**Howard University**

**College of Engineering and Architecture**

**Department of Electrical Engineering & Computer Science**

**Large Scale Programming**

**Fall 2025**

**Midterm Exam**

October 20, 2025

## ****Instructions****

* **Exam Format:**  
  Your examination consists of both **essay** and **programming** problems.
* **Essay Questions:**  
  Complete all essay (rationale) questions **inline in this document**.  
  My preference is **Microsoft Word (.docx)**, but **.txt** or **.pdf** are acceptable alternatives.  
  Upload your completed essay file to your repository under the package:

org.howard.edu.lsp.midterm.doc

You may use **any file name**.

* **Programming Problems:**  
  Each programming problem must be uploaded to your repository using the package specified in the question.  
  For example:

org.howard.edu.lsp.midterm.question1

* **Committing Your Work:**  
  If using a third-party IDE or tool to commit, commit early and often.  
  Do not wait until the end of the exam to push your code.  
  If you encounter problems committing, you may manually upload your code to your repository.  
  If you are unable to commit or upload, you may zip your project and email it to  
  bwoolfolk@whiteboardfederal.com.  
  ⚠️ This will result in a 20% deduction from your final exam score.
* **Citations and References:**  
  You must cite all references for any material obtained from the internet.  
  Any AI-generated content (e.g., ChatGPT conversations) must be included in full.  
  Each package you upload must include a references document corresponding to that package’s content.  
  ⚠️ Failure to provide references will result in a zero for that question.
* **Exam Policy:**  
  This is an OPEN BOOK, OPEN NOTES exam.  
  Collaboration of any kind is strictly prohibited. Any violations will be handled in accordance with **university academic integrity guidelines**.

**Question 1. (20 pts.)**

**Given the following, analyze the class below and answer the below questions. This question does NOT require you to write any code.**

package org.howard.edu.lsp.studentPortalHelper;

import java.io.\*;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.util.\*;

public class StudentPortalHelper {

// Data cache (in-memory)

private static final Map<String, String> cache = new HashMap<>();

// GPA calculation

public static double computeGPA(List<Integer> grades) {

if (grades == null || grades.isEmpty()) return 0.0;

int sum = 0;

int count = 0;

for (int g : grades) { sum += g; count++; }

double avg = (double) sum / count;

// simple mapping: 90–100=A=4, 80–89=B=3, etc.

if (avg >= 90) return 4.0;

if (avg >= 80) return 3.0;

if (avg >= 70) return 2.0;

if (avg >= 60) return 1.0;

return 0.0;

}

// CSV export to disk

public static void exportRosterToCsv(String filename, List<String> names) {

try (PrintWriter pw = new PrintWriter(new FileWriter(filename))) {

pw.println("name");

for (String n : names) {

pw.println(n);

}

} catch (IOException e) {

System.err.println("Failed to export roster: " + e.getMessage());

}

}

// Email formatting

public static String makeWelcomeEmail(String studentName) {

return "Welcome " + studentName + "! Please visit the portal to update your profile.";

}

// Date formatting (UI concern)

public static String formatDateForUi(LocalDate date) {

return date.format(DateTimeFormatter.ofPattern("MM/dd/yyyy"));

}

// Payment processing (stub)

public static boolean processTuitionPayment(String studentId, double amount) {

if (amount <= 0) return false;

// pretend to call external gateway...

return true;

}

// Password strength check (security)

public static boolean isStrongPassword(String pwd) {

if (pwd == null || pwd.length() < 8) return false;

boolean hasDigit = false, hasUpper = false;

for (char c : pwd.toCharArray()) {

if (Character.isDigit(c)) hasDigit = true;

if (Character.isUpperCase(c)) hasUpper = true;

}

return hasDigit && hasUpper;

}

// Ad-hoc caching

public static void putCache(String key, String value) {

cache.put(key, value);

}

public static String getCache(String key) {

return cache.get(key);

}

}

**Tasks:**

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. Cohesion is the principle where a class is designed to have a specific purpose. High cohesion means that the class is easier to maintain and can be reused more than low cohesion classes.

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.)*

I believe the class has low cohesion. According to Arthur Riel’s object-oriented design heuristics, a class should represent only one concept or purpose. This class has multiple unrelated responsibilities which violates this heuristic. Since they are all in one class, it reduces the reusability and maintainability of the design. A general approach to refactoring the class would be to split the class into several other classes that each handle a single responsibility or purpose. GPA calculation would be in the GpaCalculator class, roster export would be in the RosterExporter class, email formatting would be in the EmailFormatting class, date formatting would be in the UiFormatting class, payments and payment handling would be in the Payment class, security would be in PasswordValidator class and caching would be in the CacheManager class. This would ensure that the program is easier to maintain and reuse.

**Reference:**

[**https://www.geeksforgeeks.org/java/cohesion-in-java/**](https://www.geeksforgeeks.org/java/cohesion-in-java/)

**AI Transcript:**

**Prompt: How can I make my response better?  
Response: Excellent — you’re on the right track. Your part (a) is well-written, and your part (b) starts strong. Here’s how you can complete and polish part (b) to earn full credit (15/15 points) based on Arthur Riel’s heuristics and cohesion principles 👇**

**(b) Analysis and Refactoring Plan**

**I believe the StudentPortalHelper class demonstrates low cohesion. According to Arthur Riel’s object-oriented design heuristics, *“A class should represent one and only one concept”* and *“Classes that contain unrelated behaviors should be split.”***

**This class currently violates that heuristic. It mixes multiple, unrelated responsibilities:**

* **Academic logic: computeGPA()**
* **File I/O: exportRosterToCsv()**
* **Email/message formatting: makeWelcomeEmail()**
* **UI concerns: formatDateForUi()**
* **Financial logic: processTuitionPayment()**
* **Security: isStrongPassword()**
* **Caching/data storage: putCache() and getCache()**

**Each of these areas belongs to a different subsystem (academics, communications, payments, security, etc.). By combining them into one “helper” class, the design reduces readability, reusability, and maintainability. Any change in one concern (for example, updating payment logic) risks introducing side effects or confusion in others (like GPA calculation).**

**(b) Recommended Refactoring**

**To improve cohesion, the class should be split into several focused classes, each handling a single responsibility. For example:**

| **Concern** | **New Class** | **Responsibility** |
| --- | --- | --- |
| **GPA calculation** | **GpaCalculator** | **Compute GPAs, grade averages, etc.** |
| **Roster export** | **RosterExporter** | **Handle CSV file output for student rosters.** |
| **Email formatting** | **EmailService** | **Generate or send student emails.** |
| **Date formatting (UI)** | **UiFormatter** | **Format dates for the user interface.** |
| **Payment handling** | **PaymentProcessor** | **Process tuition or fee payments.** |
| **Security** | **PasswordValidator** | **Check password strength and rules.** |
| **Caching** | **CacheManager** | **Manage temporary in-memory cache.** |

**This refactoring would yield smaller, cohesive classes that are easier to maintain, test, and reuse independently.**

**(b) Summary Sentence**

**In summary, StudentPortalHelper violates Riel’s heuristic that a class should have one purpose and therefore exhibits low cohesion. Refactoring it into multiple single-purpose classes would align with object-oriented design best practices and result in higher cohesion and a more maintainable architecture.**

**✅ Grading Expectation:**

* **Identifies low cohesion (✔)**
* **References a Riel heuristic correctly (✔)**
* **Explains why the class violates it (✔)**
* **Proposes a clear restructuring plan (✔)**

**→ This would earn full credit (15/15 points) on part (b).**

**Would you like me to show a short paragraph version (for a written exam answer) that fits neatly within 2–3 concise paragraphs?**

**Question 2. (20 pts.)**

Write a class AreaCalculator in the package org.howard.edu.lsp.midterm.question2 with the following **overloaded methods**: This should be uploaded to your repo.

// Circle area

public static double area(double radius)

// Rectangle area

public static double area(double width, double height)

// Triangle (base & height) area

public static double area(int base, int height)

// Square (side length) area

public static double area(int side)

**Requirements:**

Each method should compute and return the correct area.

* Circle area: π (use class Math.PI) × r²
* Rectangle area: width × height
* Triangle area: ½ × base × height
* Square area: side²
* For all methods: throw an IllegalArgumentException if any dimension is ≤ 0.

Create a class named Main that invokes each overloaded method **statically** to produce **exactly** the following output:

Circle radius 3.0 → area = 28.274333882308138

Rectangle 5.0 x 2.0 → area = 10.0

Triangle base 10, height 6 → area = 30.0

Square side 4 → area = 16.0

Finally, invoke **at least one** of the area methods with a value that causes an IllegalArgumentException to be thrown.

* Catch the exception using a try/catch block.
* Print an **error message** to System.out. (Any message is fine.)

Briefly (2–3 sentences as a comment in class Main) explain if **overloading** or simply use methods with different names, i.e., rectactangleArea, circleArea, etc..

| **Category** | **Description** | **Points** |
| --- | --- | --- |
| **1. Implementation** | Correct use of **method overloading** (same name, different signatures), correct formulas, and proper exception handling in each method. | **10** |
| **2. Program Behavior** | Main correctly invokes all methods statically, produces the required output exactly, and includes a working exception demonstration. | **6** |
| **3. Conceptual Understanding** | Brief explanation of why or why not overloading is the better design choice. | **4** |

**Question 3.**

**Given the following, answer the below questions.**

**(20 pts.)**

**Given:**

A car manufacturer uses Java software to track current vehicles being built. The UML diagram below shows an excerpt of the current software structure. You should assume the presence of other appropriate fields and methods

**Car**

getTrimLevel()

**Engine**

accelerate()

getFuelLevel()

**Base**

**Sports**

**Luxury**

**Electric**

**Petrol**

Each car can be built to one of three trim levels: Base, Luxury or Sport. They can also be configured with an electric or petrol engine. At various points in the manufacturing process the customer can choose to change the trim level.

**Task:**

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

The current structure does not support changing the car’s trim levels. The diagram shows the trim levels Base, Sport and Luxury as subclasses of the Car class. Inheritance fixes an object’s type when it is created. An instance cannot be changed from one level to another at runtime such as from Sport to Luxury. Thus, the current structure does not support the customers changing their trim level during manufacturing since this would require creating a new object rather than modifying the existing one.

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? (10 pts.)

I would design an interface such as TrimBehavior with methods that allow trim-specific behavior such as applyFeatures(Car) etc. All the trim levels would implement TrimBehavior and the Car class would keep a TrimBehavior trim field and would delegate trim-specific work to that field which would be trim.applyFeatures(this). Customers could change the trim using car.setTrim(new (the trim level eg. SportTrim()).

**AI Transcript:**

Prompt: How am I able to answer this question?  
Response: Short answer: No — as drawn it does not support changing a car’s trim level at runtime (i.e. while a Car instance is being built). The UML shows Base, Sport and Luxury as subclasses of Car. Because inheritance fixes an object’s type at instantiation, you cannot simply “switch” an instance from Base to Luxury. Below I explain why and give safe alternatives you can use instead.

Why the current design prevents runtime trim changes

1. Trim is modeled with inheritance (subclasses)  
   The diagram places Base, Sports, Luxury under Car as subtype boxes. That strongly implies trim levels are implemented as different subclasses of Car. In Java the class of an object is fixed once created — you cannot change an object’s class at runtime. So a new Base(...) object remains a Base object until it’s garbage-collected; you can’t turn that same object into a Luxury with a language feature.
2. Polymorphic behavior is tied to type  
   Any behavior that differs by trim (overridden methods, fields present only in certain trims) will be implemented by different method overrides / fields in subclasses. If you wanted to change trim you would need all code paths that interact with the car to start treating it as a different runtime type — which Java does not allow.
3. Workarounds are awkward and error-prone
   * You could create a new Luxury object, copy the shared state from the old Base (VIN, progress, options), and discard the old one. That works but:
     + It is clumsy and requires boilerplate copy code.
     + It can break identity semantics (is the new object the “same” car?).
     + Code that held references to the original Base instance will not automatically see the new instance.
   * You could attempt to change behavior by setting boolean flags externally, but that defeats the purpose of using subclasses in the first place and leads to a messy mix of inheritance + state flags.
4. Separation of concerns / persistence issues  
   If persistence (DB) or other subsystems rely on the concrete class, replacing objects or copying them is more complex (IDs, history, references, serialization).

How to support changeable trim (recommended designs)

You should model the trim as data/composition, not as subclasses. Here are common, clean approaches:

1) Simple: enum or field on Car

* Add a TrimLevel enum (BASE, SPORT, LUXURY) and a trim field on Car.
* Car methods check trim to decide behavior (or use helper methods).
* Changing trim is simply car.setTrim(TrimLevel.LUXURY) — trivial and safe.
* Pros: simplest, easy to persist & copy.
* Cons: if behavior per trim is complex (many different operations), if/else or switch statements can get verbose.

2) Strategy pattern (composition with polymorphism)

* Define an interface TrimPolicy or TrimBehavior with methods for trim-specific behavior, e.g. applyFeatures(Car), getPackagePriceAdjustment(), etc.
* Implement BaseTrim implements TrimBehavior, SportTrim implements TrimBehavior, LuxuryTrim implements TrimBehavior.
* Car keeps a TrimBehavior trim field and delegates trim-specific work to it: trim.applyFeatures(this).
* To change trim: car.setTrim(new LuxuryTrim()).
* Pros: preserves polymorphic behavior without changing object type; clean & extensible.

3) State pattern (if trim change triggers lifecycle changes)

* If trim change affects not only behavior but the *state machine* (allowed operations change depending on manufacturing phase), use the State pattern where Car delegates to a current TrimState object that can be swapped.

4) Decorator (if trim is “add-on” features)

* If trims are essentially compositions of optional features, the Decorator pattern can be used to attach feature sets; trimming becomes adding/removing decorators.
* More complex and usually overkill for simple trim enums.

Practical considerations and recommendation

* If trim is just a label and a few price/feature differences: use an enum + helper functions — simplest and perfectly fine for the assignment.
* If trims have complex, varying behavior (many methods differ): use the Strategy pattern so you can keep behavior polymorphic while still swapping at runtime.
* Avoid subclassing Car to represent trims unless trims are *permanent* and you never expect to change them after construction.

Example (conceptual, not code)

* Bad (current): class Base extends Car, class Luxury extends Car → cannot change instance type.
* Good: class Car { TrimBehavior trim; ... setTrim(TrimBehavior t) { this.trim = t; } ... }  
  Or simply class Car { TrimLevel trim; ... setTrim(TrimLevel t) { this.trim = t; } ... }.

Minor note about engines

The same logic applies to engines. If the UML shows Electric and Petrol as subclasses of Engine, you can still support swapping engines if Car holds an Engine reference (composition): car.setEngine(new ElectricEngine(...)). That is simpler than swapping the Car subclass, and is the recommended approach.

Final takeaway

The current subclass-per-trim design is fine if trims are immutable in your domain, but it does not support customers changing trim during manufacturing. To support dynamic trim changes, refactor so Car *has* a trim (data or strategy) instead of *is* a specific trim subclass. This preserves polymorphic behavior and makes runtime changes simple and safe.

**Question 4. (30 pts.)**

Design and implement a small **smart-campus device** system showing both **class inheritance** (concrete classes extend a common abstract class) and **interface implementation**.

**1) Abstract Base Class — Device**

**The following Device class is partially implemented for you.  
You must not modify this code, but you will use it in your subclasses:**

package org.howard.edu.lsp.midterm.question4;

public abstract class Device {

private String id;

private String location;

private long lastHeartbeatEpochSeconds;

private boolean connected;

// PROVIDED CONSTRUCTOR

public Device(String id, String location) {

if (id == null || id.isEmpty() || location == null || location.isEmpty()) {

throw new IllegalArgumentException("Invalid id or location");

}

this.id = id;

this.location = location;

this.lastHeartbeatEpochSeconds = 0;

this.connected = false;

}

public String getId() {

return id;

}

public String getLocation() {

return location;

}

public long getLastHeartbeatEpochSeconds() {

return lastHeartbeatEpochSeconds;

}

public boolean isConnected() {

return connected;

}

protected void setConnected(boolean connected) {

this.connected = connected;

}

public void heartbeat() {

this.lastHeartbeatEpochSeconds = System.currentTimeMillis() / 1000;

}

public abstract String getStatus();

}

**You will extend this class** in your DoorLock, Thermostat, and Camera implementations.  
All subclasses must call super(id, location) in their constructors.

**2) Capability Interfaces (behaviors only)**

**Networked**

void connect();

void disconnect();

boolean isConnected();

Behavior:

* connect() brings the device online by setting connected = true.
* disconnect() sets connected = false.
* isConnected() reports the current connection state.  
  (Concrete classes may satisfy this using Device’s protected setter and public getter.)

**BatteryPowered**

int getBatteryPercent(); // 0..100

void setBatteryPercent(int percent);

Behavior:

* getBatteryPercent() returns current battery %.
* setBatteryPercent(int) updates it; throw IllegalArgumentException if outside 0..100 inclusive.

**3) Concrete Devices (must extend Device and implement interfaces)**

**All fields must be private. Implement methods exactly as specified.**

1. **DoorLock — extends Device, implements Networked, BatteryPowered**

**Private fields**

private int batteryPercent;

**Constructor**

public DoorLock(String id, String location, int initialBattery)

* Call super(id, location).
* Initialize battery by calling setBatteryPercent(initialBattery) (enforces 0..100).

**Implemented methods**

// Networked

@Override public void connect() { setConnected(true); }

@Override public void disconnect() { setConnected(false); }

@Override public boolean isConnected() { return super.isConnected(); }

// BatteryPowered

@Override public int getBatteryPercent() { return batteryPercent; }

@Override public void setBatteryPercent(int percent) {

if (percent < 0 || percent > 100) throw new IllegalArgumentException("battery 0..100");

this.batteryPercent = percent;

}

// Status

@Override public String getStatus() {

String connStatus = isConnected() ? "up" : "down";

return "DoorLock[id=" + getId() + ", loc=" + getLocation() +

", conn=" + connStatus + ", batt=" + batteryPercent + "%]";

}

**B) Thermostat — extends Device, implements Networked**

**Private fields**

private double temperatureC;

**Constructor**

public Thermostat(String id, String location, double initialTempC)

* Call super(id, location).
* Initialize temperatureC to initialTempC.

**Accessors**

public double getTemperatureC();

public void setTemperatureC(double temperatureC);

**Implemented methods**

// Networked

@Override public void connect() { setConnected(true); }

@Override public void disconnect() { setConnected(false); }

@Override public boolean isConnected() { return super.isConnected(); }

// Status

@Override public String getStatus() {

String connStatus = isConnected() ? "up" : "down";

return "Thermostat[id=" + getId() + ", loc=" + getLocation() +

", conn=" + connStatus + ", tempC=" + temperatureC + "]";

}

**C) Camera — extends Device, implements Networked, BatteryPowered**

**Private fields**

**private int batteryPercent;**

**Constructor**

public Camera(String id, String location, int initialBattery)

* Call super(id, location).
* Initialize battery by calling setBatteryPercent(initialBattery).

**Implemented methods**

// Networked

@Override public void connect() { setConnected(true); }

@Override public void disconnect() { setConnected(false); }

@Override public boolean isConnected() { return super.isConnected(); }

// BatteryPowered

@Override public int getBatteryPercent() { return batteryPercent; }

@Override public void setBatteryPercent(int percent) {

if (percent < 0 || percent > 100) throw new IllegalArgumentException("battery 0..100");

this.batteryPercent = percent;

}

// Status

@Override public String getStatus() {

String connStatus = isConnected() ? "up" : "down";

return "Camera[id=" + getId() + ", loc=" + getLocation() +

", conn=" + connStatus + ", batt=" + batteryPercent + "%]";

}

**4) Provided Driver**

**Do not modify this file. Your classes must compile and run with it unchanged.**

package org.howard.edu.lsp.midterm.question4;

import java.util.\*;

public class Main {

public static void main(String[] args) {

Device lock = new DoorLock("DL-101", "DormA-1F", 85);

Device thermo = new Thermostat("TH-202", "Library-2F", 21.5);

Device cam = new Camera("CA-303", "Quad-North", 72);

// === Invalid battery test ===

System.out.println("\n== Exception test ==");

try {

Device badCam = new Camera("CA-404", "Test-Lab", -5);

System.out.println("ERROR: Exception was not thrown for invalid battery!");

} catch (IllegalArgumentException e) {

System.out.println("Caught expected exception: " + e.getMessage());

}

// === Heartbeat demonstration ===

System.out.println("\n== Heartbeat timestamps BEFORE ==");

for (Device d : Arrays.asList(lock, thermo, cam)) {

System.out.println(d.getId() + " lastHeartbeat=" + d.getLastHeartbeatEpochSeconds());

}

lock.heartbeat();

thermo.heartbeat();

cam.heartbeat();

System.out.println("\n== Heartbeat timestamps AFTER ==");

for (Device d : Arrays.asList(lock, thermo, cam)) {

System.out.println(d.getId() + " lastHeartbeat=" + d.getLastHeartbeatEpochSeconds());

}

// === Base-class polymorphism ===

List<Device> devices = Arrays.asList(lock, thermo, cam);

System.out.println("\n== Initial status via Device ==");

for (Device d : devices) {

System.out.println(d.getStatus());

}

// === Interface polymorphism: Networked ===

System.out.println("\n== Connect all Networked ==");

for (Device d : devices) {

if (d instanceof Networked) {

((Networked) d).connect();

}

}

// === Interface polymorphism: BatteryPowered ===

System.out.println("\n== Battery report (BatteryPowered) ==");

for (Device d : devices) {

if (d instanceof BatteryPowered) {

BatteryPowered bp = (BatteryPowered) d;

System.out.println(d.getClass().getSimpleName() + " battery = " + bp.getBatteryPercent() + "%");

}

}

// === Final status check ===

System.out.println("\n== Updated status via Device ==");

for (Device d : devices) {

System.out.println(d.getStatus());

}

}

}

**5) Brief Rationale (2–4 sentences)**

* Why is Device defined as an abstract class?
* How do the Networked and BatteryPowered interfaces add behavior to your concrete classes?
* Is this design an example of *multiple inheritance* in Java? Explain why or why not.

**Grading (30 pts)**

| **Category** | **Description** | **Points** |
| --- | --- | --- |
| **Implementation** | Correct use of inheritance and interfaces; meets all required method signatures and behaviors; uses the provided Device constructor; correctly implements Networked and BatteryPowered; uses setConnected(boolean) properly; validates inputs. | **15** |
| **Program Behavior** | Code compiles and runs with the provided Main.java unchanged; heartbeat behavior works; base-class and interface polymorphism demonstrated; exception thrown for invalid battery input; getStatus() output matches required formats. | **9** |
| **Rationale** | Clear, thoughtful, and specific answers to the four questions above. References to the student's own code are present. Shows conceptual understanding of abstraction, interface-based behavior, and multiple inheritance in Java. | **6** |

**Question 5 (10 pts)**  
**Reflection on AI Use in Learning and Problem Solving**

Discuss your personal experience using **AI tools** (such as ChatGPT, GitHub Copilot, or others) before and during this course.  
In your response, address the following points:

1. How have you used AI to support your learning or programming in this course?
2. What benefits or limitations did you encounter?
3. Looking ahead, how do you expect AI to influence the way you solve problems **academically or professionally**?

Your answer should be **1–2 well-developed paragraphs.**

When I used ChatGPT before this course, I was very skeptical. I used it for making practice problems and aiding in my understanding of topics.