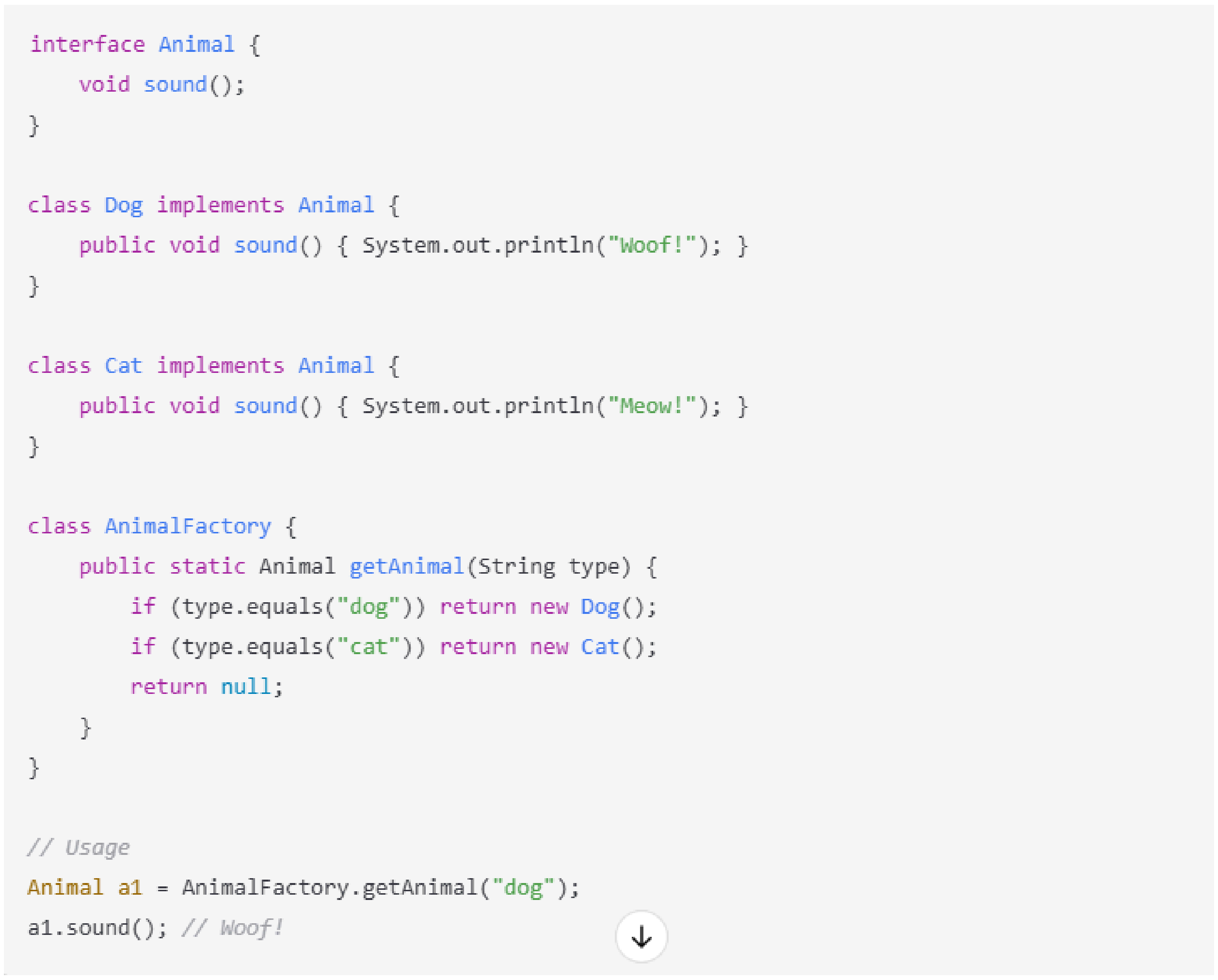
Design patterns are proven, reusable solutions to common problems in software design. They help make code more flexible, reusable, and maintainable.

Creational Patterns – Focus on object creation

✅ **Factory Pattern – Creates objects based on input, without exposing the creation logic.**

Example:
We have an interface `Animal` with a method `sound()`. Based on input, we create either a `Dog` or `Cat`.



```java
interface Animal {
    void sound();
}

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

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

class AnimalFactory {
    public static Animal getAnimal(String type) {
        if (type.equals("dog")) return new Dog();
        if (type.equals("cat")) return new Cat();
        return null;
    }
}

// Usage
Animal a1 = AnimalFactory.getAnimal("dog");
a1.sound(); // Woof!
```

Singleton Pattern – Ensures a class has only one instance and provides global access to it.

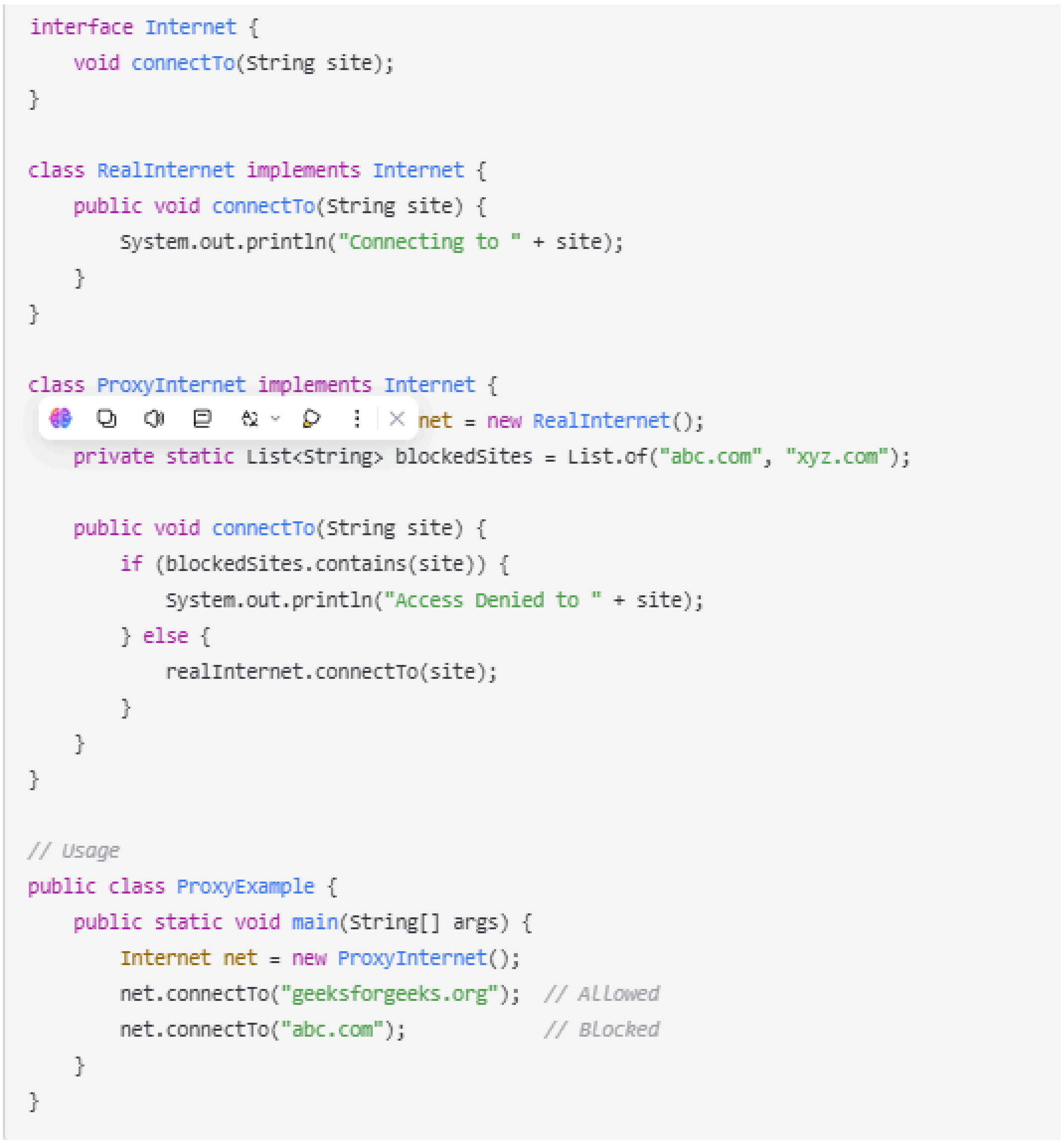
```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}

// Usage
Singleton s1 = Singleton.getInstance();
```

Builder Pattern – Used for step-by-step object construction.

```java
class Pizza {
    private String base, topping, size;

    private Pizza(Builder builder) {
        this.base = builder.base;
        this.topping = builder.topping;
        this.size = builder.size;
    }

    public void showPizza() {
        System.out.println(size + " " + base + " pizza with " + topping);
    }

    public static class Builder {
        private String base, topping, size;

        public Builder setBase(String base) { this.base = base; return this; }
        public Builder setTopping(String topping) { this.topping = topping; return this; }
        public Builder setSize(String size) { this.size = size; return this; }

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

// Usage
Pizza pizza = new Pizza.Builder()
                    .setBase("Thin Crust")
                    .setTopping("Paneer")
                    .setSize("Large")
                    .build();
pizza.showPizza(); // Large Thin Crust pizza with Paneer
```

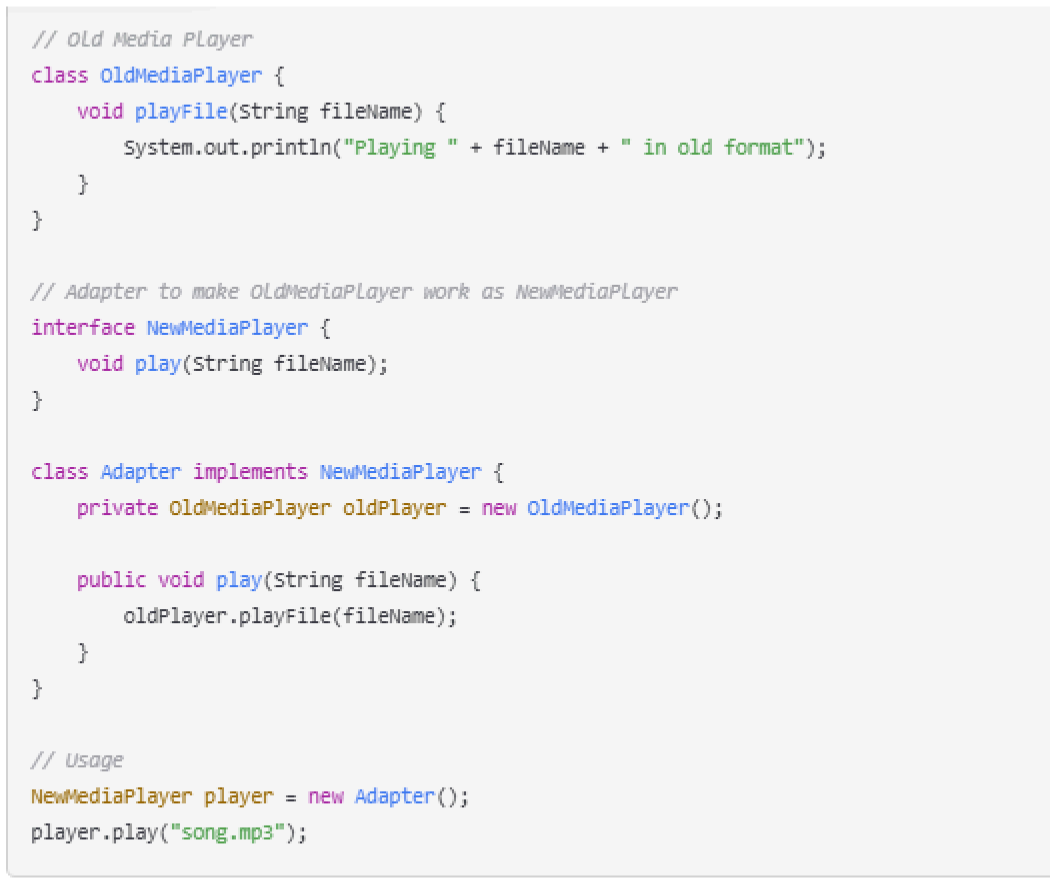
Structural Patterns – Focus on organizing classes and objects.

✅ **Proxy Pattern – Controls access to another object (e.g., for security or performance).**



```java
interface Internet {
    void connectTo(String site);
}

class RealInternet implements Internet {
    public void connectTo(String site) {
        System.out.println("Connecting to " + site);
    }
}

class ProxyInternet implements Internet {
    [illegible]net = new RealInternet();
    private static List<String> blockedSites = List.of("abc.com", "xyz.com");

    public void connectTo(String site) {
        if (blockedSites.contains(site)) {
            System.out.println("Access Denied to " + site);
        } else {
            realInternet.connectTo(site);
        }
    }
}

// Usage
public class ProxyExample {
    public static void main(String[] args) {
        Internet net = new ProxyInternet();
        net.connectTo("geeksforgeeks.org");  // Allowed
        net.connectTo("abc.com");            // Blocked
    }
}
```

Adapter Pattern – Allows incompatible interfaces to work together.

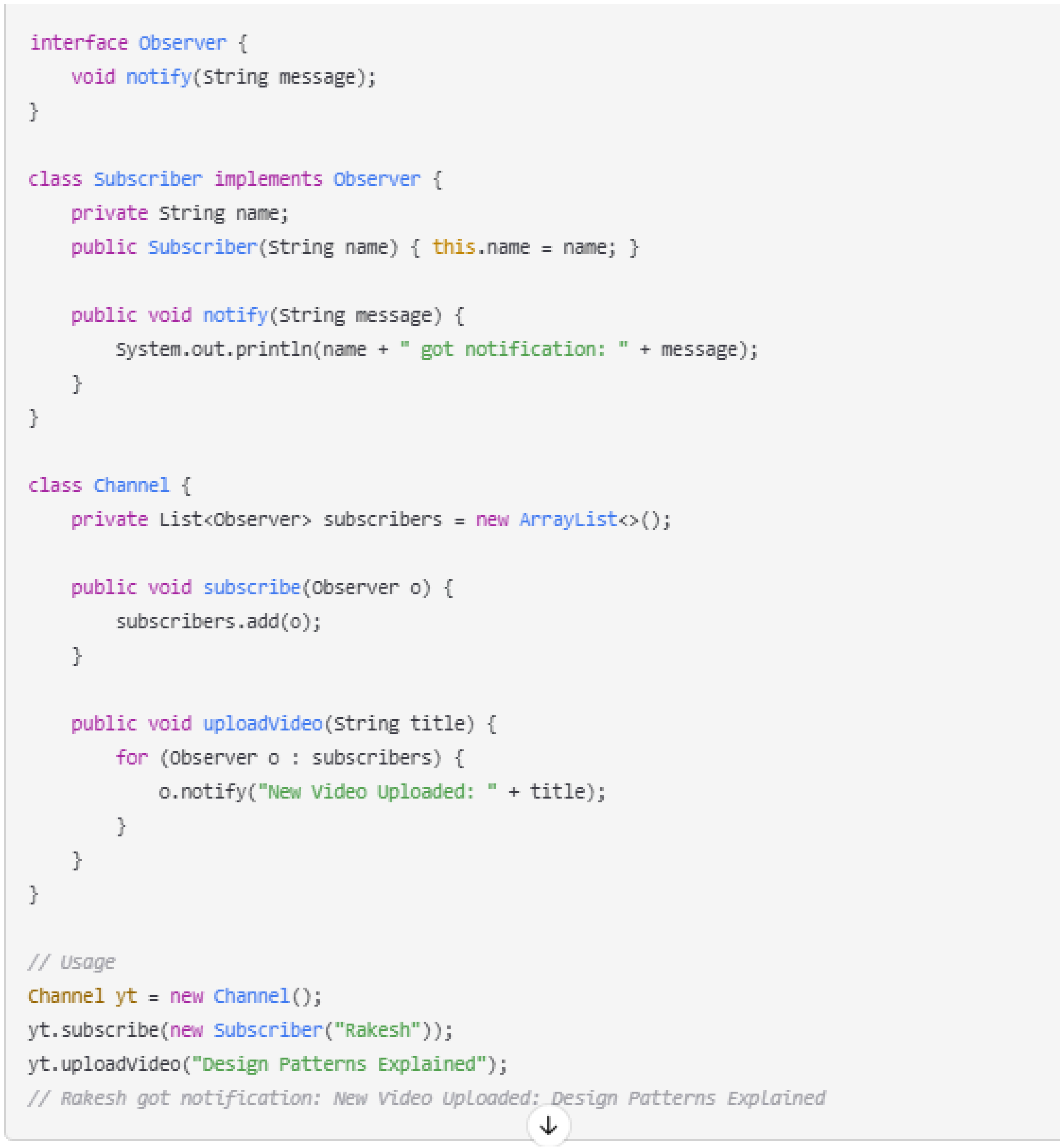


```java
// Old Media Player
class OldMediaPlayer {
    void playFile(String fileName) {
        System.out.println("Playing " + fileName + " in old format");
    }
}

// Adapter to make OldMediaPlayer work as NewMediaPlayer
interface NewMediaPlayer {
    void play(String fileName);
}

class Adapter implements NewMediaPlayer {
    private OldMediaPlayer oldPlayer = new OldMediaPlayer();

    public void play(String fileName) {
        oldPlayer.playFile(fileName);
    }
}

// Usage
NewMediaPlayer player = new Adapter();
player.play("song.mp3");
```

Behavioral Patterns – Focus on communication between objects.

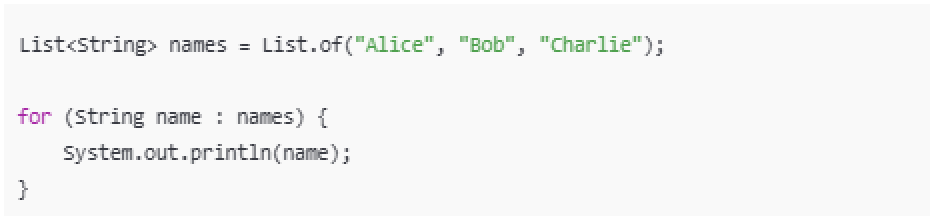
✅ Observer Pattern – Notifies subscribed objects when something changes.

Example: YouTube uploads a new video, subscribers get notified.



```java
interface Observer {
    void notify(String message);
}

class Subscriber implements Observer {
    private String name;
    public Subscriber(String name) { this.name = name; }

    public void notify(String message) {
        System.out.println(name + " got notification: " + message);
    }
}

class Channel {
    private List<Observer> subscribers = new ArrayList<>();

    public void subscribe(Observer o) {
        subscribers.add(o);
    }

    public void uploadVideo(String title) {
        for (Observer o : subscribers) {
            o.notify("New Video Uploaded: " + title);
        }
    }
}

// Usage
Channel yt = new Channel();
yt.subscribe(new Subscriber("Rakesh"));
yt.uploadVideo("Design Patterns Explained");
// Rakesh got notification: New Video Uploaded: Design Patterns Explained
```

Iterator Pattern – Lets you access elements of a collection one by one.



```java
List<String> names = List.of("Alice", "Bob", "Charlie");

for (String name : names) {
    System.out.println(name);
}
```