COMP2511

WEEK 5

Are original chips lacking in flavour?

IMPORTANT NOTICE

- Assignment-i is due IN UNDER TWO HOURS!
- Assignment-ii pairs are available on TEAMS.
- Make sure your lab marks are correct

A G E N D A

- Strategy Pattern
- Observer Pattern
- State Pattern

What is a pattern???

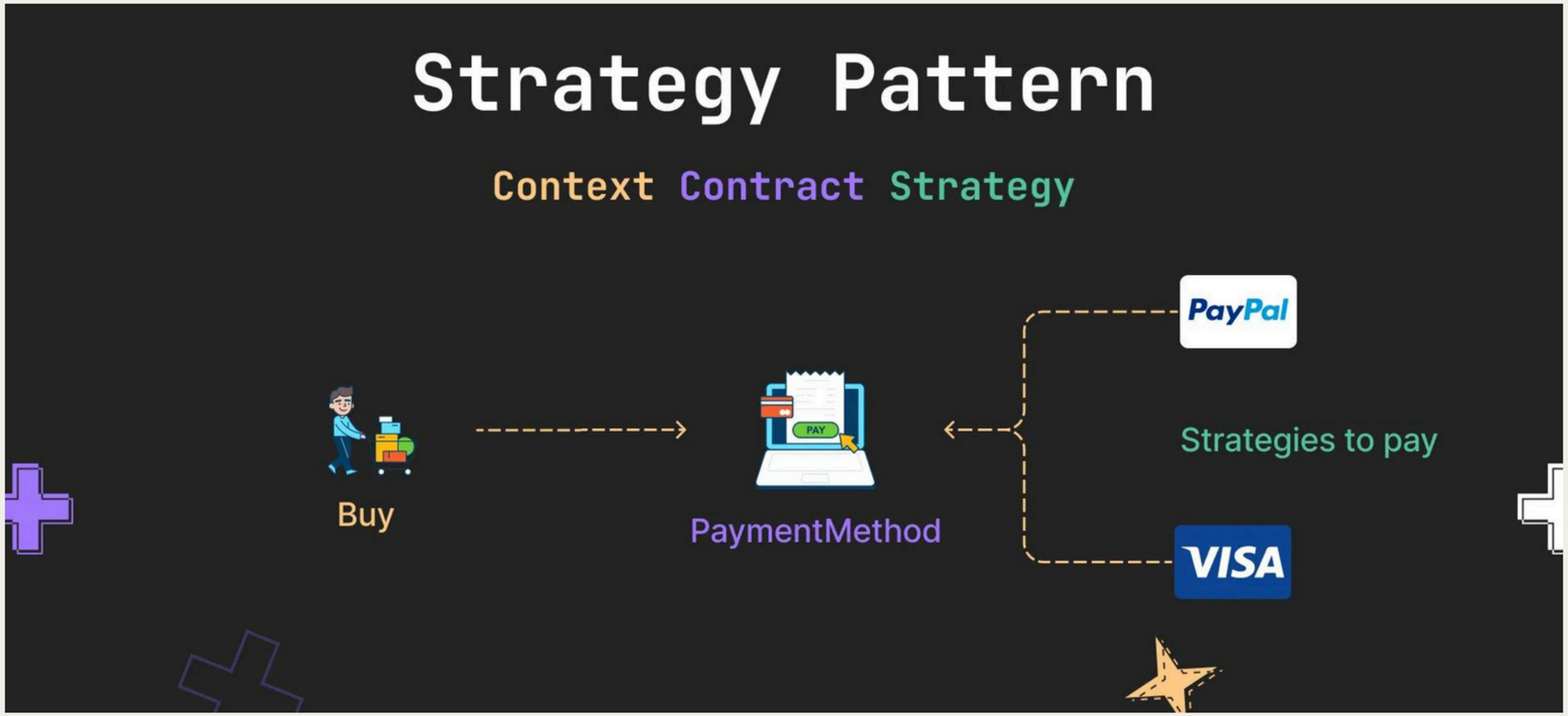
- Refers to design pattern
- Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customise to solve a particular design problem in your code.
  - Somewhat like a “template” for you to use, to help better design your system
    - See thats 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

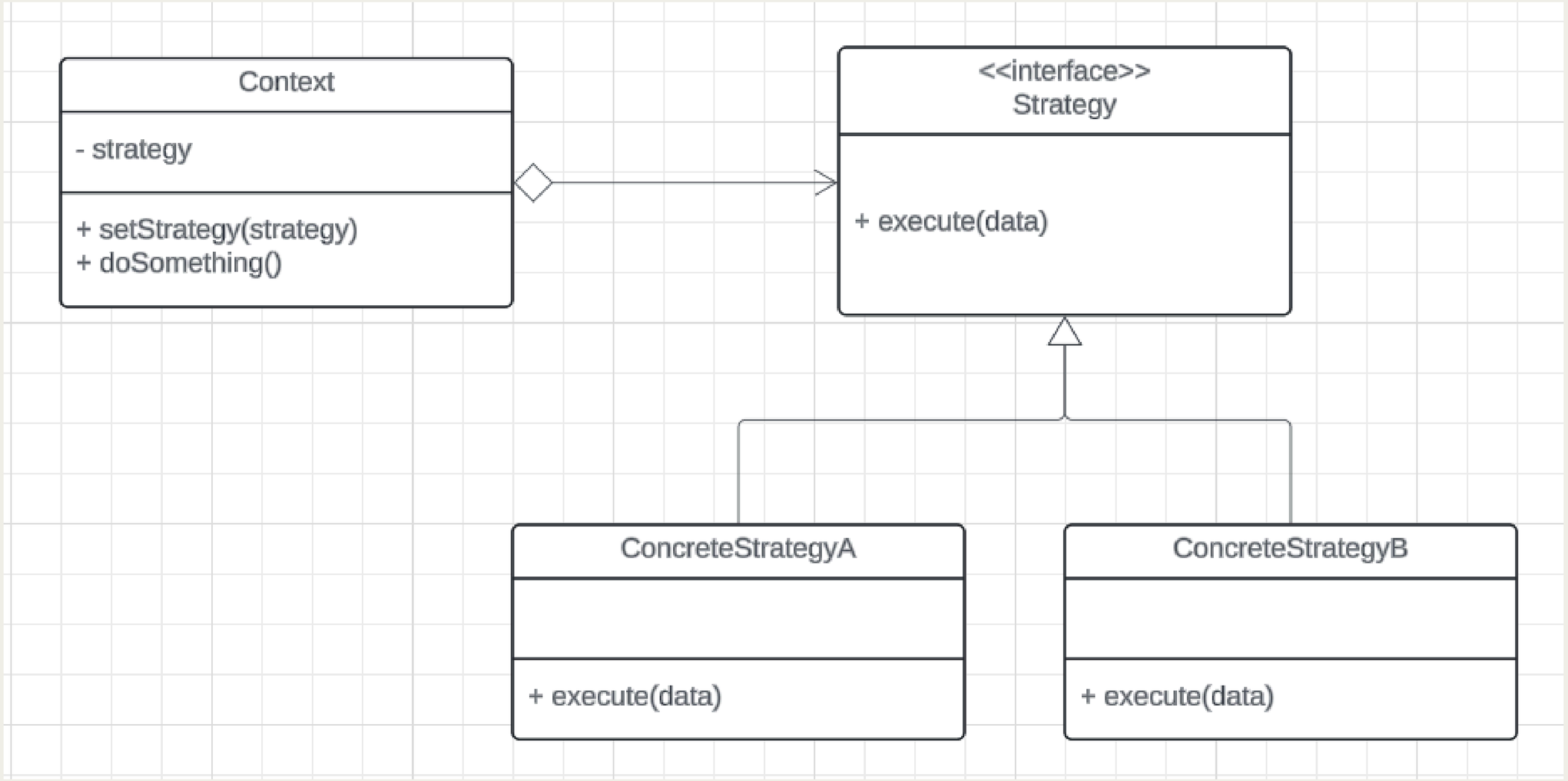
What type of design pattern is strategy?

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 four 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%
  - Discount - where registered members get a 15% discount and standard customers pay normal prices
- The prices displayed on the menu are the ones for standard customers in all settings

STRATEGY PATTERN

```java
public class Restaurant {
    ...
    public double cost(List<Meal> order, String payee) {
        switch (chargingStrategy) {
            case "standard":
                return order.stream().mapToDouble(meal -> meal.getCost()).sum();
            case "holiday":
                return order.stream().mapToDouble(meal -> meal.getCost() * 1.15).sum();
            case "happyHour":
                if (members.contains(payee)) {
                    return order.stream().mapToDouble(meal -> meal.getCost() * 0.6).sum();
                } else {
                    return order.stream().mapToDouble(meal -> meal.getCost() * 0.7).sum();
                }
            case "discount":
                if (members.contains(payee)) {
                    return order.stream().mapToDouble(meal -> meal.getCost() * 0.85).sum();
                } else {
                    return order.stream().mapToDouble(meal -> meal.getCost()).sum();
                }
            default: return 0;
        }
    }
}
```

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

STRATEGY PATTERN

```
public class Restaurant {
    ...
    public double cost(List<Meal> order, String payee) {
        switch (chargingStrategy) {
            case "standard":
                return order.stream().mapToDouble(meal -> meal.getCost()).sum();
            case "holiday":
                return order.stream().mapToDouble(meal -> meal.getCost() * 1.15).sum();
            case "happyHour":
                if (members.contains(payee)) {
                    return order.stream().mapToDouble(meal -> meal.getCost() * 0.6).sum();
                } else {
                    return order.stream().mapToDouble(meal -> meal.getCost() * 0.7).sum();
                }
            case "discount":
                if (members.contains(payee)) {
                    return order.stream().mapToDouble(meal -> meal.getCost() * 0.85).sum();
                } else {
                    return order.stream().mapToDouble(meal -> meal.getCost()).sum();
                }
            default: return 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 Restaurant {
    ...
    public void displayMenu() {
        double modifier = 0;
        switch (chargingStrategy) {
            case "standard": modifier = 1; break;
            case "holiday": modifier = 1.15; break;
            case "happyHour": modifier = 0.7; break;
            case "discount": modifier = 1; break;
        }

        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

```
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();
}
```

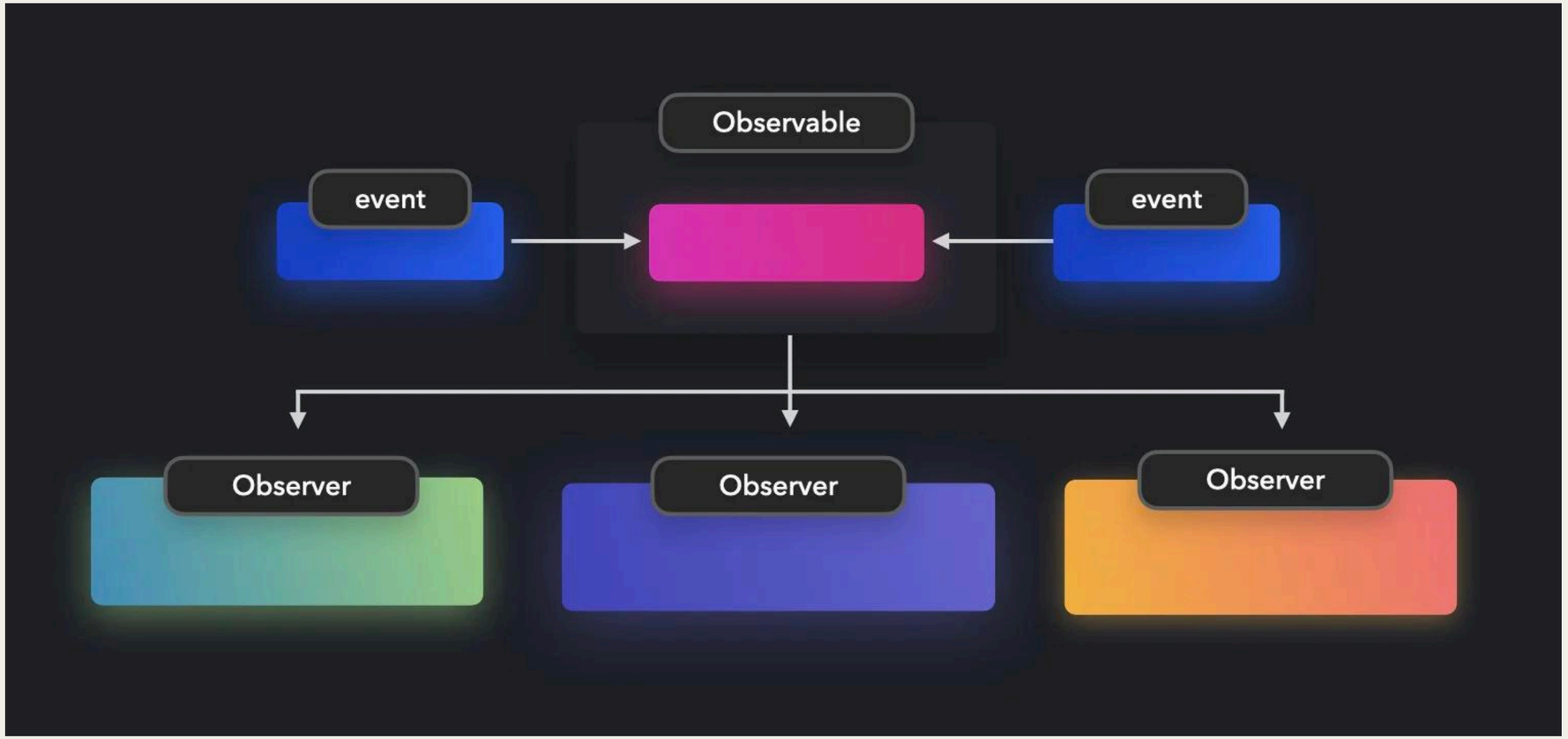
The prices on meals often vary in different circumstances. The restaurant has four different price settings:

- 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%
- Discount - where registered members get a 15% discount and standard customers pay normal prices

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.

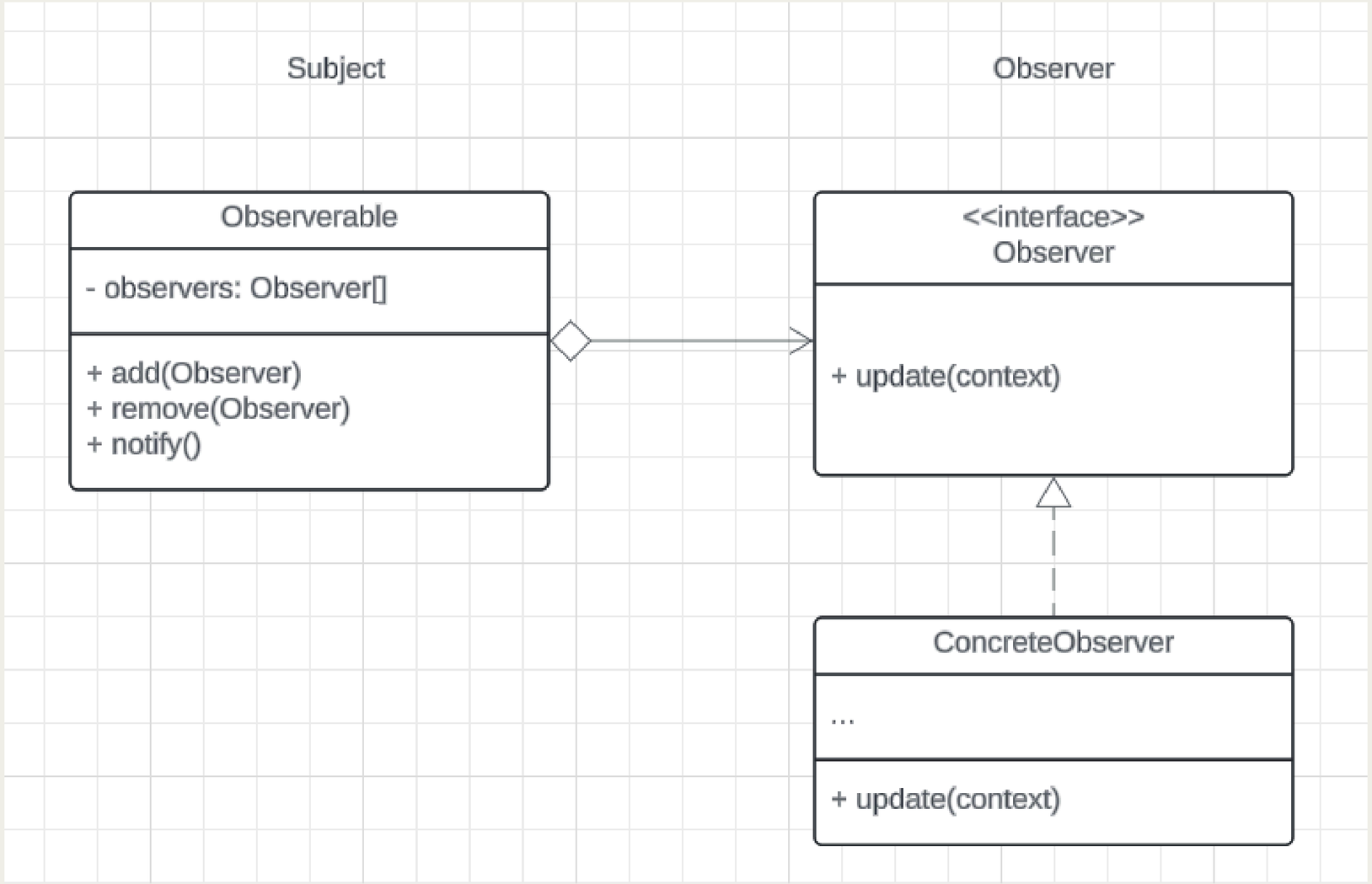


OBSERVER PATTERN

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

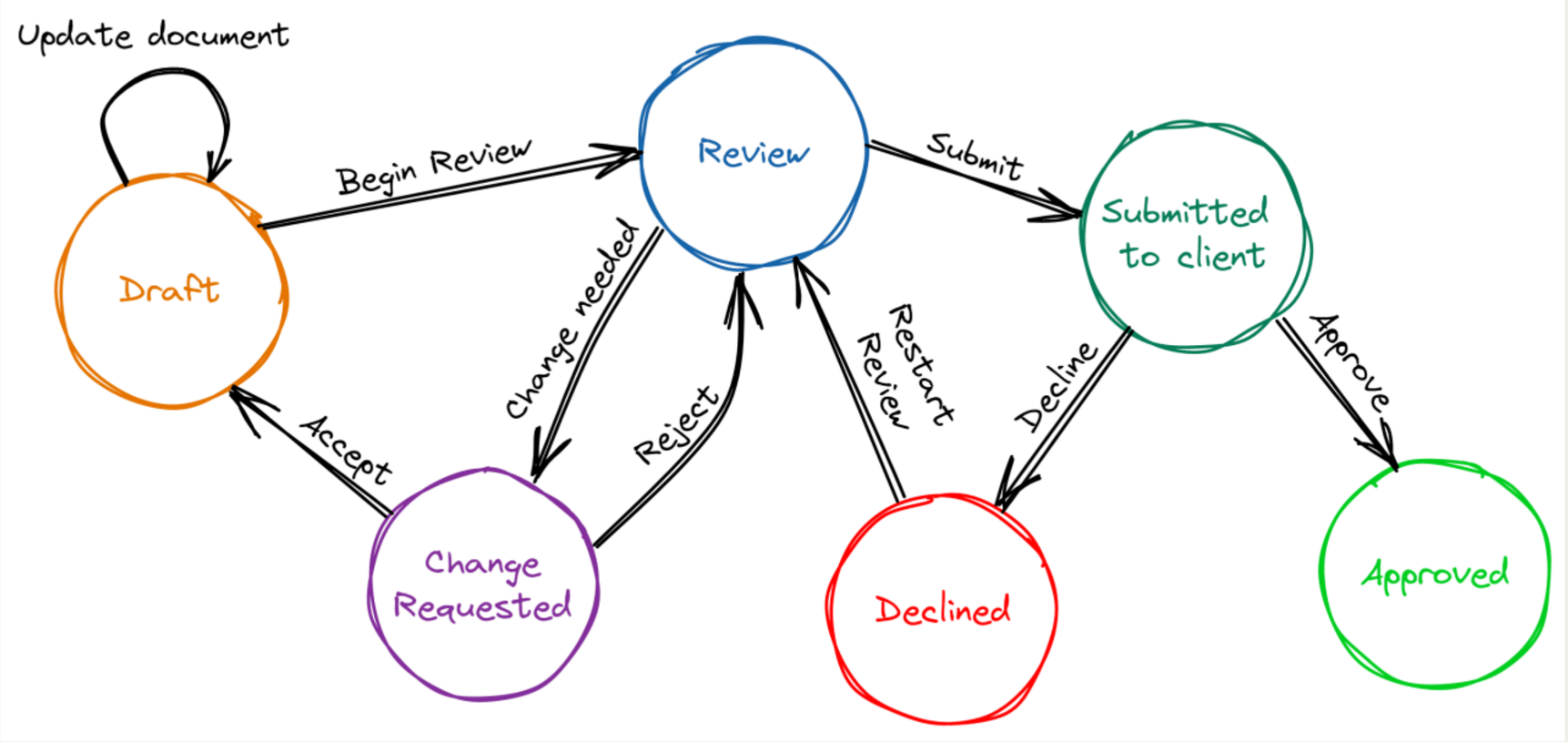
In src/youtube, create a model for the following requirements of a Youtube-like video creating and watching service using the Observer Pattern:

- 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

State Pattern

STATE PATTERN

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

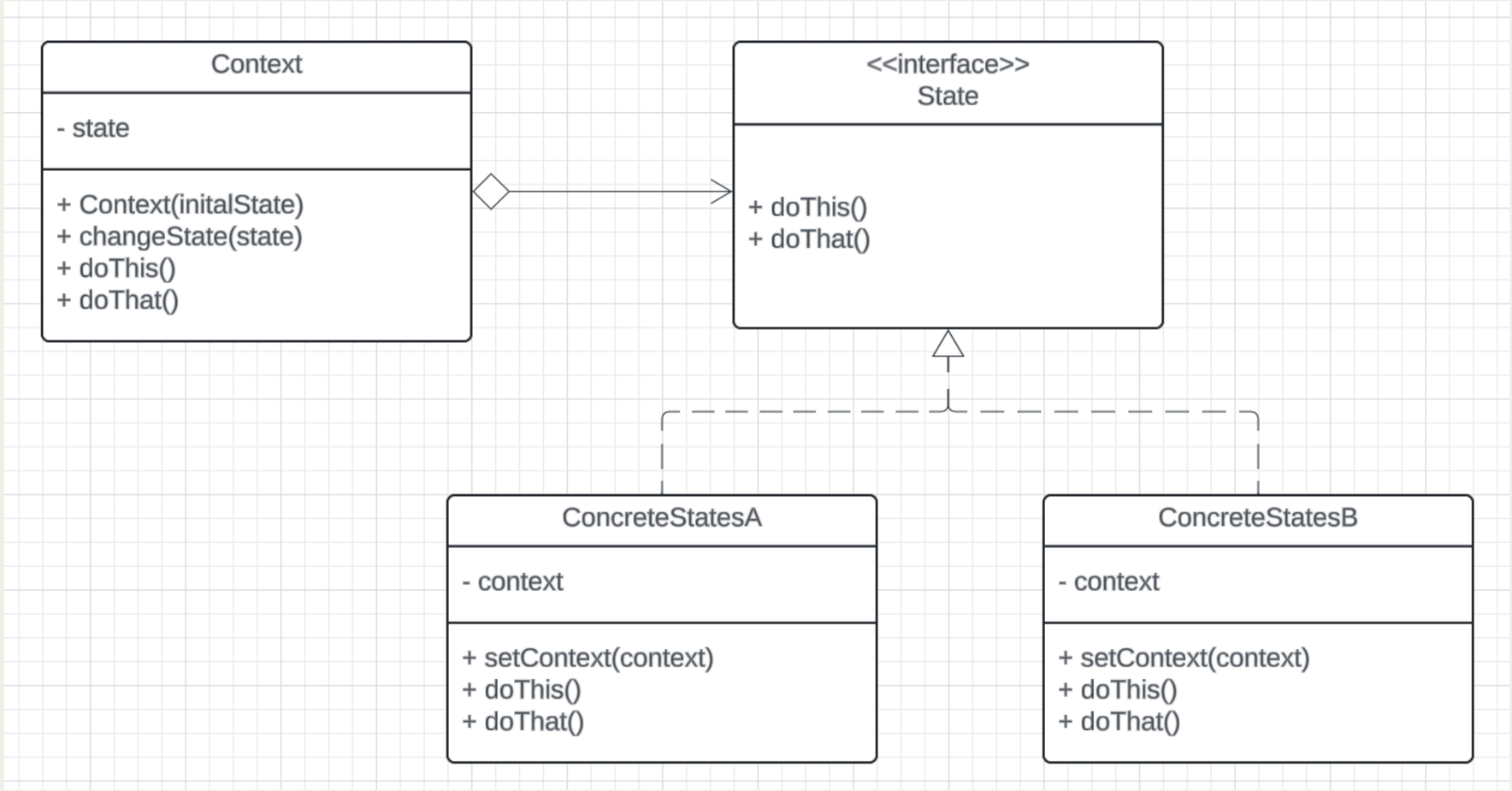


STATE PATTERN

Allows an object to alter its behaviour at run-time when its internals state changes.

- Similar to a state machine
- Clean way for an object to partially change its type at run-time
- Cannot add new functionality at run-time, can only switch
- Reduces conditional complexity (less if/switch statements)
- Encapsulates state-based behaviour and uses delegation to switch between behaviours

STATE PATTERN



STATE PATTERN

Continues from previous exercise/demo.
Extend your solution to accommodate the following requirements:

- Users can view a video, and a viewing has a series of states:
  - Playing state - the video is playing
  - Ready state - the video is paused, ready to play
  - Locked state - the video is temporarily 'locked' from being watched

State	Locking	Playing	Next
Locked	If playing, switch to Ready State	Switch to Ready State	Return Error: Locked
Playing	Stops playing and switches to Locked State	Pauses video and switches to Ready State	Starts playing the next video
Ready	Changes to locked state	Starts playback and changes to playing state	Returns Error: Locked (cannot move to next video while paused)

LAB LAB

REEEEEEE