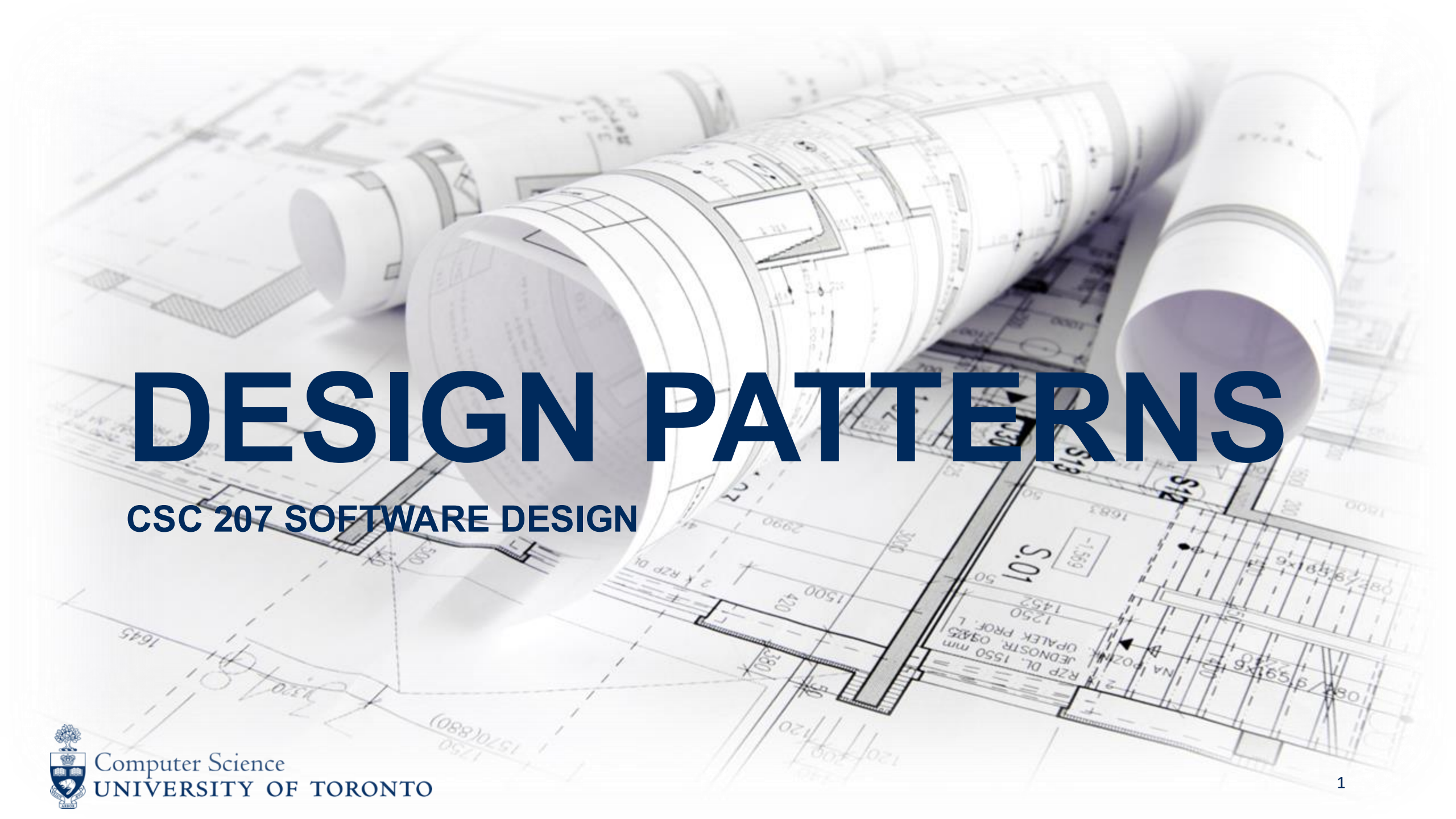


DESIGN PATTERNS

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

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).

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:

- 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

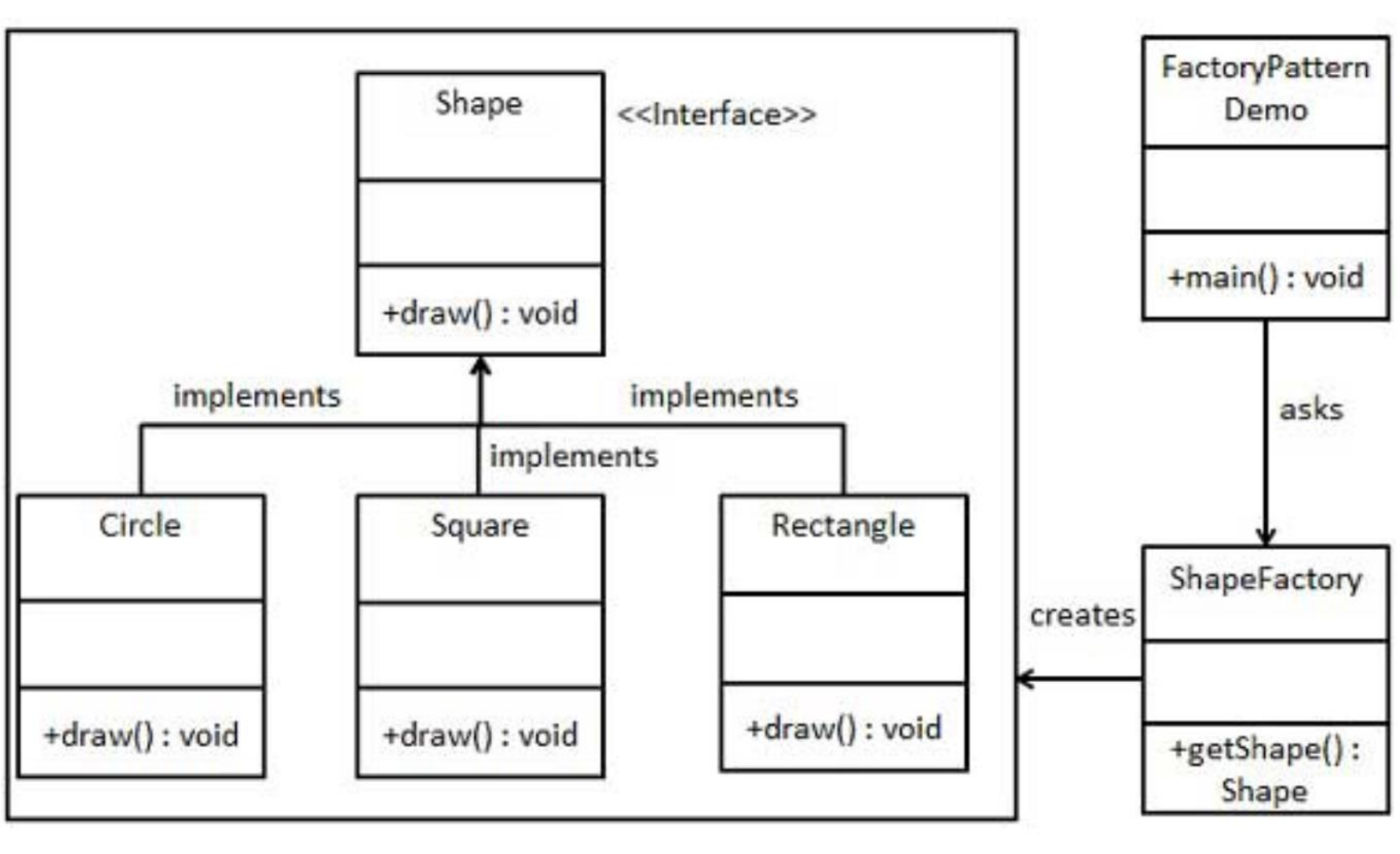
CREATIONAL PATTERNS

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



CREATIONAL PATTERNS

BUILDER

Computer Science
UNIVERSITY OF TORONTO

BUILDER DESIGN PATTERN

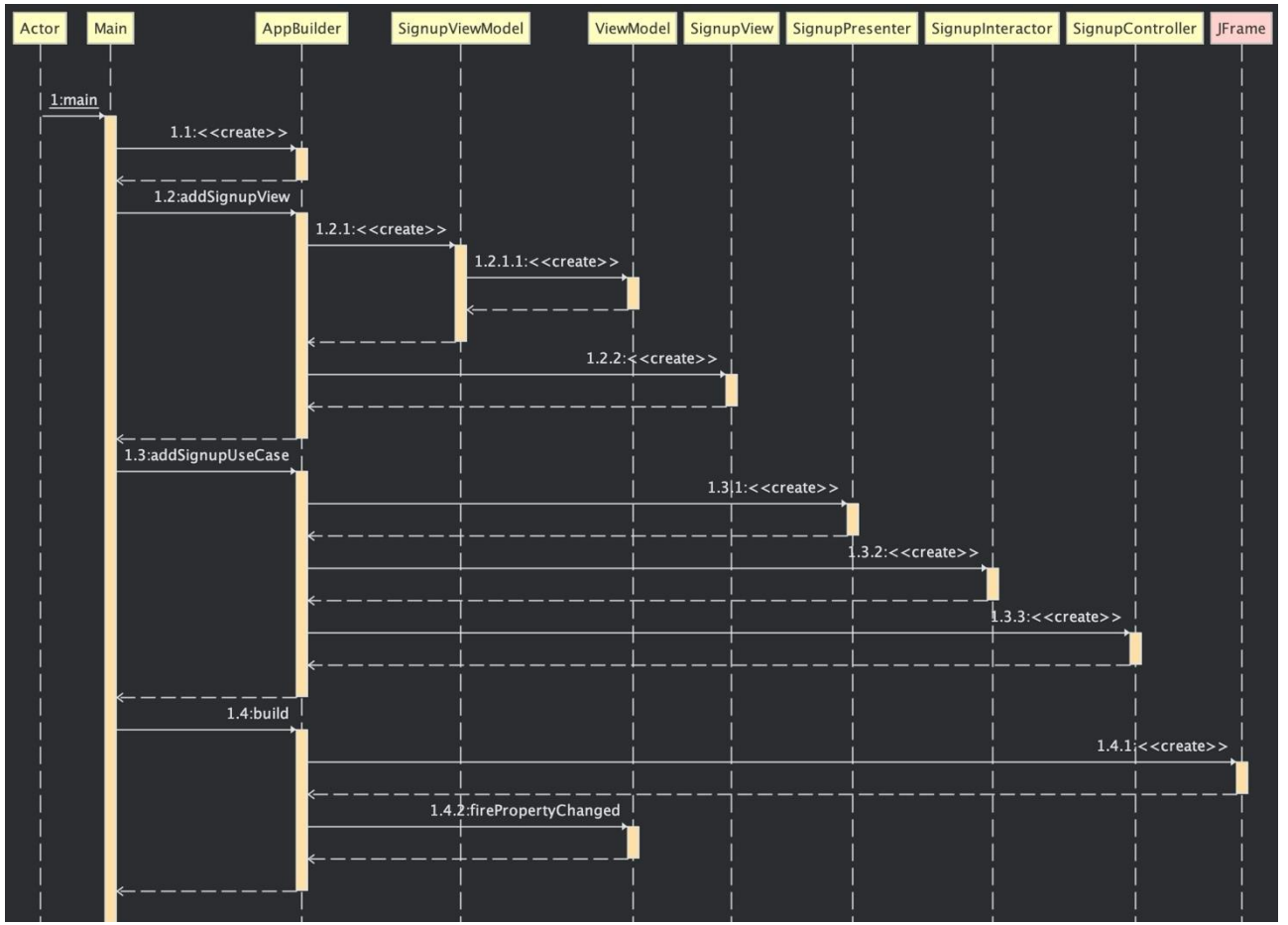
- Problem:
  - Need to create a complex structure of objects in a step-by-step fashion.
- Solution:
  - Create a Builder object that creates the complex structure.

APP BUILDER VERSION OF LOGIN CODE

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

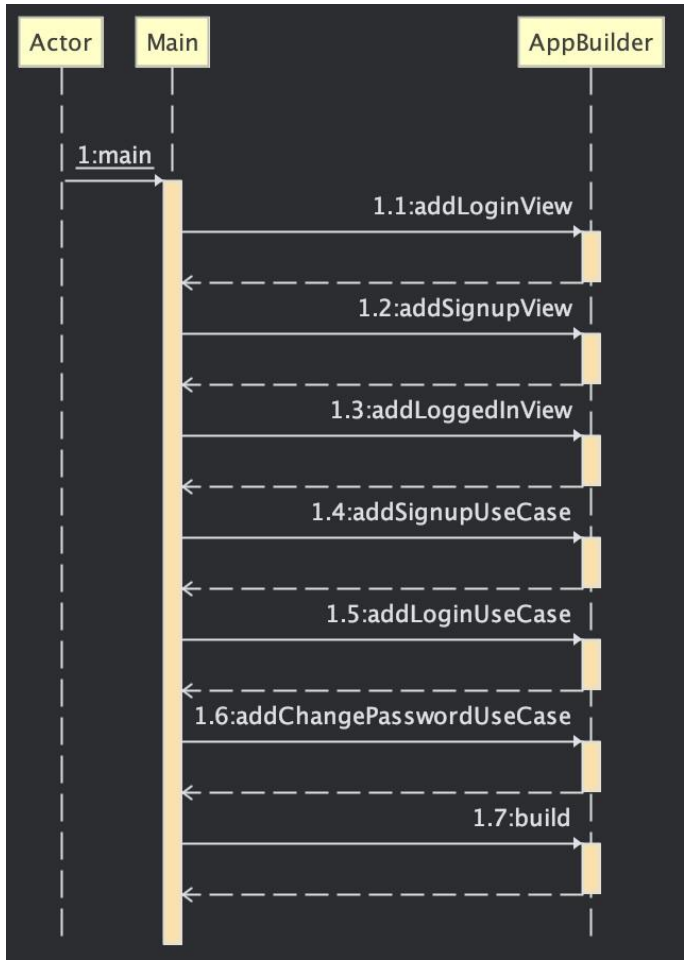
BUILDER DESIGN PATTERN (SEQ DIAGRAM)

- This is the sequence diagram for when we create the login app 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 the final JFrame is created by the Builder for us when we build the app.



BUILDER DESIGN PATTERN (SEQ DIAGRAM)

- This is the sequence diagram from the last slide, but 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

BEHAVIOURAL PATTERNS

STRATEGY

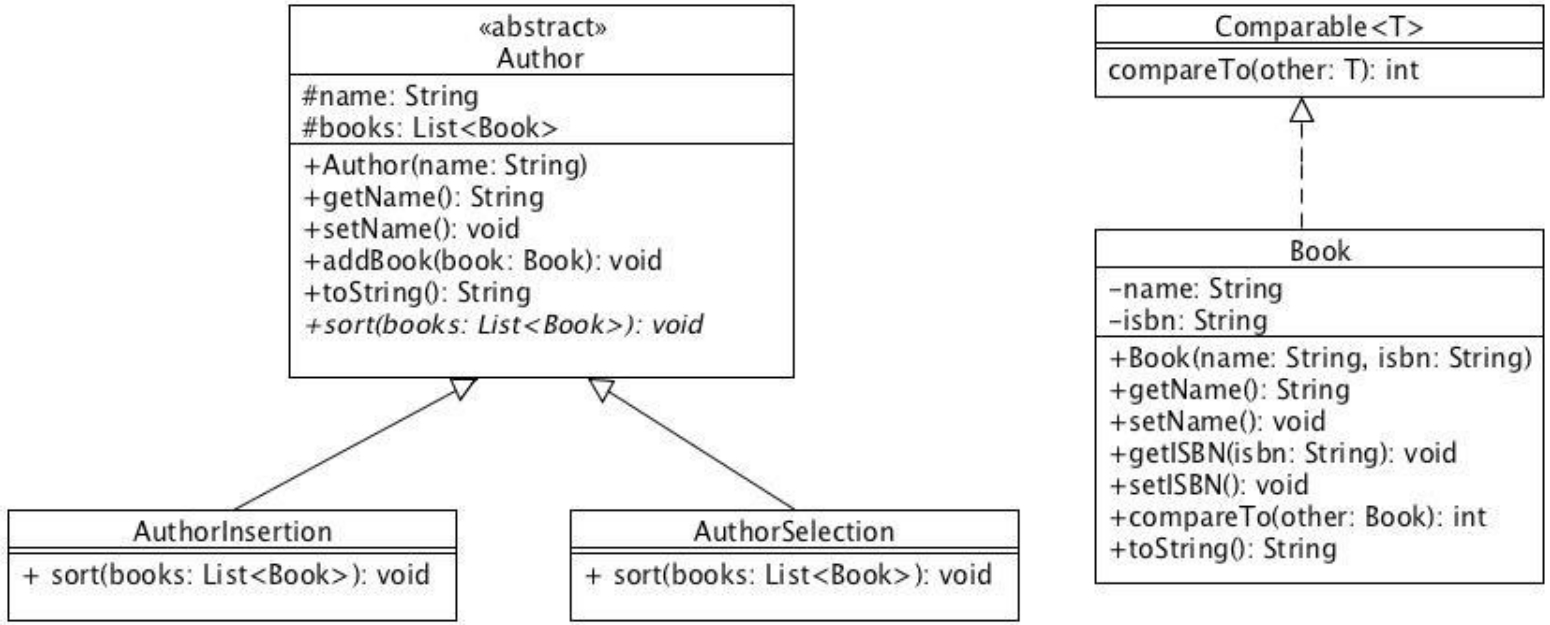
Computer Science
UNIVERSITY OF TORONTO

STRATEGY DESIGN PATTERN

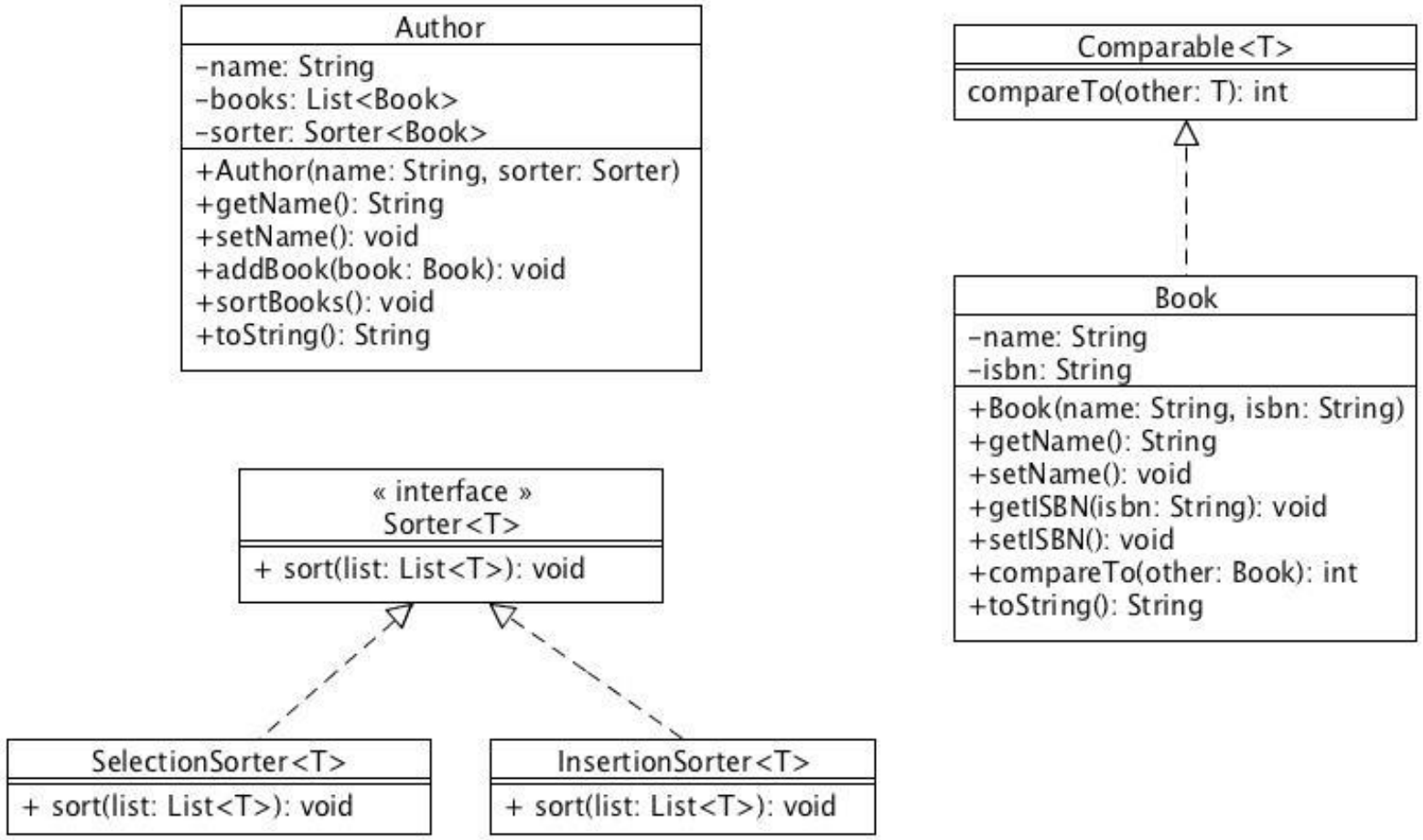
- Problem:
  - multiple classes differ only in how they are implemented
    - the high-level logic is the same except for which algorithm is being used to solve part of the task
  - other classes may also benefit from the code implementing the algorithms, but the code is currently coupled to the class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

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

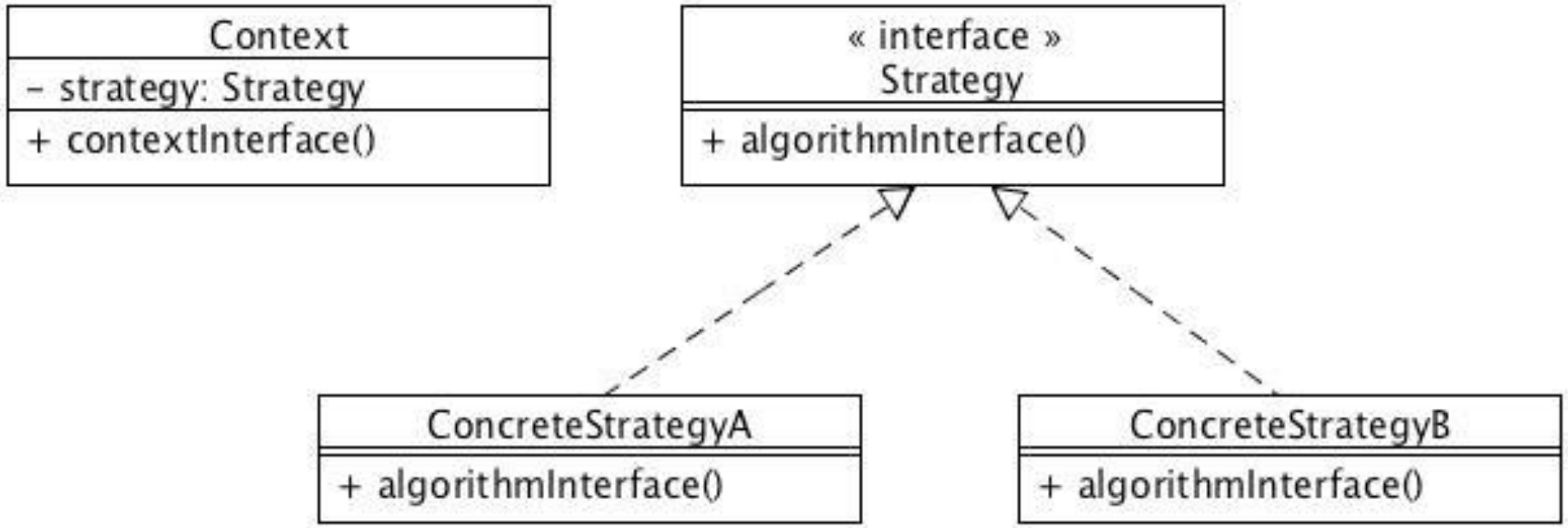
EXAMPLE: WITHOUT THE STRATEGY PATTERN



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: IMPLEMENTATION USING DELEGATION

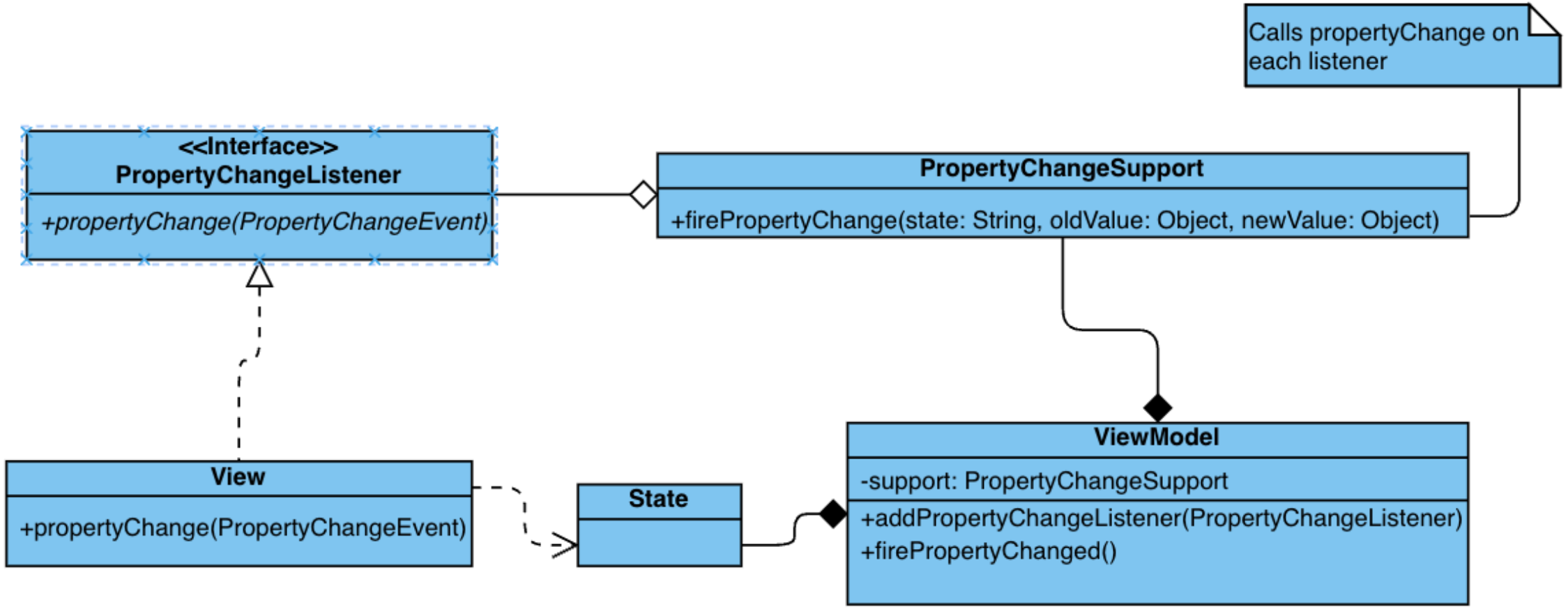


diagram created using https://online.visual-paradigm.com/

STRUCTURAL PATTERNS

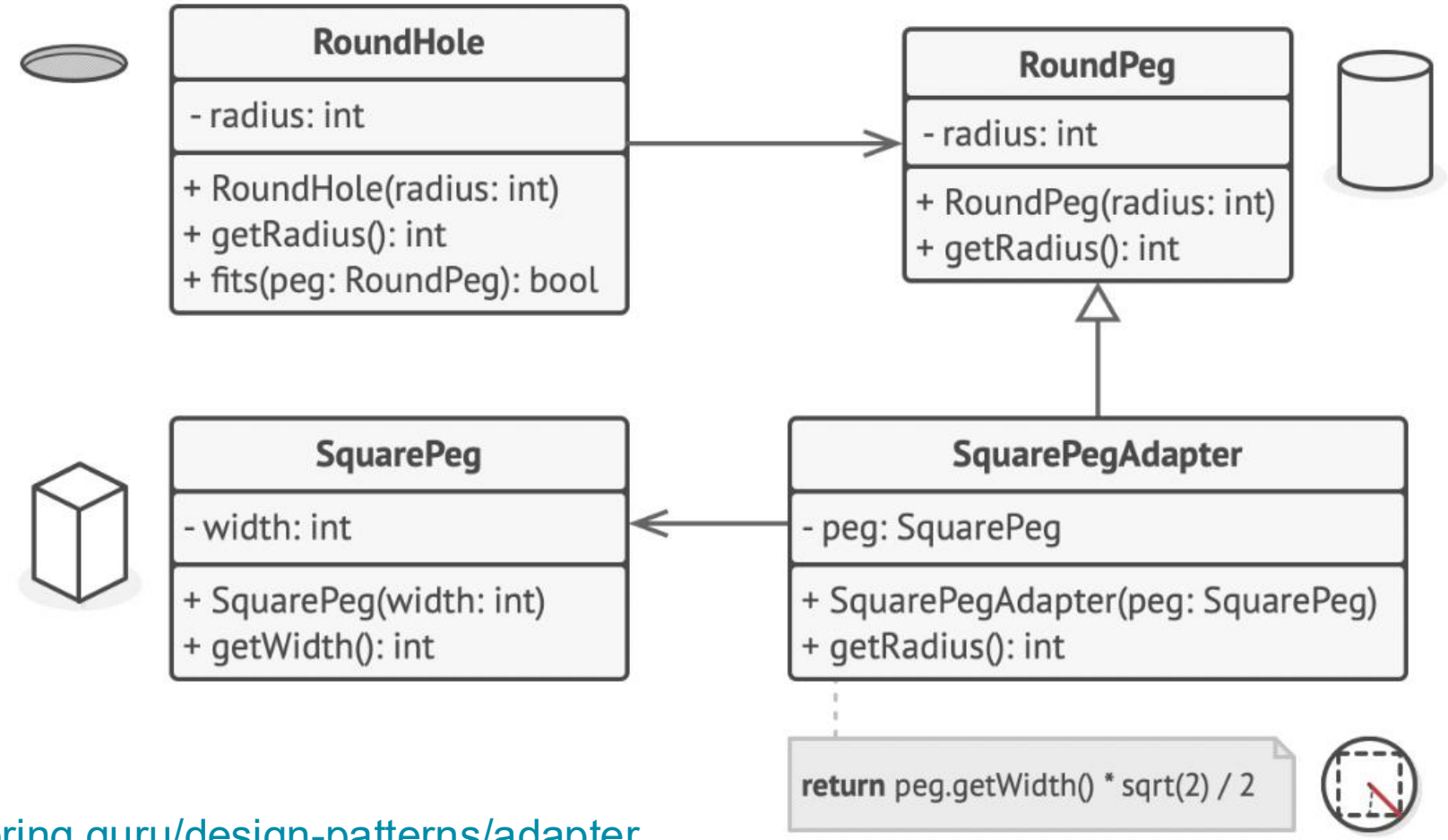
ADAPTER

Computer Science
UNIVERSITY OF TORONTO

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

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)