# **HOWARD UNIVERSITY**

# **LARGE SCALE PROGRAMMING**

# **FALL 2025**

# **MIDTERM EXAMINATION**

**Name: Manushi Khatri**

**SID: @03086760**

**# QUESTION 1**

1. **Should a well-designed class have high or low cohesion?**

**Answer:**

A well-designed class should have **high cohesion**. When everything inside a class relates to one clear purpose, the code stays organized and predictable. High cohesion also makes it easier to understand what the class does, fix bugs without breaking other pieces, and reuse the code later. In short, a cohesive class focuses on one responsibility instead of trying to do a little bit of everything.

1. **Analyze *StudentPortalHelper* and propose refactor.**

**Answer:**

The *StudentPortalHelper* has **Low cohesion**. The class mixes unrelated responsibilities such as:

* GPA calculation (business rule)
* CSV export (I/O)
* Email Template (presentation/content)
* Date formatting for UI (presentation concern)
* Payment processing (integration)
* Password policy (security)
* Ad-hoc cache (in-memory storage)

These tasks aren’t closely related, the class shows low cohesion and mixes presentation logic, business logic, and data handling all in one place.

Refactor proposal:

If I were refactoring it, I’d separate each concern into its own smaller classes such as:

* *GradeService* (Compute GPA)
* *RosterExporter* (exportRosterToCsv)
* *EmailTemplates* (makeWelcomeEmail)
* *UiFormatters* (formatDateForUi)
* *PaymentService* (processTuitionPayment)
* *PasswordPolicy* (isStrongPassword)
* Cache<K,V> or use java.util.Map via a proper caching component

Each of these classes would focus on one clear purpose, making the system easier to test and maintain. Implementing this idea would result in higher cohesion, lower Coupling, clearer ownership of changes and easier unit testing. Following this single-responsibility idea reflects Riel’s heuristics for a good object-oriented design.

**# QUESTION 2**

**Answer:**

Using method overloading is cleaner and more maintainable than naming methods like *rectangleArea()* or *circleArea()*. Overloading lets us reuse the same method name “area” for different shapes, keeping the interface consistent and demonstrating polymorphism without cluttering the class with many similar names.

**# QUESTION 3**

1. **Does the current structure support runtime trim change?**

**Answer:**

Right now, the trim level (Base, Luxury, or Sport) is represented through inheritance, where each trim is its own subclass of *Car*. That means once you create a *BaseCar*, it stays a *BaseCar* and you can’t simply “upgrade” it to a *LuxuryCar* without making a new object. Because of that, the current structure does not support changing trims at runtime.

1. **Refactor using composition.**

**Answer:**

A better approach would use composition instead of inheritance. The *Car* class could hold a TrimLevel field that follows an interface with methods like *getName()* or *getFeatures()*. Then each trim (Base, Luxury, Sport) becomes its own class that implements that interface. If a user wants to change trims, the program can just call *car.setTrimLevel(new LuxuryTrim())*. This design is more flexible and avoids hard-coding behavior inside subclasses.

**# QUESTION 4 RATIONALE**

* **Why is *Device* defined as an abstract class?**

**Answer:**

*Device* is defined as an abstract class because it encapsulates common state/behavior (id, location, heartbeat, connection flag) and requires subclasses to define *getStatus()*, enforcinf a shared contract without all allowing direct instantiation

* **How do the *Networked* and *BatteryPowered* interfaces add behavior to your concrete classes?**

**Answer:**

*Networked* and *BatteryPowered* declare orthogonal capabilities that different devices can implement, enabling interface-based polymorphism.

* **Is this design an example of multiple inheritance in java? Explain why or why not.**

**Answer:**

Java doesn’t support multiple inheritance of classes. So, this isn’t multiple inheritance in the C++ sense, because Java only allows a class to extend one superclass. However, a class can implement many interfaces, which achieves the same goal of combining multiple behaviors safely.

**# QUESTION 5**

**Answer:**

Throughout this course, I have used AI tools like ChatGPT to clarify assignment requirements, review syntax, and provide short code examples when I wasn’t sure how to start implementing a certain function. Sometimes I asked AI to show me a basic example of how something works, like how to overload a method, use an abstract class, or handle exceptions, so I could understand the structure before writing my code. These examples helped me think through design choices and spot logical mistakes before compiling.

However, I also learned that AI can oversimplify problems or give code that compiles but doesn’t fully meet some requirements, so I always make sure to verify adjust it to fit the exact requirements. AI tools made me a faster and more confident programmer by offering feedback and quick demonstrations, but I’m aware of the risk of overreliance. Going forward, I’ll continue using AI mainly for brainstorming, conceptual explanations, and coding references while ensuring that I fully understand the reasoning and implementation.