***FALL LSP 2024 MIDTERM***

Citations- <https://softwareengineering.stackexchange.com/questions/337031/association-of-objects-in-uml-static-class-when-creating-but-not-storing>

<https://medium.com/@dorinbaba/using-clean-architecture-to-ensure-separation-of-concerns-c4a9b7d8f0c1>

<https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/>

<https://www.w3resource.com/java-exercises/oop/java-oop-exercise-20.php>

Data oriented-programming Java textbook

Question 1

**a) Why the UserFitness Class Has Low Cohesion**

1. **Multiple Responsibilities**: It handles various aspects (workouts, diet, progress) rather than focusing on one specific task.
2. **Lack of Focus**: Methods and attributes are not closely related, making it unclear what the class is primarily for.
3. **Difficult to Understand**: It’s harder for developers to grasp its purpose and usage.
4. **Impacts Maintainability**: Changes in one area can inadvertently affect unrelated features.
5. **Testing Challenges**: Requires complex setup for testing all functionalities, making tests harder to write.

**b) How to Split the UserFitness Class into More Cohesive Classes**

1. **Identify Core Responsibilities**: Break down into distinct areas like workout tracking, nutrition management, progress monitoring, and user profiles.
2. **Create Separate Classes**:
   * **WorkoutTracker** for logging workouts.
   * **NutritionManager** for managing diet.
   * **ProgressMonitor** for tracking progress.
   * **UserProfile** for user information.
3. **Define Clear Interfaces**: Ensure each class has a well-defined purpose and interface.
4. **Facilitate Interaction**: Optionally create a higher-level class to coordinate interactions among the new classes.

Question 2

1. **Shared Attributes and Methods**: An abstract class allows you to define common attributes (like name, ID, etc.) and methods (like enroll, withdraw) that are shared among all students while still allowing for specific implementations in the subclasses (FirstYear, SecondYear, and ThirdYear).
2. **Extensibility**: Using an abstract class provides flexibility for future subclasses if you need to introduce new year levels or different types of students.
3. **Inheritance**: It allows the subclasses to inherit from Student, making it easier to manage year-specific behaviors and states without needing to redefine common functionality.

Regarding the flawed method removeThirdYears(List<Students> students):

The method is flawed primarily because it modifies the list while iterating over it, which can lead to a ConcurrentModificationException in Java. When you call students.remove(thirdYear), it alters the size of the list while you're still in the loop, causing unpredictable behavior.

A better approach would be to use an Iterator to safely remove elements during iteration.

Question 3

The BankAccount class is **not well-encapsulated**. Here are a few reasons why, along with suggested changes:

1. **Public Fields**: The fields accountHolderName, accountNumber, balance, and interestRate are public. This exposes the internal state of the class directly, allowing external code to modify these values without any validation or control.

**Change**: Make these fields private and provide getter methods for those that need to be accessed externally. Use setter methods with validation where necessary (e.g., for deposits and withdrawals).

**Exposed Implementation**: The methods addInterest, deposit, and withdraw are public and directly manipulate the balance. This could lead to inconsistencies if called in an unintended order or if there are other constraints to consider (like overdraft protection).

**Change**: Consider adding validation in withdraw to prevent overdrawing the account. You might also want to keep addInterest as a private method that can be called under controlled circumstances (like during a scheduled update).

1. **No Control over Interest Calculation**: The interest calculation is exposed directly. This could be modified by external code, which may not be desirable. **Change**: Keep the interest calculation private and provide a controlled way to update it if necessary. By making these changes, the BankAccount class would be better encapsulated, providing controlled access to its fields and ensuring its internal state remains consistent.

**Question4**

 **Static Association**: The UML diagram likely establishes a static relationship between the Car class and the trim level. If the trim level is an attribute of the Car class, changing it requires altering the object's state directly, which is not ideal for real-time updates.

 **Limited Flexibility**: If trim levels are hardcoded or represented as fixed properties, it becomes cumbersome to manage changes. For instance, if a customer decides to switch from Luxury to Sport during manufacturing, the software must handle the reconfiguration of various components, which can be complex and error-prone.

 **Single Responsibility Principle Violation**: The Car class might be taking on too many responsibilities, including managing its own state for trim levels. This violates the single responsibility principle, making the code harder to maintain and understand.

 **Encapsulation Issues**: If the trim level is tightly coupled with the Car class, it limits the encapsulation of trim-specific behavior and attributes. Changes in trim-level features might necessitate changes across multiple methods or even classes.

 **Reusability Limitations**: By tying the trim level directly to the Car, you reduce the potential for reusability. If trim levels are implemented as separate classes, it allows for different trim level configurations to be reused across different car models.

1. To enable dynamic changes to the trim level of a car, you can use composition to separate the trim-level logic from the Car class. Here’s how you could refactor the structure:

Create a TrimLevel Class: Define a separate class for TrimLevel with relevant properties and behaviors for Base, Luxury, and Sport trim levels. Each trim level can encapsulate its own features and specifications.

To address the task regarding the car manufacturer's software structure in Java, let's break down the two parts:

**Modify the Car Class**: Change the Car class to have a TrimLevel object instead of a primitive representation of the trim level. This allows the Car to dynamically change its trim level by replacing the TrimLevel object.

**Implement a TrimLevel Factory**: Optionally, you could create a factory class or method that provides instances of different TrimLevels, ensuring that you maintain a consistent set of features for each trim level.

This refactored structure ultimately provides a more robust solution to manage trim levels in a dynamic manner, enhancing the overall design and functionality of the car tracking system.

Question 5

 **Identify Common Functionality**: Start by identifying the common functionalities between the two classes. List out shared methods, properties, and behaviors. This step is essential for refactoring.

 **Create a Base Class or Interface**: Implement a superclass or interface that captures the common functionality. This aligns with Riel's heuristic of "Favor Composition Over Inheritance." You might create a base class that contains shared code, which other classes can then extend or implement.

public abstract class BaseClass {

// Common methods and properties

public void commonMethod() {

// Implementation

}

}

public class ClassA extends BaseClass {

// Specific implementation for ClassA

}

public class ClassB extends BaseClass {

// Specific implementation for ClassB

}

**Utilize Composition**: If the classes differ significantly in behavior, consider using composition instead of inheritance. Create a common interface that the classes can implement, and encapsulate shared behaviors in separate components. This promotes reusability and flexibility.

public interface CommonBehavior {

void performAction();

}

public class SharedBehavior implements CommonBehavior {

public void performAction() {

// Shared implementation

}

}

public class ClassA {

private CommonBehavior behavior;

public ClassA(CommonBehavior behavior) {

this.behavior = behavior;

}

public void useBehavior() {

behavior.performAction();

}

}

public class ClassB {

private CommonBehavior behavior;

public ClassB(CommonBehavior behavior) {

this.behavior = behavior;

}

public void useBehavior() {

behavior.performAction();

}

}

Question 6

Placing database calls directly in the Car and BankAccount classes can lead to several design issues, primarily related to coupling and separation of concerns. Here’s why it is generally unwise:

**1. Tight Coupling to Infrastructure**

By embedding database logic directly within the domain classes (like Car and BankAccount), you create a tight coupling between the domain model and the database infrastructure. This means:

* **Difficulties in Testing**: Unit testing becomes challenging because the classes now depend on a database context, making it harder to test them in isolation.
* **Reduced Flexibility**: If you decide to change the database technology (e.g., from SQL to NoSQL), you would need to modify the domain classes rather than just the data access layer.

**2. Violation of Single Responsibility Principle**

The Single Responsibility Principle states that a class should have only one reason to change. When you include database functionality in your domain classes:

* **Increased Responsibilities**: Car and BankAccount are now responsible not only for representing their respective entities but also for handling data persistence. This makes the classes more complex and harder to maintain.
* **Changes in Data Handling**: Any changes in the way data is stored or retrieved (e.g., new fields, changes in the schema) will necessitate changes in the domain classes, affecting their core logic.

**3. Difficulties in Maintaining Business Logic**

If the database interactions are spread across multiple domain classes, it can lead to:

* **Duplication of Logic**: Common database operations might be duplicated across different classes, leading to inconsistencies and potential errors.
* **Challenges in Business Logic**: The domain classes should focus on encapsulating business rules and logic, not on the mechanics of data storage. Mixing these concerns can lead to a confusing codebase.

**4. Lack of Abstraction**

Inserting database calls directly into your domain classes eliminates a clear separation between the business logic and data access layers. This can lead to:

* **Tightly Coupled Architecture**: Any changes to data storage or access patterns can ripple through your entire application, complicating updates and maintenance.
* **Difficulty in Refactoring**: If you need to refactor the data access layer (for performance, for instance), the changes can be extensive and error-prone if they touch the domain classes.

**5. Poor Scalability**

As the application grows, managing persistence directly in domain classes can lead to:

* **Complicated Code Structure**: The more classes you have that perform their own database operations, the more complex your codebase becomes.
* **Scalability Issues**: Adding new features or changing existing ones may require modifying multiple classes, which can introduce bugs and complicate development efforts.

Question 7

**Common Behavior and Attributes**

* **Shared Characteristics**: There should be a clear set of common attributes and behaviors that multiple classes will share. This commonality justifies creating a superclass that encapsulates these shared properties and methods.
* **Cohesion**: The classes in the hierarchy should be cohesive, meaning that the shared behaviors should logically belong to the superclass. If the functionalities diverge too much, it may indicate that inheritance is not the best choice.

**2. Is-A Relationship**

* **Clear Hierarchical Relationship**: There should be a clear "is-a" relationship between the superclass and subclasses. For instance, if you have a superclass Animal, subclasses like Dog and Cat can be justified because they are both types of Animal.
* **Conceptual Clarity**: The hierarchy should reflect real-world relationships or logical groupings that make sense conceptually. This enhances the understanding of the system for both current and future developers.

**3. Expected Extension and Specialization**

* **Need for Specialization**: The subclasses should represent specialized versions of the superclass. Each subclass should extend or override behaviors in a way that adds value or specific functionality.
* **Future Scalability**: Consider whether you anticipate needing additional subclasses in the future. A well-designed hierarchy should be flexible enough to allow for easy addition of new types.

**4. Stable Base Class**

* **Stable Superclass**: The superclass should represent stable functionality that is unlikely to change frequently. If the base class is subject to frequent changes, it can lead to complications for all subclasses, making the hierarchy brittle.
* **Generic Functionality**: The superclass should ideally provide generic functionality that is broadly applicable to all subclasses, ensuring that changes to the base class benefit all derived classes.

**5. Polymorphism Requirements**

* **Polymorphic Behavior**: If there is a need for polymorphic behavior—where subclasses can be treated as instances of the superclass—this is a strong indication that an inheritance structure may be beneficial. This allows methods to operate on objects of the superclass and invoke overridden methods in the subclasses.

Question 8: Difference Between Inheritance and Composition

Inheritance:

Definition: Inheritance is a mechanism where a new class (subclass) derives from an existing class (superclass), inheriting its properties and methods. This creates an "is-a" relationship.

Use Case: It is often used when there is a clear hierarchical relationship between classes, allowing for shared behavior and code reuse.

Example: If you have a Vehicle superclass and subclasses like Car and Truck, both Car and Truck inherit common attributes like speed and capacity.

Composition:

Definition: Composition involves building complex objects by combining simpler objects (components). This creates a "has-a" relationship rather than an "is-a" relationship.

Use Case: It is used when classes can contain references to other classes, promoting flexibility and allowing behavior to be changed at runtime.

Example: A Car class might have a Engine object, a Wheel object, and a Transmission object, representing the components that make up the car.

Choosing Between Them:

When to Use Inheritance: Choose inheritance when you have a strong hierarchical relationship and the subclasses need to share behavior with the superclass. It is suitable for well-defined categories where polymorphism is beneficial.

When to Use Composition: Choose composition when you want to achieve greater flexibility and decoupling. It allows for easier changes and enhancements in behavior without affecting the entire class hierarchy. Composition is also preferable when there is no clear "is-a" relationship.

Question 9: Meaning of Cohesion

Cohesion:

Definition: Cohesion refers to how closely related and focused the responsibilities of a single module or class are. It measures the degree to which the elements of a module belong together.

Types of Cohesion: Cohesion can range from low (where the module has unrelated functionalities) to high (where the module's responsibilities are closely related and serve a single purpose).

High vs. Low Cohesion:

High Cohesion: High cohesion is generally better because it leads to classes or modules that are easier to understand, maintain, and reuse. When a class has a well-defined purpose, it is easier to manage and modify without affecting other parts of the system.

Low Cohesion: Low cohesion is undesirable as it indicates that a class is trying to do too many unrelated things, making it complex and difficult to understand. This can lead to code that is harder to maintain and more prone to errors.