Week 2 Tutorial Solutions

Table of contents

1 Multiple Choice Questions:

1. B) The degree to which a system is composed of discrete components.
2. B) Minimal impact on other components when a component is changed.
3. C) Pure Fabrication
4. B) Describing how users interact with the system to achieve specific goals.

2 Short Answer Questions:

1. Aggregation is a special type of association in object-oriented design where one class owns an instance of another class. For example, in a university management system, a `Department` class can have an aggregation relationship with a `Student` class, where the `Department` class has a collection of `Student` objects.

2. Functional requirements specify the functional behaviors and capabilities that a software system should provide, while non-functional requirements define the system's quality attributes, such as performance, security, usability, and reliability. Functional requirements describe what the system should do, while non-functional requirements describe how well the system should perform those functions.

3. Polymorphism in Java can be demonstrated with method overriding. Here's an example:

```
class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}
```

In this example, the `Dog` and `Cat` classes override the `makeSound()` method from the base `Animal` class, exhibiting polymorphic behavior.

4. Use cases are an essential tool in object-oriented design for understanding and documenting the interactions between users and the system. They describe the sequence of steps and actions that a user performs to achieve a specific goal or task within the system. Use cases help in identifying system requirements, defining the system's scope, and facilitating communication between stakeholders.

5. The Law of Demeter, also known as the Principle of Least Knowledge, suggests that an object should have limited knowledge about other objects in the system. It should only interact with its immediate dependencies, not with objects further down the object chain. Here's an example in Java:

```
class Student {
    private String name;
    private Course course;

    public String getName() {
        return name;
    }

    public Course getCourse() {
        return course;
    }

    // Other methods and properties
}

class Course {
    private String title;
    private Department department;

    public String getTitle() {
        return title;
    }

    public Department getDepartment() {
        return department;
    }

    // Other methods and properties
}

class Department {
    private String name;

    public String getName() {
        return name;
    }

    // Other methods and properties
}

class University {
```

```
    public void printStudentDetails(Student student) {
        String studentName = student.getName();
        String courseTitle = student.getCourse().getTitle();
        String departmentName = student.getCourse().getDepartment().getName();
        System.out.println("Student: " + studentName + ", Course: " + courseTitle + ", Dep
    }
}
```

In the `University` class, the `printStudentDetails` method follows the Law of Demeter by only accessing the immediate dependencies of the `Student` object (`getName()` and `getCourse()`), without navigating further into the `Course` and `Department` objects.

3 Code-Based Questions:

1. Refactored code to improve adherence to the KISS principle:

```
public class StudentGradeCalculator {
    public double calculateFinalGrade(int[] assignmentScores, int midtermScore, int finalE
        double assignmentAverage = calculateAssignmentAverage(assignmentScores);
        double weightedMidterm = calculateWeightedScore(midtermScore, 0.3);
        double weightedFinalExam = calculateWeightedScore(finalExamScore, 0.3);
        double weightedAssignments = calculateWeightedScore(assignmentAverage, 0.4);

        double finalGrade = weightedMidterm + weightedFinalExam + weightedAssignments;

        return finalGrade;
    }

    private double calculateAssignmentAverage(int[] assignmentScores) {
        double sum = 0;
        for (int score : assignmentScores) {
            sum += score;
        }
        return sum / assignmentScores.length;
    }

    private double calculateWeightedScore(double score, double weight) {
        return (score / 100.0) * weight * 100;
    }
}
```

In this refactored version, the code is split into smaller, more focused methods, making it easier to read and maintain. The `calculateAssignmentAverage` and `calculateWeightedScore` methods encapsulate specific calculations, reducing code duplication and improving readability.

2. Reduced coupling and refactored code:

```
// EmailService.java
public class EmailService {
    public void sendEmail(String message, String recipient) {
        // Logic to send an email
        System.out.println("Sending email to " + recipient + ": " + message);
    }
}

// OrderManager.java
public class OrderManager {
    private EmailService emailService;

    public OrderManager(EmailService emailService) {
        this.emailService = emailService;
    }

    public void processOrder(String orderDetails, String customerEmail) {
        // Logic to process the order
        System.out.println("Order processed: " + orderDetails);
        emailService.sendEmail("Your order has been processed.", customerEmail);
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        EmailService emailService = new EmailService();
        OrderManager orderManager = new OrderManager(emailService);
        orderManager.processOrder("Book: Java Programming", "customer@example.com");
    }
}
```

In this refactored version, the `EmailNotification` class has been renamed to `EmailService` to better represent its responsibility. The `OrderManager` class now receives an instance of `EmailService` through its constructor, reducing the coupling between the two classes. The

`Main` class creates instances of both classes and injects the `EmailService` instance into the `OrderManager`.

This approach follows the Dependency Inversion Principle and improves testability by allowing mock implementations of `EmailService` to be injected during testing.

4 Conceptual Question:

Adding a new `Triangle` class without modifying existing code aligns with the Open/Closed Principle (OCP) because the `Shape` interface is open for extension (new classes like `Triangle` can be added), but closed for modification (existing classes like `Circle` and `Square` don't need to be changed).

The OCP states that software entities (classes, modules, functions) should be open for extension but closed for modification. This means that you should be able to add new functionality without changing the existing code.

By implementing the `Shape` interface, the `Triangle` class can be added to the system without modifying the existing `Circle` and `Square` classes or the `Shape` interface itself. The `draw()` method in the `Triangle` class can be implemented to draw a triangle shape, while the existing classes continue to work as before.

This principle promotes code reusability, maintainability, and extensibility by minimizing the risk of introducing bugs when adding new features or requirements to the system.