Name: Isaac Adjei

ID: 03060855

I use multiple books. My most recent book is Head First Java by Kathy Sierra, Berth Bates, and Trisha Gee

**Question 1 a.**

According to my understanding and readings, cohesion refers to how closely related and focused the responsibilities of a class are. In OOP, high cohesion is desirable because it ensures that a class is focused on a single responsibility or closely related set of responsibilities. And on the other hand, a class with low cohesion tries to do too many things at once, which can lead to code that is difficult to maintain, modify, or extend.

The UserFitness class exhibits low cohesion because it tries to manage multiple, unrelated aspects of a user’s fitness journey within a single class. First, the class stores user information such as userName, userAge, weight, and height. These variables relate to basic user data, which would typically belong to a class focused on user identity or profile management. However, the UserFitness class also handles workout tracking. It includes attributes like workoutType, dailySteps, and caloriesBurned, which are clearly related to recording fitness activities, rather than basic user information. This mixing of personal information and workout data is a sign of low cohesion.

Furthermore, the class also tracks nutrition data, particularly the calories consumed during meals. This is another distinct responsibility that involves dietary tracking, which is conceptually separate from both personal user information and workout tracking. By trying to handle all these different tasks in one class, the UserFitness class violates the Single Responsibility Principle (SRP), which states that a class should have only one reason to change. If, for example, a developer needs to update how workout tracking works, they might inadvertently affect how user information or meal tracking is handled, leading to potential bugs or errors.

Thus, the UserFitness class suffers from low cohesion because it combines multiple, unrelated responsibilities—managing user details, workout data, and nutrition tracking—into one entity. As a result, this class would be difficult to maintain and extend over time. It would be much more cohesive, and therefore more maintainable, if these responsibilities were separated into distinct, focused classes.

**Question 1 b.**

The UserFitness class can be broken down into more focused class where each will handle a single of the fitness management. If I were to split, I will have the classes below;

1. User Information Class:

A class responsible only for storing and managing personal details like userName, userAge, weight, and height.

Example: UserProfile.

1. Workout Tracking Class:

A class focused solely on workout-related activities, such as logging the workout type, calories burned, and daily steps.

Example: WorkoutTracker.

1. Nutrition Tracking Class:

This class would be responsible for logging meals and managing calorie intake.

Example: NutritionTracker.

1. FitnessReport Class

A class that could generate fitness reports by aggregating data from the UserProfile, WorkoutTracker, and NutritionTracker.

Example: FitnessReportGenerator.

**Question 2**

Student should be an abstract class because it represents common attributes and behaviors shared by all students, while allowing subclasses (FirstYear, SecondYear, ThirdYear) to implement year-specific behaviors. An abstract class provides shared functionality, while subclasses handle unique differences. An interface is less appropriate because some common functionality (like methods or fields) would likely be shared across all students.

The removeThirdYears method is flawed because it tries to remove elements from the students list while iterating over it, which can cause a ConcurrentModificationException. The proper way to remove elements from a list while iterating is to use an iterator.

**Question 3.**

No, the BankAccount class is not well-encapsulated because it exposes its internal fields (accountHolderName, accountNumber, balance, and interestRate) as public, meaning they can be accessed and modified directly from outside the class. This violates the principle of encapsulation, where data should be kept private and only accessed or modified through well-defined methods.

Changes I would make: Make all the fields private (private String accountHolderName; private String accountNumber; private double balance; private double interestRate;), and provide a getter and setter methods where necessary to control how the fields are accessed and modified.

**Question 4.**

The current structure does not support dynamic trim-level changes because it uses inheritance for the Car class (Base, Sports, Luxury). Inheritance creates a fixed hierarchy where an object cannot change its type at runtime. Once a Car object is created as a Base, Sports, or Luxury type, it cannot switch between these types dynamically, preventing the trim level from being changed after creation.

To support dynamic trim-level changes, I will refactor the Car class to use composition instead of inheritance. I will create a TrimLevel class (or interface) with subclasses BaseTrim, SportTrim, and LuxuryTrim, and have Car hold a reference to a TrimLevel object. This way, the trim level can be changed at runtime by swapping the TrimLevel instance.

Question 5.

Arthur Riel’s heuristic advises against code duplication. Having two classes contain large portions of the same code but differ in smaller areas will lead to a violation of the principle.

First, duplicated code increases maintenance overhead. Whenever the shared functionality needs to be updated or fixed, changes must be made in multiple places. This increases the risk of errors and inconsistencies because it’s easy to forget to update one class when modifying the other. Managing code in multiple locations also makes the system more difficult to maintain in the long run.

Another problem is that it indirectly creates tight coupling between the classes. Although they may appear to be independent, any modification to the shared functionality requires both classes to evolve together. This tightens their relationship, making it harder to modify or extend them individually. More importantly, this situation violates the DRY principle, which encourages minimizing code repetition. Repeated code is not only harder to maintain but also harder to debug and extend as the application grows.

To solve this issue, the best approach I will suggest is to promote code reuse. One solution is to use inheritance if the two classes have a logical "is-a" relationship. In this case, the common code can be moved to a superclass that both classes can inherit from. This keeps the shared functionality in one place, reducing duplication. Another solution is to use composition, where the shared code is moved to a separate helper class, and both classes use it. This is useful when the two classes don’t fit into an inheritance hierarchy but still need to reuse some functionality.

Alternatively, if the classes follow the same structure but differ in specific behaviors, the Template Method pattern can be used. Here, the common behavior is placed in a base class, while subclasses implement the varying parts.

**Question 6.**

This leads to high coupling between the business logic and the database logic.

First, SRP because the classes would now handle both their core responsibilities (e.g., managing car or account details) and database operations. This makes the code harder to maintain because changes in database structure or technology would force changes in the Car and BankAccount classes, even though the business logic hasn't changed.

Second, it reduces flexibility. If you ever need to switch from one database to another, or even just change how data is persisted, you would have to modify these classes, leading to unnecessary complexity and potential errors.

A better approach is to use a Data Access Object or repository pattern, where database operations are handled by separate classes. This keeps the Car and BankAccount classes focused on their primary responsibilities and maintains low coupling between business logic and persistence logic.

**Question 7.**

When designing a new class hierarchy using inheritance, the following conditions should be met: there must be a clear "is-a" relationship between the superclass and subclasses, meaning the subclass should be a specialized form of the superclass. The superclass should define shared functionality and attributes that all subclasses will use, ensuring code reuse. Each subclass should either extend or override some behavior of the superclass to specialize it. Additionally, the hierarchy should be stable, meaning the relationships are unlikely to change often, and polymorphism should be useful, allowing subclasses to be treated as instances of the superclass for flexible behavior.

**Question 8**

Inheritance is when one class (the subclass) derives from another class (the superclass), meaning the subclass automatically inherits the properties and methods of the superclass. This represents an "is-a" relationship. For example, a Car class can inherit from a Vehicle class because a car is a vehicle. Inheritance is used when you want to create a clear hierarchy, and the subclass shares common behavior with the superclass but may also have its own specialized behaviors.

Composition is when a class contains objects of other classes, representing a "has-a" relationship. For example, a Car class might have an Engine object because a car has an engine. With composition, you create complex classes by combining simpler, independent objects. Composition is used when you want more flexibility, as you can change the behavior of the composed objects without affecting the main class or hierarchy.

I would use inheritance when there is a strong “is-a” relationship and the hierarchy is unlikely to change. I will use composition when flexibility and reusability are more important, and I want to avoid tight coupling that comes with inheritance.

**Question 9**

Cohesion refers to how closely related the responsibilities and methods of a class or module are. In other words, it's about how well the different parts of a class or component work together to achieve a single purpose.

High cohesion means that a class or module has a focused and specific responsibility, with all its methods and attributes working towards the same goal. Low cohesion, on the other hand, happens when a class tries to do too many unrelated things, which makes it harder to understand, maintain, or modify.

High cohesion is better because it makes your code more organized and easier to maintain. When a class has high cohesion, changes in one part of the system are less likely to affect unrelated parts, making the code more modular and easier to debug. Low cohesion, on the other hand, leads to messy, complicated code that's harder to extend and more prone to errors.