VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED PROGRAMMING

ASSIGNMENT REPORT

Lecturer: Truong Tuan Anh

Student: Tran Nguyen Gia Phat - 2153681

HO CHI MINH CITY, 12th June 2023

Table of Contents

I. INTRODUCTION TO DESIGN PATTERNS....4

II. TYPES OF DESIGN PATTERNS....5

2.1. CREATIONAL PATTERNS....5

2.1.1. SINGLETON....5

2.1.2. FACTORY....7

2.1.3. ABSTRACT FACTORY....10

2.1.4. BUILDER....14

2.1.5. PROTOTYPE....19

2.2. STRUCTURAL PATTERNS....22

2.2.1. ADAPTER....22

2.2.2. COMPOSITE....26

2.2.3. PROXY....30

2.2.4. FLYWEIGHT....32

2.2.5. FAÇADE....37

2.2.6. BRIDGE....40

2.2.7. DECORATOR....43

2.3. BEHAVIOURAL PATTERNS....48

2.3.1. TEMPLATE METHOD....48

2.3.2. MEDIATOR....50

2.3.3. CHAIN OF RESPONSIBILITY....54

2.3.4. OBSERVER....58

2.3.5. STRATEGY 62
2.3.6. COMMAND 65
2.3.7. STATE 69
2.3.8. VISITOR 73
2.3.9. ITERATOR 77
2.3.10. INTERPRETER 80
2.3.11. MEMENTO 83
III. NEW PROGRAMMING APPROACHES 86
3.1. DATA WRANGLING 86
3.2. SMART CONTRACT 89
IV. REFERENCES 91

I. INTRODUCTION TO DESIGN PATTERNS

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Design patterns can speed development by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause significant problems and improves code readability for coders and architects familiar with the patterns.

Gangs Of Four Design Patterns was first published in 1994 and is one of the most popular books for learning design patterns. The book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

In summary, 23 design patterns in OOP can be divided into three categories:

1. **Creational**: The design patterns that deal with the creation of an object.
2. **Structural**: The design patterns in this category deals with the class structure such as Inheritance and Composition.
3. **Behavioral**: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

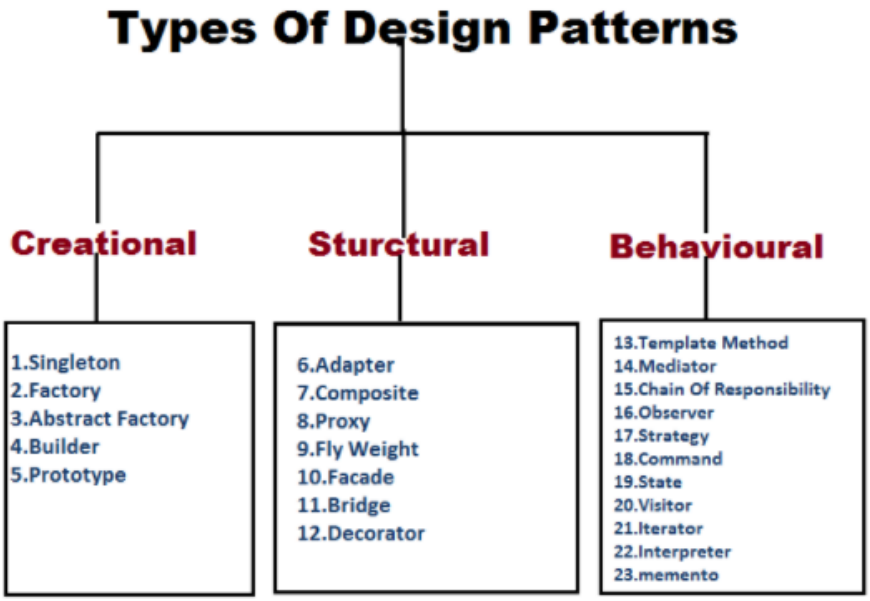


Figure 1: 23 Design Patterns

II. TYPES OF DESIGN PATTERNS

2.1. CREATIONAL PATTERNS

2.1.1. SINGLETON

The Singleton design pattern is a creational pattern that restricts the instantiation of a class to one object. It is used when only one class instance is required to coordinate actions across a system, such as a single database connection shared by multiple objects or a single configuration manager in an application.

There are two forms of singleton design pattern: Early Instantiation (creation of instance at load time) and Lazy Instantiation (creation of instance when required).

Here is an example of a Singleton implementation in Java:

```
public class Singleton {
    // Declare a private static variable to hold the Singleton instance
    private static Singleton instance = null;
    // Make the constructor private to restrict instantiation from outside
the class
    private Singleton() {// Initialization code here}
    // Provide a public static method to get the Singleton instance
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

To use the Singleton in the program, call the **getInstance()** method:

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

Advantages of Singleton Pattern:

- Ensures only a single instance of the class is created.
- Provides a global access point to the instance.

Disadvantages of Singleton Pattern:

- Violates the single responsibility principle by addressing two problems simultaneously.
- Challenging to write unit test cases due to potential changes in the global state.
- Risk of debugging errors when other components or modules alter the global state.

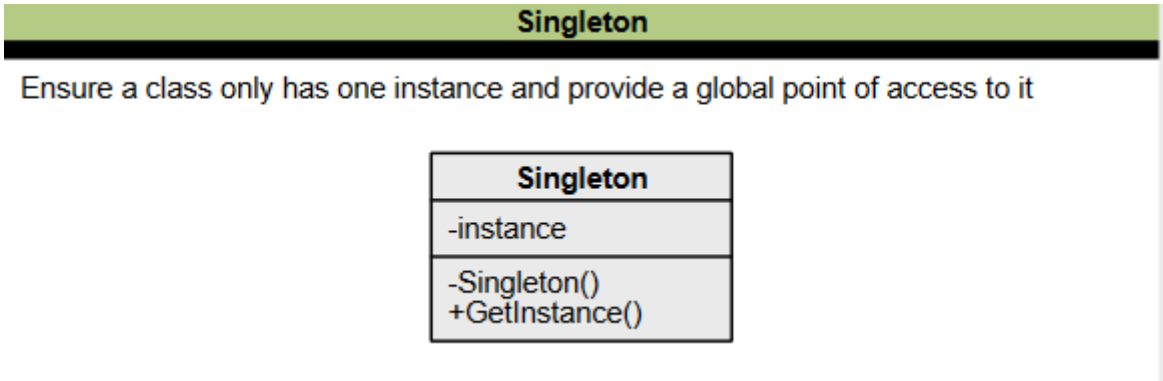


Figure 2: Class diagram of Singleton pattern

2.1.2. FACTORY

The Factory Design Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It is used when there is a need to create objects based on some input or configuration without specifying the exact class to be instantiated. This pattern promotes loose coupling and makes the code more robust, less coupled, and easy to extend.

Implementation:

1. Define a factory method inside an interface.
2. Let the subclass implements the above factory method and decides which object to create.

Here is an example of a Factory implementation in Java. First, we create an interface **Shape**:

```
public interface Shape {
    void draw();
}
```

Next, we define the concrete classes for each shape:

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Now, we create a **ShapeFactory** class that will be responsible for creating the appropriate shape object based on the input:

```
public class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        } else if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        }
        return null;
    }
}
```

Finally, the client code can use the **ShapeFactory** to create shape objects and call their draw method:

```
public class Client {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        Shape rectangle = shapeFactory.getShape("RECTANGLE");
        rectangle.draw();

        Shape circle = shapeFactory.getShape("CIRCLE");
        circle.draw();
    }
}
```

In this example, we have decoupled the object creation from the client code. The **ShapeFactory** class is responsible for deciding which object type to create based on the input provided by the client. The client needs to call the factory class's **getShape** method and pass the desired shape type without worrying about the actual implementation of the creation of objects.

Advantages of Factory Pattern:

- Modular expandability of the application
- Good testability.
- Significant method names.

Disadvantages of Factory Pattern:

- High number of required classes.
- Extension of the application is very elaborate.

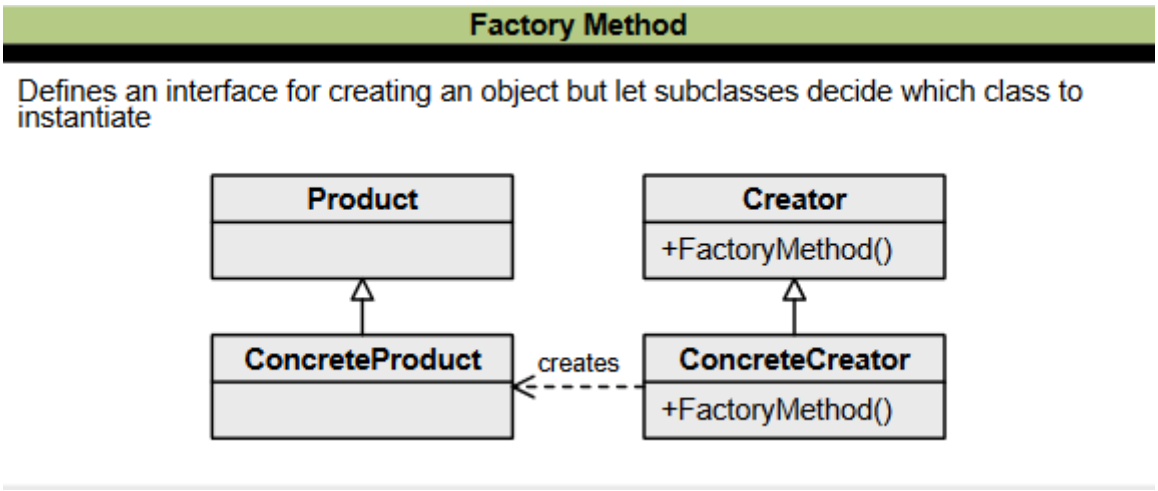


Figure 3: Class diagram of Factory pattern

2.1.3. ABSTRACT FACTORY

The Abstract Factory design pattern is a creational pattern that provides a framework for creating families of related or dependent objects without specifying their concrete classes. It is another abstraction layer over the Factory pattern, creating a super-factory that produces other factories.

The main components of the Abstract Factory pattern are:

1. **AbstractFactory**: An interface for operations that create abstract product objects.
2. **ConcreteFactory**: Implements the operations declared in the AbstractFactory to create concrete product objects.
3. **Product**: Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
4. **Client**: Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Let's consider an example to demonstrate the Abstract Factory pattern in Java. We will create a simple vehicle factory that produces cars and bikes. There are two types of vehicles: regular and sports. First, define the abstract products (interfaces for cars and bikes):

```
public interface ICar {
    void drive();
}
public interface IBike {
    void ride();
}
```

Next, create the concrete products (classes that implement the interfaces):

```
public class RegularCar implements ICar {
    @Override
    public void drive() {
        System.out.println("Driving a regular car");
    }
}
public class SportsCar implements ICar {
    @Override
    public void drive() {
        System.out.println("Driving a sports car");
    }
```

```
}
public class RegularBike implements IBike {
    @Override
    public void ride() {
        System.out.println("Riding a regular bike");
    }
}
public class SportsBike implements IBike {
    @Override
    public void ride() {
        System.out.println("Riding a sports bike");
    }
}
```

Now, define the abstract factory (interface for creating vehicles):

```
public interface IVehicleFactory {
    ICar createCar();
    IBike createBike();
}
```

Create the concrete factories (classes that implement the abstract factory interface):

```
public class RegularVehicleFactory implements IVehicleFactory {
    @Override
    public ICar createCar() {
        return new RegularCar();
    }
    @Override
    public IBike createBike() {
        return new RegularBike();
    }
}
public class SportsVehicleFactory implements IVehicleFactory {
    @Override
    public ICar createCar() {
        return new SportsCar();
    }
    @Override
    public IBike createBike() {
        return new SportsBike();
    }
```

```
}
```

Finally, the client uses the abstract factory and abstract product interfaces to create the desired objects:

```
public class Client {
    public static void main(String[] args) {
        IVehicleFactory factory = new RegularVehicleFactory();
        ICar car = factory.createCar();
        car.drive();

        IBike bike = factory.createBike();
        bike.ride();

        factory = new SportsVehicleFactory();

        car = factory.createCar();
        car.drive();

        bike = factory.createBike();
        bike.ride();
    }
}
```

In this example, the client code is decoupled from the concrete classes of the products. It only uses the interfaces provided by the Abstract Factory and Abstract Product classes, making adding new types of vehicles or factories easy without modifying the client code.

Differences:

- The main difference between a “factory method” and an “abstract factory” is that the factory method is a single method, and an abstract factory is an object.
- The factory method is just a method that can be overridden in a subclass, whereas the abstract factory is an object with multiple factory methods.
- The Factory Method pattern uses inheritance and relies on a subclass to instantiate the desired object.

Advantages of Abstract Factory Pattern:

- It provides a way to create families of related objects without specifying their concrete classes.
- It promotes loose coupling between the client code and the concrete implementations, allowing for easier maintenance and extensibility.

Disadvantages of Abstract Factory Pattern:

- It can result in a complex codebase due to the introduction of multiple abstract factory interfaces and their corresponding concrete implementations.
- Adding new product variants may require modifying the existing codebase, which can be cumbersome in larger projects.

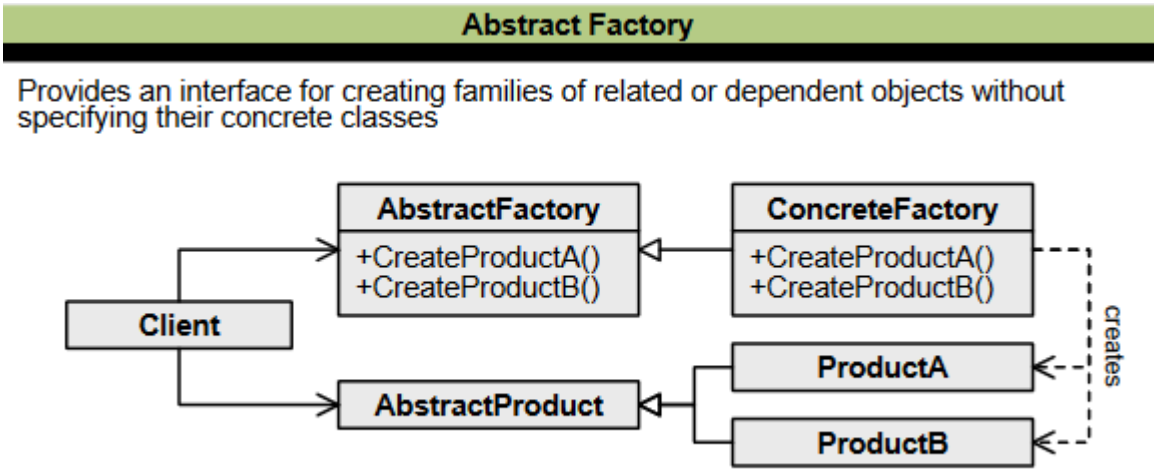


Figure 4: Class diagram of Abstract Factory pattern

2.1.4. BUILDER

The Builder Design Pattern is a creational pattern that deals with constructing complex objects. It separates the construction process from its representation, allowing the same construction process to create different representations of the same object. The pattern is beneficial when the object creation process is complex and involves multiple steps.

The Builder pattern involves four main components:

1. **Abstract Builder**: An interface defining all the steps required to create the concrete product.
2. **Concrete Builder**: A class implementing the Abstract Builder interface, providing an implementation for all the abstract methods, and responsible for constructing and assembling the individual parts of the product.
3. **Director**: A class that takes individual processes from the Builder and defines the sequence to build the product.
4. **Product:** The class of the object we want to create using the Builder pattern.

Implementation:

1. Create a static nested class (the Builder class) within the class you want to build (the Product class). The naming convention is to name the Builder class with the Product class name followed by "Builder."
2. Copy all the arguments from the Product class to the Builder class.
3. Create a public constructor in the Builder class with all the required attributes as parameters.
4. Create methods in the Builder class to set the optional parameters. These methods should return the same Builder object after setting the optional attribute.
5. Provide a build() method in the Builder class that will return the Product object needed by the client program. It would help to have a private constructor in the Product class with the Builder class as an argument.

Here is an example of Builder pattern in Java. First we create the Product class:

```
class Pizza {
    private String dough;
    private String sauce;
    private String topping;
    public void setDough(String dough) {
        this.dough = dough;
    }
    public void setSauce(String sauce) {
        this.sauce = sauce;
    }
    public void setTopping(String topping) {
        this.topping = topping;
    }
    public String getDescription() {
        return "Pizza with " + dough + " dough, " + sauce + " sauce, and
" + topping + " topping.";
    }
}
```

Next we create the abstract Builder class:

```
abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
        return pizza;
    }
    public void createNewPizza() {
        pizza = new Pizza();
    }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Then define the Concrete Builder classes:

```
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("thin crust");
    }
    public void buildSauce() {
        pizza.setSauce("tomato");
```

```
    }
    public void buildTopping() {
        pizza.setTopping("ham and pineapple");
    }
}
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() {
        pizza.setDough("thick crust");
    }
    public void buildSauce() {
        pizza.setSauce("spicy tomato");
    }
    public void buildTopping() {
        pizza.setTopping("pepperoni and jalapeno");
    }
}
```

Now we create the Director class to handle concrete builder types:

```
// Director class
class Cook {
    private PizzaBuilder pizzaBuilder;
    public void setPizzaBuilder(PizzaBuilder pizzaBuilder) {
        this.pizzaBuilder = pizzaBuilder;
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza() {
        pizzaBuilder.createNewPizza();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

Finally, in the Client side:

```
public class Client {
    public static void main(String[] args) {
        Cook cook = new Cook();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
```

```
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        cook.setPizzaBuilder(hawaiianPizzaBuilder);
        cook.constructPizza();
        Pizza hawaiianPizza = cook.getPizza();
                        System.out.println("Hawaiian     Pizza:     "     +
hawaiianPizza.getDescription());

        cook.setPizzaBuilder(spicyPizzaBuilder);
        cook.constructPizza();
        Pizza spicyPizza = cook.getPizza();
       System.out.println("Spicy Pizza: " + spicyPizza.getDescription());
    }
}
```

In this example, we have a **Pizza** class that represents the product we want to build. We also have an abstract **PizzaBuilder** class and two concrete builder classes, **HawaiianPizzaBuilder** and **SpicyPizzaBuilder**, which inherit from the abstract builder.

The **Cook** class acts as a director and is responsible for constructing the pizza step by step using the chosen builder. The client code creates a cook, sets the desired builder, constructs the pizza, and then retrieves the constructed pizza object.

Using the Builder design pattern, you can separate the construction logic from the client code, allowing you to create different types of pizzas with various combinations of dough, sauce, and toppings in a flexible and organized manner.

Advantages of Builder Pattern:

- It provides a clear separation between the construction and representation of an object, making the code more readable and maintainable.
- It allows you to construct objects step by step, providing flexibility and control over the construction process, especially when dealing with complex objects.

Disadvantages of Builder Pattern:

- Requires the implementation of multiple classes (builders), which may increase code complexity.
- It can introduce additional overhead due to the creation of builder objects and method calls, especially for simple object construction scenarios.

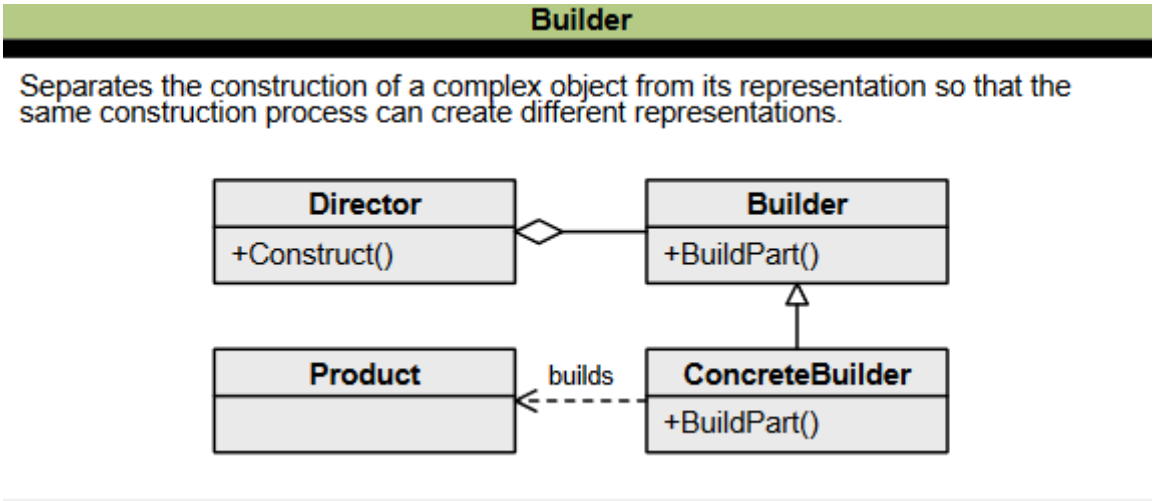


Figure 5: Class Diagram of Builder pattern

2.1.5. PROTOTYPE

The Prototype design pattern is a creational design pattern that allows you to create new objects by copying existing ones rather than creating them from scratch. This approach can save resources and time, especially when object creation is heavy or expensive.

The Prototype pattern involves three main components:

1. **Prototype**: This is the prototype of the actual object. It is usually an interface or an abstract class that defines methods for cloning itself.
2. **ConcretePrototype**: This class implements the Prototype interface or abstract class for cloning.
3. **Client**: The client is responsible for using the prototype to create new objects by asking it to clone itself.

Here is an example of a Prototype implementation in Java. First we create a prototype interface:

```
public interface Prototype {
    Prototype clone();
}
```

Next we create ConcretePrototype class:

```
public class ConcretePrototype implements Prototype {
    private String data;
    public ConcretePrototype(String data) {
        this.data = data;
    }
    @Override
    public Prototype clone() {
        return new ConcretePrototype(data);
    }
    public void displayData() {
        System.out.println("Data: " + data);
    }
}
```

Finally, in the Client side:

```
public class Client {
    public static void main(String[] args) {
            ConcretePrototype prototype = new ConcretePrototype("Hello,
World!");
        prototype.displayData();

            ConcretePrototype clonedPrototype = (ConcretePrototype)
prototype.clone();
        clonedPrototype.displayData();
    }
}
```

In this example, we define a **Prototype** interface with a **clone()** method. The **ConcretePrototype** class implements the **Prototype** interface and overrides the **clone()** method to create a new object with the same data. The **Client** class demonstrates how to use the **ConcretePrototype** to create and clone objects.

Advantages of Prototype Pattern:

- It allows for creating new objects by cloning existing ones, reducing the need for subclassing and the complexity of object creation.
- It provides a convenient way to produce multiple variations of an object, as each clone can be modified independently.

Disadvantages of Prototype Pattern:

- The cloning process can be complex and error-prone, especially when dealing with objects with complex internal structures or dependencies.
- Cloning can create duplicate objects with shared mutable states, resulting in unexpected behavior if not handled carefully.

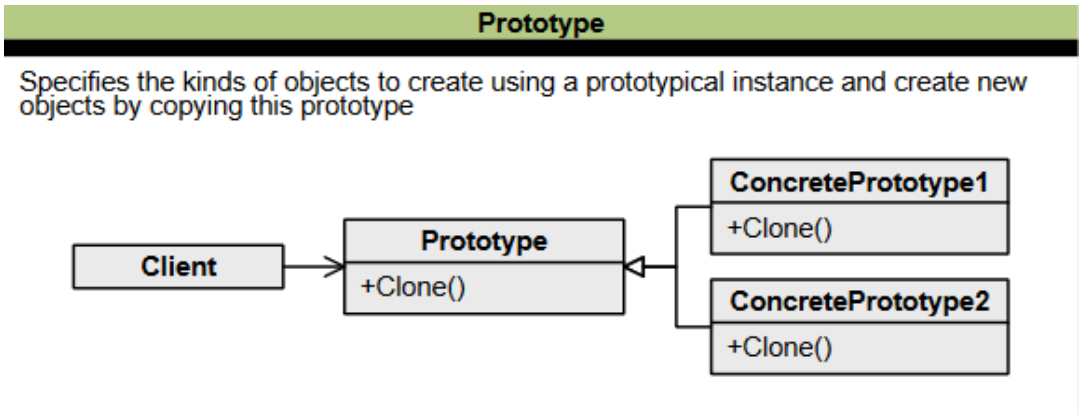


Figure 6: Class diagram of Prototype pattern

2.2. STRUCTURAL PATTERNS

2.2.1. ADAPTER

The Adapter Design Pattern is a structural design pattern that allows incompatible interfaces or objects to work together. It acts as a bridge between two incompatible objects, enabling communication and collaboration. The pattern helps manage codebase complexity, work with legacy systems, and reuse third-party libraries without making changes to the application.

The Adapter design pattern consists of the following main components:

1. **Target**: This is the desired interface that the client code expects to work with. It defines the methods the client will use to interact with the Adapter.
2. **Adapter**: This is the class that implements the Target interface. It acts as a bridge between the client code and the Adaptee. It adapts the interface of the Adaptee to match the Target interface. The Adapter class is responsible for translating the method calls from the client code to the appropriate method calls on the Adaptee.
3. **Adaptee**: This is the existing class or component that needs to be adapted. It has a different interface or method signature incompatible with the Target interface. The Adaptee class provides the actual implementation of the functionality that the client code wants to use.
4. **Client**: This code uses the Target interface to interact with the Adaptee indirectly through the Adapter. The client code is unaware of the Adaptee's existence and interacts with the Adapter as if it were an implementation of the Target interface.

The adapter pattern has two versions: the class adapter pattern (using inheritance) and the object adapter pattern (using composition). The object adapter pattern is more popular due to its flexibility and lack of side effects when changing existing code.

Here is an example implementation of Adapter pattern in Java. First we define the exising class (Legacy code):

```
// Adaptee
class LegacyRectangle {
```

```
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("LegacyRectangle: draw(x1: " + x1 + ", y1: " +
y1 + ", x2: " + x2 + ", y2: " + y2 + ")");
    }
}
```

Next we create Target interface and Adapter class:

```
// Target interface
interface Shape {
    void draw(int x, int y, int width, int height);
}
// Adapter
class RectangleAdapter implements Shape {
    private LegacyRectangle legacyRectangle;
    public RectangleAdapter(LegacyRectangle legacyRectangle) {
        this.legacyRectangle = legacyRectangle;
    }
    @Override
    public void draw(int x, int y, int width, int height) {
        int x1 = x;
        int y1 = y;
        int x2 = x + width;
        int y2 = y + height;
        legacyRectangle.draw(x1, y1, x2, y2);
    }
}
```

Finally, the Client side use only methods defined in Target interface

```
// Client code
public class Client {
    public static void main(String[] args) {
        LegacyRectangle legacyRectangle = new LegacyRectangle();
        Shape shapeAdapter = new RectangleAdapter(legacyRectangle);
        // The client code works with the Shape interface
        shapeAdapter.draw(10, 20, 50, 70);
    }
}
```

In this example, we have an existing **LegacyRectangle** class that uses a different method signature for drawing rectangles. We want to adapt this class to fit into our new system, which expects objects that implement the **Shape** interface.

To achieve this, we create the **RectangleAdapter** class, which implements the **Shape** interface. The adapter class takes an instance of **LegacyRectangle** in its constructor. When the draw method is called on the adapter, it converts the parameters to match the legacy method signature and delegates the call to the **LegacyRectangle** object.

The client code creates an instance of the **LegacyRectangle** class and wraps it with the **RectangleAdapter**. It can then call the draw method on the **Shape** interface, and the adapter internally handles the translation to the legacy method. This allows the client code to work with the **Shape** interface, while still being able to utilize the functionality of the legacy **LegacyRectangle** class.

Advantages of the Adapter Pattern:

- It provides compatibility between incompatible interfaces, allowing objects with different interfaces to work together.
- Allows to reuse existing classes without modifying their code, thus promoting code reusability and minimizing the impact on existing systems.

Disadvantages of the Adapter Pattern:

- It can add complexity to the codebase by introducing additional classes and indirection.
- This may lead to performance overhead due to the need for additional method invocations and conversions between interfaces.

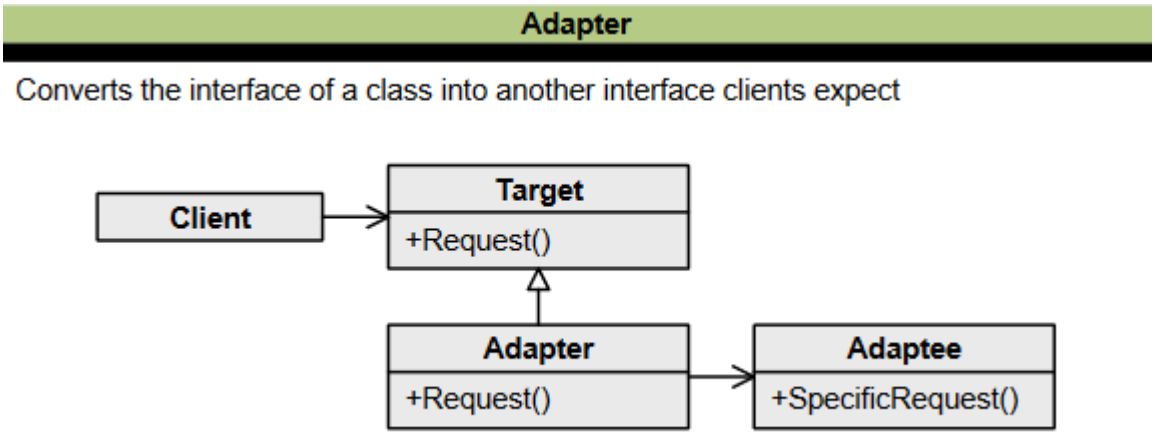


Figure 7: Class diagram of Adapter pattern

2.2.2. COMPOSITE

The Composite Design Pattern is a structural pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects and compositions of objects uniformly, which simplifies the client code when dealing with complex object structures.

The Composite pattern has four components:

1. **Component:** The component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface familiar to all classes as appropriate.
2. **Leaf:** Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
3. **Composite:** Composite stores child components and implements child-related operations in the component interface.
4. **Client:** The client manipulates the objects in the composition through the component interface.

Here is an example of using the Composite pattern in Java. First we create an interface that defines common methods:

```
import java.util.ArrayList;
import java.util.List;
// Component interface
interface Employee {
    void showDetails();
}
```

Next we create concrete class for Leaf and Composite:

```
// Composite class
class Manager implements Employee {
    private String name;
    private List<Employee> subordinates;
    public Manager(String name) {
        this.name = name;
        subordinates = new ArrayList<>();
    }
```

```
    public void add(Employee employee) {
        subordinates.add(employee);
    }
    public void remove(Employee employee) {
        subordinates.remove(employee);
    }
    @Override
    public void showDetails() {
        System.out.println("Manager: " + name);
        for (Employee employee : subordinates) {
            employee.showDetails();
        }
    }
}
// Leaf class
class Developer implements Employee {
    private String name;
    public Developer(String name) {
        this.name = name;
    }
    @Override
    public void showDetails() {
        System.out.println("Developer: " + name);
    }
}
```

Finally, in the Client code:

```
public class Client {
    public static void main(String[] args) {
        Employee john = new Developer("John");
        Employee sarah = new Developer("Sarah");
        Manager bob = new Manager("Bob");
        bob.add(john);
        bob.add(sarah);
        Employee mike = new Developer("Mike");
        Employee jane = new Developer("Jane");
        Manager alice = new Manager("Alice");
        alice.add(mike);
        alice.add(jane);
```

```
        alice.add(bob);
        alice.showDetails();
    }}
```

In this example, we have a company hierarchy represented using the Composite design pattern. The **Employee** interface defines the common operations for both managers and developers. The **Manager** class is a composite and can contain other employees (managers or developers) as its subordinates. The **Developer** class is a leaf and represents an individual employee.

In the **Client** class, we create a hierarchy of employees and print their details using the **showDetails()** method. The output will display the structure of the organization, including managers and developers.

Advantages of the Composite Pattern:

- Simplifies the client code by treating individual objects and compositions uniformly, enabling the client to work with both consistently.
- It allows adding new components to the structure without modifying the existing code.

Disadvantages of the Composite Pattern:

- This can make the design overly general, leading to potential performance overheads when working with complex structures.
- It may be unsuitable for scenarios where the leaf and composite objects have significantly different behavior or interfaces.

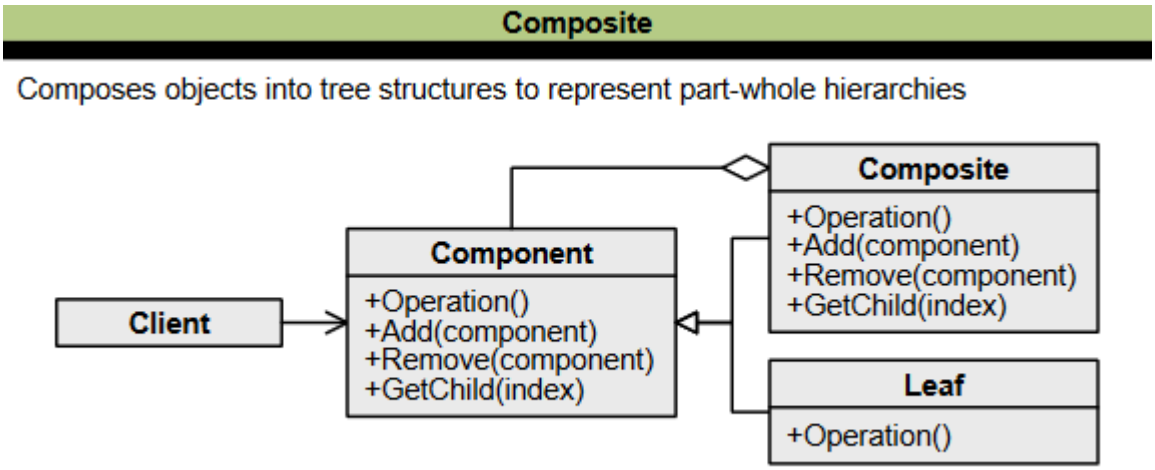


Figure 8: Class diagram of Composite pattern

2.2.3. PROXY

The Proxy design pattern is a structural design pattern that provides a substitute or placeholder for another object to control its access, creation, or behavior. It is used when we want to provide controlled access to functionality.

The Proxy pattern is helpful in the following cases:

- When you want to have a simplified version of an object or access the object more securely.
- When additional functionality should be provided when accessing an object, such as checking access rights for sensitive objects.

The main components of the Proxy design pattern are:

1. **Subject**: This defines the standard interface that both the RealSubject and Proxy classes implement, allowing the Proxy to substitute for the RealSubject.
2. **RealSubject:** This represents the object the Proxy class controls access to, providing functionality.
3. **Proxy:** This acts as a surrogate for the RealSubject, implementing the same interface as the Subject. It controls access to the RealSubject, adding extra functionality or performing additional tasks before or after accessing the real object.
4. **Client:** The Client is the code that interacts with the Proxy to perform operations on the RealSubject indirectly.

Here is an example of Proxy design pattern implementation in Java. First we create the Subject interface:

```
interface Image {
    void display();
}
```

Then we create RealSubject and Proxy classes inherited from the Subject:

```
// RealSubject class
class RealImage implements Image {
    private String filename;
```

```
    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }
    private void loadFromDisk() {
        System.out.println("Loading image from disk: " + filename);
    }
    @Override
    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}
// Proxy class
class ImageProxy implements Image {
    private RealImage realImage;
    private String filename;
    public ImageProxy(String filename) {
        this.filename = filename;
    }
    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

Finally, in the Client side:

```
// Client code
public class Client {
    public static void main(String[] args) {
        // Create a proxy object
        Image image = new ImageProxy("image.jpg");
        // The image is loaded only when required
        image.display();
        // The image is not loaded again
        image.display();
    }
```

```
}
```

Advantages of Proxy pattern:

- Provides a level of indirection, allowing for additional functionality or control over access to the underlying object.
- Supports lazy initialization, where the expensive object is created only when needed, improving performance.

Disadvantages of Proxy pattern:

- It introduces additional complexity and requires the creation of proxy objects.
- This can decrease performance if the proxy operations are computationally expensive or introduce significant overhead.

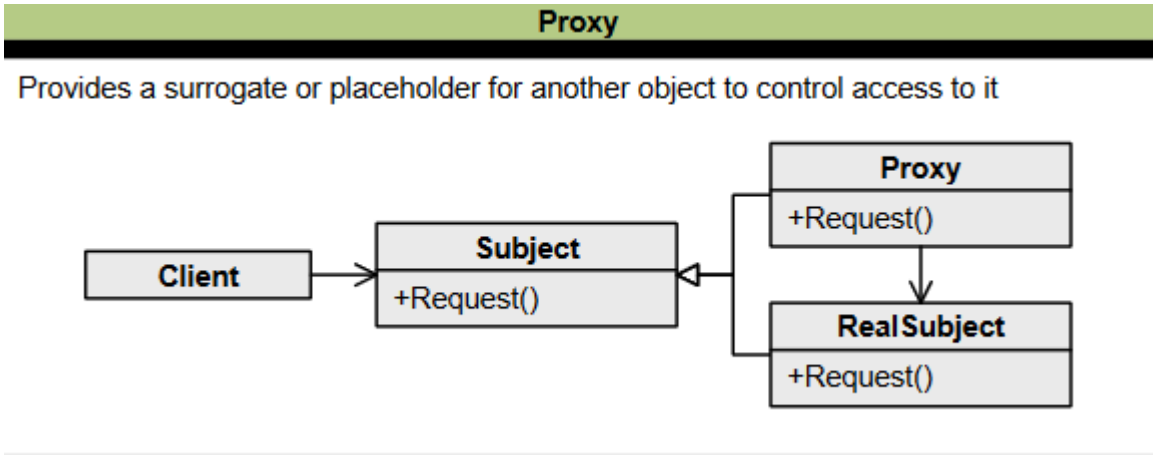


Figure 9: Class diagram of Proxy pattern

2.2.4. FLYWEIGHT

The Flyweight design pattern is a structural pattern that reduces memory usage and improves performance by sharing common parts of the state between multiple objects instead of keeping all the data in each object. This pattern is beneficial when there is a need to create objects of almost similar nature.

The state of a Flyweight object is composed of an intrinsic component, which is shared with similar objects, and an extrinsic component, which can be manipulated by client code. Intrinsic properties make the object unique, while extrinsic properties are set by the client code and used to perform different operations. To implement the Flyweight pattern, you need to:

1. Divide the object's properties into intrinsic and extrinsic properties.
2. Create a Flyweight factory that returns shared objects.

The main components of the Flyweight design pattern are:

1. **Flyweight interface:** Defines the standard interface that flyweight objects should implement.
2. **ConcreteFlyweight:** This represents the concrete implementation of a flyweight object and contains an intrinsic state that can be shared across multiple contexts.
3. **FlyweightFactory:** Manages creating and retrieving flyweight objects, ensuring their sharing and reuse.
4. **Client:** Uses flyweight objects and may store extrinsic states unique to each context.

Here is an example of Flyweight pattern in Java. First we create the Flyweight interface:

```
import java.util.HashMap;
import java.util.Map;
// Flyweight interface
interface Shape {
    void draw();
}
```

Then we define Concrete Flyweight and Flyweight Factory classes:

```
// Concrete flyweight
class Circle implements Shape {
```

```
    private String color;
    public Circle(String color) {
        this.color = color;
    }
    @Override
    public void draw() {
        System.out.println("Drawing Circle: Color - " + color);
    }
}
// Flyweight factory
class ShapeFactory {
    private static final Map<String, Shape> circleMap = new HashMap<>();
    public static Shape getCircle(String color) {
        Circle circle = (Circle) circleMap.get(color);
        if (circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating a new Circle: Color - " +
color);
        }
        return circle;
    }
}
```

Finally, in the Client code:

```
public class Client {
    public static void main(String[] args) {
        Shape redCircle = ShapeFactory.getCircle("Red");
        redCircle.draw();
        Shape greenCircle = ShapeFactory.getCircle("Green");
        greenCircle.draw();
        Shape blueCircle = ShapeFactory.getCircle("Blue");
        blueCircle.draw();
        Shape redCircle2 = ShapeFactory.getCircle("Red");
        redCircle2.draw();
    }
}
```

In this example, the Flyweight pattern is used to efficiently create and reuse **Circle** objects. The **ShapeFactory** acts as a flyweight factory and maintains a pool of existing **Circle** objects based on their color. The **Circle** class represents the concrete flyweight that provides the **draw()** method. The **Client** class acts as the client code that requests and uses the **Circle** objects from the ShapeFactory. The client code retrieves **Circle** objects from the factory and calls the **draw()** method on each object.

When the client requests a **Circle** object from the factory, it first checks if an object with the requested color already exists in the pool. If it exists, it returns the existing object. Otherwise, it creates a new **Circle** object, adds it to the pool, and returns the new object. This way, the Flyweight pattern ensures the efficient sharing and reuse of common objects, minimizing memory usage.

Advantages of Flyweight pattern:

- Reducing memory usage by sharing a common intrinsic state among multiple objects.
- Sharing flyweight objects can improve performance by avoiding the creation of unnecessary objects.

Disadvantages of Flyweight pattern:

- Implementing the Flyweight pattern may introduce additional complexity, especially when managing the extrinsic state.
- The Flyweight pattern is most effective when many similar objects have shared states, making it less suitable for scenarios with complex or varying states.

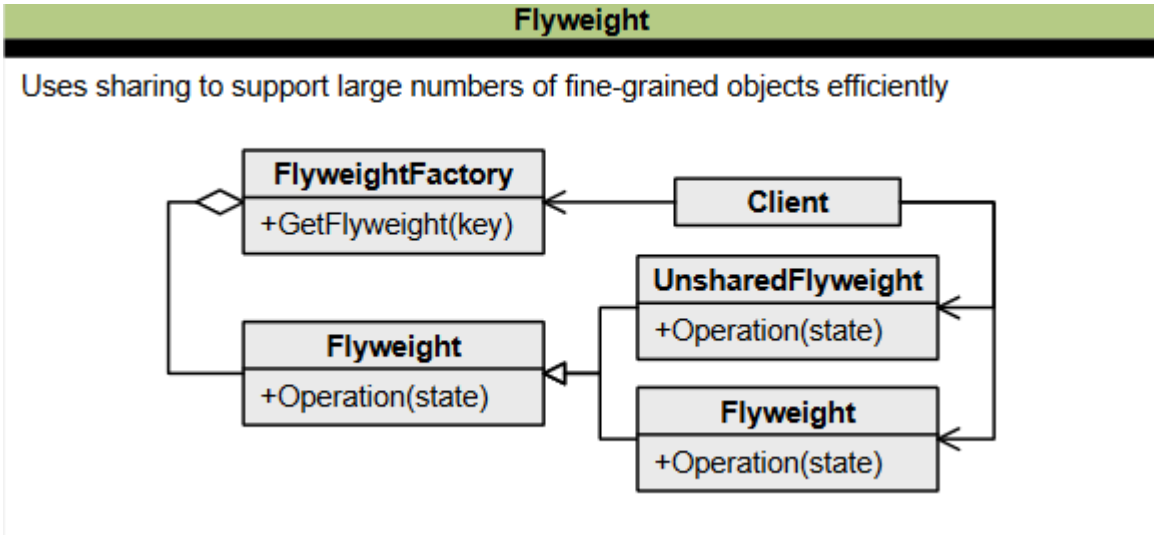


Figure 10: Class diagram of Flyweight pattern

2.2.5. FAÇADE

The Facade design pattern is a structural design pattern that provides a simplified interface to a complex system, library, or framework, making it easier to use and understand. It involves creating a single class that offers simplified methods required by the client while delegating calls to methods of existing system classes. This pattern hides the complexities of the underlying system and provides a more user-friendly experience.

The main components of the Facade pattern are:

1. **Facade**: Provides a simplified interface to a complex system or set of classes, hiding the underlying complexity.
2. **Subsystems**: The individual classes or components that make up the complex system. The facade delegates client requests to these subsystems.

Here is an simple implementation of Façade pattern. First we create subsystem classes:

```
class Subsystem1 {
    public void operation1() {
        System.out.println("Subsystem 1: Operation 1");
    }
}
class Subsystem2 {
    public void operation2() {
        System.out.println("Subsystem 2: Operation 2");
    }
}
class Subsystem3 {
    public void operation3() {
        System.out.println("Subsystem 3: Operation 3");
    }
}
```

Next we create Façade class:

```
class Facade {
    private Subsystem1 subsystem1;
    private Subsystem2 subsystem2;
    private Subsystem3 subsystem3;
    public Facade() {
```

```
        subsystem1 = new Subsystem1();
        subsystem2 = new Subsystem2();
        subsystem3 = new Subsystem3();
    }
    public void performOperation() {
        System.out.println("Facade: Performing operation using
subsystems...");
        subsystem1.operation1();
        subsystem2.operation2();
        subsystem3.operation3();
    }
}
```

Finally, in the Client code:

```
public class Client {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.performOperation();
    }
}
```

Advantages of Facade pattern:

- Provides a simplified interface to a complex system, making it easier to understand and use.
- Decouples the client code from the subsystems, allowing for easier maintenance and modifications.

Disadvantages of Facade pattern:

- If the facade performs many operations or involves multiple subsystems, there might be a performance overhead due to additional method calls and indirection.
- The facade may not expose all the functionalities of the underlying subsystems, limiting the flexibility for clients who require fine-grained control.

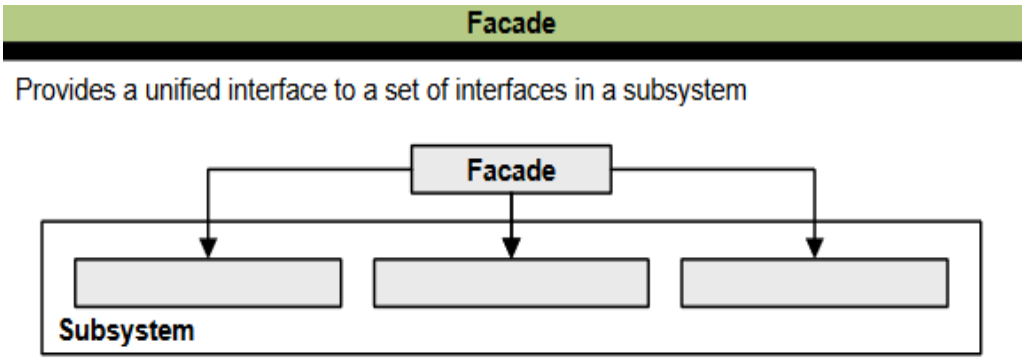


Figure 11: Class diagram of Façade pattern

2.2.6. BRIDGE

The Bridge design pattern is a structural design pattern that allows you to separate the abstraction from the implementation, enabling both to be developed independently. It is used to decouple an abstraction from its implementation so that the two can vary independently, which helps achieve loose coupling between class abstraction and its implementation.

The main components of the Bridge pattern are:

1. **Abstraction**: This is the core of the bridge design pattern and defines the main functionality. It contains a reference to the implementor.
2. **Refined Abstraction**: This is an extension of the abstraction that takes the finer details one level below and hides them from the implementers
3. **Implementor**: This defines the interface for implementation classes. The interface does not need to correspond directly to the abstraction interface and can be very different.
4. **ConcreteImplementor**: This provides the concrete implementation of the implementor by implementing the operations provided by the Implementor interface.

Here is an example of Bridge pattern in Java:

```
// Abstraction
interface Shape {
    void draw();
}
// Concrete Implementor
class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}
// Concrete Implementor
class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
```

```
}

// Implementor
interface Color {
    void applyColor();
}
// Concrete Implementor
class Red implements Color {
    @Override
    public void applyColor() {
        System.out.println("Applying Red color");
    }
}
// Concrete Implementor
class Blue implements Color {
    @Override
    public void applyColor() {
        System.out.println("Applying Blue color");
    }
}
// Refined Abstraction
class ShapeWithColor implements Shape {
    private final Color color;

    public ShapeWithColor(Color color) {
        this.color = color;
    }
    @Override
    public void draw() {
        System.out.print("Drawing ");
        color.applyColor();
    }
}
// Client
public class Client {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape redCircle = new ShapeWithColor(new Red());
        Shape blueRectangle = new ShapeWithColor(new Blue());
```

```
        circle.draw(); // Drawing Circle
        redCircle.draw(); // Drawing Red color
        blueRectangle.draw(); // Drawing Blue color
    }
}
```

In this example, we have an abstraction interface **Shape** that defines the **draw()** method. We also have two concrete implementor classes **Circle** and Rectangle that implement the Shape interface. The **Color** interface is another implementor interface, and we have two concrete implementor classes Red and Blue that implement it.

The **ShapeWithColor** class is a refined abstraction that extends the **Shape** interface and holds a reference to a **Color** object. It overrides the **draw()** method and delegates the color functionality to the **Color** object.

In the **Client,** we create instances of the **Shape** interface and use the **ShapeWithColor** class to add color functionality to the shapes. We can draw a plain circle, a red circle, or a blue rectangle by combining different shapes with different colors using the Bridge pattern.

Advantages of the Bridge pattern:

- It provides a clear separation between abstraction and implementation, allowing them to vary independently.
- It enhances extensibility by adding new abstractions and implementations without modifying existing code.

Disadvantages of the Bridge pattern:

- It can increase the complexity of the code by introducing additional abstraction and implementation layers.
- It may be overkill for simple scenarios with only one abstraction and one implementation.

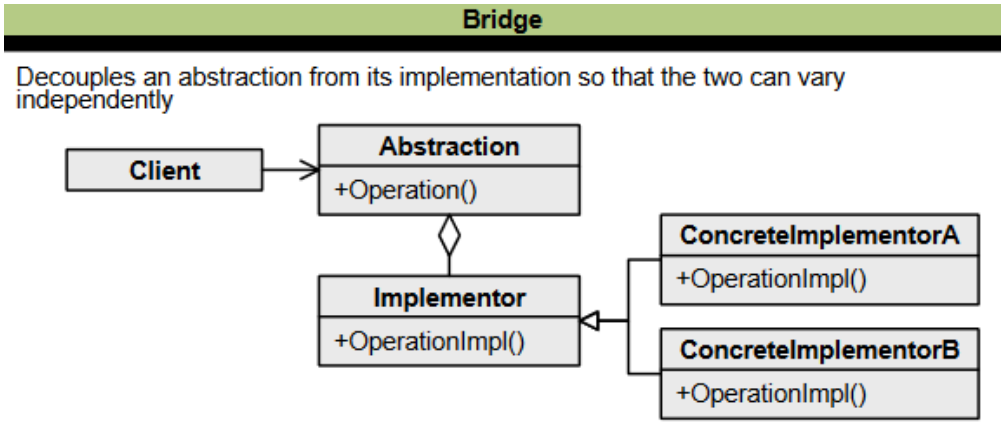


Figure 12: Class diagram of Bridge pattern

2.2.7. DECORATOR

The Decorator Design Pattern is a structural pattern that allows you to dynamically add functionality and behavior to an object without affecting the behavior of other existing objects within the same class. It is beneficial for adhering to the Single Responsibility Principle, as it enables functionality to be divided between classes with unique areas of concern. It also adheres to the Open-Closed Principle by allowing the functionality of a class to be extended without modification.

The critical components of the Decorator pattern include:

1. **Component**: This is an abstract class or interface that serves as the supertype for the decorator pattern.
2. **ConcreteComponent**: You can decorate these objects with different behaviors at runtime. They inherit from the component interface and implement its abstract functions.
3. **Decorator**: This class is abstract and has the same supertype as the object it will decorate. In the class diagram, you will see two relationships between the component and decorator classes. The first relationship is one of inheritance; every decorator is a component. The second relationship is one of composition; each decorator has a (or wraps a) component.
4. **Concrete Decorator**: These individual decorators give a component a specific behavior. You should note that each concrete decorator has an instance variable that holds a reference to a component.

Here is an example of Decorator implementation in Java.

```
// The interface for the component
interface Coffee {
    String getDescription();

    double getCost();
}
// Concrete component implementation
class SimpleCoffee implements Coffee {
```

```
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }
    @Override
    public double getCost() {
        return 1.0;
    }
}
// Decorator abstract class
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;
    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
// Concrete decorator adding milk
class Milk extends CoffeeDecorator {
    public Milk(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}
// Usage example
```

```
public class Client {
    public static void main(String[] args) {
        // Create a simple coffee
        Coffee coffee = new SimpleCoffee();
        System.out.println("Description: " + coffee.getDescription());
        System.out.println("Cost: $" + coffee.getCost());
        // Decorate the coffee with milk
        Coffee coffeeWithMilk = new Milk(coffee);
        System.out.println("Description: " +
coffeeWithMilk.getDescription());
        System.out.println("Cost: $" + coffeeWithMilk.getCost());
    }
}
```

In this example, we have a **Coffee** interface representing the component. The SimpleCoffee class implements the **Coffee** interface, providing the base implementation.

The **CoffeeDecorator** class is an abstract class that implements the **Coffee** interface. It acts as a base decorator class and holds a reference to a **Coffee** object. The **Milk** class is concrete decorator that extends the **CoffeeDecorator** class. It adds specific functionality to the decorated coffee by modifying the **getDescription** and **getCost** methods.

Advantages of Decorator pattern:

- It provides flexibility to add or modify the behavior of an object at runtime without affecting other objects.
- Allows for a combination of behaviors through the stacking of decorators.

Disadvantages of Decorator pattern:

- It can result in many small classes if there are many combinations of decorators, which can make the codebase more complex.
- The order of decorators may be necessary, and managing the order can become challenging.

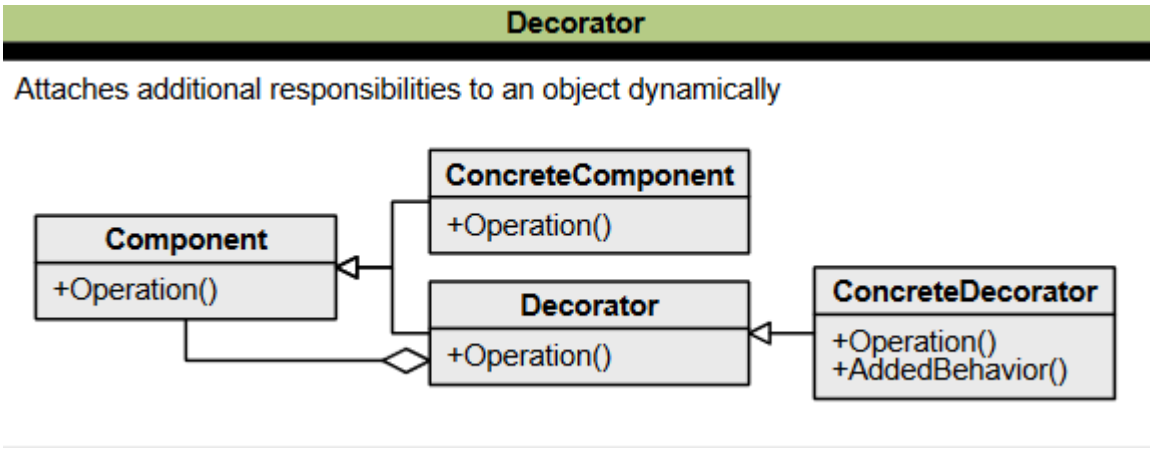


Figure 13: Class diagram of Decorator pattern

2.3. BEHAVIOURAL PATTERNS

2.3.1. TEMPLATE METHOD

The Template Method design pattern is a behavioral design pattern that defines the skeleton of an algorithm in a superclass, allowing subclasses to override specific steps of the algorithm without changing its overall structure. It is one of the most straightforward design patterns to understand and implement, and it is commonly used in framework development to avoid code duplication.

The main components of the Template Method pattern are:

1. **Abstract Class**: It contains the typical algorithm steps as abstract methods or hooks meant to be implemented by the subclasses.
2. **ConcreteClass**: Each concrete class can customize or override specific algorithm steps while keeping others intact.

Here is an sample Template Method implementation in Java:

```
abstract class AbstractClass {
    public void templateMethod() {
        step1();
        step2();
        step3();
    }
    abstract void step1();
    abstract void step2();
    void step3() {
        System.out.println("Default implementation of step3");
    }
}

class ConcreteClass extends AbstractClass {
    void step1() {
        System.out.println("Executing step1");
    }
    void step2() {
        System.out.println("Executing step2");
    }
```

```
}
public class Client {
    public static void main(String[] args) {
        AbstractClass myClass = new ConcreteClass();
        myClass.templateMethod();
    }
}
```

In this example, we have an abstract class **AbstractClass** that defines the template method **templateMethod()**. It also declares abstract methods **step1()** and **step2()** which are meant to be implemented by the concrete classes.

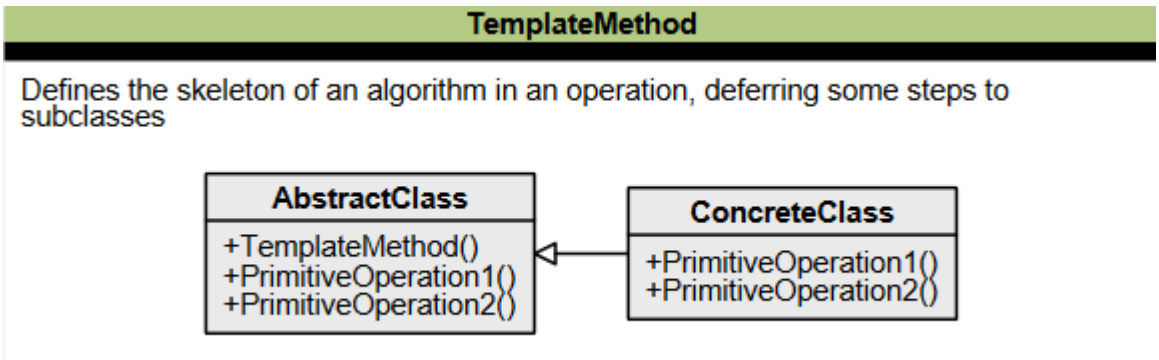
The **ConcreteClass** extends the AbstractClass and provides its own implementations for **step1()** and **step2().** It inherits the default implementation of **step3()** from the abstract class.

Advantages of Template Method pattern:

- Subclasses can reuse this structure and focus on implementing specific steps, reducing code duplication.
- It provides flexibility in implementing algorithm variations without affecting the overall structure. Changes to the algorithm can be made in one central location without affecting the subclasses.

Disadvantages of Template Method pattern:

- Modifying the algorithm dynamically during runtime or switching between different algorithm implementations can be challenging.
- Introducing abstract classes and subclasses in the pattern can increase the complexity of the class hierarchy. This may make the codebase harder to understand and maintain, especially when dealing with many subclasses.



2.3.2. MEDIATOR

The Mediator design pattern is a widely used behavioral design pattern that helps decouple objects by introducing a layer between them to handle their interaction. This pattern is proper when there are multiple objects with complex communication, and it aims to reduce the coupling between these objects by forcing them to communicate only through a mediator object.

The main components of the Mediator pattern are:

1. **Mediator**: Defines the interface for communication between colleague objects.
2. **ConcreteMediator**: Implements the mediator interface and coordinates communication between colleague objects.
3. **Colleague**: Defines the interface for communication with other colleagues.
4. **ConcreteColleague**: Implements the colleague interface and communicates with colleagues through its mediator.

Here is an sample implementation of Mediator pattern in Java.

```
import java.util.ArrayList;
import java.util.List;
// Mediator Interface
interface ChatMediator {
    void sendMessage(String message, User user);

    void addUser(User user);
}
// Concrete Mediator
class ChatMediatorImpl implements ChatMediator {
    private List<User> users;

    public ChatMediatorImpl() {
        this.users = new ArrayList<>();
    }

    @Override
    public void sendMessage(String message, User user) {
```

```
        for (User u : users) {
            if (u != user) {
                u.receiveMessage(message);
            }
        }
    }
    @Override
    public void addUser(User user) {
        users.add(user);
    }
}
// Colleague
abstract class User {
    protected ChatMediator mediator;
    protected String name;
    public User(ChatMediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }
    public abstract void sendMessage(String message);
    public abstract void receiveMessage(String message);
}
// Concrete Colleague
class ChatUser extends User {
    public ChatUser(ChatMediator mediator, String name) {
        super(mediator, name);
    }
    @Override
    public void sendMessage(String message) {
        System.out.println(name + " sends: " + message);
        mediator.sendMessage(message, this);
    }
    @Override
    public void receiveMessage(String message) {
        System.out.println(name + " receives: " + message);
    }
}
// Example usage
public class Client {
```

```
    public static void main(String[] args) {
        ChatMediator mediator = new ChatMediatorImpl();
        User user1 = new ChatUser(mediator, "User1");
        User user2 = new ChatUser(mediator, "User2");
        mediator.addUser(user1);
        mediator.addUser(user2);
        user1.sendMessage("Hello, everyone!");
        user2.sendMessage("Hey User1, how are you?");
    }
}
```

In this example, we have a **ChatMediator** interface that defines the communication methods for sending messages and adding users. The **ChatMediatorImpl** class is the concrete implementation of the mediator, which keeps track of the users and handles the communication between them.

The **User** class is an abstract class that represents the colleagues in the system. It holds a reference to the mediator and provides abstract methods for sending and receiving messages. The **ChatUser** class is a concrete implementation of **User** that overrides the message-related methods to send and receive messages through the mediator.

Advantages of Mediator pattern:

- Decouples colleagues by promoting loose coupling between them, making it easier to modify and extend their interactions independently.
- Centralizes control and communication logic, simplifying the system's complexity and enhancing maintainability.

Disadvantages of Mediator pattern:

- The mediator can become a single point of failure and performance bottleneck if it becomes overloaded with responsibilities.
- Adding new colleagues may require modifying the mediator interface and implementation, potentially leading to code changes in multiple classes.

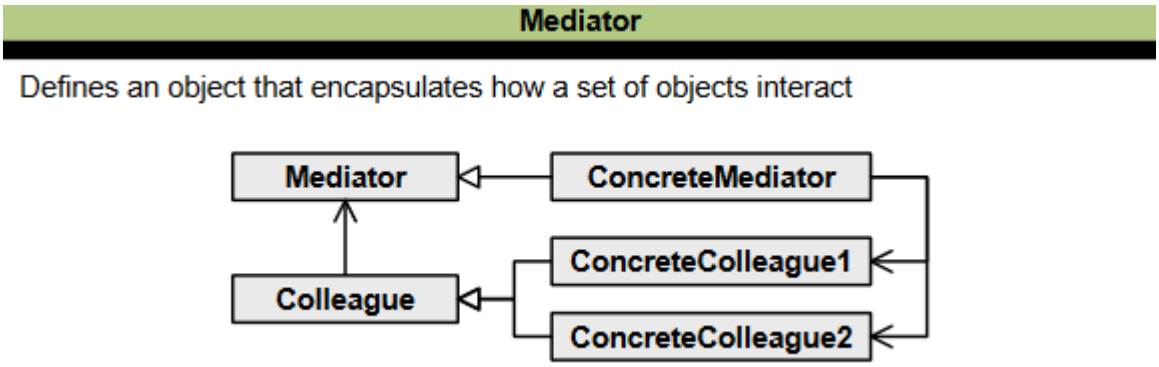


Figure 15: Class diagram of Mediator pattern

2.3.3. CHAIN OF RESPONSIBILITY

The Chain of Responsibility design pattern is a behavioral design pattern that aims to achieve loose coupling in software design by allowing a request from a client to pass through a chain of objects to process the request. The objects in the chain decide who will process the request and whether it needs to be sent to the next object in the chain or not.

The main components of the Chain of Responsibility pattern are:

1. **Handler:** An interface or abstract class that defines the method to handle the request and dispatch it to the chain of handlers. It has a reference to the first handler in the chain but does not know about the rest of the handlers.
2. **Concrete Handlers:** These are the actual handler classes that process the request. They are chained together in a specific order.
3. **Client:** The request's originator, which accesses the handler to handle the request.

Here is an example of Chain of Responsibility implementation in Java;

```
// Handler interface
public interface Handler {
    void setNext(Handler next);

    void handleRequest(Request request);
}
// Concrete handler 1
public class ConcreteHandler1 implements Handler {
    private Handler next;
    @Override
    public void setNext(Handler next) {
        this.next = next;
    }
    @Override
    public void handleRequest(Request request) {
        if (request.getType().equals("Type1")) {
            System.out.println("Request handled by ConcreteHandler1");
        } else if (next != null) {
            next.handleRequest(request);
        } else {
```

```
            System.out.println("Request cannot be handled");
        }
    }
}
// Concrete handler 2
public class ConcreteHandler2 implements Handler {
    private Handler next;
    @Override
    public void setNext(Handler next) {
        this.next = next;
    }
    @Override
    public void handleRequest(Request request) {
        if (request.getType().equals("Type2")) {
            System.out.println("Request handled by ConcreteHandler2");
        } else if (next != null) {
            next.handleRequest(request);
        } else {
            System.out.println("Request cannot be handled");
        }
    }
}
// Request class
public class Request {
    private String type;

    public Request(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }
}
// Client class
public class Client {
    public static void main(String[] args) {
        // Create handlers
        Handler handler1 = new ConcreteHandler1();
```

```
        Handler handler2 = new ConcreteHandler2();
        // Set up the chain of responsibility
        handler1.setNext(handler2);
        // Create requests
        Request request1 = new Request("Type1");
        Request request2 = new Request("Type2");
        Request request3 = new Request("Type3");
        // Handle the requests
        handler1.handleRequest(request1); // Output: Request handled by
ConcreteHandler1
        handler1.handleRequest(request2); // Output: Request handled by
ConcreteHandler2
        handler1.handleRequest(request3); // Output: Request cannot be
handled
    }
}
```

In this example, we have two concrete handlers (**ConcreteHandler1** and **ConcreteHandler2**) that implement the Handler interface. Each handler has a reference to the next handler in the chain. The **handleRequest()** method in each concrete handler checks if it can handle the request based on its criteria. If it can handle the request, it performs the necessary actions. If it cannot handle the request, it passes the request to the next handler in the chain. The last handler in the chain handles the case where no handler can handle the request.

Advantages of CoR pattern:

- Decouples the sender and receiver of a request, making the code more flexible and maintainable.
- Allows for dynamic addition and removal of responsibility by rearranging the chain members or their order, improving the flexibility of objects assigned tasks.
- Multiple objects can handle a request, and the handler can be determined at runtime.

Disadvantages of CoR pattern:

- It is not guaranteed that the relevant handler will receive a request.
- It can affect the performance of the system and make debugging more difficult.

- This may lead to an increased number of implementation classes and maintenance problems if most of the code is common in all implementations.

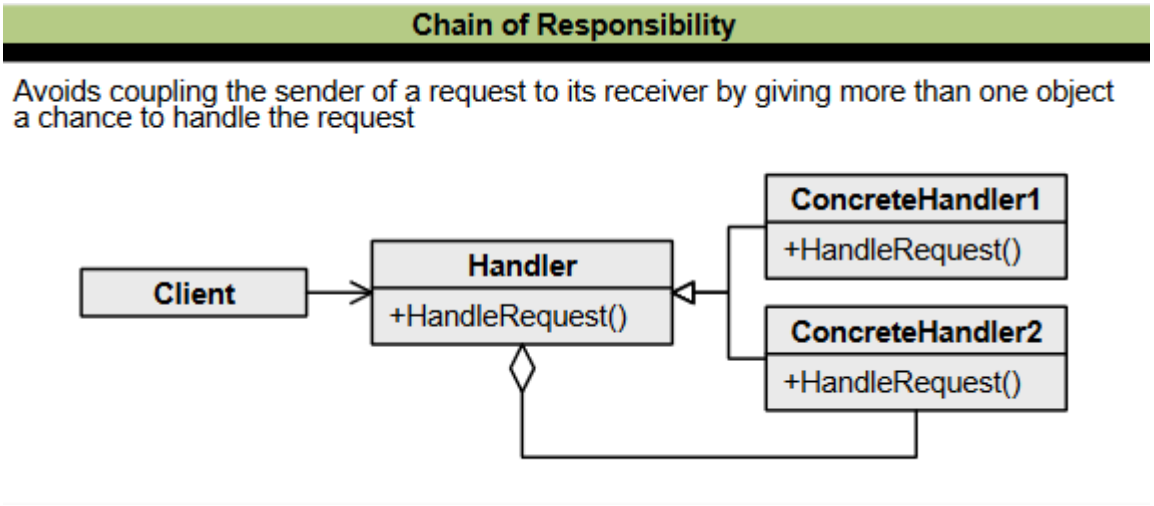


Figure 16: Class diagram of CoR pattern

2.3.4. OBSERVER

The Observer design pattern is a behavioral design pattern that defines a one-to-many dependency between objects. When the state of one object (the subject) changes, all its dependents (observers) are notified and updated automatically. This pattern is also known as the publish-subscribe pattern and is widely used in event-driven systems.

There are two primary components of the Observer pattern:

1. **Subject:** The object whose state is to be observed. It maintains a list of observers and notifies them of any state changes, usually by calling one of their methods. The subject is responsible for maintaining this list and notifying observers when its state changes.
2. **Observers:** The objects wanting to be informed about changes in the subject's state. Observers register and unregister themselves with the subject to be notified of state changes. They update their state to synchronize with the subject's state when notified.

Here is an example of Observer implementation in Java.

```
import java.util.ArrayList;
import java.util.List;
// Subject (Observable)
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
// Concrete Subject
class WeatherStation implements Subject {
    private double temperature;
    private List<Observer> observers;
    public WeatherStation() {
        this.observers = new ArrayList<>();
    }
    public void setTemperature(double temperature) {
        this.temperature = temperature;
        notifyObservers();
    }
    @Override
```

```
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
}
// Observer
interface Observer {
    void update(double temperature);
}
// Concrete Observer
class TemperatureDisplay implements Observer {
    @Override
    public void update(double temperature) {
        System.out.println("Temperature Display: " +
temperature);
    }
}
// Concrete Observer
class Fan implements Observer {
    @Override
    public void update(double temperature) {
        if (temperature > 25) {
            System.out.println("Fan: Turning on the fan");
        } else {
            System.out.println("Fan: Turning off the fan");
        }
    }
}
// Client code
public class Client {
```

```
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();
        TemperatureDisplay temperatureDisplay = new
TemperatureDisplay();
        Fan fan = new Fan();
        weatherStation.registerObserver(temperatureDisplay);
        weatherStation.registerObserver(fan);
        weatherStation.setTemperature(20.5); // Updates both
temperature display and fan
        weatherStation.setTemperature(30.2); // Updates
temperature display and turns on the fan
        weatherStation.removeObserver(fan);
        weatherStation.setTemperature(18.7); // Updates only
temperature display
    }
}
```

The **TemperatureDisplay** and **Fan** classes implement the **Observer** interface, which defines the update method. Each observer implements the update method to perform its specific logic when notified of changes in the subject's state.

When the **WeatherStation's** temperature changes, it calls the **notifyObservers()**, which iterates over all registered observers and calls their update method, passing the updated temperature as a parameter.

Advantages of Observer design pattern:

- Observers are loosely coupled with the subject, allowing for easy addition or removal of observers without modifying the subject.
- Observers can respond to changes in the subject's state, enabling event-driven and reactive systems.

Disadvantages of Observer design pattern:

- The pattern can incur performance overhead when there are many observers, as each observer needs to be notified.
- Observers may receive updates they are not interested in, requiring additional logic to filter or handle specific events.

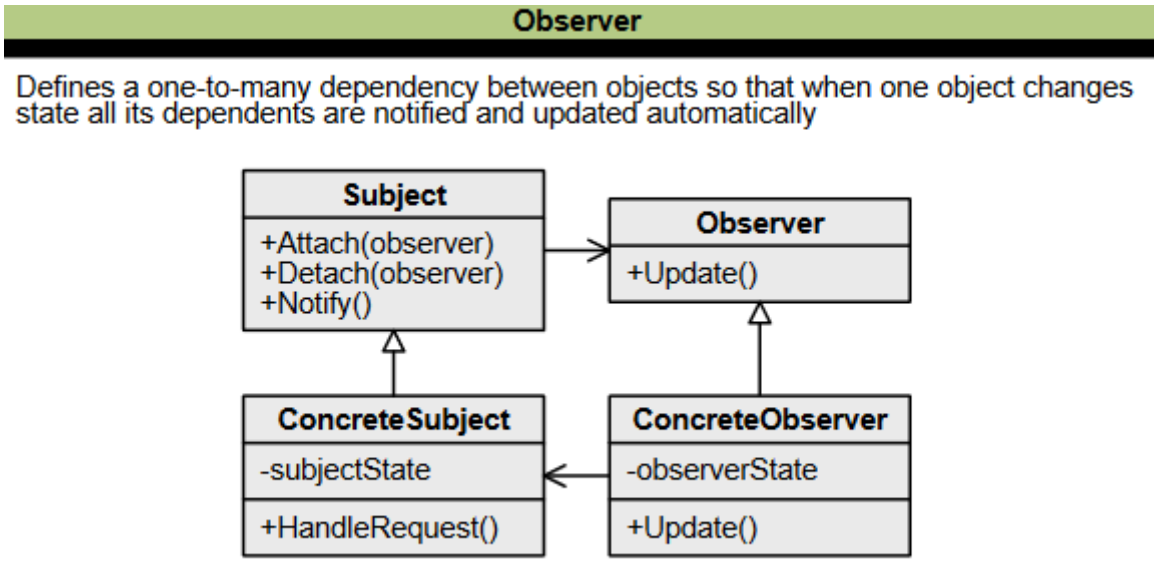


Figure 17: Class diagram of Observer pattern

2.3.5. STRATEGY

The Strategy Design Pattern is a behavioral design pattern that allows you to dynamically change the behavior of an object by encapsulating it into different strategies. This pattern enables an object to choose from multiple algorithms and behaviors at runtime rather than statically choosing a single one. In other words, it allows the behavior of an object to be selected at runtime.

The Strategy pattern consists of three main components:

1. **Context:** The class that contains the object whose behavior needs to be changed dynamically.
2. **Strategy:** The interface or abstract class that defines the standard methods for all the algorithms that the Context object can use.
3. **Concrete Strategy:** The class that implements the Strategy interface and provides the actual implementation of the algorithm.

Here is an example of how to implement the Strategy pattern in Java:

```
// Strategy interface
interface SortingStrategy {
    void sort(int[] numbers);
}
// Concrete Strategy 1
class BubbleSortStrategy implements SortingStrategy {
    public void sort(int[] numbers) {
        // Bubble sort implementation
    }
}
// Concrete Strategy 2
class QuickSortStrategy implements SortingStrategy {
    public void sort(int[] numbers) {
        // Quick sort implementation
    }
}
// Context class
class SortingContext {
    private SortingStrategy strategy;
```

```
    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }
    public void sort(int[] numbers) {
        strategy.sort(numbers);
    }
}
// Client code
public class Client {
    public static void main(String[] args) {
        SortingContext context = new SortingContext();
        context.setStrategy(new BubbleSortStrategy());
        int[] numbers = { 4, 2, 7, 1, 3 };
        context.sort(numbers);
        context.setStrategy(new QuickSortStrategy());
        context.sort(numbers);
    }
}
```

To use the Strategy pattern, you first create a **Context** object. Then, you create one or more **Concrete Strategy** objects that implement the Strategy interface. Finally, you set the Strategy object for the **Context** object by calling its **setStrategy()** method. At runtime, the **Context** object uses the selected **Concrete Strategy** object to perform its operations.

Advantages of Strategy pattern:

- Encapsulates individual algorithms or behaviors into separate classes, promoting better code organization and maintainability.
- Allows for runtime switching of strategies, enabling dynamic behavior changes without modifying the context class.

Disadvantages of Strategy pattern:

- Increases the complexity of the code by introducing multiple classes and interfaces.
- Clients must be aware of and choose the appropriate strategy, which can lead to additional code and decision-making overhead.

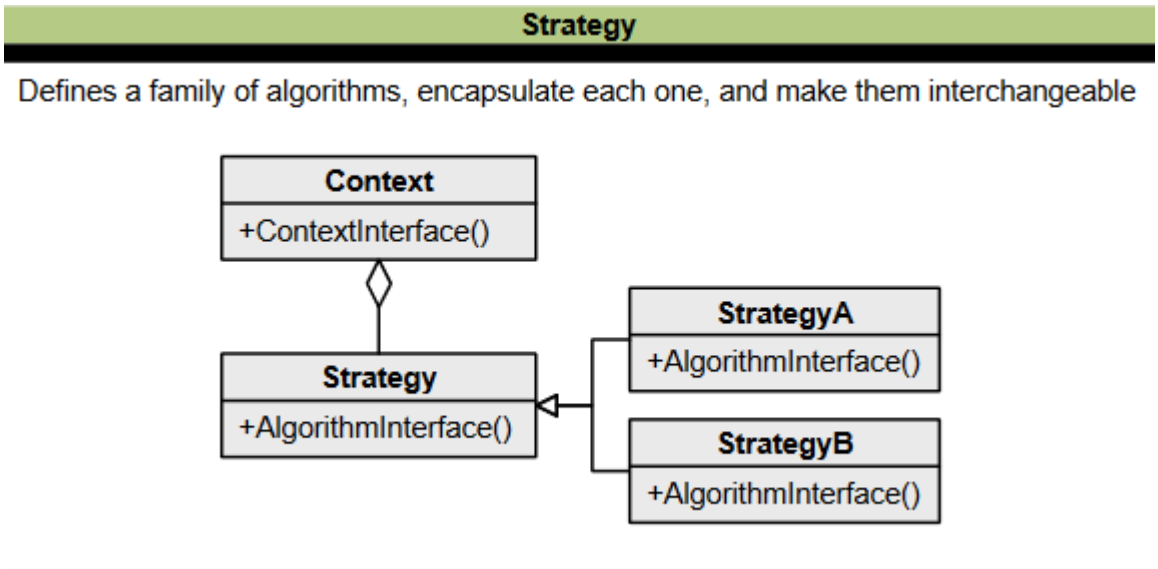


Figure 18: Class diagram of Strategy pattern

2.3.6. COMMAND

The Command design pattern is a behavioral pattern that turns a request into a stand-alone object containing all information about the request. This allows you to pass requests as method arguments, delay or queue a request's execution, and support undoable operations.

The main components of the Command pattern are:

1. **Command:** This is the core of the Command design pattern, defining the contract for implementation.
2. **Receiver:** This is the object that receives the request and performs the corresponding action.
3. **Invoker:** This object is responsible for executing the command and may perform bookkeeping about the command execution.
4. **Client:** This is responsible for creating and associating command and receiver objects, instantiating the invoker object, and executing the action method.

Here is an example of Command design pattern in Java:

```
// Receiver class
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }
    public void turnOff() {
        System.out.println("Light is off");
    }
}
// Command interface
interface Command {
    void execute();
}
// Concrete command classes
class TurnOnCommand implements Command {
    private Light light;
    public TurnOnCommand(Light light) {
        this.light = light;
```

```
    }
    public void execute() {
        light.turnOn();
    }
}
class TurnOffCommand implements Command {
    private Light light;
    public TurnOffCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.turnOff();
    }
}
// Invoker class
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
// Client code
public class Client {
    public static void main(String[] args) {
        // Create the receiver object
        Light light = new Light();
        // Create the command objects
        Command turnOnCommand = new TurnOnCommand(light);
        Command turnOffCommand = new TurnOffCommand(light);
        // Create the invoker object
        RemoteControl remote = new RemoteControl();
        // Set the commands
        remote.setCommand(turnOnCommand);
        remote.pressButton(); // Turns on the light
        remote.setCommand(turnOffCommand);
        remote.pressButton(); // Turns off the light
```

```
    }
}
```

In this example, we have a **Light** class that acts as the receiver of the command. The **Command** interface defines the **execute()** method that all concrete command classes must implement. The **RemoteControl** class acts as the invoker, which sets the command and triggers its execution through the **pressButton()** method.

In the client code, we create an instance of the **Light** object, create the command objects for turning the light on and off, create an instance of the **RemoteControl**, set the commands on the remote control, and finally, we can press the button to execute the commands.

Advantages of Command design pattern:

- Decouples the sender of a request from the receiver, allowing them to vary independently and promoting a more flexible and maintainable codebase.
- Easy to add new commands without modifying existing code, as new command classes can be created and integrated seamlessly.

Disadvantages of Command design pattern:

- The pattern introduces additional classes and indirection, which can make the code more complex and harder to understand.
- The pattern may introduce a slight performance overhead due to the need for additional objects and indirection, which is often negligible in most applications.

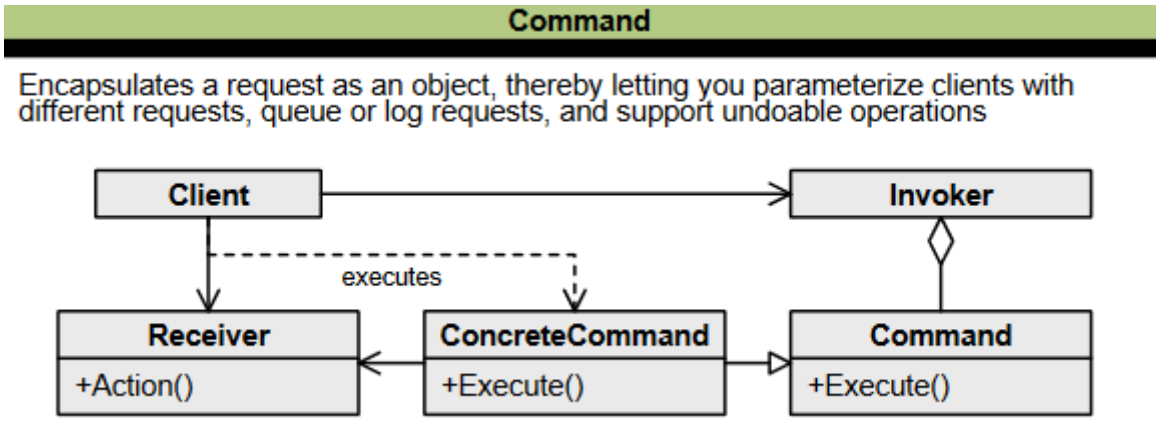


Figure 19: Class diagram of Command pattern

2.3.7. STATE

The State Design Pattern is a behavioral design pattern that allows an object to change its behavior based on its internal State. This pattern is used when an object's behavior depends on its State and needs to change its behavior at runtime depending on that State.

The main components of the State Design Pattern are:

1. **Context:** Defines an interface for clients to interact with. It references concrete state objects, which may be used to define the current State of objects.
2. **State:** Defines an interface for declaring what each concrete State should do.
3. **ConcreteState:** Provides the implementation for methods defined in State.

To implement the State Design Pattern, you can follow these steps:

1. Define a Context class that presents a single interface to the outside world and maintains a pointer to the current State.
2. Create a State interface or abstract class that defines the methods for each state-specific behavior.
3. Implement ConcreteState classes that implement the methods defined in the State interface.
4. In the Context class, delegate state-specific behavior to the current state object instead of implementing state-specific behavior directly.

Here is an example of how to implement State design pattern in Java:

```
// Context class
class Context {
    private State state;
    public Context() {
        state = new StateA(); // Initial state
    }
    public void setState(State state) {
        this.state = state;
    }
    public void request() {
```

```
        state.handle(this);
    }
}
// State interface
interface State {
    void handle(Context context);
}
// Concrete state classes
class StateA implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling State A");
        // Perform state-specific operations
        // Transition to another state if necessary
        context.setState(new StateB());
    }
}
class StateB implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling State B");
        // Perform state-specific operations
        // Transition to another state if necessary
        context.setState(new StateC());
    }
}
class StateC implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Handling State C");
        // Perform state-specific operations
        // Transition to another state if necessary
        context.setState(new StateA());
    }
}
// Client code
public class Client {
    public static void main(String[] args) {
        Context context = new Context();
```

```
        context.request(); // Output: Handling State A
        context.request(); // Output: Handling State B
        context.request(); // Output: Handling State C
        context.request(); // Output: Handling State A
    }
}
```

In this example, we have a **Context** class that maintains a reference to the current state. The State interface defines the **handle()**, which represents the state-specific behavior. There are three concrete state classes: **StateA**, **StateB**, and **StateC**. Each state class implements the **handle()** with its own specific behavior and may transition to another state if needed.

In the Client code, we create a **Context** object and perform a sequence of requests, which triggers the corresponding state handling and state transitions. The output shows the current state being handled.Note that this is a simplified example, and in a real-world scenario, the state classes would likely have more complex behavior and interactions.

Advantages of State design pattern:

- Each state class encapsulates its behavior, making adding or modifying states easier without impacting other parts of the code.
- Eliminates significant conditional statements by representing each state as a separate class, improving code readability and maintainability.

Disadvantages of State design pattern:

- As a separate class represents each state, the pattern can increase the number of classes, which may be complex to manage in large systems.
- If the transitions between states are appropriately managed, it is possible to avoid ending up in an inconsistent state or missing a required state transition, leading to unexpected behavior.

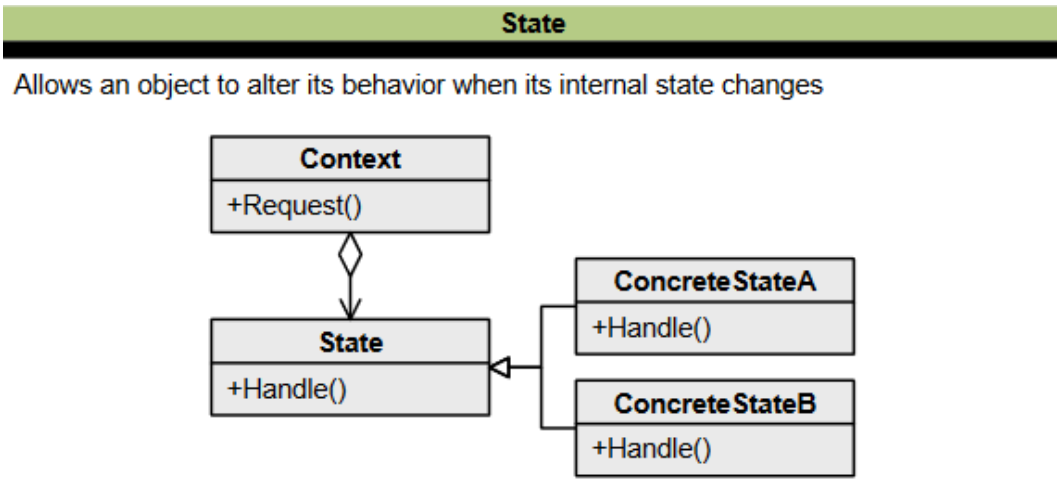


Figure 20: Class diagram of State pattern

2.3.8. VISITOR

The Visitor design pattern is a behavioral design pattern that allows you to separate the algorithm from the object structure on which it operates. It is used when you need to operate on a group of similar objects without changing the object structure itself. This pattern is beneficial for implementing the open/closed principle, which states that software entities should be open for extension but closed for modification.

The main components of the Visitor pattern are:

1. **Visitor:** An interface or abstract class that declares the visit operations for all visitable classes.
2. **ConcreteVisitor:** A class implementing the visit methods declared in the Visitor interface. Each ConcreteVisitor is responsible for different operations.
3. **Element:** An interface that declares the accept operation, which enables an object to be "visited" by the visitor object.
4. **ConcreteElement:** A class that implements the Element interface and defines the accept operation. The visitor object is passed to this object using the accept operation.
5. **Client:** A class with access to the object structure can instruct the objects to accept a Visitor to perform the appropriate operations.

Here is an example of Visitor design pattern in Java to create an AST node:

```
// Abstract Expression class
interface Expression {
    void accept(ExpressionVisitor visitor);
}
// Binary Expression class
class BinaryExpression implements Expression {
    int left;
    int right;
    public String operator;
    public BinaryExpression(int left, int right, String operator) {
        this.left = left;
        this.right = right;
        this.operator = operator;
```

```
    }
    @Override
    public void accept(ExpressionVisitor visitor) {
        visitor.visit(this);
    }
}
// Unary Expression class
class UnaryExpression implements Expression {
    int operand;
    public String operator;
    public UnaryExpression(int operand, String operator) {
        this.operand = operand;
        this.operator = operator;
    }
    @Override
    public void accept(ExpressionVisitor visitor) {
        visitor.visit(this);
    }
}
// Visitor interface
interface ExpressionVisitor {
    void visit(BinaryExpression expression);
    void visit(UnaryExpression expression);
}
// Concrete Visitor implementation
class AST implements ExpressionVisitor {
    @Override
    public void visit(BinaryExpression expression) {
        System.out.print("BinExpr(" + expression.left + " " +
expression.operator + " " + expression.right + ")");
    }
    @Override
    public void visit(UnaryExpression expression) {
        System.out.print("UnExpr(" + expression.operator +
expression.operand + ")");
    }
}
// Usage example
public class Client {
```

```
    public static void main(String[] args) {
        Expression binExp = new BinaryExpression(3, 5, "+");
        Expression unExp = new UnaryExpression(2, "-");
        // Create the visitor
        ExpressionVisitor visitor = new AST();
        // Traverse the AST
        binExp.accept(visitor);
        unExp.accept(visitor);
    }
}
```

In this example, we have the **Expression** interface representing the abstract expression, with two concrete implementations: **BinaryExpression** and **UnaryExpression**. The **Expression** interface defines the accept method that accepts a visitor.

The **ExpressionVisitor** interface declares two methods: **visit(BinaryExpression expression)** and **visit(UnaryExpression expression)**, which will be implemented by concrete visitor classes. The **AST** class is a concrete visitor implementation that prints the **AST** by visiting each expression node in the tree.

Advantages of Visitor pattern:

- Allows adding new operations to a set of classes without modifying those classes.
- Separates the data structure from the operations performed on it, improving maintainability and extensibility.

Disadvantages of Visitor pattern:

- Requires modifying the element classes to implement the visitor interface, which can be cumbersome in large and complex object hierarchies.
- Adding new element types to the hierarchy requires modifying the element classes, the visitor interface, and its implementations.

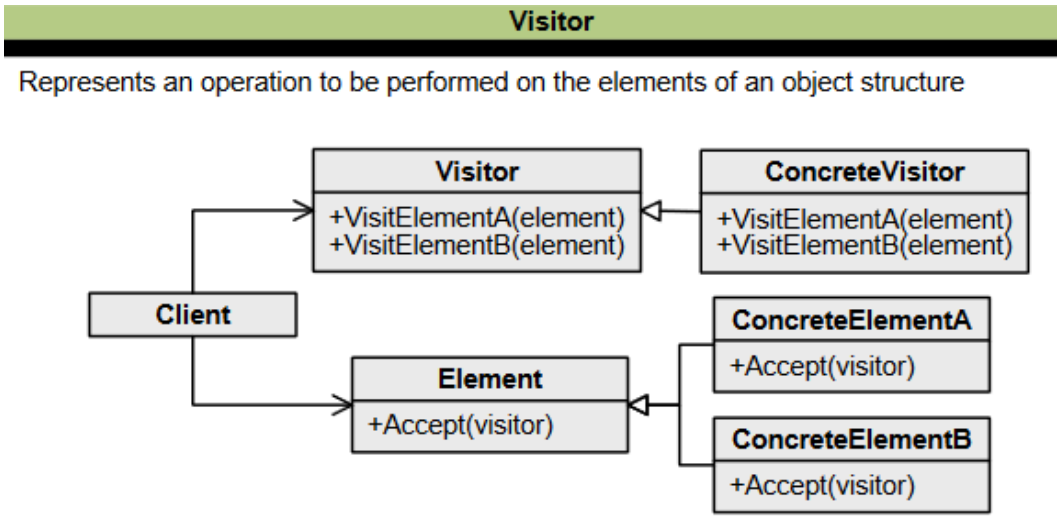


Figure 21: Class diagram of Visitor pattern

2.3.9. ITERATOR

The Iterator design pattern is a behavioral pattern that provides a way to access the elements of a collection or aggregate object sequentially without exposing its underlying representation. It decouples the traversal algorithms from the collection, which means that one traversal algorithm can be used with different collections without modifying the algorithm or the collection.

The main components of the Iterator pattern are:

1. **Iterator**: Defines the methods for accessing and traversing elements.
2. **Collection**: Defines the contract for a collection and provides a method for creating an Iterator object.
3. **Concrete Iterator:** Implements the Iterator interface and includes the actual iteration logic.

Here is an example of the Iterator pattern in Java:

```
import java.util.Iterator;
// Define a custom collection class
class MyCollection<T> implements Iterable<T> {
    private T[] items;
    private int size;
    public MyCollection(T[] items) {
        this.items = items;
        this.size = items.length;
    }
    @Override
    public Iterator<T> iterator() {
        return new MyIterator();
    }
    // Define a custom iterator class
    private class MyIterator implements Iterator<T> {
        private int index;
        public MyIterator() {
            this.index = 0;
        }
        @Override
        public boolean hasNext() {
```

```
            return index < size;
        }
        @Override
        public T next() {
            if (hasNext()) {
                return items[index++];
            } else {
                throw new java.util.NoSuchElementException();
            }
        }
    }
}
public class Client {
    public static void main(String[] args) {
        String[] names = { "Alice", "Bob", "Charlie", "David" };
        MyCollection<String> collection = new MyCollection<>(names);
        // Iterate over the collection using the iterator
        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
    }
}
```

In this example, we have a custom collection class called **MyCollection** that implements the **Iterable** interface. It contains an inner class called **MyIterator**, which implements the **Iterator** interface.

The **MyIterator** class keeps track of the current index while iterating over the collection. It provides the **hasNext()** method to check if there are more elements and the **next()** method to retrieve the next element. If there are no more elements and **next()** is called, it throws a **NoSuchElementException**.

Advantages of Iterator Pattern:

- Encapsulates the iteration logic, providing a standardized way to access collection elements, regardless of its internal structure.
- Allows for the traversal of a collection without exposing its implementation details, promoting encapsulation and information hiding.

Disadvantages of Iterator Pattern:

- It adds a layer of abstraction, which can increase complexity and reduce code readability in simple scenarios.
- Modifying the underlying collection while an iteration is in progress may lead to undefined behavior or exceptions, requiring additional precautions to handle such scenarios.

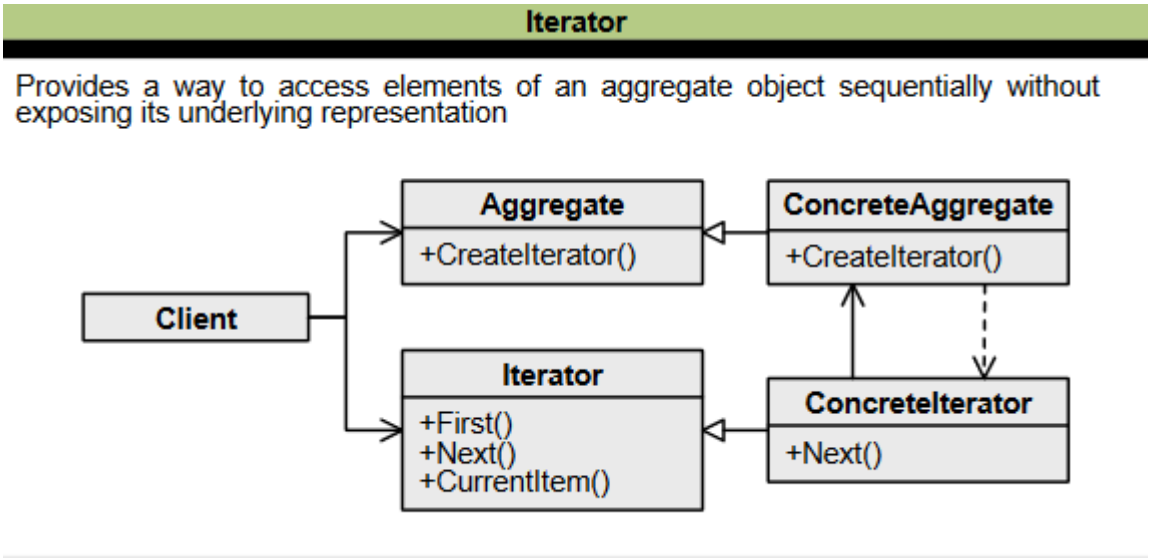


Figure 22: Class diagram of Iterator pattern

2.3.10. INTERPRETER

The Interpreter design pattern is a behavioral design pattern that defines a grammatical representation of a language and provides an interpreter to deal with this grammar. It is mainly used in compilers and other language processing programs, such as SQL parsing and symbol processing engines.

The main components of the Interpreter pattern are:

1. **Abstract Expression:** It declares the standard interface for all expressions in the language. Both terminal and non-terminal expressions implement this component.
2. **Terminal Expression:** It implements the interpret method and provides the specific implementation for evaluating a particular type of terminal expression. Terminal expressions do not have any sub-expressions.
3. **Non-terminal Expression:** It implements the interpret method but provides the logic to combine or evaluate the sub-expressions to produce the final result.
4. **Context:** This is the context or environment in which the expressions are evaluated.
5. **Client:** The client code creates and manipulates the expressions using the abstract syntax tree (AST). It builds the AST by combining different expressions and then calls the interpret method on the root expression to perform the interpretation.

Here is an example of the Interpreter pattern in Java:

```
// Abstract expression
interface Expression {
    int interpret();
}
// Terminal expression
class NumberExpression implements Expression {
    private int number;
    public NumberExpression(int number) {
        this.number = number;
    }
    @Override
    public int interpret() {
        return number;
```

```
    }
}
// Non-terminal expression
class AdditionExpression implements Expression {
    private Expression leftExpression;
    private Expression rightExpression;
    public AdditionExpression(Expression leftExpression, Expression
rightExpression) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }
    @Override
    public int interpret() {
        return leftExpression.interpret() + rightExpression.interpret();
    }
}
// Client code
public class Client {
    public static void main(String[] args) {
        Expression expression = new AdditionExpression(
                new NumberExpression(5),
                new AdditionExpression(
                        new NumberExpression(10),
                        new NumberExpression(2)));
        int result = expression.interpret();
        System.out.println("Result: " + result);
    }
}
```

Advantages of Interpreter pattern:

- It allows for easy modification or extension of the grammar or language rules without affecting the overall structure.
- It can simplify grammar by dividing complex rules into smaller, more manageable expressions.

Disadvantages of Interpreter pattern:

- It may incur performance overhead due to the recursive nature of evaluating expressions.

- As the grammar and number of expressions grow, the pattern can become complex and harder to maintain.

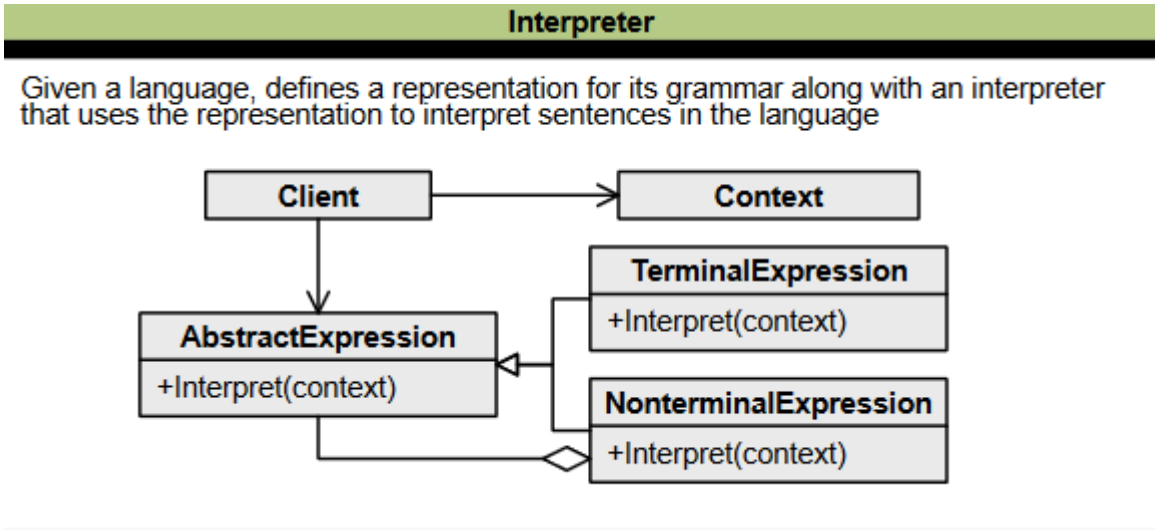


Figure 23: Class diagram of Interpreter pattern

2.3.11. MEMENTO

The Memento design pattern is a behavioral design pattern that allows an object to save and restore its state without violating encapsulation. This pattern is proper when you want to implement undo or rollback functionality in your application or to save and restore the state of an object at a later time.

The Memento pattern involves three main components:

1. **Originator:** This object has an internal state that needs to be saved and restored. It exposes a method to create a memento object that captures the current internal state of the object.
2. **Memento:** This is an object that captures the internal state of the originator object. It is typically implemented as a simple data structure storing relevant state information.
3. **Caretaker:** This is the object responsible for saving and restoring the state of the originator object. It maintains a list of memento objects and can revert the originator object to a previous state by restoring a memento from the list.

Here is an example of the Memento pattern in Java:

```
// Originator class
import java.util.Stack;
class TextEditor {
    private String text;
    public void setText(String text) {
        this.text = text;
    }
    public String getText() {
        return text;
    }
    public TextEditorMemento save() {
        return new TextEditorMemento(text);
    }
    public void restore(TextEditorMemento memento) {
        this.text = memento.getText();
    }
}
```

```java
// Memento class
class TextEditorMemento {
    private String text;
    public TextEditorMemento(String text) {
        this.text = text;
    }
    public String getText() {
        return text;
    }
}
// Caretaker class
class TextEditorHistory {
    private Stack<TextEditorMemento> mementoStack = new Stack<>();
    public void push(TextEditorMemento memento) {
        mementoStack.push(memento);
    }
    public TextEditorMemento pop() {
        return mementoStack.pop();
    }
}
// Example usage
public class Client {
    public static void main(String[] args) {
        TextEditor textEditor = new TextEditor();
        TextEditorHistory history = new TextEditorHistory();
        // Set initial text
        textEditor.setText("Hello, World!");
        // Save initial state
        history.push(textEditor.save());
        // Update text
        textEditor.setText("Hello, Me!");
        // Save new state
        history.push(textEditor.save());
        // Restore previous state
        textEditor.restore(history.pop());
        System.out.println(textEditor.getText()); // Output: Hello, Me!
    }
}
```

Advantages of Memento pattern:

- It allows an object to save and restore its internal state without exposing its implementation details, promoting encapsulation.
- It provides a simple way to implement undo/redo functionality by storing multiple states and allowing easy restoration.

Disadvantages of Memento pattern:

- If the originator object has a large state, storing and managing multiple mementos can consume significant memory and impact performance.
- The pattern can introduce increased complexity and additional overhead, especially when dealing with complex object hierarchies and deep state dependencies.

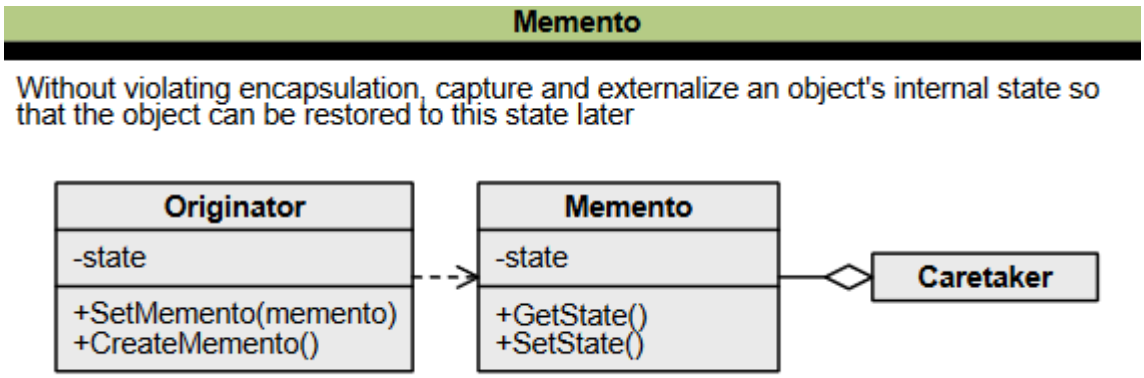


Figure 24: Class diagram of Memento pattern

III. NEW PROGRAMMING APPROACHES

3.1. DATA WRANGLING

Data wrangling is the process of transforming and cleaning raw data into a more usable and understandable format for data analysis. It involves processes designed to explore, transform, and validate raw datasets from their messy and complex forms into high-quality data.

The data wrangling process typically involves the following steps:

1. **Data Discovery**: This step involves understanding the data by exploring and familiarizing yourself with the source.
2. **Data Structuring**: In this step, raw data is restructured to suit the analytical model your enterprise plans to deploy. This step is essential for more straightforward computation and analysis in the later steps.
3. **Data Cleaning**: Raw data has some errors that must be fixed before data is passed on to the next stage. Data cleaning involves tackling outliers and correcting or deleting insufficient data completely. This step is essential in assuring the overall quality of the data.
4. **Data Enriching**: You have become familiar with the data by this stage. Now is the time to ask yourself whether you need to embellish the raw data. Do you want to augment it with other data?
5. **Data Validating**: This activity surfaces data quality issues, which must be addressed with the necessary transformations. The validation rules require repetitive programming steps to check your data's authenticity and quality.
6. **Data Publishing**: Once all the above steps are completed, the final output of your data-wrangling efforts is pushed downstream for your analytics needs

It is important to note that the data wrangling process is iterative, and some of the steps may not be necessary, while others may need repeating. Additionally, the steps may not occur in the same order. However, following these steps can help ensure your data is clean, reliable, and ready for analysis. Here are some key points to keep in mind:

- Data wrangling is essential because it prepares your data for the data mining process, the stage of analysis when you look for patterns or relationships in your dataset that can guide actionable insights. Your data analysis can only be as good as the data itself. If you analyze insufficient data, it is likely that you will draw ill-informed conclusions and will not be able to make reliable, data-informed decisions.
- Data wrangling can be time-consuming and taxing on resources, mainly when done manually. This is why many organizations institute policies and best practices that help employees streamline the data cleanup process—for example, requiring that data include certain information or be in a specific format before it is uploaded to a database.
- The data wrangling process typically involves six iterative steps of discovering, structuring, cleaning, enriching, validating, and publishing data before it is ready for analytics.
- The final step of the data wrangling process is publishing, which involves preparing the data for future use, including providing notes and documentation of your wrangling process and creating access for other users and applications.
- Data wrangling is worth the effort because insights gained during the data wrangling process can be invaluable and likely affect the future course of a project. Skipping or rushing this step will result in poor data models that impact an organization's decision-making and reputation.
- Data wrangling can be used in various applications, such as fraud detection and customer behavior analysis, to provide precise insights and empower data scientists to discover data trends.
- Python is a popular language for data wrangling, and many libraries and tools are available for this purpose.
- The data wrangling process typically involves gathering, assessing, and cleaning data, then storing, analyzing, and visualizing the wrangled data.

Here is sample Python code that demonstrates different aspects of data wrangling, including creating data, data replacing, data filtering, and creating a DataFrame:

```
import pandas as pd
# Creating a dictionary of data
data = {'Name': ['John', 'Jane', 'Tom', 'Alice', 'Bob'],
        'Age': [25, 22, 30, 27, 24],
        'Gender': ['M', 'F', 'M', 'F', 'M'],
        'City': ['New York', 'Los Angeles', 'Chicago', 'Houston',
'Phoenix'],
        'Salary': [50000, 60000, 70000, 55000, 65000]}

# Creating a DataFrame from the dictionary
df = pd.DataFrame(data)

# Replacing values in a column
df['City'] = df['City'].replace({'Los Angeles': 'LA', 'Chicago': 'CHI'})

# Filtering data
df_filtered = df[(df['Age'] > 24) & (df['Salary'] > 55000)]

# Creating a new DataFrame from filtered data
df_filtered = pd.DataFrame(df_filtered, columns=['Name', 'Age',
'Salary'])
# Displaying the new DataFrame
print(df_filtered)
```

3.2. SMART CONTRACT

Smart contracts are self-executing digital contracts stored on a blockchain and automatically enforce the terms of an agreement between parties. They are designed to operate without the need for intermediaries or trusted third parties. Smart contracts can automate various processes, from financial transactions to supply chain management.

However, developers should be aware of some limitations to smart contracts. These include:

- Smart contracts cannot get information about real-world events because they cannot send HTTP requests. This is by design, as relying on external information could jeopardize consensus, which is essential for security and decentralization.
- Smart contracts have a maximum size of 24KB, or they will run out of gas. This can be circumnavigated by using The Diamond Pattern.

To negotiate, draft, and adjudicate smart contracts, non-technical parties may need to enter into a written agreement with the brilliant contract programmer, similar to the contract that parties may enter into with a provider of services for Electronic Data Interchange (EDI) transactions today. Additionally, insurance companies could create policies to protect contracting parties from the risk that smart contract code does not perform the functions specified in the text of an agreement.

Smart contract technology must be easy to understand and use, allowing developers to focus on business logic and not on writing in security and privacy. The codebase should be minimal to write and maintain, and infrastructure requirements should be automatically handled as part of the smart contract system. Smart contracts should be designed with the key benefits of smart contracts in mind and be fit for purpose, including native accommodation of multi-party transactions, data privacy, simultaneous communication, and strong security.

Smart contracts are encoded transactions that specify each party's rights and obligations, providing for the simultaneous execution of obligations by all parties. They can improve settlement options by creating a path for low-risk, complex, multi-party transactions. Smart contracts can operate on the blockchain or a traditional database, adding value to any complex transaction where efficiency and data privacy are paramount.

When developing smart contracts, it is essential to consider security. A study by Trail of Bits found that smart contract vulnerabilities are more like vulnerabilities in other systems than the literature would suggest. Automated static or dynamic analysis tools could detect many of the most critical flaws. However, almost 50% of findings are likely never found by automated tools, even if the state-of-the-art advances significantly. Manually produced unit tests, even extensive ones, likely offer either weak or no protection against the flaws an expert auditor can find.

Here is a simple example of a Smart Contract written in Solidity, the programming language used to write smart contracts on the Ethereum blockchain:

```
pragma solidity ^0.7.0;

contract HelloWorld {
   string public message;

   constructor(string memory initMessage) {
      message = initMessage;
   }

   function update(string memory newMessage) public {
      message = newMessage;
   }
}
```

This smart contract is called "HelloWorld" and has two functions:

- **constructor:** This function is executed once when the contract is first created and sets the initial value of the *message* variable.
- **update:** This function updates the value of the *message* variable.

The message variable is declared as a public string, which means it can be accessed by anyone on the blockchain. The ***memory*** keyword is used to indicate that the variable should be stored in memory. To deploy this smart contract on the Ethereum blockchain, you would need to compile it using a Solidity compiler and then deploy it using a blockchain wallet such as MetaMask.

IV. REFERENCES

INTRODUCTION

1. DigitalOcean. (2022, August 3). *Gangs of four (GOF) design patterns*. https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns

SINGLETON

1. FreeCodeCamp.org. (2022, July 20). *Singleton design pattern*. https://www.freecodecamp.org/news/singleton-design-pattern-with-javascript/
2. GeeksforGeeks. (2020, November 29). *Singleton design pattern*. https://www.geeksforgeeks.org/singleton-design-pattern/
3. Javatpoint. (n.d.). *Singleton design patterns*. https://www.javatpoint.com/singleton-design-pattern-in-java

FACTORY

1. Better Programming. (2021, January 4). *Understanding the factory method design pattern*. https://betterprogramming.pub/understanding-the-factory-method-design-pattern-f5ec631c99d8
2. DigitalOcean. (2022, August 3). *Factory design pattern in Java*. https://www.digitalocean.com/community/tutorials/factory-design-pattern-in-java
3. IONOS Digital Guide. (n.d.). *Factory pattern: The key information on the factory method pattern*. https://www.ionos.com/digitalguide/websites/web-development/what-is-a-factory-method-pattern/

ABSTRACT FACTORY

1. Better Programming. (2021, January 16). *Understanding the abstract factory design patterns*. Medium. https://betterprogramming.pub/understanding-the-abstract-method-design-patterns-bc416aaaf076
2. GeeksforGeeks. (2022, December 26). *Abstract factory pattern*. GeeksforGeeks. https://www.geeksforgeeks.org/abstract-factory-pattern/

BUILDER

1. GeeksforGeeks. (2022a, December 5). *Builder design pattern*. https://www.geeksforgeeks.org/builder-design-pattern/
2. HowToDoInJava. (2022b, December 4). *Builder design pattern*. https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/

PROTOTYPE

1. GeeksforGeeks. (2022, December 25). *Prototype design pattern*. https://www.geeksforgeeks.org/prototype-design-pattern/
2. Level Up Coding. (2021, May 18). *A look at the prototype design pattern*. https://levelup.gitconnected.com/a-look-at-the-prototype-design-pattern-3e4032b072d2

ADAPTER

1. Dzone. (2018, May 27). Design patterns explained: *Adapter Pattern with code examples*. https://dzone.com/articles/design-patterns-explained-adapter-pattern-with-cod
2. GeeksforGeeks. (2023, January 27). *Adapter Pattern*. https://www.geeksforgeeks.org/adapter-pattern/

COMPOSITE

1. GeeksforGeeks. (2021, September 1). *Composite design pattern*. https://www.geeksforgeeks.org/composite-design-pattern/
2. IONOS Digital Guide. (n.d.*). Composite pattern: Sample solutions for part-whole hierarchies*. https://www.ionos.com/digitalguide/websites/web-development/composite-pattern/

PROXY

1. GeeksforGeeks. (2022, December 5). *Proxy design pattern.* https://www.geeksforgeeks.org/proxy-design-pattern/
2. Scaler. (n.d.). *Proxy Design Pattern*. https://www.scaler.com/topics/design-patterns/proxy-design-pattern/

FLYWEIGHT

1. GeeksforGeeks. (2021, September 1). *Flyweight design pattern*. https://www.geeksforgeeks.org/flyweight-design-pattern/
2. Java Development Journal. (2022, October 4). *Flyweight design pattern.* https://www.javadevjournal.com/java-design-patterns/flyweight-design-pattern/

FAÇADE

1. GeeksforGeeks. (2023, February 6). *Facade design pattern*: Introduction. https://www.geeksforgeeks.org/facade-design-pattern-introduction/

BRIDGE

1. GeeksforGeeks. (2023a, January 11). *Bridge design pattern*. https://www.geeksforgeeks.org/bridge-design-pattern/

DECORATOR

2. IONOS Digital Guide. (n.d.). *Decorator pattern: A pattern for dynamic class expansion.* https://www.ionos.com/digitalguide/websites/web-development/what-is-the-decorator-pattern/
3. Makeuseof. (2022, December 22). *What is the decorator design pattern?* https://www.makeuseof.com/decorator-design-pattern/

TEMPLATE METHOD

1. GeeksforGeeks. (2022, December 5). *Template method design pattern.* https://www.geeksforgeeks.org/template-method-design-pattern/
2. O'Reilly Online Learning. (n.d.). *The advantages and disadvantages of the Template Method pattern*. https://www.oreilly.com/library/view/learning-python-design/9781785888038/ch08s05.html

MEDIATOR

1. GeeksforGeeks. (2022, September 30). *Mediator design pattern*. https://www.geeksforgeeks.org/mediator-design-pattern/
2. Java Developer Central. (2020, March 23). *Mediator design pattern*. https://javadevcentral.com/mediator-design-pattern

CHAIN OF RESPONSIBILITY

1. DZone. (2022a, February 15). *Chain of responsibility in microservices*. https://dzone.com/articles/chain-of-responsibility-in-microservices
2. GeeksforGeeks. (2022b, December 5). *Chain of responsibility design pattern*. https://www.geeksforgeeks.org/chain-responsibility-design-pattern/

OBSERVER

1. DEV Community. (2021, April 27). *Understanding design patterns: Observer.* https://dev.to/carlillo/understanding-design-patterns-observer-2ajp
2. IONOS Digital Guide. (n.d.). *Observer pattern*. https://www.ionos.com/digitalguide/websites/web-development/what-is-the-observer-pattern/

STRATEGY

1. GeeksforGeeks. (2023, April 22). *Strategy pattern: Set 1 (introduction).* https://www.geeksforgeeks.org/strategy-pattern-set-1/
2. Medium. (2022, September 11). *Software engineering practices: Strategy pattern*. https://medium.com/@wenjh1998/software-engineering-practices-strategy-pattern-fb8697cc087c

COMMAND

1. DigitalOcean. (2022, August 3). *Command design pattern.* https://www.digitalocean.com/community/tutorials/command-design-pattern
2. O'Reilly Online Learning. (n.d.). *Advantages and disadvantages of Command pattern*. https://www.oreilly.com/library/view/learning-python-design/9781785888038/ch07s04.html

STATE

1. DevIQ. (n.d.). *State design pattern*. https://deviq.com/design-patterns/state-design-pattern
2. GeeksforGeeks. (2023, January 16). *State design pattern.* https://www.geeksforgeeks.org/state-design-pattern/

VISITOR

1. GeeksforGeeks. (2019, March 12). *Visitor design pattern*. https://www.geeksforgeeks.org/visitor-design-pattern/
2. IONOS Digital Guide. (n.d.). *Visitor pattern: Explanations and examples*. https://www.ionos.com/digitalguide/websites/web-development/visitor-pattern/

ITERATOR

1. Darren Finch. (2021, January 8). *Iterator design pattern - behavioral design patterns*. https://darrenfinch.com/iterator-design-pattern-behavioral-design-patterns/
2. GeeksforGeeks. (2022, April 8). Iterator pattern. https://www.geeksforgeeks.org/iterator-pattern/

INTERPRETER

1. GeeksforGeeks. (2018, February 22). *Interpreter design pattern*. https://www.geeksforgeeks.org/interpreter-design-pattern/
2. opencodez. (2019, December 10). *Interpreter design pattern*. https://www.opencodez.com/java/interpreter-design-pattern.htm

MEMENTO

1. Medium. (2022, December 24). *What is the memento design pattern?* https://justgokus.medium.com/what-is-the-memento-design-pattern-41a29ba1f93d

DATA WRANGLING

1. Business Insights Blog. (2021, January 19). *Data wrangling: What it is & why it's important.* https://online.hbs.edu/blog/post/data-wrangling
2. Coursera. (n.d.-a). *What is data wrangling? definition, steps, and why it matters*. https://www.coursera.org/articles/data-wrangling
3. GeeksforGeeks. (2023, April 26). *Data wrangling in python*. https://www.geeksforgeeks.org/data-wrangling-in-python/
4. Javapoint. (n.d.-b). *Data wrangling.* https://www.javatpoint.com/data-wrangling
5. Simplilearn. (2023, June 6). *What is data wrangling? benefits, tools, examples and skills.* https://www.simplilearn.com/data-wrangling-article
6. University of Phoenix. (n.d.-c). *Guide to data wrangling*. https://www.phoenix.edu/blog/data-wrangling.html

SMART CONTRACT

1. Corporate Finance Institute. (2023, May 12). *Smart contracts*. https://corporatefinanceinstitute.com/resources/valuation/smart-contracts/
2. Ethereum. (n.d.-a). *Introduction to smart contracts*. https://ethereum.org/en/developers/docs/smart-contracts/
3. Forbes Magazine. (2021, January 8). *How smart contracts bring real-world improvements to post-trade settlement.* https://www.forbes.com/sites/forbesfinancecouncil/2021/01/07/how-smart-contracts-bring-real-world-improvements-to-post-trade-settlement/?sh=298e84032289
4. IBM. (n.d.-b). *What are smart contracts on Blockchain?* https://www.ibm.com/topics/smart-contracts
5. The Harvard Law School Forum on Corporate Governance. (2018, May 26). *An introduction to smart contracts and their potential and inherent limitations.* https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations/