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:	07-09-23
Roll No:	9596
Team Members:	Reanne Dcosta, Atharva Dalvi, Nicole Falcao

Rubrics for Evaluation:

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.

Step 1: Introduction to Object-Oriented Design

Let's introduce the core Object-Oriented

concepts:

```
class Animal:
   def_init_(self, name):
      self.name = name

   def
      speak(self)
      : pass

class
   Dog(Animal):
   def
   speak(self):
      return f"{self.name} says Woof!"

class
   Cat(Animal):
   def
   speak(self):
      return f"{self.name} says Meow!"

dog =
Dog("Buddy") cat =
Cat("Whiskers")
animals = [dog, cat]

for animal in

animals:

   print(animal.speak())
```

Step 2: Defining the Sample Project

For this step, let's define a simple library management system project as an example:

```
class Book:
   def_init_(self, title, author):
      self.title = title
      self.author = author
```

```
class Patron:
    def_init_(self, name):
        self.name = name

class Library:
    def_init_(self):
        self.books = []
        self.patrons =
        []
    def add_book(self, book):
        self.books.append(book)

    def add_patron(self, patron):
        self.patrons.append(patron)
```

Step 3: Cohesion in Design

In this step, we emphasize cohesion in the Library class:

```
class Library:
    def_init_(self):
        self.books = []
        self.patrons =
        []

    def add_book(self, book):
        self.books.append(book)

    def add_patron(self, patron):
        self.patrons.append(patron)
```

Step 4: Coupling in Design

We strive for low coupling by ensuring that Library doesn't need to know the internal details of Book and Patron:

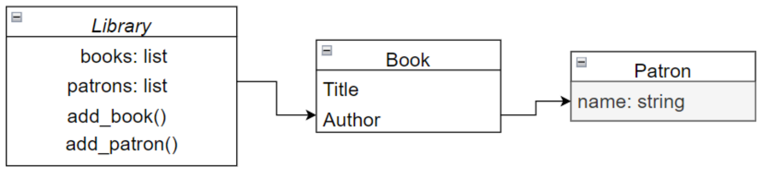
```
class Library:
    def_init_(self):
        self.books = []
        self.patrons =
        []

    def add_book(self, book):
        self.books.append(book)

    def add_patron(self, patron):
        self.patrons.append(patron)
```

Step 5: Applying OO Principles

We design classes with high cohesion and low coupling:

```
# Library
System class
Book:
    def_init_(self, title, author):
        self.title = title
        self.author = author

class Patron:
    def_init_(self, name):
        self.name = name

class Library:
    def_init_(self):
        self.books = []
        self.patrons =
        []

    def add_book(self, book):
        self.books.append(book)

    def add_patron(self, patron):
        self.patrons.append(patron)
```

Step 6: Class Diagrams



Step 7: Design Review

In this stage, students would present their class diagrams and design to their peers for review.
Design Review Meeting: Library Management System

Presenters: Alice and Bob

Reviewers: Carol and Dave

Agenda:

Presentation of Class Diagrams.
Discussion on Cohesion and Coupling.
Feedback and Suggestions.
Presentation of Class Diagrams:

Alice:

"Here's the class diagram for our library management system. We have three main classes: Library, Book, and Patron."
"The Library class has two lists, one for books and one for patrons."
"We use composition arrows to show that Library contains instances of Book and Patron classes."
"We have methods in Library for adding books and patrons."
Bob:

"In the Book class, we've included attributes like title and author to store book information."
"The Patron class has name as the primary attribute."
"We tried to maintain high cohesion within each class and keep the coupling between classes minimal."
Discussion on Cohesion and Coupling:

Carol:

"I see that you've used composition to model the relationship between Library and Book and Patron. This seems appropriate, as it shows that a library contains books and patrons."
"It's good to see that the responsibilities of each class are well-defined. For instance, the Library class is responsible for managing books and patrons, which is high cohesion."
"The coupling between classes is also low, which is a positive aspect. Classes don't need to know much about each other."
Dave:

"I agree with Carol. Your design seems well-organized and modular. However, I noticed that you haven't included any methods or attributes for book checkout or return. Is that a deliberate choice?"
Feedback and Suggestions:

Alice:

"That's a good point, Dave. We should probably add methods to handle book checkout and return. It's an essential functionality of a library."
Bob:

"I also think we can add more attributes to the Book class, like ISBN and availability. It would be helpful to know if a book is available or checked out."
Carol:

"Overall, your design looks good. Just make sure to continuously refine it as you work on the project. It's a good start!"

Conclusion:
Alice and Bob will make the suggested improvements to their design based on the feedback.
The review session highlighted the importance of maintaining high cohesion and low coupling in the design.
The students gained valuable insights into the design process and received constructive feedback.

POSTLAB:

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

Assessing the level of cohesion and coupling in a software design is a crucial step in ensuring that the software is maintainable, scalable, and robust. Cohesion measures how closely related the elements within a module or component are, while coupling measures the level of interdependence between different modules or components.

1. *Understand the Software Design*: Start by thoroughly understanding the software design. This includes reviewing the architecture, component relationships, and how data and control flow through the system. You need a clear picture of how the various modules or components are organized.

2. *Identify Modules or Components*: Break down the software into modules or components. These can be classes, functions, or larger architectural components.

3. *Analyze Cohesion*:

- *Functional Cohesion*: Assess whether each module or component has a single, well-defined responsibility. Modules should do one thing and do it well. If a module is responsible for multiple tasks, it exhibits low cohesion. Suggest splitting or refactoring such modules into smaller, more focused units.
- *Sequential Cohesion*: Check if modules are dependent on the output of other modules in a sequential order. High sequential cohesion can lead to rigidity and may make the software harder to change. Consider breaking dependencies and making modules more independent.

4. *Analyze Coupling*:

- *Content Coupling*: Examine if modules directly access each other's internal data or variables. Tight coupling can lead to maintenance challenges and make it difficult to replace or modify components. Encourage using interfaces or abstractions to reduce content coupling.
- *Common Coupling*: Determine if multiple modules share a global data structure. This can create unintended dependencies and affect the software's reliability. Encourage encapsulation of shared data or the use of messaging or event-driven mechanisms to reduce common coupling.
- *Control Coupling*: Check if modules influence the control flow of other modules (e.g., by altering the sequence of execution). High control coupling can lead to unpredictable behavior. Encourage clearly defined interfaces and minimize the use of global variables and function calls that alter control flow.

5. *Identify Potential Areas for Improvement*:

- Prioritize areas with low cohesion and high coupling for improvement. Consider breaking down monolithic modules into smaller, more focused ones.
- Encourage the use of design patterns, such as the SOLID principles, to improve cohesion and reduce coupling.
- Promote encapsulation and information hiding to minimize content coupling.
- Use dependency injection and interfaces to reduce the impact of changes in one module on others, reducing coupling.
- Ensure that communication between modules is well-defined and follows a consistent pattern, reducing control coupling.

6. *Testing*: After making changes to improve cohesion and reduce coupling, conduct thorough testing to ensure that the software still functions as expected and that the changes have not introduced new issues.

7. *Documentation*: Update documentation to reflect the changes and the rationale behind them.

Improving cohesion and reducing coupling can lead to a more maintainable, scalable, and adaptable software system, making it easier to extend, modify, and troubleshoot. It's an ongoing process that should be integrated into the software development lifecycle.

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

Designing a class hierarchy using Object-Oriented principles like encapsulation and inheritance is a fundamental part of creating well-structured and maintainable software. Let's design a class hierarchy for a problem domain related to "Shapes" to illustrate how these principles can be applied.

Problem Domain: Shapes

In this problem domain, we'll represent different geometric shapes, such as circles, rectangles, and triangles.

1. *Shape (Base Class)*
   - This is the base class that will define common properties and methods for all shapes.
   - Common properties: `color`, `filled` (whether the shape is filled with color or not).
   - Methods: `getArea()` (to calculate the area of the shape), `getPerimeter()` (to calculate the perimeter of the shape), and getters and setters for color and filled.

2. *Circle (Derived from Shape)*
   - Inherits from the `Shape` class.
   - Additional property: `radius`.
   - Methods:
     - `getArea()` (override to calculate the area of a circle).
     - `getPerimeter()` (override to calculate the circumference).

3. *Rectangle (Derived from Shape)*
   - Inherits from the `Shape` class.
   - Additional properties: `width` and `height`.
   - Methods:
     - `getArea()` (override to calculate the area of a rectangle).
     - `getPerimeter()` (override to calculate the perimeter of a rectangle).

4. *Triangle (Derived from Shape)*
   - Inherits from the `Shape` class.
   - Additional properties: `side1`, `side2`, and `side3` (representing the lengths of the triangle's sides).
   - Methods:
     - `getArea()` (override to calculate the area of a triangle).

- `getPerimeter()` (override to calculate the perimeter of a triangle).

Encapsulation:
- Encapsulation is achieved by making the instance variables (properties) of each class private and providing public getters and setters for these properties. For example, you would provide public methods in each class to set and get color and filled status.

Inheritance:
- Inheritance is utilized to create specialized classes (Circle, Rectangle, Triangle) that inherit common properties and methods from the base class (Shape). This promotes code reusability and helps organize the classes in a hierarchical structure.

Here's a simplified Python example to demonstrate the class hierarchy:

```
class Shape:
    def_init_(self, color, filled): self.
        color = color
        self._filled = filled

    def getArea(self):
        pass

    def getPerimeter(self):
        pass

    def getColor(self):
        return self._color

    def setColor(self, color):
        self._color = color

    def isFilled(self):
        return self._filled

    def setFilled(self, filled):
        self._filled = filled

class Circle(Shape):
    def_init_(self, color, filled, radius):
        super()._init_(color, filled)
        self._radius = radius

    def getArea(self):
        return 3.14 * self._radius * self.__radius

    def getPerimeter(self):
        return 2 * 3.14 * self._radius

class Rectangle(Shape):
    def_init_(self, color, filled, width, height): super().
        init_(color, filled)
        self._width = width
        self._height = height

    def getArea(self):
        return self._width * self.__height

    def getPerimeter(self):
        return 2 * (self._width + self.__height)

class Triangle(Shape):
    def_init_(self,  color,  filled,  side1,  side2,  side3):
        super()._init_(color, filled)
        self._side1  =  side1
        self._side2  =  side2
        self._side3 = side3

    def getArea(self):
        # Implement area calculation for a triangle
        pass

    def getPerimeter(self):
        return self._side1 + self.__side2 + self.__side3
```

This class hierarchy demonstrates encapsulation and inheritance to model different shapes in an object-oriented way. Additional methods can be implemented as needed for more complex calculations and functionality.

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

Cohesion and coupling play crucial roles in determining the maintainability, extensibility, and reusability of software in a real-world project. Let's evaluate their impact in a real-world scenario:

1. Impact on Maintainability:

Cohesion: High cohesion within a module or class makes it easier to understand, modify, and maintain that module. Code with high cohesion is typically focused on a single responsibility, and changes to that responsibility affect only a limited portion of the codebase. This results in easier debugging and maintenance.

Coupling: Low coupling between modules or classes means they are less dependent on each other. When one module needs to be modified, it has a minimal impact on other modules. This reduces the risk of introducing unexpected side effects and makes it easier to maintain the software.

2. Impact on Extensibility:

Cohesion: High cohesion can enhance extensibility. When a module is designed with a clear, single responsibility, adding new features or functionalities related to that responsibility is straightforward. Developers can work on separate modules with a clear purpose, allowing for easy extension.

Coupling: Low coupling supports extensibility by minimizing the ripple effect of changes. When you need to add new features, changes are less likely to impact other parts of the software. This allows for more straightforward integration of new functionality without breaking existing code.

3. Impact on Reusability:

Cohesion: High cohesion encourages code reusability. Well-encapsulated modules with a single purpose can be reused in various parts of the software or in different projects, as they are self-contained and easy to understand. Developers can confidently reuse such components without fear of unintended side effects.

Coupling: Low coupling promotes code reusability because modules that are loosely coupled can be used independently of each other. A well-designed, loosely coupled module can be extracted and reused in different projects with minimal or no modifications.

Real-World Scenario Example:

Imagine a real-world scenario where a software project is being maintained and extended. The project's initial design had low cohesion and high coupling, making it challenging to understand and maintain. Each change introduced unexpected side effects in other parts of the software. The team recognized these issues and decided to re-design the project with an emphasis on cohesion and coupling.

After the redesign:

Maintainability: Code is easier to maintain because each module has high cohesion, focusing on a single responsibility. Changes in one module have minimal impact on others.

Extensibility: Adding new features has become more efficient because modules are designed with extensibility in mind. Developers can create new, well-cohesive modules that integrate seamlessly with the existing code.

Reusability: Reusable modules have been identified and isolated with low coupling. These modules can be easily reused in other projects or within different parts of the current project.

In this scenario, the project's redesign resulted in improved maintainability, extensibility, and reusability, leading to a more robust and adaptable software system.