Jean Joiles

Large Scale Programming Midterm

October 20,2025

1)Using one or more Arthur Riel heuristics, analyze whether the StudentPortalHelper class demonstrates high or low cohesion.

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**, meaning that all of its methods and responsibilities are closely related and support one clear purpose. High cohesion makes a class easier to understand, maintain, and modify, and it follows Arthur Riel’s heuristic that a class should have only one reason to change. When a class stays focused on a single responsibility, developers can update or extend it without risking unexpected side effects in unrelated areas.

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)

*(If you believe the class already has good cohesion, justify why no changes are necessary.)*

The StudentPortalHelper class has low cohesion because it combines many unrelated concerns—GPA calculation, CSV export, email formatting, UI date formatting, payment processing, password validation, and caching into one utility class. According to Riel’s heuristics, this violates cohesive design by creating a “kitchen-sink” class with multiple reasons to change. To improve the structure, the class should be refactored into multiple smaller classes such as GpaCalculator, RosterCsvExporter, EmailFormatter, PaymentService, and PasswordPolicy. Each class would handle a single responsibility, increasing cohesion, improving maintainability, and making the system easier to test and evolve.

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

If we model trim levels (Base, Luxury, Sport) as subclasses of Car (for example, BaseCar extends Car, LuxuryCar extends Car, and so on), we run into a major limitation: we can’t change the trim level on the same car object while it’s being built. Because the trim is baked into the class type itself, switching from **Base → Luxury** would mean creating a brand-new object and manually copying over all the car’s existing state. This risks losing the original object’s identitysuch as its ID or any references to it in work orders, queues, or tracking systems which breaks the idea that one car should move through manufacturing as a single continuous entity.

The situation becomes even worse if engine types (Electric, Petrol, etc.) are also represented through inheritance. You quickly end up with a messy explosion of subclasses like LuxuryElectricCar, SportPetrolCar, and many more. That not only clutters the design, but also makes it nearly impossible to reconfigure cars at runtime.

In short, tying trim and engine options to inheritance forces configuration to be part of the object’s type, which prevents clean, dynamic changes as the car progresses through the system.

b)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 support changing the trim level during the manufacturing process, the design should be refactored so that Car uses composition instead of inheritance. Instead of having separate subclasses such as BaseCar, LuxuryCar, or SportCar, the Car class should contain a Trim object as one of its internal fields. In this design, the trim is no longer fixed by the class type; instead, it becomes a component that can be swapped at runtime. For example, the Car class could hold a reference like private Trim trim; and provide a setter method such as setTrim(Trim newTrim). When the customer decides to upgrade or downgrade, the program can simply replace the existing trim object with a different Trim implementation (e.g., new LuxuryTrim()), allowing the same Car instance to change configuration without recreating or duplicating it. This composition-based approach cleanly supports dynamic trim changes and avoids the limitations of subclass inheritance, where behaviors are locked in at object creation time and cannot be modified later.

4) Rational: Device is an abstract base class because it captures shared state and behavior (id, location, heartbeat, connection flag) that all devices need while requiring each concrete device to supply its own getStatus() implementation. The Networked and BatteryPowered interfaces add orthogonal capabilities; classes opt in to these behaviors and can be used via interface polymorphism (as shown in Main). This is **not** multiple inheritance of implementation—Java allows a single concrete superclass (Device) plus multiple interfaces, which provide behavior contracts without code duplication.

5) Before and during this course, I used AI tools as a way to support my programming and improve my understanding of new concepts. When I struggled with certain topics such as method overloading, inheritance, or exceptions I used AI to re-explain them in simpler language and provide small code examples I could learn from. While working on assignments, I also used AI to help interpret error messages, check for common mistakes, and suggest alternative approaches when I felt stuck. This made my learning process faster and less frustrating, because I didn’t have to pause for long periods searching through documentation or forums. However, I also noticed limitations: AI can sometimes produce incorrect code or give answers that don’t fully match the assignment requirements, so I still had to review everything carefully, rely on my notes, and make sure the final decisions came from my own understanding.

Looking ahead, I expect AI to continue influencing the way I learn and solve problems in both academic and professional environments. The biggest benefit is having a tool that can brainstorm ideas, speed up debugging, and explain complex topics in different ways until something finally “clicks.” At the same time, I see the importance of using AI responsibly not as a shortcut to avoid thinking, but as a companion that can guide me while I stay in control of the technical and creative decisions. In the future, I believe AI will help me write cleaner code, learn new programming tools more quickly, and become a more efficient problem solver, but I also know I must apply my own judgment, testing, and critical thinking to produce reliable work.