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:

```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:

```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:

```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:

```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:

```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

Decorator pattern

Facade pattern

Flyweight pattern

Proxy pattern

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