Week1:

Process Oriented versus Object Oriented

Process Orientation

- Divide programs into structured processes
- Focus on the structure of the processes “procedure”
- Data is scattered as needed.
- Data & operations on data are loosely coupled.

Object-orientation

- Divide programs into units of data
- Focus on the structure of data units (The fundamental unit is Object)
- Spread operations (processes) to associate with data

  (an object has data and behavior)
- Based on models built from real-world concepts.

Objects & Classes

Objects Identity “each object can be uniquely identified andtreated as a distinct entity”

Object Orientation Concepts

- Encapsulation
- inheritance;
- polymorphism;
- abstract classes.

Abstract Class

- An abstract class is a class without instances.

Interfaces versus abstract classes?

Interfaces:

- A class may implement many interfaces.
- An interface only defines public methods properties.
- Used to organize objects that share some common property.

Abstract classes:

- A class has one single superclass.
- An abstract class may also specify private methods and properties.
- Used to group closely related objects.

Week 2&3: change to Rick's application using Encapsulation concept and enumerations, and interface.

Fat Interfaces?

• Fat interface = general purpose interface ≠ client-

specific interface

- can cause bizarre couplings between its clients
- when one client forces a change, all other clients are affected

Break a fat interface into many separate interfaces

- targeted to a single client or a group of clients
- clients depend only on the methods they use (and not on other clients'needs)
- impact of changes to one interface are not as big
- probability of a change reduces
- no interface pollution

1- D: Dependency inversion principle

Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

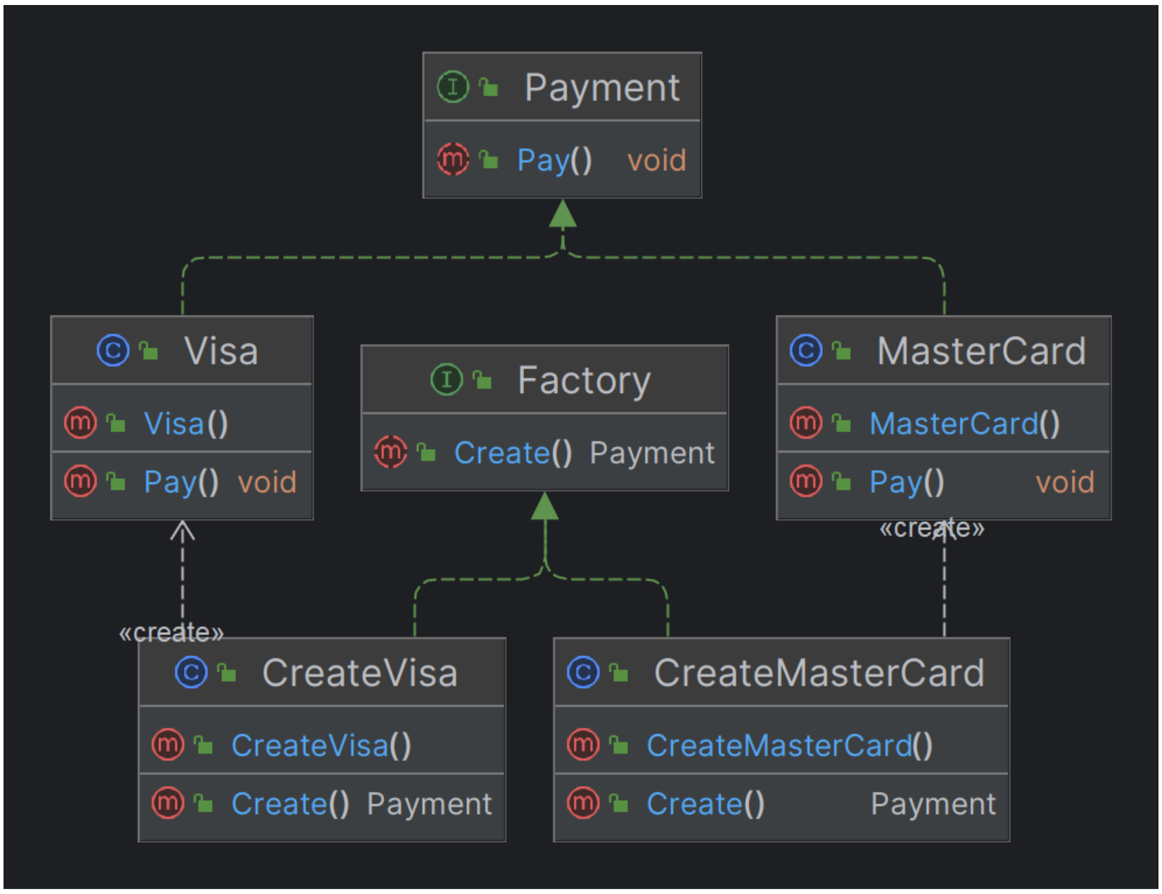
Design Patterns.

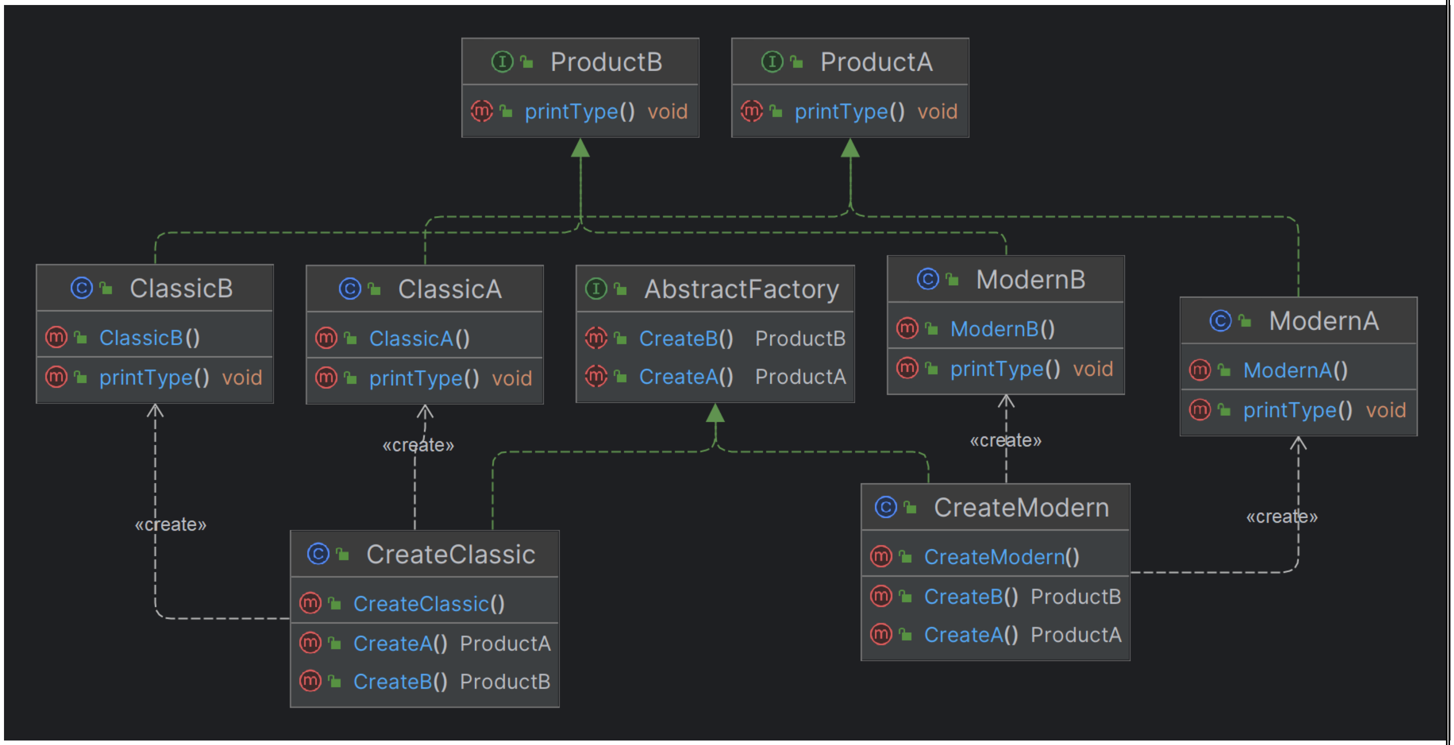
1- The Factory Method Pattern Defined and Abstract Factory (Creational Pattern)

• The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate.

• Factory Method lets a class defer instantiation to subclasses.

• abstract Factory Method defines an interface for creating a family of the same category of objects, but lets subclasses decide which class to instantiate.

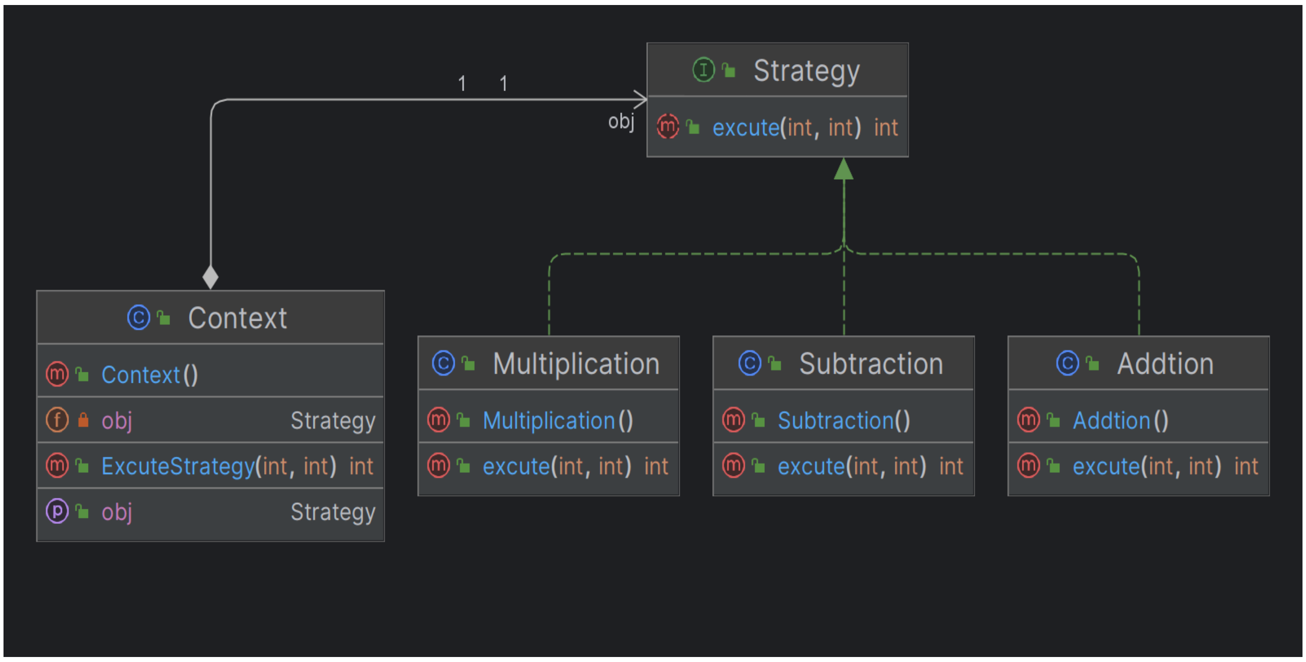




The Strategy Pattern (Behavioural Pattern).

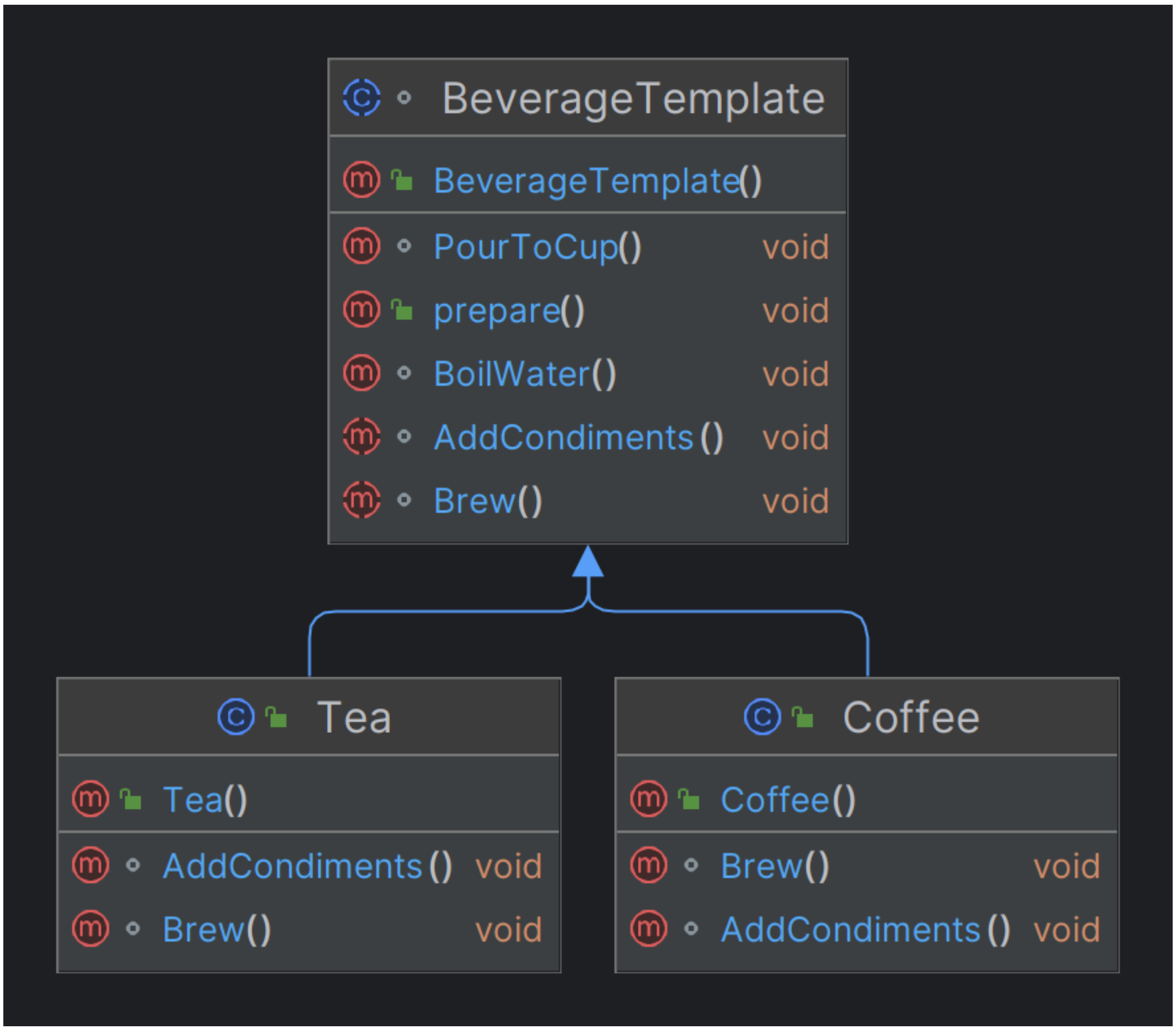
- **Strategy pattern's intent (According to GOF).**

  Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.



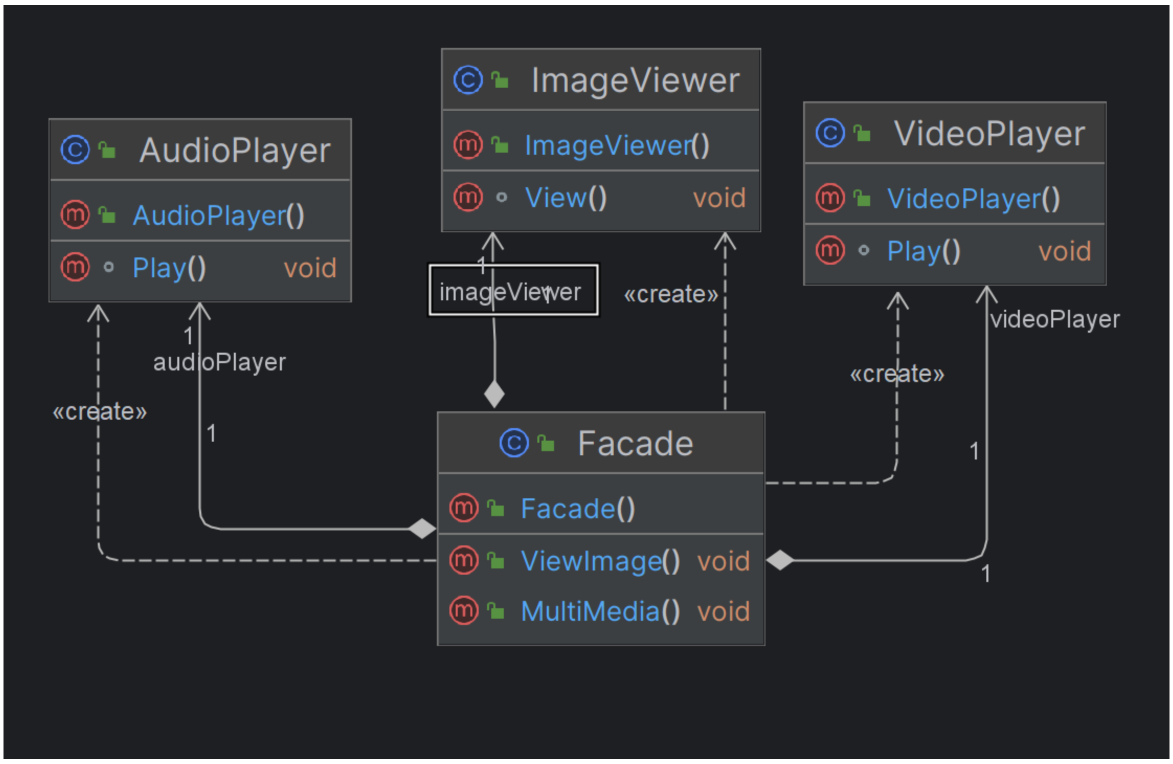
Template Method (Behavioural Pattern).

Define an abstract class that has one algorithm of some steps (concrete,abstract) and some subclasses that implement this abstract methods



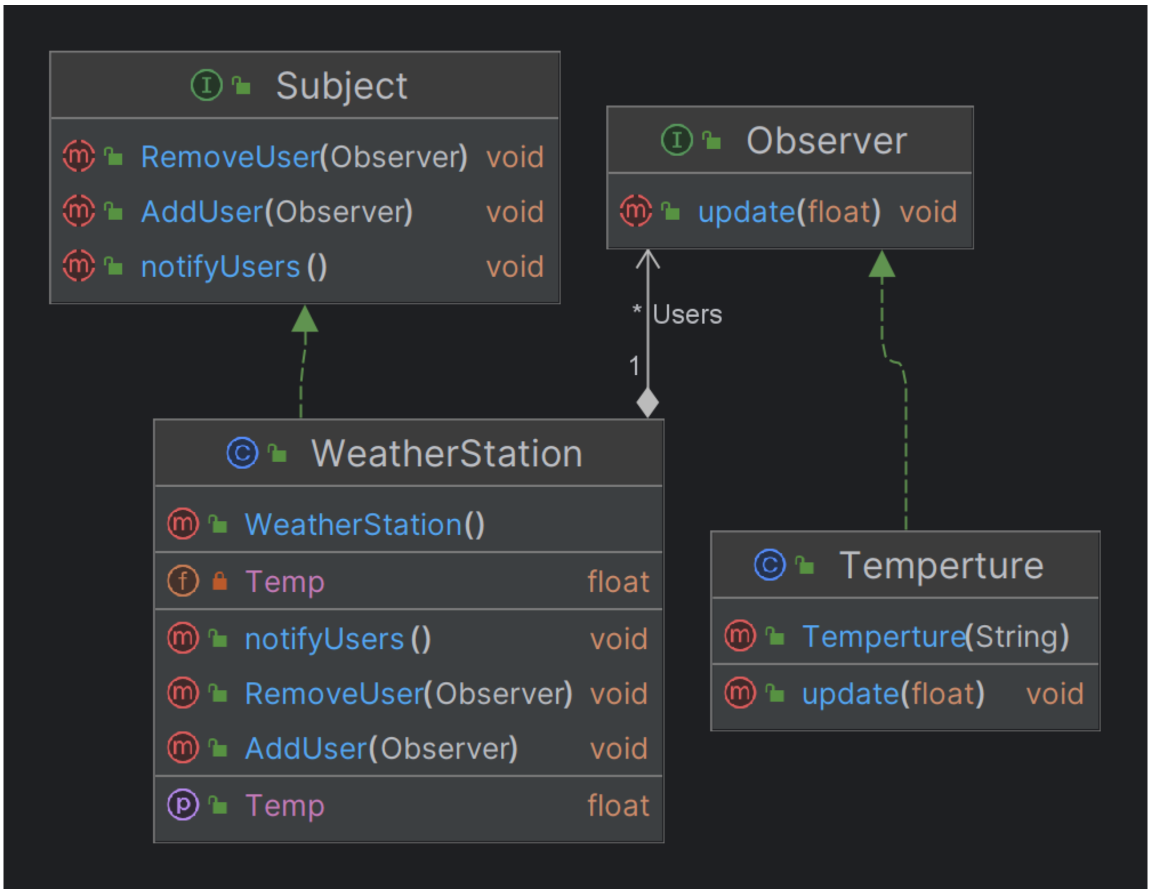
Façade Design Pattern (Structural Pattern).

Define the class that encapsulate some related or independent subsystems and the client deal with this subsystem by object from this class



observer Design Pattern (Behavioural Pattern).

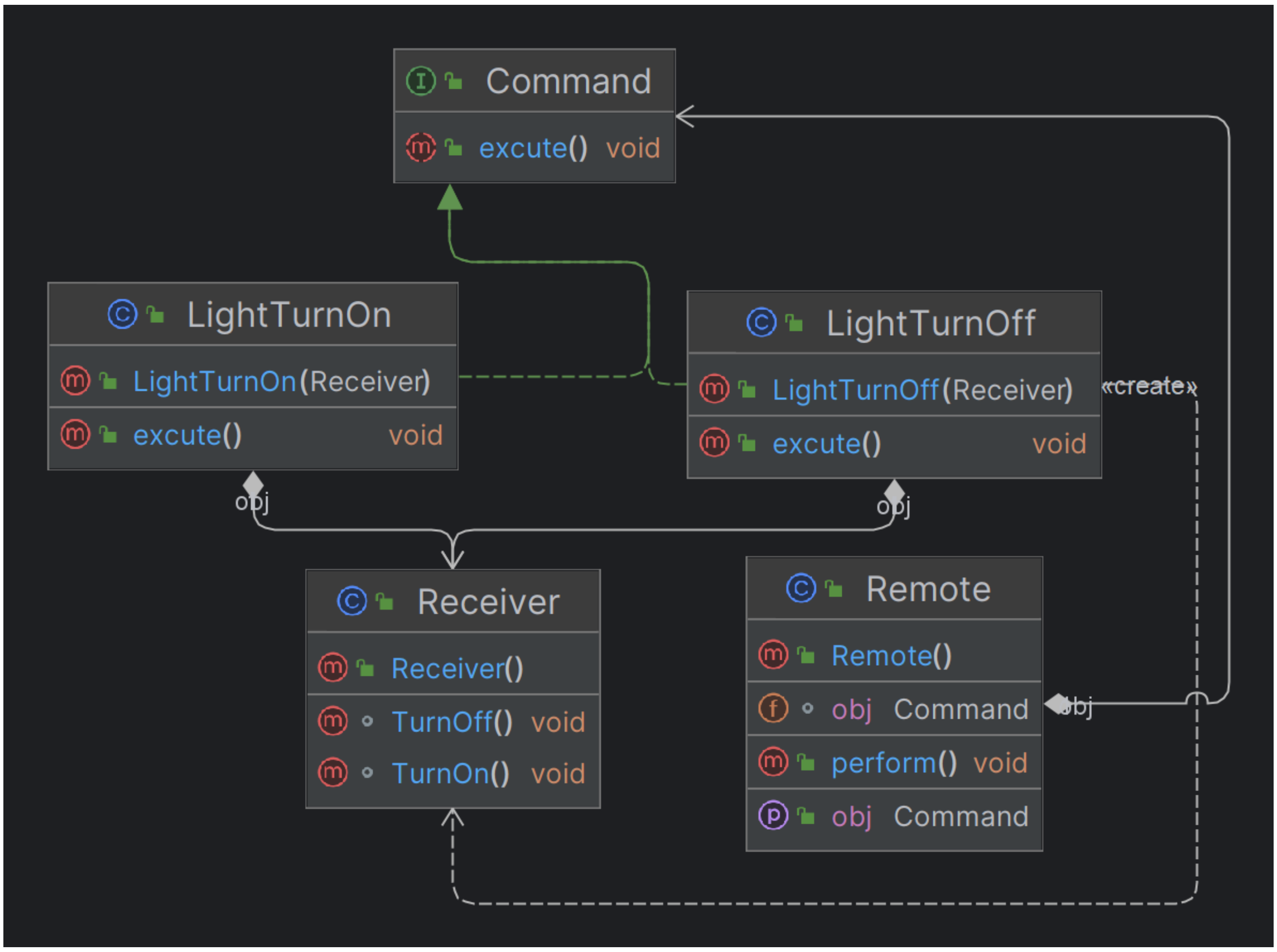
Define an subject (Store) that has array of Observers (Customer) and some operations such that addcustomer, remove customer and notify customer and the class observers that has attribute and update to update the customer with news



Command Design Pattern (Behavioural Pattern).

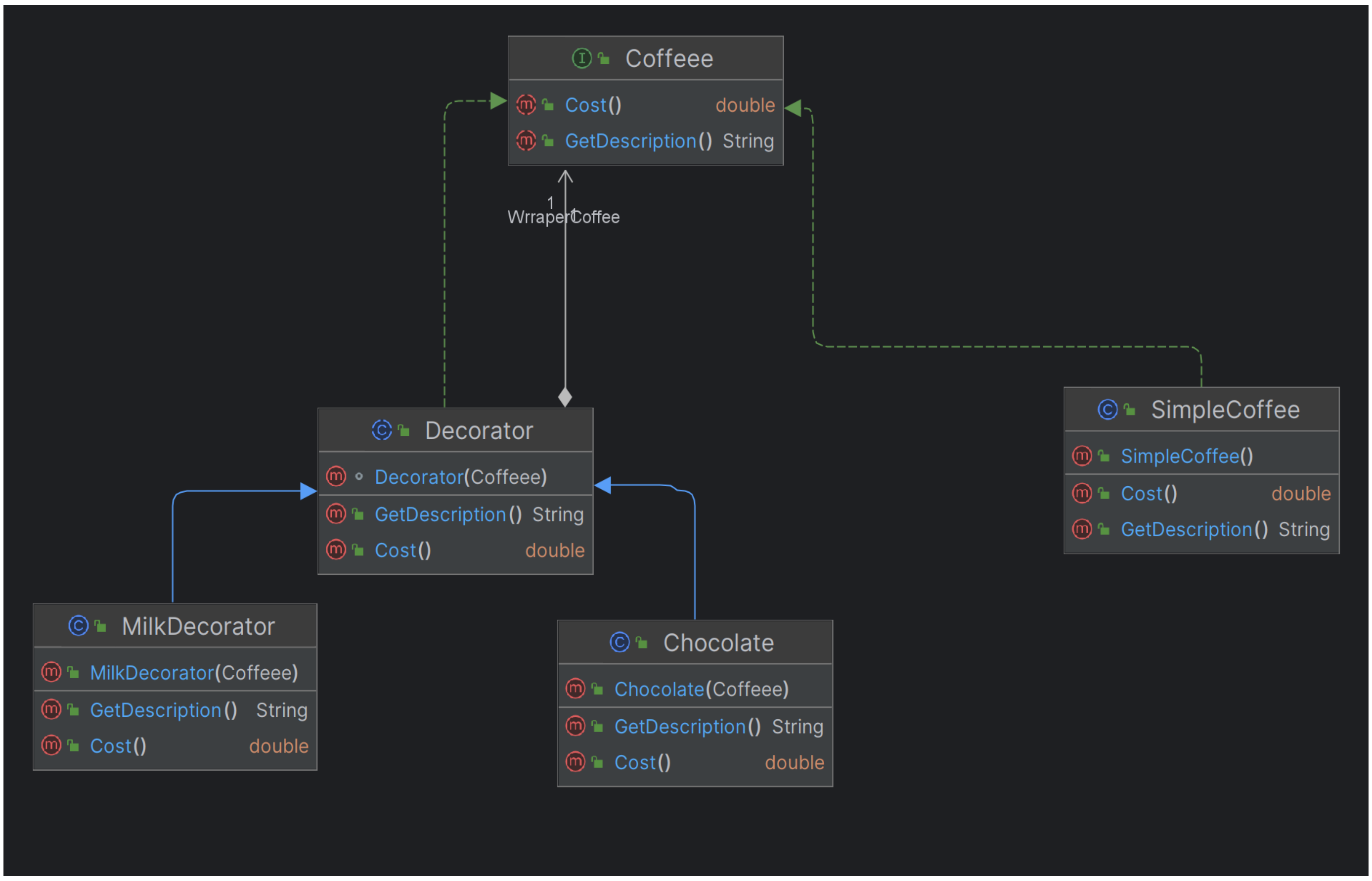
– Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

– A request is turned into a stand-alone object that contains all information about the request



Decorated Design Pattern (Structual Pattern)

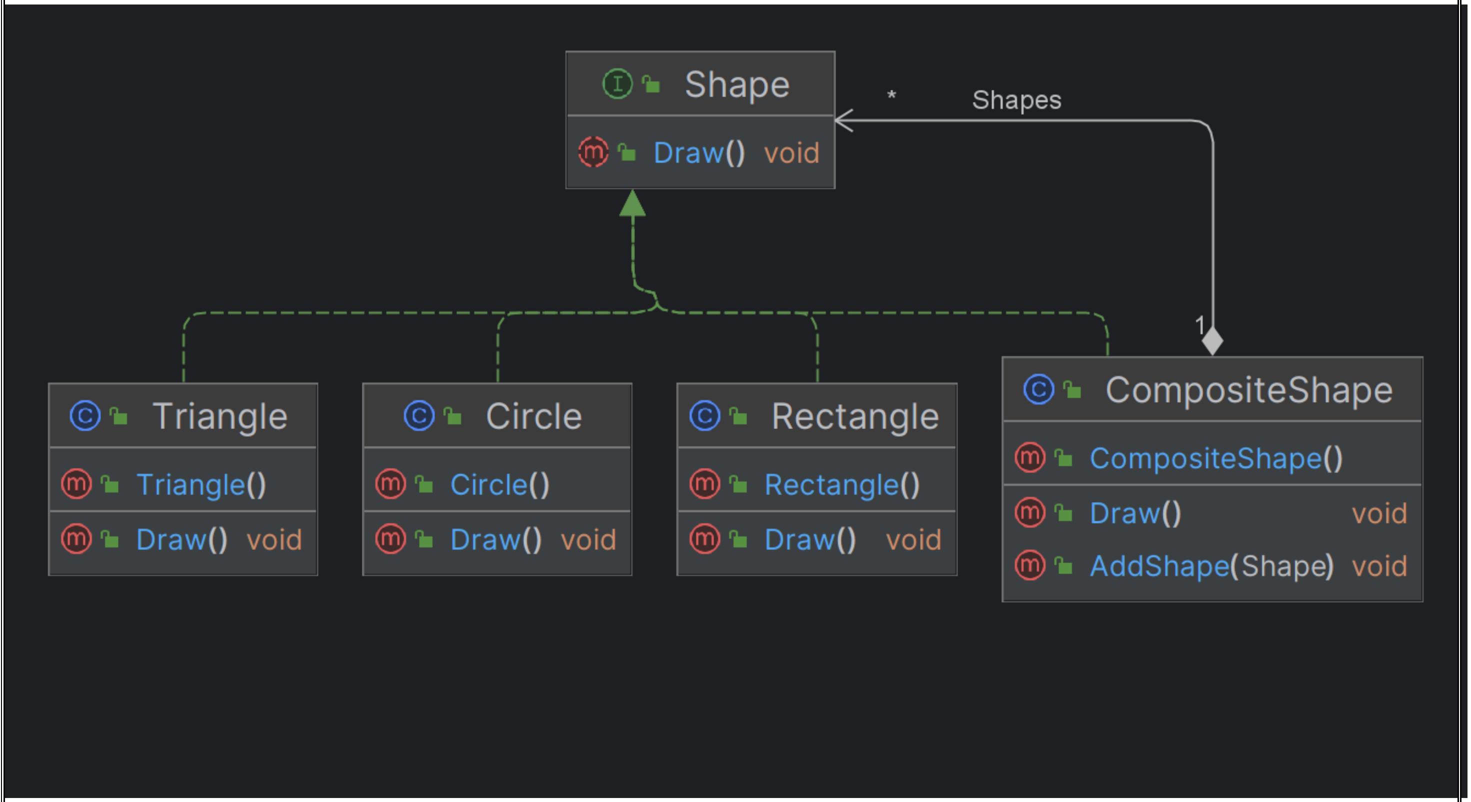
-is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors



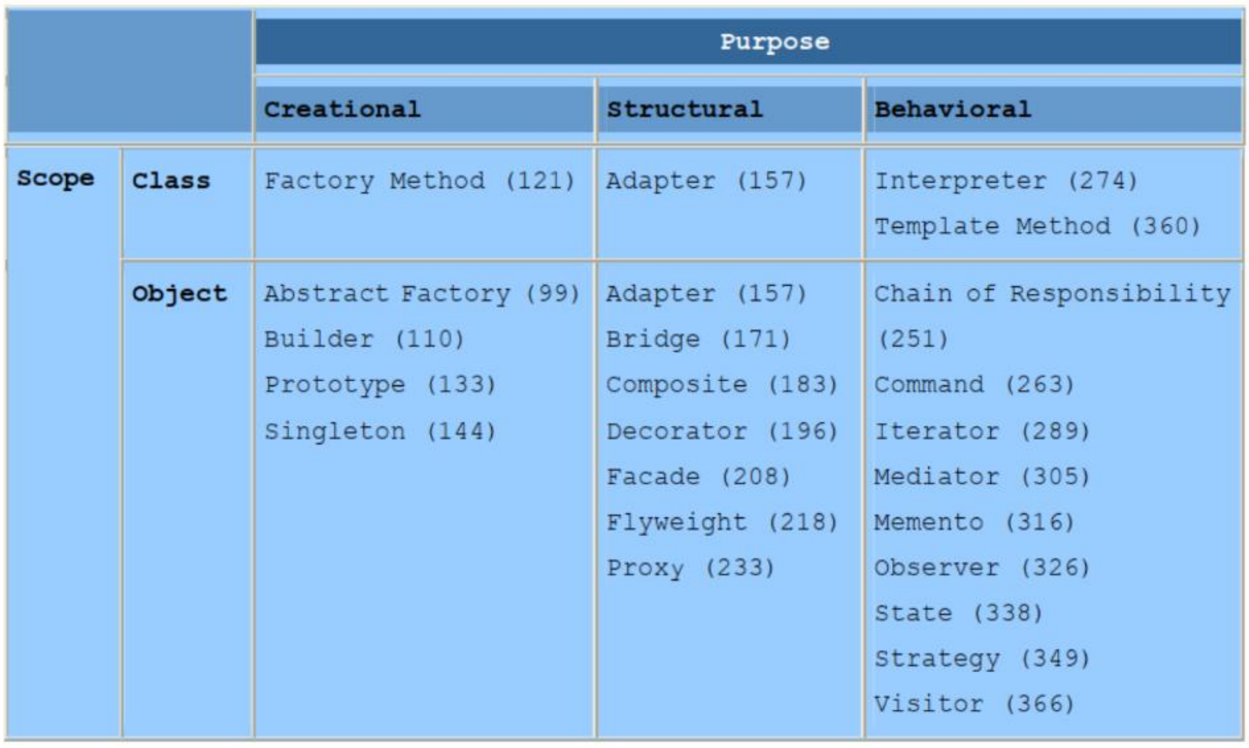
Composite Design Pattern (Structual Pattern)

- Compose objects into tree structures to represent part-whole hierarchies with arbitrary depth and width.
- Treat individual objects and compositions of

objects uniformly



Classification of design Pattern (Important)



		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Object	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

We classify design patterns by two criteria.

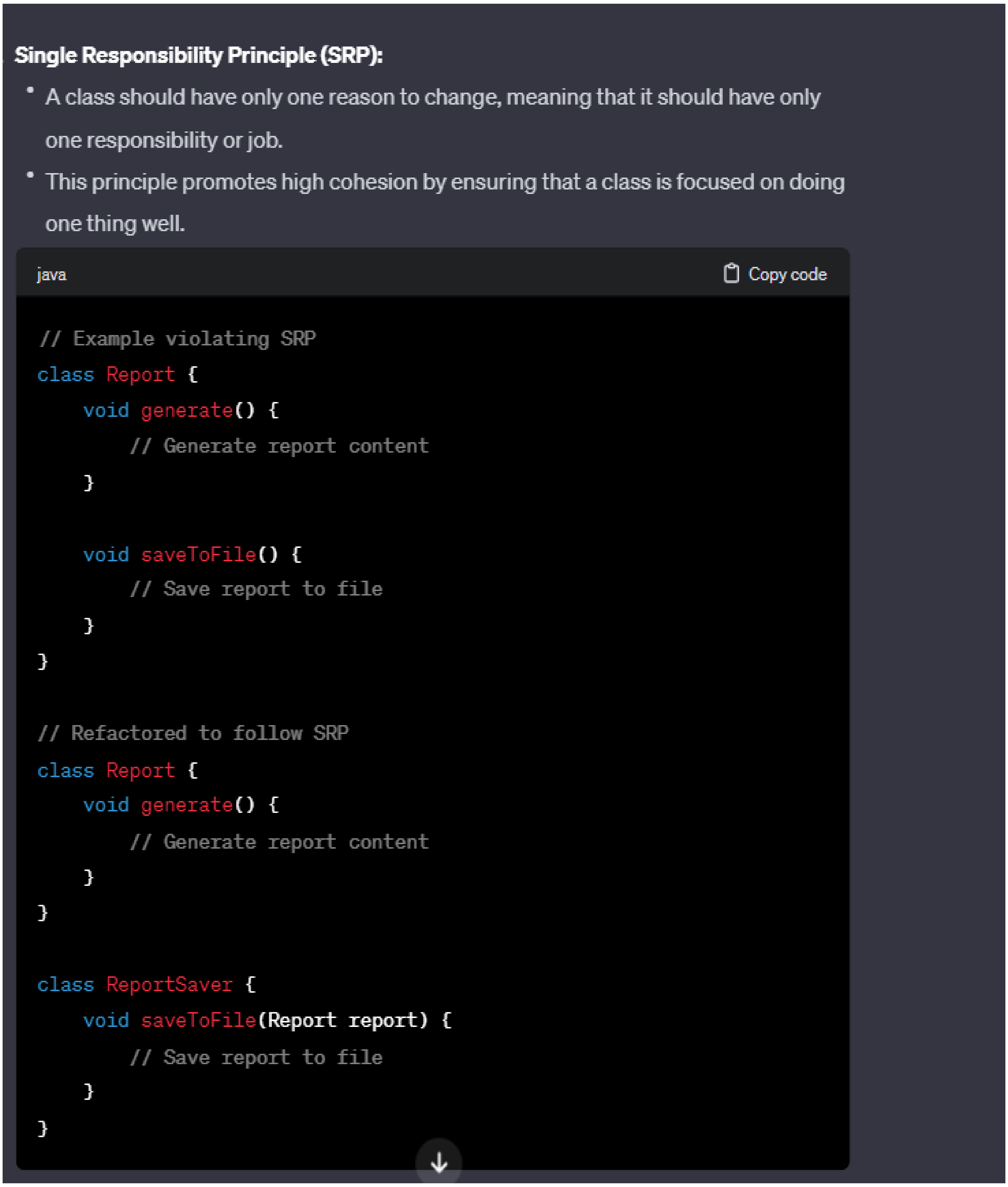
1) **Scope**: specifies whether the pattern applies mainly to classes or objects.

- **Class patterns**: deal with relationships between classes and their subclasses Relationships are established through inheritance, and they are static-fixed at compile time.
- **Object patterns**: deal with object relationships, which can be changed at runtime and are more dynamic.

2) **Purpose:** reflects what a pattern does.

- **Creational**
- **Structural**
- **Behavioural**

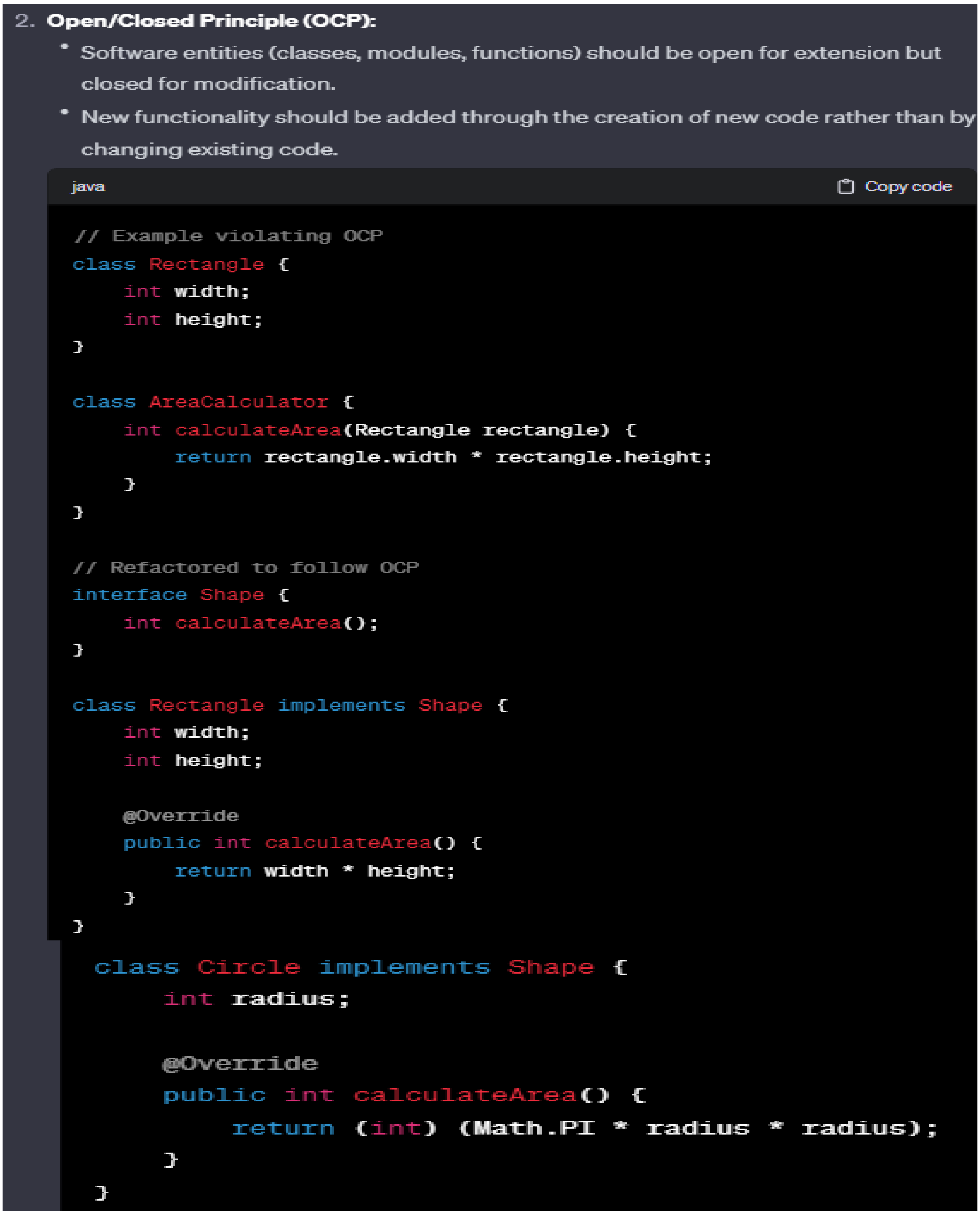
Solid Principles



Single Responsibility Principle (SRP):

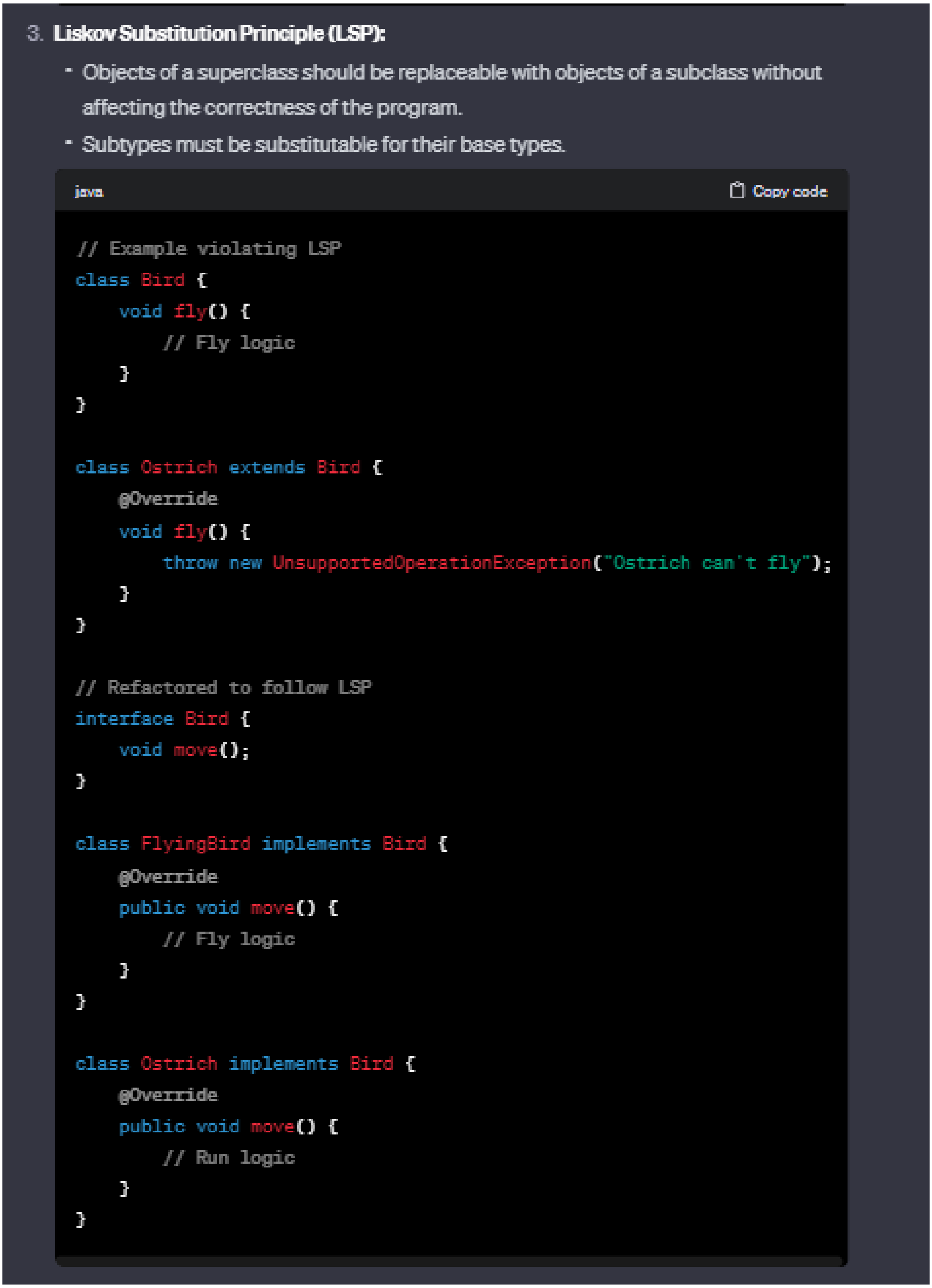
- A class should have only one reason to change, meaning that it should have only one responsibility or job.
- This principle promotes high cohesion by ensuring that a class is focused on doing one thing well.

```java
// Example violating SRP
class Report {
    void generate() {
        // Generate report content
    }

    void saveToFile() {
        // Save report to file
    }
}

// Refactored to follow SRP
class Report {
    void generate() {
        // Generate report content
    }
}

class ReportSaver {
    void saveToFile(Report report) {
        // Save report to file
    }
}
```



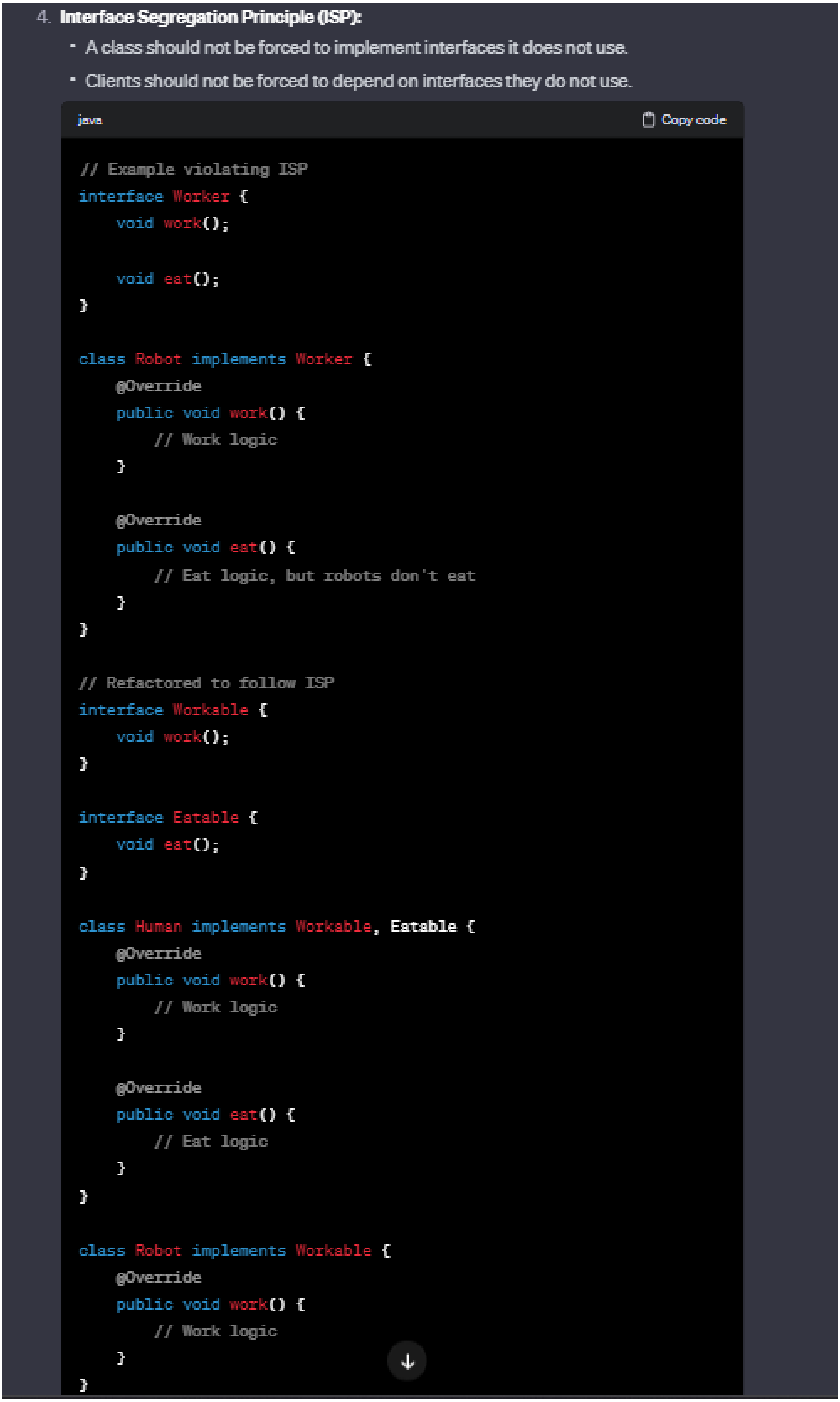
2. **Open/Closed Principle (OCP):**
   - Software entities (classes, modules, functions) should be open for extension but closed for modification.
   - New functionality should be added through the creation of new code rather than by changing existing code.

```java
// Example violating OCP
class Rectangle {
    int width;
    int height;
}

class AreaCalculator {
    int calculateArea(Rectangle rectangle) {
        return rectangle.width * rectangle.height;
    }
}

// Refactored to follow OCP
interface Shape {
    int calculateArea();
}

class Rectangle implements Shape {
    int width;
    int height;

    @Override
    public int calculateArea() {
        return width * height;
    }
}
class Circle implements Shape {
    int radius;

    @Override
    public int calculateArea() {
        return (int) (Math.PI * radius * radius);
    }
}
```



3. **Liskov Substitution Principle (LSP):**
   - Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
   - Subtypes must be substitutable for their base types.

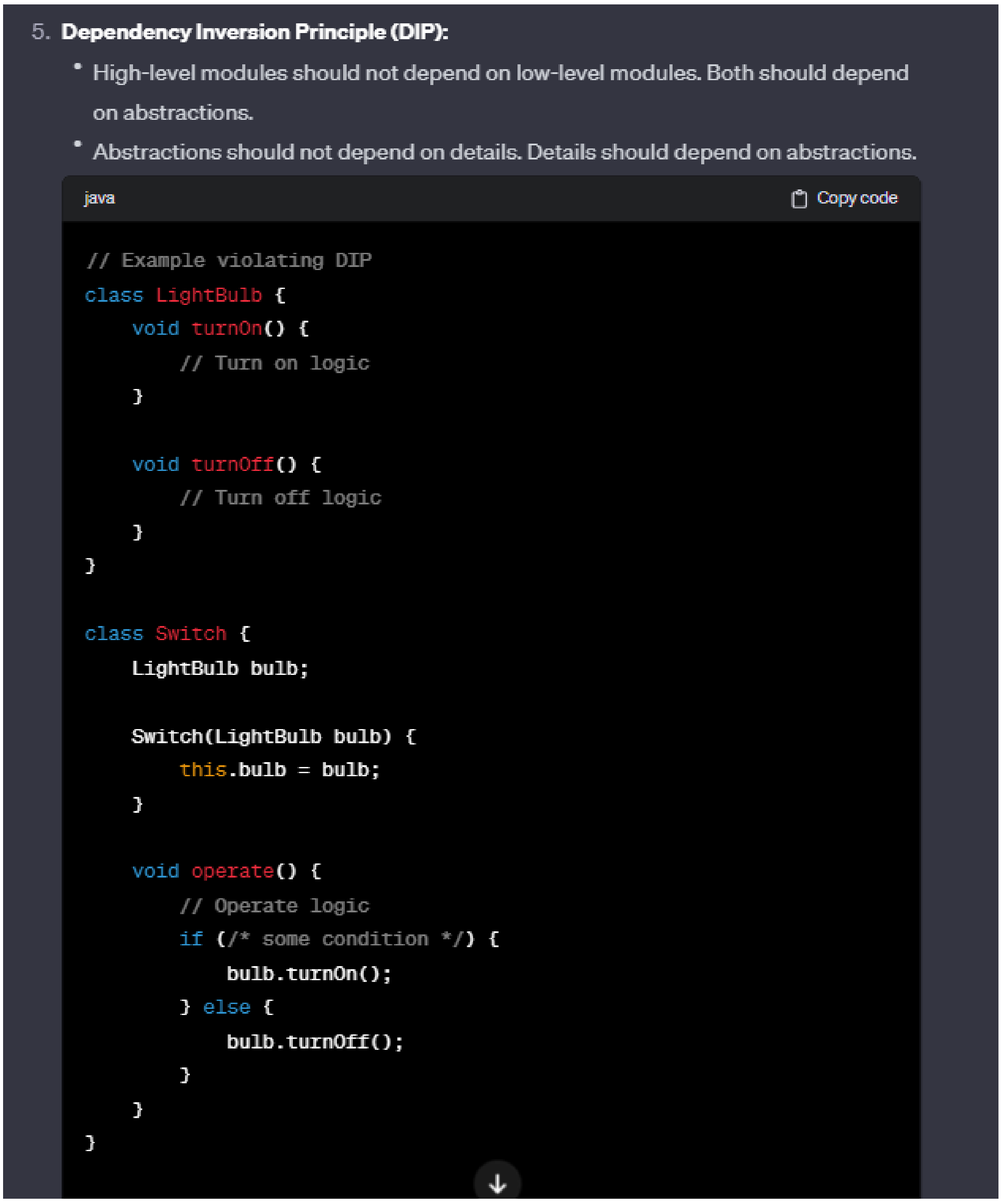
```java
// Example violating LSP
class Bird {
    void fly() {
        // Fly logic
    }
}

class Ostrich extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Ostrich can't fly");
    }
}

// Refactored to follow LSP
interface Bird {
    void move();
}

class FlyingBird implements Bird {
    @Override
    public void move() {
        // Fly logic
    }
}

class Ostrich implements Bird {
    @Override
    public void move() {
        // Run logic
    }
}
```



4. **Interface Segregation Principle (ISP):**
   - A class should not be forced to implement interfaces it does not use.
   - Clients should not be forced to depend on interfaces they do not use.

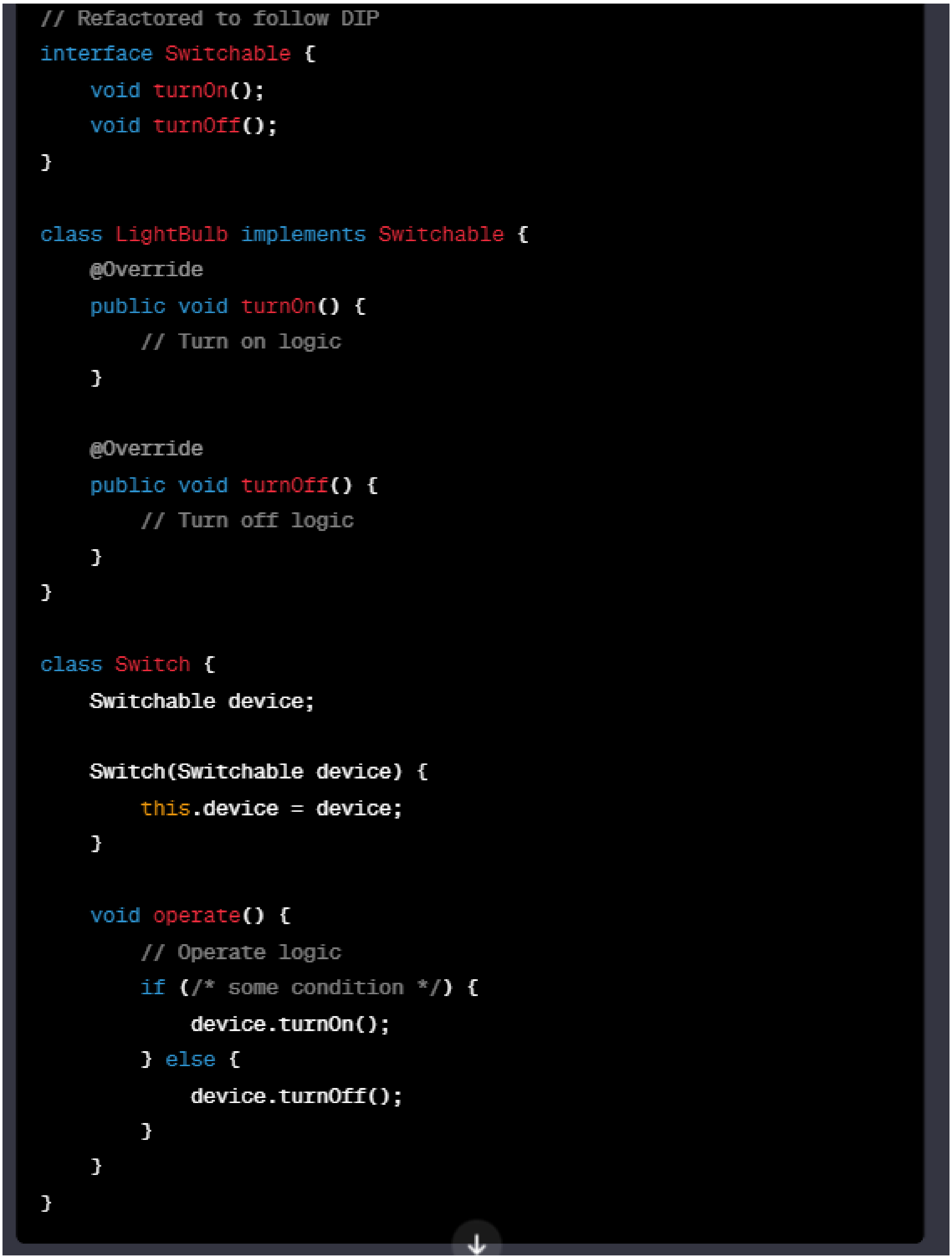
java Copy code

```java
// Example violating ISP
interface Worker {
    void work();

    void eat();
}

class Robot implements Worker {
    @Override
    public void work() {
        // Work logic
    }

    @Override
    public void eat() {
        // Eat logic, but robots don't eat
    }
}

// Refactored to follow ISP
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    @Override
    public void work() {
        // Work logic
    }

    @Override
    public void eat() {
        // Eat logic
    }
}

class Robot implements Workable {
    @Override
    public void work() {
        // Work logic
    }
}
```



5. **Dependency Inversion Principle (DIP):**
   - High-level modules should not depend on low-level modules. Both should depend on abstractions.
   - Abstractions should not depend on details. Details should depend on abstractions.

```java
// Example violating DIP
class LightBulb {
    void turnOn() {
        // Turn on logic
    }

    void turnOff() {
        // Turn off logic
    }
}

class Switch {
    LightBulb bulb;

    Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    void operate() {
        // Operate logic
        if (/* some condition */) {
            bulb.turnOn();
        } else {
            bulb.turnOff();
        }
    }
}
```



```java
// Refactored to follow DIP
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        // Turn on logic
    }

    @Override
    public void turnOff() {
        // Turn off logic
    }
}

class Switch {
    Switchable device;

    Switch(Switchable device) {
        this.device = device;
    }

    void operate() {
        // Operate logic
        if (/* some condition */) {
            device.turnOn();
        } else {
            device.turnOff();
        }
    }
}
```