**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.)

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.)*

Answer:

#### ****a) Should a well-designed class have high or low cohesion? (5 pts)****

A well-designed class should have **high cohesion**.  
High cohesion means that all the methods and data inside a class work together toward one clear purpose or responsibility. When a class focuses on a single concept, it becomes easier to maintain, reuse, and test. Low cohesion, on the other hand, makes a class harder to understand because it mixes unrelated behaviors.

#### ****b) Analyze the StudentPortalHelper class using Arthur Riel heuristics (15 pts)****

**Cohesion level:** ❌ Low cohesion

According to Arthur Riel’s heuristic “A class should represent one and only one concept,” this StudentPortalHelper class violates that rule. It mixes responsibilities from multiple domains:

* **Academic logic** – computeGPA()
* **File I/O** – exportRosterToCsv()
* **UI formatting** – formatDateForUi()
* **Email creation** – makeWelcomeEmail()
* **Payment processing** – processTuitionPayment()
* **Security** – isStrongPassword()
* **Caching/Storage** – putCache() / getCache()

These unrelated concerns mean the class acts as a “God class” or utility dump, making it difficult to maintain or extend.

**Refactoring approach:**  
To improve cohesion, each responsibility should be separated into its own focused class:

* GpaCalculator – handles GPA logic
* CsvExporter – handles roster export
* EmailService – formats or sends emails
* PaymentProcessor – manages tuition payments
* PasswordValidator – checks password strength
* CacheManager – handles in-memory caching

Each class would then follow Single Responsibility Principle (SRP) and exhibit high cohesion.  
Alternatively, a common interface such as StudentService could coordinate these smaller classes while maintaining clear separation of concerns.

### ****AI Transcript –****

User: Can you help me understand what “cohesion” means for classes in Java? My midterm question asks if a well-designed class should have high or low cohesion.

AI: Sure! Cohesion measures how closely related a class’s methods and data are. A class with high cohesion focuses on one main purpose, which makes it easier to maintain and test. Low cohesion means the class does too many unrelated things.

User: The StudentPortalHelper class in my exam has methods for GPA, payments, email, CSV export, password checking, and caching. What does that mean for cohesion?

AI: That shows low cohesion. According to Arthur Riel’s heuristic “A class should represent one and only one concept,” it violates that rule because it mixes academic logic, UI, file I/O, security, and storage all in one class.

User: How could I fix it?

AI: You can refactor it by splitting the responsibilities into smaller, focused classes — for example:

- GpaCalculator for computing GPA

- CsvExporter for file output

- EmailService for emails

- PasswordValidator for security

- CacheManager for caching

This improves clarity, testability, and follows the Single Responsibility Principle.

User: Perfect, thanks! That matches exactly what my professor’s question is asking for.

**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.)
2. 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.)

**Answer:**

**a) Does the current structure support changing trim mid-build?**  
If trims are modeled as subclasses like BaseCar extends Car, LuxuryCar extends Car, and SportCar extends Car, then the trim is **baked into the type at construction**. Changing trim later would require replacing the object or unsafe type switching, which breaks substitutability and complicates persistence. Similarly, if engine type is fixed by subclassing (ElectricCar, PetrolCar), mixing trim and engine via inheritance explodes combinations and prevents easy runtime changes.

**b) Refactor using composition (Strategy pattern).**  
Keep Car as a concrete class that **has-a** Trim and **has-a** Engine.

Car

- Trim trim; // interface or abstract type: BaseTrim, LuxuryTrim, SportTrim

- Engine engine; // interface: ElectricEngine, PetrolEngine

Car delegates behavior such as price add-ons, features, or performance to trim and engine. To change trim, call car.setTrim(new LuxuryTrim()). This composition cleanly supports runtime changes without changing the car’s type, and it avoids the inheritance-combination explosion.

### ****AI Transcript –****

User: My exam question talks about a car manufacturer that builds Base, Luxury, and Sport trims, and asks if the current structure supports changing the trim mid-build. Can you explain what that means?

AI: It sounds like the current structure probably uses inheritance — like BaseCar, LuxuryCar, and SportCar subclasses. If that’s the case, the trim is locked at object creation, so you can’t easily switch from Base to Luxury because it would require changing the object’s type.

User: Right, so that means the structure doesn’t really support dynamic changes?

AI: Exactly. Inheritance hard-codes those differences. It works for fixed relationships but not for flexible runtime changes such as switching trim levels or engine types.

User: The question also asks how to refactor it using composition. How would that work?

AI: Instead of inheritance, use composition. Let `Car` have fields for `Trim` and `Engine` that are interfaces or abstract types — like:

- `Trim trim;`

- `Engine engine;`

You can then create concrete classes such as `BaseTrim`, `LuxuryTrim`, and `SportTrim`, plus `ElectricEngine` and `PetrolEngine`.

At runtime, you can simply call `car.setTrim(new LuxuryTrim())` to change the trim without altering the car’s class.

User: So composition makes it possible to change the trim dynamically?

AI: Yes — that’s the key benefit. Composition avoids an inheritance explosion and lets the object’s behavior change through contained objects rather than type switching.

**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 + "%]";

}

Answer:

**a) Does the current structure support changing trim mid-build?**  
If trims are modeled as subclasses like BaseCar extends Car, LuxuryCar extends Car, and SportCar extends Car, then the trim is **baked into the type at construction**. Changing trim later would require replacing the object or unsafe type switching, which breaks substitutability and complicates persistence. Similarly, if engine type is fixed by subclassing (ElectricCar, PetrolCar), mixing trim and engine via inheritance explodes combinations and prevents easy runtime changes.

**b) Refactor using composition (Strategy pattern).**  
Keep Car as a concrete class that **has-a** Trim and **has-a** Engine.

Car

- Trim trim; // interface or abstract type: BaseTrim, LuxuryTrim, SportTrim

- Engine engine; // interface: ElectricEngine, PetrolEngine

Car delegates behavior such as price add-ons, features, or performance to trim and engine. To change trim, call car.setTrim(new LuxuryTrim()). This composition cleanly supports runtime changes without changing the car’s type, and it avoids the inheritance-combination explosion.

**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** |

**Answer:**

 **Why is Device abstract?** It captures shared state/behavior (id, location, heartbeat, connection flag) but leaves getStatus() to subclasses, preventing direct instantiation of an incomplete device.

 **How do interfaces add behavior?** Networked and BatteryPowered declare capabilities that multiple devices can implement. This enables interface polymorphism in Main (e.g., connect all Networked, report all BatteryPowered) without forcing unrelated classes into a rigid inheritance tree.

 **Multiple inheritance?** Not of classes—Java forbids that. But a class can implement many interfaces, so this is **multiple interface inheritance**, not multiple class inheritance.

### ****AI Transcript –****

User: My midterm question 4 asks why Device is abstract, how the interfaces add behavior, and if this design uses multiple inheritance. Can you help me explain that?

AI: Sure! The Device class is abstract because it defines common fields and behavior (like id, location, heartbeat, and connection state) but leaves some parts, like getStatus(), for subclasses to implement. That prevents incomplete objects from being instantiated directly.

User: Got it. So what do the Networked and BatteryPowered interfaces do?

AI: They define shared capabilities rather than shared structure. Networked handles connecting/disconnecting, and BatteryPowered defines methods for managing battery percentage. This allows polymorphism — for example, you can connect all networked devices or check all battery-powered ones regardless of their specific types.

User: Makes sense. The last part asks if this is multiple inheritance.

AI: It’s not multiple class inheritance since Java doesn’t allow that. But a class can implement multiple interfaces — that’s called multiple interface inheritance. It gives flexibility without the problems of multiple class inheritance (like ambiguity from diamond-shaped hierarchies).

User: Perfect, that matches what the question is asking for.

**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.**

**Answer:**

I used AI tools such as ChatGPT and GitHub Copilot to support my learning and programming throughout the course. They helped me break down requirements, generate starter code, and spot common pitfalls (for example, input validation, exception handling, and cohesive class design). When I got stuck, I asked targeted questions and compared AI suggestions to the rubric to ensure alignment with the assignment.

The benefits were speed, idea generation, and immediate feedback; the main limitations were that AI outputs can be confidently wrong or not aligned with my professor’s conventions if I don’t guide it precisely. Going forward, I expect AI to remain a force multiplier: I’ll use it for brainstorming, code reviews, test generation, and documentation—while keeping myself responsible for architectural decisions, correctness, and academic integrity. *(Per the exam policy, I am including my AI interactions as references.)*

### ****AI Transcript –****

User: My midterm reflection question asks how I used AI tools like ChatGPT or GitHub Copilot during the course, the benefits or limitations, and how I expect AI to influence my future problem-solving. Can you help me write that?

AI: Sure! You can start by explaining how you used AI for understanding coding problems, generating starter code, debugging, and clarifying programming concepts. Mention that it helped you work faster and learn more efficiently.

User: That’s true — I used it mostly to check syntax, get explanations for Java concepts, and review logic when I got stuck.

AI: Perfect. You can then describe benefits like faster learning, idea generation, and immediate feedback. For limitations, mention that AI isn’t always accurate and sometimes needs guidance to match your professor’s expectations.

User: And for the last part, about future use?

AI: You can say that you expect to continue using AI for brainstorming, documentation, code reviews, and professional development — but with academic integrity and personal accountability. That shows awareness and responsibility.

User: Nice, that matches the tone I wanted — helpful but self-aware.