Design Document for ABCD Company Mobile App

Fundamentals of Software Design: CS374

Antoine Gaton

October 20, 2024

Contents

[Introduction 3](#_Toc140489050)

[Requirement (a) 3](#_Toc140489051)

[Design Pattern Used For Requirement (a) 3](#_Toc140489052)

[Name of the Design Pattern and Category of the Design Pattern 3](#_Toc140489053)

[UML Diagram of the Design Pattern 4](#_Toc140489054)

[Discussion 4](#_Toc140489055)

[Requirement (b) 5](#_Toc140489056)

[Design Pattern Used for Requirement (b) 5](#_Toc140489057)

[Name of the Design Pattern and Category of the Design Pattern 5](#_Toc140489058)

[UML Diagram of the Design Pattern 6](#_Toc140489059)

[Discussion 6](#_Toc140489060)

[Requirement (c) 7](#_Toc140489061)

[Design Pattern used for Requirement (c) 7](#_Toc140489062)

[Name of the Design Pattern and Category of the Design Pattern 7](#_Toc140489063)

[UML Diagram of the Design Pattern 8](#_Toc140489064)

[Discussion 8](#_Toc140489065)

[References (4 or More References in APA Format) 9](#_Toc140489066)

# Introduction

ABCD Company is developing a new mobile app to enhance customer experience by providing most of the functions currently available on their website. As the software architect, I have been tasked with designing this app based on specific requirements. This document outlines the design patterns chosen to address each requirement, along with UML diagrams and explanations for each choice.

# Requirement (a)

The new Mobile App Interface object class must re-use the existing Interface object class that is created for the company's Web services.

## Design Pattern Used for Requirement (a)

* *Name of the design pattern*: Adapter Pattern
* *Category of the design pattern*: Structural Pattern

## Name of the Design Pattern and Category of the Design Pattern

* *Adapter Pattern*: is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a wrapper between two objects, catching calls for one object and transforming them to format and interface recognizable by the second object.
  + **Key Characteristics**:
    - Interface Compatibility: Enables classes with incompatible interfaces to collaborate by converting one interface to another that the client expects.
    - Single Responsibility: Isolates conversion logic into a separate adapter class without altering the original classes.
    - Transparency: Allows clients to interact with the adapter without needing to know it is an adapter.
    - Flexibility: Facilitates using existing classes in new contexts without modifying their source code.
    - Wrapper Class: The adapter serves as a wrapper, sitting between the client and the adaptee, translating requests between them.
  + **Example Use Case**:
    - In a payment processing system, you want to integrate a third-party payment service that uses a different interface than your system. Instead of modifying your entire system, an adapter is created to translate your system’s requests into calls that the third-party service understands. This allows you to integrate the service without changing your existing system's core.
* *Structural Pattern*: in software design is one of the three categories of design patterns (the other two being Creational and Behavioral patterns). Structural patterns focus on the organization and relationships between objects and classes, making it easier to design flexible and efficient structures. They aim to simplify the system by defining how objects or classes should be composed to form larger, more complex structures, while ensuring that these systems remain flexible and maintainable.
  + **Key goals of Structural Patterns include**:
    - Facilitating relationships between entities to enhance flexibility.
    - Making it easier to change the composition of objects without affecting the overall system.
    - Promoting code reusability and scalability.

## UML Diagram of the Design Pattern

A screen shot of a computer

Description automatically generated

## Discussion

The Adapter Pattern is chosen for this requirement because it allows the new Mobile App Interface (Target) to work with the existing Web Interface (Adaptee) without modifying the existing code. This pattern translates the interface of one class into another interface that clients expect, enabling classes to work together that couldn't otherwise be due to incompatible interfaces.

* In this case:
  + The Client represents the mobile app components that will use the interface.
  + The Target is the desired interface for the mobile app.
  + The Adaptee is the existing Web Interface class.
  + The Adapter is the new Mobile App Interface class that adapts the existing Web Interface to work with the mobile app.

This pattern allows for the reuse of the existing Web Interface while providing a suitable interface for the mobile app, fulfilling the requirement efficiently.

# Requirement (b)

Whenever any new data, through either of the two interfaces, have been stored in the data server repository, a notification message will be sent to both Interface objects.

## Design Pattern Used for Requirement (b)

* *Name of the design pattern*: Observer Pattern
* *Category of the design pattern*: Behavioral Pattern

## Name of the Design Pattern and Category of the Design Pattern

* *Observer Pattern*: is a behavioral design pattern that defines a one-to-many relationship between objects so that when one object, the *subject*, changes its state, all of its dependent objects, the *observers,* are automatically notified and updated. This pattern is often used in event-driven systems where an object needs to inform multiple objects about a state change without needing to know about all the dependent objects.
  + **Key Characteristics**:
    - Loose coupling between the subject and observers.
    - The subject doesn't need to know the concrete class of the observers, only that they implement a specific interface.
    - Allows dynamic subscription and removal of observers.
  + **Example Use Case**: In a graphical user interface (GUI), buttons (subjects) can notify multiple listeners (observers) when clicked, triggering different actions such as saving a file, updating a log, or sending data.
* *Behavioral Pattern*: is one of the three categories of design patterns (along with Creational and Structural patterns). Behavioral patterns focus on communication and interaction between objects, defining how objects cooperate to perform tasks and delegate responsibilities. These patterns manage the flow of control and the responsibilities between interacting objects, promoting flexible, scalable, and maintainable systems.
  + **Key Characteristics**:
    - **Focus on Communication**: Behavioral patterns concentrate on how objects communicate and interact to achieve complex tasks.
    - **Delegation of Responsibility**: They help in clearly defining how tasks and responsibilities are shared among objects.
    - **Improved Flexibility**: These patterns reduce tight coupling between interacting objects, making the system more adaptable to changes.
    - **Manage Workflow**: They facilitate better control over workflows and data flows within a system.
  + **Example Use Case**: In a messaging system, the **Observer Pattern** can be used to notify multiple services (like email, SMS, and app notifications) when a user receives a new message. The system doesn't need to know the specifics of each notification service but can rely on the pattern to notify all observers (services) when the message event occurs. This makes the system easier to extend with new notification services in the future.

## UML Diagram of the Design Pattern

A diagram of a computer program

Description automatically generated with medium confidence

## Discussion

The Observer Pattern is ideal for this requirement as it establishes a one-to-many dependency between objects. When the state of one object (the subject) changes, all its dependents (observers) are notified and updated automatically.

* In this scenario:
  + The Subject is the data server repository.
  + The ConcreteSubject is the actual implementation of the data server.
  + The Observer interface represents the notification mechanism.
  + The ConcreteObserver classes are the Web Interface and Mobile App Interface.

When new data is stored in the repository (ConcreteSubject), it calls the notify() method, which in turn calls the update() method on all registered observers (both interface objects). This ensures that both interfaces are notified of any changes in the data server, fulfilling the requirement efficiently.

# Requirement (c)

The Mobile App Interface object class must be able to support multiple types of mobile operating systems, such as Android, iOS, and Windows Phone.

## Design Pattern Used for Requirement (c)

* Name of the design pattern: Factory Method Pattern
* Category of the design pattern: Creational Pattern

## Name of the Design Pattern and Category of the Design Pattern

* *Factory Method Pattern*: Is a creational design pattern that defines an interface for creating objects but allows subclasses to alter the type of objects that will be created. Instead of instantiating objects directly, this pattern provides a method to create objects in a way that promotes loose coupling and flexibility, particularly in cases where the exact type of object to be created isn’t known until runtime.
  + **Key Characteristics**:
    - Encapsulation of Object Creation: The process of object creation is encapsulated in a factory method, promoting flexibility in changing the object types without modifying the client code.
    - Loose Coupling: The client code depends on abstractions rather than concrete classes, reducing direct dependencies between components.
    - Subclass Responsibility: Subclasses override the factory method to specify the type of objects that should be created.
    - Promotes Extensibility: It’s easy to introduce new types of objects by adding new subclasses without affecting existing code.
  + **Example Use Case**: A logistics system needs to create different types of transport objects (e.g., truck, ship, airplane) depending on the delivery method. Instead of hardcoding object creation, a factory method can be used, where each subclass (like TruckTransport or ShipTransport) creates the appropriate transport type. This makes it easier to extend the system when new transport types are introduced.
* *Creational Pattern*: is one of the three main categories of design patterns (alongside Structural and Behavioral patterns). Creational patterns deal with the process of object creation, providing solutions to instantiate objects in a way that enhances flexibility and reusability. These patterns abstract the object creation process, allowing a system to decide which objects to create and how to create them at runtime.
  + **Key Characteristics**:
    - Object Creation Abstraction: They abstract the object creation process, providing flexible mechanisms for instantiating objects.
    - Flexible Instantiation: They allow objects to be created in a flexible manner, often deciding which class to instantiate at runtime.
    - Encourages Reusability: The patterns promote reusability by decoupling object creation logic from the client code.
    - Manages Complex Creation: They are useful when object creation involves complex logic or when objects need to be created in different ways depending on context.
  + **Example Use Case**: In a game development system, different types of characters (warrior, mage, archer) need to be created. A creational pattern, such as the Factory Method or Abstract Factory, can be used to abstract away the process of creating these characters. This allows the system to dynamically decide which character type to instantiate, simplifying the client code and making the system easier to extend for new character types.

## UML Diagram of the Design Pattern

A screenshot of a computer

Description automatically generated

## Discussion

The Factory Method Pattern is chosen for this requirement because it provides an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. This pattern is particularly useful when dealing with product families, such as different mobile operating systems.

* In this design:
  + MobileAppInterface is the abstract creator class that declares the factory method createOSSpecificInterface().
  + AndroidInterface, iOSInterface, and WindowsPhoneInterface are concrete creators that implement the factory method to produce the appropriate interface for each OS.
  + OSSpecificInterface is the common interface for all OS-specific interfaces.
  + AndroidSpecificInterface, iOSSpecificInterface, and WindowsPhoneSpecificInterface are concrete products that implement the OS-specific operations.

This pattern allows the Mobile App Interface to support multiple types of mobile operating systems by delegating the creation of OS-specific interfaces to its subclasses. It provides flexibility to add support for new operating systems in the future without modifying existing code, thus fulfilling the requirement effectively.

# References (4 or More References in APA Format)

1. Refactoring.Guru. (n.d.). Design Patterns. <https://refactoring.guru/design-patterns>
2. Tutorials Point. (n.d.a). Design patterns – Adapter pattern. <https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm>
3. Tutorials Point. (n.d.b). Design patterns – Factory pattern. <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>
4. Tutorials Point. (n.d.c). Design patterns – Observer pattern. <https://www.tutorialspoint.com/design_pattern/observer_pattern.htm>