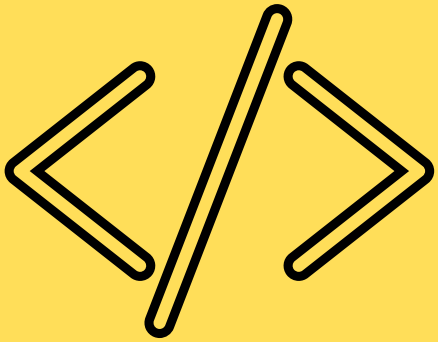

Design Patterns

(With Examples)

- singleton
- prototype
- factory
- session
- observer
- adapter
- decorator
- facade

ABDULHAKIM SIRKO

@abdulhakimsirko

1. Singleton Design Pattern

The singleton pattern is a design pattern that ensures a class has only one instance and provides a global point of access to that instance. The basic idea behind the pattern is to create a single instance of a class, and to provide a global point of access to that instance.

Some key features of the singleton pattern are:

- The singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.
- The singleton pattern uses a private constructor and a static method to ensure that only one instance of the class is created.
- The singleton pattern provides a global point of access to the single instance of the class, typically by using a static property or method.

There are a number of benefits to using the singleton pattern:

- It ensures that a class has only one instance and provides a global point of access to that instance.
- It can be used to control access to resources that are shared across the system, such as a database connection or a configuration object.
- It can be used to implement caching, logging, and other system-wide services.

Example:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // private constructor to prevent instantiation outside this 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 it from being instantiated from outside the class. The ***getInstance()*** method is a public static method that returns a single instance of the ***Singleton*** class. The ***instance*** variable is declared as private and static so that it can only be accessed through the ***getInstance()*** method.

The ***getInstance()*** method checks if the ***instance*** variable is null, and if it is, it creates a new instance of the ***Singleton*** class. If the ***instance*** variable is not null, it simply returns the existing instance.

This way, there will always be only one instance of the ***Singleton*** class, and it can be accessed from anywhere in the code by calling the ***getInstance()*** method.

2. Prototype Design Pattern

The prototype pattern is a design pattern that allows you to create new objects by copying an existing object, rather than creating new objects by using a class's constructor. The basic idea behind the pattern is that you create a prototype of an object, and then use that prototype to create new instances of the object.

Some key features of the prototype pattern:

- The prototype pattern allows you to create new objects by copying an existing object, rather than creating new objects using a class's constructor.
- The prototype object acts as a blueprint for creating new objects.
- The new objects are created by copying the prototype.

There are a number of benefits to using the prototype pattern:

- It allows you to create new objects more quickly and easily than if you were to use a class's constructor.
- It allows you to create new objects that are copies of existing objects, rather than creating new objects that are completely independent.

- It allows you to create new objects that have the same state as the prototype but can be modified independently.

The prototype pattern can be implemented in many ways, but one common way is to create an interface called **Clonable** or **Prototype** that defines a method for creating a copy of the object. Classes that want to support being copied can implement this interface and provide an implementation for the method.

Example:

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

class Person implements Clonable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(Person other) {
        this(other.name, other.age);
    }

    public Clonable clone() {
        return new Person(this);
    }
}
```

In this example, the ***Person*** class implements the ***Clonable*** interface, and provides an implementation of the ***clone()*** method that creates a new ***Person*** object that is a copy of the original object. The prototype pattern can be useful in a variety of situations, some common use cases are:

- When a class's constructor requires a lot of arguments or is otherwise complex to use. By using a prototype, you can create new objects by copying an existing object, which can be simpler than calling a constructor with many arguments.

- When a class is part of a large object hierarchy, and the class you are interested in is deep down in the hierarchy. By using a prototype, you can create a new instance of the class by copying an existing instance, rather than having to navigate through the object hierarchy to create a new instance.
- When you want to create a new object that is similar to an existing object, but with some small modifications. You can use the prototype pattern to create a copy of the existing object, and then make the necessary modifications to the copy.
- When you want to create new objects that are identical to existing objects, but are independent of the existing objects. In other words, changes made to the new objects do not affect the existing objects, and vice versa.
- When you want to create a lot of similar objects quickly, the prototype pattern can be an efficient way to create many new objects that are identical or similar to an existing object, without having to go through the expense of creating them from scratch using a constructor or factory method.
- When you have a complex and mutable object and you don't want to expose the actual implementation to client or exposing it would be too risky. The prototype pattern allows you to give the client a copy of the object, so he can work with it without changing the original one.

3. Factory Design Pattern

The factory pattern is a design pattern that provides a way to create objects without specifying the exact class of object that will be created. The basic idea behind the pattern is that you define a factory class that has a method for creating objects, but the factory class does not know or care about the specific class of object that will be created. Instead, it relies on a set of rules or a configuration to determine what class of object to create.

Some key features of the factory pattern:

- The factory pattern provides a way to create objects without specifying the exact class of object that will be created.

- The factory class has a method for creating objects, but the factory class does not know or care about the specific class of object that will be created.
- The factory class relies on a set of rules or a configuration to determine what class of object to create.

There are several types of factory pattern, but the most common one is the "Factory Method" pattern which defines an interface for creating an object, but let subclasses to alter the class of objects that will be created.

There are a number of benefits to using the factory pattern:

- It allows you to create objects without specifying the exact class of object that will be created, which can make your code more flexible and easier to change.
- It allows you to encapsulate the process of creating objects, making it easier to change the way objects are created in the future, without affecting the code that uses the factory.
- It can make your code more reusable, as you can use the factory to create objects in a variety of contexts.

An example of factory pattern could be a class that creates different types of vehicles like cars, trucks, and buses.

In this example, the **VehicleFactory** class is the factory class, it has a method called **getVehicle()** that creates objects of different types of vehicle based on the type passed in as an argument. The **IVehicle** interface is the factory method, it defines the interface for creating the objects and the concrete classes **Car**, **Truck**, **Bus** are the objects that implement the factory method.

```java
interface IVehicle {
    void drive();
}

class Car implements IVehicle {
    public void drive() {
        System.out.println("Driving a Car.");
    }
}

class Truck implements IVehicle {
    public void drive() {
        System.out.println("Driving a Truck.");
    }
}

class Bus implements IVehicle {
    public void drive() {
        System.out.println("Driving a Bus.");
    }
}

class VehicleFactory {
    public IVehicle getVehicle(String vehicleType) {
        if (vehicleType.equalsIgnoreCase("car")) {
            return new Car();
        } else if (vehicleType.equalsIgnoreCase("truck")) {
            return new Truck();
        } else if (vehicleType.equalsIgnoreCase("bus")) {
            return new Bus();
        } else {
            return null;
        }
    }
}
```

This way, the code that uses the factory doesn't need to know or care about the specific classes of objects that are being created, as long as they implement the **IVehicle** interface.

4. Session design pattern

A "session" in software development refers to a period of time in which a user interacts with a software application. The session design pattern is a way of managing the state of a user's session within an application, typically through the use of a session object.

The session object is responsible for keeping track of information about a user's session, such as their authentication status, their preferences, the items in their shopping cart, etc. The session object can be stored in memory, on the client side using cookies, or on the server side using a session ID.

The session design pattern provides a way to store and retrieve session data, and to keep track of a user's session throughout the lifetime of an application. It is typically used in web applications, where a user's session state needs to be maintained across multiple HTTP requests.

Some key features of the session design pattern:

- The session design pattern provides a way to store and retrieve session data, and to keep track of a user's session throughout the lifetime of an application.
- The session object is responsible for managing the state of a user's session within an application, and can be stored in memory, on the client side using cookies, or on the server side using a session ID.
- The session object can be used to store information about a user's authentication status, their preferences, the items in their shopping cart, etc.

An example of session design pattern implementation in a web application could be a login session:

In this example, the **Session** class is implemented as a singleton, it has methods for logging in, logging out, checking authentication status and getting the user.

The class implements a private constructor, which ensures that only one instance of the class can be created, and a static method for retrieving the single instance.

This way, the session object can be accessed from any part of the application, in order to check the user's authentication status, retrieve their user object and maintain their session throughout the application.

```
class Session {
    private static final Session instance = new Session();
    private boolean isAuthenticated;
    private User user;

    private Session() {}

    public static Session getInstance() {
        return instance;
    }
    public void login(User user) {
        this.isAuthenticated = true;
        this.user = user;
    }
    public void logout() {
        this.isAuthenticated = false;
        this.user = null;
    }
    public boolean isAuthenticated() {
        return this.isAuthenticated;
    }
    public User getUser() {
        return this.user;
    }
}
```

5. Session design pattern

The observer pattern is a design pattern that allows objects to be notified of changes in the state of other objects, without the objects being tightly coupled to one another. The basic idea behind the pattern is to create a mechanism for objects to "observe" other objects and to be notified when the state of the observed objects changes.

Some key features of the observer pattern:

- The observer pattern allows objects to be notified of changes in the state of other objects, without the objects being tightly coupled to one another.
- The observer pattern is based on the publish-subscribe model, where objects "publish" notifications of changes in their state, and other objects "subscribe" to receive those notifications.

- The observer pattern uses interfaces to define the "publish" and "subscribe" methods, allowing different classes to implement them in different ways.

There are a number of benefits to using the observer pattern:

- It allows objects to be notified of changes in the state of other objects without being tightly coupled to them.
- It promotes loose coupling between objects, making it easier to change the implementation of one object without affecting the other objects that use it.
- It allows objects to be notified of changes in the state of other objects even if they don't know the other objects' classes.

An example of observer pattern could be an application that notifies the user when the stock price of a specific company has changed.

```
interface Observer {
    void update(double price);
}

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

class StockPriceObserver implements Observer {
    private String name;
    private double price;
    private Subject stock;

    public StockPriceObserver(String name, Subject stock) {
        this.name = name;
        this.stock = stock;
        this.stock.registerObserver(this);
    }

    public void update(double price) {
        this.price = price;
        display();
    }
```

```
    public void display() {
        System.out.println(name + " - Current Price : $" + price);
    }
}

class StockPriceSubject implements Subject {
    private ArrayList<Observer> observers;
    private double price;

    public StockPriceSubject() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(price);
        }
    }

    public void setPrice(double price) {
        this.price = price;
        notifyObservers();
    }
}
```

This example shows how the Observer pattern can be implemented to keep track of the stock prices of a company.

There are two main classes: **StockPriceObserver** and **StockPriceSubject**, which represent the observer and the subject, respectively.

The **StockPriceObserver** registers with the **StockPriceSubject** as an observer, and it is notified whenever there is a change in the stock price.

The **StockPriceSubject** maintains a list of observers and notifies them whenever there is a change in the stock price.

The example defines an interface **Observer** with a single method **update()**, and an interface **Subject** with three methods **registerObserver()**, **removeObserver()**, and **notifyObservers()**.

The **StockPriceObserver** class implements the **Observer** interface, while the **StockPriceSubject** class implements the **Subject** interface.

Whenever the price of a stock changes, the **setPrice()** method is called on the **StockPriceSubject** object, which updates the price and then notifies all the registered observers by calling their **update()** method.

The **StockPriceObserver** objects, in turn, update their price and call their **display()** method to show the current stock price.

6. Adapter design pattern

The adapter pattern is a design pattern that allows incompatible classes to work together by converting the interface of one class into an interface that another class expects. The basic idea behind the pattern is to create an "adapter" class that can adapt the interface of an existing class to the interface that is expected by other classes.

Some key features of the adapter pattern:

- The adapter pattern allows incompatible classes to work together by converting the interface of one class into an interface that another class expects.
- The adapter pattern uses an "adapter" class that can adapt the interface of an existing class to the interface that is expected by other classes.
- The adapter pattern can be used to create a link between classes that couldn't otherwise be linked due to incompatible interfaces.

There are a number of benefits to using the adapter pattern:

- It allows incompatible classes to work together by adapting one class's interface to the interface that another class expects.

- It promotes loose coupling between classes, making it easier to change the implementation of one class without affecting the classes that use it.
- It allows classes to work together that couldn't otherwise due to incompatible interfaces.

Example:

An example of adapter pattern could be a situation where a class called **Square** wants to reuse another class **Rectangle** but the problem is that **Square** expects to have a method **calculateArea** and **Rectangle** has the methods **calculateArea(length: double, width:double)**.

```
interface Shape {
    double calculateArea();
}

class Square {
    double side;

    public Square(double side) {
        this.side = side;
    }

    public double calculateArea() {
        return side*side;
    }
}

class Rectangle {
    double length;
    double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double calculateArea(double length, double width) {
        return length * width;
    }
}
```

In this example, the **Rectangle** class has a method **calculateArea(length: double, width:double)** but **Square** class expects to have a method **calculateArea()**.

```
class RectangleAdapter implements Shape {
    private Rectangle rectangle;

    public RectangleAdapter(Rectangle rectangle) {
        this.rectangle = rectangle;
    }

    @Override
    public double calculateArea() {
        return rectangle.calculateArea(rectangle.length, rectangle.width);
    }
}
```

The **RectangleAdapter** class implements the **Shape** interface and adapt the **Rectangle** class to the expected interface by square class, by implementing **calculateArea()** that internally calls **calculateArea(length: double, width:double)**.

This way the Square class can use the rectangle class through the Adapter class and both classes can work together.

7. Decorator design pattern

The decorator pattern is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. The basic idea behind the pattern is to create a "decorator" class that can be used to add or override behavior for an existing class.

Some key features of the decorator pattern:

- The decorator pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.
- The decorator pattern uses a "decorator" class that can be used to add or override behavior for an existing class.
- The decorator pattern can be used to add new behavior to an existing class without the need to create a new subclass.

There are a number of benefits to using the decorator pattern:

- It allows behavior to be added to an individual object without affecting the behavior of other objects from the same class.
- It promotes code reuse by allowing behavior to be added to an existing class without the need to create a new subclass.
- It allows behavior to be added or overridden dynamically at runtime.

Example:

Let's say we have an interface called **Coffee** that defines the basic behavior of a coffee object:

```
public interface Coffee {
    double getCost();  // returns the cost of the coffee
    String getDescription();  // returns a description of the coffee
}
```

We also have a concrete implementation of this interface called **SimpleCoffee**:

```
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1.0;
    }

    @Override
    public String getDescription() {
        return "Simple coffee";
    }
}
```

Now, let's say we want to create some decorator classes that will add extra functionality to a **Coffee** object. For example, we could create a **Milk** decorator that adds milk to the coffee:

```java
public class Milk implements Coffee {
    private final Coffee coffee;

    public Milk(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.5;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", milk";
    }
}
```

In this example, the **Milk** decorator takes a **Coffee** object in its constructor and stores it in an instance variable. The **getCost()** method returns the cost of the **Coffee** object passed in, plus the cost of the milk (0.5). The **getDescription()** method returns the description of the **Coffee** object passed in, plus the string "milk".

8. Facade design pattern

The facade pattern is a design pattern that provides a simplified interface to a complex system. The idea behind the facade pattern is to create a single, simplified interface that hides the complexity of the underlying system and makes it easier to use. The facade defines a set of methods that present a simplified and easy-to-understand view of the functionality provided by the underlying system.

The facade pattern is often used in situations where a system is made up of many different components, each with its own interface and set of responsibilities. By using a facade, you can create a single, unified interface that makes it easier for other parts of the system to interact with the system as a whole.

Here are some key features of the facade pattern:

- The facade defines a simplified interface that can be used to access the functionality provided by the underlying system.
- The facade hides the complexity of the underlying system and makes it easier to use.
- The facade can be used to decouple the other parts of the system from the details of the underlying system.

There are a number of benefits to using the facade pattern:

- It makes the system easier to use, as the simplified interface is easier to understand than the complexity of the underlying system.
- It makes the system more flexible, as changes to the underlying system can be hidden behind the facade.
- It makes the system more maintainable, as changes to the underlying system do not affect the other parts of the system that use the facade.

The facade pattern is a commonly used pattern in many different programming languages and environments. It can be implemented in many ways, and the specific implementation details depend on the programming language and the system being built.

Example:

Let's say we have a complex subsystem consisting of multiple classes that need to be used together to accomplish a task. Instead of exposing the complexity of the subsystem to the client code, we can create a facade that provides a simple interface for the client to interact with the subsystem.

```
public class SubsystemClassA {
    public void methodA() {
        System.out.println("SubsystemClassA: methodA");
    }
}
```

```
public class SubsystemClassB {
    public void methodB() {
        System.out.println("SubsystemClassB: methodB");
    }
}

public class SubsystemClassC {
    public void methodC() {
        System.out.println("SubsystemClassC: methodC");
    }
}

public class Facade {
    private SubsystemClassA subsystemA;
    private SubsystemClassB subsystemB;
    private SubsystemClassC subsystemC;

    public Facade() {
        subsystemA = new SubsystemClassA();
        subsystemB = new SubsystemClassB();
        subsystemC = new SubsystemClassC();
    }

    public void doTask() {
        subsystemA.methodA();
        subsystemB.methodB();
        subsystemC.methodC();
    }
}
```

In this example, we have three subsystem classes, **SubsystemClassA**, **SubsystemClassB**, and **SubsystemClassC**, each with their own set of methods.

We also have a **Facade** class that provides a simple interface for the client code to interact with the subsystem.

The **Facade** class has instances of each of the subsystem classes, and a single public method called **doTask()** that calls the necessary methods of each subsystem class in the correct order to accomplish the task.

By using the facade pattern, the complexity of the subsystem is hidden from the client code, which only needs to interact with the simple **doTask()** method of the **Facade** class.