**SOLID**

**Single Responsibility Principle (SRP)**

* A class should have only one reason to change, meaning it should have only one responsibility.
* Each class or module should focus on a single task or functionality. If a class has more than one responsibility, changes in one responsibility can affect the other, leading to tight coupling and difficulty in maintenance.
* E.G., A class Invoice should handle only invoice-related tasks like calculating totals. Generating invoice reports should be handled by a separate InvoiceReport class.

**Open Closed Principle (OCP)**

* A class should be open for extension, but closed for modification.
* New features or functionality should be added by extending the existing code (e.g., through inheritance or composition), rather than modifying the existing codebase. This prevents breaking the existing functionality and reduces the risk of introducing bugs.
* E.G., If you have a PaymentProcessor class, and you need to support a new payment type (e.g., PayPal), instead of modifying the existing class, you can create a new class (e.g., PayPalProcessor) that extends or implements the base PaymentProcessor.

**Liskov Substitution Principle (LSP)**

* Derived classes must be substitutable for their base classes.
* If a subclass is used in place of its parent class, the application should function correctly without altering its behavior. Subclasses must honor the contract defined by the parent class and not introduce unexpected behavior.
* E.G., If a Bird class has a method fly(), then a subclass like Penguin (which cannot fly) violates this principle. Instead, Bird should define a more general behavior, and specific behaviors like fly() should be handled in subclasses. *🡪 See mock test q2*

**Interface Segregation Principle (ISP)**

* A class should not be forced to implement interfaces it does not use.
* Instead of creating large, monolithic interfaces, split them into smaller, more specific ones. This way, classes only need to implement the methods they actually need, reducing unnecessary dependencies.
* E.G., Instead of a single Machine interface with methods like print(), scan(), and fax(), create separate interfaces such as Printer, Scanner, and FaxMachine. A Printer class only needs to implement the Printer interface.

**Dependency Inversion Principle (DIP)**

* High-level modules should not depend on low-level modules; both should depend on abstractions.
* Instead of directly depending on concrete implementations, classes should depend on abstractions (e.g., interfaces or abstract classes). This makes the system more flexible and easier to adapt to changes.
* E.G., A Notification class should not depend on specific implementations like EmailSender or SMSSender. Instead, it should depend on an interface MessageSender, allowing you to swap out EmailSender or SMSSender without modifying the Notification class.

|  |  |
| --- | --- |
| **SOLID** | **Benefit** |
| SRP | Easier to maintain and test by isolating functionality. |
| OCP | Reduces risk of breaking existing code and simplifies adding new features. |
| LSP | Ensures consistency and correctness when using polymorphism. |
| ISP | Reduces unnecessary dependencies and increases modularity |
| DIP | Promotes flexibility, testability, and decoupling between high- and low-level modules. |

|  |  |  |
| --- | --- | --- |
| **Other Principles** | **Explanation** | **Example** |
| Tell Don’t Ask | Instead of asking for data and acting on it, tell objects what to do to encapsulate logic. | * bankAccount.transferFunds() * if (account.getBalance() > amount) {...} |
| Once and Only Once | Every piece of knowledge or logic should have a single representation in the system. | * Reusing calculateDiscount() method. * Duplicating discount logic across methods. |
| Law of Demeter (LoD) | Methods should only interact with closely related objects or data they directly control. | * car.cleanSparkPlugs() * car.getEngine().getSparkPlugs().clean() |
| Package Principles | Organize classes into logical, cohesive packages based on domain or function. | * com.bank.account for account classes. * Mixing unrelated classes in one package. |
| DRY (Don’t Repeat) | Avoid redundant code by representing logic or knowledge in a single place. | * Centralized freuqently used method, such as validateInput(). * Repeating input validation in multiple places. |
| YAGNI (You Aren’t Gonna Need It) | Avoid implementing features that aren’t currently needed to meet requirements. | * Building only login functionality. * Adding unused user roles and permissions at the earlier stage of development. |

**Factory Method**

**Intention:** Provide a method to create objects without specifying their exact class.

**Purpose:** Simplifies object creation by delegating it to a factory method to ensure flexibility.

**Use Case:** Promotes code reuse and decouples object creation from the client.

**Structure:** Defines a factory method in an abstract class/interface.

**Key Mechanism:** Subclasses override the factory method to create specific types of objects.

Participant class:

* **Creator:** Declares the factory method, which returns an object of type Product.
* **Concrete Creator:** Overrides the factory method to return an instance of a ConcreteProduct.
* **Product:** Defines the interface of objects the factory method creates.
* **Concrete Product:** Implements the Product interface.

Pros & Cons:

|  |  |
| --- | --- |
| You **avoid tight coupling** between the creator and the concrete products.  **Single Responsibility Principle**. You can move the product creation code into one place in the program, making the code easier to support.  **Open/Closed Principle**. You can introduce new types of products into the program without breaking existing client code. | The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you’re introducing the pattern into an existing hierarchy of creator classes. |

**Abstract Factory**

**Intention:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Purpose:** Groups related factories for consistent object creation.

**Use Case:** Ideal when multiple families of objects need to be created with similar structures.

**Structure:** Defines an abstract factory interface with multiple factory methods for product families.

**Key Mechanism:** Concrete factories implement the abstract factory interface to create specific families of objects.

Participant class:

* **Abstract Factory:** Declares the interface for creating families of related objects.
* **Concrete Factory:** Implements the abstract factory interface to produce specific families of products.
* **Abstract Product:** Declares the interface for a type of product object.
* **Concrete Product:** Implements the interface of the abstract product.

Pros & Cons:

|  |  |
| --- | --- |
| You can be sure that the products you’re getting from a factory are compatible with each other.  You **avoid tight coupling** between concrete products and client code.  **Single Responsibility Principle.** You can extract the product creation code into one place, making the code easier to support.  **Open/Closed Principle.** You can introduce new variants of products without breaking existing client code. | The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern. |

**Prototype**

**Intention:** Create new objects by copying an existing object, known as prototype.

**Purpose:** Avoids the cost of creating objects from scratch.

**Use Case:** Useful when object creation is resource-intensive or when dynamic behaviour is required.

**Structure:** Relies on a prototype interface or abstract class with a clone() method.

**Key Mechanism:** Concrete prototypes implement the clone() method to create copies of themselves.

Participant class:

* **Client:** Request objects from the prototype by cloning them.
* **Prototype:** Declares an interface for cloning itself.
* **Concrete Prototype:** Implements the cloning operation to return a copy of itself.

Pros & Cons:

|  |  |
| --- | --- |
| You can clone objects without coupling to their concrete classes.  You can get rid of repeated initialization code in favor of cloning pre-built prototypes.  You can produce complex objects more conveniently.  You get an alternative to inheritance when dealing with configuration presets for complex objects.  Based on **Open Closed Principle.** | Cloning complex objects that have circular references might be very tricky. |

**Adapter**

**Intention:** Convert the interface of one class into another interface expected by the client.

**Purpose:** Enables classes with incompatible interfaces to work together.

**Use Case:** Common when integrating legacy systems with new components.

**Structure:** Involves a target interface, an adapter class, and an adaptee class.

**Key Mechanism:** The adapter implements the target interface and holds an instance of the adaptee to translate requests to the adaptee.

Participant class:

* **Client:** The system or code that expects to use the Target interface.
* **Target:** Defines the interface expected by the client.
* **Adapter:** Adapts the Adaptee to the Target interface.
* **Adaptee:** Defines the existing interface that need to be adapted.

Pros & Cons:

|  |  |
| --- | --- |
| * ***Single Responsibility Principle*.** You can separate the interface or data conversion code from the primary business logic of the program. * ***Open/Closed Principle*.** You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface. | The overall complexity of the code increases because you need to introduce a set of new interfaces and classes. Sometimes it’s simpler just to change the service class so that it matches the rest of your code. |

**Bridge**

**Intention**: Separate abstraction from implementation so that they can vary independently.

**Purpose:** Decouples high-level abstraction from low-level implementation, reducing hierarchical explosion in case the combination of these two via subclassing results in exponential growth of classes.

**Use Case:** Useful for extending functionality hierarchies without creating tightly coupled code.

**Structure:** Composed of an abstraction interface and an implementation interface, both connected via composition.

**Key Mechanism:** Concrete abstractions delegate behavior to implementations.

Participant class:

* **Implementor:** Declares the interface for the implementation.
* **Concrete Implementor:** Implements the Implementor interface.
* **Abstraction:** Defines the interface for high-level control and delegates implementation to the Implementor.
* **Refined Abstraction:** Extends the interface defined by the Abstraction.

Pros & Cons:

|  |  |
| --- | --- |
| * You can create platform-independent classes and apps. * The client code works with high-level abstractions. It isn’t exposed to the platform details. * ***Open/Closed Principle*.** You can introduce new abstractions and implementations independently from each other. * ***Single Responsibility Principle*.** You can focus on high-level logic in the abstraction and on platform details in the implementation. | * You might make the code more complicated by applying the pattern to a highly cohesive class. |

**Decorator**

**Intention:** Dynamically add new responsibilities to objects without altering their structure.

**Purpose:** Provides a flexible alternative to subclassing for extending behavior.

**Use Case:** Used to add features to objects in a modular and composable way.

**Structure:** Involves a component interface, a concrete component, and decorators that wrap the component.

**Key Mechanism:** Each decorator implements the component interface and adds additional behavior.

Participant class:

* **Component:** Defines the interface for objects that can have responsibilities added dynamically.
* **Concrete Component:** Implements the Component interface and defines the base behaviour.
* **Decorator:** Maintains a reference to a Component object and defines an interface conforming to Component.
* **Concrete Decorator:** Adds responsibilities to the component.

Pros & Cons:

|  |  |
| --- | --- |
| * You can extend an object’s behavior without making a new subclass. * You can add or remove responsibilities from an object at runtime. * You can combine several behaviors by wrapping an object into multiple decorators. * ***Single Responsibility Principle*.** You can divide a monolithic class that implements many possible variants of behavior into several smaller classes. | * It’s hard to remove a specific wrapper from the wrappers stack. * It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack. * The initial configuration code of layers might look pretty ugly. |

**Command**

**Intention:** Encapsulate a request as an object, allowing parameterization of requests and queuing them.

**Purpose:** Decouples the request and the invoker of the request.

**Use Case:** Useful for implementing undo/redo functionalities.

**Structure:** Involes a command interface, concrete commands, a receiver, and an invoker.

**Key Mechanism:** The invoker executes commands that interact with receivers.

Participant class:

* **Command:** Declares an interface for executing an operation.
* **Concrete Command:** Impleements Execute by invoking the corresponding operations on the receiver.
* **Client:** Creates Concrete Command objects and sets their receivers.
* **Invoker:** Asks the command to carry out the request.
* **Receiver:** Knows how to perform the operations associated with a command.

Pros & Cons:

|  |  |
| --- | --- |
| * ***Single Responsibility Principle*.** You can decouple classes that invoke operations from classes that perform these operations. * ***Open/Closed Principle*.** You can introduce new commands into the app without breaking existing client code. * You can implement **undo/redo**. * You can implement deferred execution of operations. * You can assemble a set of simple commands into a complex one. | * The code may become more complicated since you’re introducing a whole new layer between senders and receivers. |

**Chain of Responsibility**

**Intention:** Pass a request along a chain of handlers until it is handled.

**Purpose:** Decouples sender and receiver by giving multiple objects a chance to handle a request.

**Use Case:** Used in scenarios like logging, event processing, or approval workflows.

**Structure:** Consists of a handler interface, concrete handlers, and a client.

**Key Mechanism:** Each handler forwards the request if it cannot handle it.

Participant class:

**Client**: Initiates requests to the handler.

**Handler:** Defines the interface for handling requests and setting the next handler.

**Concrete Handler:** Processes requests or forwards them to the next handler in the chain.

Pros & Cons:

|  |  |
| --- | --- |
| * You can control the order of request handling. * ***Single Responsibility Principle*.** You can decouple classes that invoke operations from classes that perform operations. * ***Open/Closed Principle*.** You can introduce new handlers into the app without breaking the existing client code. | * Some requests may end up unhandled. |

**Memento**

**Intention:** Capture and restore the internal state of an object without exposing its details.

**Purpose:** Provides a way to implement undo or rollback functionality.

**Use Case:** Ideal for applications requiring versioning or reversible operations.

**Structure:** Involves a memento, an originator, and a caretaker.

**Key Mechanism:** The originator creates a memento containing its state, which the caretaker stores and later restores.

Participant class:

* **Originator:** Creates a memento containing its state and can restore its state from a memento.
* **Memento:** Stores the internal state of the originator without violating encapsulation.
* **Caretaker**: Maintains a collection of mementos and manages the originator’s state restoration.

Pros & Cons:

|  |  |
| --- | --- |
| * You can produce snapshots of the object’s state without violating its encapsulation. * ***SRP.*** You can simplify the originator’s code by letting the caretaker maintain the history of the originator’s state. * ***Open Closed Principle*.** The addition of undo/rollback functionality without modifying the core logic of the originator or caretaker. | * The app might consume lots of RAM if clients create mementos too often. * Caretakers should track the originator’s lifecycle to be able to destroy obsolete mementos. * Most dynamic programming languages, such as PHP, Python and JavaScript, can’t guarantee that the state within the memento stays untouched. |

**Observer**

**Intention:** Define a one-to-many dependency between objects so that when one changes state, all its dependents are notified.

**Purpose:** Facilitates communication between objects without tight coupling.

**Use Case:** Common in event-driven systems or GUIs. MVC architecture.

**Structure:** Includes a subject, concrete subject, observer, and concrete observer.

**Key Mechanism**: Observers register with a subject and are notified of changes.

Participant class:

* **Subject:** Defines an interface for attaching, detaching and notifying Observer objects.
* **Observer:** Defines the interface for objects that should be notified of changes.
* **Concrete Subject:** Maintains a list of observers, stores states, and notifies them of changes in state.
* **Concrete Observer:** Implements the Observer interface and updates itself when notified.

Pros & Cons:

|  |  |
| --- | --- |
| * ***Open/Closed Principle*.** You can introduce new subscriber classes without having to change the publisher’s code (and vice versa if there’s a publisher interface). * Loose coupling between publisher and subscriber (publisher do not need to know the subscribers) * You can establish relations between objects (attach observers to subject) at runtime. | Problems to determining whether the update to the publisher is of relevance to all subscribers or just some of them. Sending an update signal to all subscribers - **communication overhead** |

**State**

**Intention:** Manage state-specific behavior in an object in a way that delegates behavior ot the object’s current state.

**Purpose:** Allows an object to change its behaviour when its internal state changes.

**Use Case:** Focuses on changing the behaviour of an object based on its state. Alternative to switch and if-else statements.

**Structure:** Involves a context class that maintains an instance of s state subclass.

**Key Mechanism:** The state subclasses implement behaviour associated with a particular state of the context.

Participant class:

* **Context:** Maintains a reference to a state object and delegates behavior to the current state.
* **State:** Declares an interface for encapsulating state-specific behaviour.
* **Concrete State:** Implements the behaviour associated with a specific state of the Context.

Pros & Cons:

|  |  |
| --- | --- |
| * ***Single Responsibility Principle*.** Organize the code related to particular states into separate classes. * ***Open/Closed Principle*.** Introduce new states without changing existing state classes or the context. * Simplify the code of the context by eliminating bulky state machine conditionals. | * Applying the pattern can be overkill if a state machine has only a few states or rarely changes. |

**Strategy**

**Intention:** Define a family of algorithms, encapsulate each one, and make them interchangeable.

**Purpose:** Allows the algorithm to vary independently from the clients that use it.

**Use Case:** Focuses on selecting an algorithm at run time.

**Structure:** Involves a context class that maintains a reference to a strategy interface.

**Key Mechanism:** Concrete strategy classes implement the strategy interface and define specific algorithms.

Participant class:

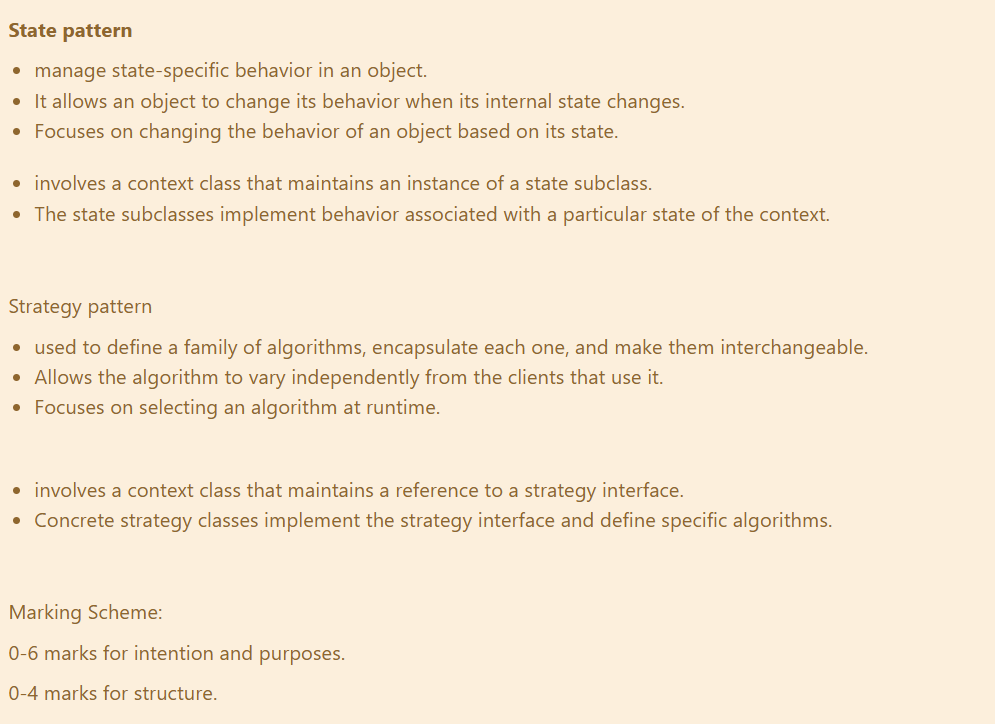
* **Context:** Maintains a reference to a strategy object and allows algorithms to be interchangeable.
* **Strategy:** Declares an interface for the family of algorithms.
* **Concrete Strategy:** Implements the algorithm defined by the Strategy interface.

|  |  |
| --- | --- |
| * You can swap algorithms used inside an object at runtime. * You can isolate the implementation details of an algorithm from the code that uses it. * You can replace inheritance with composition. * ***Open/Closed Principle*.** You can introduce new strategies without having to change the context. | * If you only have a couple of algorithms and they rarely change, there’s no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern. * Clients must be aware of the differences between strategies to be able to select a proper one. * A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you’d have used the strategy objects, but without bloating your code with extra classes and interfaces. |

**Mock Test**

A black text on a white background

Description automatically generated



A screenshot of a computer program

Description automatically generated

A screenshot of a computer

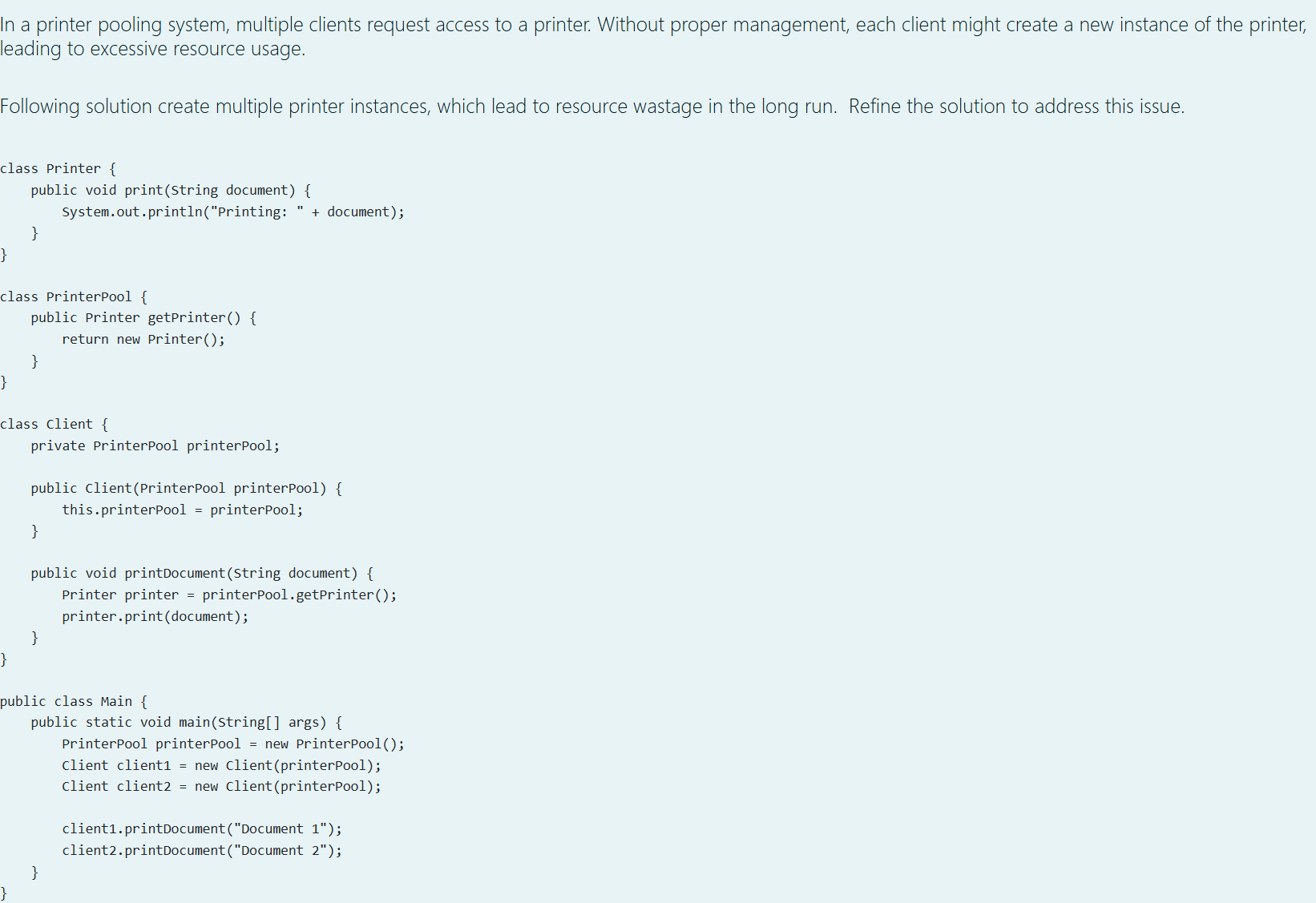
Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated



A screenshot of a computer program

Description automatically generated

A close-up of a text

Description automatically generated

A screenshot of a computer

Description automatically generated