**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.)

Yes, well designed classes should have high cohesion.

A class with high cohesion has a single task that it does well. The methods and data outlined in class work well together in order to serve a single purpose. With low cohesion increases the chance of a certain class having unrelated functionalities that don’t mesh well together and complicate the code.

High Cohesion is easier to maintain and test. Any changes to specific function of a program will likely only have to be done in one or two classes and it will be obvious which class should be changed. While in classes with low cohesion changing a function might affect another function of the program since they are all nested within a single class. This increases the risk of breaking the program’s functionality.

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.)*

1) StudentPortalHelper Class: This class has low cohesion. It has completely unrelated static methods that don’t work together to perform a single task or share data. It breaks three of Arthur Riel’s heutrics:

* Heuristic 2.8 A class should capture one and only one key abstraction: The class breaks this because it captures several functions. It grades and computes GPA, does notifications, checks security, caching, etc. This is too much functionality in one class
* Hueristic 3.2 Do not create god classes/objects in your system: This is similar to last point I made. The StudentPortalHelper Class deals with any functionality even remotely related to the “Student Portal.” As such, many other class will have to depend on this single class with is not ideal

2) The StudentPortalHelper class should be broken down into smaller classes that more precisely deal with 1 function/abstraction according to Heuristic 2.8. For example

* A single class for grading and gpa Calucation like “GpaCalculator”
* A single class for Email Notifications like “EmailFormatter”
* A single class for CSV reporting like “CSV reporter”
* A single class for UI that deals with issues like the Date. For example “UI formatter”
* A single class for security or passwords like “PasswordValidator”
* A single class for Caching like “PortalCache”
* Lastly a single class for payments like “TutitionProcessor”

This replaces the god class and makes the program more maintainable and readable.

**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 structure does not support changing the trim level at anytime.

In the diagram Car and the trim levels(Base, Sports and Luxury) have an Inheritance relationship. A car object would be instantiated as one of these subclasses (Car myCar = new Base();) for example.

Object types are static and immutable in Java. This means that one myCar is set as a Base car, you will not be able to change the object’s type to Luxury or Sports later on.

In order to do that you would have to delete the previous Base class object and create a new object under the Luxury class and then copy all the data from the old object over to this new one. This would be very inefficient in the long run.

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.)

Just like stated in the hint, we need to change the relationship between Car and the trim levels from a Inheritance to a Composition relationship. Instead of using Car as a base class for trims, we instead create a new class called Trim that will have methods like getTrimName() or getFeatures().

From here we make Base, Sports and Luxury classes under the Trim Base class. Then within the Cr class we add a field that takes in an object of type Trim. Introduce a Setter function in the Car class that allows us to dynamically change the Trim by creating a new Trim object. Now we can change the Trim at any time.

**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?

I have use AI mainly as a tool for checking the correctness of my work and giving me outlines on how I should tackle certain problems/programs. If I need to learn a new resource AI is also good at pointing you to certain sources that will help accelerate your learning.

1. What benefits or limitations did you encounter?

Benefits: Able to complete work much faster since either the brainstorming phase or the debugging/testing phase is reduced heavily by AI. Now I can go into a project with an exact plan to follow or if I already have plan, get to check the correctness of the steps/results I have.

Limitations: While AI is very useful it tends to produce buggy code, especially if you do not understand the specific stack or area you’re working in. If you try to use it to debug its own mistakes it will often hallucinate solutions that most of the time will not work even if you specify the exact issue that you’re facing. This is why having some knowledge is beneficial since you will be able to see the areas where ai is struggling and either fix it or guide it to the solution accordingly.

1. Looking ahead, how do you expect AI to influence the way you solve problems **academically or professionally**?

I expect it to be used as a tool to help software engineers and problem solvers complete and outline problems. At my previous internship AI trained on the company’s data was used to explain certain code in codebases for new engineers, fix the formatting of certain code making sure it matched company coding standards, help write project documents based on bugs that were being reported etc. I expect it to be used in a similar if not a more important way in the coming future.

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

**AI/ SOURCES USED:**

No outside sources were used for the completion of questions 1,3 or 5.

Gemini was used for question 1,3. For question 1 AI was used to check the correctness of my understanding of the cohesion principles in the question. It was used similarly for question 3.

**Question 1 Prompt and Answer**

**Prompt:** Based on this question {insert question} I think that the Cohesion is very low and the class is currently acting as a god class. Similarly, I think that breaking it down into different classes is the best option, what do you think about this answer

**Answer**: Yes, your analysis is perfectly correct. That's a strong, concise summary of the core problem.

Here's a breakdown of *why* your answer is so accurate, which can help you flesh out your final response:

Why Your Analysis Is Correct

* Low Cohesion: You're spot on. The class has extremely low cohesion. Its methods have almost nothing to do with each other. It's a random collection of responsibilities:
  + computeGPA (Academic business logic)
  + exportRosterToCsv (Reporting / Data export)
  + makeWelcomeEmail (Notification logic)
  + formatDateForUi (Presentation / UI logic)
  + processTuitionPayment (Financial logic)
  + isStrongPassword (Security logic)
* God Class: Your "god class" assessment is also correct. This is a classic "Utility God Class" (or "grab bag"). It violates several of Riel's heuristics:
  + Heuristic 2.8: "A class should capture one and only one key abstraction." This class captures at least six different abstractions.
  + Heuristic 3.2: "Do not create god classes/objects in your system." This class is a textbook example. It will become a central dependency for every *other* part of the application, making the entire system brittle and hard to change.

Why Your Solution Is Correct

* Breaking It Down: Your solution is the standard and correct way to fix this. This directly follows Heuristic 2.10: "Spin off non-related information into another class."

By breaking it down, you would create several new, highly-cohesive classes, each focused on *one abstraction* (Heuristic 2.8). For example:

* PasswordValidator
* PaymentService
* UiFormatter
* GpaCalculator
* RosterExporter

Your answer identifies the correct problem (low cohesion, god class) and the correct solution (break it down), and it's directly supported by Riel's heuristics. That's a strong, complete answer.

**Question 3 Prompt and Answer**

**Prompt:** For this specific question {insert question} I think the current structure doesn’t work because in order to change a the trim of an object, you would have to delete the class and replace it entirely. How would a method of having Trim as a different class instead that Car can have as field work. Here we can have a method where it can change the trim object connected to the car class.

**Answer:** You are exactly right on both points. Your analysis of the problem is perfect, and your proposed solution is the correct, flexible object-oriented approach.

Here is a more detailed explanation of *why* your thinking is correct, which you can use for your answer.

**The Problem: Static Inheritance ("is-a")**

You correctly identified that the current structure uses **Inheritance**. A specific car object is created as *one* type, like new Sports().

* In a language like Java, an object's fundamental type is **static and immutable**. It is "born" as a Sports object and it *must* remain a Sports object for its entire life.
* As you said, the only way to "change" the trim is to **destroy the original object and create a new one** (new Luxury()), which is not a real change. This is a very rigid design and fails the requirement.
* This is a classic mistake: using inheritance (an "is-a" relationship) when the part (the trim) is really something the main object *has* (a "has-a" relationship) and that part needs to change.

**The Solution: Dynamic Composition ("has-a")**

Your idea of "having Trim as a different class instead that Car can have as field" is the correct solution. This is called **Composition**.

This design pattern is often called the **Strategy Pattern**. The "Trim" is a "strategy" (a set of behaviors or properties) that the Car uses, and you want to be able to change that strategy at runtime.

… *It goes on to give java program examples of this principle*