## **OOAD Assignment 6**

## **Problem 6: Questions: Decide which Pattern**

What pattern best meets each of these situations below?

1. The model stores a collection of blocks. Blocks can be medal or wood; can be painted, sanded, or chrome plated; and can sometimes be radioactive or magnetized. What design pattern would allow the system to easily add new types of blocks without changing existing code?
   * Explain how two design *principles* apply to this design.
2. The model for a game stores robot. The robot navigates a maze that has obstacles. While playing the game, the robot can be upgraded with new parts that change its abilities like speed, weapons, and shields. Which design principle allows the robot object to change its behaviour at runtime in flexible ways.
   * Explain how two design *principles* apply to this design.
3. The model stores a phone number, and the UI (a keypad) allows the user to enter digits one at a time. A different part of the UI wants to respond to each key being entered; however a) the two parts of the UI should be decoupled, and b) the model should be decoupled from the UI (model knows nothing about the UI). Which pattern would allow this?
   * Explain how two design *principles* apply to this design.
4. A library supports recursively searching a directory for files. It allows the client code to provide it an object to filter the results. For each file which the library finds, it will ask the filter object if that file should be accepted or rejected. (See Java's [FileFilter](https://docs.oracle.com/javase/8/docs/api/java/io/FileFilter.html))
   * Explain how two design *principles* apply to this design.

1)

To allow the system to easily add new types of blocks without changing existing code, the "Factory Method" design pattern would be suitable for this scenario.

1. Factory Method Design Pattern:

The Factory Method pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. In the context of the block collection, we can create an abstract BlockFactory class that defines a method for creating blocks. Then, we can have different concrete BlockFactory subclasses for each type of block (medal, wood), which would handle the creation of specific block instances.

Here's how the design would work:

1. Create an abstract `Block` class with common properties like `painted`, `sanded`, `chromePlated`, `radioactive`, `magnetized`, etc.

2. Create concrete classes for each type of block, such as `MedalBlock` and `WoodBlock`, which inherit from the abstract `Block` class.

3. Create an abstract `BlockFactory` class with a method, let's say `createBlock`, that returns a new `Block` instance.

4. Create concrete `BlockFactory` subclasses for each type of block, like `MedalBlockFactory` and `WoodBlockFactory`. Each factory subclass should implement the `createBlock` method, creating an instance of the respective block type.

By using the Factory Method pattern, you can easily add new types of blocks without changing existing code. To add a new type of block (e.g., `GlassBlock`), you would create a new concrete `GlassBlock` class that extends the `Block` class and a corresponding `GlassBlockFactory` that extends the `BlockFactory` class. This way, you can achieve code flexibility and modularity.

2. Design Principles:

a. Open/Closed Principle (OCP):

The Factory Method pattern adheres to the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. In this context, adding new types of blocks is done by creating new concrete classes and subclasses, without altering the existing code. The existing code that uses the abstract `BlockFactory` and `Block` classes remains unchanged and can work with the new block types added through inheritance.

b. Single Responsibility Principle (SRP):

The Single Responsibility Principle suggests that a class should have only one reason to change. By using the Factory Method pattern, the responsibility of creating different types of blocks is separated into distinct `BlockFactory` subclasses. Each factory subclass is responsible for creating one specific type of block, keeping the code focused on a single responsibility. This makes the codebase more maintainable and easier to understand.

In summary, the Factory Method design pattern along with the Open/Closed Principle and Single Responsibility Principle, provides a flexible and maintainable solution for adding new types of blocks

to the system without affecting the existing codebase.

2)

The design principle that allows the robot object to change its behavior at runtime in flexible ways is the "Strategy" design pattern.

1. Strategy Design Pattern:

The Strategy pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one of them, and makes them interchangeable. It allows the client (in this case, the robot object) to change the algorithm or behavior at runtime without altering the robot's code. In the context of the game's robot, this pattern enables the robot to be upgraded with new parts that change its abilities (speed, weapons, shields) dynamically during gameplay.

Here's how the design would work:

1. Create an abstract `Robot` class that defines the common behavior of the robot, such as navigation through the maze and interaction with the game environment.

2. Create concrete subclasses for the different types of robots, like `BasicRobot`, `FastRobot`, `ArmedRobot`, etc. Each subclass would represent a different strategy with specific abilities.

3. Create an interface or abstract class, let's say `UpgradeStrategy`, which defines the methods related to the robot's abilities, such as `upgradeSpeed()`, `upgradeWeapons()`, and `upgradeShields()`.

4. Implement concrete classes for different upgrade strategies, like `SpeedUpgrade`, `WeaponsUpgrade`, and `ShieldsUpgrade`. Each upgrade strategy class would provide specific implementations for the upgrade methods.

5. Add a set method in the `Robot` class to set the current upgrade strategy. This method allows the robot to switch between different upgrade strategies at runtime.

By using the Strategy pattern, the robot object can easily change its behavior at runtime by switching to different upgrade strategies. During the game, if the player acquires a new part that improves the robot's speed, weapons, or shields, the robot can apply the corresponding upgrade strategy dynamically without needing to modify its core logic.

2. Design Principles:

a. Open/Closed Principle (OCP):

The Strategy pattern adheres to the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. In this context, when new upgrade parts are introduced, you create new concrete classes for the corresponding upgrade strategy without altering the existing robot or maze navigation code. This allows the robot to be extended with new abilities (strategies) without modifying its core implementation, making the code more maintainable and extensible.

b. Dependency Inversion Principle (DIP):

The Dependency Inversion Principle suggests that high-level modules should not depend on low-level modules; both should depend on abstractions. By using the Strategy pattern, the `Robot` class (high-level module) depends on the `UpgradeStrategy` interface (abstraction) rather than concrete upgrade classes (low-level modules). This decouples the robot from specific upgrade implementations, enabling the robot to easily switch between different upgrade strategies without tightly coupling it to the concrete upgrade classes.

In summary, the Strategy design pattern, along with the Open/Closed Principle and Dependency Inversion Principle, enables the robot object in the game to change its behavior at runtime in flexible ways by dynamically applying different upgrade strategies. This approach promotes code reusability, extensibility, and maintainability in the game's design.

3)

The design pattern that would allow the two parts of the UI (phone number input and response handling) to be decoupled and the model to be decoupled from the UI is the "Observer" design pattern.

1. Observer Design Pattern:

The Observer pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, such that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically. In this scenario, the phone number input component (subject) would notify the response handling component (observer) whenever a digit is entered, and the response handling component would respond accordingly.

Here's how the design would work:

1. Define an interface, let's call it `NumberInputObserver`, with a method, for example, `onDigitEntered(digit: string)`. This interface will be implemented by the response handling component.

2. The phone number input component (subject) will maintain a list of `NumberInputObserver` objects (response handling components) that want to be notified when a digit is entered.

3. Whenever a digit is entered into the phone number input component, it will call the `onDigitEntered` method on all the registered `NumberInputObserver` objects, passing the entered digit as a parameter.

4. The response handling component (observer) will implement the `NumberInputObserver` interface, which will allow it to receive notifications about the entered digits.

By using the Observer pattern, the two parts of the UI (phone number input and response handling) are decoupled. The phone number input component does not need to know anything about the response handling component, and vice versa. Each component can operate independently, and the communication between them is established through the observer interface.

2. Design Principles:

a. Single Responsibility Principle (SRP):

The Single Responsibility Principle suggests that a class should have only one reason to change. By using the Observer pattern, the responsibility of responding to the entered digits is separated from the responsibility of handling the phone number input. The phone number input component is responsible for capturing user input and notifying the observers, while the response handling component is responsible for handling the response based on the entered digits. This separation of responsibilities makes the codebase more maintainable and easier to understand.

b. Dependency Inversion Principle (DIP):

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. The Observer pattern follows this principle by introducing an abstract interface (`NumberInputObserver`) that defines the contract for the response handling components to implement. This allows the phone number input component to depend on the abstraction (the interface) rather than on specific response handling implementations. As a result, the phone number input and response handling components are loosely coupled, promoting flexibility and ease of modification.

In summary, the Observer design pattern, along with the Single Responsibility Principle and Dependency Inversion Principle, facilitates a decoupled design where the phone number input and response handling components can operate independently, and the model remains decoupled from the UI. This approach enhances modularity, reusability, and maintainability in the design of the UI components.

4)

Two design principles that apply to this design are the Open/Closed Principle (OCP) and the Dependency Inversion Principle (DIP).

1. Open/Closed Principle (OCP):

The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In the context of the library supporting file searching with a filter, the design adheres to this principle by allowing the client code to provide a custom filter object to the library. The library's core file searching algorithm remains unchanged, but it is open for extension because it can work with different filter implementations without modifying its code.

The `FileFilter` interface (similar to Java's FileFilter) acts as an abstraction that defines the contract for filter objects. Client code can create custom filter classes by implementing this interface and passing the filter object to the library. This allows the library to use the provided filter to accept or reject files during the recursive search without altering its implementation.

By adhering to the Open/Closed Principle, the library design promotes code flexibility and reusability. New filter types can be introduced by creating additional classes that implement the `FileFilter` interface, without affecting the existing library code. This makes it easy to adapt the library to different filtering requirements without extensive modifications.

2. Dependency Inversion Principle (DIP):

The Dependency Inversion Principle suggests that high-level modules should not depend on low-level modules; both should depend on abstractions. In the context of the library, the file search algorithm (the high-level module) depends on the abstraction of the `FileFilter` interface (the low-level module) rather than depending on specific filter implementations.

The library doesn't know the details of the filter objects passed by the client code; it only knows that they adhere to the `FileFilter` interface. This decouples the library from the specific filtering logic, promoting loose coupling between the file search algorithm and the filtering process. As a result, the library can remain agnostic about the specific filtering criteria, making it more versatile and easier to maintain.

By following the Dependency Inversion Principle, the library design is more flexible and extensible. It allows the client code to introduce new filter implementations without modifying the library, and it also allows the library to be used with other filtering mechanisms, improving overall modularity and reducing potential code entanglements.

In summary, by adhering to the Open/Closed Principle and the Dependency Inversion Principle, the library design enables recursive searching of a directory for files with flexible filtering options. The design promotes code extensibility, modularity, and maintainability by allowing easy integration of new filtering logic without changing the core file searching algorithm.