National University
Of Computer and Emerging Sciences

Spring-2025 **Assignment# 04** **Max. Marks: 10**

Course Code: SE-4001

Course Title: Software Re-Engineering

Question # 1: Provide Object Oriented Adapter Design Pattern Code for the Scenario given below.

Imagine you work for a company called TechNotify, which provides a notification system for businesses to send alerts, reminders, and messages to their users. TechNotify has been offering two services for notifications: EmailService for email notifications and SMSService for SMS notifications. Both of these services have a common interface, with methods like sendMessage() and checkStatus(), which allow businesses to integrate them easily into their systems.

Recently, TechNotify has decided to expand its offerings by adding support for push notifications, which are becoming more popular among users due to their real-time nature. After evaluating several push notification providers, TechNotify selects PushService to handle this new feature. However, PushService has an entirely different API from EmailService and SMSService. It uses methods like createPushNotification() and checkPushNotificationStatus(), which are incompatible with the existing system that expects methods like sendMessage() and checkStatus().

The issue arises because TechNotify's existing notification system is already in use by many clients, and rewriting the entire system to accommodate PushService would require significant time and resources. Rather than refactoring the entire system, TechNotify can use the Adapter Pattern to integrate PushService while keeping the existing functionality intact.

The Adapter Pattern allows TechNotify to create a wrapper (the adapter) around the PushService API, which converts its methods into the methods expected by the existing system. The adapter will implement the common interface that both EmailService and SMSService use, such as sendMessage() and checkStatus(). Internally, the adapter will map these method calls to the corresponding methods in PushService, such as createPushNotification() and checkPushNotificationStatus(). This approach allows TechNotify to continue using the same interface for all notification services, regardless of their underlying differences.

For example, when TechNotify's system calls sendMessage(), the adapter will internally invoke createPushNotification() in PushService, passing the necessary parameters like title, message, and recipientId. Similarly, when checkStatus() is called, the adapter will map it to checkPushNotificationStatus() in PushService, ensuring that the status is checked in a consistent way across all services.

By using the Adapter Pattern, TechNotify can integrate PushService without disrupting its existing system. The core logic of sending notifications remains the same, and clients using the system for email and SMS notifications won't experience any changes in behavior. This approach also allows TechNotify to add more notification services in the future. For example, if TechNotify decides to integrate VoiceService for voice-based notifications, it can simply create a new adapter for that service without altering the existing codebase.

The Adapter Pattern allows TechNotify to seamlessly add PushService to its notification system without the need for extensive changes. It ensures that the system remains flexible and easy to extend, enabling future integrations with other services while preserving the current functionality.

Question # 2: Implement Object Oriented Factory Design Pattern for the Scenario given below.

Imagine you are working for an airline company that has developed a flight booking system called SkyBook. The SkyBook system allows users to book flights, select their seat preferences, and choose additional services like meal options, extra baggage, and priority boarding. The system has been successful for a while, but the company now wants to expand its services to include multiple types of flights. Specifically, they want to offer three distinct classes of flights: Economy, Business, and First Class. Each flight class will have different pricing, services, and rules regarding baggage allowances, seat reservations, and meal options.

The challenge, however, is that each of these flight classes will have unique features, and the system should be able to handle them dynamically without much manual intervention. To meet this need, the development team decides to implement the Factory Pattern, a creational design pattern that provides an interface for creating objects without specifying the exact class of object that will be created. This allows the system to manage different types of flight objects and their associated features in a modular, scalable way.

In the SkyBook system, when a customer selects a flight class, the system needs to instantiate the correct class of flight (Economy, Business, or First Class) with the appropriate services and attributes. For example, the Economy Class will have basic services like limited baggage and standard meal options. The Business Class will offer more luxurious amenities, such as larger seats, priority boarding, and additional baggage. Finally, the First Class will include premium services such as gourmet meals, extra baggage, and access to exclusive lounges.

To handle this variability, the Factory Pattern will be implemented. The FlightFactory class will act as a factory responsible for creating instances of the different flight classes. When a customer selects their preferred flight class, the factory will determine which specific class of flight to create based on that input, and instantiate the appropriate class with the corresponding features. This approach ensures that the core system logic remains clean and easily extendable, as it centralizes the creation of flight objects in one place, rather than scattered across various parts of the system.

The FlightFactory will have a method, say createFlight(), which will take in a parameter representing the selected class of flight (Economy, Business, or First Class). Based on this input, it will create an instance of the corresponding flight class and return it. Each flight class will implement a common interface, say Flight, ensuring that all flight types share a common set of operations, such as bookSeat(), calculatePrice(), and displayAmenities(). This common interface allows the system to treat all flight types uniformly, making the system easier to maintain and extend with new classes in the future.

For example, if the customer selects Business Class, the factory will create an object of BusinessClassFlight, which implements the common Flight interface but has its own specific features, such as priority boarding and additional baggage. Similarly, if the customer chooses First Class, the factory will create an object of FirstClassFlight with exclusive services.

By using the Factory Pattern, the system avoids having complex conditional logic or duplicated code when creating different flight types. The main advantage of this pattern in the SkyBook system is that it allows the company to add more flight classes in the future without modifying the core system logic. For instance, if the company decides to introduce a Premium Economy Class or a Charter Flight in the future, the development team can simply create new classes and update the factory to handle the new types, without impacting the rest of the booking system.

The Factory Pattern in this scenario helps SkyBook to manage different types of flights efficiently. It allows the system to create various types of flights in a flexible, maintainable, and scalable way while keeping the codebase clean and easy to extend. This pattern is ideal for scenarios where objects have similar behaviors but differ in specific characteristics, making it the perfect solution for SkyBook's flight booking system.

Good Luck!