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

* A well-design class should have high cohesion. High cohesion means that a class’s responsibilities are strongly related and focused on a single conceptual purpose. According to Arthur Riel’s heuristics, classes should group related data and behavior;high cohesion improves maintainability, testability, readability and reduces coupling because responsibilities are not scattered across unrelated methods. When a class performs many unrelated tasks it becomes harder to change, harder to unit test, and introduces hidden dependencies.

1. **Analysis of StudentPortalHelper and suggested refactoring**

**Cohension assessment:**

* StudentPortalHelper demonstrates low cohesion. Using Riel’s heuristics especially “Keep related data and behavior in one place” and “Avoid god classes”, here this class violates those guideliness. It mixes presentation/UI formatting (formatDateForUi), CSV export (exportRosterToCsv), GPA calculation (computeGPA), payment processing (processTuitionPayment), password validation (isStrongPassword), caching (getCache/putCache), and email content creation (makeWelcomeEmail) in a single utility class. These responsibilities are conceptually unrelated.

**Refactoring approach (high level):**

1. Split into focused lasses (one responsibility each): Like
   1. GpaCalculate - computeGPA(List<Integer>) and any grade normalization.
   2. RosterExporter - exportRosterToCsv(String filename, List<String> names) , it handles file I/O and error reporting.
   3. EmailComposer - makeWelcomeEmail(String studentName) and other template functions.
   4. UiFormatter or DateFormatter - formateDateForUi(LocalDate)
   5. PaymentProcessor- processTuitionPayment(String studentId, double amount) - this would depend on PaymentGateway interface.
   6. PasswordValidator - isStrongPassword(String pwd)
   7. CacheManager - wrapper for cache map (abstracted so it can be swapped out with a real cache).
2. Use dependency injection and small interfaces where appropriate: e.g. PaymentProcessor depends on PaymentGateway interface so it can be mocked in tests and replaced with a real implementation later
3. Move constants and configuration (Date formats, CSV headers) into configuration holders or constants classes.
4. Unit tests: each small class is easily unit testable, side effects (file, payment gateways) are isolated and mocked.

Hence, The StudentPortalHelper class has low cohesion. Refactoring it into multiple, well-named classes focused on single responsibilities as listed above. This improves Riel’s heuristic, smaller classes with focused behaviors, fewer surprises and decreased coupling.

**Question3:**

1. **Explain in detail why the current structure does or doesnot support this.**

* So, based on the UML, if Car obtains trim-level behavior by static inheritance (i.e. Base, Sports, Luxury are subclasses of Car), then switching a car between trim levels at runtime is not supported: object identity and class membership are fixed at instantiation time in Java. If the trim level is implemented as the concrete class of the car (new Sports() vs new Luxury()), we cannot simply change a car’s class instance to another subclass without creating a new object, this breaks continuity of state and identity if we want a persistent single Car object whose trim changes.

1. **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?**

* Here, we can use composition like giving Car a Trim field (an interface) instead of relying on subclassing for trim .
* “ interface Trim {
* String getTrimName();
* double getPerformanceFactor();
* // other trim-specific behaviors
* }
* class BaseTrim implements Trim { ... }
* class SportTrim implements Trim { ... }
* class LuxuryTrim implements Trim { ... }
* class Car {
* private Trim trim; // composition
* private Engine engine; // composition for engine types
* // ...
* public String getTrimLevel() { return trim.getTrimName(); }
* public void setTrimLevel(Trim newTrim) { this.trim = newTrim; }
* }

“

* Car maintains its identity and state (VIN, serial, options, production progress) while its trim object can be replaced at any time, setTrimLevel(...) performs the change. This supports dynamic trim changes without reconstructing the entire Car.
* Using a Trim interface supports polymorphism for trim behavior and avoids class explosion. The engine (electric or petrol) should similarly be a component Engine with subtypes ElectricEngine and PetrolEngine, assigned to the Car at runtime
* Some other considerations can be like: If some trim change requires structural changes (e.g. different parts that require rework), the setter can validate and execute reconfiguration steps.
* This refactor follows Arthur Riel’sheuristic to “Prefer delegation over inheritance” when dynamic behavior is needed and increases flexibility while keeping responsibilities clear.

**Question 4:**

1. Device is defined aas an abstractclass because it represents a general concept ( a hardware device) that provides shared attributes (ID, location, connection status, heartbeat) and common behavior for all concrete device types but it should not be instantiated directly. So, declaring it as abstract enforces that only specific subclasses can create real objects while inheriting those core features.
2. Network defines network-related behavior (connect(), disconnect(), isConnected()), allowing a Device subclass to communicate with the system. BatterYPowered defines battery-specific behavior(getBatteryPercent(), setBatteryPercent()), letting only devices with batteries report or validate charge levels.

DoorLock and Camera implement this interface, while Thermostat doesnot .

So, by implementing these interfaces, each concrete device class inherits the shared base structure from Device and gains additional behaviors determined by which interfaces it implements. This approach aligns with interface segregation and allows the system to treat devices polymorphically based on capabilities instead of class type.

1. This design is not an example of class-based multiple inheritance. In Java, a class can extend only one superclass (Device) but can implement multiple interfaces (Networked, BatteryPowered). Thus, although a class like Camera inherits behavior and fields from a single abstract base (Device), it can also acquire additional capabilities from multiple interfaces.

**Question 5:**

* During this midterm, I used AI tools ChatGpt for brainstorming design decomposition, and also to get quick examples for Queestion 4 abstract classes.
* The AI helped speed up iteration by suggesting reasonable class splits and also for phrasing Javadocs, however I validated and adapted some suggestion because Ai responses can be imprecise or contain subtle errors for some edge cases.
* The key benefits were productivity gain and exposure to different edge cases. Limitations included occasional incorrect details (so I always cross-checked).
* Looking ahead, I expect AI to remain a companion for brainstorming but I will continue to critically verify outputs, keep academic integrity and records prompts/excerpts for reproducibility.