**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, meaning all its methods and data are closely related and focused on a single, well-defined purpose (Pressman & Maxim, 2020). This focus is superior because a class that does only one thing is far easier to understand, maintain, test, and reuse. For example, a PaymentGateway class is highly cohesive if it *only* handles transactions, refunds, and verifications. In contrast, a class with low cohesion, often called a "God Class" or "junk drawer" (Riel, 1996), mixes unrelated responsibilities, which makes it confusing, difficult to debug, and impossible to reuse. Therefore, high cohesion is the practical application of the **Single Responsibility Principle (SRP)**, a core design concept stating that a class should have only one reason to change (Martin, 2018).

**Internet Works Cited**

1 Martin, R. C. (2018). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

2 Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.

3 Riel, A. J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley Professional.

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)

The class demonstrates extremely low cohesion and changes are necessary. The redesign strategy would be to break the class apart. I would refactor the class by creating several new, highly cohesive classes, each responsible for one of the abstractions currently mixed in StudentPortalHelper. My general approach would be:

1 Create a SecurityValidator class:

⁃ Move public static boolean isStrongPassword(String pwd) to this new class.

2 Create a PaymentService class:

⁃ Move public static boolean processTuitionPayment(...) to this class. This class would later hold configurations for the payment gateway.

3 Create an AcademicService class:

⁃ Move public static double computeGPA(List<Integer> grades) to this class.

4 Create a NotificationService class:

⁃ Move public static String makeWelcomeEmail(String studentName) to this class. This class could later be expanded to handle other notifications (e.g., SMS, portal alerts).

5 Create a FormatUtils (or UiHelper) class:

⁃ Move public static String formatDateForUi(LocalDate date) to this class. This isolates presentation-layer logic.

6 Create a RosterExporter class:

⁃ Move public static void exportRosterToCsv(...) to this new class. This isolates I/O operations.

7 Create a PortalCache class:

⁃ Move the cache map and the putCache/getCache methods to this class, making it a dedicated, cohesive caching utility.

After moving all methods and data to their new, cohesive classes, the StudentPortalHelper class would be empty and can be deleted. This refactoring results in a set of components that are understandable, testable, and reusable.

**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**

**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:**

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

• 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.)

The current structure, as depicted in the UML diagram above, does not support the requirement for a customer to dynamically change a car's trim level during the manufacturing process. The diagram shows that Base, Sports, and Luxury are subclasses of Car. This is an example of implementation inheritance, which creates a static and rigid "is-a" relationship. When an object is created (e.g., new Sports()), its type is permanently fixed as Sports. To change the trim level from Sports to Luxury, the original object would have to be destroyed and a completely new Luxury object created, losing any state or history associated with the original car instance. This approach is immutable and inefficient for tracking a single vehicle whose attributes change over time. I would choose to remove Car as a base class, and refactor it as a concrete class that contains a trim level as one of its attributes. This is achieved by creating a Trim interface and having concrete classes like Base, Luxury, and Sport implement it. The Car class would then hold a reference to a Trim object, this object can be swapped out at any time without changing the Car object itself. This "has-a" relationship is flexible and allows the car's behavior and features to be modified dynamically by simply assigning a new Trim object (e.g., car.setTrim(new LuxuryTrim())) to the car instance. This is a key advantage of using composition over inheritance.

AI Reference:

My Input: Check if my argument against the current structure is valid and back my claim that the current structure is no sufficient for user modification with clear refactoring examples.

AI Answer:

**Analysis of the Current Structure**

The current structure, as depicted in the UML diagram, **does not support** the requirement for a customer to dynamically change a car's trim level during the manufacturing process. The diagram shows that Base, Sports, and Luxury are subclasses of Car. This is an example of implementation inheritance, which creates a static and rigid "is-a" relationship. When an object is created (e.g., new Sports()), its type is permanently fixed as Sports. To change the trim level from Sports to Luxury, the original object would have to be destroyed and a completely new Luxury object created, losing any state or history associated with the original car instance. This approach is inflexible and inefficient for tracking a single vehicle whose attributes change over time.

**Refactoring to Support Dynamic Changes**

To allow the trim level to change dynamically, the structure should be refactored to favor composition over inheritance. Instead of Car being a base class, it should be a concrete class that *contains* a trim level as one of its attributes. This is achieved by creating a Trim interface and having concrete classes like Base, Luxury, and Sport implement it. The Car class would then hold a reference to a Trim object, which can be swapped out at any time without changing the Car object itself. This "has-a" relationship is flexible and allows the car's behavior and features to be modified dynamically by simply assigning a new Trim object (e.g., car.setTrim(new LuxuryTrim())) to the car instance, fulfilling the requirement perfectly.

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

**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.

The Device class is defined as abstract because it provides a common template of state and behavior for all smart devices but is too generic to be instantiated on its own, forcing concrete subclasses to implement the getStatus() method. Interfaces like Networked and BatteryPowered add optional "capabilities," allowing classes such as DoorLock and Camera to flexibly gain features that others, like Thermostat, may not require. This design is not an example of multiple inheritance; rather, it uses single class inheritance (extends Device) to acquire shared state and implementation. It combines this with implementing multiple interfaces (implements Networked, BatteryPowered) to adopt various behavior contracts, which is Java's safe and powerful alternative to inheriting from multiple superclasses.

**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:

• How have you used AI to support your learning or programming in this course?

• What benefits or limitations did you encounter?

• 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.**

Since my freshman year at Howard starting a computer science track, I have found AI coding tools like Cursor, Windsurf, and Lovable to be extremely helpful. I first started using coding companions when competing in various hackathons; this helped with configuring front-end user interfaces and helped me to pick up on full-stack development. I expect AI to continue influencing the way I solve problems when it comes to programming. I’ve had to get comfortable with many languages like Python, Rust, Java, TypeScript, and others. Without having AI as a debugging tool and programming companion, I don’t think I would be able to have much versatility in my programming capabilities. Using AI, I don’t have to fixate so much on perfect code logic implementation; as long as I can architect the framework for whatever software solution or program I am looking to build, I can always ideate and improve my work if I prompt AI to clearly understand my goal and implications. The biggest limitation I have faced when leveraging AI is its context competency. At times AI can lack a general sense of context and may struggle to understand the bigger picture of a project, which can hinder their ability to provide truly valuable assistance. However, with advancements in AI technology and continued training, these limitations are gradually being overcome. I foresee in the future new AI models will be developed with greater capabilities that can help in problem solving in a wide range of use cases. I expect these advancements to provide more advantages in problem-solving especially is research and development sectors.