Department of Computer Engineering

Academic Term: First Term 2023-24

Class: T.E /Computer Sem – V / Software Engineering

Practical No:	**7**
Title:	**Design using Object Oriented approach with emphasis on Cohesion and Coupling**
Date of Performance:	**11-09-2023**
Roll No:	**9587**
Team Member:	**Sania Almeida**

Sr. No	Performance Indicator	Excellent	Good	Below Average	Total Score
1	On time Completion & Submission (01)	01 (On Time)	NA	00 (Not on Time)	
2	Theory Understanding(02)	02(Correct)	NA	01 (Tried)	
3	Content Quality (03)	03(All used)	02 (Partial)	01 (rarely followed)	
4	Post Lab Questions (04)	04(done well)	3 (Partially Correct)	2(submitted)	

Signature of the Teacher:

Lab Experiment 07

Experiment Name: Design Using Object-Oriented Approach with Emphasis on Cohesion and Coupling in Software Engineering

Objective: The objective of this lab experiment is to introduce students to the Object-Oriented (OO) approach in software design, focusing on the principles of cohesion and coupling. Students will gain practical experience in designing a sample software project using OO principles to achieve high cohesion and low coupling, promoting maintainable and flexible software.

Introduction: The Object-Oriented approach is a powerful paradigm in software design, emphasizing the organization of code into objects, classes, and interactions. Cohesion and Coupling are essential design principles that guide the creation of well-structured and modular software.

Lab Experiment Overview:

1. Introduction to Object-Oriented Design: The lab session begins with an introduction to the Object Oriented approach, explaining the concepts of classes, objects, inheritance, polymorphism, and encapsulation.

2. Defining the Sample Project: Students are provided with a sample software project that requires design and implementation. The project may involve multiple modules or functionalities. 3. Cohesion in Design: Students learn about Cohesion, the degree to which elements within a module or class belong together. They understand the different types of cohesion, such as functional, sequential, communicational, and temporal, and how to achieve high cohesion in their design. 4. Coupling in Design: Students explore Coupling, the degree of interdependence between modules or classes. They understand the types of coupling, such as content, common, control, and stamp coupling, and strive for low coupling in their design.

5. Applying OO Principles: Using the Object-Oriented approach, students design classes and identify their attributes, methods, and interactions. They ensure that classes have high cohesion and are loosely coupled.
6. Class Diagrams: Students create Class Diagrams to visually represent their design, illustrating the relationships between classes and their attributes and methods.
7. Design Review: Students conduct a design review session, where they present their Class Diagrams and receive feedback from their peers.
8. Conclusion and Reflection: Students discuss the significance of Object-Oriented Design principles, Cohesion, and Coupling in creating maintainable and flexible software. They reflect on their experience in applying these principles during the design process.

Learning Outcomes: By the end of this lab experiment, students are expected to:

- Understand the Object-Oriented approach and its core principles, such as encapsulation, inheritance, and polymorphism.
- Gain practical experience in designing software using OO principles with an emphasis on Cohesion and Coupling.

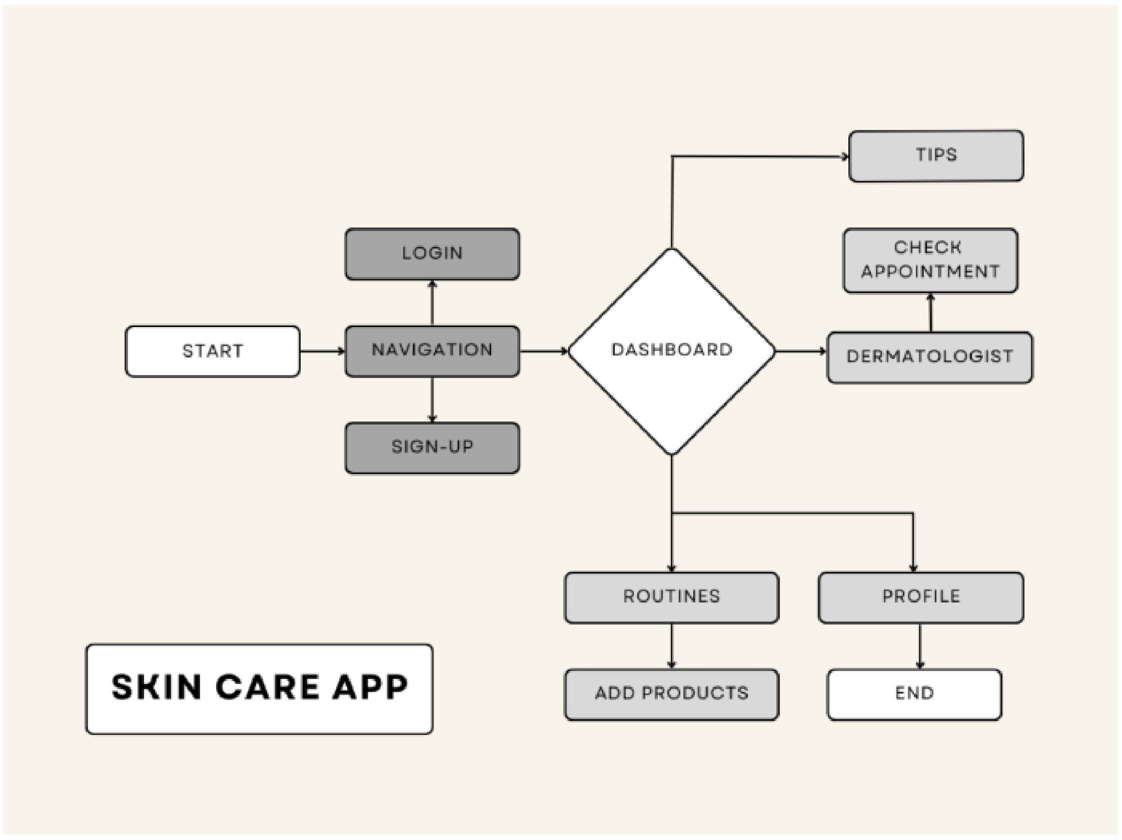
- Learn to identify and implement high cohesion and low coupling in their design, promoting modular and maintainable code.
- Develop skills in creating Class Diagrams to visualize the relationships between classes. • Appreciate the importance of design principles in creating robust and adaptable software.

Pre-Lab Preparations: Before the lab session, students should review Object-Oriented concepts, such as classes, objects, inheritance, and polymorphism. They should also familiarize themselves with the principles of Cohesion and Coupling in software design.

Materials and Resources:

- Project brief and details for the sample software project
- Whiteboard or projector for creating Class Diagrams
- Drawing tools or software for visualizing the design

Conclusion: The lab experiment on designing software using the Object-Oriented approach with a focus on Cohesion and Coupling provides students with essential skills in creating well-structured and maintainable software. By applying OO principles and ensuring high cohesion and low coupling, students design flexible and reusable code, facilitating future changes and enhancements. The experience in creating Class Diagrams enhances their ability to visualize and communicate their design effectively. The lab experiment encourages students to adopt design best practices, promoting modular and efficient software development in their future projects. Emphasizing Cohesion and Coupling in the Object-Oriented approach empowers students to create high-quality software that meets user requirements and adapts to evolving needs with ease.



The provided information outlines some of the shortcomings and potential upgrades for Skin Care App. Let's break down the shortcomings and possible upgrades:

Shortcomings of the Skin Care App:

1. Limited User Engagement: The app relies heavily on passive content consumption, lacking interactive elements beyond questionnaires.

2. Lack of Personalization: The app does not offer highly personalized recommendations based on the user's skin type, concerns, and history.

3. Missing Social Integration: It lacks features for users to share their routines and tips with others, reducing community engagement.

4. Limited Appointment Management: The dermatologist page only allows checking appointments but doesn't offer appointment scheduling or reminders.

5. Minimal User Feedback: The app doesn't facilitate user feedback, which is essential for improving its content and features.

Upgrades for the Future:

1. Enhanced Interactivity: Introduce interactive elements like quizzes, interactive tutorials, and community forums to boost user engagement.

2. Personalization Engine: Implement an AI-driven system to provide personalized skincare routines and tips based on user profiles and skin analysis.

3. Social Sharing: Allow users to share their skincare routines and tips, fostering a sense of community and knowledge exchange.

4. Appointment Booking: Add a feature for users to schedule and manage dermatologist appointments within the app, with reminders and virtual consultations.

5. Feedback Mechanism: Incorporate a feedback system to collect user input and reviews for continuous improvement.

6. AI Chatbot: Integrate a chatbot to provide real-time assistance, answer user queries, and guide them through their skincare journey.

7. Visual Skin Analysis: Enable users to upload images for skin analysis, helping them track progress and receive tailored recommendations.

POSTLABS:

a. Analyze a given software design and assess the level of cohesion and coupling, identifying potential areas for improvement.

To analyze the level of cohesion and coupling in a given software design and identify potential areas for improvement, we need to understand what cohesion and coupling mean in the context of software design.

Cohesion refers to how closely related the components (modules, classes, functions, etc.) within a software system are to each other in terms of their responsibilities and functionality. High cohesion means that elements within a module or component are tightly related and work together for a common purpose. Low cohesion suggests that the elements within a module have unrelated or loosely related responsibilities.

Coupling refers to the degree of dependence between different modules or components within a software system. Low coupling means that modules interact with each other minimally, while high coupling implies a significant level of interdependence between modules.

Here's how to assess and improve cohesion and coupling:

Assessing Cohesion:

1. Functional Cohesion (High): Components should have a single, well-defined purpose. If you find that components are mixing unrelated functionality, this indicates low cohesion. Consider breaking them into smaller, more focused components.

2. Sequential Cohesion (Low): If a module contains multiple functions or procedures that must be executed in a specific sequence, this indicates low cohesion. Consider separating the functions into distinct components to improve modularity.

3. Communicational Cohesion (Medium): If functions within a module work on the same data, but each function deals with a different subset of that data, you may have a case of communicational cohesion. This is not necessarily a problem but can be improved by organizing functions that work on related data into separate components.

4. Procedural Cohesion (Medium): If a module groups functions or procedures based on their execution order, even though their purposes may be unrelated, it exhibits procedural cohesion. You can improve this by organizing functions based on their related functionality.

Assessing Coupling:

1. Low Coupling (Good): A software system with low coupling isolates components, making them independent of each other. Communication between modules should be through well-defined interfaces. If there's a high degree of coupling, you might consider redesigning the system to reduce interdependence.

2. Content Coupling (High): Content coupling occurs when one component accesses the internal data of another. To reduce content coupling, use well-defined interfaces and encapsulation to hide the internal details of a component.

3. Common Coupling (Medium): Common coupling involves multiple components accessing the same shared data. While this is not necessarily problematic, ensure that shared data is handled safely to prevent data corruption and conflicts.

4. Control Coupling (Medium): Control coupling happens when one component influences the behavior of another through control parameters or flags. Reduce control coupling by using well-structured and well-documented interfaces.

Improvement Strategies:

1. Refactoring: If you find components with low cohesion, refactor them into smaller, more focused components, each with a single, well-defined purpose.
2. Encapsulation: Encapsulate data within components to reduce content coupling. Only expose necessary interfaces to other components.
3. Define Clear Interfaces: Ensure that components communicate through well-defined and documented interfaces, reducing the chance of unexpected dependencies.
4. Modularization: Break down the software into smaller, independent modules that can be developed, tested, and maintained separately.
5. Dependency Injection: Use dependency injection or inversion of control to decouple components and make them more testable and maintainable.
6. Use Design Patterns: Implement design patterns like the Singleton pattern to control the instantiation of classes or the Observer pattern for communication between components in a more decoupled manner.

The specific recommendations for improvement will depend on the details of the software design under consideration. The goal is to balance cohesion and coupling to achieve a design that is both modular and maintainable.

b. Apply Object-Oriented principles, such as encapsulation and inheritance, to design a class hierarchy for a specific problem domain.

Design a simple class hierarchy for a problem domain related to "Vehicles." In this hierarchy, we'll use Object-Oriented principles like encapsulation and inheritance to create a structured representation of various types of vehicles.

Class Hierarchy:

1. Vehicle (Base Class):
   - This is the top-level class representing all vehicles. It includes common attributes and methods that are applicable to all vehicles.

```python
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start(self):
        pass  # Method to start the vehicle

    def stop(self):
        pass  # Method to stop the vehicle
```

2. Car (Inherits from Vehicle):
   - This class represents specific types of vehicles, like cars. It inherits common attributes and methods from the `Vehicle` class but may have additional attributes and methods unique to cars.

```python
class Car(Vehicle):
    def __init__(self, make, model, year, num_doors):
        super().__init__(make, model, year)
        self.num_doors = num_doors

    def honk_horn(self):
        pass  # Method to honk the car's horn
```

3. Motorcycle (Inherits from Vehicle):
   - Similar to the `Car` class, the `Motorcycle` class represents a specific type of vehicle. It inherits attributes and methods from the `Vehicle` class but may have unique characteristics for motorcycles.

```python
class Motorcycle(Vehicle):
    def __init__(self, make, model, year, engine_size):
        super().__init__(make, model, year)
        self.engine_size = engine_size

    def wheelie(self):
        pass  # Method to perform a wheelie (example method for motorcycles)
```

In this class hierarchy:

- The `Vehicle` class is the base class, encapsulating common attributes (make, model, year) and methods (start, stop).
- The `Car` and `Motorcycle` classes inherit from `Vehicle`, inheriting these common attributes and methods while adding their specific attributes and methods.
- Encapsulation is achieved by making attributes private (by convention, by prefixing them with an underscore) and providing methods to access and manipulate the attributes. For example, you could add getter and setter methods for attributes as needed.

This class hierarchy demonstrates encapsulation and inheritance, two fundamental principles of Object-Oriented Programming (OOP). It provides a structured way to represent vehicles in a program, making it easy to create instances of different types of vehicles and call their respective methods.

c. Evaluate the impact of cohesion and coupling on software maintenance, extensibility, and reusability in a real-world project scenario.

The concepts of cohesion and coupling have a significant impact on software maintenance, extensibility, and reusability in real-world project scenarios. Let's evaluate these impacts in more detail:

Cohesion:
Cohesion refers to how closely related the components within a software system are to each other in terms of their responsibilities and functionality.
- Impact on Maintenance:
  - High cohesion leads to code that is easier to maintain. When related functionality is grouped together in a cohesive manner, it's easier to understand, update, and fix issues within a module or component.
  - Low cohesion can make maintenance challenging, as unrelated or loosely related functions are scattered across the codebase. Changes to one part of the code might have unintended consequences in other parts.
- Impact on Extensibility:
  - High cohesion simplifies extensibility. When a module is well-organized and its responsibilities are clear, adding new features or functionality is more straightforward.
  - Low cohesion can hinder extensibility, as it may require refactoring or reorganizing the code to accommodate new features, potentially introducing new defects.
- Impact on Reusability:
  - High cohesion enhances reusability. Well-structured, cohesive modules can be easily reused in other parts of the project or in different projects.
  - Low cohesion reduces reusability, as it may be difficult to extract and reuse parts of a module without carrying along unnecessary dependencies or functionality.

Coupling:

Coupling refers to the degree of dependence between different modules or components within a software system.

- Impact on Maintenance:
  - Low coupling makes maintenance easier. When modules are loosely connected, changes in one module are less likely to affect others, reducing the risk of unintended consequences.
  - High coupling can complicate maintenance. Modifications to one module may necessitate changes in multiple other modules, increasing the chances of introducing defects.
- Impact on Extensibility:
  - Low coupling simplifies extensibility. New features or components can be added with minimal impact on existing modules, promoting a modular and flexible architecture.
  - High coupling can impede extensibility. Introducing new features may require extensive changes across the codebase, increasing the risk of regression issues.
- Impact on Reusability:
  - Low coupling enhances reusability. Loosely coupled modules can be easily extracted and reused in other projects or within the same project without dragging along a web of dependencies.
  - High coupling reduces reusability, as tightly interconnected modules are often difficult to reuse independently.

Real-world Scenario:

Consider a real-world scenario of a legacy e-commerce platform:

- Cohesion:
  - Low cohesion: In the legacy code, the shopping cart logic is scattered across multiple modules with unrelated responsibilities. Maintenance is challenging, and introducing new payment methods is difficult.
  - High cohesion: After refactoring, the shopping cart functionality is grouped into a single cohesive module. Maintenance becomes easier, and adding new features like multiple shipping addresses becomes more straightforward.

- Coupling:
  - High coupling: The legacy code exhibits tight coupling between the user authentication module and the product recommendation module. Changing the authentication system could impact the recommendations and lead to unexpected behavior.
  - Low coupling: In the refactored version, the authentication and recommendation modules are decoupled. This allows for more flexible changes to the authentication system without affecting recommendations.