
- We'll talk about the Façade design pattern and many more design patterns throughout the term.

OCP

OPEN/CLOSED PRINCIPLE

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



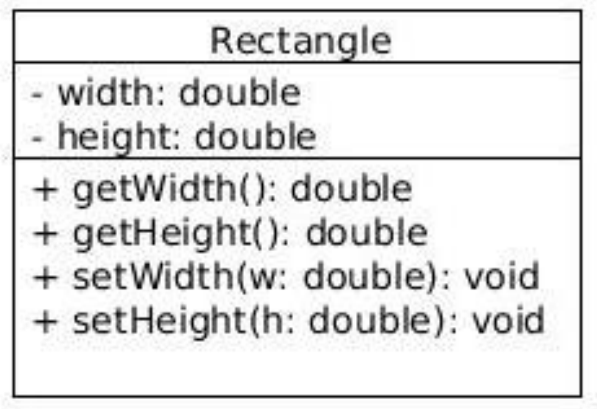


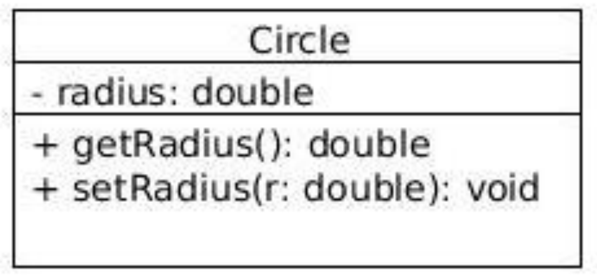
Computer Science
UNIVERSITY OF TORONTO

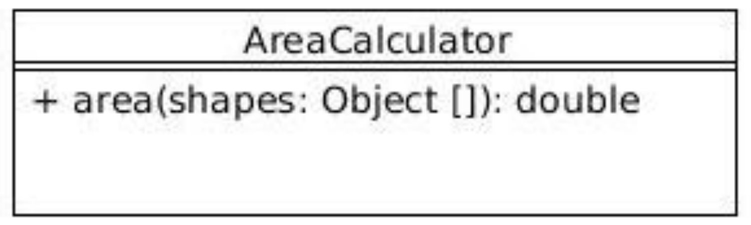
OCP

OPEN/CLOSED PRINCIPLE

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







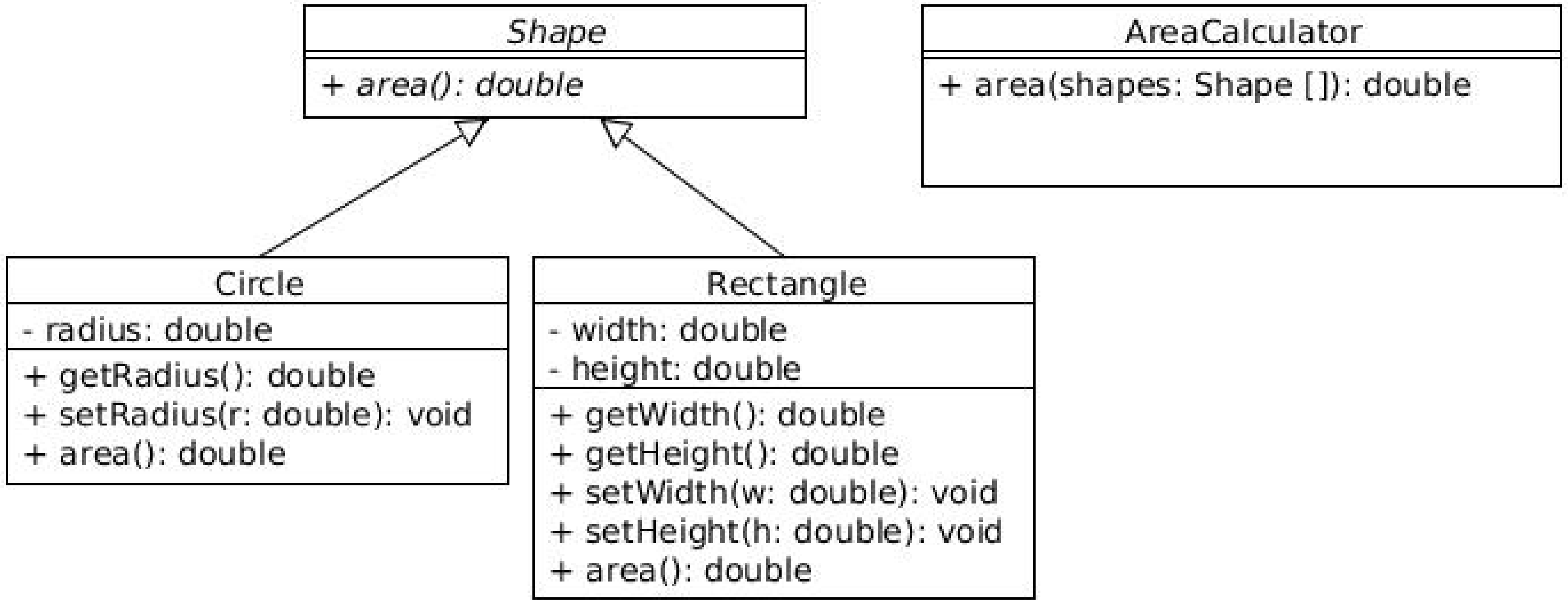
- But what if we need to add even more shapes?

Computer Science
UNIVERSITY OF TORONTO

OCP

OPEN/CLOSED PRINCIPLE

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

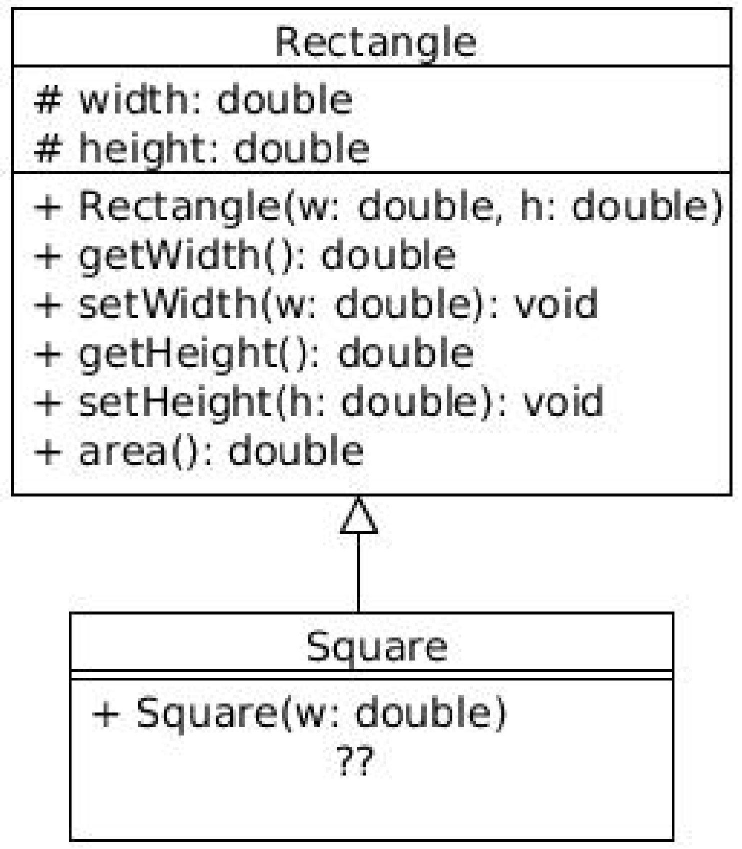


Computer Science
UNIVERSITY OF TORONTO

LISKOV SUBSTITUTION PRINCIPLE

Example

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



INTERFACE SEGREGATION PRINCIPLE

ISP

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

Computer Science
UNIVERSITY OF TORONTO

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.

Design Patterns

DESIGN PATTERNS:

TO STUDY: SIX PATTERNS, HOW TO IMPLEMENT THEM, PROBLEM THAT EACH SOLVES

1. Which six design patterns did we study in this course and how do they work?
2. What is a Dependency? How does Dependency Injection "lessen" a dependency between two classes?
3. What is the difference between a factory and a builder?
4. Of the ones we covered, which two design patterns allow you to choose between inheritance and composition? (See the next slide for a hint)
5. Does the Strategy pattern require an interface or can you implement it without one?

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”**.
- 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

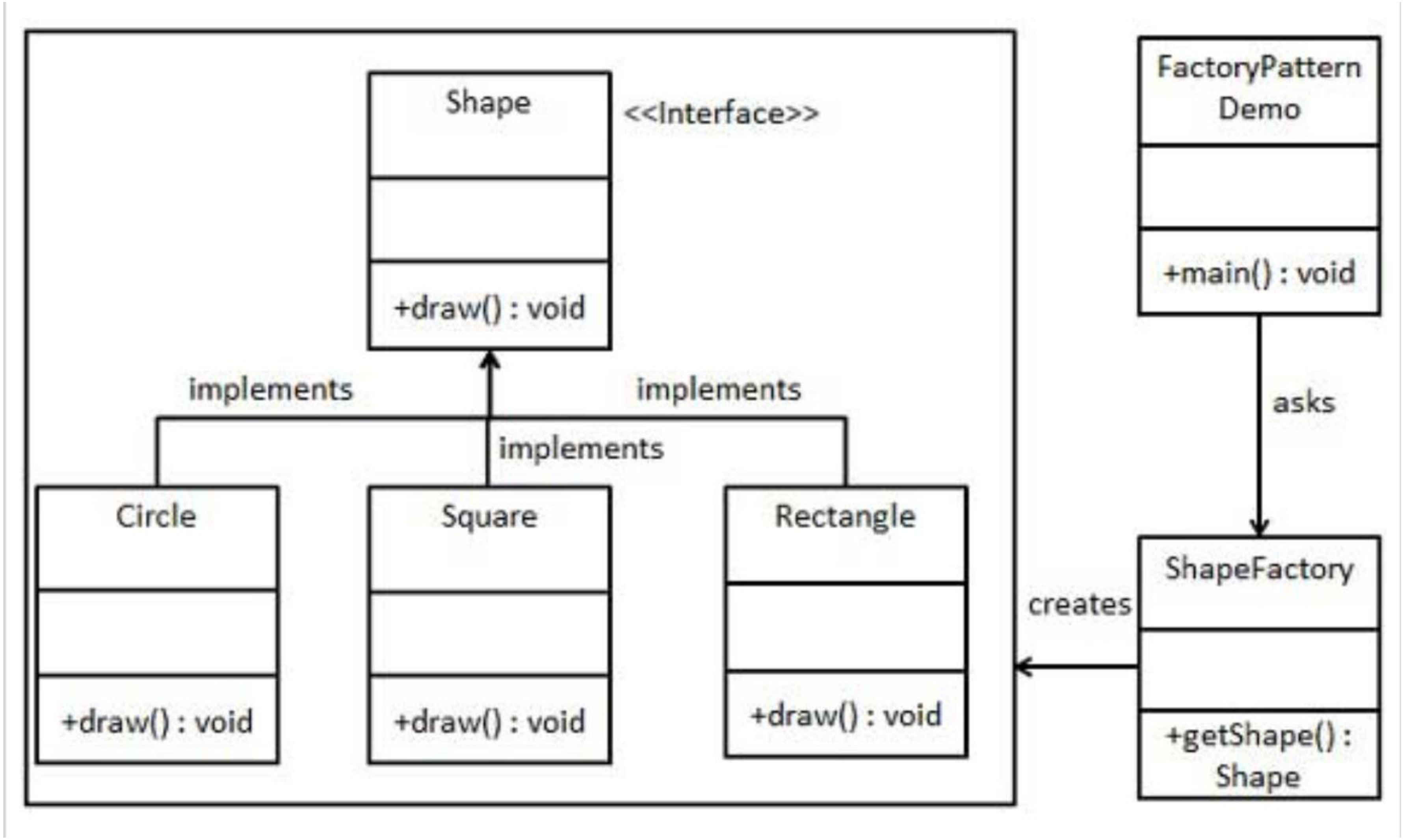
- The solution: create the `Student` objects *outside* and *inject* them into `Course`, which means pass as a parameter to a constructor or a setter or adder. This allows subclasses of `Student` to be injected into `Course`.

```
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);
    }
}
```

Bonus: Is there still another hard dependency in the code?

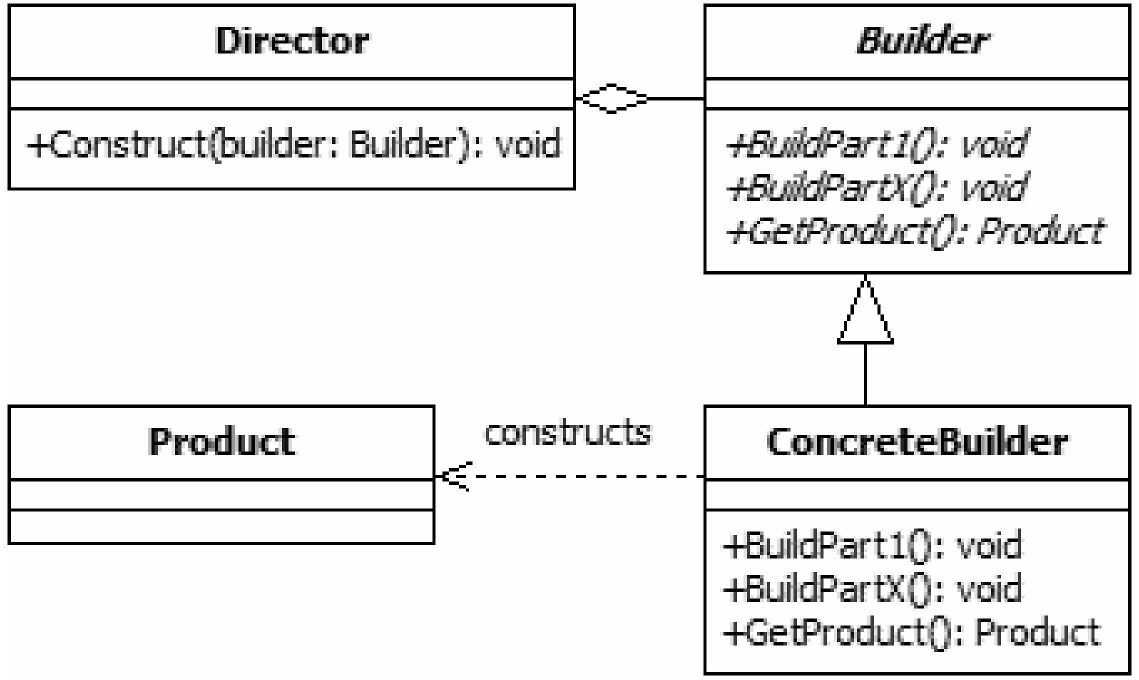
FACTORY : AN EXAMPLE



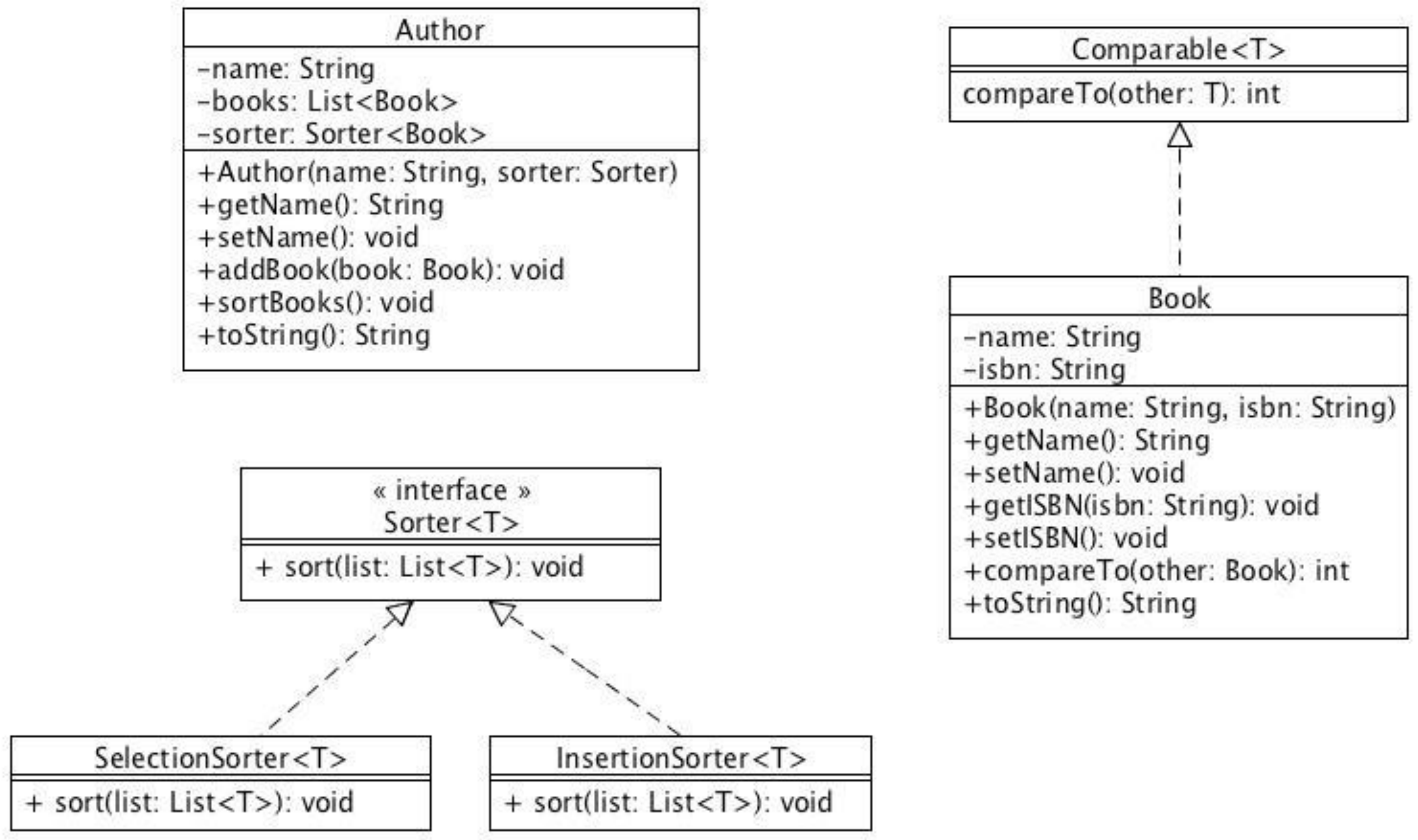
Computer Science
UNIVERSITY OF TORONTO

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.

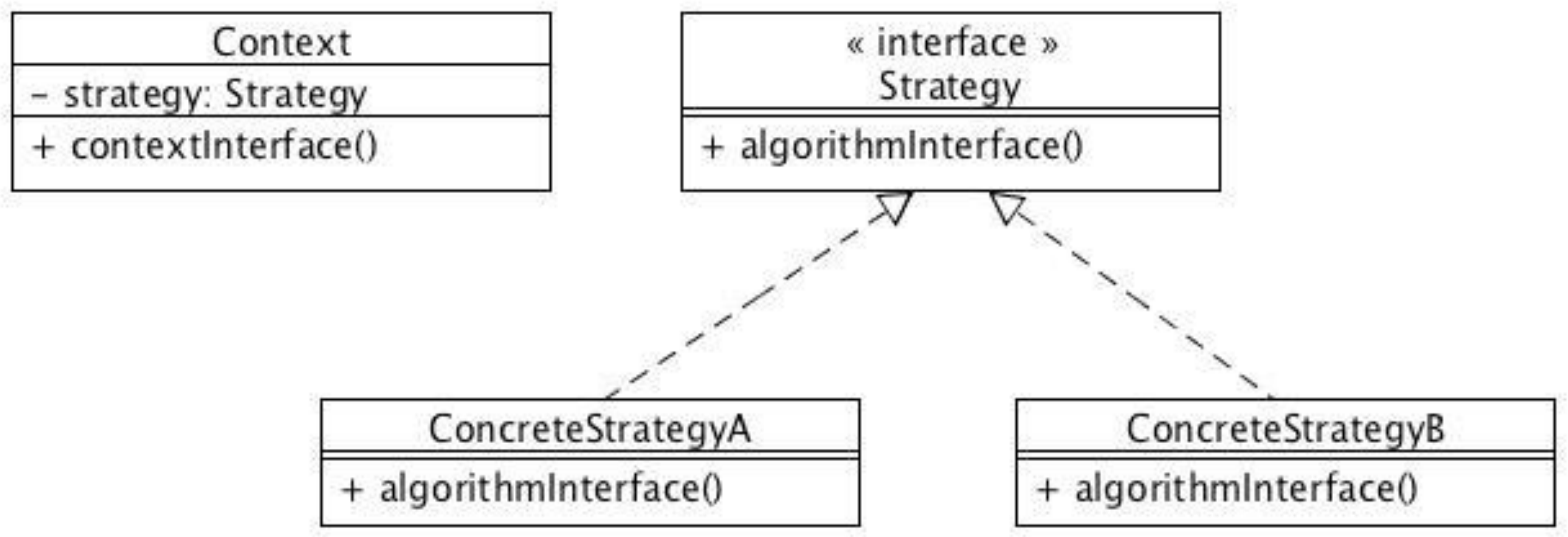


EXAMPLE: WITH THE STRATEGY PATTERN

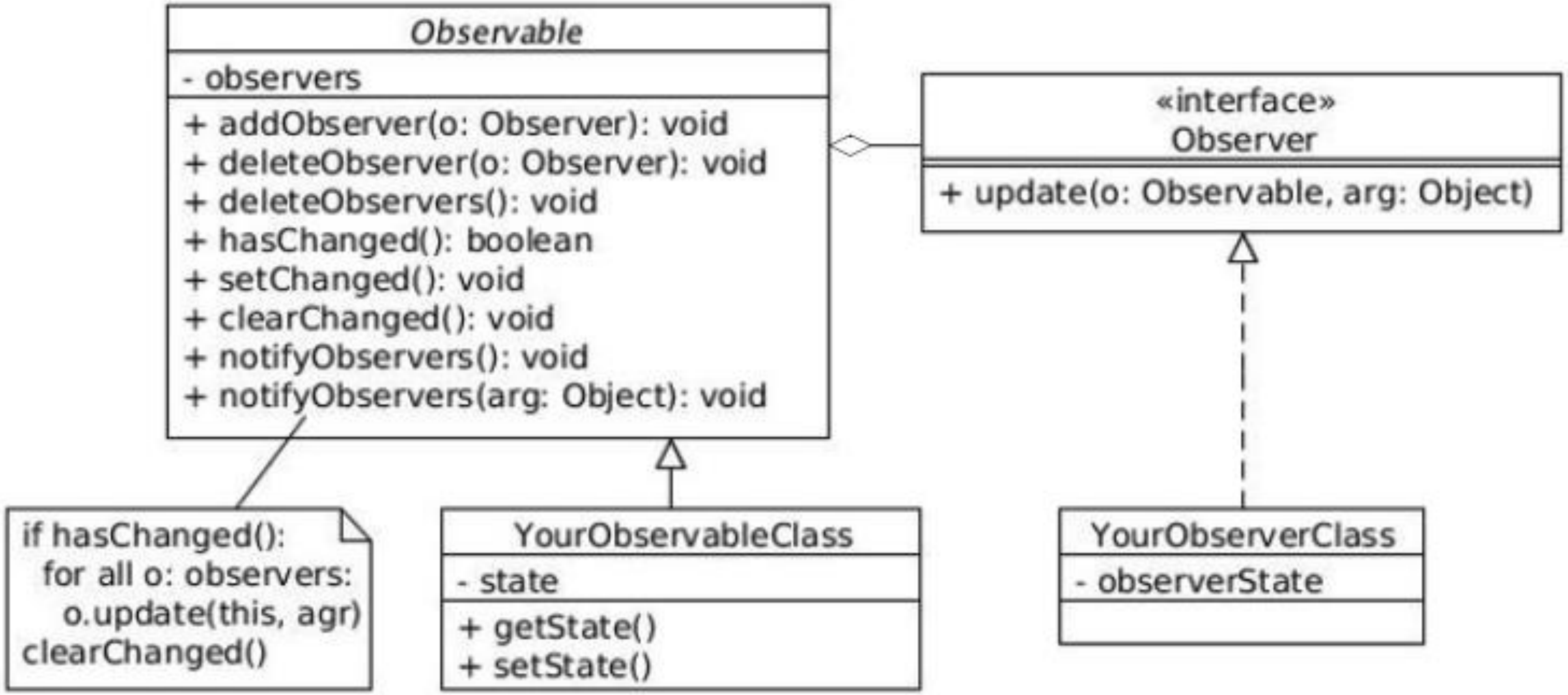


Computer Science
UNIVERSITY OF TORONTO

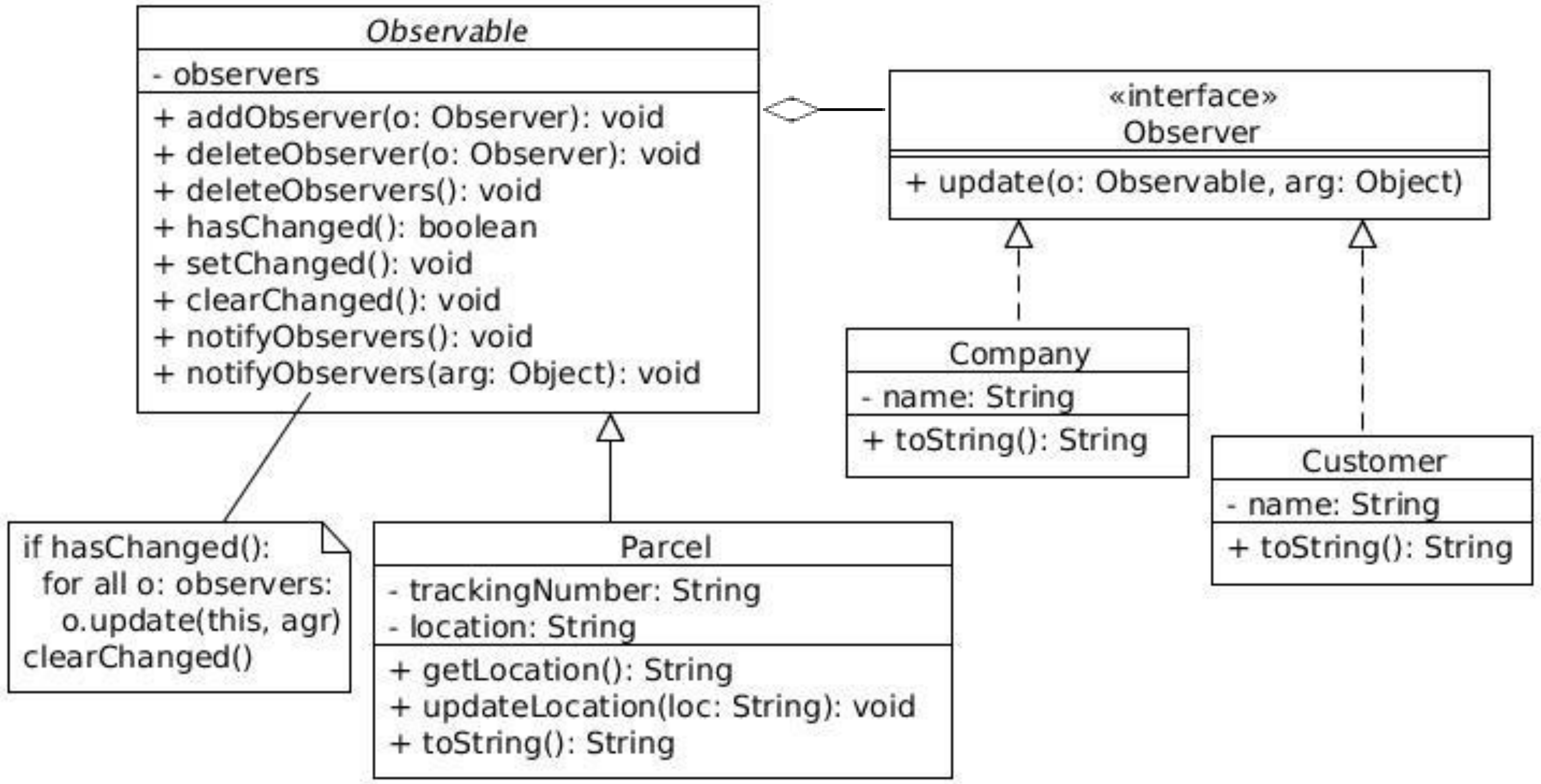
STRATEGY: STANDARD SOLUTION



OBSERVER: OLD JAVA IMPLEMENTATION

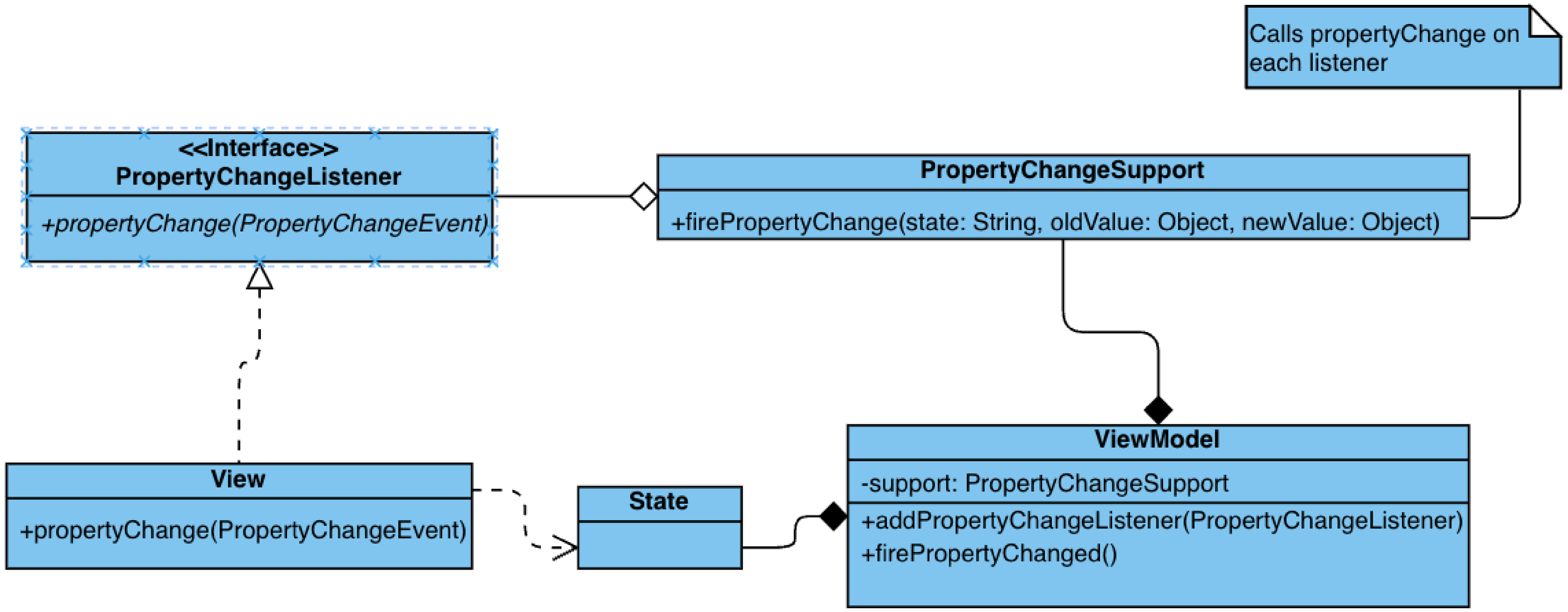


OBSERVER: PARCEL EXAMPLE IN JAVA



```
if hasChanged():
  for all o: observers:
    o.update(this, agr)
clearChanged()
```

OBSERVER: IMPLEMENTATION USING DELEGATION



Calls propertyChange on each listener
<<Interface>> PropertyChangeListener
+propertyChange(PropertyChangeEvent)
PropertyChangeSupport
+firePropertyChange(state: String, oldValue: Object, newValue: Object)
View
+propertyChange(PropertyChangeEvent)
State
ViewModel
-support: PropertyChangeSupport
+addPropertyChangeListener(PropertyChangeListener)
+firePropertyChanged()

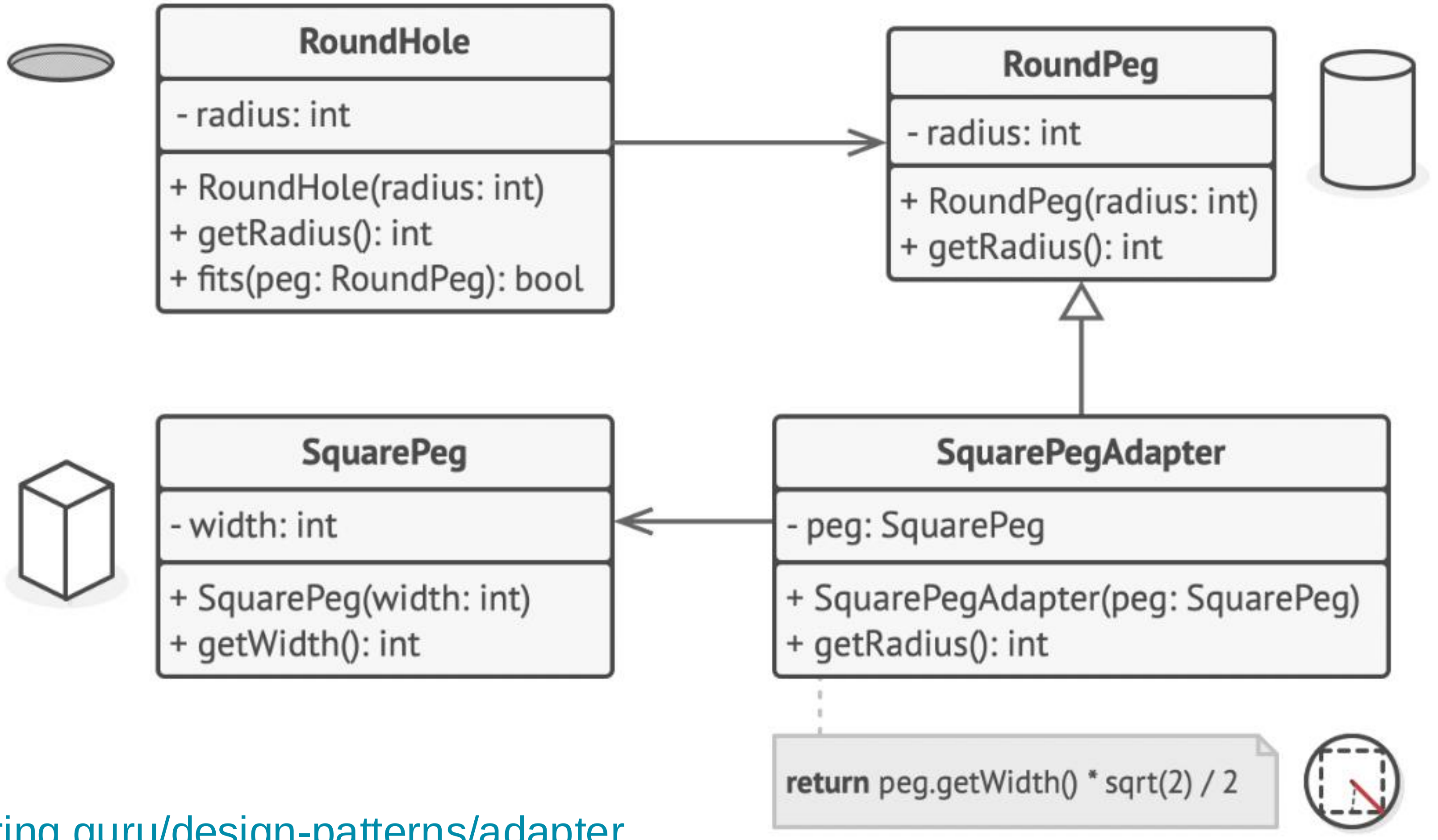
diagram created using https://online.visual-paradigm.com/

Computer Science
UNIVERSITY OF TORONTO

NEW OBSERVER PATTERN WITH "PARCEL"

- The "Parcel" class will now have a variable of type PropertyChangeSupport instead of inheriting from the Observer class.
- These two objects have the same role. They store the list of observers and call their "update" or "propertyChange" methods in in the observers.

ADAPTER DESIGN PATTERN: EXAMPLE



https://refactoring.guru/design-patterns/adapter

Computer Science
UNIVERSITY OF TORONTO

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();
}
```

Ethics

ETHICS

TO STUDY: UNIVERSAL DESIGN, CONCEPT OF "DISABILITY"

1. Compare and contrast the medical and social models of disability.

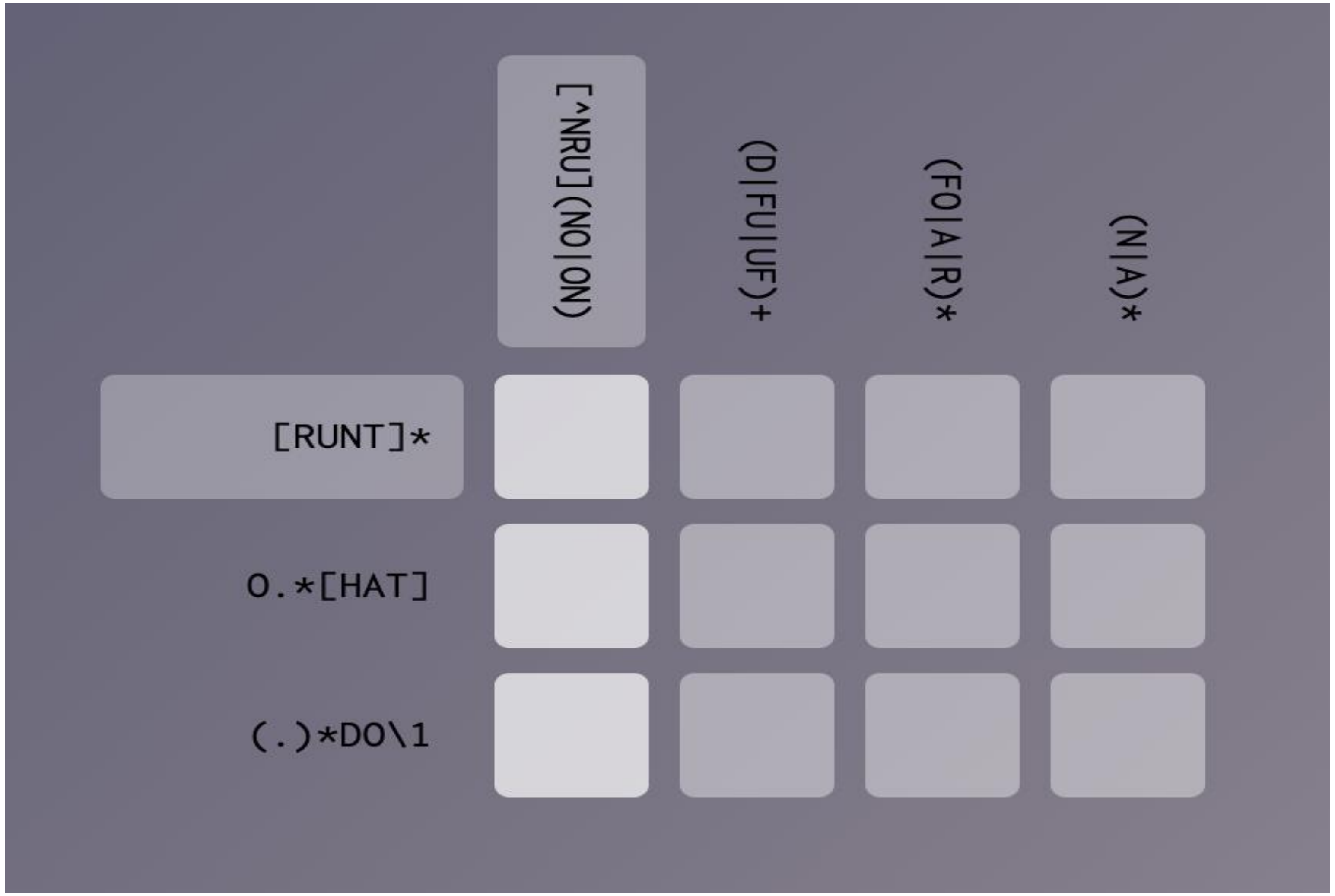
2. List three principles of universal design. Choose an app and explain how the app does or does not follow each principle.

Regular Expressions

REGULAR EXPRESSIONS (REGEX)

TO STUDY: BASIC SYMBOLS, REPETITION, WHAT DOES A REGEX DO?

1. Try this regex crossword. Which word does it spell?



Computer Science
UNIVERSITY OF TORONTO

Refactoring and Inverting Dependencies

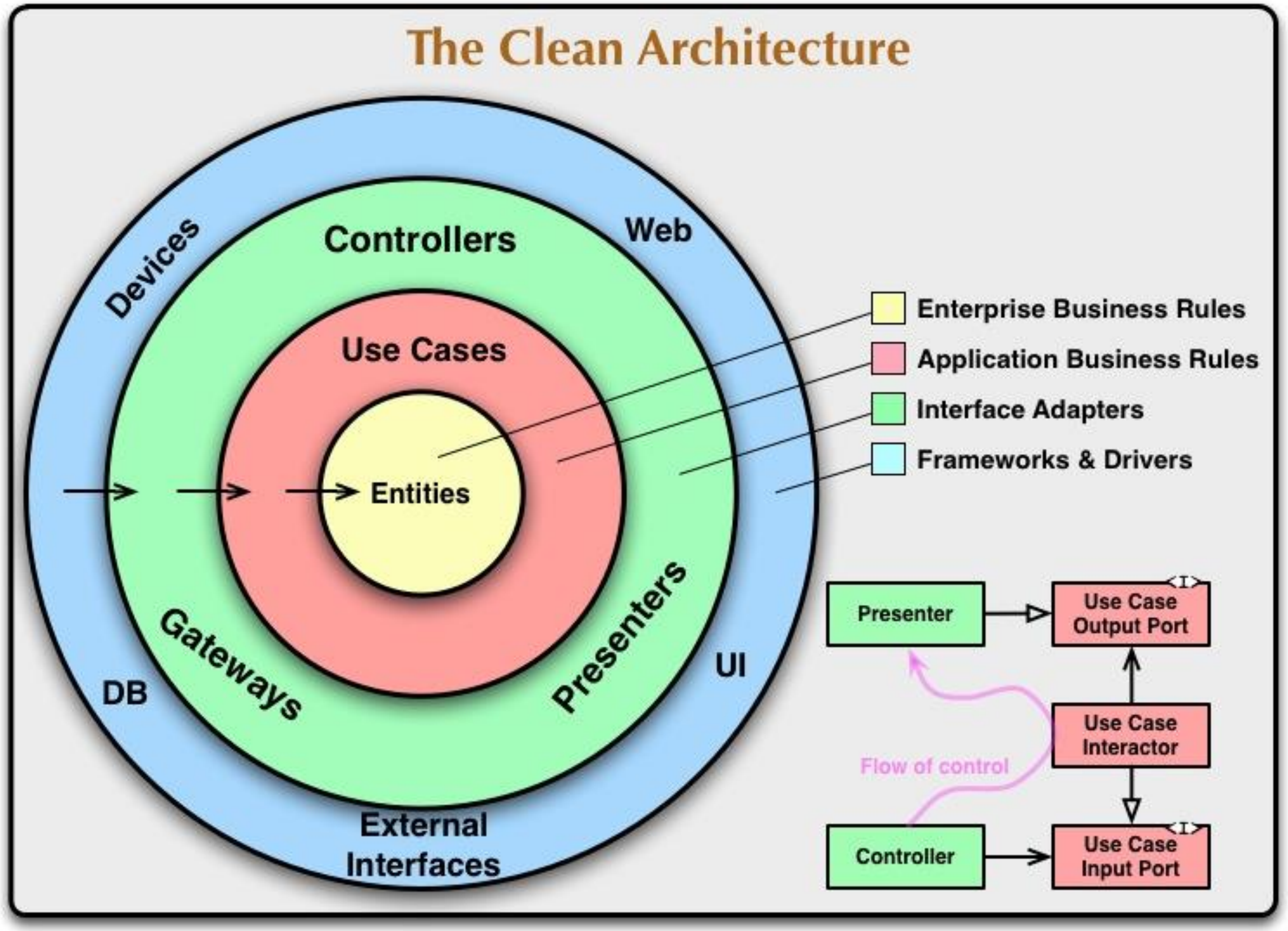
REFACTORING AND INVERTING DEPENDENCIES

TO STUDY: LAB CODE, INTERFACES, HOW TO INVERT A DEPENDENCY

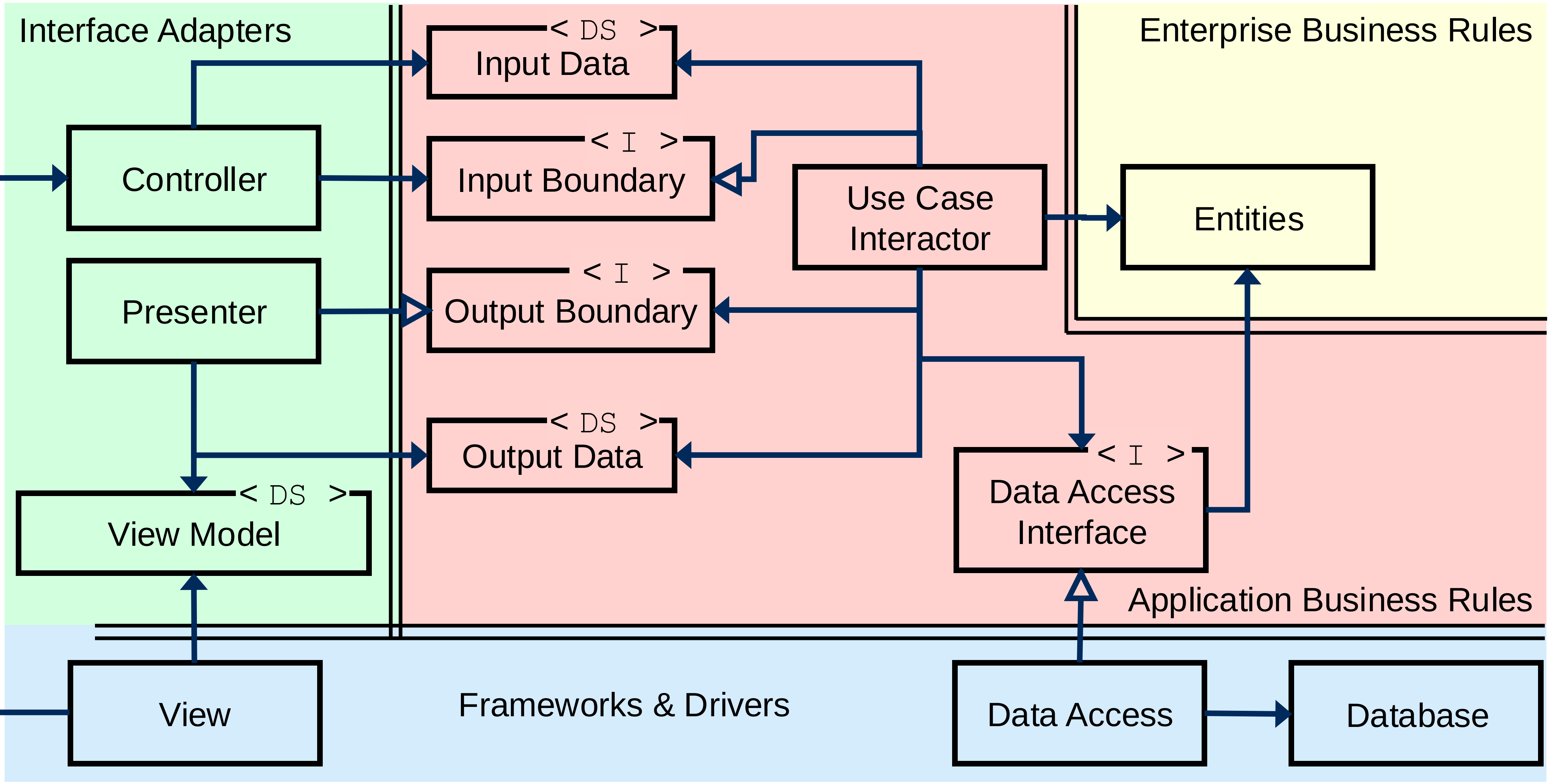
1. What is a helper method?
2. What is the difference between composition and inheritance?
3. When do you invert a dependency? How do you know you should do that?

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



EXAMPLE OF A PROGRAM WITH CLEAN ARCHITECTURE



Interface Adapters
Enterprise Business Rules
< DS >
Input Data
< I >
Input Boundary
Controller
Use Case Interactor
Entities
Presenter
< I >
Output Boundary
< DS >
Output Data
< I >
Data Access Interface
< DS >
View Model
Application Business Rules
View
Frameworks & Drivers
Data Access
Database