**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.)

=> The class UserFitness has low cohesion because it lacks a clear focus, with unrelated elements grouped together, diminishing their functionality and making maintenance more difficult. For example, we find various different types of data in this class like personal information (name, age, height, weight), workout information (daily step, calories burned, workout type) and nutrition information (calories consumed).

In addition, the method logWorkout is logging different aspects of the user’s fitness, but these information (workoutType, caloriesBurned) don't belong to the same domain concept i.e. workout information and nutrition information.

Moreover, a class should only have one responsibility and should have only one reason to change. However, UserFitness class has handled all aspects like personal information, workout tracking, and nutrition tracking.

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.)

=> To improve the cohesion, we could break UserFitness into smaller, more focused classes. We could create a UserInformation class for storing and managing personal information about the user, such as their name, age, weight, and height.

We could also create another class TrackWorkout, tracking workouts specifically, including workout types, steps taken, and calories burned during a workout. Within this class we could implement logWorkout and logSteps method.

For tracking nutrition we could implement another class TrackNutrition that with methods like logMeals and tracking calories intake.

Then we can also have another class that generate a fitness report based on the information from the TrackWorkout, TrackNutrition, and UserInfromation classes. It could aggregate the data from these individual classes and generate a fitness report.

**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

=> The student should be an abstract class because it allows to define common attributes and behaviors which all of the students share in the Student class which avoids code duplication in the subclasses FirstYear, SecondYear, and ThirdYear. We can add some default implementations of the methods in the Student class which are common to all of the subtypes while subclasses are allowed to implement specific behaviors which is for example getYear() in this case as year-specific state. This also makes sure that only the subclasses can be instantiated.

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.)

=> The above method is flawed because ThirdYear = (ThirdYear) student; it is casting Student to ThirdYear but it isn’t declaring a variable for that. It should be ThirdYear thirdYear = (ThirdYear) student instead. Another flaw is that the List<Students> students should not be modified directly while being iterated as stated by the Arthur Riel heuristic. The code above modifies the list while iterating through it, which is risky and can lead to unpredictable behavior.

**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.)

=> BankAccount is not well-encapsulated because the fields in BankAccount class are public (accountHolderName, accountNumber, balance, and interestRate), which allows external code to access and modify them directly, bypassing any logic or validation that might be necessary.

To make it well-encapsulated, we can make the fields private. This change can be made by adding private:

public class BankAccount {

private String accountHolderName;

private String accountNumber;

private double balance;

private double interestRate;

Additionally, a well-encapsulated class also allows us to control access to the internal state of objects and prevent unwanted or incorrect modifications. This can implement control logic to ensure that the methods are valid.

For example, a deposit cannot be negative:

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

} else {

System.out.println("Invalid deposit amount.");

}

}

Source:

<https://www.geeksforgeeks.org/encapsulation-in-java/>

<https://www.linkedin.com/pulse/pure-encapsulation-vs-c-why-matters-software-integrity-rehman-ahmad-k6ruf/>

**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



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.)

= > The existing design fails to support dynamic changes to trim levels—such as Base, Sports, and Luxury—because it models these trims as subclasses of the Car class. This setup inherently binds each car instance to a specific trim level at the moment of creation. Since object-oriented programming does not allow an object's type to change after instantiation, altering the trim level would require creating a new object of a different subclass. This limitation poses challenges in a manufacturing context where customers might decide to switch trim levels at various production stages.

Additionally, the design lacks flexibility because it uses inheritance to represent trim levels instead of treating the trim as a mutable attribute within the Car class. Rather than enabling the trim level to be updated dynamically, the current structure necessitates the creation of an entirely new Car object to change the trim. This approach is both inefficient and unnecessarily complex. Moreover, there's no method provided to modify the trim level after the car is created; while a getTrimLevel() method exists to retrieve the current trim, there's no mechanism to set or update it.

The structure also does not define the relationship between the Engine class—with its Electric and Petrol types—and the Car class properly. There's no clear association between a specific car and its engine configuration, making it uncertain how or if the engine type can be set or altered during the manufacturing process. This lack of explicit linkage further restricts the flexibility needed to dynamically configure or modify the car's features.

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.)

=> To refactor the structure to allow dynamic trim-level changes, we can use composition instead of inheritance. We can modify the Car class by introducing a TrimLevel class to represent the different trim levels (Base, Sports, Luxury) as an attribute of the car. This allows the Car object to hold a reference to a TrimLevel object, which can be dynamically changed during the manufacturing process.

For example:

We can have a trim level class:   
public class TrimLevel {

private String trimName;

}

Then, we can use the car class to use this object TrimLevel.

**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.

⇒There are multiple problems with this approach. As such this violates multiple design heuristics. Those are:

1. Keep related data and behavior in one place.
2. Eliminate Code Duplication
3. Classes must contain only one key abstraction
4. If two classes have common data and behavior, they should inherit from a common base class

To solve this problem, we can create a base class which has all the common attributes and functionalities found in both classes. And inherit both classes from this common class and implement the unique features in their respective classes.

But if the base class and the inherited classes don't have a simple 'is-a' relationship, we should use composition.

**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?

=> By principle, each class should have one key abstraction or responsibility. If we embed database calls there, it violates this principle, and if we need to make changes, we need to do so in two places instead of one, which means we have high coupling.

**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.

⇒ It is appropriate to create a new class hierarchy using inheritance when there are few conditions met:

1. Is-a kind of relationship present. Like a car is a vehicle, a cat is an animal. So in these cases, it is appropriate for a car class to inherit from a vehicle class and a cat from an animal.
2. It is also appropriate when different subclasses need to use a common interface.
3. Code reusability is also an important factor in using inheritance to reduce redundancy.

**Question 8. (2.5 pts.)**

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

⇒Inheritance means that we create a subclass that inherits methods and properties from a parent class like a cat class can inherit from an animal parent class. Composition means that we are building a class by combining other classes as objects.

We would prefer inheritance when a class shares common things or has a hierarchical relationship like a cat is an animal so a cat class can inherit an animal class. However, we would prefer composition for building a complex class that has many simpler parts that it can be composed of. For eg: a car class can have different simpler parts like a tire class, engine class which are not hierarchical so a composition is useful in this case.

**Question 9. (2.5 pts.)**

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

⇒ Cohesion means how much the elements within a module are related or focused to a single task. High cohesion means that there is a very specific task that a particular module is solving while low cohesion refers to a state where there are some modules that do multiple tasks.   
High cohesion is better because it is easier to maintain, debug and to make the codebase structured. It also reduces duplicating the code to do some tasks making our program more scalable.