Large Scale Programming Midterm

Question 1

a, A well-designed class should have high cohesion.
Reasoning: High cohesion means a class groups related responsibilities and has a single, well-defined purpose. This increases understandability, testability, and maintainability. Low cohesion (a "grab bag" of unrelated methods) leads to brittle code, harder testing, and difficulty reasoning about dependencies and side effects.

Arthur Riel heuristic(s) supporting this:

- *“A class should represent a single abstraction”* a class should have only behaviors that are directly related to that abstraction.

- *“Cohesion increases when methods operate on the same set of instance data”* — tightly related methods use shared fields; unrelated methods often act as procedural utilities.

b, The StudentPortalHelper class demonstrates low cohesion. This violates Arthur Riel's heuristic that states "Beware of classes that have many accessor methods defined in their public interface" and "A class should capture one and only one key abstraction."

Evidence of low cohesion:

1. Multiple unrelated responsibilities: GPA calculation, file I/O, email formatting, UI concerns (date formatting), payment processing, security (password validation), and caching
2. No clear single purpose: The class name suggests it helps with a student portal, but it performs diverse, unrelated tasks
3. "God class" pattern: This utility class tries to do everything related to students, violating the Single Responsibility Principle

Refactoring Approach:

I would decompose this class into multiple focused classes, each with high cohesion:

1. GradeCalculator - handles computeGPA() only
2. RosterExporter - handles exportRosterToCsv()
3. EmailService - handles makeWelcomeEmail()
4. UIFormatter - handles formatDateForUi()
5. PaymentProcessor - handles processTuitionPayment()

6. PasswordValidator - handles isStrongPassword()
7. CacheManager - handles putCache() and getCache()

Each class would focus on a single responsibility, making the system more maintainable, testable, and adhering to solid principles. The original StudentPortalHelper could become a facade that coordinates these services if needed, but each service would independently handle its specific domain concern.

Question 3

a, The current structure does NOT support changing a car's trim level during the manufacturing process for the following reasons:

1. Static Inheritance Hierarchy: The trim levels (Base, Sports, Luxury) are modeled as subclasses of Car. Once a car object is instantiated (e.g., new Sports()), its class type is fixed at compile-time. In Java, you cannot change an object's class at runtime a Sports object cannot become a Luxury object without creating an entirely new instance.

2. Loss of State: If a customer wants to change from Base to Luxury trim during manufacturing, the system would need to:

- Destroy the existing Base object
- Create a new Luxury object
- Transfer all manufacturing state (build progress, VIN, customer info, engine configuration) to the new object

This is error-prone, inefficient, and violates the principle of object identity preservation.

3. Violation of Open/Closed Principle: The Car class hierarchy is closed to runtime modification. Adding the ability to change trim would require significant refactoring of the inheritance structure, potentially breaking existing code.

4. getTrimLevel() Returns Fixed Value: The getTrimLevel() method likely returns a hardcoded value based on the subclass type (e.g., Sports.getTrimLevel() returns "Sports"). This cannot change without changing the object's type.

In summary: Using inheritance to model trim levels treats trim as part of the car's identity rather than as a changeable attribute, making dynamic trim changes impossible without destroying and recreating objects.

b, To enable dynamic trim-level changes, refactor the design to use composition with the Strategy Pattern:

Proposed Design:

```
Car (concrete class)

  - vin: String

  - engine: Engine

  - trimLevel: TrimLevel  (composition - THIS IS KEY)

  + getTrimLevel(): TrimLevel

  + setTrimLevel(TrimLevel): void

  + getEngine(): Engine

  + setEngine(Engine): void


<<interface>> TrimLevel

  + getName(): String

  + getFeatures(): List<String>

  + getPriceModifier(): double


BaseTrim implements TrimLevel

SportsTrim implements TrimLevel

LuxuryTrim implements TrimLevel


Engine (remains as-is)

   ├── Electric

   └── Petrol
```

Key Modifications:

1. Convert trim to a HAS-A relationship: Instead of Car being subclassed by Base/Sports/Luxury, Car now contains a TrimLevel object as a field.

5) Brief Rationale

1. Why is Device defined as an abstract class?

Device is defined as abstract because it represents a general concept that should not be instantiated directly. It provides common functionality (id, location, heartbeat, connection state) that all concrete devices share, while requiring subclasses to implement device-specific behavior through the abstract getStatus() method. This enforces a contract that all devices must provide status information while allowing implementation flexibility.

2. How do the Networked and BatteryPowered interfaces add behavior to concrete classes?

The Networked and BatteryPowered interfaces define contracts for optional capabilities that devices may have. Not all devices need both capabilities (e.g., Thermostat is only Networked, not BatteryPowered). By implementing these interfaces, concrete classes gain the ability to be treated polymorphically based on their capabilities. For example, the Main class can iterate through all devices and connect only those that implement Networked, without knowing their specific types. This demonstrates interface-based polymorphism and allows flexible composition of behaviors.

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

Explain why or why not. This is NOT true for multiple inheritance. Java does not support multiple inheritance of classes (a class cannot extend multiple classes). However, Java does support multiple interface implementation, which is what we use here. DoorLock and Camera implement both Networked and BatteryPowered interfaces, allowing them to inherit method contracts (but not implementation) from multiple sources. They still extend only one class (Device). This is sometimes called "multiple inheritance of type" but differs from languages like C++ where you can inherit implementation from multiple parent classes. Java's approach avoids the "diamond problem" while still providing behavioral flexibility through interfaces.

Question 5: Reflection on AI Use

Throughout this course, I have used AI tools like ChatGPT and Claude primarily for three purposes: explaining unfamiliar concepts with concrete examples, debugging code by analyzing error messages, and verifying my understanding of design patterns and Java syntax. The main benefit has been receiving immediate, tailored explanations that help me learn faster. However, I've encountered limitations. AI sometimes provides outdated information or makes incorrect assumptions about project requirements, forcing me to critically evaluate outputs rather than blindly accepting them.

Looking ahead, I expect AI to significantly influence how I solve problems both academically and professionally. It will accelerate routine tasks like generating boilerplate code, researching unfamiliar APIs, and exploring alternative solutions quickly. However, complex problems requiring architectural decisions, domain expertise, or business context understanding will still demand human judgment. The key skill will be knowing when to leverage AI as an intelligent assistant versus when independent critical thinking is essential. Success will come from combining strong fundamentals with effective AI collaboration, not from replacing one with the other.