DESIGN PRINCIPLES AND PATTERNS

06. DESIGN PATTERN 2

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

Content

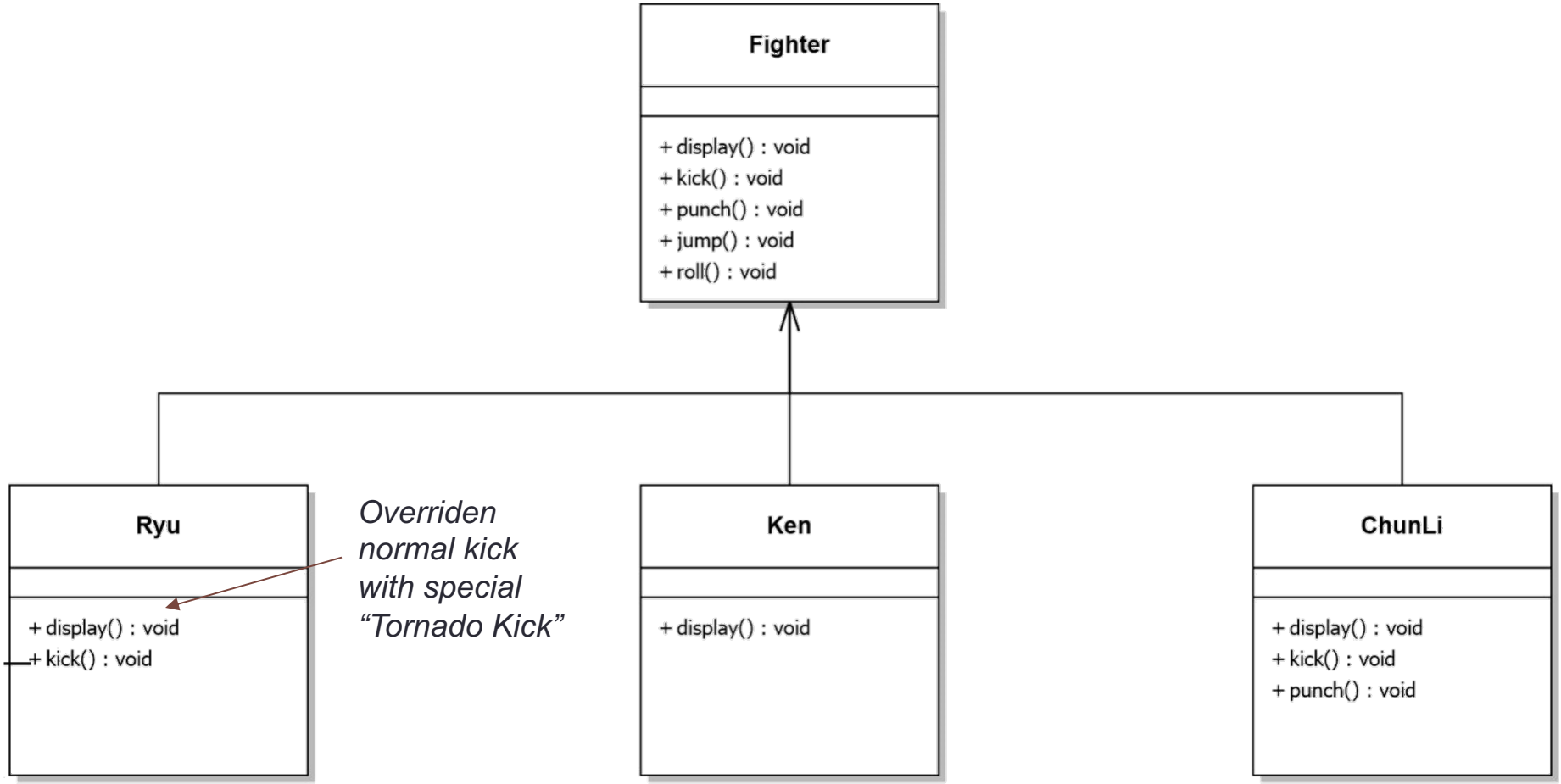
1. Strategy
2. Observer
3. Adapter
4. State

Example 1: Street Fighter Game

- A character has four moves: kick, punch, roll and jump
  - Mandatory: kick and punch moves
  - Optional: roll and jump
- How would you model your classes?
  - Suppose initially you use inheritance and abstract out the common features in a Fighter class and let other characters subclass Fighter class

Street Fighter Game: Subclass

- Any character with specialized move can override that action in its subclass



Street Fighter Game: Subclass (2)

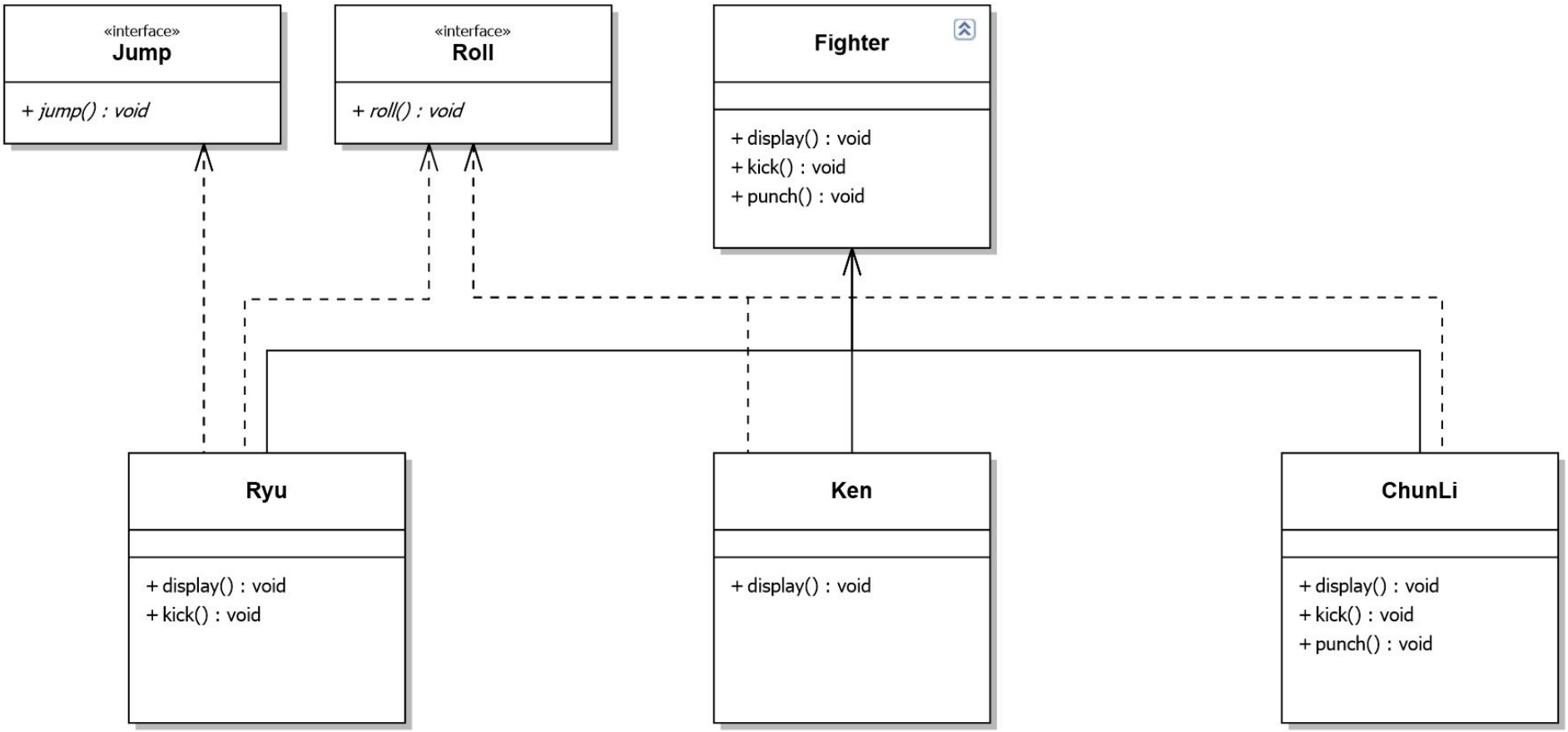
- What if a character doesn't perform jump move?
  - It still inherits the jump behavior from super class
  - Although you can override jump to do nothing in that case but you may have to do so for many existing classes and take care of that for future classes too

➔ Difficult to maintain

➔ Violate LSP

Street Fighter Game: Subclass and Interface

- Put optional behaviors to interfaces

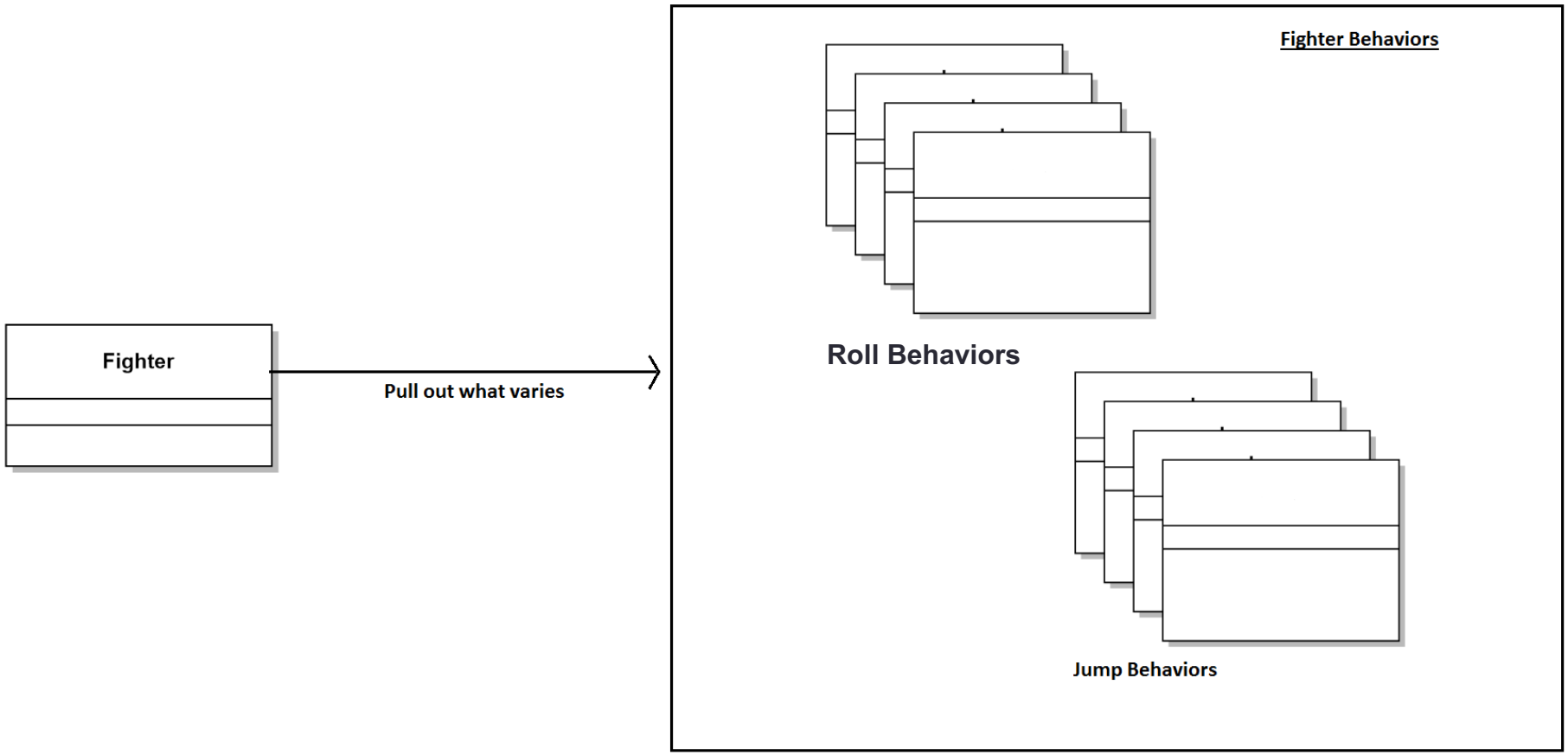


Street Fighter Game: Subclass and Interface (2)

- What if jump and/or roll behaviors are the same for some or all characters?
  - Bad code reusability: Duplication code
  - Multiple inheritance?
    - Inheritance duplication, diamond problem
    - Not support in many languages
- What if the player evolves/wants to change from this character to another one when PLAYING?
  - Nont possible with Inheritance

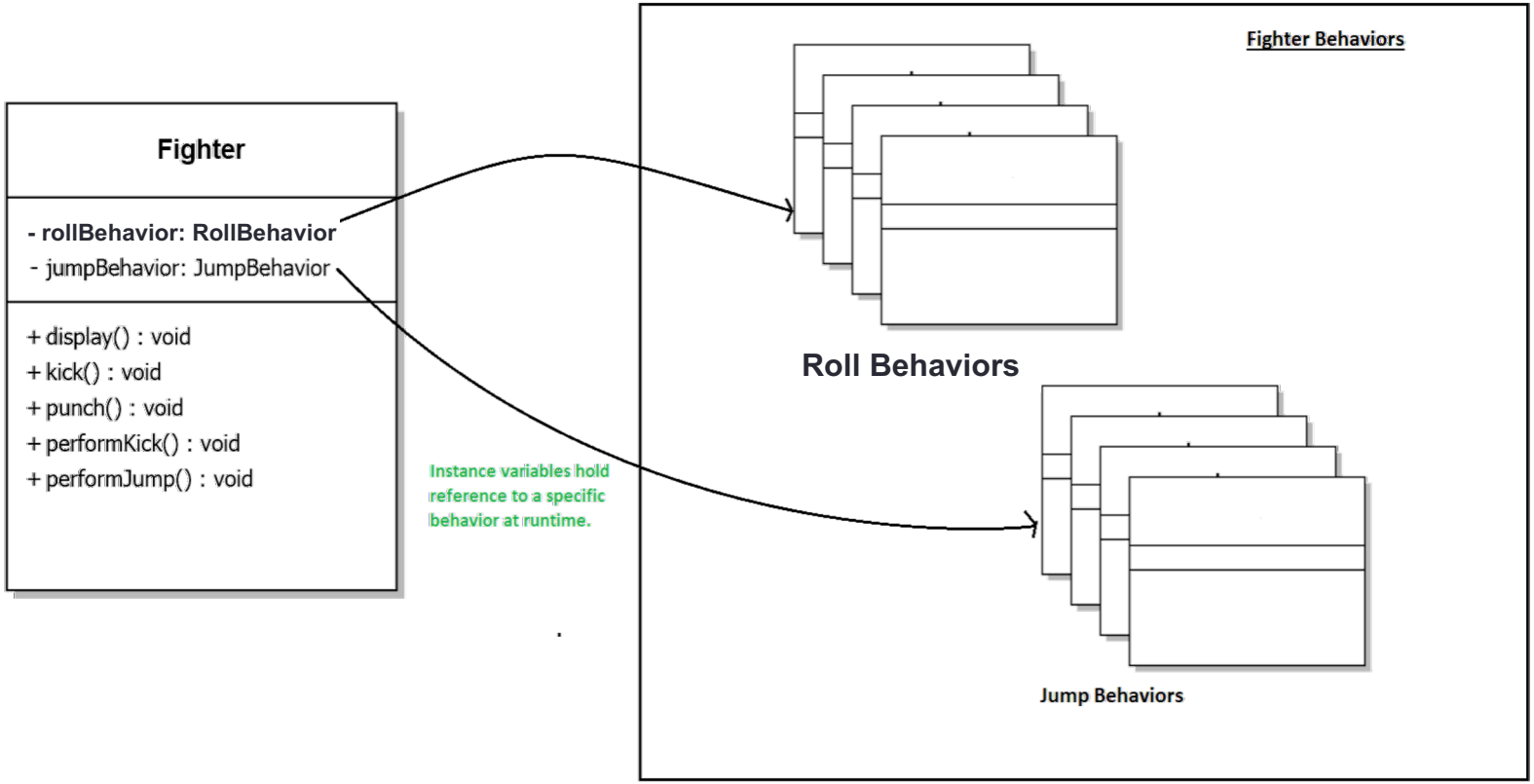
Street Fighter Game: Strategy Pattern

- Put behaviors that may vary across different classes in future and separate them from the rest
  - i.e. jump and roll behaviors

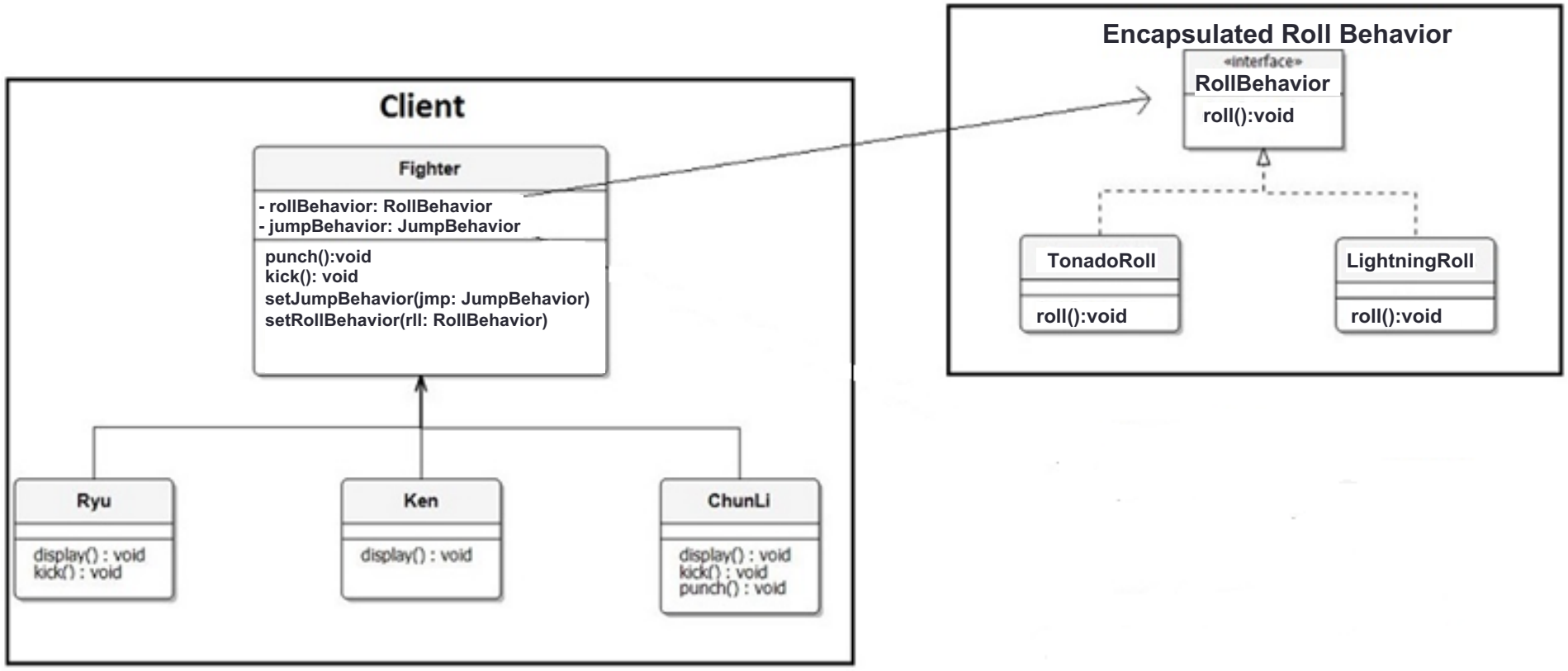


Street Fighter Game: Strategy Pattern (2)

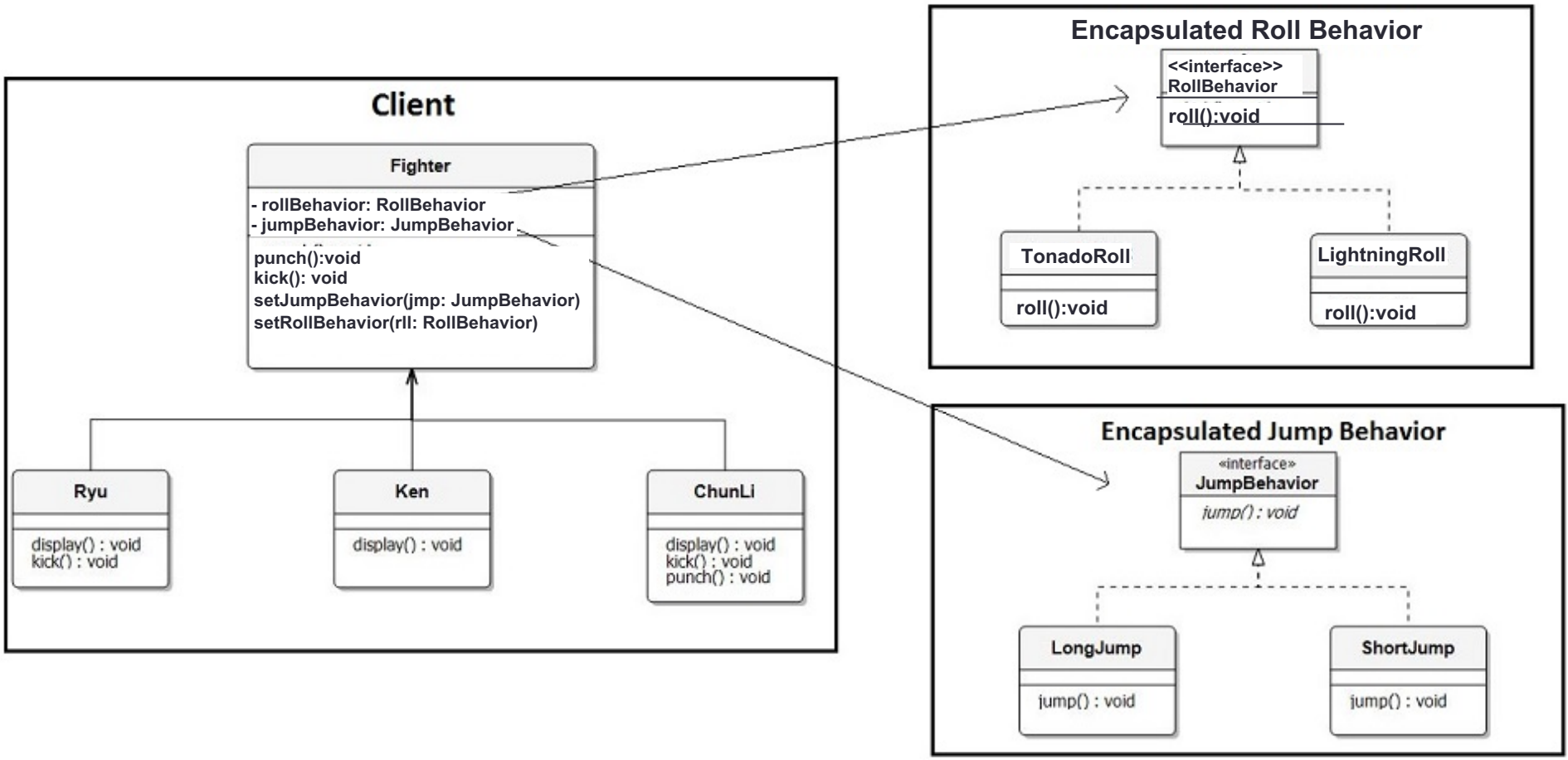
- Fighter delegates its jump and roll behaviors
  - Instead of using jump and roll methods defined in Fighter or its subclass



Street Fighter Game: Strategy Pattern (3)

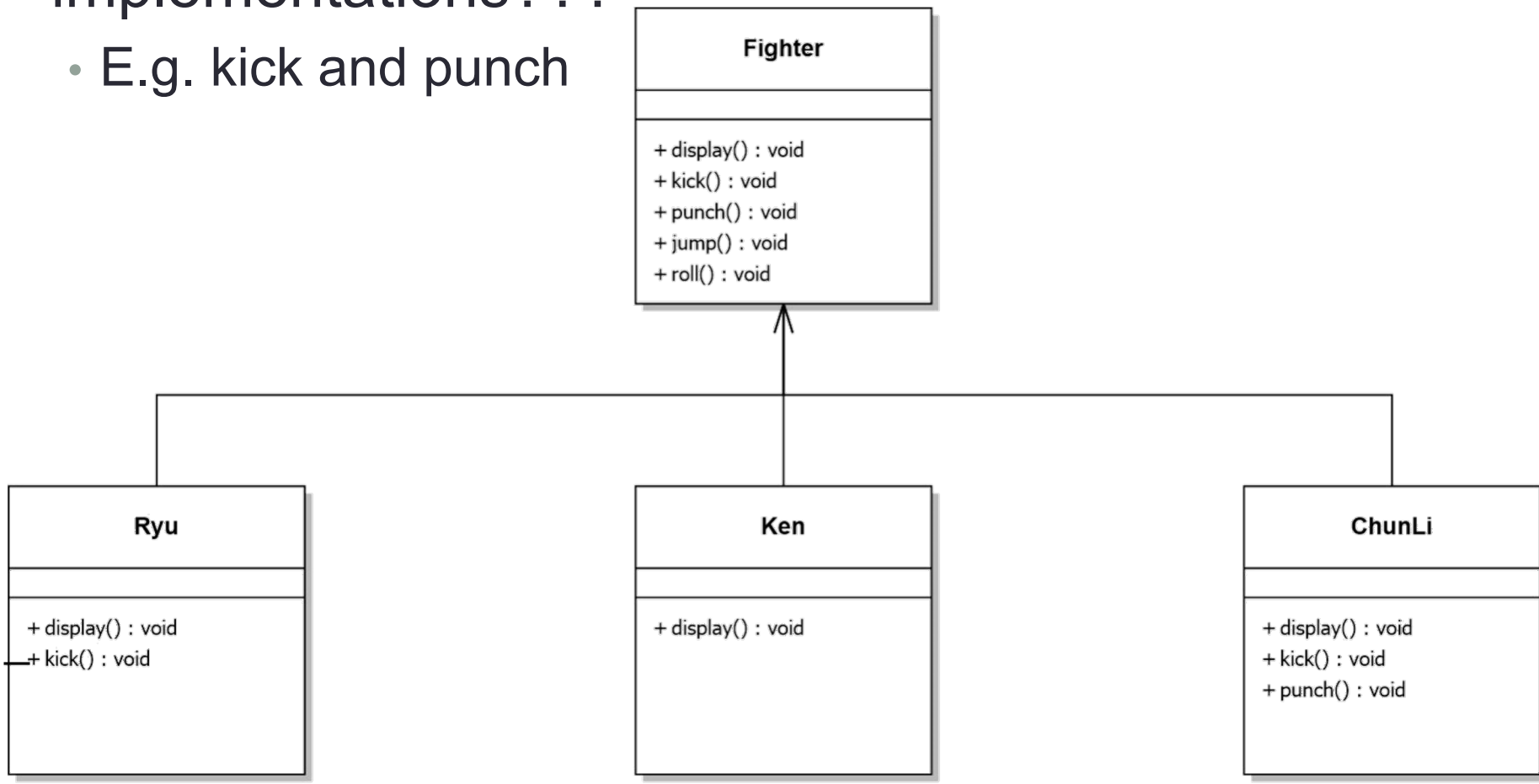


Street Fighter Game: Strategy Pattern (4)

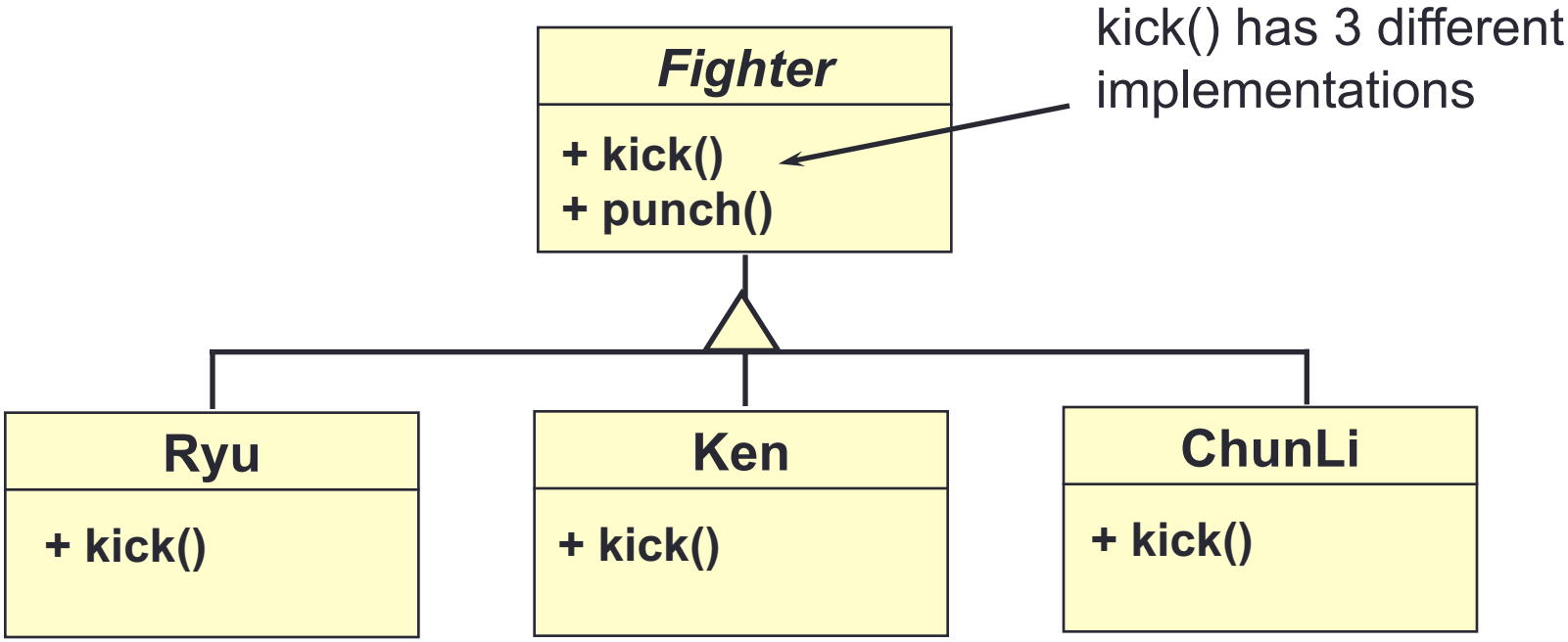


Street Fighter Game: More discussion

- What if mandatory behaviors have different implementations???
  - E.g. kick and punch

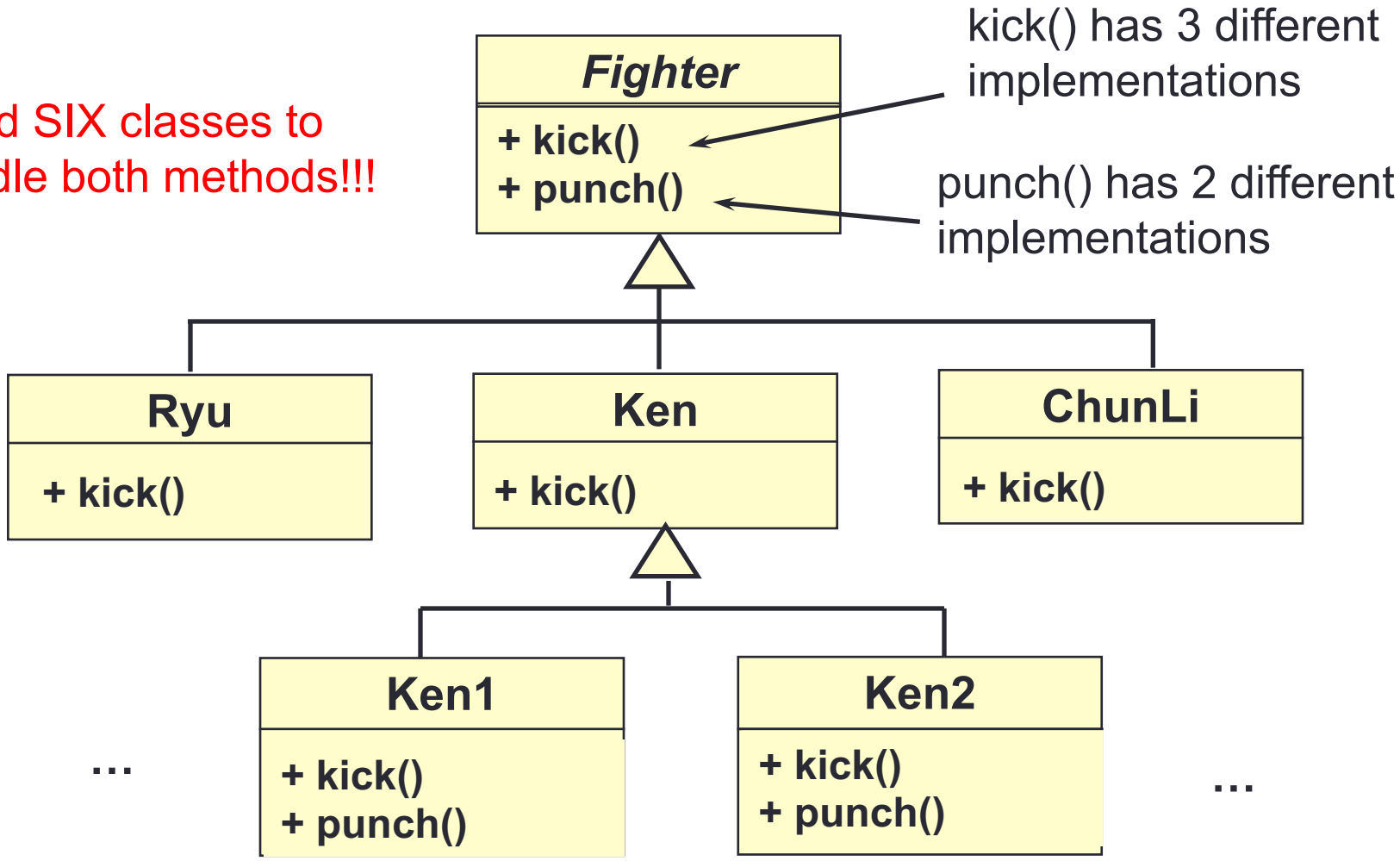


Street Fighter Game: Subclassing

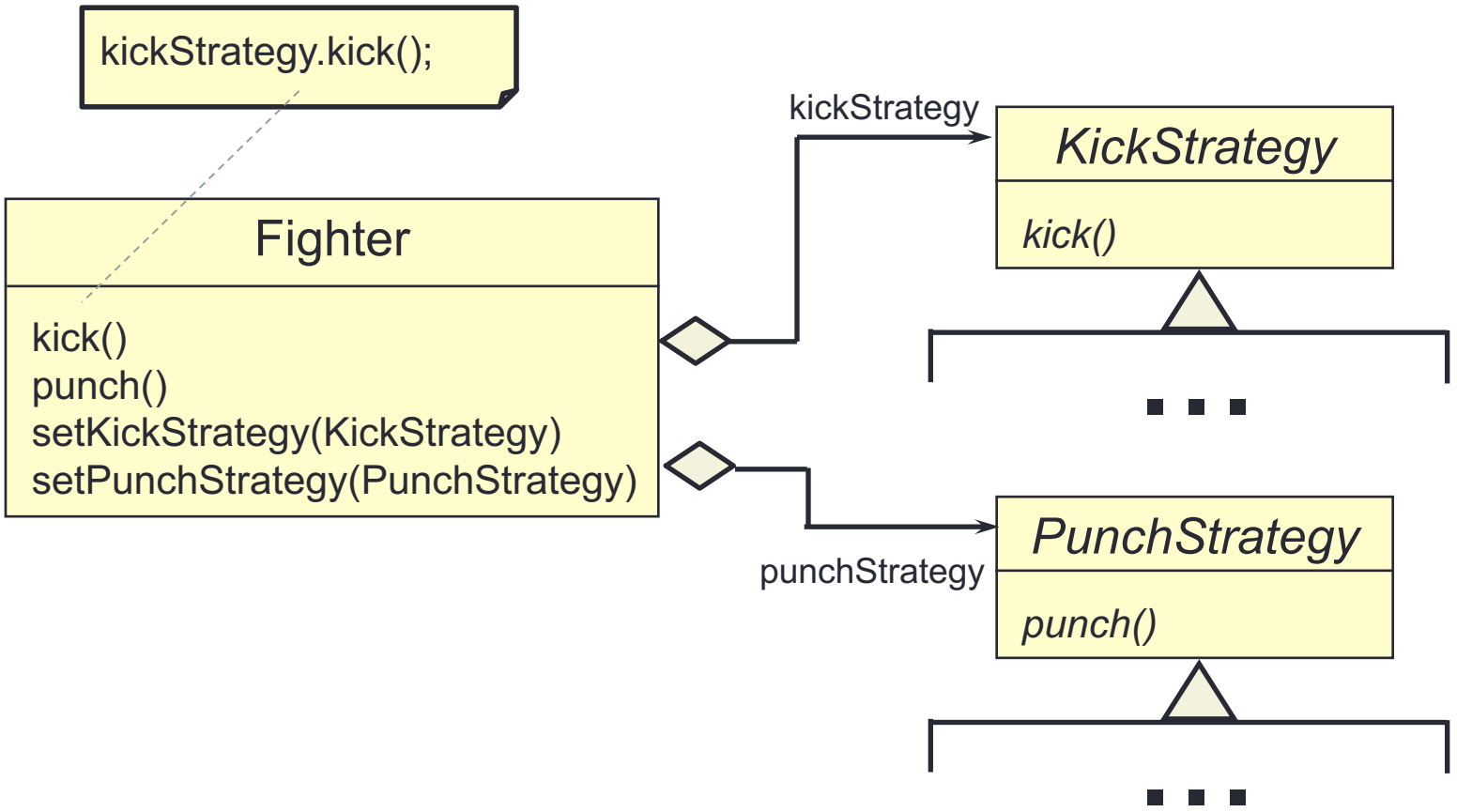


Street Fighter Game: Subclassing

Need SIX classes to handle both methods!!!



Street Fighter Game: Strategy

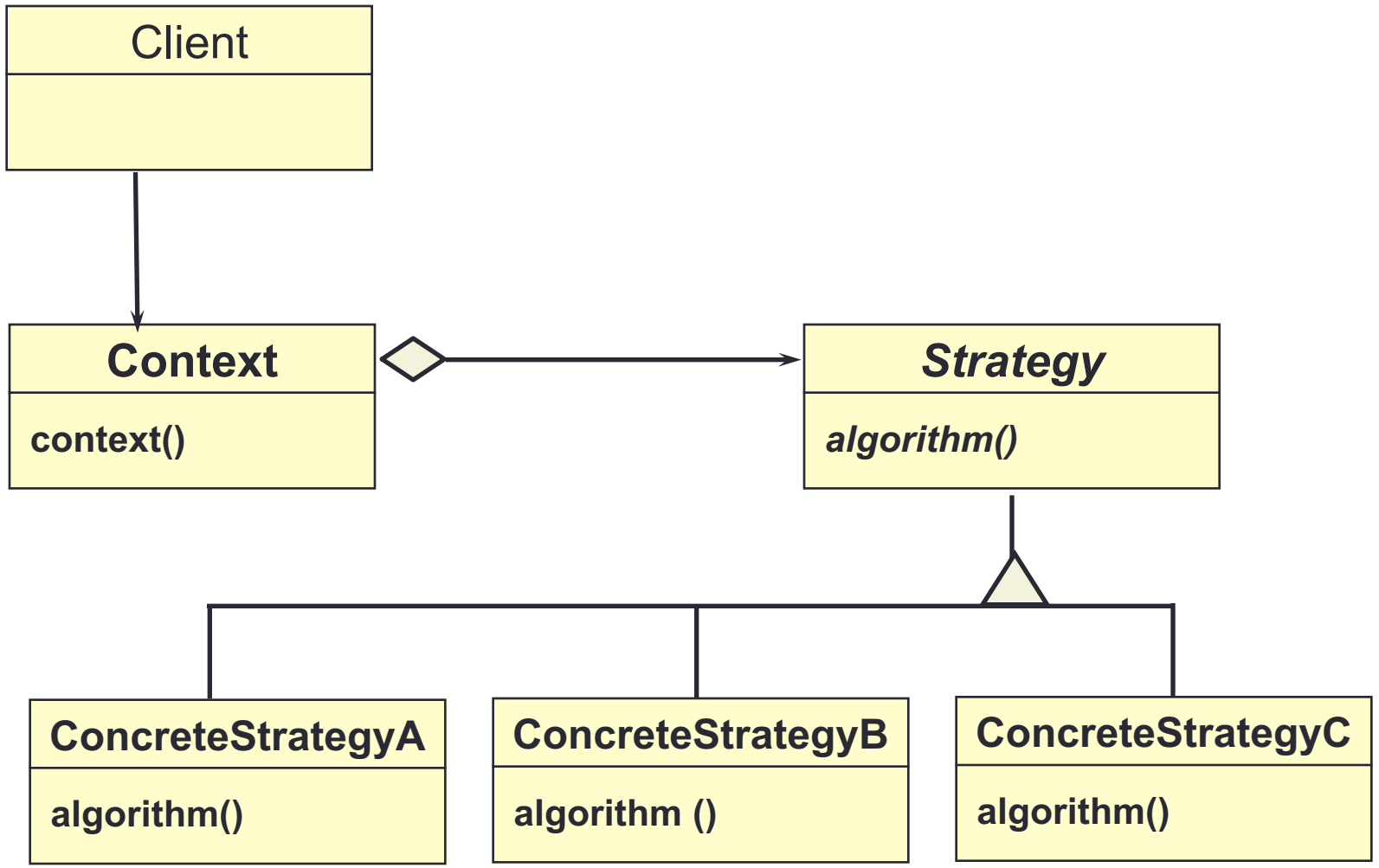


Strategy Design Pattern

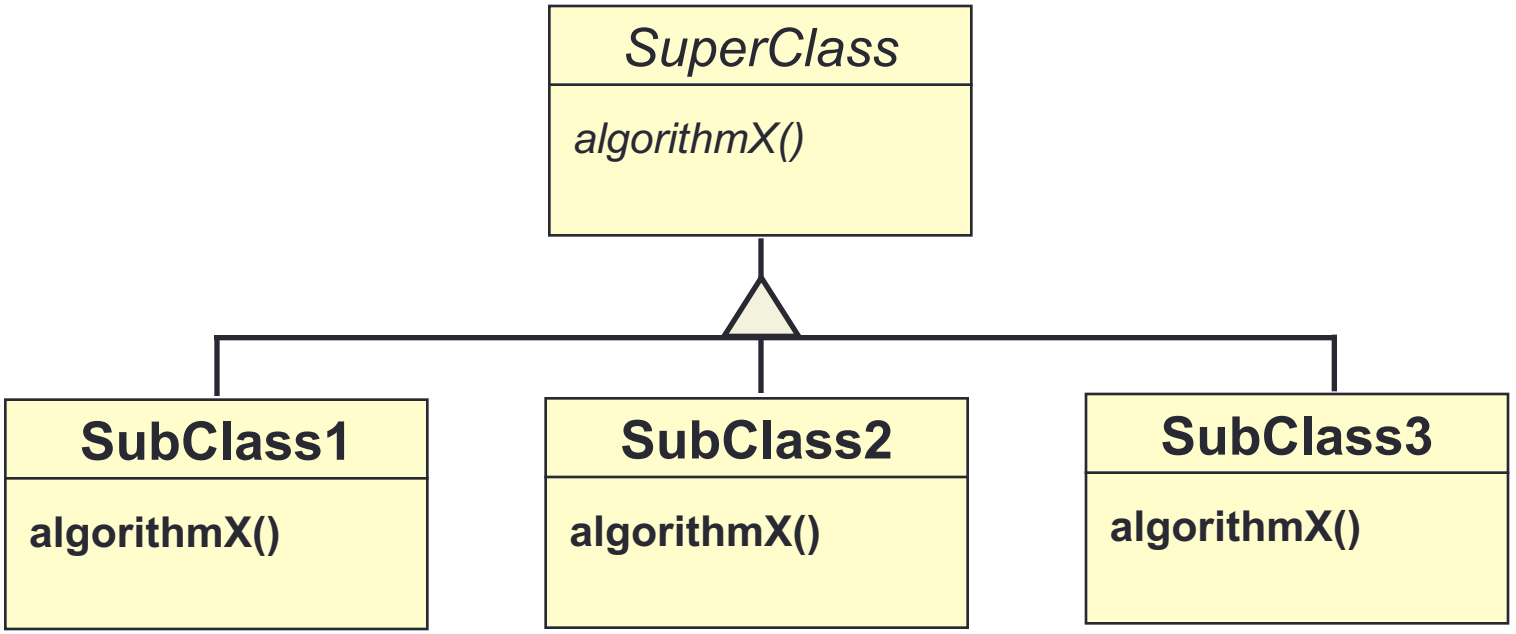
- Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior
  - Separate behaviors that may vary across different classes in future from the rest
- Enables an algorithm's behavior to be selected at runtime
  - defines a family of algorithms,
  - encapsulates each algorithm, and
  - makes the algorithms interchangeable within that family

➔ Favor Composition to Inheritance for Reuse

Strategy Pattern: Structure

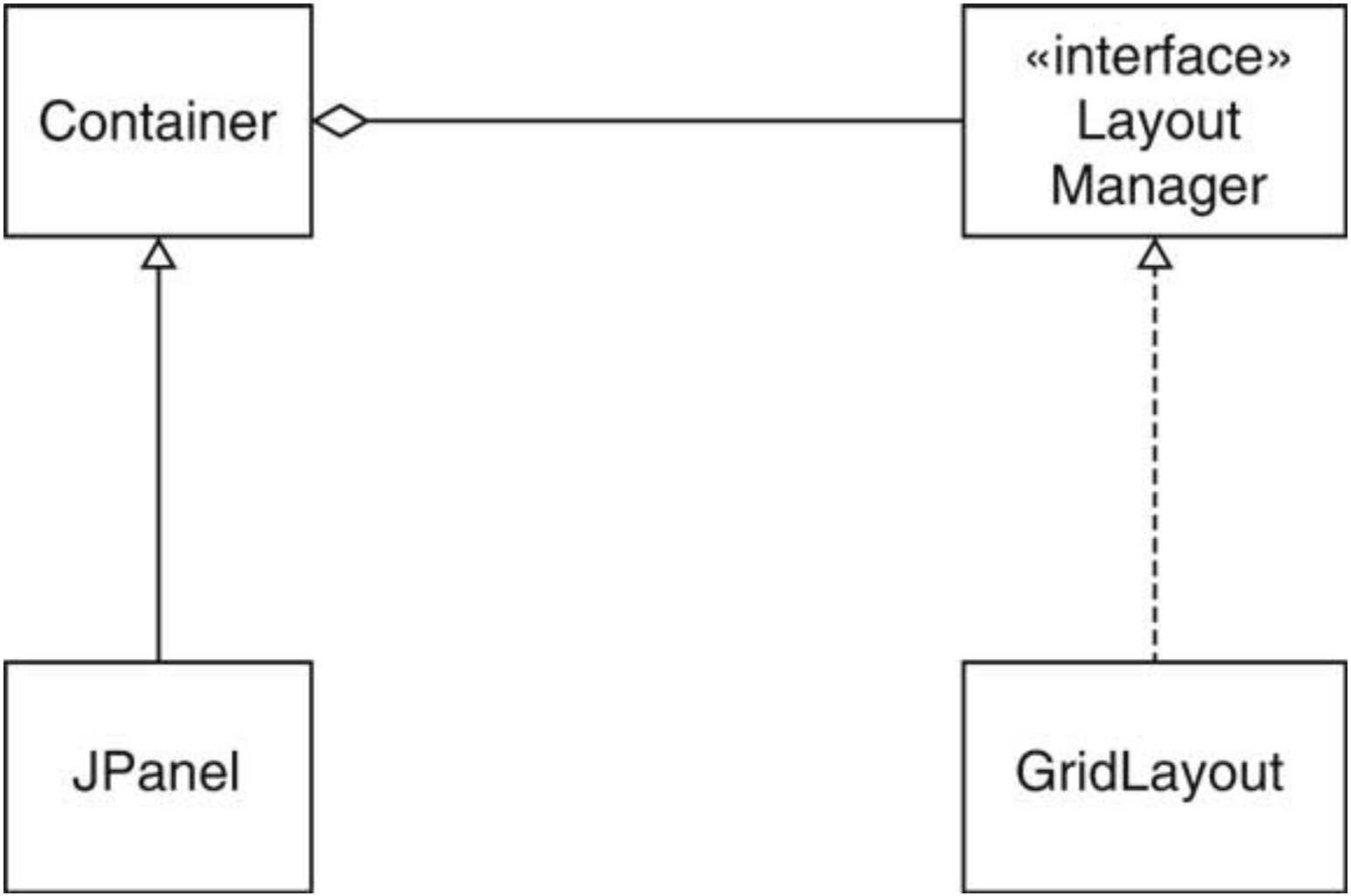


Subclassing vs Strategy: More discussion

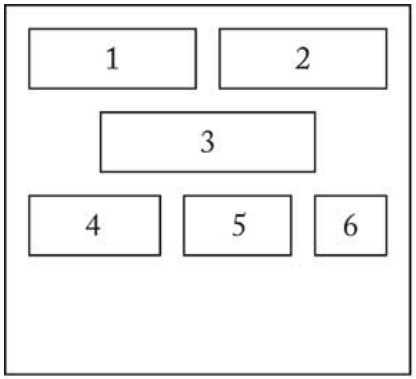


What if SubClass1 needs to change to SubClass2 at runtime?

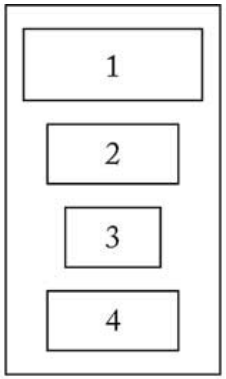
Strategy: Layout Managers



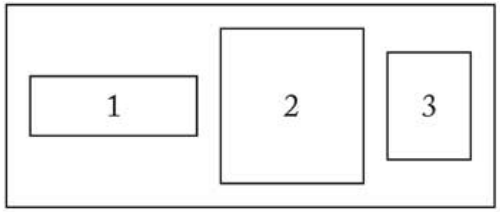
Strategy: Layout Managers



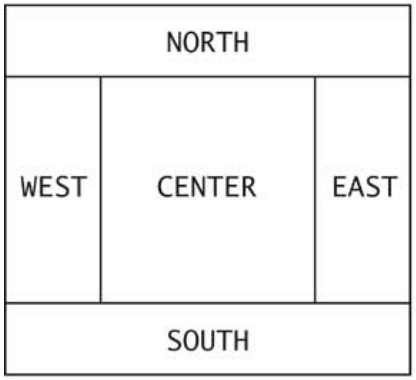
FlowLayout



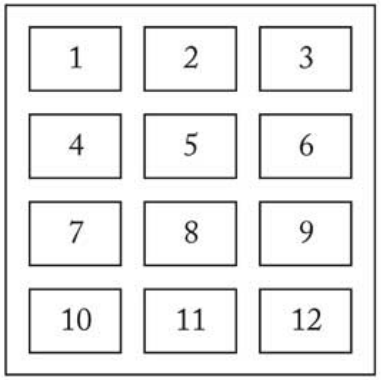
BoxLayout (vertical)



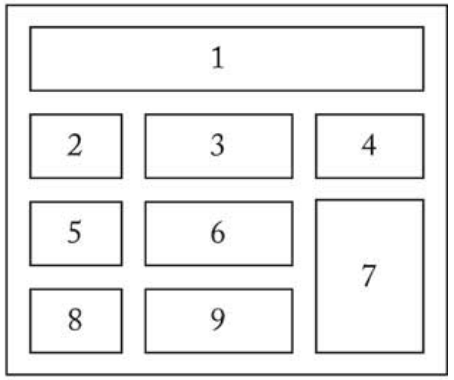
BoxLayout (horizontal)



BorderLayout

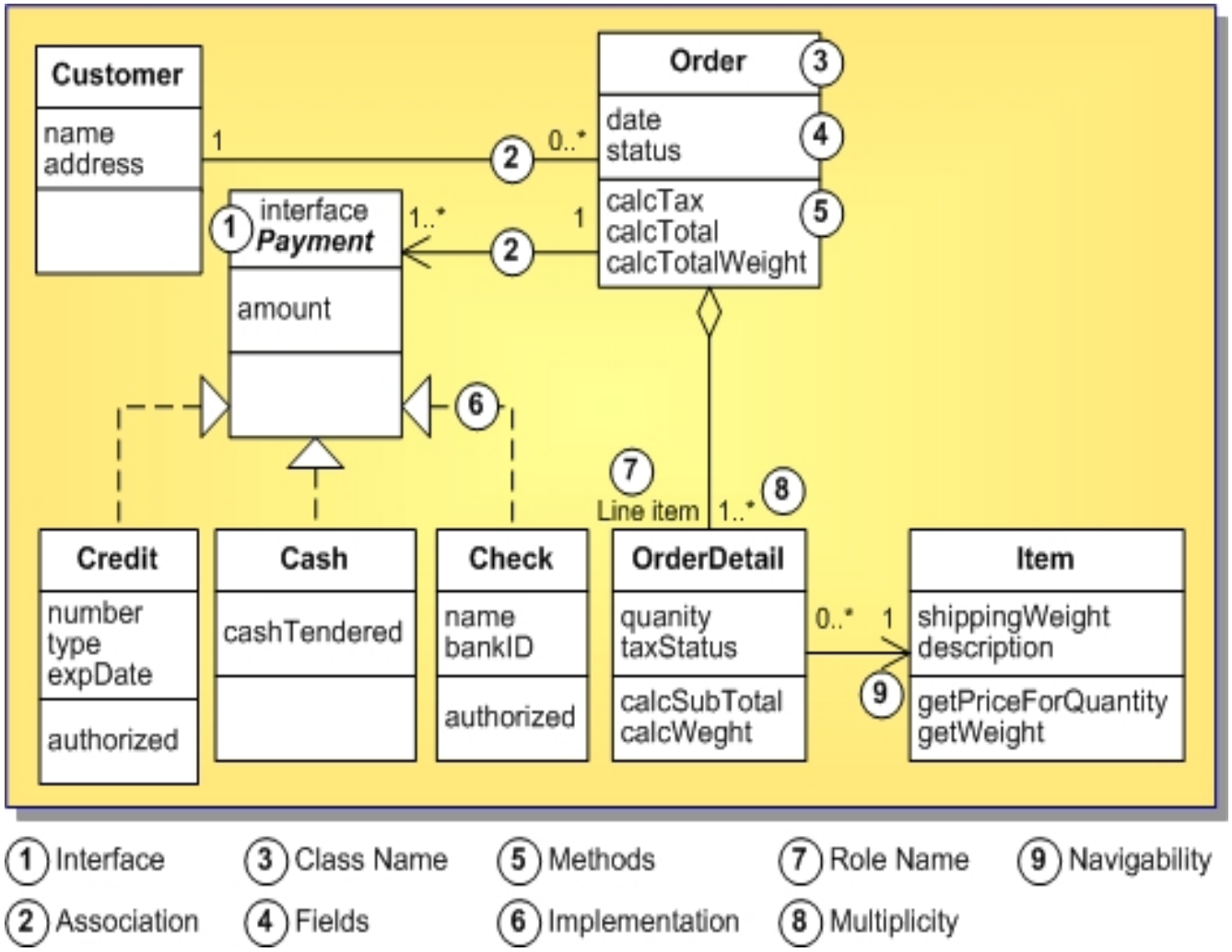


GridLayout

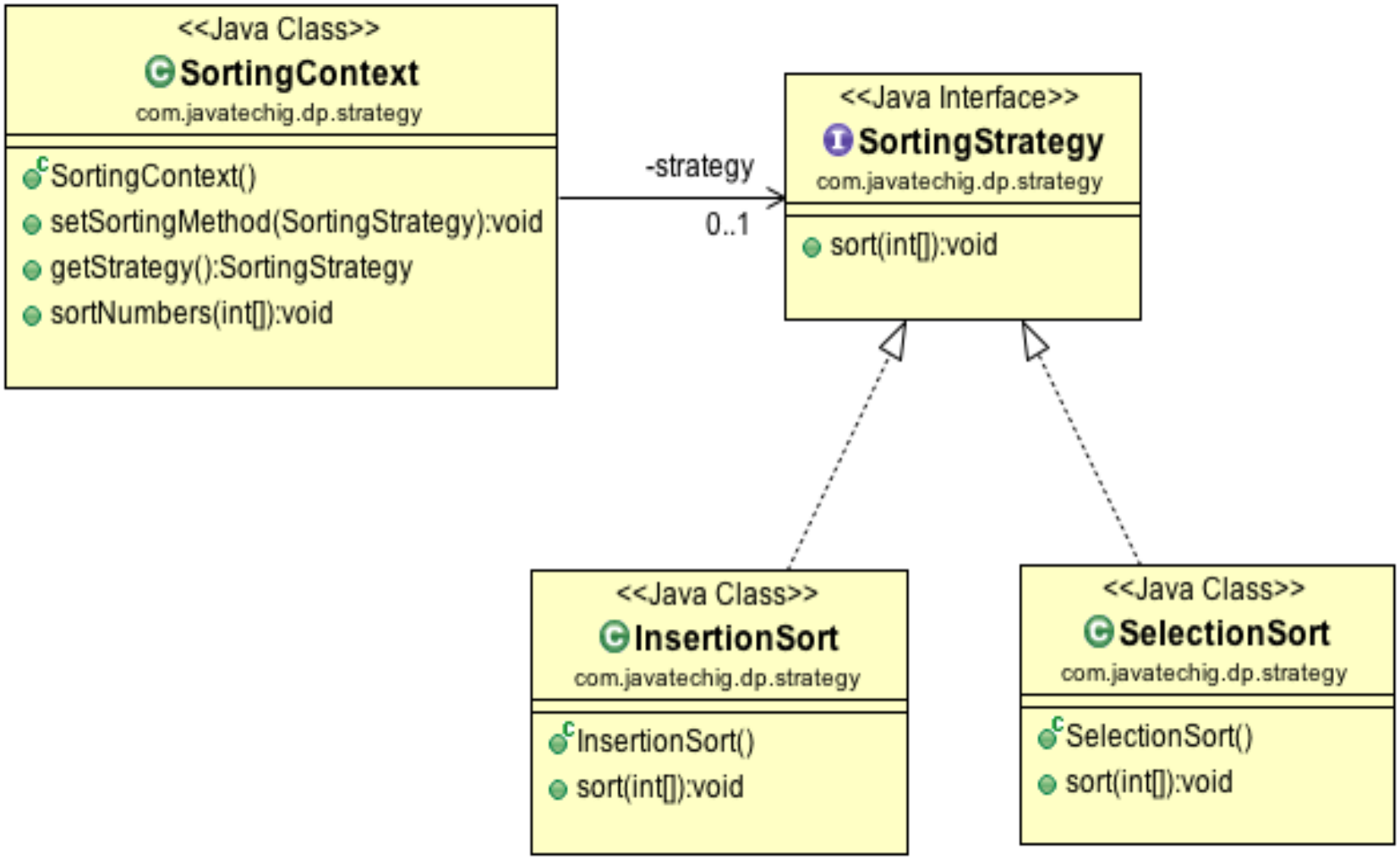


GridBagLayout

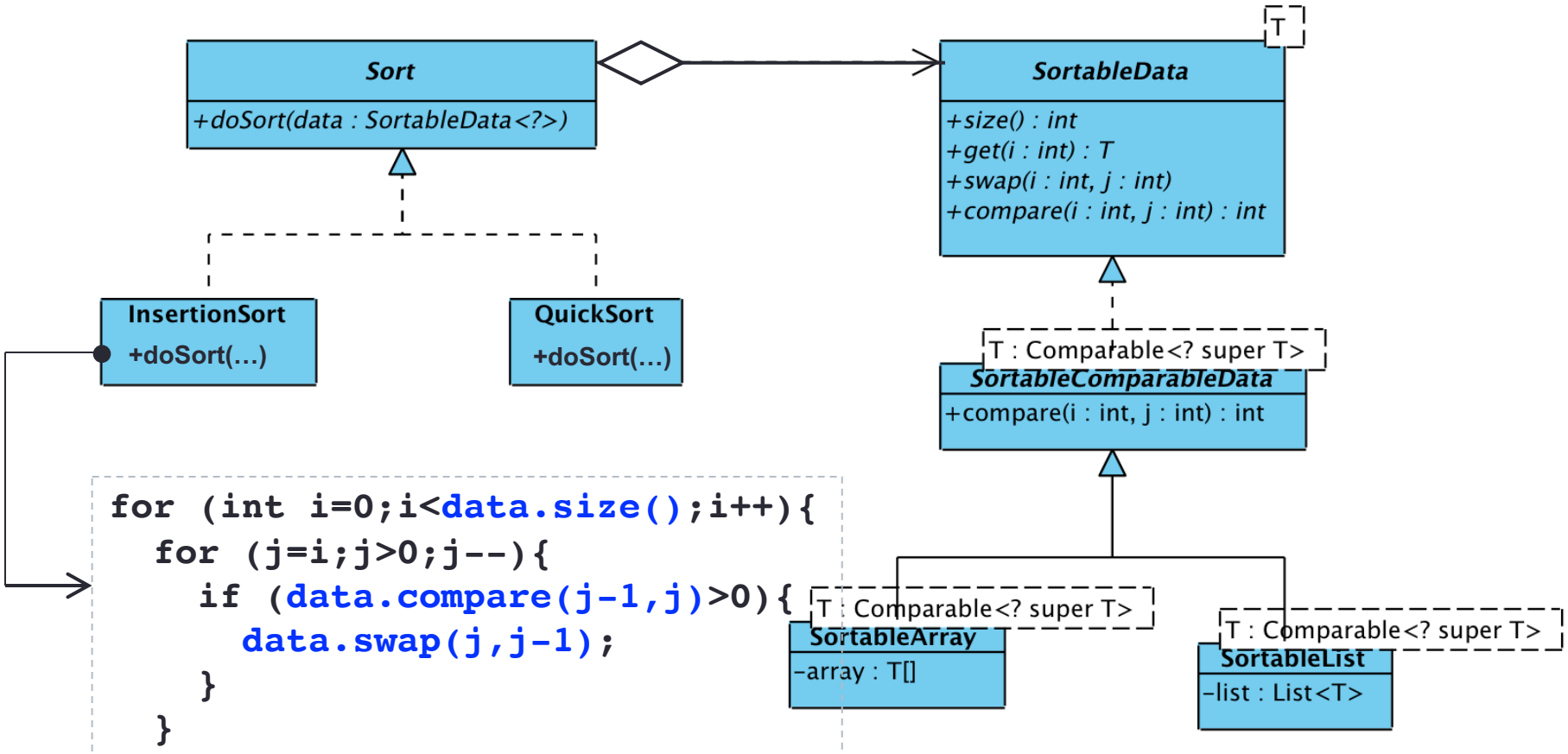
Strategy: Payment Method



Strategy: Sorting Algorithms



Template Method and Strategy Pattern

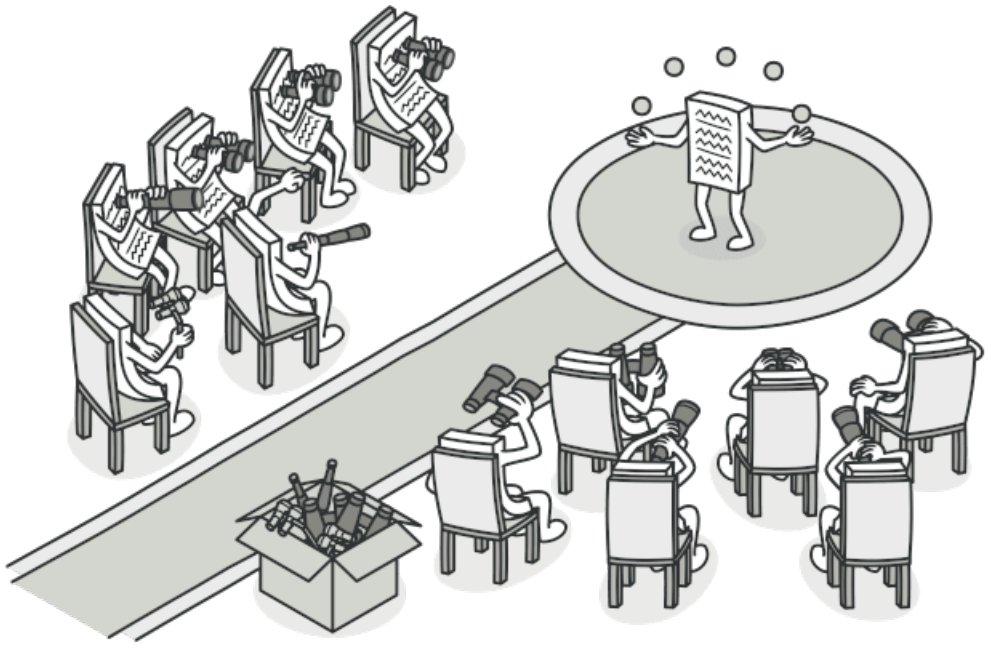


Practice: Applying Strategy in Codebase

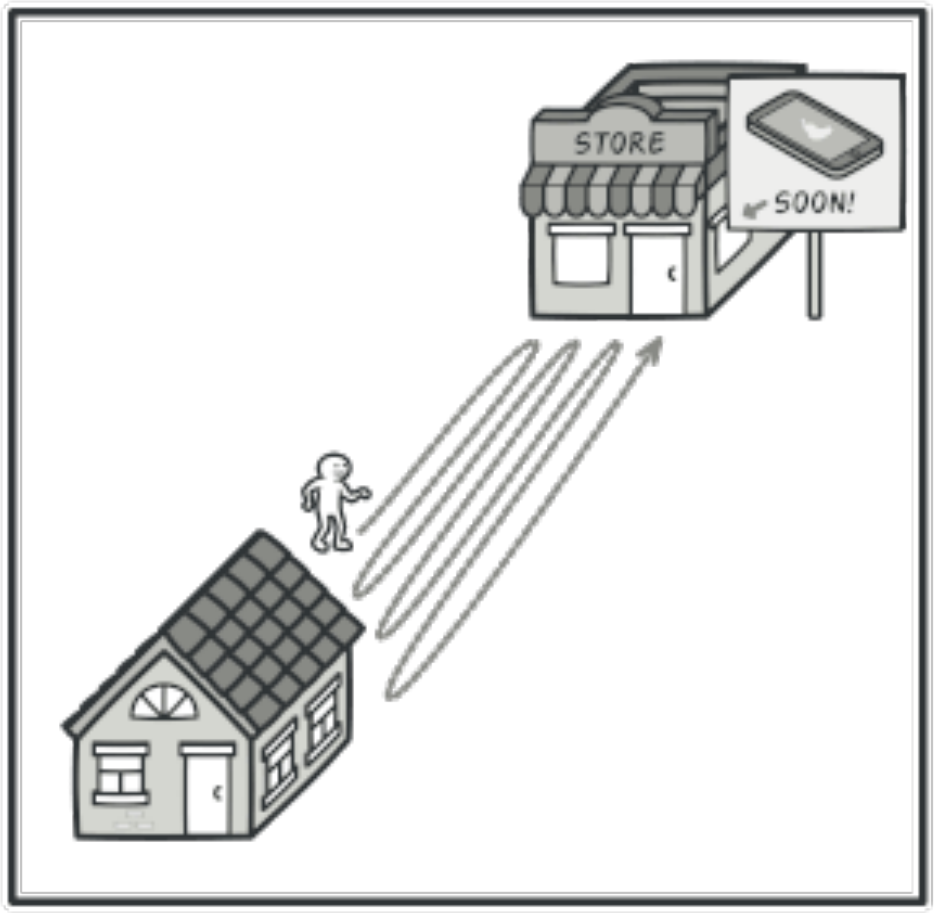
- Can we apply Strategy in any part of the codebase for a specific requirement/a better design?

Content

1. Strategy
2. Observer
3. Adapter
4. State

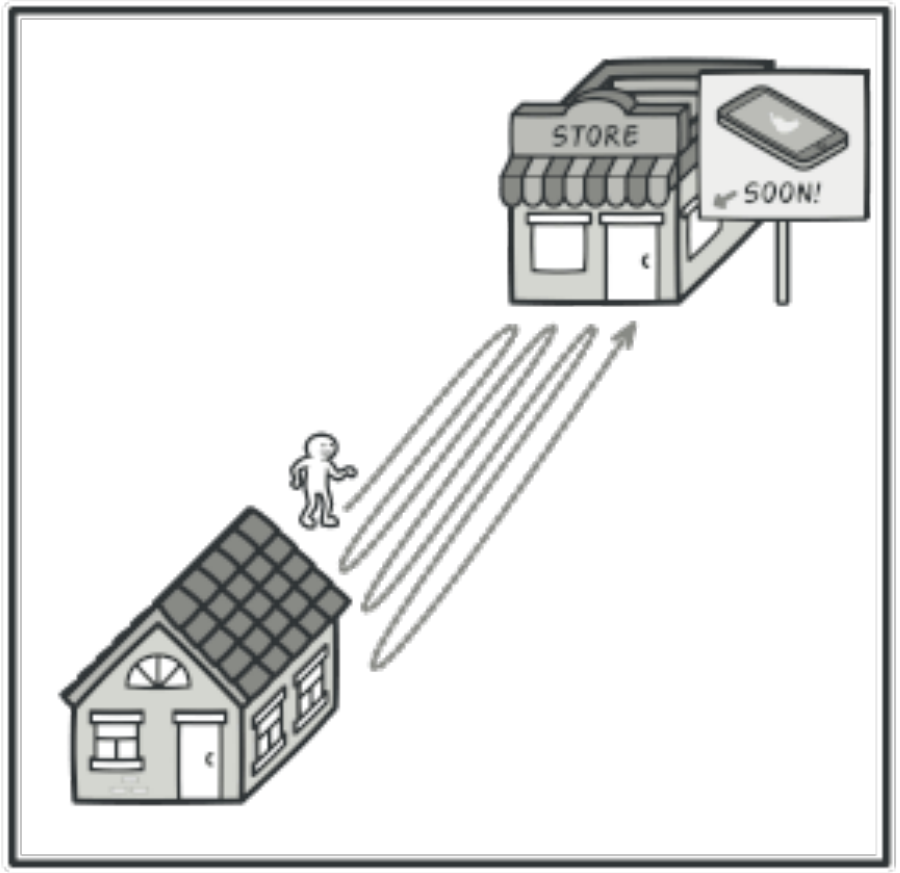


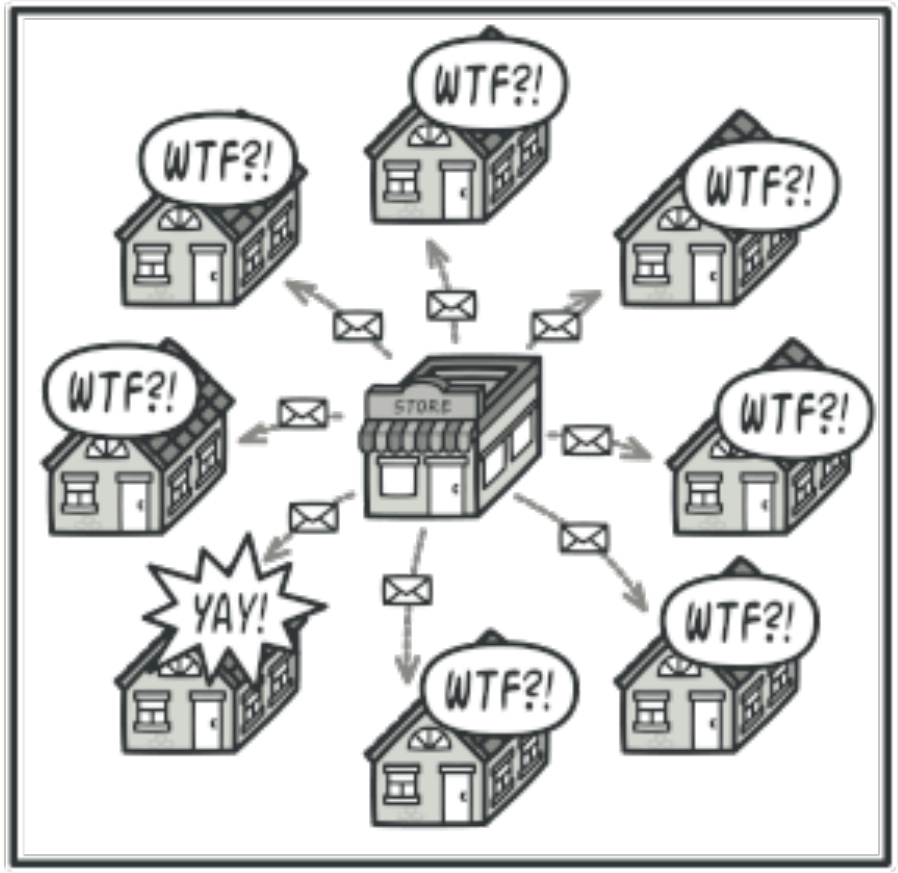
New product in the Store



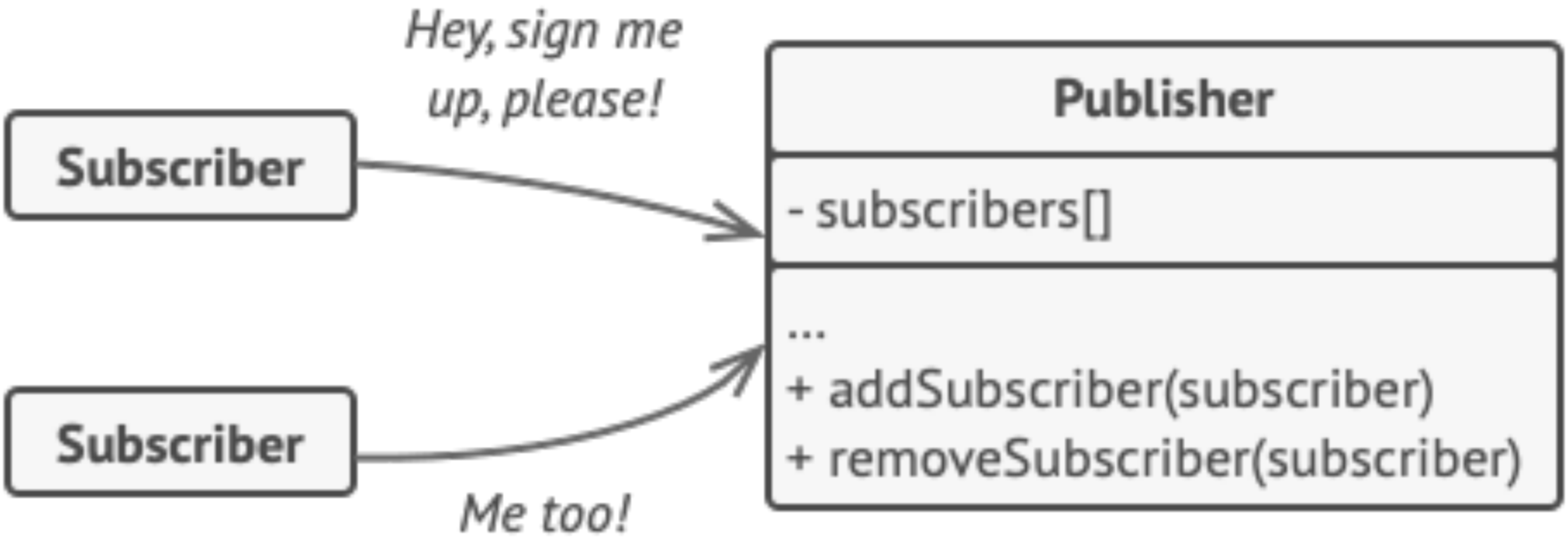
Continuously checking if the store has new products or not

New product in the Store – Send to all

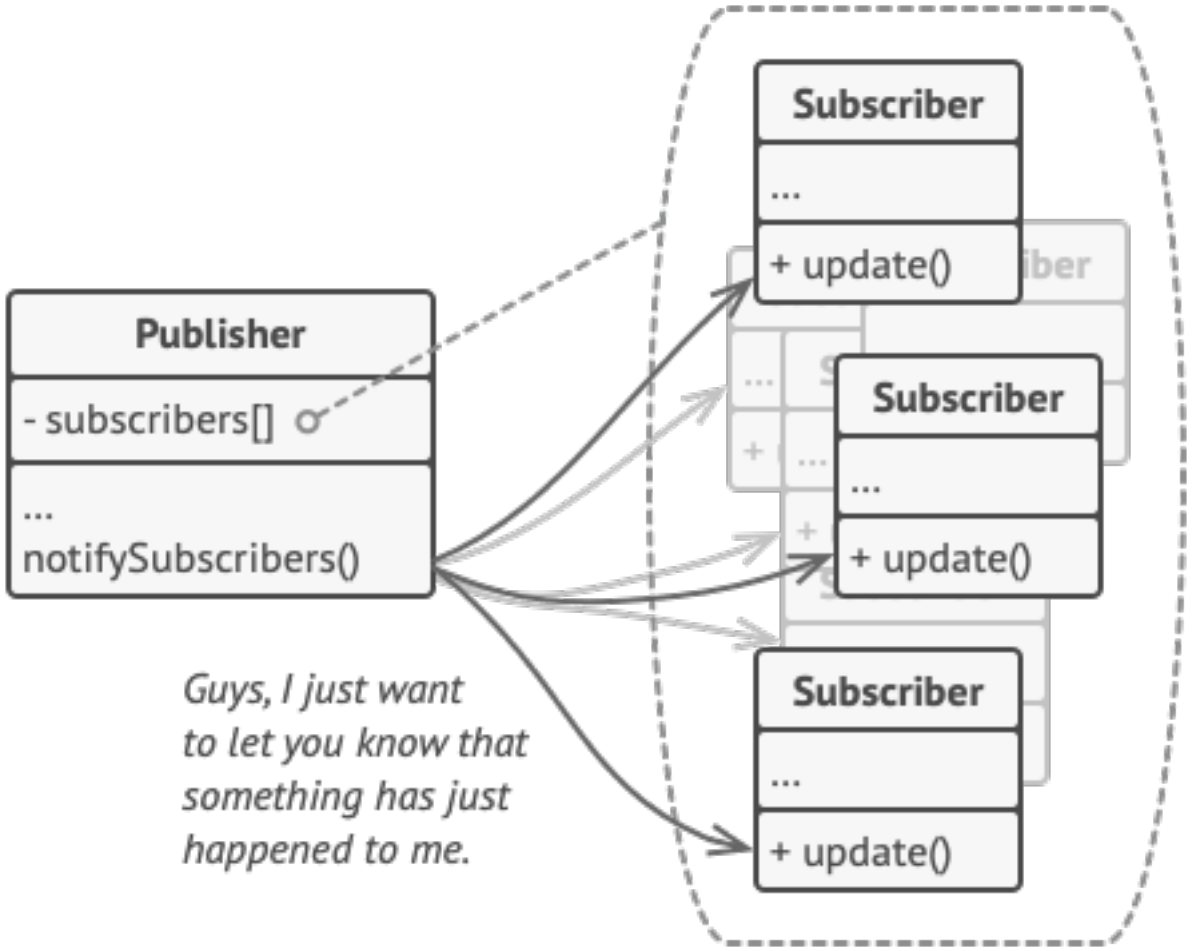




Observer: Subscription Model



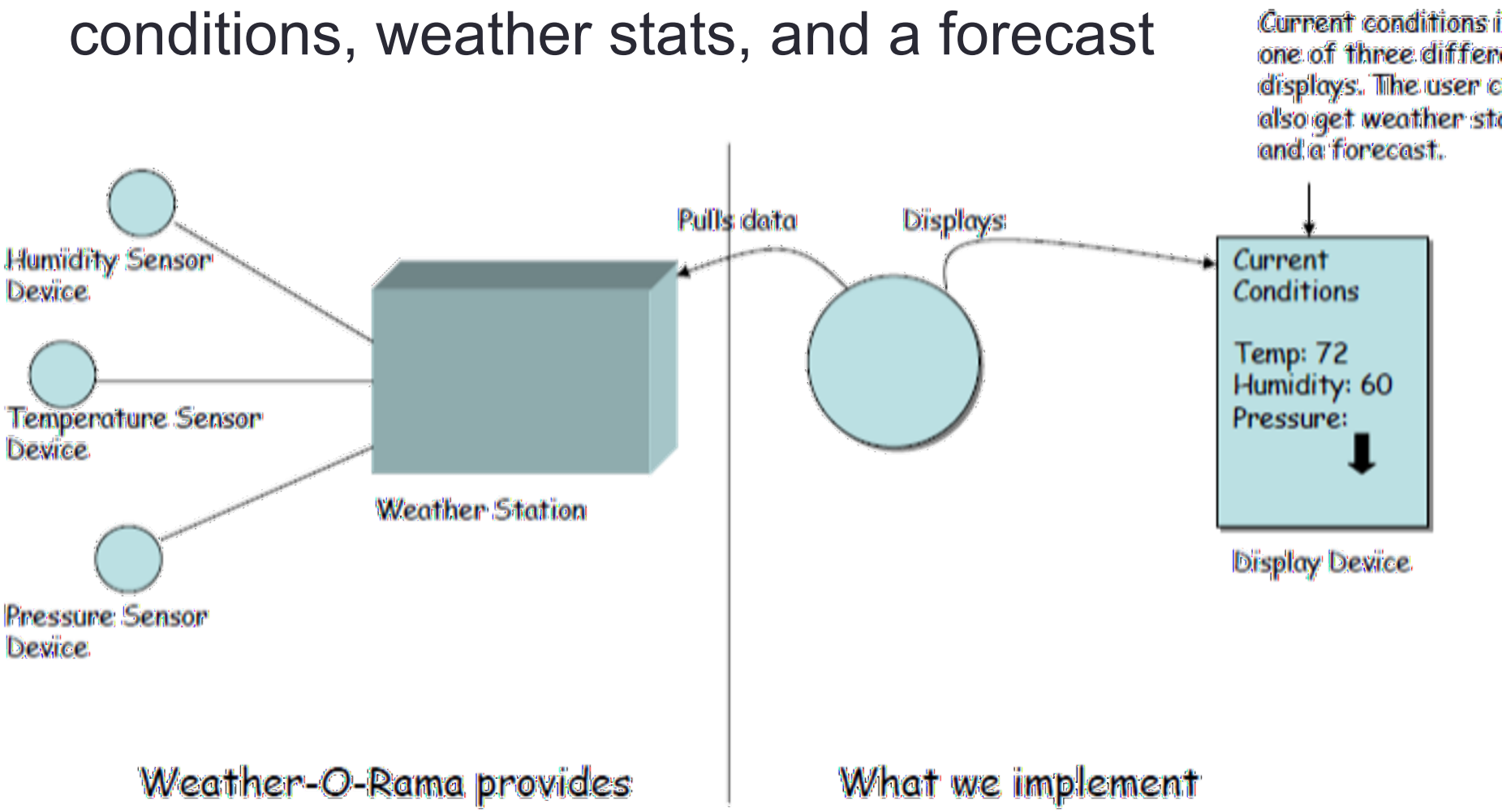
Observer: Notification to Subscribers



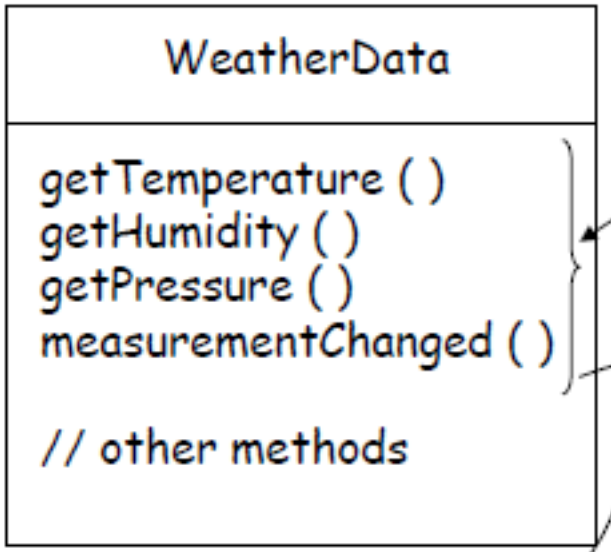
Magazine Subscription Model

The Weather-O-Rama: More Example

- WeatherData has update three displays for currents conditions, weather stats, and a forecast



WeatherData Class: Idea



These three methods return the most recent weather measurements for temperature, humidity, and pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated information from the Weather Station

A clue: what we need to add!

```
/*
 * This method gets called whenever the
 * measurements have been updated.
 */
public void measurementsChanged ( ){
      // Your code goes here
}
```

WeatherData Class: First version

```
public class WeatherData {
    // instance variable declarations
    public void measurementsChanged ( ) {
        float temp = getTemperature ( );
        float humidity = getHumidity ( );
        float pressure = getPressure ( );

        currentConditionsDisplay.update (temp, humidity, pressure);
        statisticsDisplay.update (temp, humidity, pressure);
        forecastDisplay.update (temp, humidity, pressure);
    }
    // other WeatherData methods here
}
```

Grab the most recent measurements by calling the WeatherData's getter methods (already implemented)

Now update the displays.

Call each display element to update its display, passing it the most recent measurements.

What's Wrong with the First version?

```
public class WeatherData {
    // instance variable declarations
    public void measurementsChanged ( ) {
        float temp = getTemperature ( );
        float humidity = getHumidity ( );
        float pressure = getPressure ( );

        currentConditionsDisplay.update (temp, humidity, pressure);
        statisticsDisplay.update (temp, humidity, pressure);
        forecastDisplay.update (temp, humidity, pressure);
    }
    // other WeatherData methods here
}
```

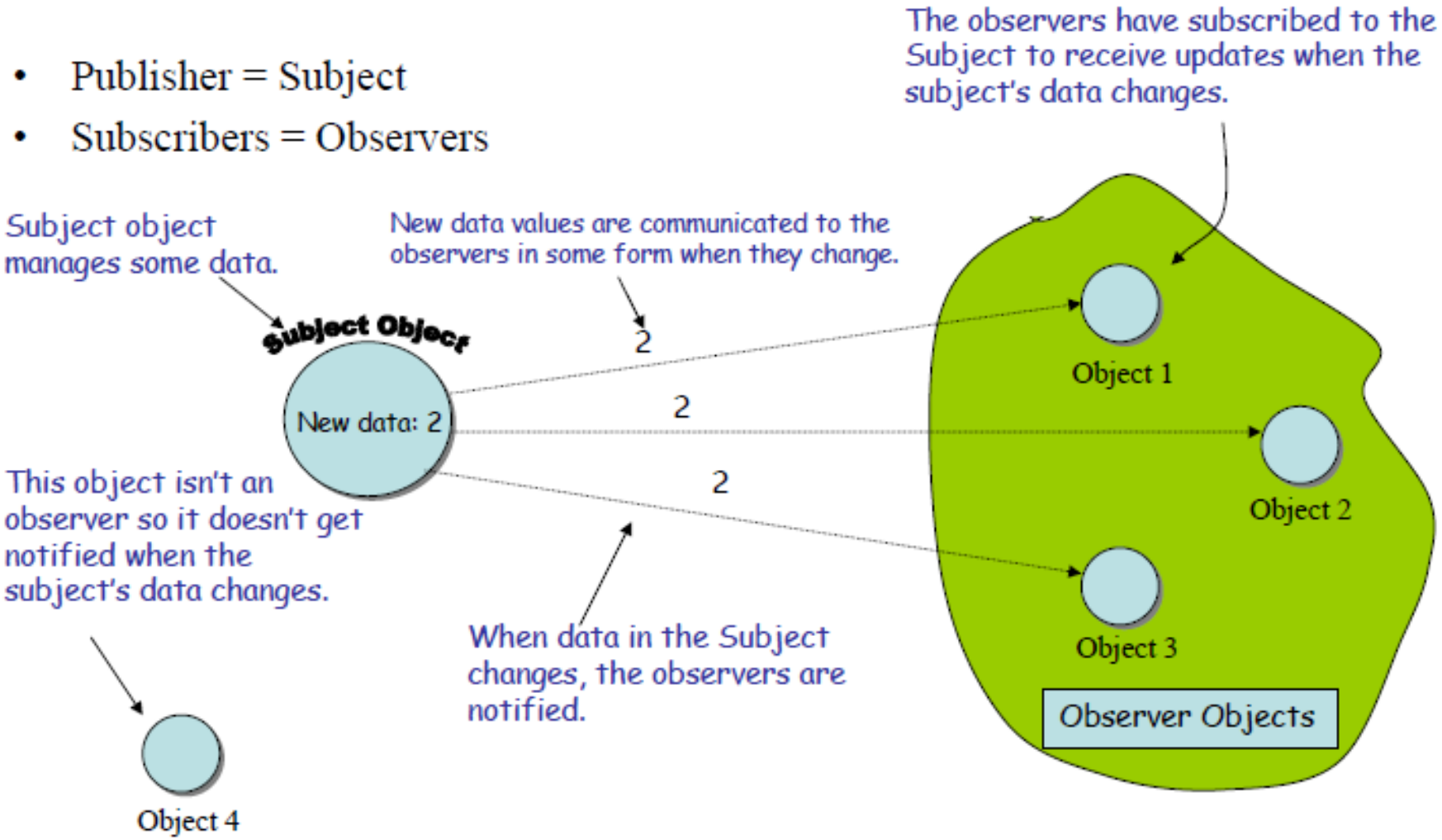
Area of change, we need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

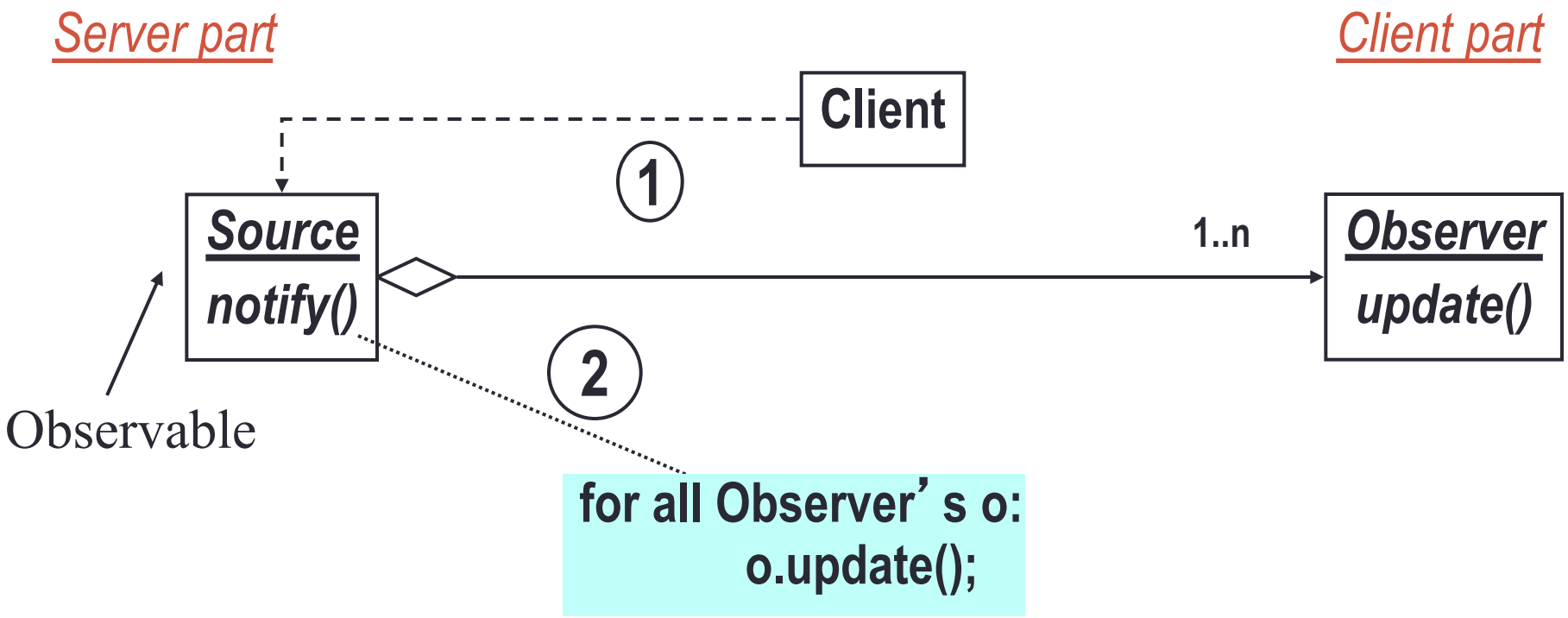
At least we seem to be using a common interface to talk to the display elements...they all have an update ( ) method that takes temp, humidity and pressure values.

Publishers + Subscribers = Observer Pattern

- Publisher = Subject
- Subscribers = Observers



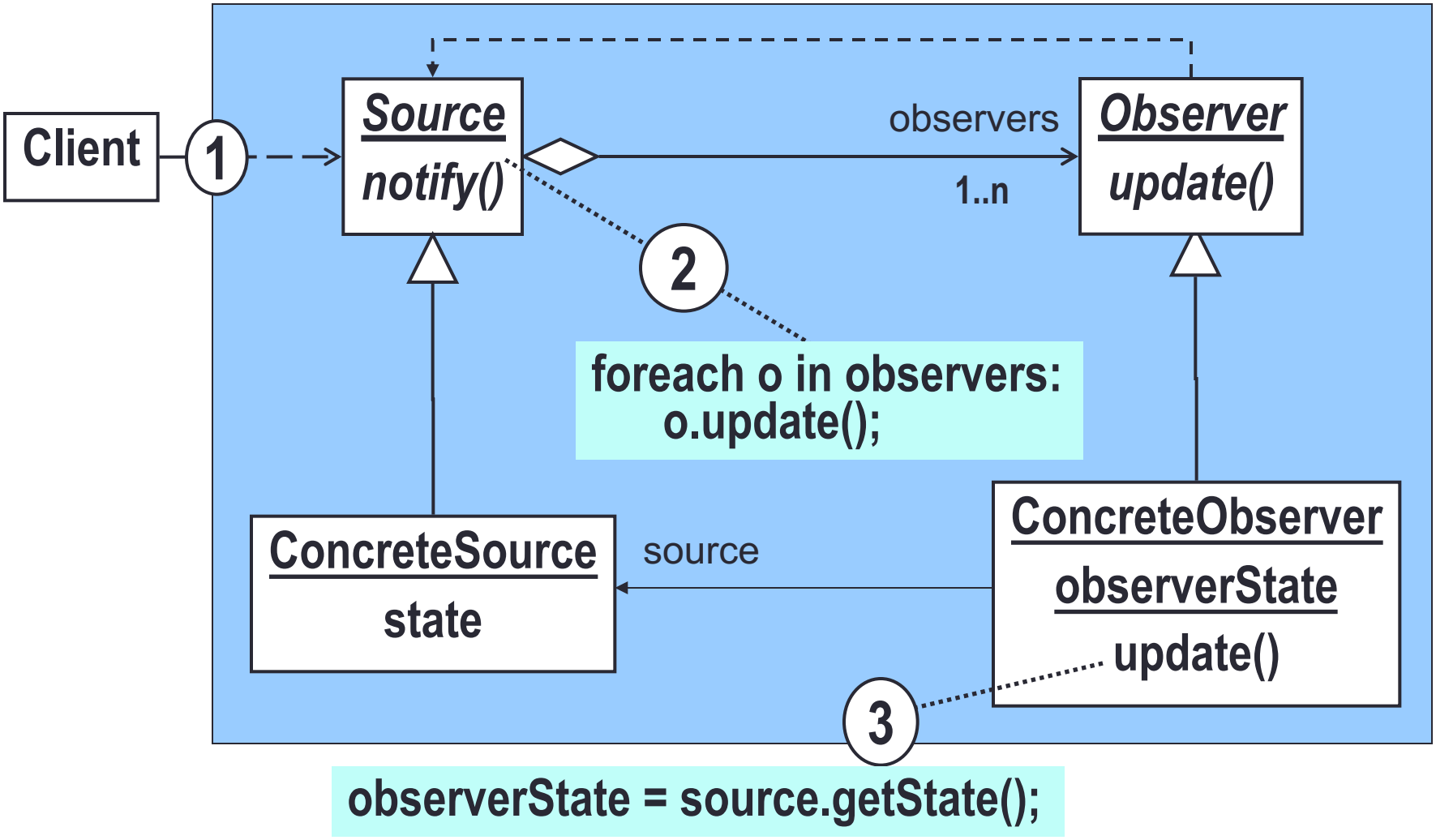
Observer Design Pattern



Server part

Client part

Observer: Structure



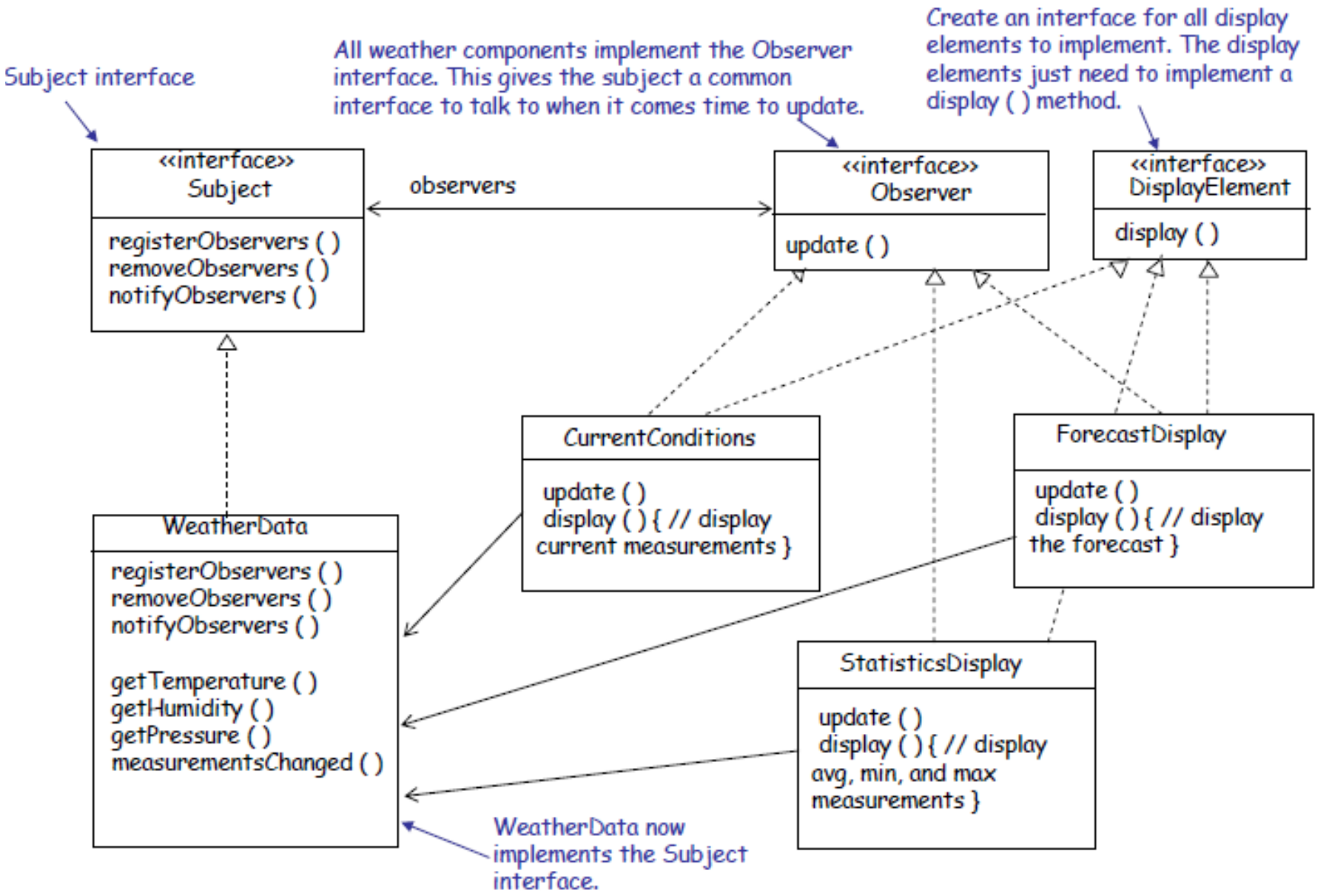
Observer Pattern: Intent

- To keep a set of objects up to date with the state of a designated object
  - one-to-many dependency between objects so that when one object (subjet) changes state, all of its dependents (observers) are notified and updated automatically
- Some observers may observe more than one subject (many-to-many relation)
- The update should specify which subject changed

Design Choices

- When to notify Observers:
  - Automatic on each change
  - Triggered by client
- How to communicate information about the change:
  - Push: subject gives details to observers
  - Pull: subject notifies only that a change has occurred, observer queries subject about details

Designing the Weather Station



Implementing the Weather Station

```
public interface Subject {
    public void registerObserver (Observer o);
    public void removeObserver (Observer o);
    public void notifyObservers ( );
}
```

Both of these methods take an Observer as an argument, that is the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update (float temp, float humidity, float pressure);
}
```

The Observer interface is implemented by all observers, so they all have to implement the update ( ) method.

These are the state values the Observers get from the Subject when a weather measurement changes.

```
public interface DisplayElement {
    public void display ( );
}
```

The DisplayElement interface just includes one method, display ( ), that we will call when the display element needs to be displayed.

WeatherData and Subject Interface

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData ( ){
        observers = new ArrayList ( );
    }
    public void registerObserver (Observer o) {
        observers.add(o);
    }
    public void removeObserver (Observer o) {
        int j = observer.indexOf(o);
        if (j >= 0) {
            observers.remove(j);
        } }
    public void notifyObservers ( ) {
        for (int j = 0; j < observers.size(); j++) {
            Observer observer = (Observer)observers.get(j);
            observer.update(temperature, humidity, pressure);
        }}
    public void measurementsChanged ( ) {
        notifyObservers ( ); }
    // add a set method for testing + other methods.
}
```

Added an ArrayList to hold the Observers, and we create it in the constructor

Here we implement the Subject Interface

Notify the observers when measurements change.

The Display Elements

Implements the Observer and DisplayElement interfaces

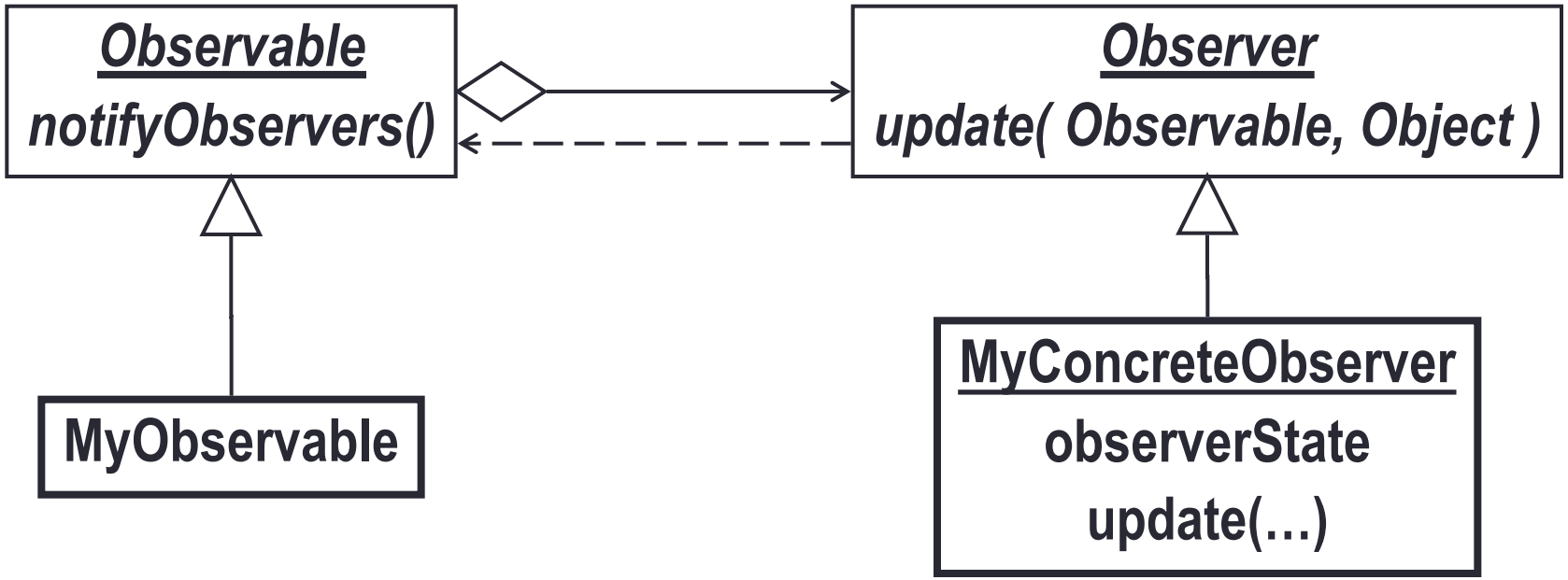
```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperatue;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay (Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver (this);
    }
    public void update (float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display ( );
    }
    public void display ( ){
        System.out.println(" Current conditions : " + temperature + " F degrees  and " + humidity + " % humidity" );
    }
}
```

The constructors passed the weatherData object (the subject) and we use it to register the display as an observer.

When update ( ) is called, we save the temp and humidity and call display ( )

The display ( ) method just prints out the most recent temp and humidity.

Example: Observer in the Java API



Key: *Java API Class* **Developer Class**

Model-View-Controller pattern : (Observable, Observer, (Client & Setup))

data GUIs

Observer: Advantages

- Subject(s) update Observers using a common interface
- Observers are loosely coupled in that Subject(s) knows nothing about them, other than they implement the Observer interface.
- Don’t depend on a specific order of notification for your Observers.

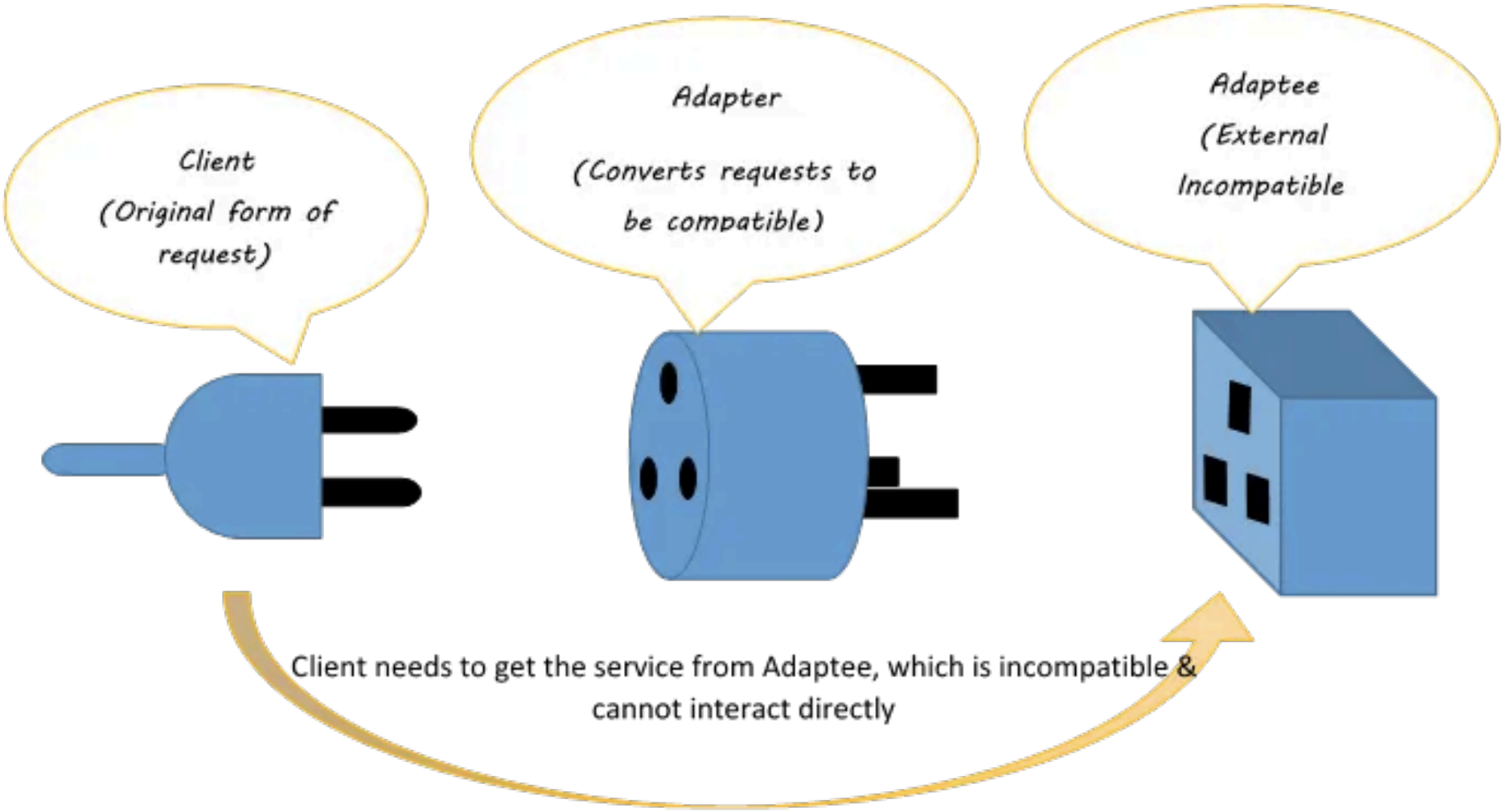
Practice: Applying Observer in Codebase

- Can we apply Observer in any part of the codebase for a specific requirement/a better design?

Content

1. Strategy
2. Observer
3. Adapter
4. State

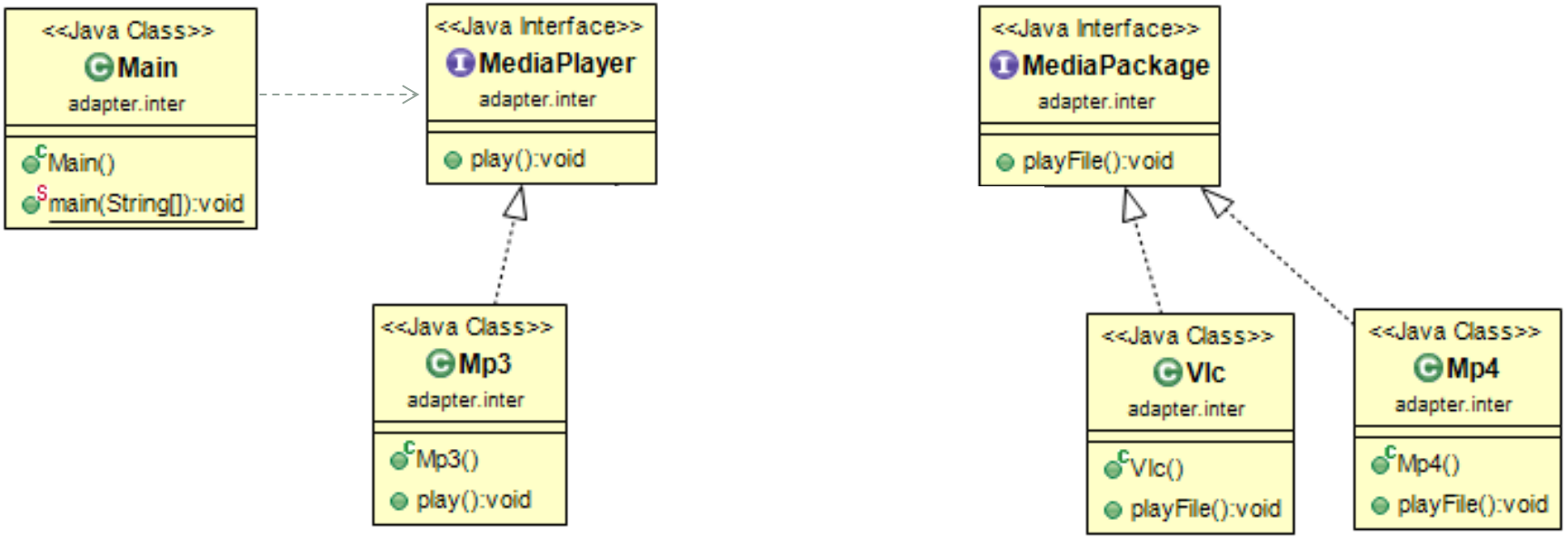
Adapter: Connect Incompatible Interfaces



Exercise: Connect MediaPlayer to MediaPackage

- Existing supported format: mp3
- New formats from MediaPackage library: VLC, mp4

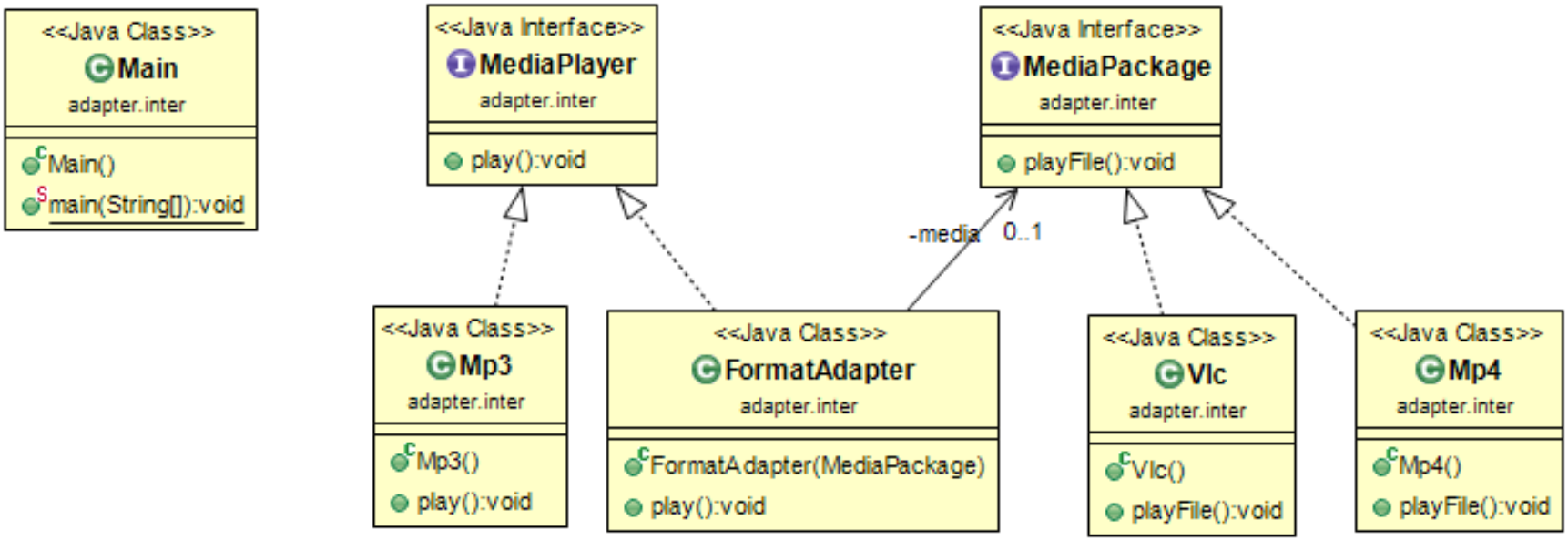
➔ How to support these new formats?



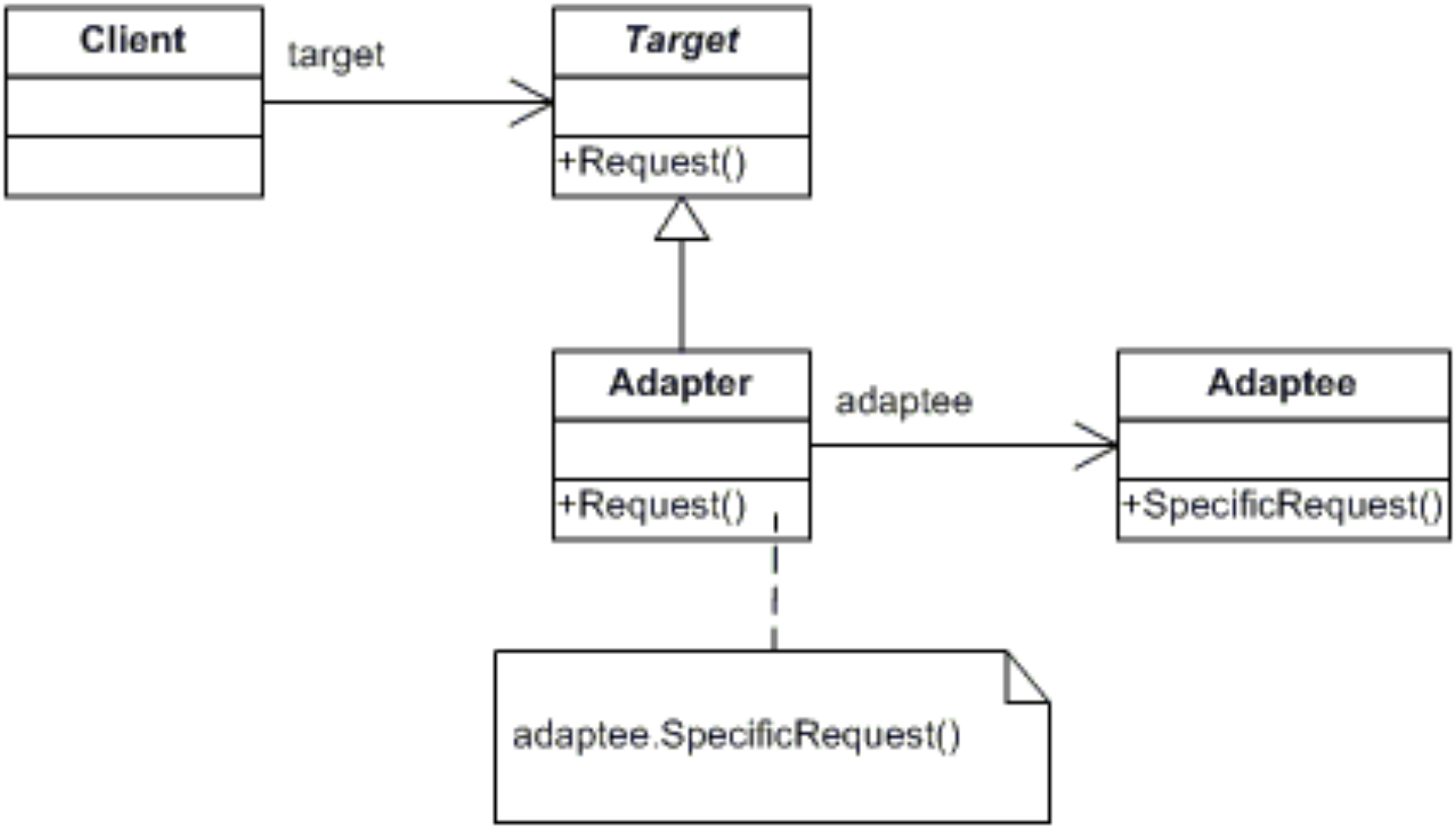
Adapter: Intent

- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface
  - Also know as Wrapper

Adapter: Playing New Format

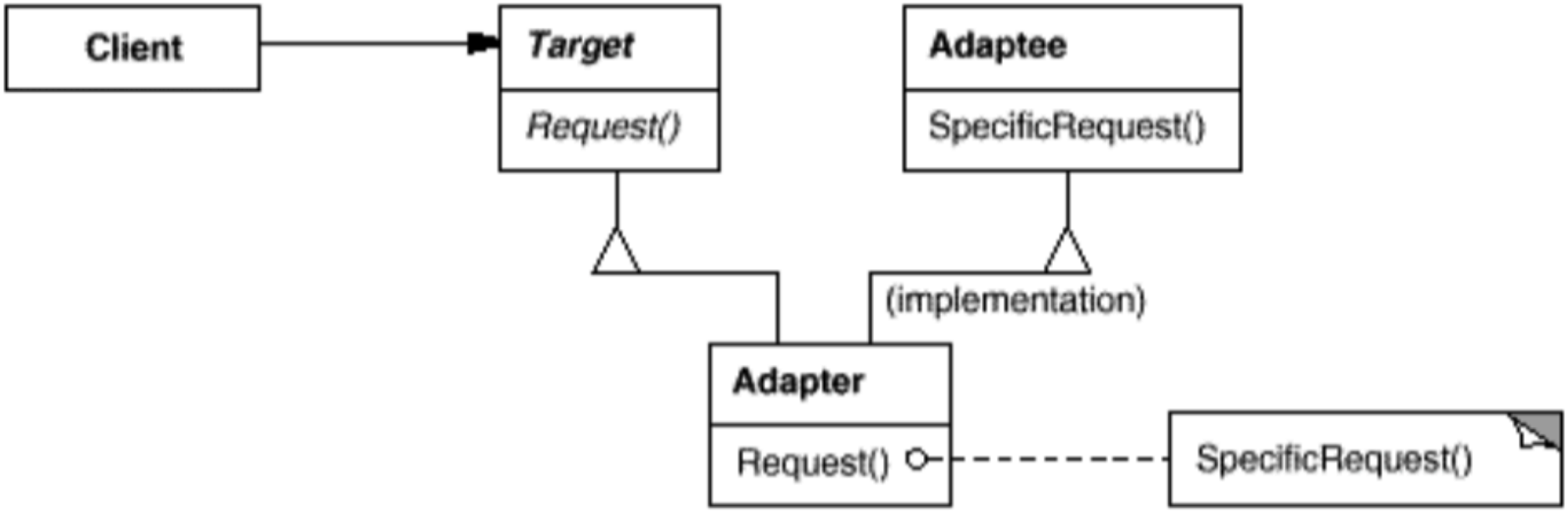


Adapter (object) Pattern: Structure

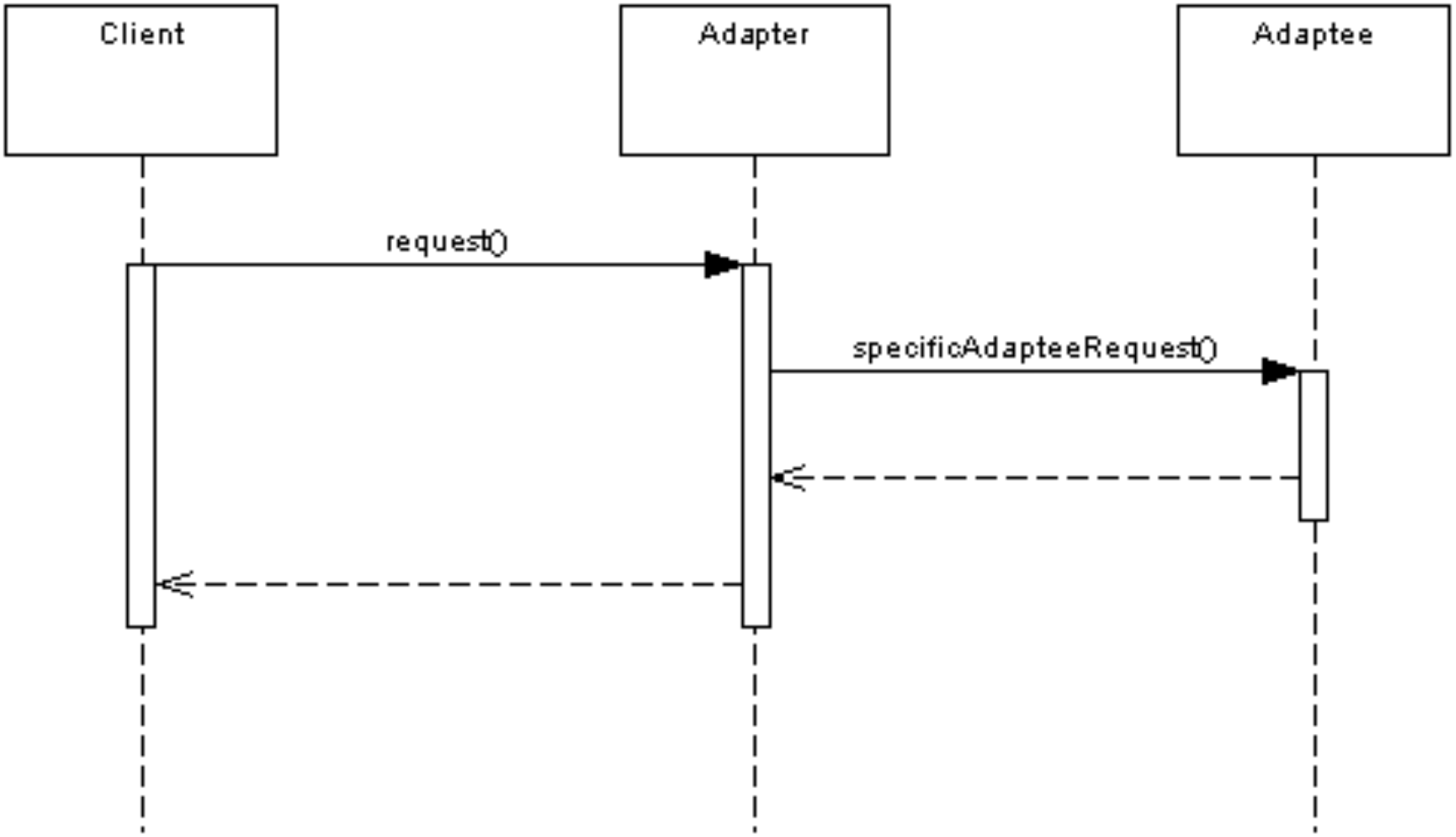


Adapter (class) Pattern: Structure

- Multi-inheritance?



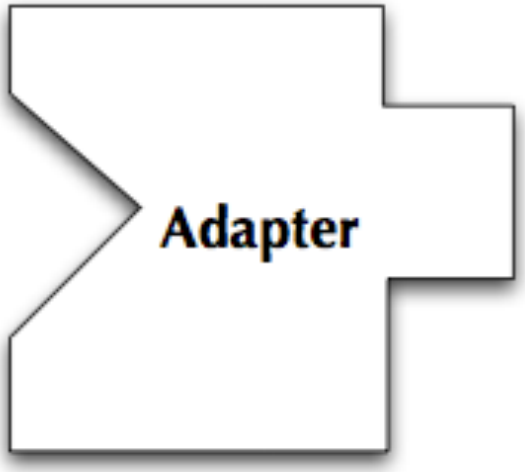
Adapter: Collaboration



E.g. Software Adapters

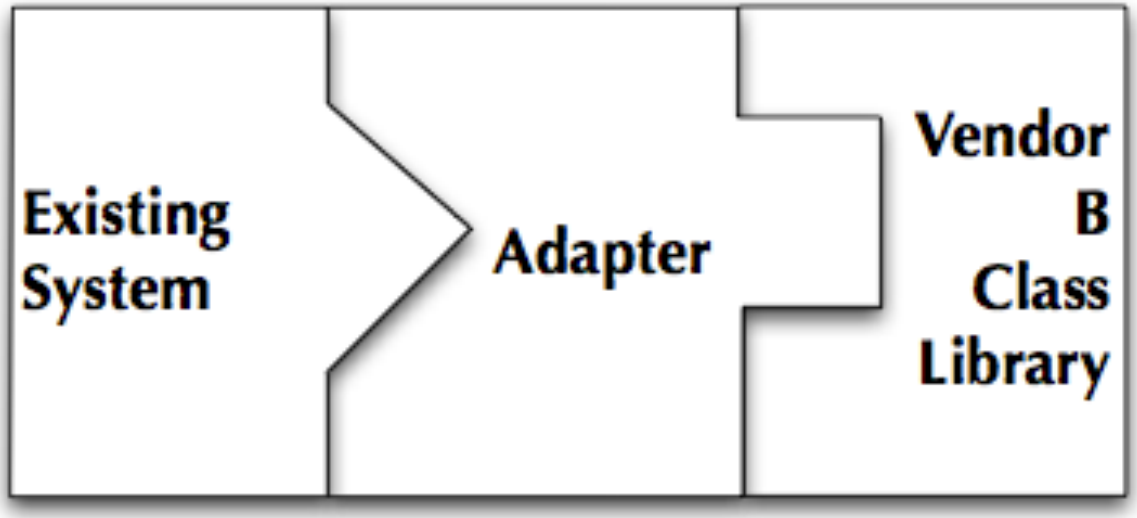
Interface Mismatch
Need Adapter

Create Adapter



And then...

E.g. Software Adapters



…plug it in

Benefit: Existing system and new vendor library do not change — new code is isolated within the adapter

Motivation

- Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
- We can not change the library interface, since we may not have its source code
- Even if we did have the source code, we probably should not change the library for each domain-specific application

Adapter: Applicability

- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces

Practice: Applying Adapter in Codebase

- Can we apply Adapter in any part of the codebase for a specific requirement/a better design?

Content

1. Strategy
2. Observer
3. Adapter
4. State

Example: TV Remote

TV Remote object with a simple button to perform action

- if the State is ON, it will turn on the TV
- if state is OFF, it will turn off the TV

We can implement it using **if-else condition**

```
public class TVRemoteBasic {

    private String state="";

    public void setState(String state){
        this.state=state;
    }

    public void doAction(){
        if(state.equalsIgnoreCase("ON")){
            System.out.println("TV is turned ON");
        }else if(state.equalsIgnoreCase("OFF")){
            System.out.println("TV is turned OFF");
        }
    }

    public static void main(String args[]){
        TVRemoteBasic remote = new TVRemoteBasic();

        remote.setState("ON");
        remote.doAction();

        remote.setState("OFF");
        remote.doAction();
    }

}
```

WHAT IS PROBLEM & SOLUTION?

TV Remote

Problem in TV Remote

If else condition for different states

- Client code should know the specific values to use for setting the state of remote,
- If number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend
- Design will become more cleaner by using enums and switches or multiple if then else
  - But still the above problems

Problem in Software Design

Tired of conditionals?

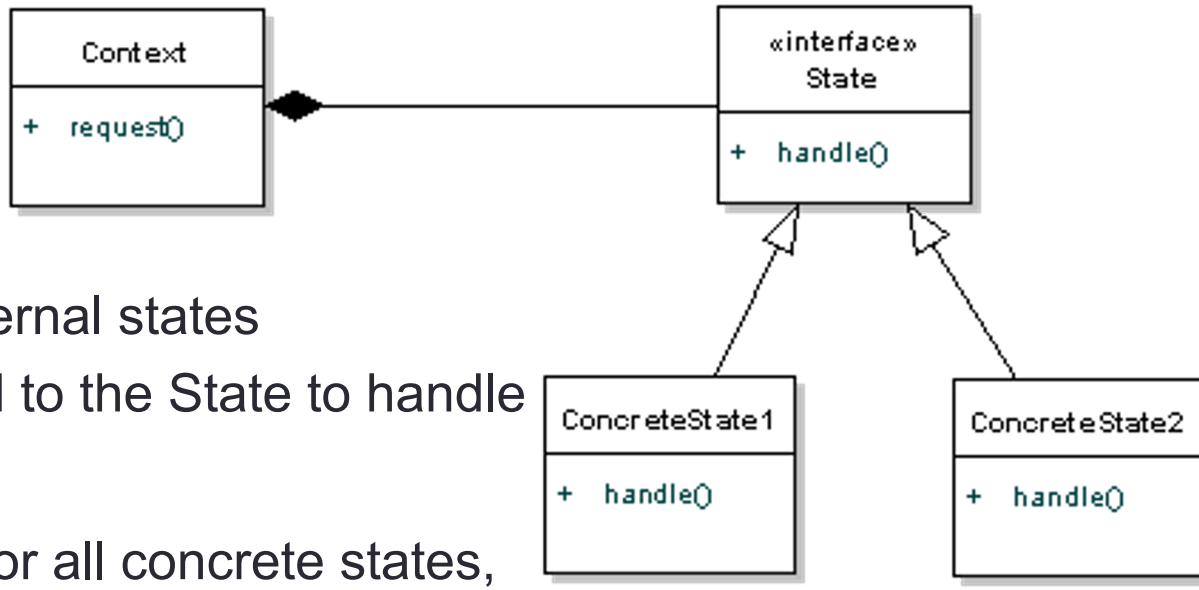
- When writing code, our classes often go through a series of transformations
- What starts out as a simple class will grow as behavior is added
- if you didn't take the necessary precautions, your code will become difficult to understand and maintain.
- Too often, the state of an object is kept by creating multiple Boolean attributes and deciding how to behave based on the values
- This can become cumbersome and difficult to maintain when the complexity of your class starts to increase
- This is a common problem on most projects

Solution for TV Remote

- State Design Pattern (Behavioral Pattern)
  - Creating objects which represent various states and a context object whose behavior varies as its state object changes

“Allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.”

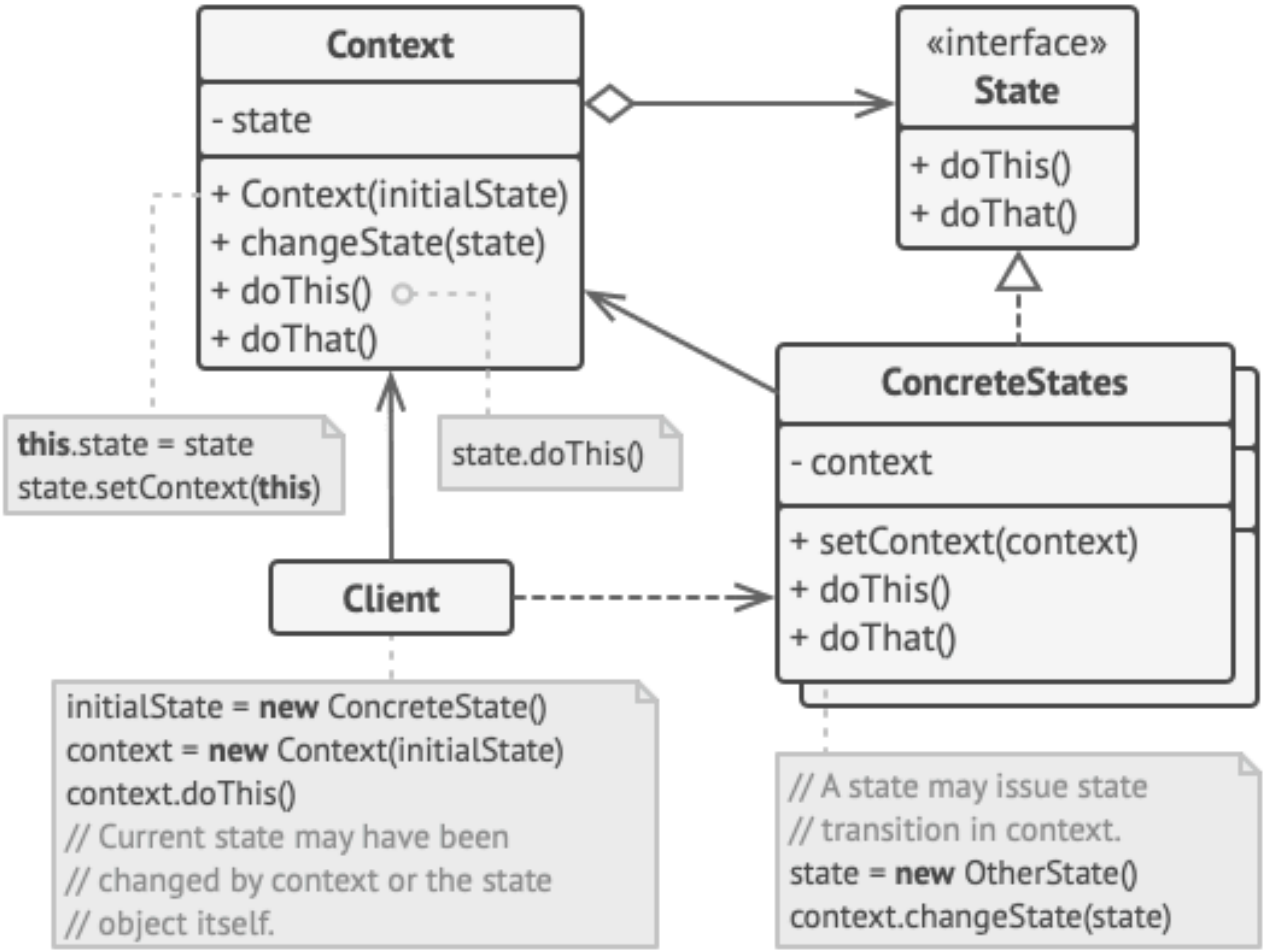
State Design Pattern: Structure



- Context
  - Have a number of internal states
  - request() is delegated to the State to handle
- State interface
  - A common interface for all concrete states, encapsulating all behaviour associated with a particular state.
- ConcreteState
  - implements it's own implementation for the request

When a Context changes state, what really happens is that we have a different ConcreteState associated with it.

State Design Pattern: Structure



Quiz: Design State DP for TV Remote?

Advantages of State DP

- Implement polymorphic behavior is clearly visible
- The chances of error are less and it's very easy to add more states for additional behavior making it more robust,
- Easily maintainable and flexible.
- Avoiding if-else or switch-case conditional logic in this scenario
- Avoiding inconsistent states
- Putting all associated behavior together in one state object

Practice: Applying State in Codebase

- Can we apply State in any part of the codebase for a specific requirement/a better design?