Question 1:

### **Part a) High vs. Low Cohesion (5 pts.)**

**A well-designed class should have HIGH cohesion.**

High cohesion means all the methods in a class work together toward one clear purpose. This aligns with Arthur Riel's Heuristic 2.8: "A class should capture one and only one key abstraction."

When a class has high cohesion, it's easier to understand, test, and maintain. You know exactly what it does, and changes to one part don't break unrelated functionality. Low cohesion creates messy "utility dump" classes where everything is thrown together, making the code confusing and brittle.

### **Part b) Cohesion Assessment and Redesign (15 pts.)**

**This class has LOW cohesion.**

**Why?** The StudentPortalHelper class tries to do way too many unrelated things:

* Calculate GPAs (academic logic)
* Export CSV files (file I/O)
* Format emails and dates (UI/presentation)
* Process payments (external integration)
* Validate passwords (security)
* Manage a cache (data infrastructure)

These responsibilities have nothing to do with each other. They'd change for completely different reasons—a new payment gateway has nothing to do with how we format dates. This violates Heuristic 2.8 (one key abstraction per class) and creates a "God class" that's hard to maintain and test.

**How to fix it:**

Break this into separate, focused classes:

* **GradeCalculator** - handles GPA computation
* **RosterExporter** - exports rosters to files
* **EmailFormatter** - creates email messages
* **DateFormatter** - formats dates for UI
* **PaymentService** - processes tuition payments
* **PasswordValidator** - checks password strength
* **CacheService** - manages cached data (or use an existing framework)

Each class would have one clear job and high cohesion. This makes the system easier to understand, test, and modify without breaking unrelated features.

**Question 3 Answer**

## **a) Why the current structure does NOT support dynamic trim-level changes (10 pts)**

The current structure uses **inheritance** where Base, Sports, and Luxury are subclasses of Car. This creates a fundamental problem:

1. **Fixed at instantiation**: When you create a new Sports(), that object is permanently a Sports car. You cannot change an object's class type after it's created in Java.
2. **Cannot switch trim levels**: To change from Sports to Luxury, you would need to:
   1. Create an entirely new Luxury object
   2. Copy all the state from the old Sports object
   3. Replace all references to the old object
   4. This breaks object identity and is inefficient
3. **getTrimLevel() is in Car**: This suggests trim level is an inherent property tied to the class hierarchy, not a changeable attribute.
4. **Contradicts the requirement**: The problem states customers can "choose to change the trim level" during manufacturing, but inheritance makes this impossible without destroying and recreating objects.

## **b) Refactoring solution using composition (10 pts)**

**Solution: Replace inheritance with composition**

Create a separate TrimLevel class hierarchy:

TrimLevel (interface/abstract class)  
├── BaseTrim  
├── SportsTrim  
└── LuxuryTrim

Modify Car to use composition:

class Car {  
 private TrimLevel trimLevel; // Composition  
 private Engine engine;  
   
 public void setTrimLevel(TrimLevel newTrim) {  
 this.trimLevel = newTrim;  
 }  
   
 public TrimLevel getTrimLevel() {  
 return trimLevel;  
 }  
}

**Benefits:**

* Change trim dynamically: car.setTrimLevel(new LuxuryTrim())
* Same Car object, different behavior
* Matches the Engine design (also uses composition)
* Maintains object identity throughout manufacturing process

# **Question 4 Rationale (6 pts)**

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

Device is abstract because it defines common behavior and state (id, location, connected status, heartbeat) that all devices share, but the getStatus() method must be implemented differently by each device type. It provides a template with concrete methods like heartbeat() while forcing subclasses to define their own status format.

## **How do the Networked and BatteryPowered interfaces add behavior?**

The interfaces define contracts for specific capabilities that only some devices have. Networked adds connect/disconnect behavior (implemented by all three devices), while BatteryPowered adds battery management (only DoorLock and Camera). This allows polymorphic treatment—code can work with any BatteryPowered device without knowing if it's a DoorLock or Camera.

## **Is this design an example of multiple inheritance in Java?**

No, this is not true multiple inheritance. Java doesn't allow extending multiple classes, but it does allow implementing multiple interfaces. DoorLock extends one class (Device) and implements two interfaces (Networked, BatteryPowered). Interfaces only provide method signatures, not implementation or state, so there's no ambiguity like in languages with true multiple inheritance.

**Question 5 Answer - Reflection on AI Use (10 pts)**

My personal experience with using AI before this course consisted of mostly school-related topics, and the way that I use AI in this course is very similar to how I used it beforehand. This includes things like researching topics, strengthening code quality, and understanding different error messages and how to remediate them. With this being the case, I feel confident in my ability to wield the tool to my advantage while still getting my learning's worth. Some of the benefits I encountered in this course were the AI's expertise in many different IDEs and programming languages. I needed help several times with how to work in Eclipse, and Claude for the most part was able to solve all of my problems. However, some limitations came in the same manner—on occasion, if I had trouble with pushing files to GitHub, or my current problem which is fixing my file tree structure, I haven't been able to solve it as easily. The AI struggles with environment-specific issues that require deeper context about my particular setup, which reminds me that these tools have boundaries in what they can diagnose and fix remotely.

As I look ahead, I expect AI to influence the way I solve problems by allowing me full insight into what I am making decisions on. I for one like to maintain autonomy on certain subjects and decisions, but artificial intelligence will allow for the information needed to be readily accessible, more so than ever before. Rather than replacing my judgment, AI serves as a powerful research assistant that can quickly surface relevant documentation, explain unfamiliar concepts, and suggest alternative approaches I might not have considered. This means I can make more informed decisions faster, while still maintaining control over the final implementation and architectural choices. In my future career, I anticipate using AI to handle routine debugging and research tasks, freeing up mental energy for the creative and strategic aspects of software development that truly require human insight and experience.