COMP2511

WEEK 5

Are original chips lacking in flavour?

IMPORTANT NOTICE

- Assignment-i due yesterday, Well Done!!
- Assignment-ii to be released soonTM
- Check your lab marks are correct.

A G E N D A

- Streams
- Strategy Pattern
- Observer Pattern

Streams

via Code Example

STREAMS

Common uses of streams are:

- forEach
- filter
- map
- reduce

Sort of similar to the Array prototypes/methods in JavaScript

What is a pattern???

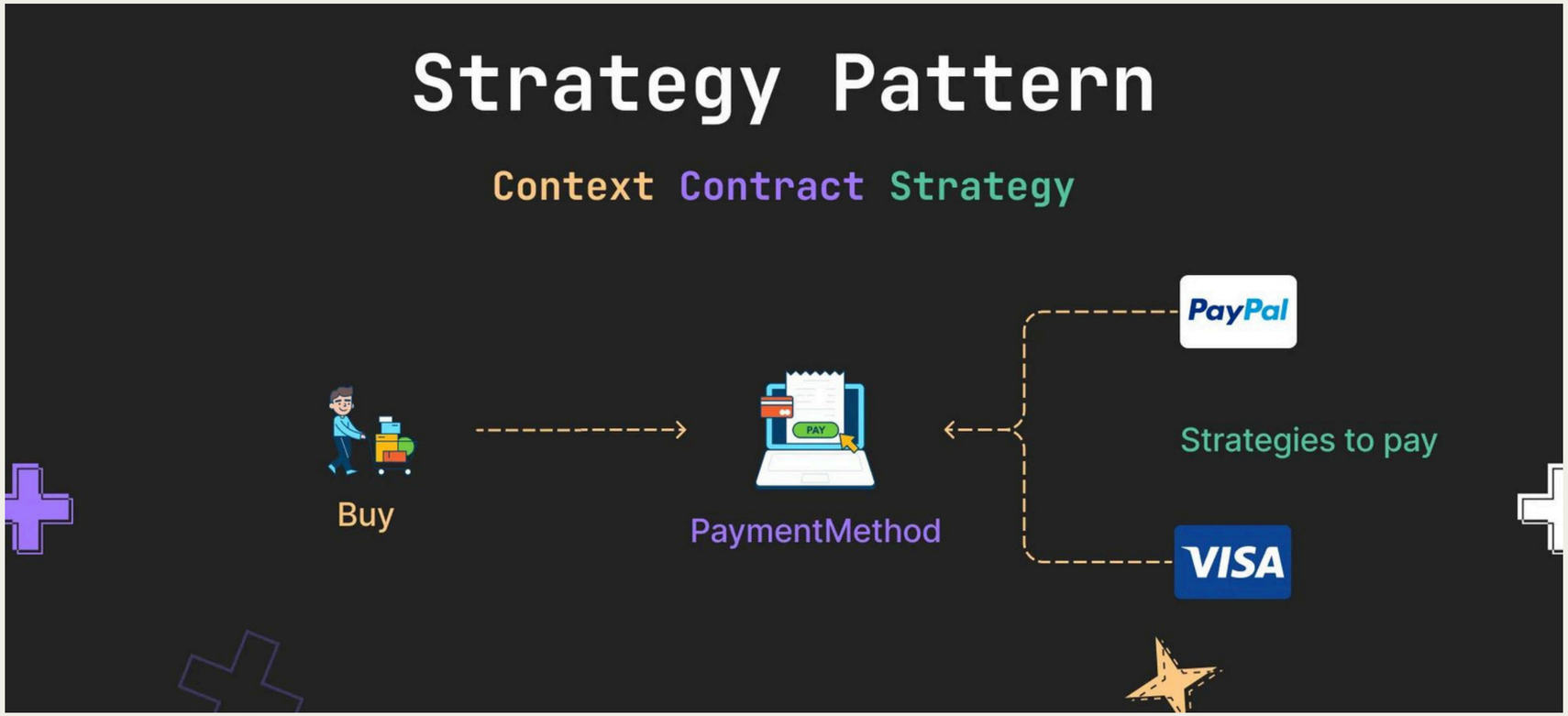
- Refers to design pattern
- Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customise to solve a particular design problem in your code.
  - Somewhat like a “template” for you to use, to help better design your system
    - See that’s funny because the template pattern exists :joy:
  - NOTE: If you use the wrong pattern, this could be considered bad design

Really helpful website: https://refactoring.guru/

Strategy Pattern

STRATEGY PATTERN

Strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime.



STRATEGY PATTERN

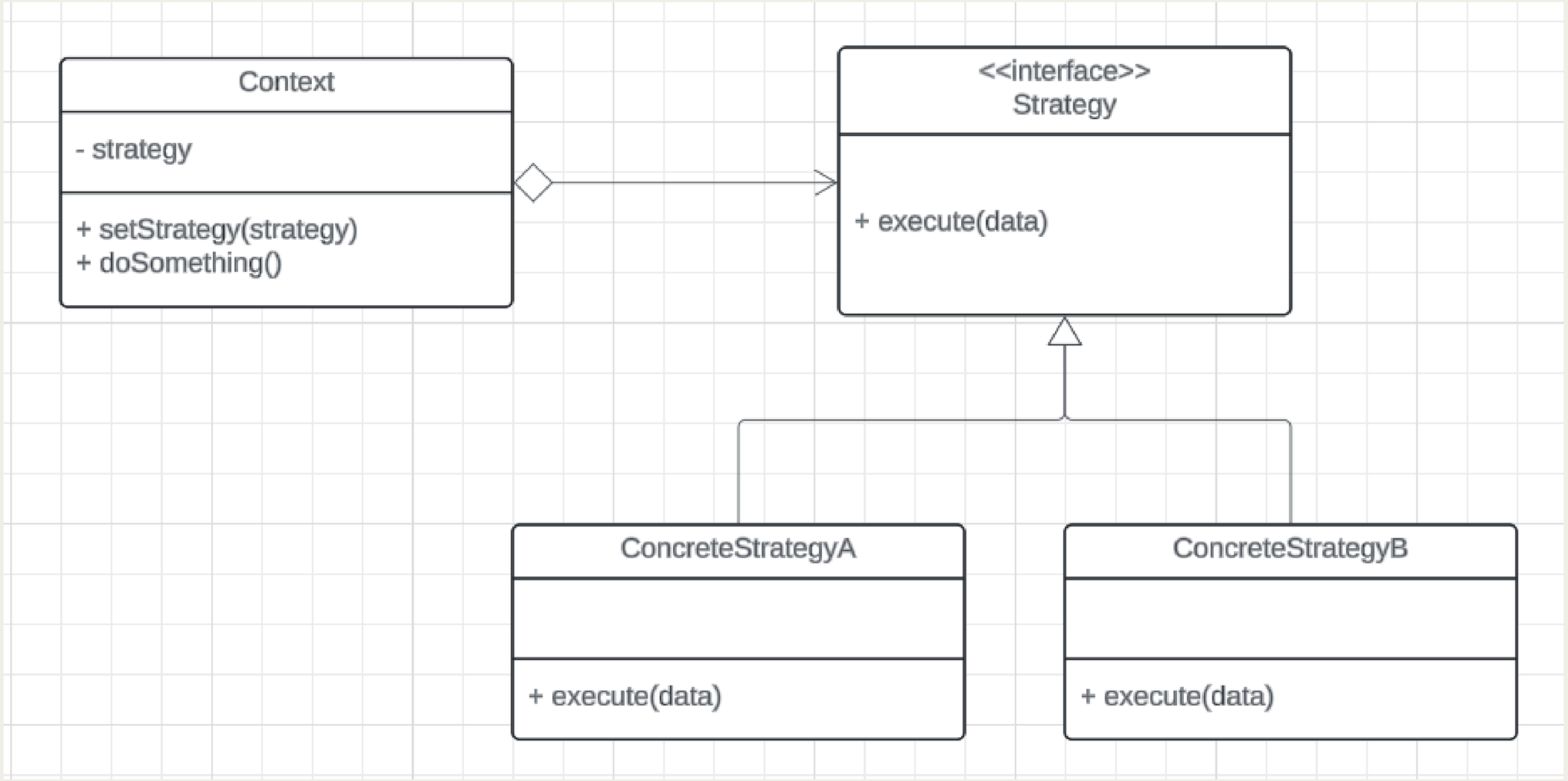
What type of design pattern is strategy?

Behavioral Pattern

Behavioral patterns are patterns concerned with algorithms and the assignment of responsibility between object

- Uses composition instead of inheritance
- Allows for dependency injection (Selects and adapts an algorithm at run time). Change the behaviour at runtime.
- Encapsulates interchangeable behaviours and uses delegation to decide which one to use
- Useful when you want to share behaviour across an inheritance tree

STRATEGY PATTERN



STRATEGY PATTERN

Inside src/restaurant is a solution for a restaurant payment system with the following requirements:

- The restaurant has a menu, stored in a JSON file. Each meal on the menu has a name and price
- The system displays all of the standard meal names and their prices to the user so they can make their order
- The user can enter their order as a series of meals, and the system returns their cost
- The prices on meals often vary in different circumstances. The restaurant has three different price settings:
  - Standard - normal rates
  - Holiday - 15% surcharge on all items for all customers
  - Happy Hour - where registered members get a 40% discount, while standard customers get 30%
- The prices displayed on the menu are the ones for standard customers in all settings

STRATEGY PATTERN

```java
public class Resturant {
  // ...
      public double cost(List<Meal> order, String payee) {
        return switch (chargingStrategy) {
            case "standard" -> order.stream().mapToDouble(meal -> meal.getCost()).sum();
            case "holiday" -> order.stream().mapToDouble(meal -> meal.getCost() * 1.15).sum();
            case "happyHour" -> {
                if (members.contains(payee)) {
                    yield order.stream().mapToDouble(meal -> meal.getCost() * 0.6).sum();
                } else {
                    yield order.stream().mapToDouble(meal -> meal.getCost() * 0.7).sum();
                }
            }
            default -> 0;
        };
    }
}
```

- How does the code violate the open/closed principle?
- How does this make the code brittle?

STRATEGY PATTERN

```java
public class Resturant {
  // ...
      public double cost(List<Meal> order, String payee) {
        return switch (chargingStrategy) {
            case "standard" -> order.stream().mapToDouble(meal -> meal.getCost()).sum();
            case "holiday" -> order.stream().mapToDouble(meal -> meal.getCost() * 1.15).sum();
            case "happyHour" -> {
                if (members.contains(payee)) {
                    yield order.stream().mapToDouble(meal -> meal.getCost() * 0.6).sum();
                } else {
                    yield order.stream().mapToDouble(meal -> meal.getCost() * 0.7).sum();
                }
            }
            default -> 0;
        };
    }
}
```

- How does the code violate the open/closed principle?
- How does this make the code brittle?

Not closed for modification / open for extension - we need to continually add to the switch statement.

Makes code more brittle as new requirements cause things to break/more difficult to extend functionality.

STRATEGY PATTERN

Same thing here! if new cases need to be added, the class's method itself needs to be changed. Cannot be extended!

```
public class Resturant {
  // ...
    public void displayMenu() {
        double modifier = switch (chargingStrategy) {
            case "standard" -> 1;
            case "holiday" -> 1.15;
            case "happyHour" -> 0.7;
            default -> 0;
        };

        for (Meal meal : menu) {
            System.out.println(meal.getName() + " - " + meal.getCost() * modifier);
        }
    }

}
```

STRATEGY PATTERN

To fix these issues, we can introduce a strategy pattern and move all the individual case logic into their own classes

```java
public interface ChargingStrategy {
  /**
   * The cost of a meal.
   */
  public double cost(List<Meal> order, boolean payeeIsMember);

  /**
   * Modifying factor of charges for standard customers.
   */
  public double standardChargeModifier();
}
```

The prices on meals often vary in different circumstances. The restaurant has three different price settings:

- Standard - normal rates
- Holiday - 15% surcharge on all items for all customers
- Happy Hour - where registered members get a 40% discount, while standard customers get 30%

STRATEGY PATTERN

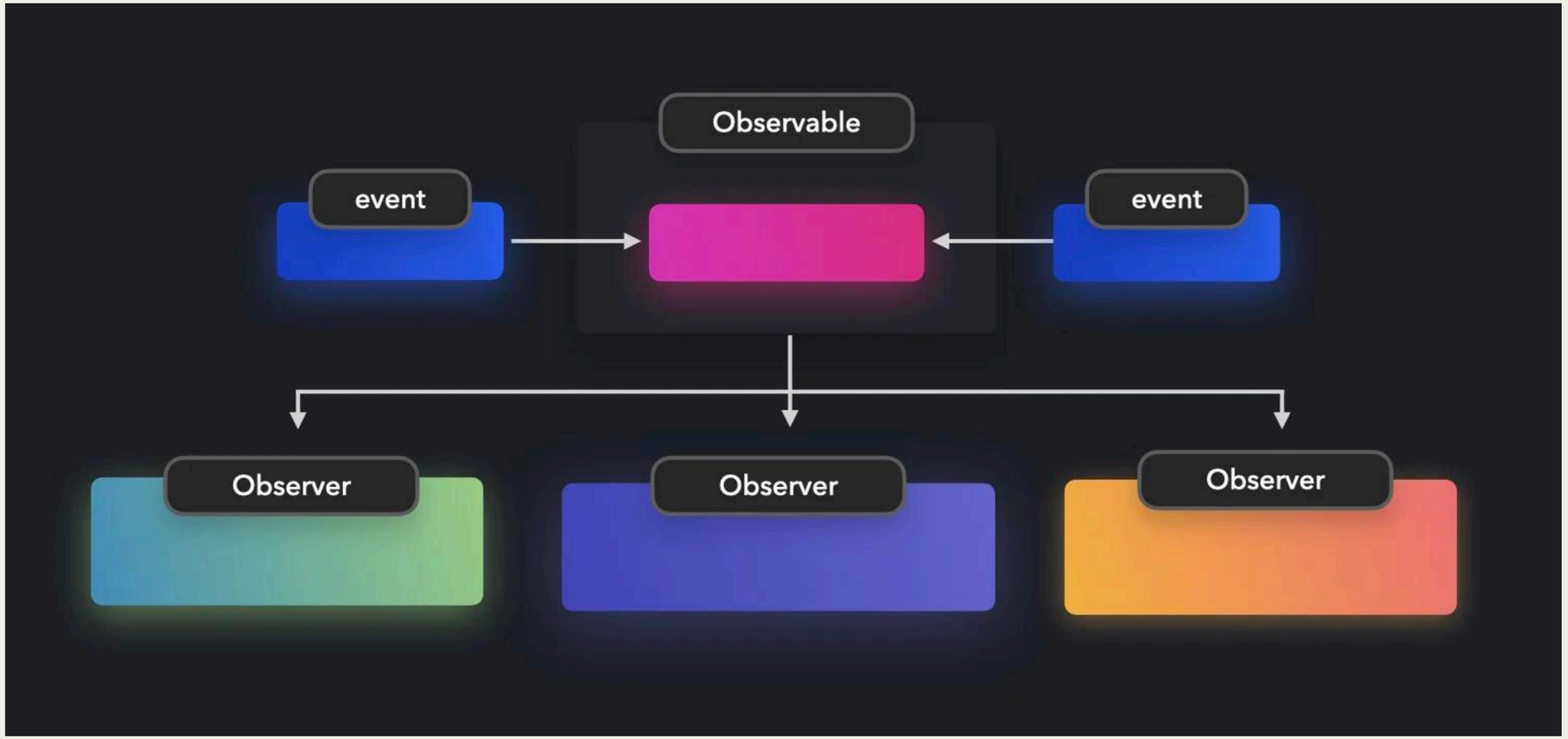
Extend the system to add the following pricing strategy:

- Prize Draw: A special promotion where every 100th customer (since the start of the promotion) gets their meal for free!

Observer Pattern

OBSERVER PATTERN

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

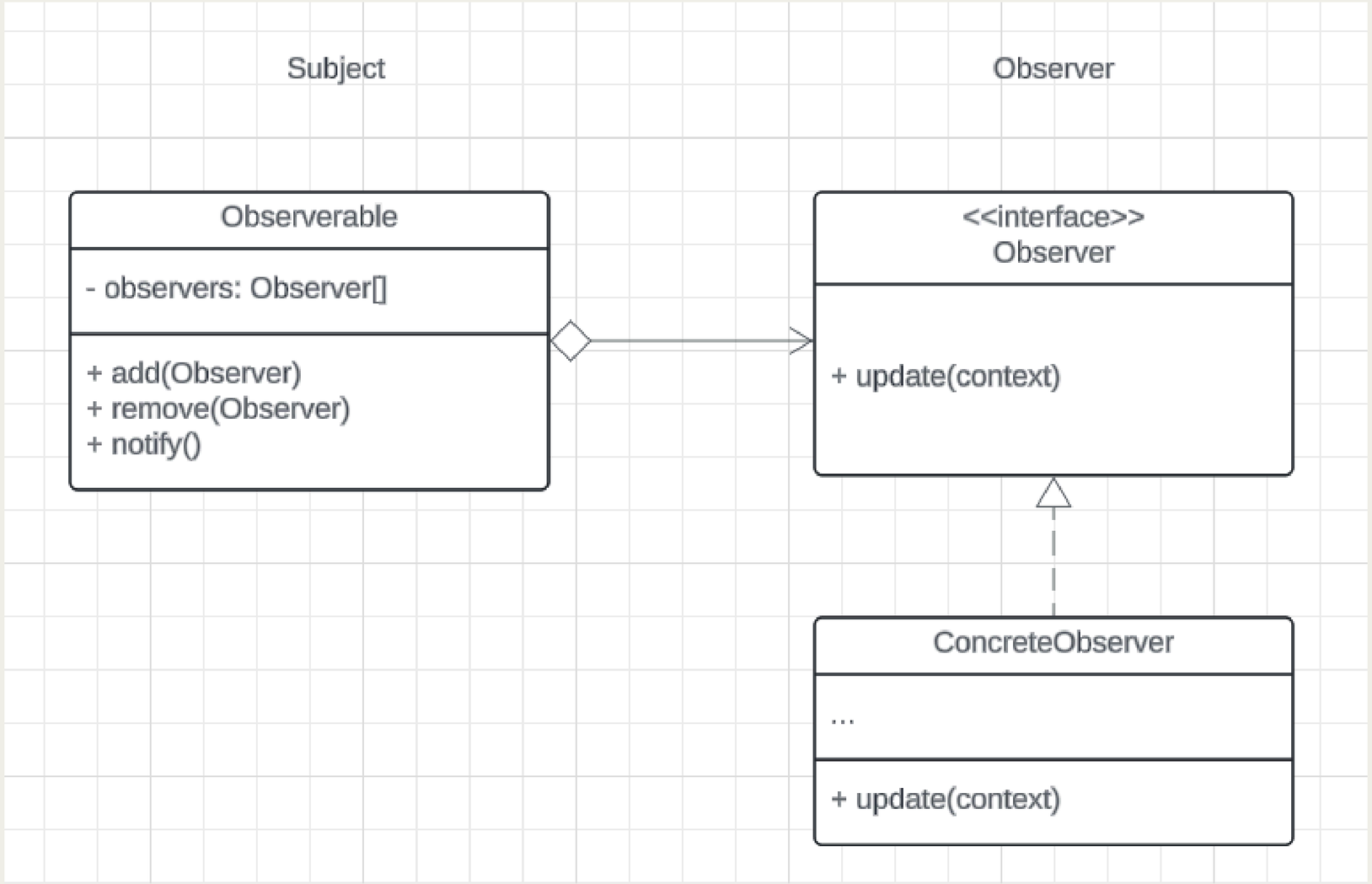


OBSERVER PATTERN

An object (subject) maintains a list of dependents called observers. The subject notifies the observers automatically of any state changes.

- Used to implement event handling systems ("event driven" programming).
- Able to dynamically add and remove observers
- One-to-many dependency such that when the subject changes state, all of its dependents (observers) are notified and updated automatically
- Loosing coupling of objects that interact with each other.

OBSERVER PATTERN



OBSERVER PATTERN

In src/youtube, create a model for the following requirements of a Youtube-like video creating and watching service using the Observer Pattern:

- A Producer has a name, a series of subscribers and videos
- When a producer posts a new video, all of the subscribers are notified that a new video was posted
- A User has a name, and can subscribe to any Producer
- A video has a name, length and producer

LAB LAB

REEEEEEE