/ JAVA DB SCHOOL

Design Patterns

/ Overview

Design Patterns

- Ease the development of applications and software architectures

- Allow programmers to make their program reusable

- A design pattern has:

1. A name – usually composed of one or two words
2. A problem – specifies a problem or a context to be solved
3. The solution – describes the architecture, responsibilities and relations between components. It defines an abstract description of solving the problem
4. Consequences – shows the results and compromises involved in applying the pattern

Categories

By purpose:

- Creational – manage the way objects are being created
- Structural – manage classes organization
- Behavioural – manage communication patterns between objects

By domain:

- Classes Patterns – define relations between classes and subclasses
- Object Patterns – define relations between objects

Categories

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory method	Adapter (class)	Interpreter Template method
	Object	Abstract factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

/ Creational Design Patterns

Singleton

- Ensures that a specific class can create only one instance of a class
- Examples: classes used for: a file explorer system, a database mapper, etc.

```
private static Singleton uniqueInstance;

private Singleton() { }

public static Singleton instanta() {
    if (uniqueInstance == null) {
        uniqueInstance = new Singleton();
    }
    return uniqueInstance;
}
```

Factory

- Defines an interface for creating objects
- Allows subclasses to choose an implementation

```
class GeometricObject { }
class Circle extends GeometricObject { }
class Square extends GeometricObject { }
class Rectangle extends GeometricObject { }

class Factory {
    public GeometricObject createGeometricObject(String type) {
        switch (type) {
            case "circle": return new Circle();
            case "square": return new Square();
            case "rectangle": return new Rectangle();
        }
        return null;
    }
}
```

Builder

- Intends to separate the creation of an object from its representation
- Step by step approach

```
class House {
    private boolean hasFoundation;
    private boolean hasFirstStore;
    private boolean hasSecondStore;
    private boolean hasRoof;

    public House(HouseBuilder houseBuilder) {
        this.hasFoundation = houseBuilder.hasFoundation;
        this.hasFirstStore = houseBuilder.hasFirstStore;
    }

    @Override
    public String toString() {
        return "House{" +
                "hasFoundation=" + hasFoundation +
                ", hasFirstStore=" + hasFirstStore +
                '}';
    }
```

Builder

```
...
static class HouseBuilder {
    private boolean hasFoundation;
    private boolean hasFirstStore;

    public HouseBuilder() { }

    public HouseBuilder hasFoundation(boolean hasFoundation) {
        this.hasFoundation = hasFoundation;
        return this;
    }

    public HouseBuilder hasFirstStore(boolean hasFirstStore) {
        this.hasFirstStore = hasFirstStore;
        return this;
    }

    public House build() {
        House house = new House(this);
        return house;
    }
}
```

Builder

```
    ...
    public static void main() {
        House house = new House.HouseBuilder()
                .hasFoundation(true)
                .hasFirstStore(true)
                .build();
        System.out.println(house);
    }

}
```

/ Structural Design Patterns

Decorator (Wrapper)

- Attaches additional responsibilities to an object, in a dynamic way without affecting other objects

- An alternative to class extension

```
class GeometricObjectDecorator extends GeometricObject {
    protected GeometricObject decoratedObject;
    public GeometricObjectDecorator(GeometricObject decoratedObject) {
        this.decoratedObject = decoratedObject;
    }
}

class ColorDecorator extends GeometricObjectDecorator {
    public ColorDecorator(GeometricObject decoratedObject) {
        super(decoratedObject);
    }
}
```

/ Behavioral Design Patterns

Observer

- Allows mapping one-to-many relations between objects, such that when an object changes its state, it notifies all its subscribers (observers)

- Constituted by a subject and its observers

Observer

```
class Subject {
    private int state;
    public Observer observer;
    private static Subject subject;
    public int getState() {
        return state;
    }

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

    public void attachObserver(Observer observer) {
        this.observer = observer;
    }

    public void notifyObserver() {
        observer.receiveUpdate();
    }
}
```

Observer

```
class Observer {
    Subject subject;

    public Observer(Subject subject) {
        this.subject = subject;
        this.subject.attachObserver(this);
    }

    public void receiveUpdate() {
        System.out.println("State = " + subject.getState());
    }

    public static void main(String[] args) {
        subject = new Subject();
        Observer observer = new Observer(subject);
        subject.setState(1);
        subject.setState(2);
    }
}
```

Command

- An object is used to encapsulate information needed to perform an action at a later time
- The information consists of the method name, the object owning the method, and parameters for the method
- Usually implemented with a queue, where commands are added and wait to be peeked

Command

```
abstract class Command {
    public abstract void execute(Dog dog);

    public static void main(String[] args) {
        Dog dog = new Dog();
        CommandStay c1 = new CommandStay();
        CommandBring c2 = new CommandBring();
        c1.execute(dog);
        c2.execute(dog);
    }
}
```

Command

```
class Dog {
    public void stay() {
        System.out.println("Dog received stay command");
    }

    public void bring() {
        System.out.println("Dog received bring command");
    }
}

class CommandStay extends Command {
    public void execute(Dog dog) {
        dog.stay();
    }
}
class CommandBring extends Command {
    public void execute(Dog dog) {
        dog.bring();
    }
}
```

/ Practice, practice, practice

Modern Pizza Factory

Implement a PizzaFactory that can create the following pizzas: Marguerita, ProsciuttoFunghi, Capriciosa and QuatroStagioni.

- Make the PizzaFactory a singleton class
- Implement a decorator that can add topping to each type of pizza
- Add a Client class and implement an Observer inside PizzaFactory. Allow clients to be notified about new baked pizzas.
- Add a Command pattern to allow clients to give orders to the PizzaFactory.
- Test the class in a main method

/ Q&A


MOBILE/ACADEMY