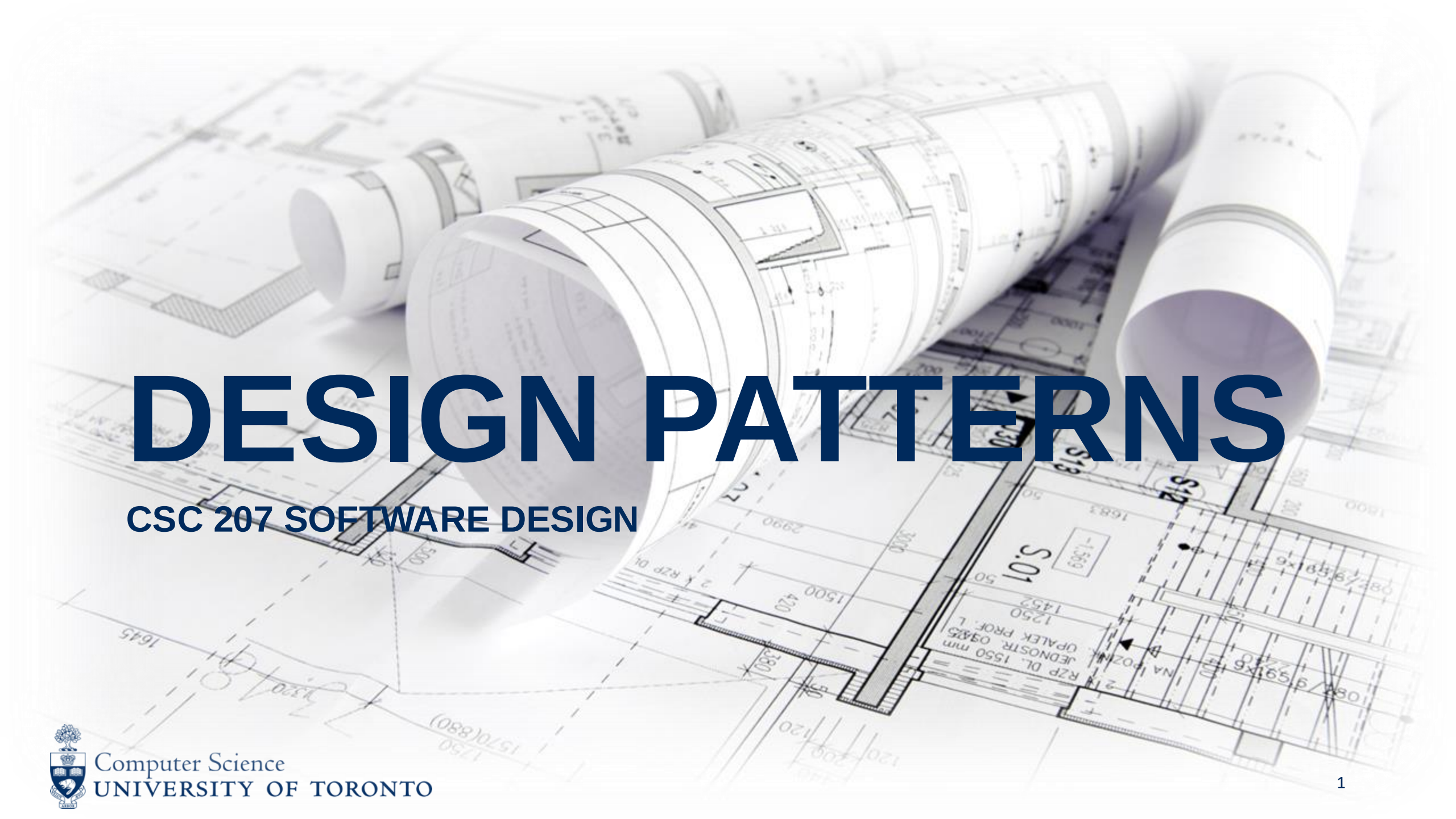


DESIGN PATTERNS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

Course Overview

Tools (Weeks 1-4)

- Java
- Version Control
- Software Tools

Design (Weeks 5-8)

- Clean Architecture
- SOLID
- Design Patterns

Professional Topics (Weeks 9-12)

- Ethics
- Internships
- GenAI

- Last week we discussed testing in Clean Architecture and packages.
- This week, we will:
  - introduce **design patterns**, and
  - have our first embedded ethics module on **user diversity**

Questions to be answered this week...

- What is a design pattern?
- What are the three categories of design patterns we will cover, and what are examples of each?
- Why shouldn't we design software applications only for the majority?
- What is relational harm?

LEARNING OUTCOMES

- Know what a design pattern is.
- Know the three broad categories of design patterns.
- Recognize some common design patterns and understand how to apply them.

DESIGN PATTERNS

- 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 single programming language.

- You can learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).

LOOSE COUPLING, HIGH COHESION

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

REMINDER: SOLID

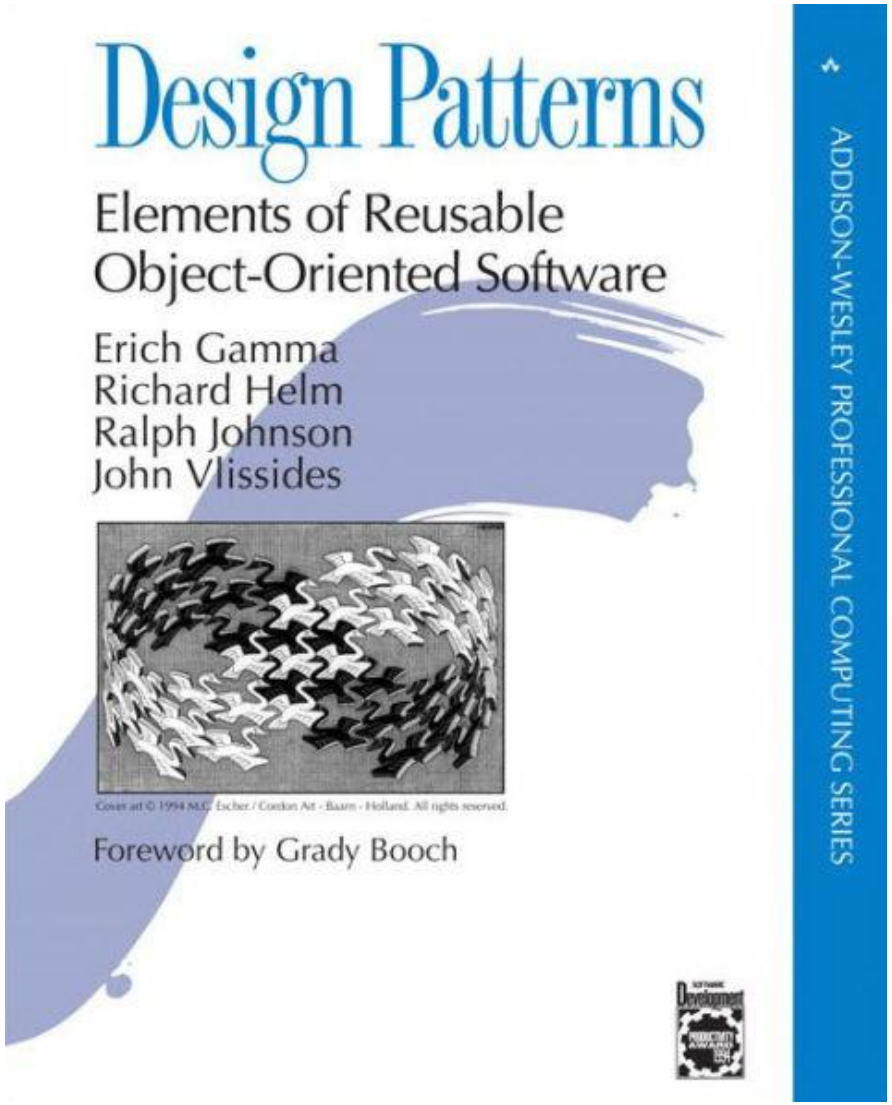
- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Keep these in mind as we discuss each design pattern!

GANG OF FOUR

- First codified by the Gang of Four in 1995

- Original 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 the 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

DESIGN PATTERNS, CATEGORIES OF

Creational (https://en.wikipedia.org/wiki/Creational_pattern)

- Patterns related to how we create instances of our classes.

Behavioural (https://en.wikipedia.org/wiki/Behavioral_pattern)

- Patterns related to how instances of our classes communicate.

Structural (https://en.wikipedia.org/wiki/Structural_pattern)

- Patterns related to how classes can naturally fit together.

DESIGN PATTERNS THAT WE WILL COVER

Creational:

- Dependency Injection, Simple Factory, Builder

Behavioural:

- Strategy, Observer

Structural:

- Adapter, Façade

https://sourcemaking.com/design_patterns/ has detailed explanations of many design patterns, which you may find useful for your project and beyond this course.

Similarly, https://refactoring.guru/design-patterns also provides code examples of various patterns

CREATIONAL PATTERNS

DEPENDENCY INJECTION

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.
- Example:
  - a class `Course` may depend on a class `Student` because a `Course` contains instances of class `Student`.

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 DESIGN PATTERN

- Problem:
  - We are writing a class, and we need to assign values to the instance variables, but we don’t want to introduce a hard dependency.

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?

DEPENDENCY INJECTION: IN PRACTICE

- Where have we seen Dependency Injection before in Clean Architecture?

- How does the Dependency Inversion Principle (SOLID) relate to the Dependency Injection Design Pattern?

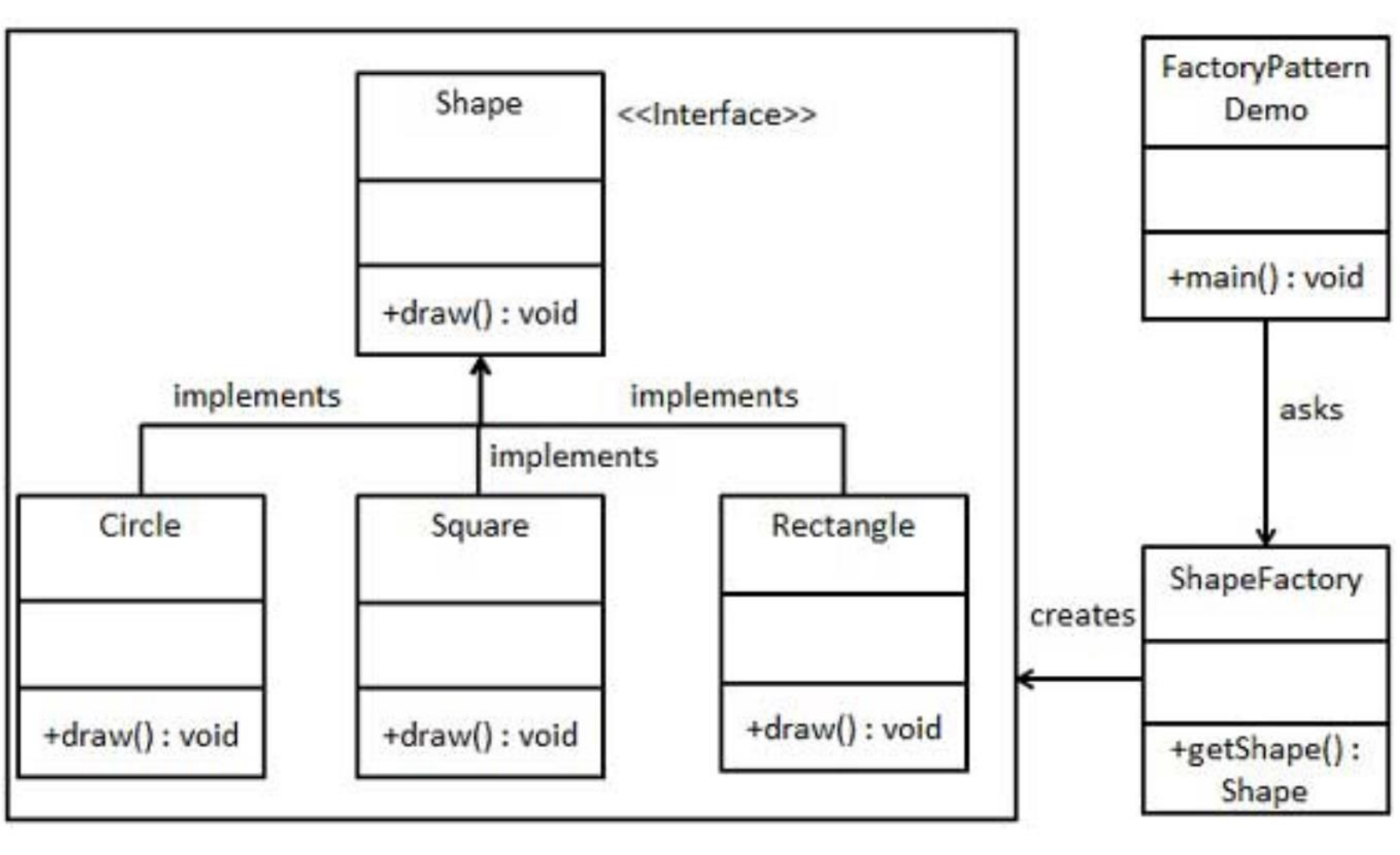
CREATIONAL PATTERNS

SIMPLE FACTORY

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.
  - Later, we might want to change the types of the objects we are creating (so avoiding hard dependencies!)

FACTORY : AN EXAMPLE



FACTORY : IN PRACTICE

- What do we gain by having the factory be responsible for creating instances of objects for us?
- Is it still a "factory" if the method only returns instances of one class (say "Rectangle") and not instances of a subclass?
  - In other words, is it still a factory if we replace a ShapeFactory with a RectangleFactory, where the Rectangle class has no subclasses?
  - Is it still useful to replace ShapeFactory with RectangleFactory if all we need are Rectangle objects?

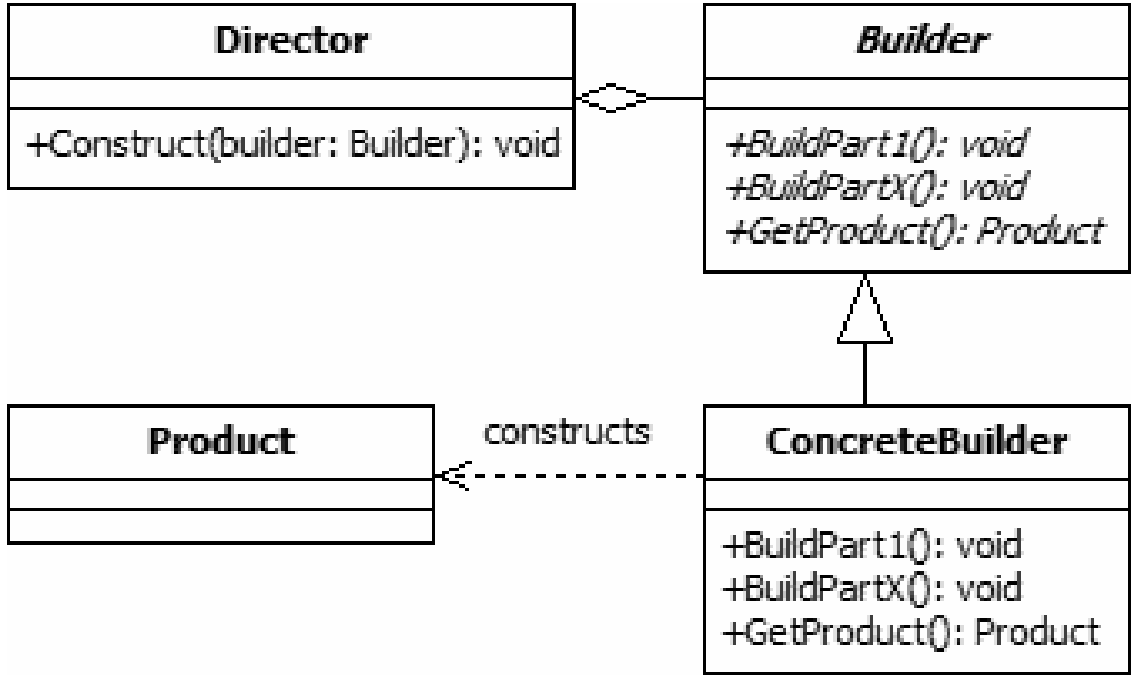
CREATIONAL PATTERNS

BUILDER

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.



WITHOUT APP BUILDER VERSION (ONLY SIGNUP SHOWN)

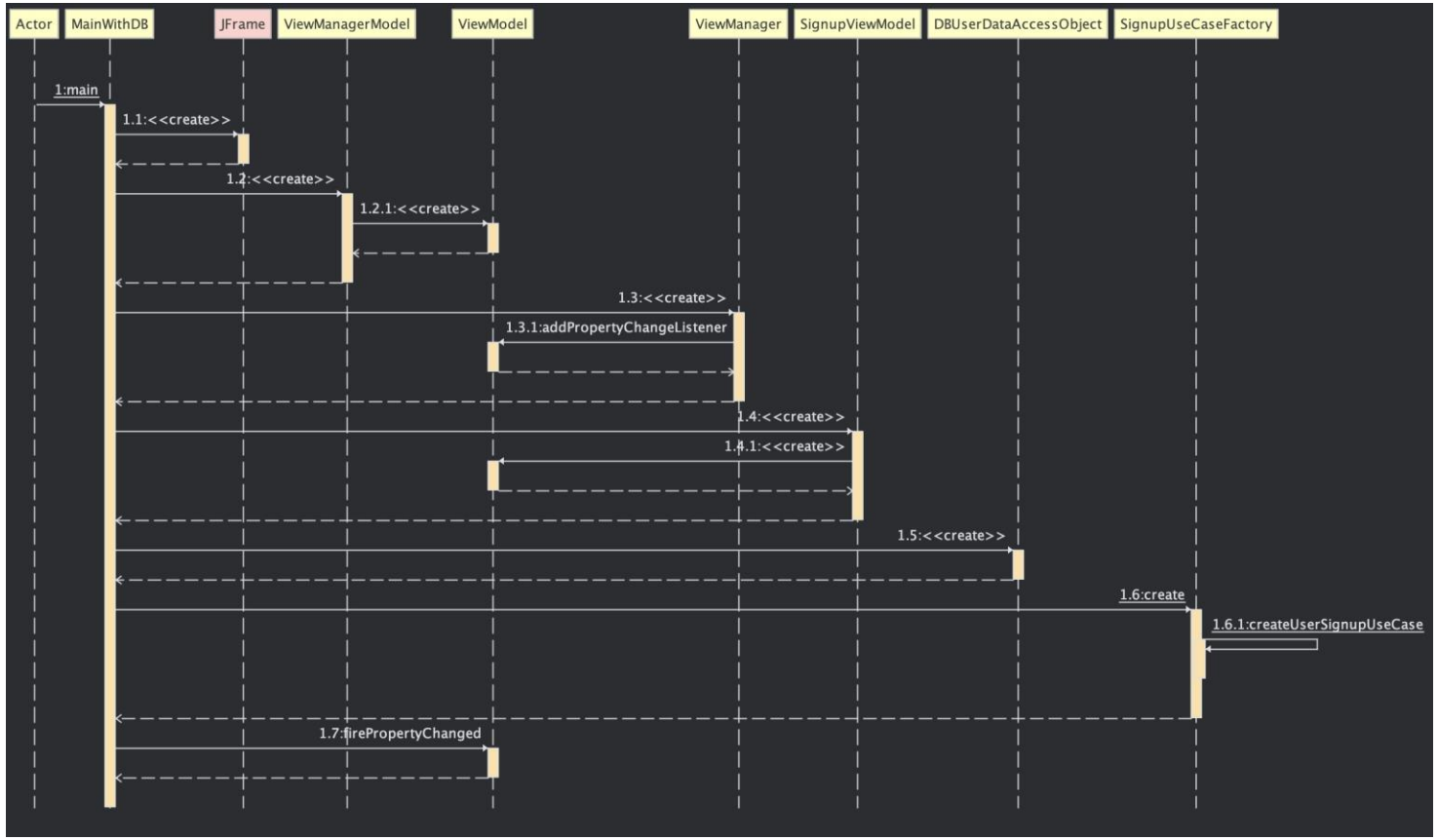
```
final JFrame application = new JFrame("Login Example");

final CardLayout cardLayout = new CardLayout();

final JPanel views = new JPanel(cardLayout);
application.add(views);

final ViewManagerModel viewManagerModel = new ViewManagerModel();
new ViewManager(views, cardLayout, viewManagerModel);

final SignupViewModel signupViewModel = new SignupViewModel();

final DBUserDataAccessObject userDataAccessObject = new DBUserDataAccessObject(new CommonUserFactory());

final SignupView signupView = SignupUseCaseFactory.create(viewManagerModel, loginViewModel,
        signupViewModel, userDataAccessObject);
views.add(signupView, signupView.getViewName());
```

APP BUILDER VERSION

```
final AppBuilder appBuilder = new AppBuilder();
final JFrame application = appBuilder
                                        .addLoginView()
                                        .addSignupView()
                                        .addLoggedInView()
                                        .addSignupUseCase()
                                        .addLoginUseCase()
                                        .addChangePasswordUseCase()
                                        .build();
```

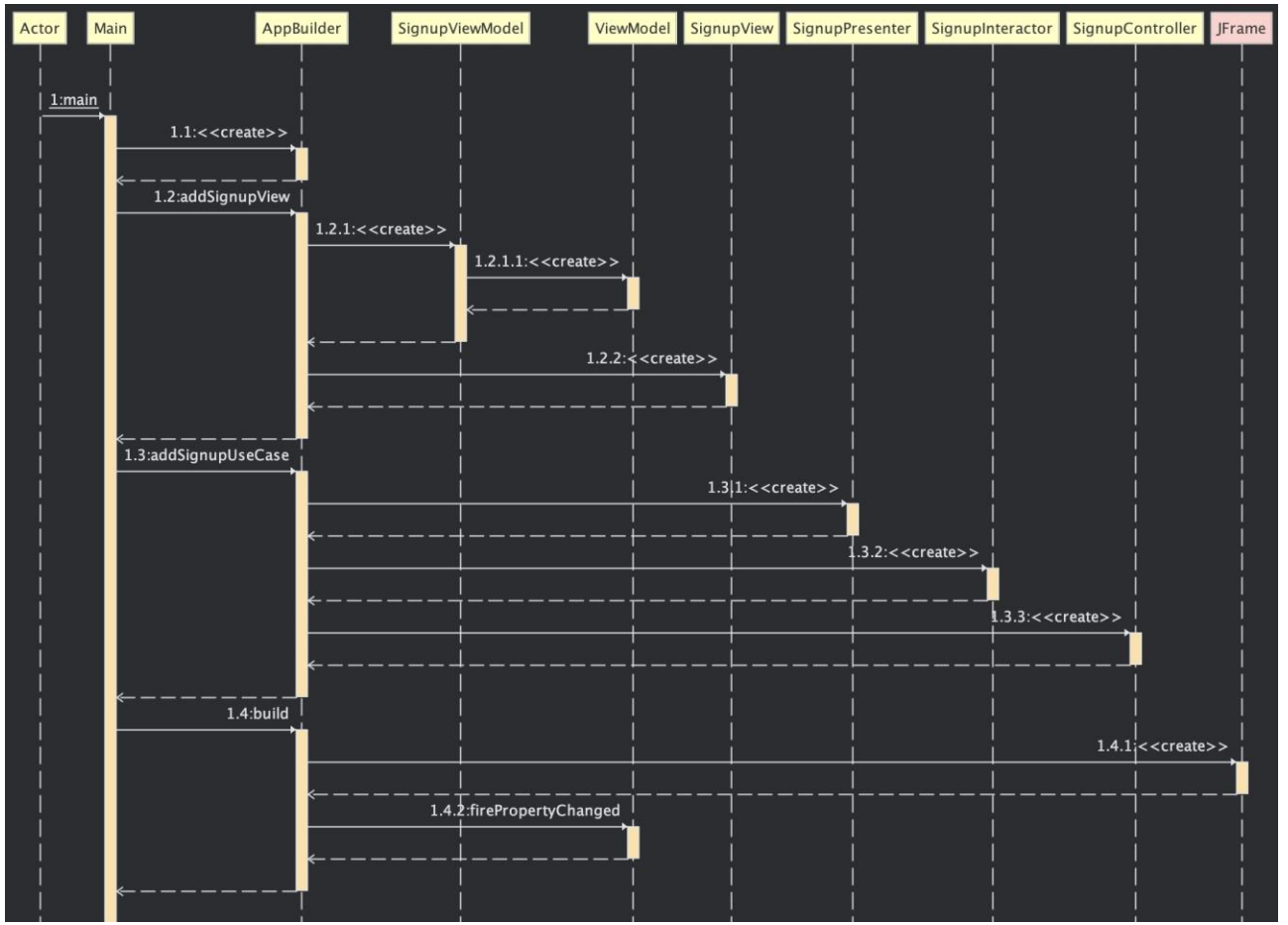
BUILDER DESIGN PATTERN: BEFORE (SEQ DIAGRAM)

- This is the sequence diagram for method `Main.main` from the Phase 1 of the homework.
- This is just **a subset of the diagram** for setting up the Signup part of the CA engine.
- Note the Factory which helps hide some of the details for us!



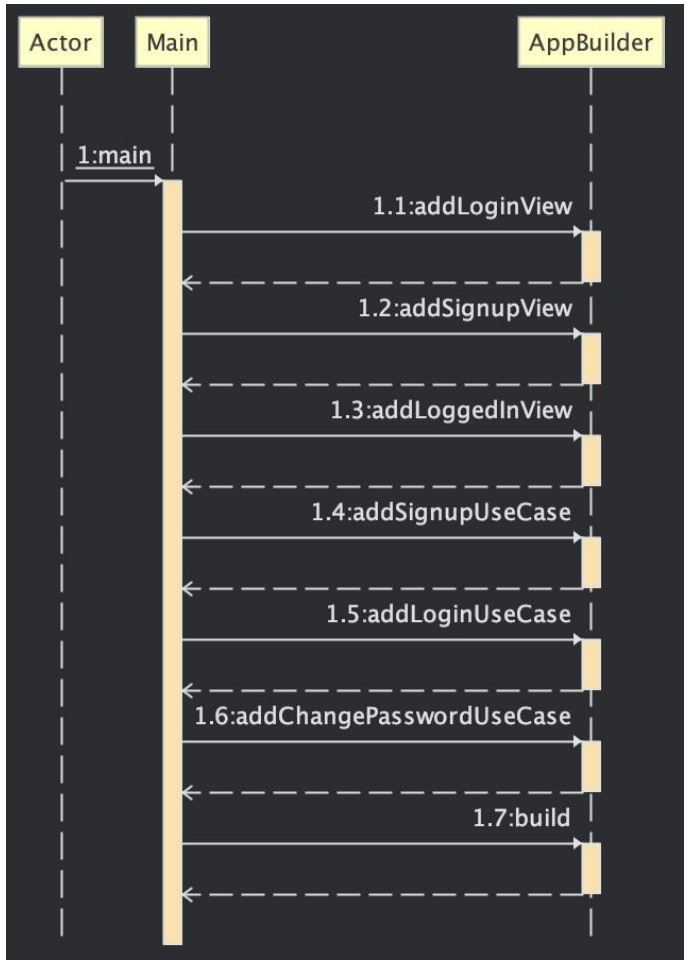
BUILDER DESIGN PATTERN: AFTER (SEQ DIAGRAM)

- This is the sequence diagram for when we create the lab-5 application in Main.main, but only showing the part related to the Signup Use Case.
- This is just **a subset of the diagram** for setting up the Signup part of the CA engine.
- Note that now the final JFrame is created by the Builder for us when we build the app.



BUILDER DESIGN PATTERN: AFTER (SEQ DIAGRAM)

- This is the sequence diagram for when we create the lab-5 application in Main.main like before.
- We have hidden the calls to constructors; you can generate this in IntelliJ to see *all* the calls that take place (there are a lot!).
- The details are hidden in the AppBuilder!



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

- A comparison of Factory and Builder https://medium.com/javarevisited/design-patterns-101-factory-vs-builder-vs-fluent-builder-da2babf42113

BUILDER DESIGN PATTERN: IN PRACTICE

- Where have we seen builders before?
- How complicated does an object have to be, to require a builder?
- Which SOLID principles does the Builder design pattern follow?

BEHAVIOURAL PATTERNS

STRATEGY

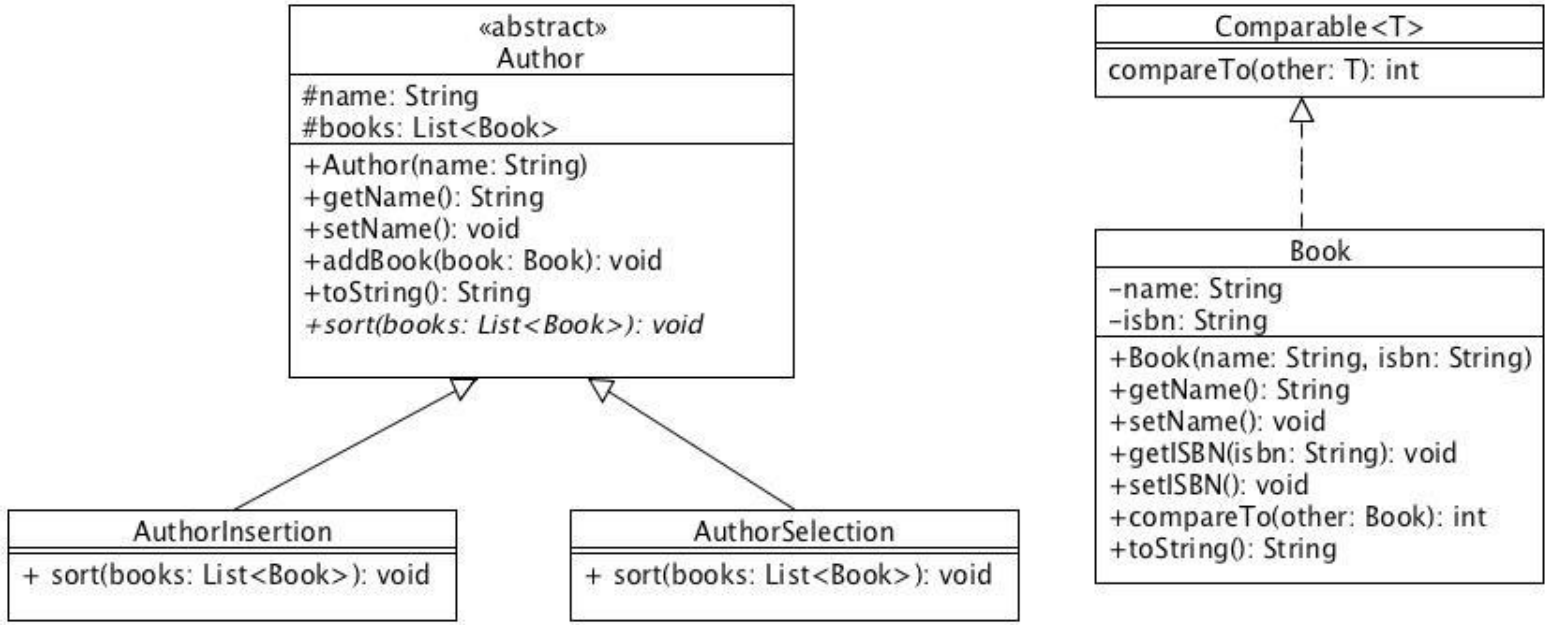
Computer Science
UNIVERSITY OF TORONTO

STRATEGY DESIGN PATTERN

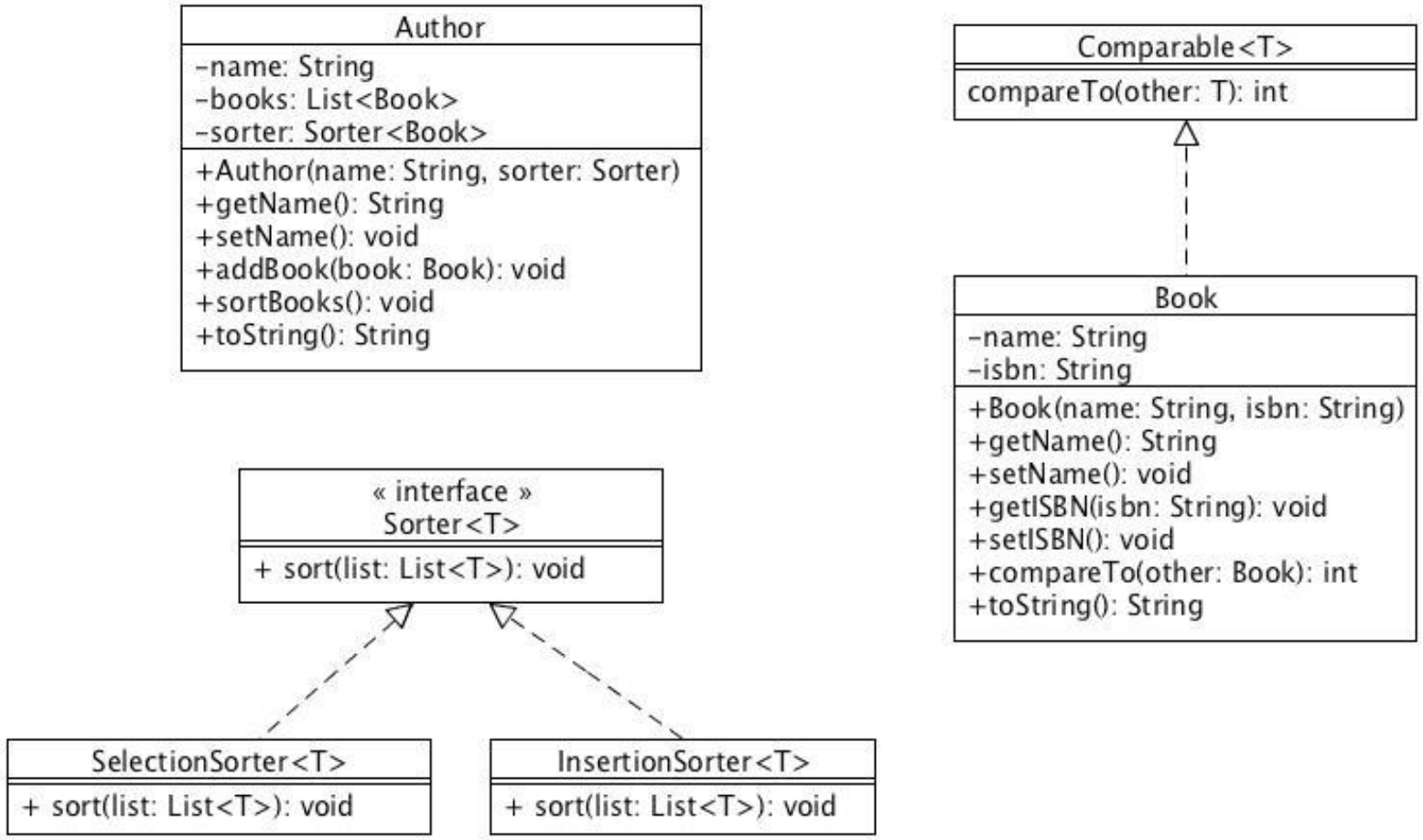
- Problem:
  - multiple classes differ only in how they are implemented
    - the high-level logic is the same except for which algorithm is being used to solve part of the task
  - other classes may also benefit from the code implementing the algorithms, but the code is currently coupled to the class using a specific algorithm.

- Goal:
  - want to **decouple** — separate — the implementation of a class from the implementation of the algorithms which it may use.

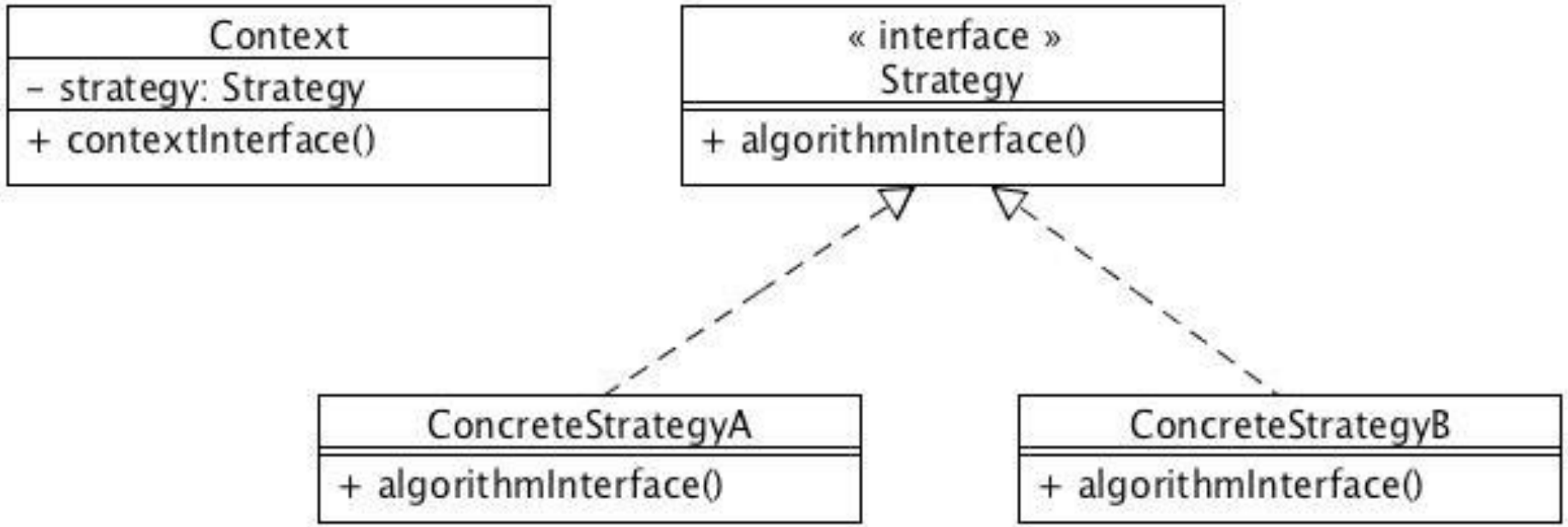
EXAMPLE: WITHOUT THE STRATEGY PATTERN



EXAMPLE: WITH THE STRATEGY PATTERN



STRATEGY: STANDARD SOLUTION



STRATEGY PATTERN: IN PRACTICE

- What counts as a strategy? Does it have to be an algorithm?

- Which of the SOLID principles are followed by this pattern?

BEHAVIOURAL PATTERNS

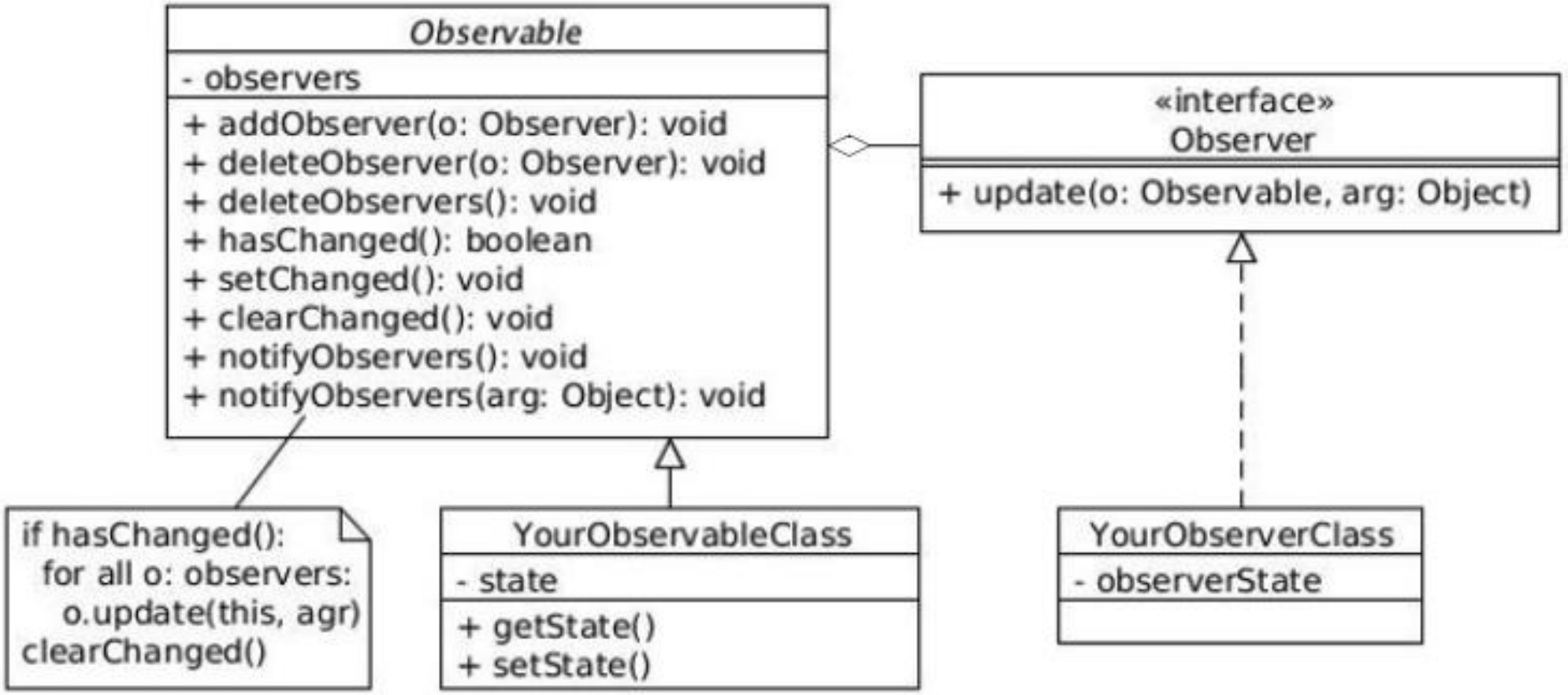
OBSERVER

Computer Science
UNIVERSITY OF TORONTO

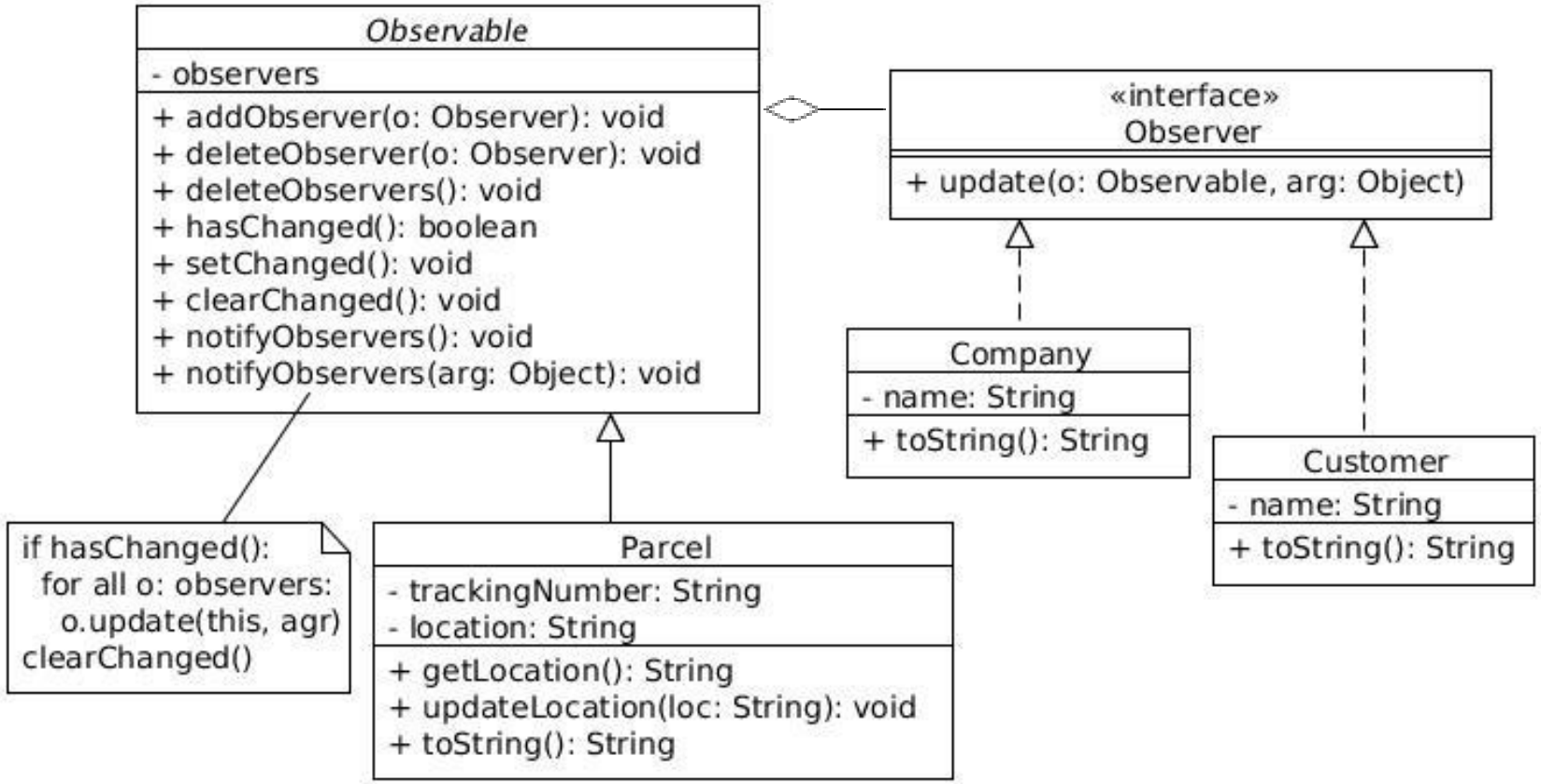
OBSERVER DESIGN PATTERN

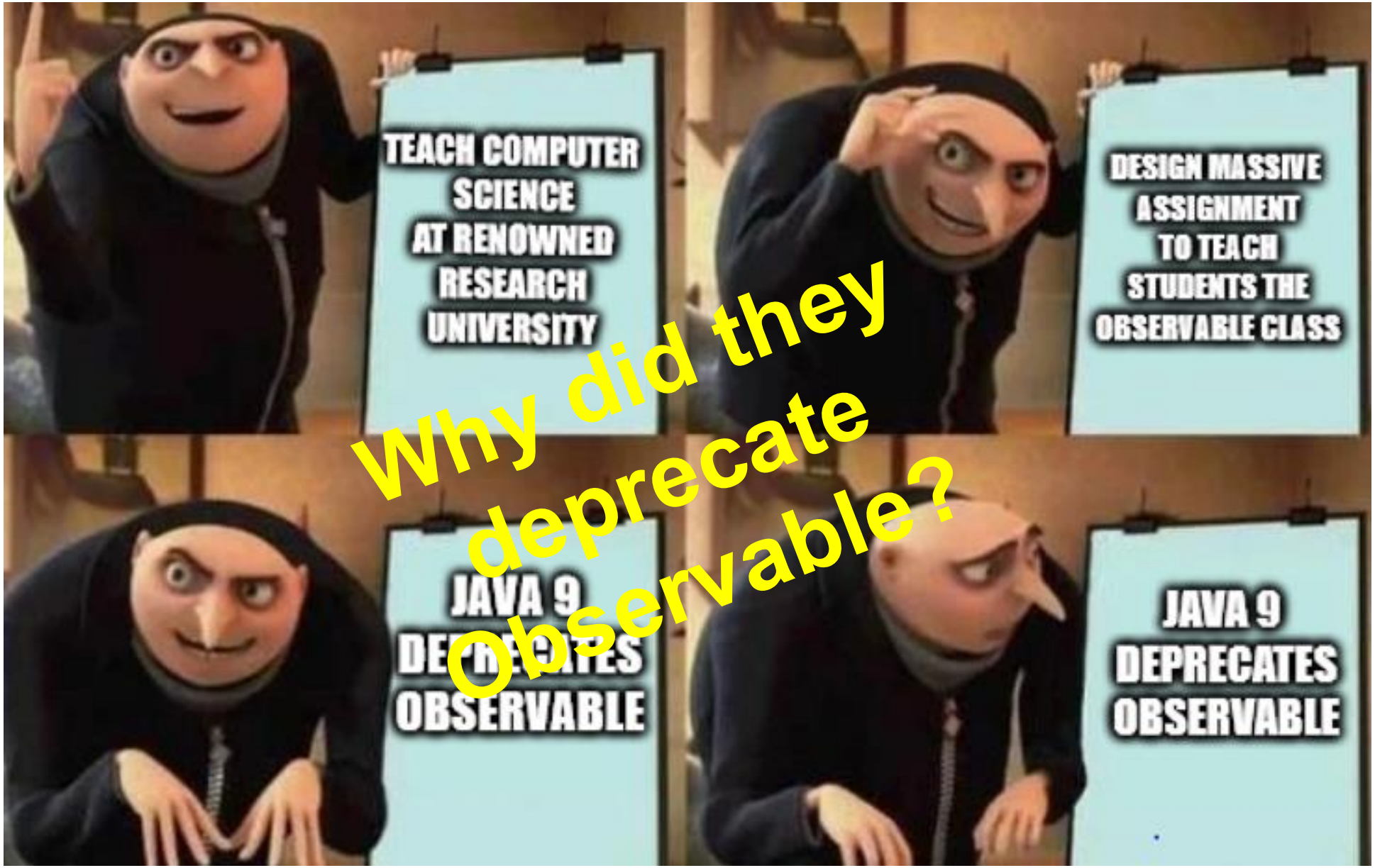
- Problem:
  - Need to maintain consistency between related objects.
  - Two aspects, one dependent on the other (**cause and effect**)
  - An object should be able to notify other objects about changes to itself without making assumptions about who these objects are.
  - You want one object to "listen” for changes in another

OBSERVER: OLD JAVA IMPLEMENTATION



OBSERVER: PARCEL EXAMPLE IN JAVA




TEACH COMPUTER SCIENCE AT RENOWNED RESEARCH UNIVERSITY
DESIGN MASSIVE ASSIGNMENT TO TEACH STUDENTS THE OBSERVABLE CLASS
JAVA 9 DEPRECATES OBSERVABLE
JAVA 9 DEPRECATES OBSERVABLE
Why did they deprecate Observable?

BUT WHAT HAVE THEY DONE?!?

Class Observable

java.lang.Object
    java.util.Observable

Deprecated.

This class and the `Observer` interface have been deprecated. The event model supported by `Observer` and `Observable` is quite limited, the order of notifications delivered by `Observable` is unspecified, and state changes are not in one-for-one correspondence with notifications. For a richer event model, consider using the `java.beans` package. For reliable and ordered messaging among threads, consider using one of the concurrent data structures in the `java.util.concurrent` package. For reactive streams style programming, see the `Flow` API.

OBSERVER: IMPLEMENTATION USING DELEGATION

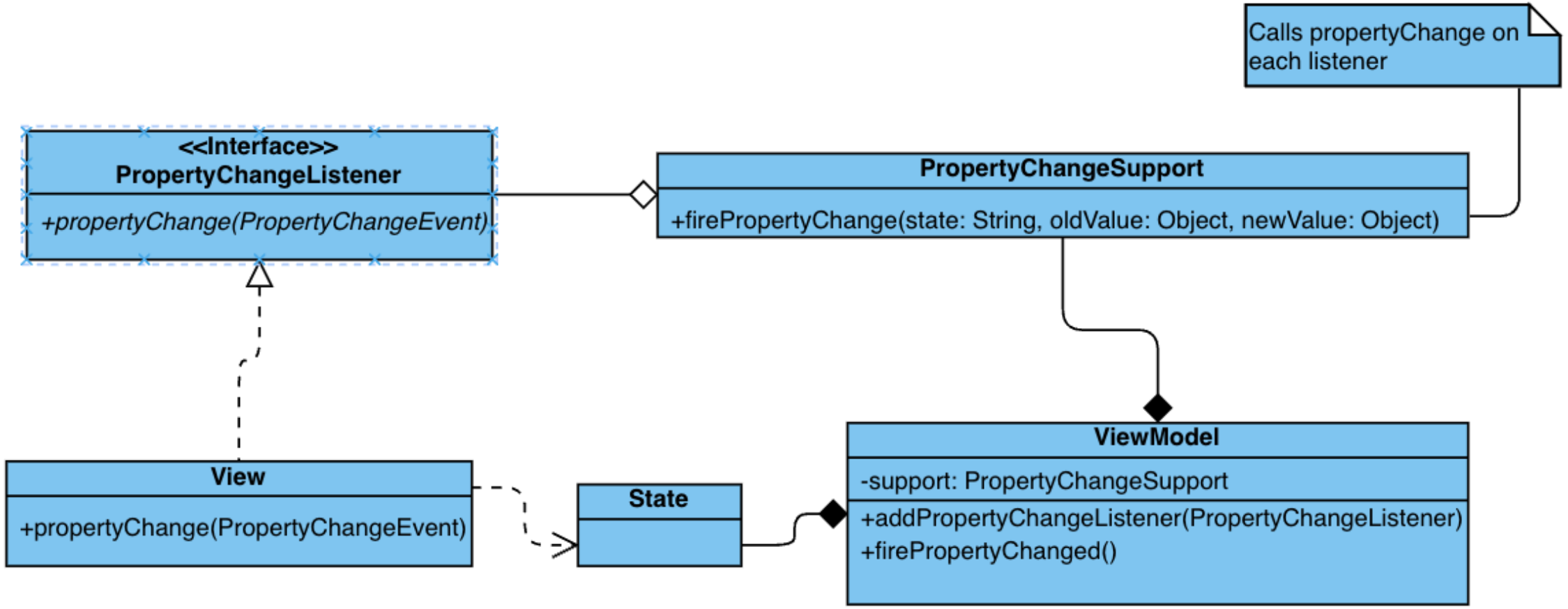


diagram created using https://online.visual-paradigm.com/

OBSERVER: IN PRACTICE

- Where have we already seen observers?
- Which part(s) of clean architecture can benefit from the observer pattern?
- This is a good pattern to implement across a boundary. Why is that?

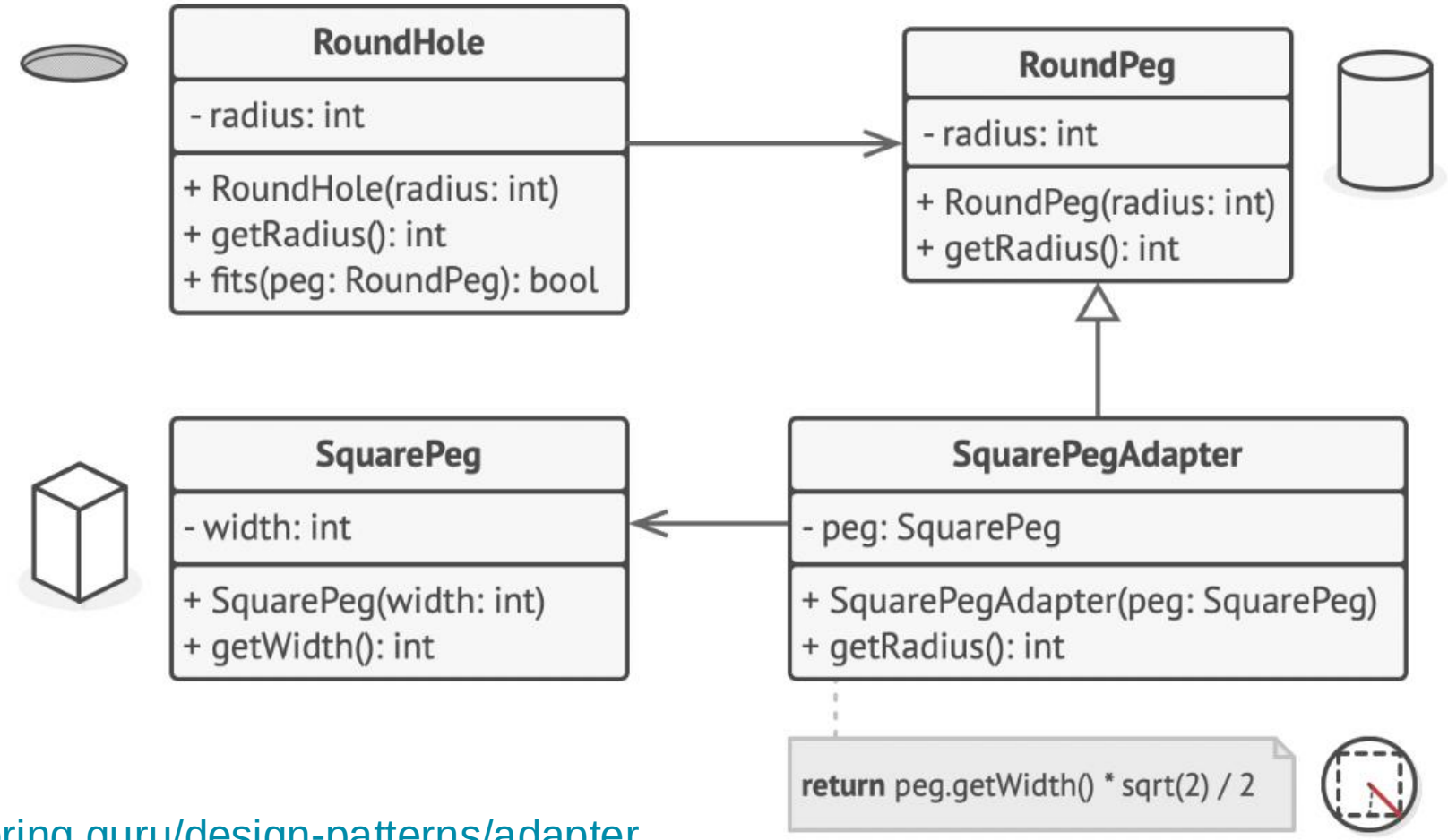
STRUCTURAL PATTERNS

ADAPTER

ADAPTER DESIGN PATTERN

- Problem:
  - Want to reuse a class that already exists, but it does not have the methods (public interface) required by the rest of the program.
- Solution 1 (use inheritance):
  - Create a subclass that extends the old class and includes the missing methods.
- Solution 2 (use a wrapper + delegation):
  - Create a container class that has an instance of the old class as a variable. The rest of the program can call the container's methods, which then call the old class's methods.

ADAPTER DESIGN PATTERN: EXAMPLE



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

ADAPTER DESIGN PATTERN: IN PRACTICE

- Which SOLID principles are followed by this pattern?
- Where have we seen adapters before?
- When might you NOT want to use this pattern?

STRUCTURAL PATTERNS

FAÇADE

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();
}
```

FAÇADE DESIGN PATTERN: IN PRACTICE

- When did we see an example of a Façade? Which SOLID principle was it demonstrating?
- How do Façade classes create a boundary within your program?
- https://en.wikipedia.org/wiki/Facade_pattern has a nice discussion of Adapter, Façade, and Decorator — the last one not being a pattern covered in this course)