Name: Fikir Demeke

Class: Large Scale Programming

Date: October 20, 2025

Midterm Answer

**Question 1. (20 pts.)**

**Given the following, analyze the class below and answer the below questions. This question does NOT require you to write any code.**

**a) Should a well-designed class have high or low cohesion? Explain and defend your answer. (5 pts.)**

A well designed class should have high cohesion. High cohesion means all methods and attributes of a class serve one clear and unified purpose. According to lecture notes, in Arthur Riel’s object-oriented heuristics, “a class should capture one and only one key abstraction.” When a class focuses on a single concept it becomes easier to understand, maintain, and reuse. Low cohesion classes, however, tend to mix unrelated responsibilities and violate the principle of separation of concerns which makes code harder to debug and to extend.

**(b)** **Based on your analysis, discuss—only if you believe changes are needed—how you would reorganize or redesign the class to improve its structure. Your answer should (1) identify the class as having high, low or perfect cohesion and (2) describe a general approach to refactoring the class. If you believe the class already has good cohesion, justify why no changes are necessary. (15 pts)**

I would make a redesign of **StudentPortalHelper**:  
The **StudentPortalHelper** class performs many unrelated tasks like:

* **Academic logic:** computeGPA()
* **File I/O:** exportRosterToCsv()
* **UI formatting:** formatDateForUi()
* **Financial logic:** processTuitionPayment()
* **Security:** isStrongPassword()
* **Email generation:** makeWelcomeEmail()
* **Caching:** putCache() and getCache()

This violates Riel’s heuristic that *“*all services offered by a class must be closely related to the class’s core purpose.*”*  
Therefore, the class has **low cohesion** as it mixes database, UI, financial, and security responsibilities that belong to separate abstractions.

**Refactoring approach:**  
To improve its cohesion, I would divide the helper into single purpose classes:

| **New Class** | **Responsibility** |
| --- | --- |
| GpaCalculator | Compute GPA and grade conversions |
| RosterExporter | Export student rosters to CSV |
| EmailFormatter | Build and format welcome emails |
| SecurityValidator | Check password strength |
| PaymentProcessor | Handle tuition payments |
| CacheManager | Manage temporary in-memory data |

Each new class would contain methods that serve only one domain concern which will give us **high cohesion** and **low coupling** across the system. This directly follows Riel’s heuristics: *“*Low coupling, high cohesion*”* and that *“*A class should capture one and only one key abstraction.”

**Question 3. (20 pts.)**

**Car**

getTrimLevel()

**Engine**

accelerate()

getFuelLevel()

**Base**

**Sports**

**Luxury**

**Electric**

**Petrol**

1. **Explain in detail why the current structure does or does not support this. (10 pts.)**

The current inheritance-based structure most likely defines separate subclasses such as BaseCar, LuxuryCar, and SportCar, each extending Car. This design ties the trim level permanently to the object’s type at instantiation. Once a LuxuryCar object is created, its behavior and attributes are fixed, changing it to a SportCar would require creating an entirely new instance. That violates the principle of **high cohesion and low coupling**, makes the code harder to maintain, and duplicates shared logic (e.g., engine or chassis information) across subclasses.  
In other words, the design lacks flexibility and does not allow a car’s trim level to be modified after creation, which prevents true “dynamic configuration” during manufacturing.

1. **Describe how to refactor the structure to allow trim-level change for a car to dynamically change. Hint: How would you modify Car to use composition to solve the problem? (10 pts.)**

To make trim levels adjustable, the Car class should **use composition instead of inheritance.**Rather than hard-coding trims as subclasses, introduce a TrimLevel interface (or abstract class) that defines trim-specific behavior and properties. Then each trim (Base, Luxury, Sport) becomes a separate class implementing that interface.

public interface TrimLevel {

String getFeatures();

double getPriceAdjustment();

}

public class BaseTrim implements TrimLevel { ... }

public class LuxuryTrim implements TrimLevel { ... }

public class SportTrim implements TrimLevel { ... }

public class Car {

private TrimLevel trimLevel;

private Engine engine; // electric or petrol

public void setTrimLevel(TrimLevel newTrim) {

this.trimLevel = newTrim;

}

}

With this composition-based approach, a customer can dynamically switch trims:

car.setTrimLevel(new SportTrim());

The same Car object keeps its identity and engine configuration while updating its trim behavior. This design improves **flexibility**, **maintainability**, and **reusability** while adhering to object-oriented best practices like **encapsulation** and **low coupling**.