Object Oriented Analysis and Design Ques. Bank Solution

- Aditya Dhiman

Q1: What is the primary advantage of using modularity in object-oriented design?

Answer:
The primary advantage of modularity in object-oriented design is that it allows for dividing a system into smaller, independent, and interchangeable modules, which improves code maintainability, scalability, and reusability.

Q2: Provide a brief definition of encapsulation in object-oriented systems.

Answer:
Encapsulation refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class, while restricting direct access to some of the object's components to protect the data.

Q3: What is the purpose of inheritance in object-oriented programming?

Answer:
Inheritance allows one class (child class) to inherit the properties and methods of another class (parent class), promoting code reusability and enabling the creation of hierarchical relationships.

Q4: Explain what a class represents in object-oriented programming.

Answer:
A class is a blueprint or template in object-oriented programming that defines attributes (data) and behaviors (methods) for objects. It serves as a prototype from which objects are created.

Q5: What is the difference between aggregation and composition?

Answer:
In **aggregation**, the contained objects can exist independently of the container object, while in

composition, the contained objects are strongly dependent on the lifecycle of the container object (i.e., if the container object is destroyed, the contained objects are too).

Q6: Name one key feature of reusability in object-oriented systems.

Answer:
One key feature of reusability in object-oriented systems is **inheritance**, which allows existing code to be extended and reused in new contexts without modifying the original code.

Q7: What is an object in the context of object-oriented programming?

Answer:
An object is an instance of a class that contains both data (attributes) and methods (functions) to represent real-world entities and their behaviors in object-oriented programming.

Q8: How does encapsulation contribute to maintainability in object-oriented systems?

Answer:
Encapsulation contributes to maintainability by hiding the internal implementation details of an object and exposing only the necessary parts through a public interface. This allows developers to modify internal code without affecting other parts of the system.

Q9: Define modularity in object-oriented systems with an example.

Answer:
Modularity in object-oriented systems refers to dividing the system into independent and interchangeable modules or classes. For example, in a banking system, separate classes can be created for Account, Customer, and Transaction, allowing each to be developed and maintained independently.

Q10: What type of inheritance is demonstrated when a class inherits from another class, which in turn inherits from yet another class?

Answer:
This is an example of **multilevel inheritance**, where a class inherits from a parent class, which itself is derived from another class higher up in the hierarchy.

Q11: What Tests Can Help Find Useful Use Cases?

Answer:
To find useful use cases, techniques such as **brainstorming**, **requirements analysis**, and **interviews with stakeholders** can be used. These tests help identify the key interactions between actors and the system.

Q12: What are Use Case Diagrams?

Answer:
A Use Case Diagram is a type of UML (Unified Modeling Language) diagram that represents the interactions between users (actors) and the system to capture the system's functional requirements.

Q13: What is an activity diagram?

Answer:
An activity diagram is a UML diagram that visually represents the flow of control or the sequence of activities in a system, illustrating how a process progresses step by step.

Q14: What are interactive diagrams? List out the components involved in interactive diagrams.

Answer:
Interactive diagrams in UML include **sequence diagrams** and **communication diagrams**. These diagrams show how objects interact through message exchanges. Components include objects, lifelines, messages, and interactions.

Q15: What is the use of a component diagram?

Answer:
A component diagram is used in UML to represent the structural relationships and organization of components in a system, showing how the system's software components interact with each other.

Q16: Give the use of UML state diagram.

Answer:
A UML state diagram is used to model the various states that an object goes through in response to events, illustrating the transitions between those states and the actions that trigger state changes.

Q17: What is meant by State chart Diagrams?

Answer:
State chart diagrams, also known as state machine diagrams, depict the dynamic behavior of a system by showing the various states an object can be in and how it transitions from one state to another based on events or conditions.

Q18: Distinguish between method and message in object-oriented programming.

Answer:
A **method** is a function defined within a class that performs an action. A **message** is a communication sent to an object to invoke one of its methods.

Q19: Define Software development process.

Answer:
The software development process is a structured set of activities involved in developing software, which includes planning, designing, coding, testing, and maintaining software applications.

Q20: What is the use of Unified Process?

Answer:
The Unified Process is a software development methodology that organizes the process into phases such as **inception, elaboration, construction, and transition**. It provides a framework for iterative and incremental development.

Level B

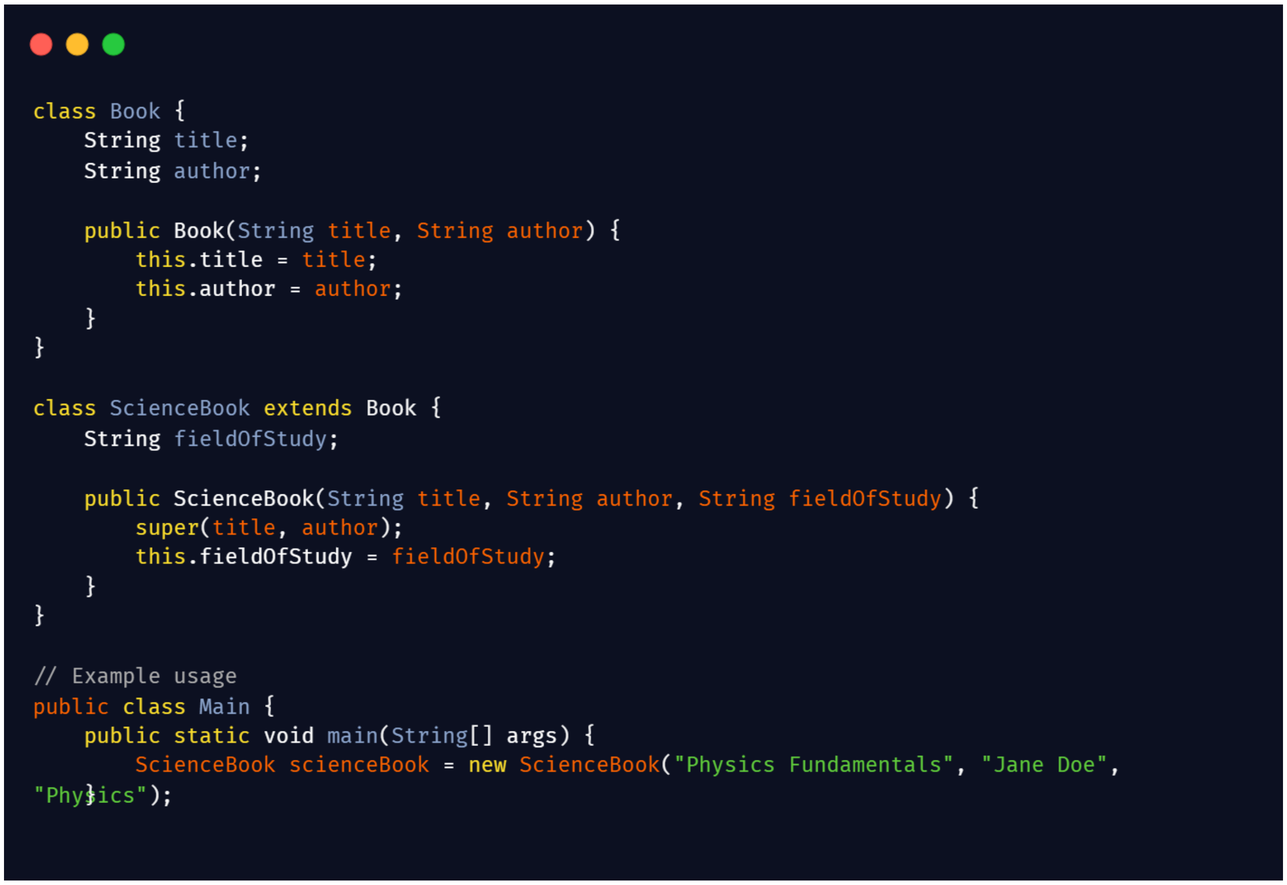
Q21: Describe how you would use the concept of inheritance to create a ScienceBook class from a Book class with the help of an example.

Concept of Inheritance

Inheritance is a core principle of object-oriented programming that allows one class (subclass) to inherit the attributes and methods of another class (superclass). This promotes code reuse and establishes a relationship between classes.

Example: Creating a ScienceBook Class

In our example, we have a Book class representing general book attributes. We will create a ScienceBook class that inherits from Book, adding specific attributes relevant to science books.



```java
class Book {
    String title;
    String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
}

class ScienceBook extends Book {
    String fieldOfStudy;

    public ScienceBook(String title, String author, String fieldOfStudy) {
        super(title, author);
        this.fieldOfStudy = fieldOfStudy;
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        ScienceBook scienceBook = new ScienceBook("Physics Fundamentals", "Jane Doe",
"Physics");
    }
```

In this example, the ScienceBook class extends Book, inheriting its properties while adding a fieldOfStudy attribute.

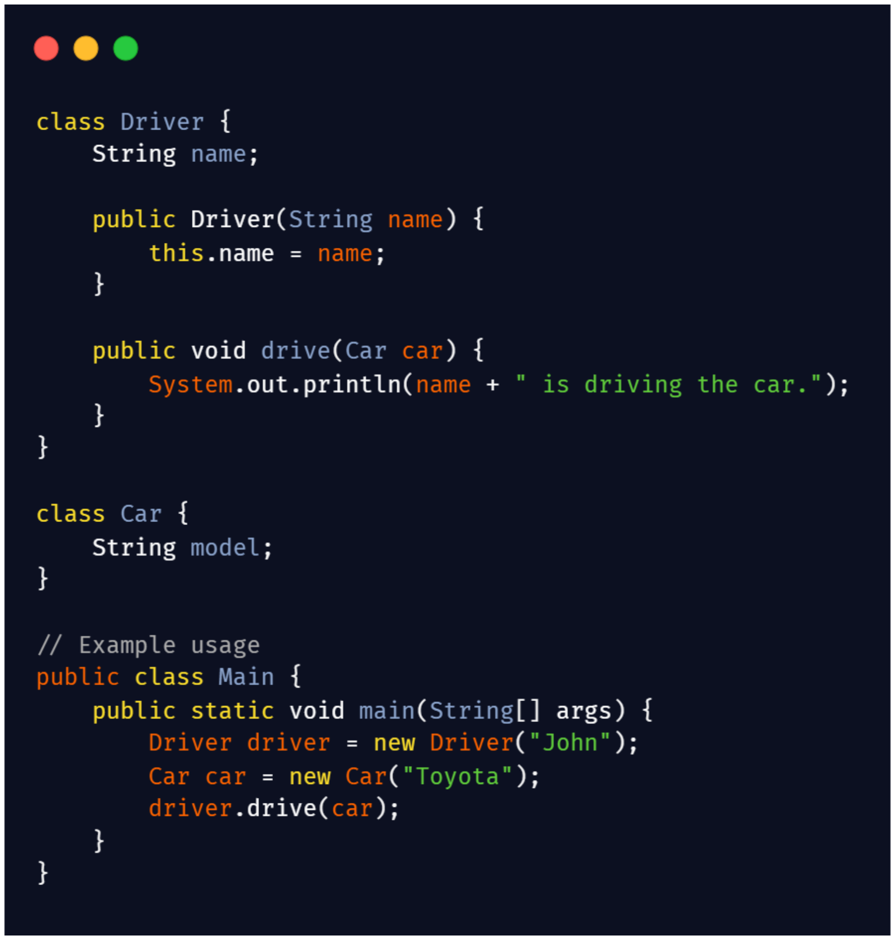
Q22: Explain the difference between unidirectional and bidirectional association with examples.

Unidirectional Association

In a unidirectional association, one class knows about another class, but not the other way around. This type of relationship is often simpler and is sufficient in many cases.

Example: Driver and Car

In this scenario, a Driver can drive a Car, but the Car does not need to know who is driving it.



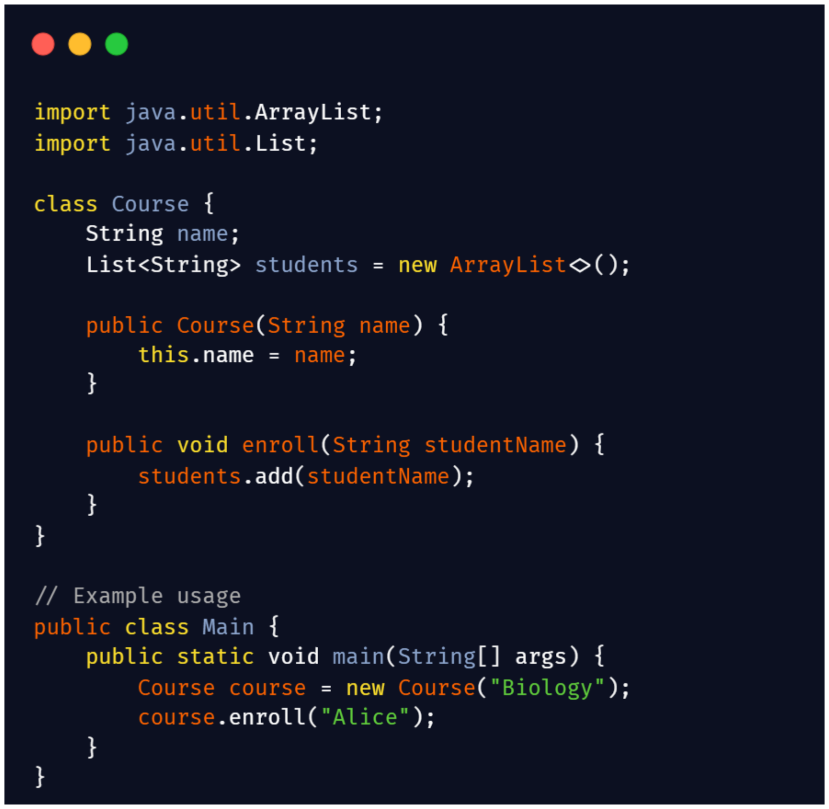
```java
class Driver {
    String name;

    public Driver(String name) {
        this.name = name;
    }

    public void drive(Car car) {
        System.out.println(name + " is driving the car.");
    }
}

class Car {
    String model;
}

// Example usage
public class Main {
    public static void main(String[] args) {
        Driver driver = new Driver("John");
        Car car = new Car("Toyota");
        driver.drive(car);
    }
}
```

Bidirectional Association

In a bidirectional association, both classes are aware of each other, allowing for more complex interactions.

Example: Student and Course

In this case, a Student can enroll in a Course, and the Course can maintain a list of enrolled Students.



```java
import java.util.ArrayList;
import java.util.List;

class Course {
    String name;
    List<String> students = new ArrayList<>();

    public Course(String name) {
        this.name = name;
    }

    public void enroll(String studentName) {
        students.add(studentName);
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        Course course = new Course("Biology");
        course.enroll("Alice");
    }
}
```

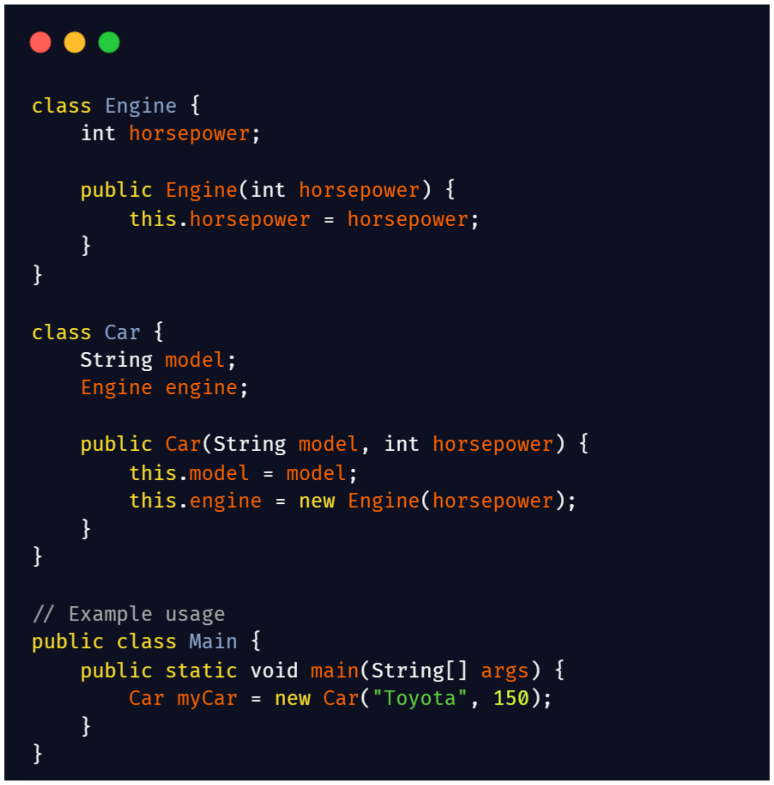
Q23: Discuss the concept of composition and provide a simple code example to illustrate it.

Concept of Composition

Composition is a strong association where one class (the whole) contains references to instances of another class (the parts). When the whole is destroyed, the parts are also destroyed, indicating a lifecycle dependency.

Example: Car and Engine

In this example, a Car class contains an instance of an Engine, signifying that the engine is a part of the car.



```
class Engine {
    int horsepower;

    public Engine(int horsepower) {
        this.horsepower = horsepower;
    }
}

class Car {
    String model;
    Engine engine;

    public Car(String model, int horsepower) {
        this.model = model;
        this.engine = new Engine(horsepower);
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 150);
    }
}
```

In this setup, if a Car object is deleted, the Engine associated with it is also deleted, showcasing composition.

Q24: What is the role of maintainability in object-oriented systems, and how does encapsulation support it?

Role of Maintainability

Maintainability is the ease with which a software system can be modified to correct faults, improve performance, or adapt to changes. It is critical for long-term sustainability and efficiency.

Encapsulation and Its Support for Maintainability
Encapsulation involves bundling the data (attributes) and methods that operate on that data within a single unit (class). This concept enhances maintainability by:

- **Hiding Complexity:** Users interact with a simplified interface without needing to understand the internal workings.
- **Reducing Interdependencies:** Changes in one part of the system require fewer changes in others, making it easier to manage updates and fixes.

Encapsulation thus allows developers to modify classes without impacting other components significantly.

Q25: Explain a) Actors b) Scenarios c) Use cases

a) Actors
Actors are entities that interact with the system. They can be users, external systems, or devices. For instance, in an online shopping system, actors include customers and administrators.

b) Scenarios
A scenario describes a specific sequence of actions taken by an actor to achieve a goal within the system. For example, a customer scenario could involve browsing products, adding them to a cart, and completing a purchase.

c) Use Cases
Use cases define functional requirements from the perspective of actors. They describe the interactions between actors and the system to achieve a specific goal. An example use case is "Purchase Product," detailing the steps a customer follows to buy an item.

Q26: What are the three kinds of Actors?

1. Primary Actors
These are the main users who initiate interactions with the system, such as account holders in a banking application.

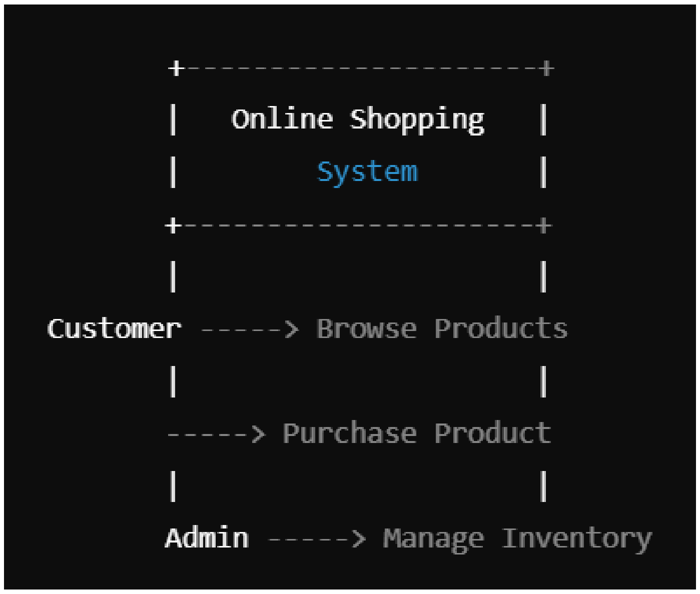
2. Secondary Actors
These actors provide services to the system. For example, in a banking application, a payment processor acts as a secondary actor by facilitating transactions.

3. Off-stage Actors
These external entities are not directly involved in the use case but can affect its outcome. An example could be regulatory bodies that enforce rules the system must follow.

Q27: What does a Use Case Diagram represent? Give an example.

Concept of Use Case Diagram
A use case diagram visually represents the interactions between actors and the system, highlighting the relationships among use cases, actors, and the system boundary.

Example Diagram
In an online shopping system, a use case diagram might include actors like Customer and Admin, with use cases such as Browse Products, Purchase Product, and Manage Inventory.



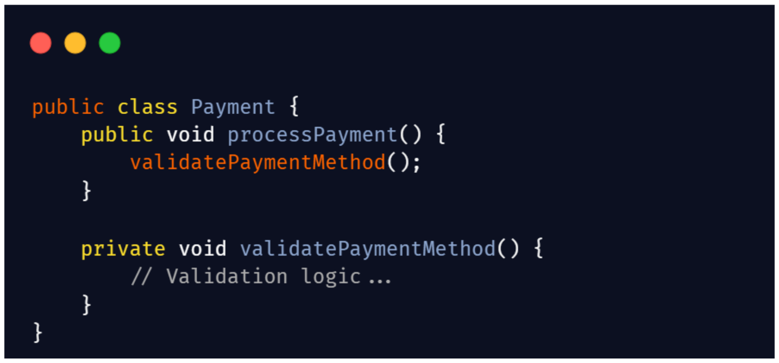
```
        +---------------------+
        |   Online Shopping   |
        |        System       |
        +---------------------+
        |                     |
Customer -----> Browse Products
        |                     |
        -----> Purchase Product
        |                     |
        Admin -----> Manage Inventory
```

This diagram illustrates how each actor interacts with the respective use cases.

Q28: Define use case relationships a) Include b) Extend Include Relationship.

a) Include Relationship
An "include" relationship is used when a use case requires another use case to complete its task. This promotes reusability.

Example: Payment Processing
The use case Process Payment might include Validate Payment Method.

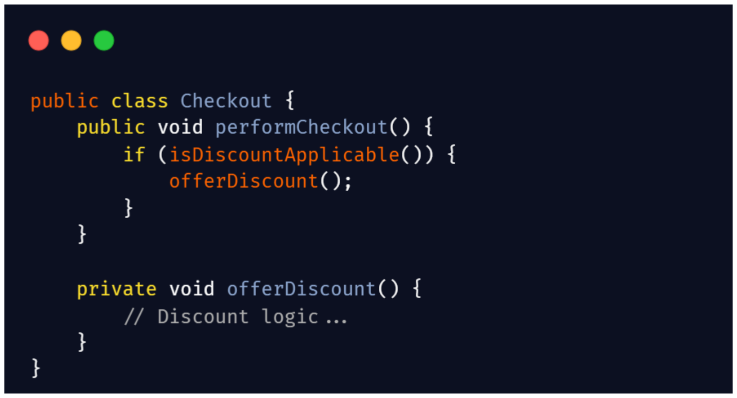

```java
public class Payment {
    public void processPayment() {
        validatePaymentMethod();
    }

    private void validatePaymentMethod() {
        // Validation logic ...
    }
}
```

b) Extend Relationship

An "extend" relationship allows one use case to add optional behavior to another use case under specific conditions.

Example: Checkout Process

The use case Check Out could be extended by Offer Discount when applicable.


```java
public class Checkout {
    public void performCheckout() {
        if (isDiscountApplicable()) {
            offerDiscount();
        }
    }

    private void offerDiscount() {
        // Discount logic ...
    }
}
```

Q29: Write a note on Unified Approach.

Overview of the Unified Approach

The Unified Approach combines best practices from various modeling techniques, offering a framework for software development.

Key Characteristics:

- **Use Case Driven:** Focuses on defining requirements through use cases.
- **Architecture Centric:** Emphasizes a solid architecture as the foundation of development.
- **Iterative and Incremental Development:** Encourages frequent revisions and improvements through iterative cycles.

This approach is effective in managing complexity in software projects.

Q30: Explain Object-Oriented Methodology.

Definition of Object-Oriented Methodology
Object-Oriented Methodology is a structured approach to software development that utilizes principles like encapsulation, inheritance, and polymorphism.

Key Components:

- **Object Modeling:** Identifying objects, their attributes, and behaviors.
- **Class Design:** Creating classes that represent real-world entities and their interactions.
- **Behavioral Modeling:** Specifying how objects interact and perform operations.

This methodology supports reusability, scalability, and easier maintenance, making it suitable for complex systems.

Level C

Q31: Compare and Contrast Aggregation and Composition

Definitions

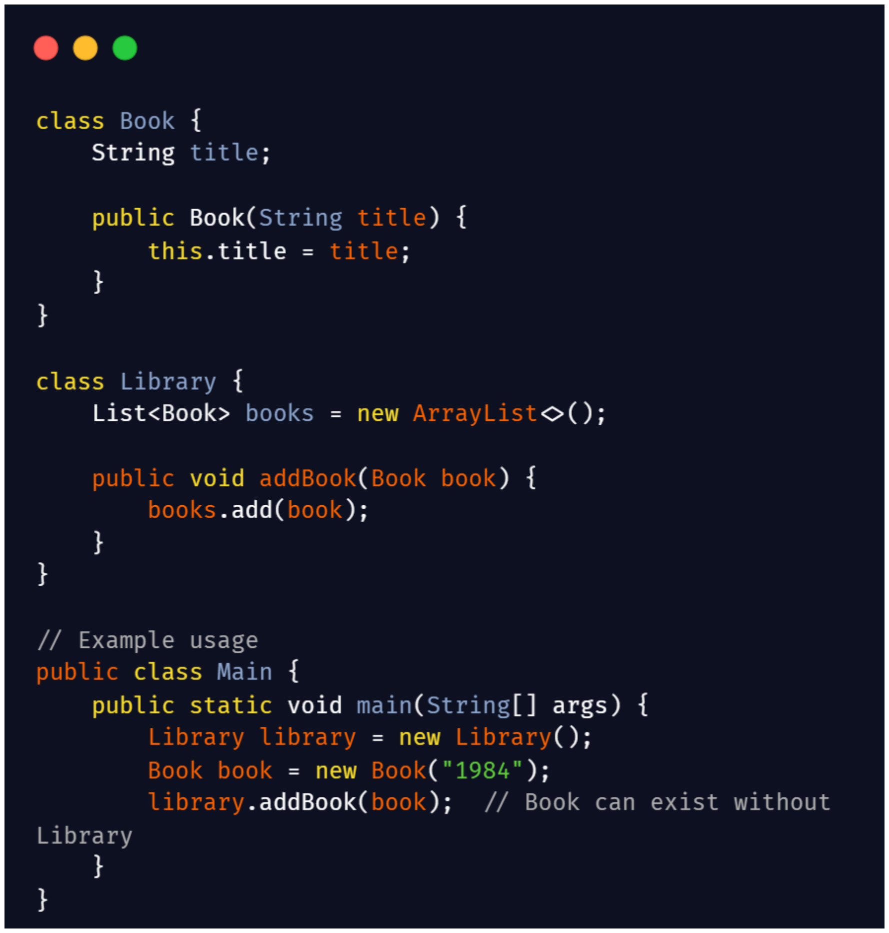
- **Aggregation** is a "has-a" relationship where the contained objects can exist independently of the container.
- **Composition** is a stronger form of aggregation where the contained objects cannot exist independently of the container.

Key Differences

Feature	**Aggregation**	**Composition**
Lifespan	Independent lifecycle	Dependent lifecycle
Ownership	No strict ownership	Strict ownership
Example	A Library has Books	A House has Rooms

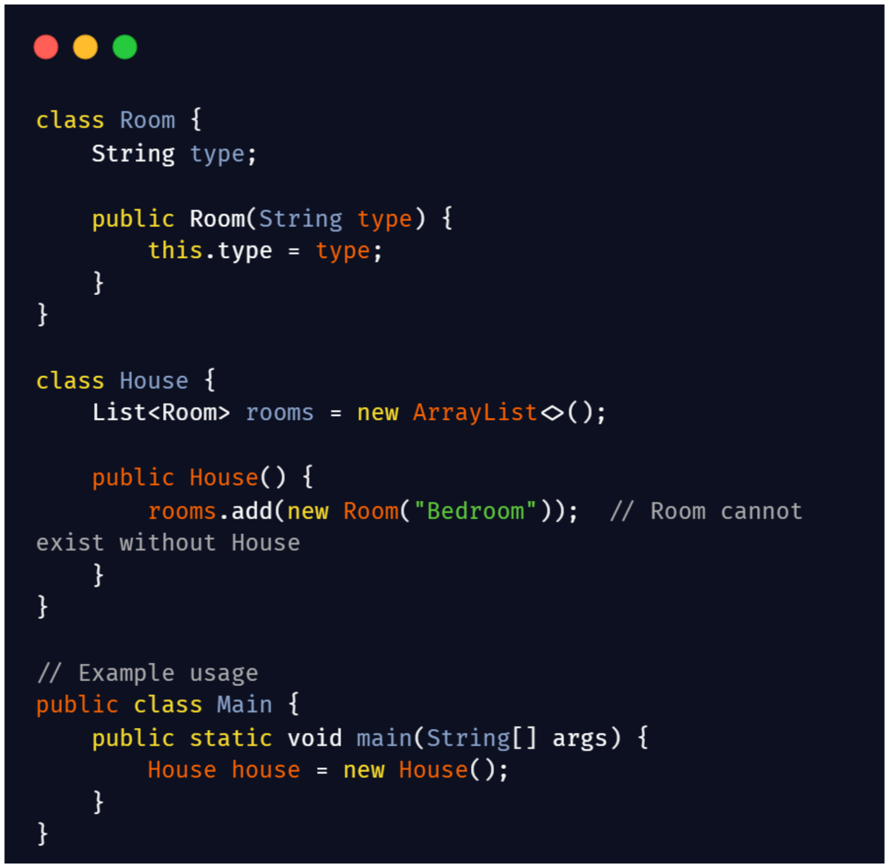
Example Code Snippets

Aggregation Example:



```java
class Book {
    String title;

    public Book(String title) {
        this.title = title;
    }
}

class Library {
    List<Book> books = new ArrayList<>();

    public void addBook(Book book) {
        books.add(book);
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        Library library = new Library();
        Book book = new Book("1984");
        library.addBook(book);  // Book can exist without
Library
    }
}
```

Composition Example:

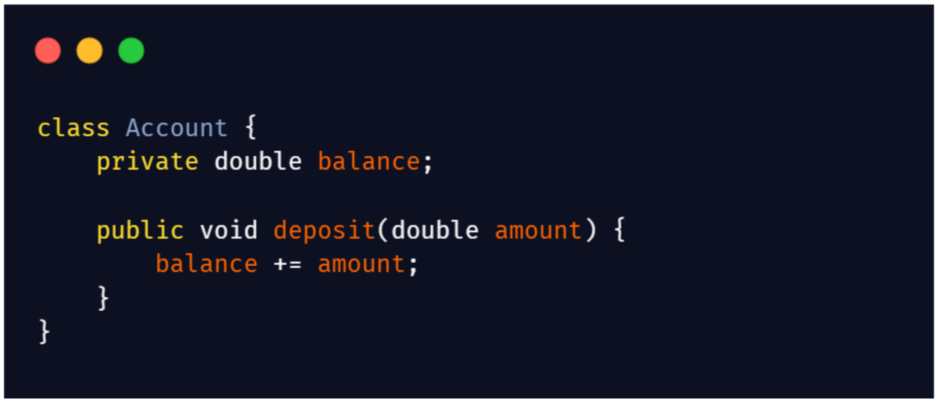


```
class Room {
    String type;

    public Room(String type) {
        this.type = type;
    }
}

class House {
    List<Room> rooms = new ArrayList<>();

    public House() {
        rooms.add(new Room("Bedroom"));  // Room cannot
exist without House
    }
}

// Example usage
public class Main {
    public static void main(String[] args) {
        House house = new House();
    }
}
```

Q32: Discuss the Object-Oriented System Development Life Cycle (OOSDLC)

The OOSDLC comprises several phases that guide the development of an object-oriented system:

1. **Requirement Analysis**
   - Identify what the system needs to achieve.
   - **Example:** Gather requirements for a banking system, such as account management and transaction processing.
2. **System Design**
   - Define the architecture and design the system using UML diagrams.
   - **Example:** Create class diagrams to represent the Account, Customer, and Transaction classes.
3. **Implementation**
   - Write code based on the designs.
   - **Example:**



```
class Account {
    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }
}
```

4. **Testing**

- Verify that the system meets the requirements.
- **Example:** Conduct unit tests for account deposit and withdrawal methods.

5. **Deployment**

- Release the system to users.
- **Example:** Deploy the banking application on a web server.

6. **Maintenance**

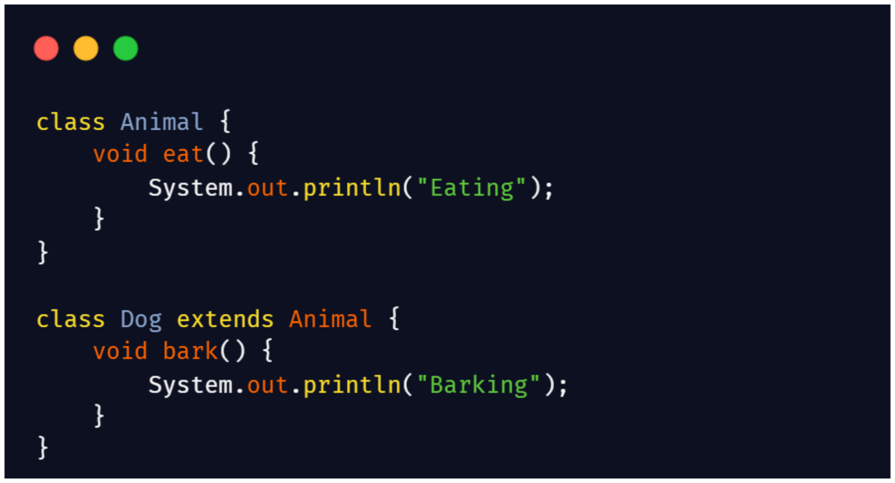
- Update and enhance the system as needed.
- **Example:** Add features like online fund transfers after initial deployment.

Q33: Explain Inheritance in Object-Oriented Programming

Definition
Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass).

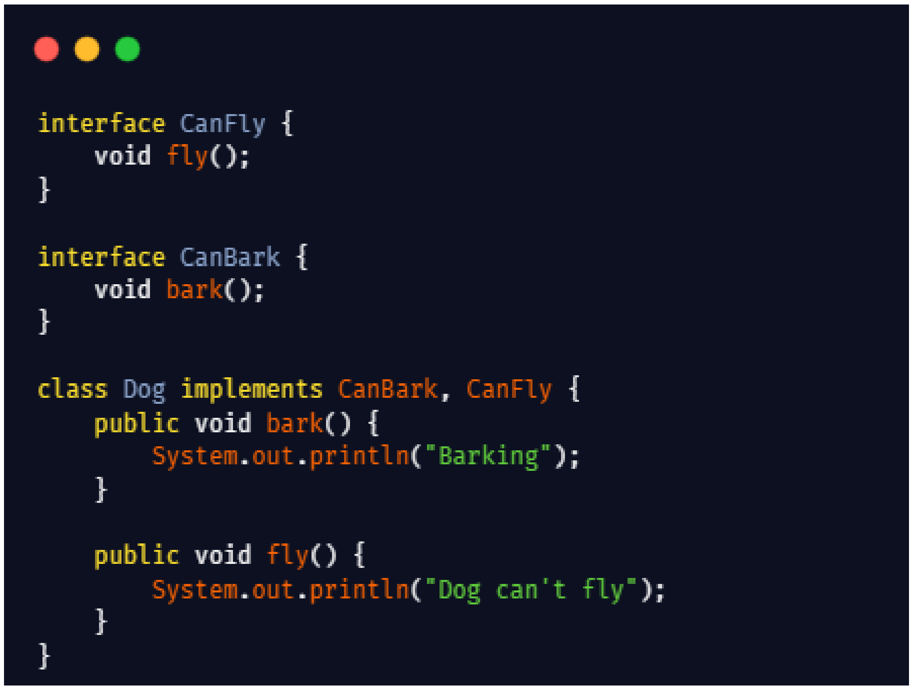
Types of Inheritance:

1. **Single Inheritance**
   - A subclass inherits from one superclass.
   - **Example:**



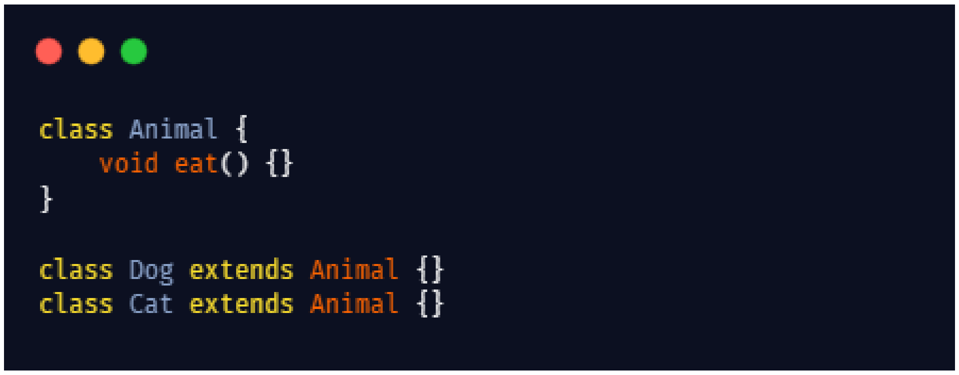
```java
class Animal {
    void eat() {
        System.out.println("Eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking");
    }
}
```

2. **Multiple Inheritance**
    - A subclass implements multiple interfaces.
    - **Example:**



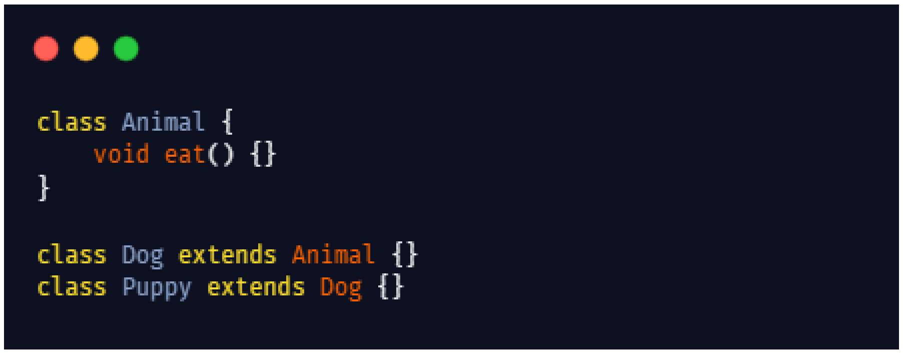
```java
interface CanFly {
    void fly();
}

interface CanBark {
    void bark();
}

class Dog implements CanBark, CanFly {
    public void bark() {
        System.out.println("Barking");
    }

    public void fly() {
        System.out.println("Dog can't fly");
    }
}
```

3. **Hierarchical Inheritance**
    - Multiple subclasses inherit from one superclass.
    - **Example:**



```java
class Animal {
    void eat() {}
}

class Dog extends Animal {}
class Cat extends Animal {}
```

4. **Multilevel Inheritance**
   - A class inherits from another subclass.
   - **Example:**

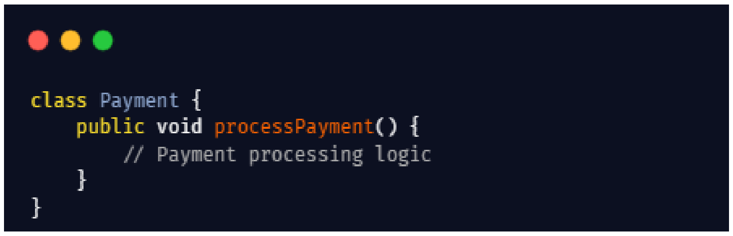


```java
class Animal {
    void eat() {}
}

class Dog extends Animal {}
class Puppy extends Dog {}
```

Q34: Importance and Benefits of Modularity, Reusability, and Extensibility

Modularity

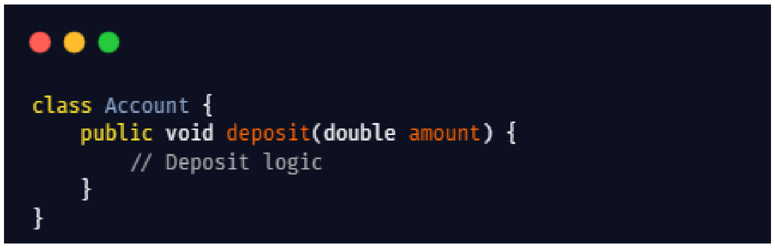
- **Definition:** The design principle that breaks down a system into smaller, manageable modules.
- **Benefit:** Simplifies maintenance and enhances collaboration.
- **Example:**



```java
class Payment {
    public void processPayment() {
        // Payment processing logic
    }
}
```

Reusability

- **Definition:** The ability to use existing components in new applications.
- **Benefit:** Saves time and reduces code duplication.
- **Example:**

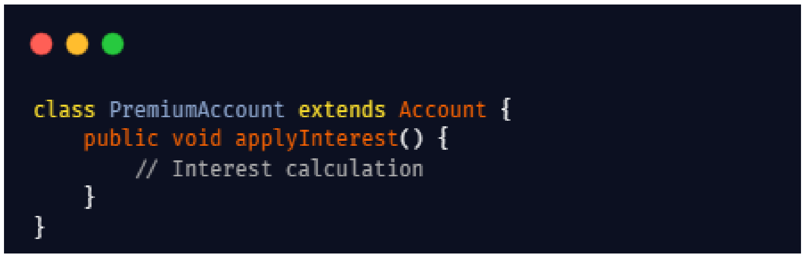


```
class Account {
    public void deposit(double amount) {
        // Deposit logic
    }
}
```

Reusing Account class in multiple banking applications.

Extensibility

- **Definition:** The ability to add new features without modifying existing code.
- **Benefit:** Facilitates updates and enhancements.
- **Example:**



```
class PremiumAccount extends Account {
    public void applyInterest() {
        // Interest calculation
    }
}
```

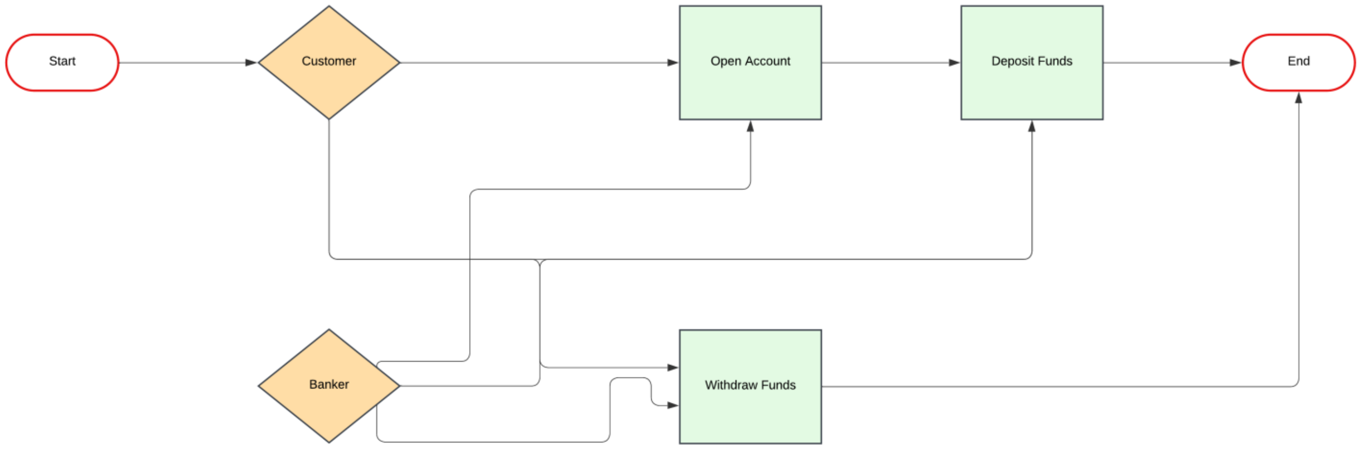
Q35: Draw and Discuss an Analysis Model for Banking System

Analysis Model Components

An analysis model includes use case diagrams, class diagrams, and sequence diagrams to represent the system.

Use Case Diagram

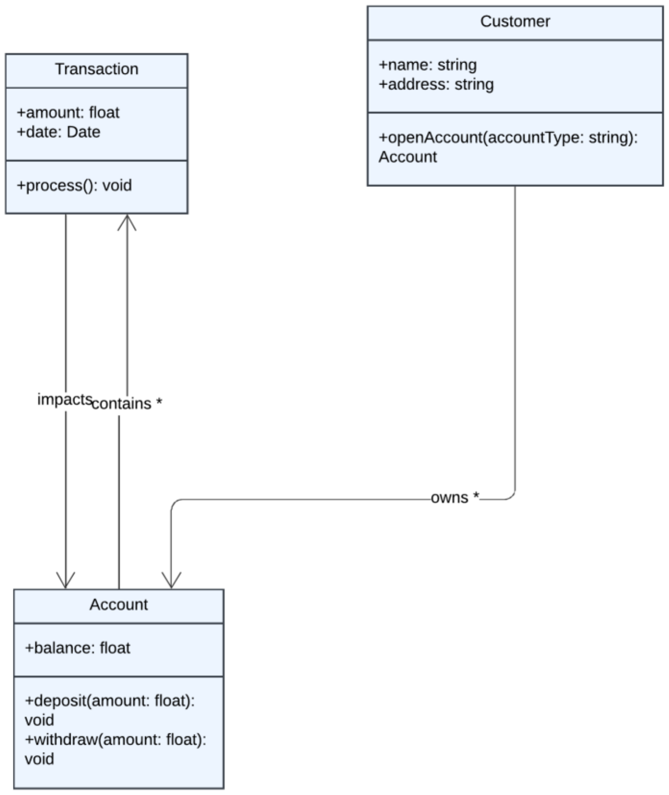
A simple use case diagram might include actors like Customer and Banker interacting with use cases such as Open Account, Deposit Funds, and Withdraw Funds.



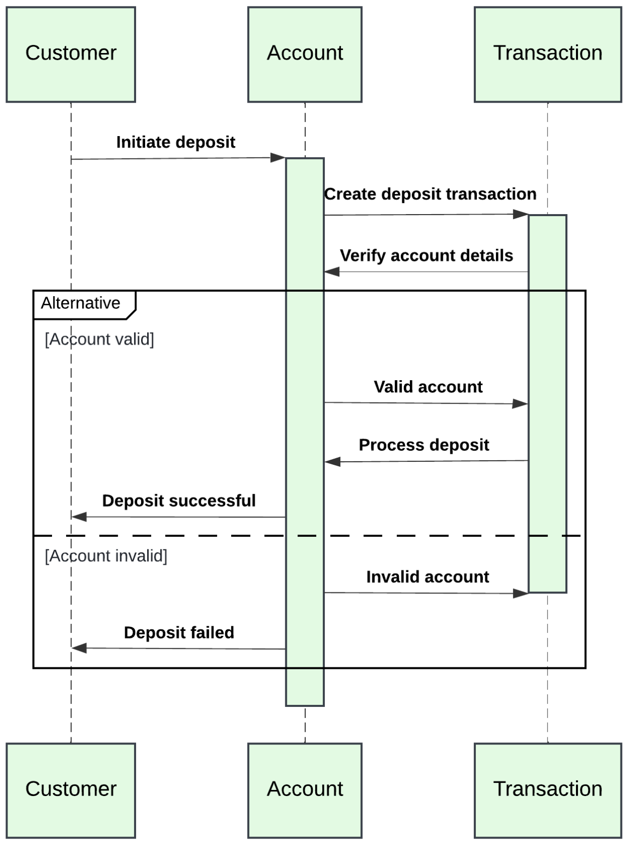
Class Diagram

Key classes might include Customer, Account, and Transaction, each with relevant attributes and methods.

- Customer: name, address, openAccount()
- Account: balance, deposit(), withdraw()
- Transaction: amount, date, process()



Sequence Diagram : sequence diagram that illustrates the process of depositing funds, focusing on the interactions between the Customer, Account, and Transaction classes.



Q37: List Out Some Scenarios that Illustrate Varying Degrees of Functional Cohesion

Functional Cohesion refers to how well the components of a module work together to achieve a single purpose. Here are some scenarios illustrating varying degrees:

1. **High Cohesion**
   - **Scenario:** A PaymentProcessor class that handles all payment-related functions (validate payment, process payment, generate receipt).
   - **Benefit:** Easy to maintain and understand.
2. **Medium Cohesion**
   - **Scenario:** A UserManager class that handles user login, registration, and profile updates.
   - **Benefit:** Related functions, but includes some unrelated tasks.

3. **Low Cohesion**
   - **Scenario:** A Utility class with methods for file handling, network requests, and string manipulations.
   - **Drawback:** Difficult to maintain; the class does too many unrelated things.
4. **No Cohesion**
   - **Scenario:** A Miscellaneous class containing random functions like logging, file IO, and user authentication.
   - **Drawback:** Very hard to manage and understand due to lack of a clear purpose.