Design patterns

Author: Daria Shutina

Source: YT Design Patterns

Creational patterns

Provide more flexibility in how the objects are actually created.

Singleton

Guarantees that only one instance of the class exists.

Simple implementation:

```
// `instance` field is static, since `getInstance` method needs to be static
public class Singleton {
    private static Singleton instance;
    private String data;

    private Singleton(String data) {
```

```
        this.data = data;
    }

    public static Singleton getInstance(String data) {
        if (instance == null) {
            instance = new Singleton(data);
        }
        return instance;
    }
}


class Main {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance("aboba");
    }
}
```

Thread-safe implementation:

```
public class Singleton {
    private static volatile Singleton instance;
    private String data;

    private Singleton(String data) {
        this.data = data;
    }

    public static Singleton getInstance(String data) {
        // use `result` variable, since `instance` is volatile
        // thus, we read directly from the main memory only once
        Singleton result = instance;
        if (result == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton(data);
                }
            }
        }
        return result;
    }
}
```

Factory

Separates the product's construction code from the code that uses this product.

It relies heavily on inheritance.

```
// Product interface
```

```
interface Product {
    void use();
}


// Concrete products
class ConcreteProductA implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA");
    }
}

class ConcreteProductB implements Product {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductB");
    }
}


// "Factory" class
class Creator {
    public static Product createProduct(String type) {
        if (type.equalsIgnoreCase("A")) {
            return new ConcreteProductA();
        } else if (type.equalsIgnoreCase("B")) {
            return new ConcreteProductB();
        }
        throw new IllegalArgumentException("Invalid product type: " + type);
    }
}


// the products are used here
public class Main {
    public static void main(String[] args) {
        Product productA = ProductFactory.createProduct("A");
        productA.use();  // Output: Using ConcreteProductA

        Product productB = ProductFactory.createProduct("B");
        productB.use();  // Output: Using ConcreteProductB
    }
}
```

Abstract factory

Allows to produce a group of related objects without specifying their concrete classes.

```
// Abstract Product A
interface ProductA {
    void use();
```

```
}

// Concrete Product A1
class ConcreteProductA1 implements ProductA {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA1");
    }
}

// Concrete Product A2
class ConcreteProductA2 implements ProductA {
    @Override
    public void use() {
        System.out.println("Using ConcreteProductA2");
    }
}









// Abstract Product B
interface ProductB {
    void interact(ProductA productA);
}

// Concrete Product B1
class ConcreteProductB1 implements ProductB {
    @Override
    public void interact(ProductA productA) {
        System.out.println("Interacting with ConcreteProductA1");
        productA.use();
    }
}

// Concrete Product B2
class ConcreteProductB2 implements ProductB {
    @Override
    public void interact(ProductA productA) {
        System.out.println("Interacting with ConcreteProductA2");
        productA.use();
    }
}









// Abstract Factory
interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}

// Concrete Factory 1
```

```java
class ConcreteFactory1 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}

// Concrete Factory 2
class ConcreteFactory2 implements AbstractFactory {
    @Override
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    @Override
    public ProductB createProductB() {
        return new ConcreteProductB2();
    }
}




// Client code
public class Main {
    public static void main(String[] args) {
        AbstractFactory factory1 = new ConcreteFactory1();
        ProductA productA1 = factory1.createProductA();
        ProductB productB1 = factory1.createProductB();
        productB1.interact(productA1);
        // Output:
        // Interacting with ConcreteProductA1
        // Using ConcreteProductA1

        AbstractFactory factory2 = new ConcreteFactory2();
        ProductA productA2 = factory2.createProductA();
        ProductB productB2 = factory2.createProductB();
        productB2.interact(productA2);
        // Output:
        // Interacting with ConcreteProductA2
        // Using ConcreteProductA2
    }
}

```

A real-world example is when there are two independent factories that can produce laptops and phones. Here, `productA = laptop` and `productB = phone`, but products at the first factory differ from product at the second factory.

Builder

Allows to produce different types and representations of an object using the same construction process. We extract the object creation code out of its class and move it to separate objects called *builders*.

- the builder has the same fields as the class
- the builder has setter-methods for every field and a `build()` method responsible for creating the class instance
- no-argument constructor is used for creating the builder

This is the basic idea:

```
public class Car {
    private final String brand;
    private final String color;
    private final String model;

    // constructor should be package-private of protected
    Car(String brand, String color, String model) {
        this.brand = brand;
        this.color = color;
        this.model = model;
    }
}



public class CarBuilder {
    private String brand;
    private String color;
    private String model;

    public CarBuilder brand(String brand) {
        this.brand = brand;
        return this;
    }

    public CarBuilder color(String color) {
        this.color = color;
        return this;
    }

    public CarBuilder model(String model) {
        this.model = model;
        return this;
    }

    public Car build() {
        return new Car(brand, color, model);
```

```
    }
}



class Main {
    public static void main(String[] args) {
        CarBuilder builder = new CarBuilder()
                .brand("Mitsubishi").color("red").model("b612");
        Car car = builder.build();
    }
}
```

Sometimes the same creation code is used to create several objects (for example, creating many `Bugatti` cars and many `Lamborgini` cars). In this case, we can use a **Director** -- a class which defines specific configurations depending on the case.

```
public class Director {
    public void buildBugatti(CarBuilder builder) {
        builder.brand("Bugatti")
                .color("Blue")
                .model("bugatti model");
    }

    public void buildLamborghini(CarBuilder builder) {
        builder.brand("Lamborghini")
                .color("Yellow")
                .model("lambo model");
    }
}
```

Using a director is optional. Still, its advantage is that it completely hides the details of the product construction from the client code:

```
public static void main(String[] args) {
    Director director = new Director();
    CarBuilder builder = new CarBuilder();

    director.buildBugatti(builder);

    Car car = builder.build();
}
```

Prototype

Every class that supports cloning is called a *prototype*.

The idea of the Prototype pattern is to delegate the object cloning process to the actual objects that are being cloned.

- the class should have a copy constructor and a `clone()` method overriden from the parent class
- the `clone()` method invokes the copy constructor
- the copy constructor both copies values of the class's fields and invokes the parent's copy constructor

```
public abstract class Vehicle {
    private final String color;
    private final String brand;

    protected Vehicle(String color, String brand) {
        this.color = color;
        this.brand = brand;
    }

    // copy constructor – should be protected
    protected Vehicle(Vehicle other) {
        this.color = other.color;
        this.brand = other.brand;
    }

    public abstract Vehicle clone();
}
```

```
public class Car extends Vehicle {
    private final String model;

    public Car(String color, String brand, String model) {
        super(color, brand);
        this.model = model;
    }

    protected Car(Car other) {
        super(other);
        this.model = other.model;
    }

    @Override
    public Car clone() {
        return new Car(this);
    }
}
```

```
public class Bus extends Vehicle {
    private final int amountOfPlaces;
```

```
3
4       Bus(String color, String model, int amountOfPlaces) {
5           super(color, model);
6           this.amountOfPlaces = amountOfPlaces;
7       }
8
9       Bus(Bus other) {
10          super(other);
11          this.amountOfPlaces = other.amountOfPlaces;;
12      }
13
14      @Override
15      public Vehicle clone() {
16          return new Bus(this);
17      }
18  }
```

Shallow vs Deep copy

Imagine, we have fields of a reference type. For example, we use a class `Engine` as a field for a `Car` class:

```
1   public class Car extends Vehicle {
2       private final String model;
3       private final Egnine engine;
4
5       // constructor
6
7       protected Car(Car other) {
8           super(other);
9           this.model = other.model;
10          this.engine = other.engine; // <------ shallow copy
11          //          = other.engine.clone(); <------ deep copy
12      }
13
14      @Override
15      public Car clone() {
16          return new Car(this);
17      }
18  }
```

- Shallow copy = we simply assign a value from the `other`'s field

```
public class Car extends Vehicle {
    private final Egnine engine;
    ...

    protected Car(Car other) {
        super(car);
        // assign values to other `Car` fields
        this.engine = other.engine;
    }
}
```

- Deep copy = we use `clone()` to get a copy of the `other`'s field

```
public class Car extends Vehicle {
    private final Egnine engine;
    ...

    protected Car(Car other) {
        super(car);
        // assign values to other `Car` fields
        this.engine = other.engine.clone();
    }
}
```

Behavioral patterns

Are about communication and assignment of responsibilities between objects.

The Chain of Responsibility

Transforms particular behaviors into stand-alone objects called *handlers*.

The basic idea is to delegate tasks to a proper handler. Each handler must either process a request or pass it along the chain.

A chain of handlers can be implemented as a linked list (using the `next` field). The client may trigger any handler in the chain, not only the first one.

Also, the Chain of Responsibility pattern allows to insert, remove or reorder handlers dynamically.

Example

Basically, the `Handler` class has a `next` field, an abstract `handle()` method:

```
public abstract class Handler {
    private Handler next;
```

```
3
4       public Handler setNextHandler(Handler next) {
5           this.next = next;
6           return next;
7       }
8
9       protected boolean handleNext(String username, String password) {
10          if (next == null)
11              return true;
12          return next.handle(username, password);
13      }
14
15      public abstract boolean handle(String username, String password);
16  }
```

Imagine we have a verification process. There are a few steps to check before the user is logged in:

1. Validate username
2. Validate password
3. Check role

For each step, we create a handler:

```
1   public class ValidUserHandler extends Handler {
2       private final Database database;
3
4       public ValidUserHandler(Database database) {
5           this.database = database;
6       }
7
8       @Override
9       public boolean handle(String username, String password) {
10          if (!database.isValidUsername(username)) {
11              System.out.println("The user '" + username + "' does not exist");
12              return false;
13          }
14          return handleNext(username, password);
15      }
16  }
17
18
19  public class ValidPasswordHandler extends Handler {
20      private final Database database;
21
22      public ValidPasswordHandler(Database database) {
23          this.database = database;
24      }
25
26      @Override
27      public boolean handle(String username, String password) {
28          if (!database.isValidPassword(username, password)) {
29              System.out.println("Wrong password for user '" + username + "'");
30          }
31          return handleNext(username, password);
```

```
    }
}


public class RoleCheckHandler extends Handler {
    private final String adminUsername;

    public RoleCheckHandler(String adminUsername) {
        this.adminUsername = adminUsername;
    }

    @Override
    public boolean handle(String username, String password) {
        if (adminUsername.equals(username)) {
            System.out.println("Loading Admin Page...");
        } else {
            System.out.println("Loading Default page...");
        }
        return handleNext(username, password);
    }
}
```

Finally, lets create a chain of handlers:

```
class Main {
    public static void main(String[] args) {
        Database database = new Database();
        Handler handler = new ValidUserHandler(database)
                .setNextHandler(new ValidPasswordHandler(database))
                .setNextHandler(new RoleCheckHandler("admin_username"));

        handler.handle("abobaUser", "abobaPwd");
    }
}
```

Instead of invoking `handle()` method explicitly, a class `AuthService` can be created:

```
public class AuthService {
    private final Handler handler;

    public AuthService(Handler handler) {
        this.handler = handler;
    }

    public boolean logIn(String username, String password) {
        if (handler.handle(username, password)) {
            System.out.println("Authorization was successful");
            return true;
        }
        return false;
    }
}


```

```
class Main {
    public static void main(String[] args) {
        Database database = Database.getInstance();
        Handler handler = // create a chain of handlers

        AuthService service = new AuthService(handler);
        service.logIn("abobaUser", "abobaPwd");
    }
}
```

Command

The idea is to turn a request (= a command) into a stand-alone object that contains everything about that request.

Thus, encapsulation principle is achieved. Every class has its own responsibility: it does not know about the inside structure of the command, it just executes the command.

The Command pattern opens a lot of interesting uses:

- passing commands as method arguments
- storing them inside other objects
- switching commands at runtime
- serializing commands, making it easy to write them to and read them from a file

Example

Basically, there is an interface `Command` which has a method `execute()`:

```
public interface Command {
    void execute();
}
```

There can be several concrete commands that implement `Command`. There is an `Invoker` (responsible for initiating requests, i.e. the user) and a `Reciever` (who invokes the command execution).

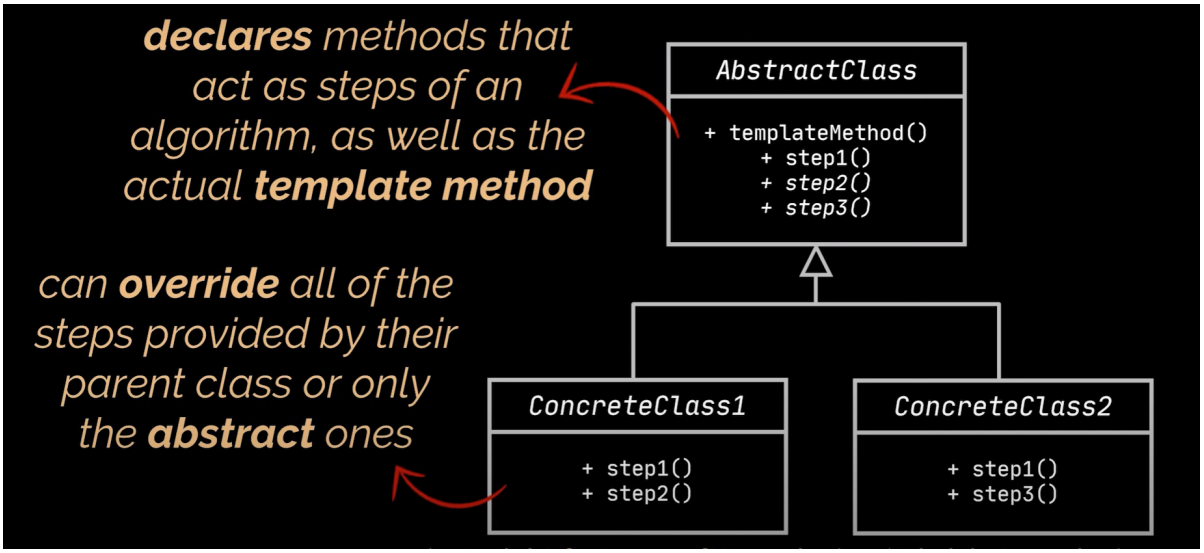
In the example below, there is a command to turn on (or turn off) the lights. There are classes for different rooms, each of them extends the `Room` class.

```
public class SwitchLightsCommand implements Command {
    private final Light light;

    public SwitchLightsCommand(Light light) {
        this.light = light;
    }

    @Override
```

```
    public void execute() {
        light.switchLights();
    }
}


public class Room {
    Command command;

    public Room() {}

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeCommand() {
        command.execute();
    }
}


class Main {
    public static void main(String[] args) {
        Room livingRoom = new LivingRoom();
        livingRoom.setCommand(
            new SwitchLightsCommand(new Light())
        );

        livingRoom.executeCommand();
    }
}
```

Template method

The Template Method pattern defines the skeleton of an algorithm in the superclass, and lets subclasses override some needed steps of the algorithm without changing its structure.



Simply, the idea is to break down the algorithm into a series of methods, then put a series of calls to these methods (called *steps*) inside a single "template method". The steps can be abstract, or have some default implementation inside the parent class.

Mediator

Defines an object that encapsulates how a set of objects interact with one another. It restricts direct communications between objects and forces them to collaborate via a mediator, hence reducing the dependencies between them (a real example, a dispatcher and pilots).

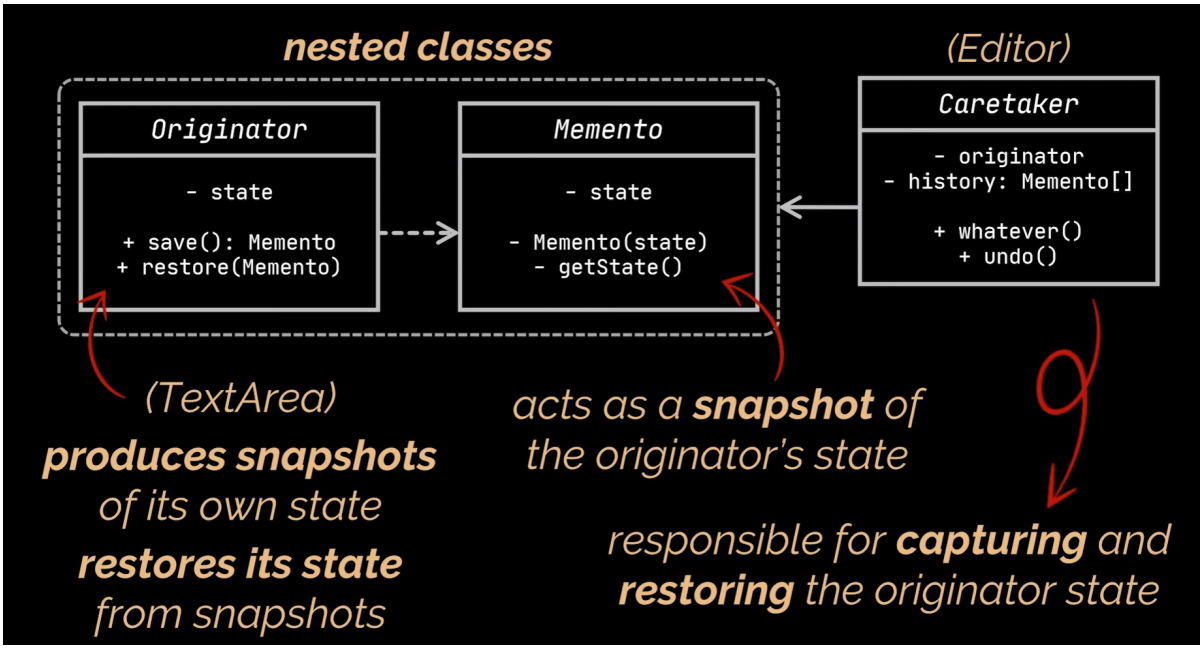
Memento

Delegates creating the snapshots to the actual owner of that state. Thus, the Single Responsibility principle is not violated.

- The implementation of the Memento pattern relies on the nested classes (see `TextArea` below)
- The full copies of the object are done
- The Caretaker class (in the example, `Editor`) delegates the creation of the object's state snapshot to the object itself

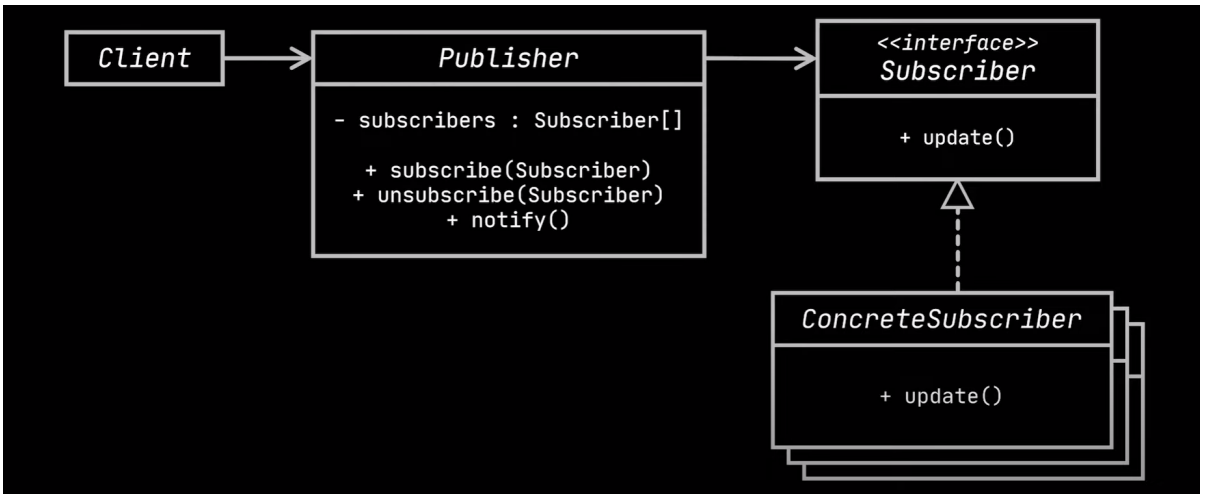
```
class TextArea {
    private String text;

    public void set(String text) {
        this.text = text;
    }

    public Memento takeSnapshot() {
        return new Memento(this.text);
    }

    public void restore(Memento memento) {
        this.text = memento.getSavedText();
    }

    public static class Memento {
        private final String text;

        // avaiable only for `TextArea`
        private Memento(String textToSave) {
            text = textToSave;
        }

        private String getSavedText() {
            return text;
        }
    }
```

```
}


class Editor {
    private Deque<Memento> history;
    private TextArea textArea;

    public Editor() {
        history = new LinkedList<>();
        textArea = new TextArea();
    }

    public void write(String text) {
        textArea.set(text);
        history.add(textArea.takeSnapshot());
    }

    public void undo() {
        textArea.restore(history.pop());
    }
}
```



Observer

Allows to change or take action on a set of objects when the state of another object changes. This can be done even if the set of objects is unknown or changes dynamically.



The Observer pattern comes in handy, when there are different types of notifications (i.e., by email or in the app). A `Publisher` (i.e., the shop) sends emails or push notifications to `Subscribers` (i.e., customers).

If there is only one way of notifying subscribers, please, **do not use the pattern**.

So, in the example below, the store sends notifications about new items. It can do it both by email and in the app:

```
public class NotificationService {
    private final List<Subscriber> customers;

    public NotificationService() {
        customers = new LinkedList<>();
    }

    public void subscribe(Subscriber subscriber) {
        customers.add(subscriber);
    }

    public void unsubscribe(Subscriber subscriber) {
        customers.remove(subscriber);
    }

    public void notifySubscribers() {
        customers.forEach(Subscriber::update);
    }
}


public class Store {
    private final NotificationService notificationService;

    public Store() {
        notificationService = new NotificationService();
    }

    public void newItemPromotion() {
        notificationService.notifySubscribers();
    }
```

```
32
33      public NotificationService getNotificationService() {
34          return notificationService;
35      }
36  }
```

The subscribers are:

```
1   public abstract class Subscriber {
2       abstract public void update();
3   }
4
5
6   public class EmailSubscriber extends Subscriber {
7       private final String email;
8
9       public EmailSubscriber(String email) {
10          this.email = email;
11      }
12
13      public void update() {
14          // Actually send an email
15      }
16  }
17
18
19  public class PushNotificationSubscriber extends Subscriber {
20      private final String username;
21
22      public PushNotificationSubscriber(String username) {
23          this.username = username;
24      }
25
26      @Override
27      public void update() {
28          // Actually send a push notification
29      }
30  }
```

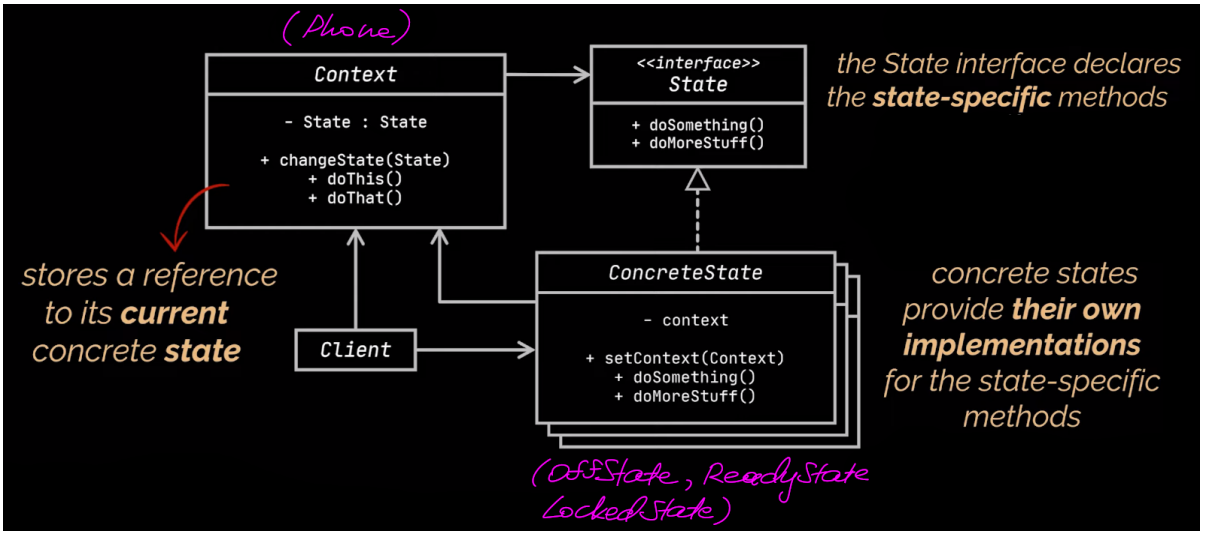
```
1   public class Main {
2       public static void main(String[] args) {
3           Store store = new Store();
4           store.getNotificationService()
5                   .subscribe(new EmailSubscriber("aboba@gmail.com"))
6                   .subscribe(new EmailSubscriber("abober@mail.ru"))
7                   .subscribe(new PushNotificationSubscriber("aboba2003"));
8
9           store.newItemPromotion();
10      }
11  }
```

```
Sending email to aboba@gmail.com
Sending email to abober@mail.ru
Sending push notification to aboba2003
```

Strategy

State

The pattern allows an object alter its behavior when its internal state changes.



For example, we have a phone which has three states: `OffState` (phone screen is off), `LockedState` (phone screen is on and locked) and `ReadyState` (phone is unlocked, we see home-screen). In reality, states are changed using Power and Home buttons.

```
public class Phone {
    private State state;

    public Phone() {
        state = new OffState(this);
    }

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

    public String lock() {
        return "Locking phone and turning off the screen";
    }

    public String home() {
        return "Going to home-screen";
    }

    public String unlock() {
        return "Unlocking the phone to home";
    }

    public String turnOn() {
        return "Turning screen on, device still locked";
    }
}
```

```
public class OffState extends State {
    public OffState(Phone phone) {
        super(phone);
    }

    @Override
    public String onHome() {
        phone.setState(new LockedState(phone));
        return phone.turnOn();
    }

    @Override
    public String onOffOn() {
        phone.setState(new LockedState(phone));
        return phone.turnOn();
    }
}
```

```
public abstract class State {
    protected Phone phone;

    public State(Phone phone) {
        this.phone = phone;
    }

    public abstract String onHome();
    public abstract String onOffOn();
}
```

```
public class ReadyState extends State {
    public ReadyState(Phone phone) {
        super(phone);
    }
    @Override
    public String onHome() {
        return phone.home();
    }
    @Override
    public String onOffOn() {
        phone.setState(new OffState(phone));
        return phone.lock();
    }
}
```

```
public class LockedState extends State {
    public LockedState(Phone phone) {
        super(phone);
    }

    @Override
    public String onHome() {
        phone.setState(new ReadyState(phone));
        return phone.unlock();
    }

    @Override
    public String onOffOn() {
        phone.setState(new OffState(phone));
        return phone.lock();
    }
}
```

Difference with Strategy pattern

States can be dependent and you can easily jump from one state to another. State pattern is about doing different things based on the current state (i.e., different results depending of the phone state).

Strategies are completely independent and unaware of each other. Strategy pattern is about having different implementations that accomplish the same thing (i.e.,).

Structural patterns

Deal with inheritance and composition to provide extra functionality.

Adapter

Allows objects with incompatible interfaces to collaborate with one another. It creates a middle-layer class that serves as a translator.

Architecturally, the adapter class is implemented using both inheritance and composition. It extends (or implements) the object with basic functionality and wraps the object which third-party functionality.

For example, we have an app with restaurant reviews which data is transferred in the XML format. We want to use a third-party service which works with data in JSON format. For this, we need an adapter which converts XML to JSON.

```
public interface IRestoApp {
    void displaysMenus(XmlData xmlData);
}

public class RestoApp implements IRestoApp {
    // constructor

    @Override
    public void displaysMenus(XmlData xmlData) {
        // Display menus using XML data
    }
}

// some third-party service
public class FancyUIService {
    // constructor

    public void displayMenus(JsonData jsonData) {
        // Display menus using JSON data
    }
}
```

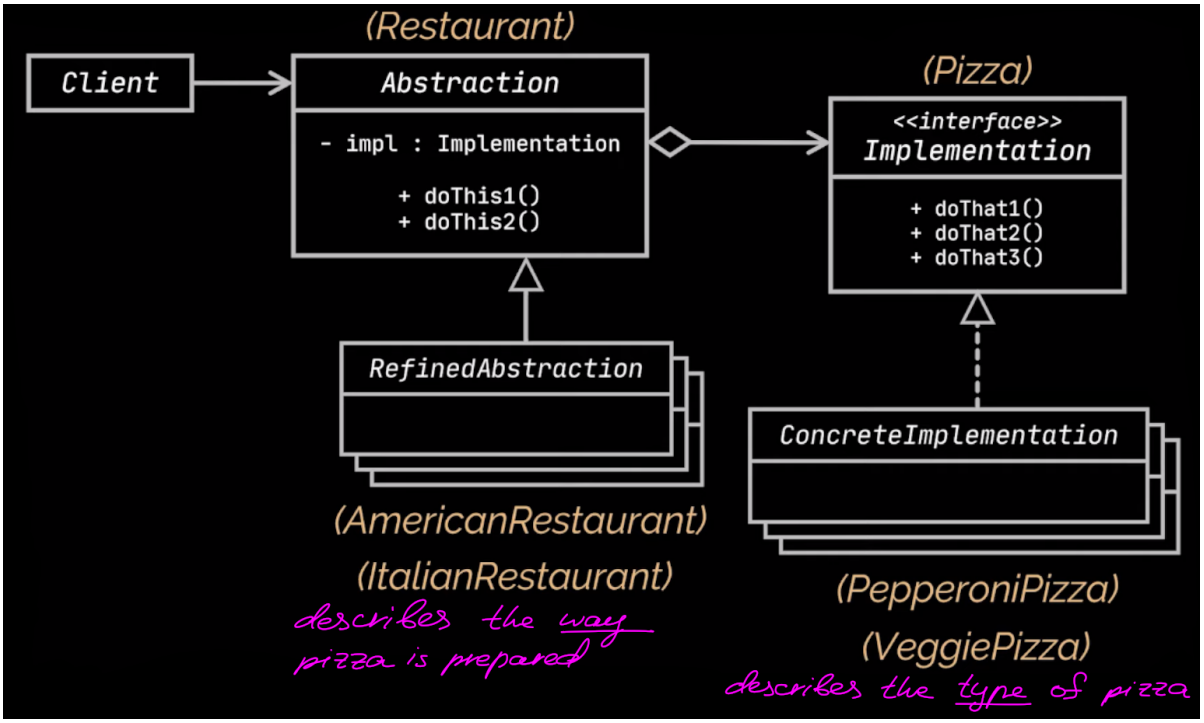
```java
public class FancyUIServiceAdapter implements IRestoApp {
    private final FancyUIService fancyUIService;

    public FancyUIServiceAdapter() {
        fancyUIService = new FancyUIService();
    }

    @Override
    public void displayMenus(XmlData xmlData) {
        JsonData jsonData = convertXmlToJson(xmlData);
        fancyUIService.displayMenus(jsonData);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        FancyUIServiceAdapter adapter = new FancyUIServiceAdapter();
        XmlData data = new XmlData();

        adapter.displayMenus(data);
    }
}
```

Bridge

The pattern splits a large class into two separate hierarchies which can be developed independently.

Two hierarchies are called **abstractions** and **implementations**. Abstraction is a high-level control layer. It delegates the work to the implementation layer.



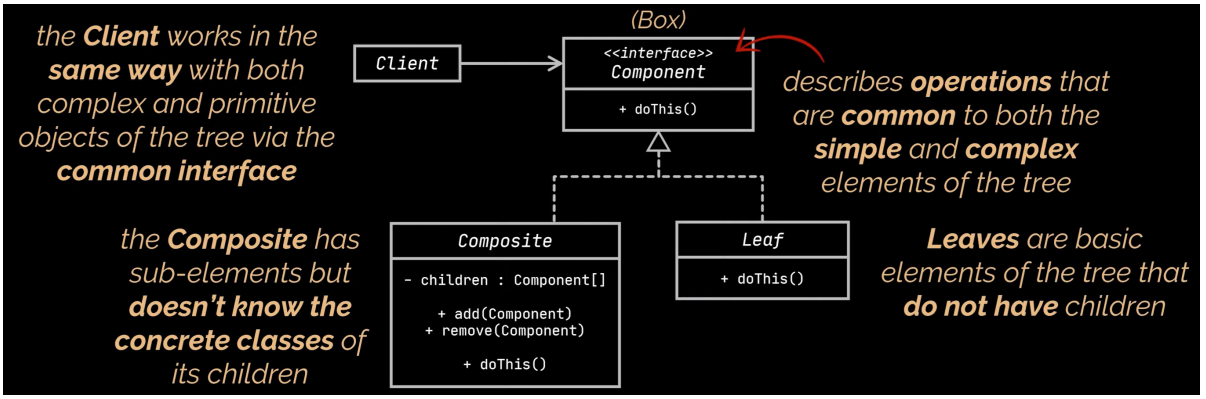
The Client works only with abstractions and does not care about the implementation details. Still, it is his/her responsibility to link the abstraction with the implementation:

```
public static void main(String[] args) {
    // link abstraction with implementation
    AmericanRestaurant restaurant = new AmericanRestaurant(new
PepperoniPizza());

    restaurant.deliver();
}
```

Composite

Applicable only if objects can form a tree. For example, an Amazon box may contain a product or a smaller Amazon box.

The pattern composes objects into tree elements and then works with these elements as if they were individual objects.



All elements share a **common interface** allowing the Client to treat Leaves and Composites uniformly (regarding the Amazon example, the common interface may have a `calculatePrice` method which will be overwritten in Leaves and Composites).

Decorator

The pattern lets you attach new behaviors to an object by placing the object inside a special wrapper.

For example, we can send notifications by mail using the `MailNotifier` class:

```
public inteface INotifier {
    void send(String msg);
    String getUsername();
}

public class MailNotifier implements INotifier {
    private final String username;
    private final DatabaseService databaseService;
```

```
9
10      public MailNotifier(String username) {
11          this.username = username;
12          databaseService = new DatabaseService();
13      }
14
15      public void send(String msg) {
16          String mail = databaseService.getMailFromUsername(username);
17          System.out.println ("Sending " + msg + " by Mail to " + mail);
18      }
19
20      public String getUsername() { return username; }
21  }
```

Now we want to add more types of notifications. For each type, we have a separate class -- `WhatsAppNotifier`, `FacebookNotifier`, etc. -- which implements `INotifier` interface.

Now we want to create a class which combines *two* types of notifiers (i.e., `WhatsAppFacebookNotifier`). Of course, we can do it. But what if the amount of possible combinations is much larger?

Instead, we can use decorators -- a separate decorator for each type of notifications.

```
1   // wrapper for the initial notifier (Mail, in our example)
2   public abstract class BaseNotifierDecorator implements INotifier {
3       private final INotifier wrapped;
4       protected final DatabaseService databaseService;
5
6       BaseNotifierDecorator(INotifier wrapped) {
7           this.wrapped = wrapped;
8           databaseService = new DatabaseService();
9       }
10
11      @Override
12      public void send(String msg) { wrapped.send(msg); }
13
14      @Override
15      public String getUsername() { return wrapped.getUsername(); }
16  }
17
18
19  // wrapper for WhatApp notifier
20  public class WhatsAppDecorator extends BaseNotifierDecorator {
21      public WhatsAppDecorator(INotifier wrapped) {
22          super(wrapped);
23      }
24
25      @Override
26      public void send(String msg) {
27          super.send(msg);
28          String phoneNumber =
    databaseService.getPhoneNbrFromUsername(getUsername());
29          System.out.println("Sending " + msg + " by WhatsApp to " +
    phoneNumber);
30      }
```

```
}


// wrapper for Facebook notifier
public class FacebookDecorator extends BaseNotifierDecorator {
    public FacebookDecorator(INotifier wrapped) {
        super(wrapped);
    }

    @Override
    public void send(String msg) {
        super.send(msg);
        String fbName = databaseService.getFbNameFromUsername(getUsername());
        System.out.println("Sending " + msg + " by Facebook to " + fbName);
    }
}
```

In this example, we consider Mail as a compulsory and initial type of notifications:

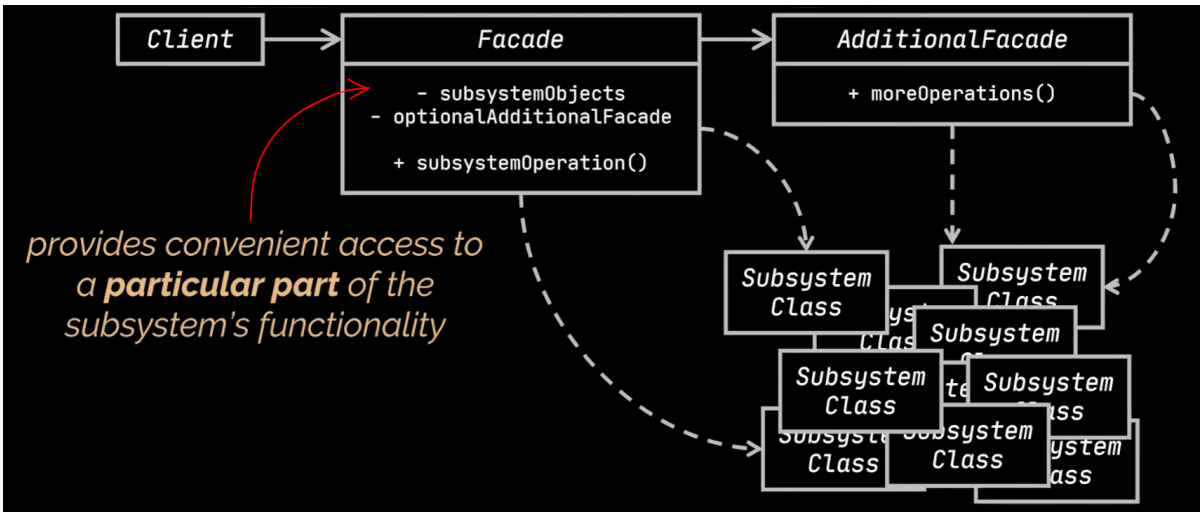
```
public static void main(String[] args) {
    INotifier notifier = new FacebookDecorator(
        new WhatsAppDecorator(
            new MailNotifier("username")
        )
    );

    notifier.send("To be or not to be? That is the question.");
}
```

When the `FacebookDecorator` class is instantiated, it invokes the constructor of `BaseNotifierDecorator` with the `wrapped` field equal to the provided argument -- `new WhatsAppDecorator(...)`. The same is with instantiating `WhatsAppDecorator`.

Here is where all the magic happens.

Facade

The idea is to provide a simplified interface to the client. The client uses the facade instead of calling the subsystem objects directly.



The code might become too dependent on the facade: the class grows too big and becomes a God object. Additional facades are added in such cases to prevent polluting the facade with unrelated features.

Flyweight

The Flyweight pattern lets you fit more objects into the available RAM by **sharing common parts** of state between multiple objects, instead of storing all of the data in each object individually.

For example, we run a book shop, and there is a `Book` class describing properties of the book:

```
public record Book (
    private final String name;
    private final String author;
    private final String type;
    private final String distributor;
    private final double price;
) {}
```

Suppose, three fields `author`, `type` and `distributor` are repeating for a lot of times. Lets create a flyweight class `BookType`:

```
public record BookType(
    String autor,
    String type,
    String distributor
) {}
```

The Flyweight factory returns flyweight possibilities that we have:

```java
public class BookTypeFactory {
    private static final Map<String, BookType> bookTypes = new HashMap<>();

    public static BookType getBookType(String autor, String type, String distributor) {
        if (!bookTypes.contains(type)) {
            bookTypes.put(type, new BookType(author, type, distributor));
        }
        return bookTypes.get(type);
    }
}
```

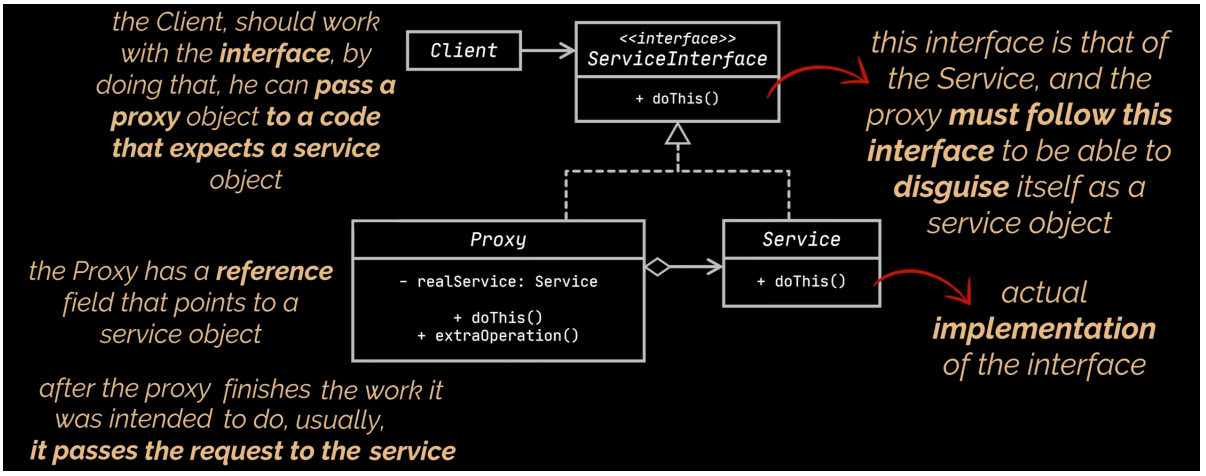
The class is flyweight meaning that, for particular books, it is invariant, context-independent, shareable and immutable at runtime. In turn, attributes inside the class may vary at runtime.

Now the book store works like this:

```java
pubic class BookStore {
    private final List<Book> books = new ArrayList<>();

    public void storeBook(String name, String author, String type,
                          String distirbutor, double price) {
        BookType bookType = BookTypeFactory.getBookType(autor, type, distributor);
        books.add(bew Book(name, price, bookType));
    }
}
```

Proxy

The pattern provides a substitute for another object and controls access to that object. It can be a "proxy server" that denies access to banned websites, or a cashing mechanism that remembers which videos you have already downloaded and refuses downloading them again.



The Proxy must implement the same interface of the original object. Thus, the Client can use Proxy at the same places where the Server is used.