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:	**9539**
Team Members:	**Crystal Fernandes**

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 gainpractical 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. Cohesionin 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. Theyunderstand 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 therelationships between classes and their attributes and methods.
7. Design Review: Students conduct a design review session, where they present their ClassDiagrams 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 onCohesion and Coupling.

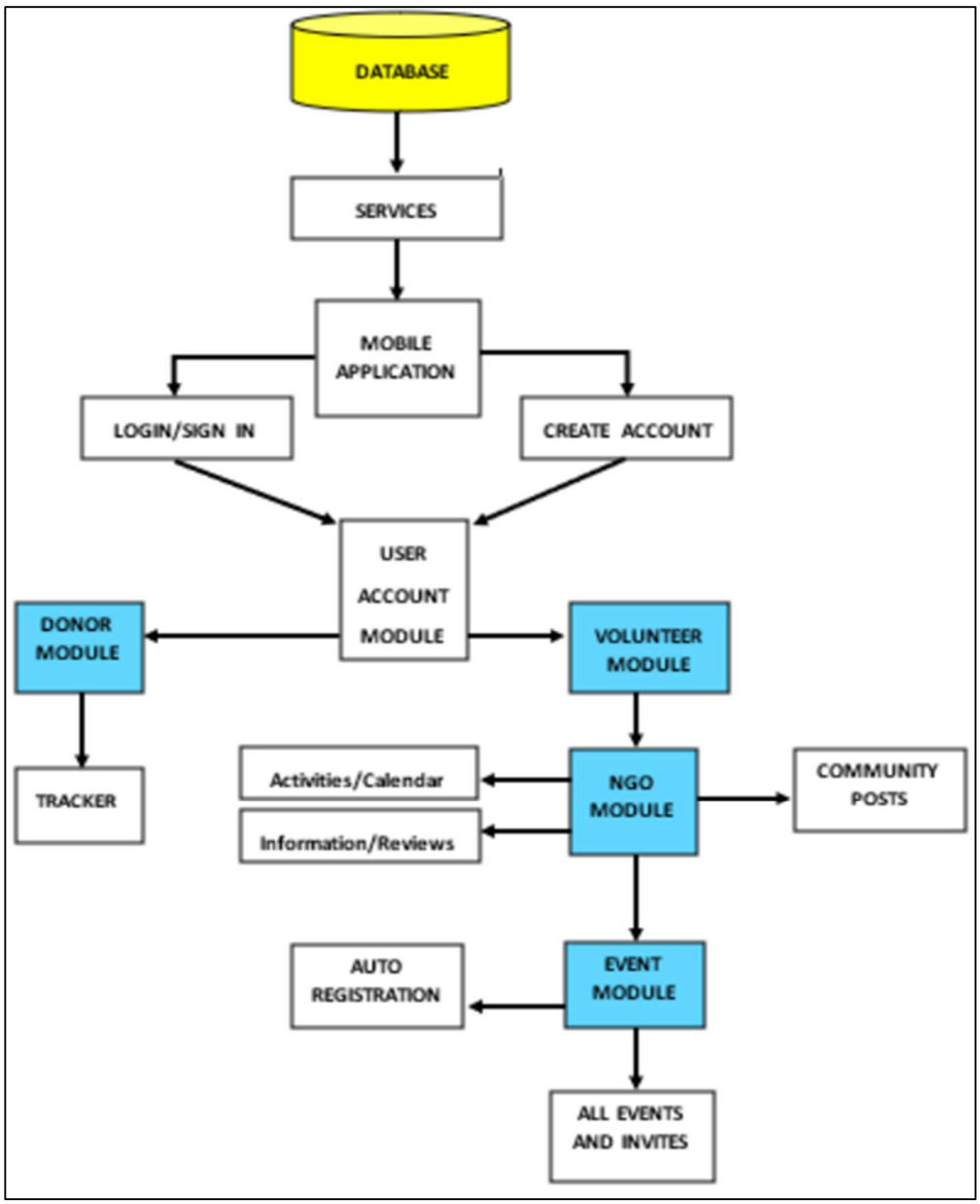
- Learn to identify and implement high cohesion and low coupling in their design, promotingmodular and maintainable code.
- Develop skills in creating Class Diagrams to visualize the relationships between classes. · Appreciate the importance of design principles in creating robust and adaptable software.

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 smart waste management systems. Let's break down the shortcomings and possible upgrades:

Shortcomings:

- Limited Geographic Reach: The platform's effectiveness could be limited if it primarily caters to a specific region or country. Expanding its reach globally might be challenging due to differences in languages, regulations, and NGO structures.
- Reliance on Internet Access: The platform relies on internet connectivity, which may not be universally available. Users in remote or underserved areas may struggle to access and use the service.
- Data Accuracy: The success of your matching algorithm heavily depends on the accuracy and completeness of user data. If users provide incorrect information or neglect to update their profiles, it could affect the quality of matches.
- Scalability: As the platform gains popularity and the user base grows, scalability issues might arise.
- Underrepresented NGOs: Smaller or less tech-savvy NGOs may not participate, limiting the platform's ability to connect volunteers with a wide range of organizations.
- NGO Verification: Verifying the legitimacy and credibility of NGOs can be challenging. Without proper

verification, there's a risk of fraudulent or poorly managed organizations being listed on the platform.

- Volunteer Commitment: Volunteers may not always fulfill their commitments to attend events, leading to wasted NGO resources and potential disruptions in event planning.
- Algorithm Efficiency: The matching algorithm's efficiency is crucial. If it takes too long to find matches or if it's inefficient in handling a large number of events and volunteers, users may become frustrated.
- Cost and Funding: Maintaining the platform, especially with additional features and scalability, can be expensive. Finding sustainable funding sources may be a significant challenge.

Upgrades:

- Machine Learning Matching: Implement machine learning algorithms to improve volunteer-NGO matching. The system could learn from user preferences and feedback to make more accurate recommendations over time.
- Language Support: Add multilingual support to facilitate communication and usage for a more diverse user base.
- NGO Verification: Introduce a verification process for NGOs. Verified NGOs could receive a badge or special status, indicating their credibility.
- Premium Features: Offer premium features for NGOs and volunteers willing to pay. This could include advanced analytics, priority matching, and enhanced visibility for NGOs.
- Volunteer Training: Provide resources and training materials for volunteers, especially if they are participating in specialized events (e.g., disaster relief).
- NGO Management Tools: Develop tools for NGOs to manage volunteers efficiently. This could include event scheduling, volunteer tracking, and communication features.
- API for Partnerships: An API that allows other organizations or platforms to integrate with NGO Finder. For example, a travel booking platform could suggest volunteering opportunities based on users' travel plans.
- Privacy and Data Security: Invest in robust data protection and privacy measures. User data should be handled with the utmost care.
- NGO Reporting Tools: Develop tools that help NGOs generate reports on the impact of their events. This can be used for accountability and transparency.
- Real-time Chat: Implement real-time chat features for better communication between volunteers and NGOs.
- Community Events: Host community events or webinars related to volunteering, social causes, and nonprofit work. This can help engage and educate users.
- Virtual Volunteering: Include opportunities for virtual volunteering, such as online tutoring or remote work, to accommodate a broader range of volunteers.
- Gamification: Introduce gamification elements to motivate and reward volunteers for their contributions.
- Crowdsourcing Features: Enable NGOs to crowdsource funds, resources, or ideas from the volunteer community.

POSTLABS:

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

Problem Domain: E-commerce Product Management

Cohesion Analysis:

- Functional Cohesion: Examine how functions and modules are organized around specific tasks. For example, consider how product search, shopping cart management, and user registration functionalities are structured. Ensure that related functions are grouped together for ease of maintenance.
- Sequential Cohesion: Check if there are dependencies requiring sequential execution. For instance, analyze if the ordering and payment processes are tightly coupled and if this can be made more modular.
- Communicational Cohesion: Evaluate components that frequently exchange data, like user profiles and order histories. Ensure these are appropriately organized and that data sharing is well-managed.
- Temporal Cohesion: Assess if components that are activated or used during the same period are grouped together. For example, consider cart management during shopping and checkout. Ensure these are logically organized.
- Procedural Cohesion: Analyze if components related to similar algorithms or procedures are well-structured. For example, payment processing or product recommendation algorithms. Ensure these are modular and reusable.

Coupling Analysis:

- Data Coupling: Check how different modules or components share data. For example, how shopping carts share data with order processing. Minimize excessive data sharing and promote well-defined data interfaces.
- Stamp Coupling: Determine if there are areas where multiple components rely on composite data structures. Identify if these can be decoupled by improving the definition of data interfaces.
- Control Coupling: Examine if there are dependencies where one module directly controls or influences the behavior of another. For instance, how the checkout process controls inventory management. Minimize control coupling for independent modules.
- Content Coupling: Identify if there are components that rely on the internal details of others. Encourage using well-defined interfaces and abstractions to reduce content coupling.
- Common Coupling: Look for modules that share global variables or resources. Address these to limit common coupling through encapsulation.

Improvement Recommendations:

- Recommend the use of design patterns, such as the Model-View-Controller (MVC) pattern, to enhance separation of concerns and reduce coupling in the user interface.
- Propose the application of principles like SOLID, DRY (Don't Repeat Yourself), and KISS (Keep It Simple, Stupid) to promote clean and maintainable design.

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

Problem Domain: E-commerce Product Management

In this problem domain, we want to design a class hierarchy to manage different types of products sold on an e-commerce website. We'll focus on two key Object-Oriented principles: encapsulation and inheritance.

1. Encapsulation:

Encapsulation involves bundling the data (attributes) and methods (functions) that operate on the data into a single unit, which is called a class. It also restricts direct access to some of an object's components. Encapsulation helps in controlling the access to the internal state of an object and ensures data integrity.

Class Hierarchy for Products:
Let's design a class hierarchy for different types of products:

Product (Base Class):
Attributes: name, price, description, stock_quantity
Methods: get_info(), update_stock()
Electronics (Inherits from Product):

Additional Attributes: brand, model, warranty_period
Additional Methods: set_warranty()
Clothing (Inherits from Product):

Additional Attributes: size, color, material
Additional Methods: get_size()
Books (Inherits from Product):

Additional Attributes: author, genre, publication_date
Additional Methods: get_author()
Encapsulation Usage:

Each product type (Electronics, Clothing, Books) encapsulates its specific attributes and methods. For instance, Electronics class encapsulates electronic product details, and Clothing class encapsulates clothing product details.

2. Inheritance:
Inheritance is a mechanism by which one class can inherit the properties (attributes and methods) of another class. It promotes code reusability and the creation of a hierarchy of classes.

Class Hierarchy for Products (Inheritance):

In our class hierarchy:
The base class Product defines common attributes and methods shared by all products.
The derived classes Electronics, Clothing, and Books inherit the attributes and methods from the Product class and add their specific attributes and methods.
For example, Electronics inherits name, price, description, and methods like get_info() from Product.

Encapsulation and Inheritance in Action:
Encapsulation ensures that attributes and methods related to a specific product type are encapsulated within that product's class, preventing direct access to unrelated attributes.
Inheritance allows the reuse of common attributes and methods from the base class (Product) by the derived classes (Electronics, Clothing, Books), promoting code reuse.

c) Evaluate the impact of cohesion and coupling on software maintenance, extensibility, and reusability in a real-world project scenario.
Scenario: E-commerce Website Development

1. Cohesion:

a. High Cohesion:
In the product management module of your e-commerce website, high cohesion means that each class or module is focused on a specific, well-defined task.

Impact:
Software Maintenance: High cohesion simplifies maintenance. If you need to update product-related features, you know exactly where to make changes without affecting other parts of the system.
Extensibility: It's easier to add new features related to products because you can make changes in a cohesive module without disrupting unrelated functionality.
Reusability: Well-cohesive classes can be reused in other parts of the system or even in other projects because they are self-contained and focused.

b. Low Cohesion:
Low cohesion implies that a class or module has multiple, unrelated responsibilities.
Impact:
Software Maintenance: Low cohesion leads to maintenance challenges. Changes in one part of the module may unintentionally affect other parts, increasing the risk of introducing bugs.
Extensibility: Extending a module with low cohesion can be complex. New features might interact unpredictably with existing code.
Reusability: Low-cohesive modules are less likely to be reusable in other projects because they are tightly coupled to their original context.

2. Coupling:

a. Loose Coupling:
Loose coupling means that different modules or classes have minimal dependencies on each other. They interact through well-defined interfaces.
Impact:
Software Maintenance: Loose coupling simplifies maintenance. Changing one module is less likely to affect others, reducing the risk of unintended side effects.
Extensibility: New features can be added with less risk of interfering with existing functionality because of the low interdependence between modules.
Reusability: Loosely coupled modules are highly reusable because they can be integrated with minimal effort into other projects.

b. Tight Coupling:
Tight coupling indicates that modules or classes are highly dependent on each other, often through direct references or shared data.
Impact:
Software Maintenance: Tight coupling complicates maintenance. Modifications in one module can propagate issues to dependent modules, leading to more extensive testing and debugging.
Extensibility: Adding new features in a tightly coupled system can be challenging because you must consider how changes impact interconnected components.
Reusability: Tightly coupled modules are less reusable in other contexts due to their strong dependencies on specific conditions.

In the context of the e-commerce website project:
High cohesion and loose coupling lead to more maintainable, extensible, and reusable code.
Low cohesion and tight coupling result in complex maintenance, challenges in adding new features, and limited reusability.