**Howard University**

**College of Engineering and Architecture**

**Department of Electrical Engineering & Computer Science**

**Large Scale Programming**

**Fall 2024**

**Midterm Exam – Part I**

October 17, 2024

**Part I: 50 pts. (essay questions)**

**Instructions:**

* **Submit completed exam to your github repository under package org.howard.edu.lsp.midterm**

**Verify that the commit(s) completed successfully**

* **OPEN BOOK, OPEN NOTES. THERE IS NO COLLABORATION ON THIS EXAM**
* **Site any and ALL references for anything obtained off the internet.**

Question 1. (10 pts.)

You are reviewing the following class in a Fitness Tracking System. The class is intended to manage various aspects of a user’s fitness activities.



Tasks:

1. Analyze the UserFitness class. Explain why this class has low cohesion. (5 pts.)

Ans: This class has a lower cohesion because it tries to manage several distinct responsibilities that are not directly related to each other. There are three different sections to handle different responsibilities.

User profile information stores user data like userName, userAge, weight, and height, which are related to personal details, not directly to fitness tracking.

Workout tracking has fields like dailySteps, caloriesBurned, and workoutType, which are related to workout logging and activity tracking.

Nutrition tracking tracks nutrition-related data such as caloriesConsumed and meals, which are unrelated to the workout tracking or user profile.  
  
Since all are different and not directly related to each other. This class has lower cohesion

1. Describe, in general, how you would split the UserFitness class into more cohesive classes. Your answer should not be a complete design. (5 pts.)

Ans: I could split UserFitness into more cohesive classes by introducing different classes and defining methods and behavior that comes under that class strictly to that class. For example I could introduce three different classes like personal information, workout tracking and nutrition tracking and have specific data and behavior defined under these three different classes.

Question 2. (5 pts.)

Given the following, answer the below questions.

A university manages its students using a program that has a class Student with subclasses FirstYear, SecondYear, and ThirdYear for year-specific state and behaviors. The program has a List that contains all Students.

Should Student be a class, an abstract class or an interface? Explain your answer

Ans: The Student should be an abstract class. This is because all the students(irrespective of what year they are in) likely share common attributes (like name, studentID, etc.) and behaviors (like registering for courses), but the specific implementation or additional behaviors may vary depending on what year they belong to . So, an abstract class works the best in this scenario

1. At the end of each year, the third-year students graduate and must be removed. This is done by passing the list to the following method:

void removeThirdYears(List<Students> students) {

for (Student student : students) {

try {

ThirdYear = (ThirdYear) student;

students.remove(thirdyear);

}

}

}

Why is the above method flawed? Hint: Another Arthur Riel heuristic (5 pts.)

Ans: The removeThirdYears method is flawed due to Concurrent Modification Exception. This happens when you modify a list (removing elements) while iterating over it using a for-each loop. This adheres to Arthur Riel's heuristic: "Avoid modifying a collection while iterating over it."

Question 3. (5 pts.)

Given the following, answer the below questions.

public class BankAccount {

public String accountHolderName;

public String accountNumber;

public double balance;

public double interestRate;

public BankAccount(

String accountHolderName,

String accountNumber,

double initialDeposit,

double interestRate) {

this.accountHolderName = accountHolderName;

this.accountNumber = accountNumber;

this.balance = initialDeposit;

this.interestRate = interestRate;

}

// Exposed implementation for adding interest

public void addInterest() {

double interest = balance \* (interestRate / 100);

balance += interest;

}

// Exposed implementation for deposits

public void deposit(double amount) {

balance += amount;

}

// Exposed implementation for withdrawals

public void withdraw(double amount) {

balance -= amount;

}

}

1. Is BankAccount well-encapsulated? Why or why not? If so, describe what changes you would make to the class. Otherwise, state nothing. (5 pts.)

Ans: No, the BankAccount class is not well-encapsulated because its fields (accountHolderName, accountNumber, balance, interestRate) are public, which violates the principle of encapsulation. To fix this, we should make the fields private and provide getter and setter methods for controlled access

Question 4. (10 pts.)

Given the following, answer the below questions.

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 not support this. (5 pts.)

Ans: The current structure likely hard-codes the trim level (Base, Luxury, Sport) into the car's attributes, making it impossible to change it dynamically. This violates the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. You can't change the car's trim without modifying the underlying class.

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? (5 pts.)

Ans: Car class can be refactored to use composition by separating the trim level into its own class. This allows the car to switch between different trim configurations dynamically.

Question 5. (5 pts.)

Suppose you are about to code a particular class and find that you have written another class with nearly the functionality that you now are developing. This means that the two classes in large parts contain the same code, but in smaller parts have different code. Explain in detail why this may be a problem and how to solve it. Give your answer in terms of Arthur Riel’s design heuristics.

Ans: If two classes contain nearly the same functionality, it may result in duplicate code, which violates Arthur Riel’s heuristic: "Avoid duplicate code across classes." Duplicating code increases maintenance effort, as any change in the shared functionality must be updated in multiple places.

This can be solved by using inheritance or composition to refactor the shared code into a single location where in inheritance creates a common superclass that holds the shared functionality and composition extracts the common code into a helper class, and both classes can delegate to this helper.

Question 6. (5 pts.)

Suppose we need to store instances of the classes Car and BankAccount in a database. To obtain low coupling it may seem as if the functionality to save an instance should be in precisely those classes, since they have the information that shall be saved. Why is it unwise to place the database calls there?

Ans: Placing database logic inside the Car or BankAccount classes would result in high coupling because those classes would depend on database technologies. This violates the Single Responsibility Principle, as these classes would have two responsibilities: making the internal state of database and and interacting with the database. This is why it would be unwise to place the database calls there.

Question 7. (5 pts.)

Given a set of requirements, what conditions must be met to make it appropriate to create a new class hierarchy using inheritance? Neither superclass nor subclass exist before, the question is about designing a new class hierarchy.

Ans: If a new class hierarchy is to be designed, the following conditions must be met:

a. Is-a relationship: The subclass should be a specialized version of the superclass.

b. Shared behavior: The superclass should define common behavior that the subclasses can inherit and optionally override.

c. Extensibility: The system should need to add new types of behavior in the future, where new subclasses could be introduced to extend functionality without modifying the existing code.

Question 8. (2.5 pts.)

Describe the difference between inheritance and composition why would you choose one over the other.

Ans: Inheritance is when a class derives from another class, inheriting its behavior and attributes.

Composition is when a class is composed of other classes, using their functionality instead of inheriting it.

I would choose composition over inheritance when I want to reuse functionality from multiple sources without creating a rigid class hierarchy and I would use inheritance over composition when there is a clear "is-a" relationship between the classes.

Question 9. (2.5 pts.)

Explain the meaning of cohesion and describe whether low or high cohesion is better.

Ans: Cohesion refers to how closely related the responsibilities of a class are.

High cohesion is better because it means that the class is focused on a single responsibility, making it easier to maintain, understand, and reuse. Low cohesion indicates that a class is trying to do too many unrelated things, which makes the code harder to manage and it makes it seem like the whole class is a functional program and disrupts the entire point of introducing class and approaching the problem with OOP concept.