Question 1:

a) Cohesion: what should a well-designed class have, and why? (5 pts)

A well-designed class should have high cohesion.

As per Arthur Riel's heuristics (and consistent with SRP), a class must represent one clear abstraction and provide strongly related services. High cohesion makes a class easier to understand, maintain, test, and reuse and limits ripple effects when requirements change ("one reason to change"). Low cohesion—a multitude of unrelated responsibilities lumped together—leads to a "utility/God class" that is brittle and hard to modify.

b) Analysis of StudentPortalHelper and refactoring approach (15 pts)

Goal: Split by abstraction and layer, so each class has one responsibility and one reason to change.

Domain / academics

GpaCalculator (pure function/service): computeGPA(List<Integer>)

Roster / persistence

RosterCsvExporter (I/O concern): exportRosterToCsv(...)

Communication / presentation

WelcomeEmailFactory or EmailTemplates: makeWelcomeEmail(...)

Presentation / UI

Avoid UI formatting in domain; use a UI-side utility or view model: DateUiFormatter (or rely on the UI layer directly)

Payments / external systems

TuitionPaymentService interface + PaymentGateway adapter (inject gateway; handle success/failure, errors, retries)

Security

PasswordPolicy / PasswordStrengthChecker: isStrongPassword(...) (make rules configurable/testable)

Infrastructure / caching

Replace ad-hoc static map with a dedicated cache abstraction: Cache<K,V> or use a library; if needed, wrap as PortalCache to centralize eviction/TTL

Additional design improvements (aligning with Riel’s heuristics):

Eliminate “Helper” dumping ground. Name classes after nouns representing clear concepts (Riel: “Prefer noun-phrased class names to verb-phrased ‘helpers’.”)

Avoid static state & utilities for testability and flexibility. Prefer instances and dependency injection (e.g., inject PaymentGateway, Clock, Cache).

Separate layers: keep domain logic (GPA) pure; move UI formatting out; isolate I/O (CSV, payments) behind ports/adapters.

Configuration over constants: e.g., password rules, date patterns, CSV headers should be configurable to avoid scattering policy.

Resulting package sketch:

org.howard.edu.lsp.student.domain

- GpaCalculator

org.howard.edu.lsp.student.presentation

- EmailTemplates

- DateUiFormatter (or do this in the UI tier)

org.howard.edu.lsp.student.infrastructure

- RosterCsvExporter

- PaymentGatewayStripe (implements TuitionPaymentService)

- PortalCache (wraps a real cache)

org.howard.edu.lsp.student.security

- PasswordStrengthChecker

- PasswordPolicy  
  
Question 3.  
**a) Does the current design let you change trims mid-build?**

**no, not cleanly**. In the typical setup for this kind of problem, trims like Base/Luxury/Sport are modeled as subclasses of Car (e.g., BaseCar, LuxuryCar, SportCar). The catch is that in Java you can’t change an object’s **class** once it’s created. So if a customer switches from Base to Luxury during production, you’d have to create a brand-new LuxuryCar and copy everything over (VIN, current station, options already installed, audit trail, etc.). That’s clunky and risky—references elsewhere in the system may still point to the old object, and you can easily miss copying some state. It also leads to a combinatorial mess if you mix trims with engine types (you end up with classes like SportElectricCar, LuxuryPetrolCar, etc.). Bottom line: baking the trim into the inheritance tree makes live trim changes awkward and error-prone.

**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.)  
  
Question 5.**Before and during this course, I’ve used AI tools mainly as a study partner and debugging assistant. For example, when I got stuck on why a unit test was failing or what a stack trace really meant, I’d paste the error context and ask for a plain-English explanation and a few hypotheses to try. I’ve also used AI to brainstorm test cases, outline class designs (e.g., separating concerns, naming methods), and to generate small code snippets like regexes or CSV parsing loops that I then rewrote to match our style. For readings and lectures, I’d ask for summaries of dense concepts (e.g., cohesion vs. coupling, strategy vs. inheritance) and then verify with the textbook/docs before I committed anything to code.

The benefits were speed and clarity: faster feedback than searching blindly, more ideas for edge cases, and better vocabulary to explain my decisions. The limitations were real, though—sometimes the tool was confidently wrong, mixed library versions, or suggested code that didn’t compile in our environment. I learned to treat outputs as drafts, to test everything, and to cross-check with official docs. Going forward, I expect AI to keep acting like a “junior pair-programmer”: great for scaffolding, code reviews, and quick references, while I stay responsible for design choices, security/privacy considerations, and final verification. In short, AI will help me move faster, but it won’t replace reading specs, writing tests, or understanding the systems I build.