DESIGN PATTERNS

Contents

Introduction

There are several different categories of design patterns that have been identified and documented in the field of software engineering. The most widely recognized categorization of design patterns is based on the "Gang of Four" (GoF) design patterns, which were first described in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

According to the GoF classification, there are three main types of design patterns: creational, structural, and behavioral patterns.

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. These patterns focus on the creation of objects and their initialization, and they include the following patterns:

- Singleton pattern
- Factory pattern
- Abstract factory pattern
- Builder pattern
- Prototype pattern

Structural patterns deal with object composition, creating relationships between objects to form larger structures. These patterns focus on the relationship between objects and how they can be composed to achieve new functionality, and they include the following patterns:

- Adapter pattern
- Bridge pattern
- Composite pattern
- Decorator pattern
- Facade pattern
- Flyweight pattern
- Proxy pattern

Behavioral patterns focus on communication between objects, what goes on between objects and how they operate together. These patterns focus on the behavior of objects and how they interact and operate together, and they include the following patterns:

- Chain of responsibility pattern
- Command pattern
- Interpreter pattern
- Iterator pattern
- Mediator pattern
- Memento pattern
- Observer pattern
- State pattern
- Strategy pattern
- Template method pattern
- Visitor pattern

Overall, there are a total of 23 design patterns in the GoF classification, divided into three main categories: creational, structural, and behavioral patterns. These patterns provide reusable solutions to common design problems and are a useful way to organize and structure code in a way that is easy to understand, maintain, and extend.

Singleton pattern

The Singleton pattern is a design pattern that ensures that a class has only one instance and provides a global access point to it. It is a useful pattern to use when it is necessary to ensure that only one instance of a class exists and when it is important to provide a global access point to that instance.

The Singleton pattern is implemented by creating a private constructor for the class, which prevents other objects from creating instances of the class. The class also includes a static method that returns the single instance of the class, creating it if necessary. This method is typically called a "factory method" because it serves as a factory for creating the single instance of the class.

Here is an example of how the Singleton pattern might be implemented in Java:

```
public class Singleton {
  // The single instance of the Singleton class
  private static Singleton instance;

  // Private constructor to prevent external instantiation
  private Singleton() {}

  // Factory method to return the single instance of the Singleton
class
  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

In this example, the Singleton class has a private constructor, which prevents other objects from creating instances of the class. The getInstance() method is a static method that returns the single instance of the class, creating it if necessary. This method serves as a global access point to the single instance of the class.

There are a few variations on the Singleton pattern, including the "double-checked locking" pattern, which is a more efficient way to implement the pattern in a multithreaded environment. However, the basic idea behind the Singleton pattern remains the same: to ensure that a class has only one instance and to provide a global access point to that instance.

The Singleton pattern is a useful pattern to use when it is necessary to ensure that only one instance of a class exists and when it is important to provide a global access point to that instance. It can be implemented in a simple and straightforward way, and it can help to improve the maintainability and extensibility of code by ensuring that there is only one instance of a particular class.

Factory pattern

The Factory pattern is a design pattern that creates objects without specifying the exact class to create. It is a useful pattern to use when it is necessary to create objects of a particular type, but the exact type is not known until runtime.

The Factory pattern is implemented by creating a factory class that has a method for creating objects. This method takes one or more arguments that specify the type of object to create, and it returns a new instance of the object. The factory class is responsible for creating the objects, and the client code simply calls the factory method to create the objects as needed.

Here is an example of how the Factory pattern might be implemented in Java:

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

class Circle implements Shape {
  @Override
  public void draw() {
    // Code to draw a circle
  }
}

class Rectangle implements Shape {
  @Override
  public void draw() {
    // Code to draw a rectangle
  }
}

public class Factory {
  // Factory method to create a shape
  public Shape getShape(String shapeType) {
    if (shapeType == null) {
      return null;
    }
    if (shapeType.equalsIgnoreCase("CIRCLE")) {
      return new Circle();
    } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
      return new Rectangle();
    }
    return null;
  }

  public static void main(String[] args)
```

```
    {
        Factory shapeFactory = new Factory();

        // Create a circle
        Shape circle = shapeFactory.getShape("CIRCLE");
        circle.draw();

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

    }
}
```

In this example, the Shape interface defines a method for drawing a shape, and the Circle and Rectangle classes implement the Shape interface. The ShapeFactory class has a factory method called getShape() that takes a string argument specifying the type of shape to create. The getShape() method returns a new instance of the appropriate type of shape based on the value of the argument.

The client code can use the ShapeFactory to create shapes as needed by calling the getShape() method and passing the appropriate argument.

The Factory pattern is a useful pattern to use when it is necessary to create objects of a particular type, but the exact type is not known until runtime. It allows the client code to create objects without knowing the exact class to create, and it can help to improve the maintainability and extensibility of code by allowing new types of objects to be added easily.

Abstract factory pattern

The Abstract Factory pattern is a design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is a useful pattern to use when it is necessary to create groups of objects that are related to each other in some way and when it is important to create these objects in a consistent manner.

The Abstract Factory pattern is implemented by creating an abstract factory class that defines an interface for creating objects. This interface includes methods for creating the various types of objects that are needed. Concrete factory classes are then created that implement the abstract factory interface and create the concrete objects. The client code uses the abstract factory interface to create the objects, and the concrete factory classes are responsible for creating the objects and ensuring that they are created in a consistent manner.

Here is an example of how the Abstract Factory pattern might be implemented in Java:

```
interface Animal {
  void speak();
}

class Dog implements Animal {
  @Override
  public void speak() {
    System.out.println("Woof!");
  }
}

class Cat implements Animal {
  @Override
  public void speak() {
    System.out.println("Meow!");
  }
}

interface AnimalFactory {
  Animal createAnimal();
}

class DogFactory implements AnimalFactory {
  @Override
  public Animal createAnimal() {
    return new Dog();
  }
}

class CatFactory implements AnimalFactory {
  @Override
```

```
  public Animal createAnimal() {
    return new Cat();
  }
}

public class AbstractFactory {
     public static void main(String[] args)
     {
          AnimalFactory dogFactory = new DogFactory();
          Animal dog = dogFactory.createAnimal();
          dog.speak();

          AnimalFactory catFactory = new CatFactory();
          Animal cat = catFactory.createAnimal();
          cat.speak();

     }
}
```

In this example, the Animal interface defines a method for making a sound, and the Dog and Cat classes implement the Animal interface. The AnimalFactory interface defines a method for creating an Animal object, and the DogFactory and CatFactory classes implement the AnimalFactory interface and create concrete Animal objects.

The client code can use the AnimalFactory interface to create Animal objects as needed by calling the createAnimal() method and passing the appropriate factory class.

The Abstract Factory pattern is a useful pattern to use when it is necessary to create groups of objects that are related to each other in some way and when it is important to create these objects in a consistent manner. It allows the client code to create objects without knowing the exact class to create, and it can help to improve the maintainability and extensibility of code by allowing new types of objects to be added easily.

Builder pattern

The Builder pattern is a design pattern that separates the construction of a complex object from its representation, allowing the same construction process to create various representations. It is a useful pattern to use when it is necessary to create complex objects that have a large number of variables, and when it is important to allow these objects to be created in a flexible and customizable way.

The Builder pattern is implemented by creating a builder class that has methods for setting the various variables that define the complex object. The builder class also has a method for creating the complex object, which returns the object in its final form. The client code uses the builder class to set the variables and create the complex object as needed.

Here is an example of how the Builder pattern might be implemented in Java:

```
class Car {
  private String make;
  private String model;
  private int year;
  private String color;
  private int horsepower;
  private int torque;

  private Car(CarBuilder builder) {
    this.make = builder.make;
    this.model = builder.model;
    this.year = builder.year;
    this.color = builder.color;
    this.horsepower = builder.horsepower;
    this.torque = builder.torque;
  }

  public static class CarBuilder {
    private String make;
    private String model;
    private int year;
    private String color;
    private int horsepower;
    private int torque;

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

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

```
    }

    public CarBuilder setYear(int year) {
      this.year = year;
      return this;
    }

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

    public CarBuilder setHorsepower(int horsepower) {
      this.horsepower = horsepower;
      return this;
    }

    public CarBuilder setTorque(int torque) {
      this.torque = torque;
      return this;
    }

    public Car build() {
      return new Car(this);
    }
  }
}


public class Builder {
  public static void main(String[] args)
  {
      Car car = new Car.CarBuilder()
         .setMake("Ford")
         .setModel("Mustang")
         .setYear(2020)
         .setColor("Red")
         .setHorsepower(450)
         .setTorque(420)
         .build();

  }
}
```

In this example, the Car class has a private constructor that takes a CarBuilder object as an argument. The CarBuilder class has methods for setting the various variables that define the Car object, and it also has a build() method that creates and returns a new Car object. The client code can use the CarBuilder class to create a Car object.

Prototype pattern

The Prototype pattern is a design pattern that allows objects to be created by copying existing objects, rather than by creating new objects from scratch. It is a useful pattern to use when it is necessary to create objects that are similar to existing objects, but with some variations.

The Prototype pattern is implemented by creating a prototype interface that defines a method for creating a copy of an object. Concrete prototype classes are then created that implement the prototype interface and provide a concrete implementation of the clone() method. The client code uses the prototype interface to create copies of the objects as needed.

Here is an example of how the Prototype pattern might be implemented in Java:

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

class ConcretePrototypeA implements Prototype {
  private String property;

  public ConcretePrototypeA(String property) {
    this.property = property;
  }

  @Override
  public Prototype clone() {
    return new ConcretePrototypeA(property);
  }
}

class ConcretePrototypeB implements Prototype {
  private int property;

  public ConcretePrototypeB(int property) {
    this.property = property;
  }

  @Override
  public Prototype clone() {
    return new ConcretePrototypeB(property);
  }
}

public class PrototypeImpl {
     public static void main(String[] args)
     {
           Prototype prototypeA = new ConcretePrototypeA("Hello");
```

```
            Prototype prototypeACopy = prototypeA.clone();

            Prototype prototypeB = new ConcretePrototypeB(123);
            Prototype prototypeBCopy = prototypeB.clone();
        }
}
```

In this example, the Prototype interface defines a clone() method that creates a copy of the object. The ConcretePrototypeA and ConcretePrototypeB classes implement the Prototype interface and provide a concrete implementation of the clone() method.

The client code can use the Prototype interface to create copies of the objects as needed by calling the clone() method.

The Prototype pattern is a useful pattern to use when it is necessary to create objects that are similar to existing objects, but with some variations. It allows objects to be created by copying existing objects, rather than by creating new objects from scratch, and it can help to improve the efficiency and performance of code by avoiding the need to create new objects unnecessarily.

Adapter pattern

The Adapter pattern is a design pattern that allows two incompatible interfaces to work together. It is a useful pattern to use when it is necessary to use an existing class, but its interface does not match the interface required by the client code.

The Adapter pattern is implemented by creating an adapter class that implements the desired interface and contains an instance of the class that needs to be adapted. The adapter class translates the calls made by the client code into calls that the adapted class can understand.

Here is an example of how the Adapter pattern might be implemented in Java:

```
interface MediaPlayer {
  void play(String audioType, String fileName);
}

class VlcPlayer implements MediaPlayer {
  @Override
  public void play(String audioType, String fileName) {
    // code to play VLC file
  }
}

class Mp4Player implements MediaPlayer {
  @Override
  public void play(String audioType, String fileName) {
    // code to play MP4 file
  }
}

class AudioPlayer implements MediaPlayer {
  MediaPlayer mediaPlayer;

  @Override
  public void play(String audioType, String fileName) {
    if (audioType.equalsIgnoreCase("vlc")) {
      mediaPlayer = new VlcPlayer();
      mediaPlayer.play(audioType, fileName);
    } else if (audioType.equalsIgnoreCase("mp4")) {
      mediaPlayer = new Mp4Player();
      mediaPlayer.play(audioType, fileName);
    } else {
      System.out.println("Invalid audio type.");
    }
  }
}
```

```
public class Adapter {
	public static void main(String[] args)
	{
		MediaPlayer audioPlayer = new AudioPlayer();

		audioPlayer.play("vlc", "beyond the horizon.vlc");
		audioPlayer.play("mp4", "alone.mp4");
		audioPlayer.play("avi", "mind me.avi");
	}
}
```

In this example, the MediaPlayer interface defines a method for playing a media file, and the VlcPlayer and Mp4Player classes implement the MediaPlayer interface and provide a concrete implementation of the play() method for playing VLC and MP4 files, respectively.

The AudioPlayer class is the adapter class that implements the MediaPlayer interface and contains an instance of the MediaPlayer interface. The AudioPlayer class translates the calls made by the client code into calls that the adapted class (either VlcPlayer or Mp4Player) can understand.

The client code can use the AudioPlayer class to play media files.

The Adapter pattern is a useful pattern to use when it is necessary to use an existing class, but its interface does not match the interface required by the client code. It allows two incompatible interfaces to work together by adapting one interface to the other, and it can help to improve the flexibility and reusability of code by allowing existing classes to be used in new ways.

Bridge pattern

The Bridge pattern is a design pattern that allows an abstraction and its implementation to be defined and modified independently. It is a useful pattern to use when it is necessary to change the implementation of an abstraction without changing the abstraction itself.

The Bridge pattern is implemented by creating an abstraction class that defines the interface for the abstraction, and a concrete implementation class that provides the implementation for the abstraction. The abstraction class contains a reference to the concrete implementation class, and it delegates calls to the implementation class as needed.

Here is an example of how the Bridge pattern might be implemented in Java:

```
interface Color {
  void applyColor();
}

class RedColor implements Color {
  @Override
  public void applyColor() {
    System.out.println("Applying red color");
  }
}

class GreenColor implements Color {
  @Override
  public void applyColor() {
    System.out.println("Applying green color");
  }
}

abstract class ShapeBridge {
  protected Color color;

  public ShapeBridge(Color color) {
    this.color = color;
  }

  public abstract void drawShape();
  public abstract void modifyBorder(int border, Color color);
}

class Triangle extends ShapeBridge {
  public Triangle(Color color) {
    super(color);
  }
```

```
  @Override
  public void drawShape() {
    System.out.print("Drawing Triangle with color ");
    color.applyColor();
  }

  @Override
  public void modifyBorder(int border, Color color) {
    System.out.println("Modifying the border length " + border + "
and color " + color);
  }
}

class RectangleBridge extends ShapeBridge {
  public RectangleBridge(Color color) {
    super(color);
  }

  @Override
  public void drawShape() {
    System.out.print("Drawing Rectangle with color ");
    color.applyColor();
  }

  @Override
  public void modifyBorder(int border, Color color) {
    System.out.println("Modifying the border length " + border + "
and color " + color);
  }
}

public class Bridge {
  public static void main(String[] args)
  {
      ShapeBridge triangle = new Triangle(new RedColor());
      triangle.drawShape();
      triangle.modifyBorder(20, new GreenColor());

      ShapeBridge rectangle = new RectangleBridge(new GreenColor());
      rectangle.drawShape();
      rectangle.modifyBorder(40, new RedColor());

  }
}
```

In this example, the Color interface defines a method for applying a color, and the RedColor and GreenColor classes implement the Color interface and provide a concrete implementation for the applyColor() method.

The Shape abstract class is the abstraction class that defines the interface for the abstraction, and it contains a reference to the Color interface. The Triangle and Rectangle classes are concrete implementation classes that provide the implementation for the Shape abstraction.

The Bridge pattern is a useful pattern to use when it is necessary to change the implementation of an abstraction without changing the abstraction itself. It allows the abstraction and its implementation to be defined and modified independently, and it can help to improve the flexibility and reusability of code by allowing the implementation of an abstraction to be changed without affecting the abstraction.

Composite pattern

The Composite pattern is a design pattern that allows a group of objects to be treated as a single object. It is a useful pattern to use when it is necessary to represent a hierarchical structure, such as a tree, where some objects are composed of other objects.

The Composite pattern is implemented by creating a component interface that defines the interface for the objects in the composite, and creating concrete component classes that implement the component interface. A composite class is then created that contains a collection of component objects, and it provides methods for adding and removing components, as well as methods for accessing and manipulating the components.

Here is an example of how the Composite pattern might be implemented in Java:

```
import java.util.ArrayList;
import java.util.List;

interface Component {
  void add(Component component);
  void remove(Component component);
  Component getChild(int index);
  void operation();
}

class Leaf implements Component {
  @Override
  public void add(Component component) {
    // this operation is not supported for leaf nodes
  }

  @Override
  public void remove(Component component) {
    // this operation is not supported for leaf nodes
  }

  @Override
  public Component getChild(int index) {
    // this operation is not supported for leaf nodes
    return null;
  }

  @Override
  public void operation() {
    // code to perform the operation for a leaf node
  }
}
```

```
public class Composite implements Component {
  private List<Component> components = new ArrayList<>();

  @Override
  public void add(Component component) {
    components.add(component);
  }

  @Override
  public void remove(Component component) {
    components.remove(component);
  }

  @Override
  public Component getChild(int index) {
    return components.get(index);
  }

  @Override
  public void operation() {
    // code to perform the operation for a composite node
    for (Component component : components) {
      component.operation();
    }
  }

  public static void main(String[] args)
  {
      Component leaf1 = new Leaf();
      Component leaf2 = new Leaf();

      Component composite = new Composite();
      composite.add(leaf1);
      composite.add(leaf2);

      composite.operation();
  }
}
```

In this example, the Component interface defines the interface for the objects in the composite, and the Leaf and Composite classes implement the Component interface and provide concrete implementations for the various methods.

The Composite class contains a collection of Component objects, and it provides methods for adding and removing components, as well as methods for accessing and manipulating the components. The operation() method of the Composite class iterates over its child components and calls the operation() method for each of them.

The Composite pattern is a useful pattern to use when it is necessary to represent a hierarchical structure, such as a tree, where some objects are composed of other objects. It allows a group of objects to be treated as a single object, and it can help to improve the flexibility and reusability of code by allowing complex structures to be built up from simpler objects.

Decorator pattern

The Decorator pattern is a design pattern that allows new behavior to be added to an existing object dynamically, without changing the object's implementation. It is a useful pattern to use when it is necessary to add new functionality to an object, but it is not possible or desirable to modify the object's implementation directly.

The Decorator pattern is implemented by creating a decorator abstract class that implements the interface of the object being decorated and contains a reference to the object being decorated. Concrete decorator classes are then created that extend the decorator abstract class and provide the desired new behavior.

Here is an example of how the Decorator pattern might be implemented in Java:

```
interface CarDecorator {
  void assemble();
}

class BasicCar implements CarDecorator {
  @Override
  public void assemble() {
    System.out.print("Basic car.");
  }
}

public abstract class Decorator implements CarDecorator {
  protected CarDecorator car;

  public Decorator(CarDecorator car) {
    this.car = car;
  }

  @Override
  public void assemble() {
    this.car.assemble();
  }

  public static void main(String[] args)
  {
      CarDecorator sportsCar = new SportsCar(new BasicCar());
      sportsCar.assemble();

      CarDecorator luxuryCar = new LuxuryCar(new BasicCar());
      luxuryCar.assemble();

  }
}
```

```
class SportsCar extends Decorator {
  public SportsCar(CarDecorator car) {
    super(car);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.print(" Adding features of Sports Car.");
  }
}

class LuxuryCar extends Decorator {
  public LuxuryCar(CarDecorator car) {
    super(car);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.print(" Adding features of Luxury Car.");
  }
}
```

In this example, the CarDecorator interface defines the interface for the object being decorated, and the BasicCar class implements the CarDecorator interface and provides a concrete implementation for the assemble() method.

The Decorator abstract class is the decorator class that implements the CarDecorator interface and contains a reference to the CarDecorator object being decorated. The SportsCar and LuxuryCar concrete decorator classes extend the Decorator abstract class and provide the desired new behavior.

In this example, the SportsCar decorator adds the features of a sports car to the basic car, and the LuxuryCar decorator adds the features of a luxury car to the basic car. The decorators are applied dynamically.

Facade pattern

The Facade pattern is a design pattern that provides a simplified interface to a complex system. It is a useful pattern to use when it is necessary to work with a complex system, but it is not desirable or necessary for the client code to be aware of the complexity of the system.

The Facade pattern is implemented by creating a facade class that provides a simplified interface to the complex system. The facade class contains a reference to the objects in the complex system and provides methods that delegate the requests from the client code to the appropriate objects in the complex system.

Here is an example of how the Facade pattern might be implemented in Java:

```
class ComplexSystem {
  public void method1() {
    // code to perform method 1 of the complex system
  }

  public void method2() {
    // code to perform method 2 of the complex system
  }

  public void method3() {
    // code to perform method 3 of the complex system
  }
}

public class Facade {
  private ComplexSystem complexSystem;

  public Facade(ComplexSystem complexSystem) {
    this.complexSystem = complexSystem;
  }

  public void methodA() {
    complexSystem.method1();
    complexSystem.method2();
  }

  public void methodB() {
    complexSystem.method2();
    complexSystem.method3();
  }

  public static void main(String[] args)
  {
      ComplexSystem complexSystem = new ComplexSystem();
```

```
        Facade facade = new Facade(complexSystem);

        facade.methodA();
        facade.methodB();
    }
}
```

In this example, the ComplexSystem class represents the complex system, and the Facade class provides a simplified interface to the complex system. The Facade class contains a reference to the ComplexSystem object and provides methods methodA() and methodB() that delegate the requests from the client code to the appropriate methods of the complex system.

In this example, the methodA() and methodB() methods of the Facade class provide a simplified interface to the complex system, allowing the client code to work with the complex system without needing to be aware of its complexity.

The Facade pattern is a useful pattern to use when it is necessary to work with a complex system, but it is not desirable or necessary for the client code to be aware of the complexity of the system. It provides a simplified interface to the complex system, making it easier for the client code to work with the system and reducing the coupling between the client code and the complex system.

Flyweight pattern

The Flyweight pattern is a design pattern that is used to minimize memory usage by sharing common data among objects. It is a useful pattern to use when it is necessary to work with a large number of objects, but it is not practical or desirable to store all of the data for each object in memory.

The Flyweight pattern is implemented by creating a flyweight abstract class that defines the interface for the objects being shared, and creating concrete flyweight classes that implement the flyweight interface and contain the shared data. A flyweight factory is then created that manages the flyweight objects and provides methods for creating and accessing them.

Here is an example of how the Flyweight pattern might be implemented in Java:

```
import java.util.HashMap;
import java.util.Map;

interface Flyweight {
  void operation(int extrinsicState);
}

class ConcreteFlyweight implements Flyweight {
  private int intrinsicState;

  public ConcreteFlyweight(int intrinsicState) {
    this.intrinsicState = intrinsicState;
  }

  @Override
  public void operation(int extrinsicState) {
    System.out.println("Intrinsic State: " + intrinsicState + ",
Extrinsic State: " + extrinsicState);
  }
}

public class FlyweightFactory {
  private Map<Integer, Flyweight> flyweights = new HashMap<>();

  public Flyweight getFlyweight(int intrinsicState) {
    Flyweight flyweight = flyweights.get(intrinsicState);
    if (flyweight == null) {
      flyweight = new ConcreteFlyweight(intrinsicState);
      flyweights.put(intrinsicState, flyweight);
    }
    return flyweight;
  }
```

```
  public static void main(String[] args)
  {
      FlyweightFactory flyweightFactory = new FlyweightFactory();

      Flyweight flyweight1 = flyweightFactory.getFlyweight(1);
      flyweight1.operation(2);
  }
}
```

In this example, the Flyweight interface defines the interface for the flyweight objects being shared, and the ConcreteFlyweight class implements the Flyweight interface and contains the shared data. The FlyweightFactory class manages the flyweight objects and provides methods for creating and accessing them.

Proxy pattern

The Proxy pattern is a design pattern that is used to provide a surrogate or placeholder object for another object. It is a useful pattern to use when it is necessary to control access to an object, or when it is necessary to add additional behavior to an object at runtime.

The Proxy pattern is implemented by creating a proxy class that implements the same interface as the object being proxied and contains a reference to the object being proxied. The proxy class provides methods that delegate the requests from the client code to the object being proxied, and may also add additional behavior to the request before it is forwarded to the object being proxied.

Here is an example of how the Proxy pattern might be implemented in Java:

```
interface Subject {
  void request();
}

class RealSubject implements Subject {
  @Override
  public void request() {
    System.out.println("Handling request.");
  }
}

public class Proxy implements Subject {
  private Subject subject;

  public Proxy(Subject subject) {
    this.subject = subject;
  }

  @Override
  public void request() {
    System.out.println("Before handling request.");
    subject.request();
    System.out.println("After handling request.");
  }

  public static void main(String[] args)
  {
      Subject subject = new Proxy(new RealSubject());
      subject.request();
  }
}
```

In this example, the Subject interface defines the interface for the object being proxied, and the RealSubject class implements the Subject interface and provides a concrete implementation for the request() method.

The Proxy class is the proxy class that implements the Subject interface and contains a reference to the Subject object being proxied. The Proxy class provides a method request() that delegates the request from the client code to the Subject object being proxied, and adds additional behavior before and after the request is forwarded to the Subject object.

In this example, the Proxy object is created with a reference to the RealSubject object and is used to handle the request from the client code. The Proxy object adds additional behavior before and after the request is forwarded to the RealSubject object, and the RealSubject object handles the request.

The Proxy pattern is a useful pattern to use when it is necessary to control access to an object, or when it is necessary to add additional behavior to an object at runtime. It provides a surrogate or placeholder object for another object, allowing the client code to work with the proxy object instead of the object being proxied.

Chain of responsibility pattern

The Chain of Responsibility pattern is a design pattern that is used to decouple the sender of a request from the receiver of the request. It is a useful pattern to use when it is necessary to process a request, but it is not known which object in a chain of objects should handle the request.

The Chain of Responsibility pattern is implemented by creating a base handler class that defines the interface for handling requests, and creating concrete handler classes that implement the handler interface and contain a reference to the next handler in the chain. A client class is then created that creates the chain of handlers and sends the request to the first handler in the chain.

Here is an example of how the Chain of Responsibility pattern might be implemented in Java:

```
interface Handler {
  void setNext(Handler handler);
  void handleRequest(int request);
}

class ConcreteHandlerA implements Handler {
  private Handler next;

  @Override
  public void setNext(Handler handler) {
    next = handler;
  }

  @Override
  public void handleRequest(int request) {
    if (request % 2 == 0) {
      System.out.println("ConcreteHandlerA handled request " +
request);
    } else {
      next.handleRequest(request);
    }
  }
}

class ConcreteHandlerB implements Handler {
  private Handler next;

  @Override
  public void setNext(Handler handler) {
    next = handler;
  }
```

```
  @Override
  public void handleRequest(int request) {
    if (request % 3 == 0) {
      System.out.println("ConcreteHandlerB handled request " +
request);
    } else {
      next.handleRequest(request);
    }
  }
}

class ConcreteHandlerC implements Handler {
  private Handler next;

  @Override
  public void setNext(Handler handler) {
    next = handler;
  }

  @Override
  public void handleRequest(int request) {
    if (request % 5 == 0) {
      System.out.println("ConcreteHandlerC handled request " +
request);
    } else {
      next.handleRequest(request);
    }
  }
}

public class ChainofResponsibility {
  public static void main(String[] args) {
    Handler handlerA = new ConcreteHandlerA();
    Handler handlerB = new ConcreteHandlerB();
    Handler handlerC = new ConcreteHandlerC();
    handlerA.setNext(handlerB);
    handlerB.setNext(handlerC);

    handlerA.handleRequest(2);
    handlerA.handleRequest(3);
    handlerA.handleRequest(4);
    handlerA.handleRequest(5);
    handlerA.handleRequest(6);
  }
}
```

In this example, the Handler interface defines the interface for handling requests, and the ConcreteHandlerA and ConcreteHandlerB classes implement the Handler interface and contain

a reference to the next handler in the chain. The Client class creates the chain of handlers and sends the request to the first handler in the chain.

The ConcreteHandlerA class handles requests that are even numbers, and the ConcreteHandlerB class handles requests that are multiples of 3. If a handler is unable to handle a request, it passes the request to the next handler in the chain.

Command pattern

The Command pattern is a design pattern that is used to encapsulate a request as an object, allowing the request to be treated as a separate entity and passed to different objects. It is a useful pattern to use when it is necessary to issue a request to an object, but it is not known which object should handle the request.

The Command pattern is implemented by creating a command interface that defines the interface for executing a request, and creating concrete command classes that implement the command interface and contain a reference to the object that should handle the request. A client class is then created that creates the command object and invokes the execute() method on the command object to issue the request.

Here is an example of how the Command pattern might be implemented in Java:

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

class ConcreteCommand implements Command {
  private Receiver receiver;

  public ConcreteCommand(Receiver receiver) {
    this.receiver = receiver;
  }

  @Override
  public void execute() {
    receiver.action();
  }
}

class Receiver {
  public void action() {
    System.out.println("Handling request.");
  }
}

class Invoker {
  private Command command;

  public Invoker(Command command) {
    this.command = command;
  }

  public void invoke() {
    command.execute();
```

```
  }
}

public class CommandPattern {
  public static void main(String[] args) {
    Receiver receiver = new Receiver();
    Command command = new ConcreteCommand(receiver);
    Invoker invoker = new Invoker(command);
    invoker.invoke();
  }
}
```

In this example, the Command interface defines the interface for executing a request, and the ConcreteCommand class implements the Command interface and contains a reference to the Receiver object that should handle the request. The Invoker class creates the command object and invokes the execute() method on the command object to issue the request.

The Receiver class provides a concrete implementation for handling the request.

In this example, the CommandPattern class creates the Receiver object, the ConcreteCommand object with a reference to the Receiver object, and the Invoker object with a reference to the ConcreteCommand object. The Invoker object is then used to issue the request by invoking the invoke() method, which in turn invokes the execute() method on the ConcreteCommand object. The ConcreteCommand object then forwards the request to the Receiver object to handle the request.

The Command pattern is a useful pattern to use when it is necessary to issue a request to an object, but it is not known which object should handle the request. It encapsulates the request as an object, allowing the request to be treated as a separate entity and passed to different objects.

Interpreter pattern

The Interpreter pattern is a design pattern that is used to interpret a language or notation. It is a useful pattern to use when it is necessary to interpret a language or notation, but the language or notation is not known in advance and may change over time.

The Interpreter pattern is implemented by creating an interpreter interface that defines the interface for interpreting a language or notation, and creating concrete interpreter classes that implement the interpreter interface and contain the logic for interpreting the language or notation. A client class is then created that creates the interpreter object and invokes the interpret() method on the interpreter object to interpret the language or notation.

Here is an example of how the Interpreter pattern might be implemented in Java:

```
interface Expression {
  int interpret();
}

class NumberExpression implements Expression {
  private int number;

  public NumberExpression(int number) {
    this.number = number;
  }

  @Override
  public int interpret() {
    return number;
  }
}

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();
  }
}
```

```
class SubtractionExpression implements Expression {
  private Expression leftExpression;
  private Expression rightExpression;

  public SubtractionExpression(Expression leftExpression, Expression
rightExpression) {
    this.leftExpression = leftExpression;
    this.rightExpression = rightExpression;
  }

  @Override
  public int interpret() {
    return leftExpression.interpret() - rightExpression.interpret();
  }
}

public class Interpreter {
  public static void main(String[] args) {
    Expression expression = new AdditionExpression(
      new NumberExpression(10),
      new NumberExpression(20)
    );
    int result = expression.interpret();
    System.out.println("Result: " + result);

    expression = new SubtractionExpression(
        new NumberExpression(10),
        new NumberExpression(20)
      );
    result = expression.interpret();
    System.out.println("Result: " + result);
  }
}
```

In this example, the Expression interface defines the interface for interpreting a language or notation, and the NumberExpression and AdditionExpression classes implement the Expression interface and contain the logic for interpreting the language or notation. The Client class creates the interpreter object and invokes the interpret() method on the interpreter object to interpret the language or notation.

The NumberExpression class represents a number in the language or notation, and the AdditionExpression class represents an addition operation in the language or notation.

Iterator pattern

The Iterator pattern is a design pattern that is used to iterate over a collection of objects. It is a useful pattern to use when it is necessary to iterate over a collection of objects, but the collection may be of different types and it is not possible or desirable to use a single loop to iterate over all of the objects.

The Iterator pattern is implemented by creating an iterator interface that defines the interface for iterating over a collection of objects, and creating concrete iterator classes that implement the iterator interface and contain the logic for iterating over the collection of objects. A client class is then created that creates the iterator object and invokes the hasNext() and next() methods on the iterator object to iterate over the collection of objects.

Here is an example of how the Iterator pattern might be implemented in Java:

```
import java.util.Arrays;
import java.util.List;

interface Iterator<T> {
  boolean hasNext();
  T next();
}

interface Collection<T> {
  Iterator<T> iterator();
}

class ConcreteIterator<T> implements Iterator<T> {
  private ConcreteCollection<T> collection;
  private int index;

  public ConcreteIterator(ConcreteCollection<T> collection) {
    this.collection = collection;
    this.index = 0;
  }

  @Override
  public boolean hasNext() {
    return index < collection.size();
  }

  @Override
  public T next() {
    if (hasNext()) {
      return collection.get(index++);
    }
    return null;
```

```
  }
}

class ConcreteCollection<T> implements Collection<T> {
  private List<T> list;

  public ConcreteCollection(List<T> list) {
    this.list = list;
  }

  public int size() {
    return list.size();
  }

  public T get(int index) {
    return list.get(index);
  }

  @Override
  public Iterator<T> iterator() {
    return new ConcreteIterator<>(this);
  }
}

public class IteratorPattern {
  public static void main(String[] args) {
    List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
    Collection<Integer> collection = new ConcreteCollection<>(list);
    Iterator<Integer> iterator = collection.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }
  }
}
```

In this example, the Iterator interface defines the interface for iterating over a collection of objects, and the ConcreteIterator class implements the Iterator interface and contains the logic for iterating over the ConcreteCollection object. The Collection interface defines the interface for creating an iterator object, and the ConcreteCollection class implements the Collection interface and contains a reference to the ConcreteIterator object.

The IteratorPattern class creates the ConcreteCollection object, invokes the iterator() method on the ConcreteCollection object to create the ConcreteIterator object, and then uses a while loop to iterate over the ConcreteCollection object by invoking the hasNext() and next() methods on the ConcreteIterator object.

Mediator pattern

The Mediator pattern is a design pattern that is used to provide a centralized communication point for a group of objects. It is a useful pattern to use when it is necessary to communicate between a group of objects, but it is not desirable or feasible to have the objects communicate directly with each other.

The Mediator pattern is implemented by creating a mediator interface that defines the interface for communication between the objects, and creating concrete mediator classes that implement the mediator interface and contain the logic for communication between the objects. The objects that need to communicate with each other are then created and given a reference to the mediator object.

Here is an example of how the Mediator pattern might be implemented in Java:

```
interface Mediator {
  void send(String message, Colleague colleague);
}

abstract class Colleague {
  private Mediator mediator;

  public Colleague(Mediator mediator) {
    this.mediator = mediator;
  }

  public void send(String message) {
    mediator.send(message, this);
  }

  public abstract void receive(String message);
}

class ConcreteMediator implements Mediator {
  private Colleague colleague1;
  private Colleague colleague2;

  public void setColleague1(Colleague colleague1) {
    this.colleague1 = colleague1;
  }

  public void setColleague2(Colleague colleague2) {
    this.colleague2 = colleague2;
  }

  @Override
  public void send(String message, Colleague colleague) {
```

```
    if (colleague == colleague1) {
      colleague2.receive(message);
    } else {
      colleague1.receive(message);
    }
  }
}

class ConcreteColleague1 extends Colleague {
  public ConcreteColleague1(Mediator mediator) {
    super(mediator);
  }

  @Override
  public void receive(String message) {
    System.out.println("Colleague1 received: " + message);
  }
}

class ConcreteColleague2 extends Colleague {
  public ConcreteColleague2(Mediator mediator) {
    super(mediator);
  }

  @Override
  public void receive(String message) {
    System.out.println("Colleague2 received: " + message);
  }
}

public class MediatorPattern {
  public static void main(String[] args) {
    Mediator mediator = new ConcreteMediator();
    Colleague colleague1 = new ConcreteColleague1(mediator);
    Colleague colleague2 = new ConcreteColleague2(mediator);
    ((ConcreteMediator) mediator).setColleague1(colleague1);
    ((ConcreteMediator) mediator).setColleague2(colleague2);
    colleague1.send("Hello, how are you?");
    colleague2.send("I'm fine, thank you. How about you?");
  }
}
```

In this example, the Mediator interface defines the interface for communication between the objects, and the ConcreteMediator class implements.

Memento pattern

The Memento pattern is a design pattern that is used to capture the internal state of an object and store it in such a way that the object can be restored to its previous state later. It is a useful pattern to use when it is necessary to store the state of an object, but it is not desirable or feasible to store the state directly on the object or to expose the object's internal state to external manipulation.

The Memento pattern is implemented by creating a memento class that stores the internal state of the object, and a caretaker class that is responsible for creating and storing the memento objects. The object whose state needs to be stored is then given a method to create a memento object and another method to restore its state from a memento object.

Here is an example of how the Memento pattern might be implemented in Java:

```
import java.util.ArrayList;
import java.util.List;

public class Memento {
  private String state;

  public Memento(String state) {
    this.state = state;
  }

  public String getState() {
    return state;
  }

  public static void main(String[] args) {
    Originator originator = new Originator();
    Caretaker caretaker = new Caretaker();
    originator.setState("State 1");
    caretaker.addMemento(originator.createMemento());
    originator.setState("State 2");
    caretaker.addMemento(originator.createMemento());
    originator.setState("State 3");
    caretaker.addMemento(originator.createMemento());
    originator.setState("State 4");
    System.out.println("Current state: " + originator.getState());
    originator.restoreFromMemento(caretaker.getMemento(1));
    System.out.println("Current state: " + originator.getState());
  }
}

class Originator {
  private String state;
```

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

  public String getState() {
    return state;
  }

  public Memento createMemento() {
    return new Memento(state);
  }

  public void restoreFromMemento(Memento memento) {
    state = memento.getState();
  }
}

class Caretaker {
  private List<Memento> mementos = new ArrayList<>();

  public void addMemento(Memento memento) {
    mementos.add(memento);
  }

  public Memento getMemento(int index) {
    return mementos.get(index);
  }
}
```

Caretakerclass is responsible for creating and storing theMementoobjects. TheOriginatorobject has a method to create aMementoobject and another method to restore its state from aMemento` object.

To use the Memento pattern, the client first creates an Originator object and sets its initial state. The client then creates a Caretaker object and adds the Memento objects created by the Originator to the Caretaker. The client can then modify the state of the Originator object as needed, and create new Memento objects to store the updated state in the Caretaker. If the client needs to restore the state of the Originator object, it can retrieve the desired Memento object from the Caretaker and pass it to the Originator's restore method.

The Memento pattern is a useful pattern for storing the state of an object in a way that is decoupled from the object itself. It allows the object to be restored to a previous state without exposing its internal representation, and it allows the object's state to be stored and retrieved without the client having to know the details of the object's implementation.

Observer pattern

The Observer pattern is a design pattern that is used to allow an object to be notified when the state of another object changes. It is a useful pattern to use when it is necessary for one object to be aware of changes in the state of another object, but it is not desirable or feasible for the objects to be directly coupled.

The Observer pattern is implemented by creating an observer interface that defines the interface for notification of state changes, and creating concrete observer classes that implement the observer interface and contain the logic for reacting to state changes. The object whose state is being observed is then given a method to register and unregister observer objects, and a method to notify the observer objects when its state changes.

Here is an example of how the Observer pattern might be implemented in Java:

```
import java.util.ArrayList;
import java.util.List;

interface Observer {
  void update(String state);
}

interface SubjectObserver {
  void registerObserver(Observer observer);
  void removeObserver(Observer observer);
  void notifyObservers();
}

class ConcreteObserver implements Observer {
  private String state;

  @Override
  public void update(String state) {
    this.state = state;
    System.out.println("Observer: state updated to " + state);
  }
}

class ConcreteSubject implements SubjectObserver {
  private List<Observer> observers = new ArrayList<>();
  private String state;

  @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(state);
    }
  }

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

public class ObserverPattern {
  public static void main(String[] args) {
    ConcreteSubject subject = new ConcreteSubject();
    Observer observer1 = new ConcreteObserver();
    Observer observer2 = new ConcreteObserver();
    subject.registerObserver(observer1);
    subject.registerObserver(observer2);
    subject.setState("State 1");
    subject.setState("State 2");
  }
}
```

In this example, the Observer interface defines the interface for notification of state changes, and the ConcreteObserver class implements the Observer interface and contains the logic for reacting to state changes. The Subject interface defines the methods for registering and unregistering observer objects, and the ConcreteSubject class implements the Subject interface and contains the logic for notifying the observer objects when its state changes.

To use the Observer pattern, the client first creates a Subject object and Observer objects. The client then registers the Observer objects with the Subject, and sets the state of the Subject. Whenever the state of the Subject changes, the Subject object notifies the registered Observer objects, which update their own state in response.

State pattern

The State pattern is a design pattern that is used to allow an object to alter its behavior when its internal state changes. It is a useful pattern to use when an object's behavior depends on its state, and it is necessary to change the object's behavior at runtime depending on the state it is in.

The State pattern is implemented by creating a state interface that defines the interface for state-specific behavior, and creating concrete state classes that implement the state interface and contain the logic for the state-specific behavior. The object whose behavior is being controlled is then given a method to set the current state, and a method to delegate behavior to the current state.

Here is an example of how the State pattern might be implemented in Java:

```
interface State {
  void handle();
}

class ConcreteStateA implements State {
  @Override
  public void handle() {
    System.out.println("Handling in ConcreteStateA");
  }
}

class ConcreteStateB implements State {
  @Override
  public void handle() {
    System.out.println("Handling in ConcreteStateB");
  }
}

class Context {
  private State state;

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

  public void handle() {
    state.handle();
  }
}

public class StatePattern {
  public static void main(String[] args) {
```

```
    Context context = new Context();
    context.setState(new ConcreteStateA());
    context.handle();
    context.setState(new ConcreteStateB());
    context.handle();
  }
}
```

In this example, the State interface defines the interface for state-specific behavior, and the ConcreteStateA and ConcreteStateB classes implement the State interface and contain the logic for the state-specific behavior. The Context class is the object whose behavior is being controlled, and it has a method to set the current state and a method to delegate behavior to the current state.

To use the State pattern, the client first creates a Context object and State objects. The client then sets the state of the Context object, and calls the handle method on the Context. The Context object delegates the behavior to the current state, which performs the state-specific behavior.

The State pattern is a useful pattern for allowing an object to alter its behavior at runtime depending on its internal state. It allows the object's behavior to be modified without changing the object's class, and it allows the object to be more flexible and adaptable to changing requirements.

Strategy pattern

The Strategy pattern is a design pattern that is used to allow an object to alter its behavior at runtime by selecting from a list of behaviors that are implemented as separate strategy classes. It is a useful pattern to use when it is necessary for an object to have multiple behaviors that can be changed at runtime, and it is not desirable or feasible to use inheritance to achieve this behavior.

The Strategy pattern is implemented by creating a strategy interface that defines the interface for the behavior, and creating concrete strategy classes that implement the strategy interface and contain the logic for the behavior. The object that needs to alter its behavior at runtime is then given a method to set the current strategy, and a method to delegate behavior to the current strategy.

Here is an example of how the Strategy pattern might be implemented in Java:

```
interface Strategy {
  void execute();
}

class ConcreteStrategyA implements Strategy {
  @Override
  public void execute() {
    System.out.println("Executing in ConcreteStrategyA");
  }
}

class ConcreteStrategyB implements Strategy {
  @Override
  public void execute() {
    System.out.println("Executing in ConcreteStrategyB");
  }
}

class StrategyContext {
  private Strategy strategy;

  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  public void execute() {
    strategy.execute();
  }
}

public class StrategyPattern {
```

```
  public static void main(String[] args) {
      StrategyContext context = new StrategyContext();
    context.setStrategy(new ConcreteStrategyA());
    context.execute();
    context.setStrategy(new ConcreteStrategyB());
    context.execute();
  }
}
```

In this example, the Strategy interface defines the interface for the behavior, and the ConcreteStrategyA and ConcreteStrategyB classes implement the Strategy interface and contain the logic for the behavior. The StrategyContext class is the object that needs to alter its behavior at runtime, and it has a method to set the current strategy and a method to delegate behavior to the current strategy.

To use the Strategy pattern, the client first creates a StrategyContext object and Strategy objects. The client then sets the strategy of the StrategyContext object, and calls the execute method on the StrategyContext. The StrategyContext object delegates the behavior to the current strategy, which performs the behavior.

The Strategy pattern is a useful pattern for allowing an object to have multiple behaviors that can be changed at runtime. It allows the object's behavior to be modified without changing the object's class, and it allows the object to be more flexible and adaptable to changing requirements.

Template method pattern

The Template Method pattern is a design pattern that is used to define the steps of an algorithm in a base class, and to allow subclasses to provide implementation for some or all of the steps. It is a useful pattern to use when it is necessary to provide a basic implementation of an algorithm, but it is desirable to allow subclasses to customize or extend the algorithm.

The Template Method pattern is implemented by creating a base class that defines the template method, which contains the steps of the algorithm, and one or more abstract methods that represent the steps of the algorithm that are left to be implemented by subclasses. Concrete subclasses are then created that provide implementation for the abstract methods.

Here is an example of how the Template Method pattern might be implemented in Java:

```
abstract class AbstractClass {
  public final void templateMethod() {
    step1();
    step2();
    step3();
  }

  protected abstract void step1();
  protected abstract void step2();
  protected void step3() {
    System.out.println("Executing step 3 in AbstractClass");
  }
}

class ConcreteClassA extends AbstractClass {
  @Override
  protected void step1() {
    System.out.println("Executing step 1 in ConcreteClassA");
  }

  @Override
  protected void step2() {
    System.out.println("Executing step 2 in ConcreteClassA");
  }
}

class ConcreteClassB extends AbstractClass {
  @Override
  protected void step1() {
    System.out.println("Executing step 1 in ConcreteClassB");
  }

  @Override
```

```
  protected void step2() {
    System.out.println("Executing step 2 in ConcreteClassB");
  }

  @Override
  protected void step3() {
    System.out.println("Executing step 3 in ConcreteClassB");
  }
}

public class TemplateMethod {
  public static void main(String[] args) {
    AbstractClass a = new ConcreteClassA();
    a.templateMethod();
    AbstractClass b = new ConcreteClassB();
    b.templateMethod();
  }
}
```

In this example, the AbstractClass defines the template method, which contains the steps of the algorithm, and the step1 and step2 methods, which are left to be implemented by subclasses. The ConcreteClassA and ConcreteClassB classes are concrete subclasses that provide implementation for the step1 and step2 methods. The step3 method is implemented in the AbstractClass, but it can be overridden by subclasses if desired.

To use the Template Method pattern, the client creates concrete subclass objects and calls the template method on the objects. The template method in the base class defines the steps of the algorithm, and the abstract methods are implemented by the concrete subclasses. The concrete subclasses can override the implementation of any of the steps if desired.

The Template Method pattern is a useful pattern for defining the steps of an algorithm in a base class, and allowing subclasses to customize or extend the algorithm. It allows the algorithm to be defined in a reusable way, and it allows subclasses to customize or extend the algorithm without changing the base class.

Visitor pattern

The Visitor pattern is a design pattern that is used to separate an algorithm from an object structure on which it operates. It is a useful pattern to use when it is necessary to perform an operation on the elements of an object structure, but it is not desirable or feasible to modify the classes of the elements in the structure.

The Visitor pattern is implemented by creating a visitor interface that defines the interface for the operation, and creating concrete visitor classes that implement the visitor interface and contain the logic for the operation. The object structure is then modified to accept the visitor, and to provide a method for the visitor to visit its elements.

Here is an example of how the Visitor pattern might be implemented in Java:

```
interface Visitor {
  void visit(Element element);
}

interface Element {
  void accept(Visitor visitor);
}

class ConcreteElementA implements Element {
  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

class ConcreteElementB implements Element {
  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

class ConcreteVisitorA implements Visitor {
  @Override
  public void visit(Element element) {
    System.out.println("Visiting " +
element.getClass().getSimpleName() + " with ConcreteVisitorA");
  }
}

class ConcreteVisitorB implements Visitor {
  @Override
  public void visit(Element element) {
```

```
    System.out.println("Visiting " +
element.getClass().getSimpleName() + " with ConcreteVisitorB");
  }
}

public class VisitorPattern {
  public static void main(String[] args) {
    Element elementA = new ConcreteElementA();
    Element elementB = new ConcreteElementB();
    Visitor visitorA = new ConcreteVisitorA();
    Visitor visitorB = new ConcreteVisitorB();
    elementA.accept(visitorA);
    elementA.accept(visitorB);
    elementB.accept(visitorA);
    elementB.accept(visitorB);
  }
}
```

In this example, the Visitor interface defines the interface for the operation, and the ConcreteVisitorA and ConcreteVisitorB classes implement the Visitor interface and contain the logic for the operation. The Element interface is implemented by the ConcreteElementA and ConcreteElementB classes, and the accept method is used to allow the visitor to visit the elements.

To use the Visitor pattern, the client creates Element and Visitor objects, and calls the accept method on the Element objects, passing the Visitor objects as arguments. The Element objects then delegate the operation to the Visitor objects, which perform the operation.

The Visitor pattern is a useful pattern for separating an algorithm from an object structure on which it operates. It allows the algorithm to be defined in a reusable way, and it allows the object structure to be modified without changing the algorithm. It also allows the algorithm to be extended or modified without changing the classes of the elements in the structure.