

**Mohammad Usama**

**20K-0190**

**Design Defects And Refactoring**

**Sec 8-A**

**Assignment 02**

**Date of submission: 26/02/2024**

**Task A**

### **Design Principle: Encapsulate What Varies**

**Scenario:**

In software development, especially in large projects, code tends to change frequently due to evolving requirements, bug fixes, or enhancements. However, frequent changes can lead to code becoming tangled and difficult to maintain. The principle of "Encapsulate What Varies" aims to address this issue by isolating the parts of the code that are prone to change.

**Explanation:**

1. **Objective**: The objective of applying the "Encapsulate What Varies" principle is to reduce the impact of frequent changes on the overall codebase and improve its maintainability.
2. **Technique**: The technique involves identifying areas of the code that are subject to change and encapsulating them behind interfaces or abstractions. By doing so, the rest of the codebase remains unaffected by changes in these encapsulated components.
3. **Example**: In the provided example, the principle is applied to a checkout process in a library management system. The original code checks various conditions directly in the checkoutBook function, making it hard to understand and prone to changes if checkout requirements evolve.
4. **Benefits**: By encapsulating what varies, the code becomes more modular, easier to understand, and less prone to unintended consequences when changes occur. It also promotes code reuse and facilitates testing.

**Task B**

### **Abstract Factory Pattern**

**Scenario:**

In software design, especially when dealing with object creation, the choice of object creation mechanism can have a significant impact on the flexibility and maintainability of the code. The Abstract Factory pattern addresses the need to create families of related objects without specifying their concrete classes.

**Explanation:**

1. **Objective**: The objective of using the Abstract Factory pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern enables the creation of objects without exposing the instantiation logic to the client.
2. **Use Case**: In the provided example, the scenario involves creating different types of furniture - chairs and sofas - in two distinct styles: modern and Victorian.
3. **Structure**: The Abstract Factory pattern involves defining abstract factories for each family of related products (chairs and sofas in this case). Concrete factory implementations are then responsible for creating specific product instances (modern or Victorian chairs and sofas).
4. **Benefits**: By employing the Abstract Factory pattern, the codebase becomes more modular, extensible, and maintainable. It promotes the separation of concerns and facilitates the creation of families of related objects in a consistent manner.