OOP Assignment

Sct 121-0938/2022

Dominic Nyoike

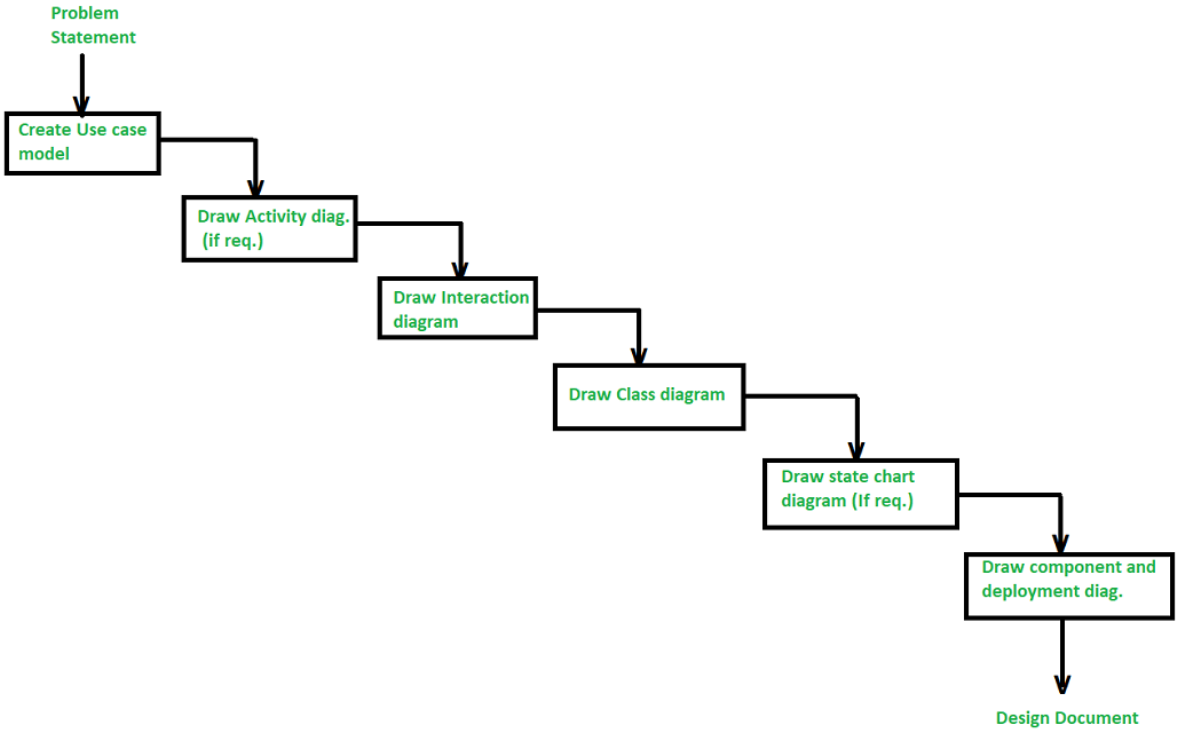
Instructions for part A: answer all the Questions in this section.

1. **(a)Using a well labeled diagram, explain the steps of creating a system using OOP principles.**

1.Use Case Modeling:

- **Identify Actors**: Recognize the actors (users, external systems, etc.) interacting with the system.
- **Create Use Cases**: Define use cases that represent specific functionalities or interactions.
- **Draw Use Case Diagram**: Visualize the relationships between actors and use cases.

2. **Activity Diagram (if needed)**:
   - An **activity diagram** captures the dynamic behavior of the system by illustrating the flow of control. It shows how activities (processes) interact within the system.
   - For instance, an activity diagram might depict the steps involved in processing a customer order.
3. **Interaction Diagrams**:
   - **Sequence Diagrams**:
     - Identify objects related to each use case.
     - Create sequence diagrams for each use case, showing the order of interactions between objects.
   - **Collaboration Diagrams**:
     - Represent interactions among objects, including messages exchanged.
     - Useful for understanding the system's dynamic behavior.
4. **Class Diagram**:
   - The **class diagram** reveals the relationships between classes (blueprints for objects).
   - Types of Relationships:
     - **Association**: Semantic connection between classes (bi-directional or unidirectional).
     - **Dependencies**: Unidirectional connections indicating one class depends on another's definition.
     - **Aggregations**: Stronger form of association, representing whole-part relationships.
     - **Generalizations**: Display inheritance relationships



I. What is the Object Modeling Techniques (OMT). [1 Marks]

It is a real-world-based modeling approach used for software modeling and designing

ii.Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP). [2 Marks]

Object-Oriented Analysis and Design (OOAD):

- **Methodology**: OOAD is a software engineering methodology that uses object-oriented concepts to design and implement software systems.
- **Techniques and Practices**:
  - **Object-Oriented Programming (OOP)**: Models real-world objects as software objects with properties and methods.
  - **Design Patterns**: Reusable solutions for common software design problems.
  - **UML Diagrams**: Standardized notation for representing different aspects of a software system.
  - **Use Cases**: Describe user interactions with the system

Object Analysis and Design (OOP):

- **Focus**: Investigates domain objects and their behavior.
- **Conceptual Solution**: Emphasizes fulfilling requirements rather than implementation.
- **Refinement**:
  - **Analysis**: Refines feature lists and creates a model of customer requirements.
  - **Design:** Transforms the model into classes for code implementation.

iv.Discuss Main goals of UML.

(a)Visualization and Communication:
UML provides a common set of diagrams and symbols to visualize system components, their relationships
(b) **Specification and Documentation**:

UML allows precise specification of system requirements, design, and behavior.

v.DESCRIBE three advantages of using object oriented to develop an information system. [3Marks]

Modularity for Easier Troubleshooting:

- In OOP, you create **self-contained objects** that encapsulate both data and functionality. For instance, consider a **Car class**. If something goes wrong with a specific car object, you know exactly where to look.

Reuse of Code Through Inheritance:

- Imagine you need to create a **RaceCar object** and a **Limousine object** in addition to your existing **Car object**. Instead of building them from scratch, you recognize commonalities

Improved Reusability and Maintainability:

- OOP promotes **code reusability**. You build components (classes) once and reuse them across different parts of your system

vi.Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept. [12 Marks]

1. **Constructor**:
    - A **constructor** is a special method within a class that gets called when an object of that class is created. It initializes the object's state (i.e., sets initial values for its attributes). Constructors have the same name as the class and do not have a return type.

```
class student {
    private String name;
    private Integer age;

    // Constructor
    public Car(String name, Integer age) {
```

```
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("name: " + name+ ", name: " + name);
    }
}

public class student {
    public static void main(String[] args) {
        name myname = new student("Lucifer", "Morningstar");
        mystudent.displayInfo();
    }
}
```

2. **Object**:
    - An **object** is an instance of a class. It represents a real-world entity or concept. Objects have attributes (data) and methods (functions) associated with them.
    - Example Java code creating an object:

```
public class ObjectExample {
    public static void main(String[] args) {
        Car myCar = new Car("Gwagon", "Civic");
        myCar.displayInfo();
    }
}
```

3. **Destructor** (Note: Java does not have explicit destructors):
    - A **destructor** is a method that gets called when an object is destroyed or goes out of scope. In Java, the garbage collector automatically handles memory deallocation, so explicit destructors are not needed.

4. **Polymorphism**:
    - **Polymorphism** allows objects of different classes to be treated as objects of a common superclass. It enables method overriding and dynamic method dispatch.
    - Example Java code demonstrating polymorphism:

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

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

public class PolymorphismExample {
   public static void main(String[] args) {
      Animal myAnimal = new cat(); // Polymorphism
      myAnimal.makeSound(); // Calls cat's overridden method
   }
}
```

5. **Class**:
    - A **class** is a blueprint or template for creating objects. It defines attributes (fields) and methods that the objects of that class will have.
    - Example Java code defining a class:

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

```
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

6. **Inheritance**:
    - **Inheritance** allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (superclass or base class). It promotes code reuse.
    - Example Java code demonstrating inheritance:

```
class Lecture {
    protected String lecId;

    public lecture(String lecId) {
        this.lecId = lecId;
    }

    public void displaylecInfo() {
        System.out.println("lecture ID: " + lecId);
    }
}

class Lecture extends lecture {
```

```
    private String department;

    public lecture(String lecId, String department) {
        super(lecId); // Call superclass constructor
        this.department = department;
    }

    public void displaylectureInfo() {
        System.out.println("Department: " + department);
    }
}
```

vi. ***EXPLAIN* the three types of associations (relationships) between objects in object oriented.**

- Association:

  Association represents a semantically weak relationship between otherwise unrelated objects. It is a “using” relationship where two or more objects interact without any ownership or parent-child connection.

- Aggregation:

  Aggregation is a specialized form of association where each object has its own lifecycle, but there exists an ownership relationship. It represents a whole/part or parent/child relationship.

  Example: An employee may belong to one or more departments in an organization. If a department is deleted, the employee object continues to exist independently.

- Composition:

  Composition is a stronger form of aggregation. It denotes a strict ownership relationship where the whole (parent) manages the lifecycle of its parts (children).

  Example: A car consists of engine, wheels, and other components. If the car is destroyed, its parts are also destroyed.

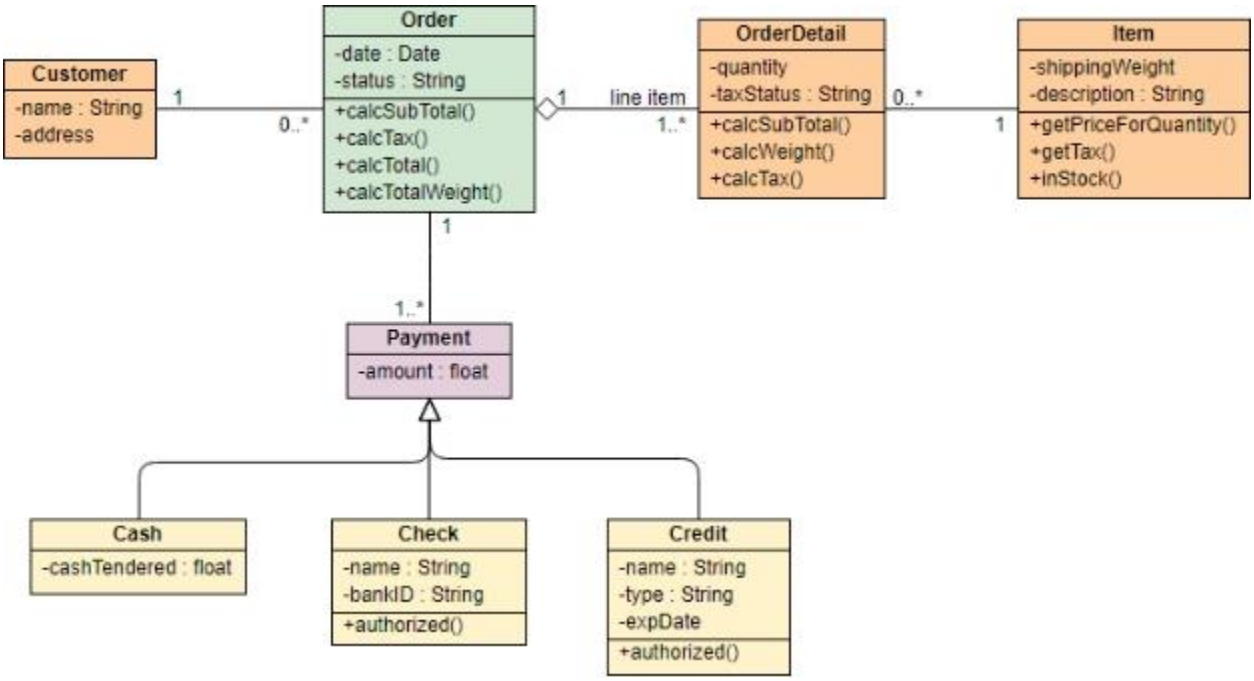
Key Property: The whole (parent) and its parts (children) have a lifecycle dependency.

1. **What is a Class Diagram?**
    - A **class diagram** is a type of **static structure diagram** in UML. It describes the structure of a system by showing:
        - **Classes**: Representing the building blocks of the system.
        - **Attributes**: Properties or data associated with each class.
        - **Operations (Methods)**: Functions or behaviors that the classes can perform.
2. **Purpose and Usage**:
    - **Visualizing Structure**: Class diagrams provide a visual representation of the static structure of a system. They help developers understand the relationships and dependencies among classes.
3. **Steps to Draw a Class Diagram**:
    - **Identify Classes**:
        - Start by identifying the key classes in your system. These represent the major components or entities.
    - **Add Attributes and Methods**:
        - For each class, list down its attributes (data members) and methods (functions).
    - **Define Relationships**:
        - Determine the relationships between classes:
            - **Association**: Represents a weak relationship between unrelated objects.
            - **Aggregation**: Depicts a whole/part relationship with ownership.
            - **Composition**: Stronger form of aggregation with strict ownership.
    - **Draw the Diagram**:
        - Use UML notation to create the class diagram:
            - Class name in the first partition.
            - Attributes in the second partition (with types).
            - Methods in the third partition (with return types).

- Connect classes using lines to represent relationships (inheritance, association, etc.).

4. **Example**: Let's consider a simple library management system:
    - **Classes**:
        - Book: Represents a book with attributes like title, author, and ISBN.
        - Library: Represents a library with attributes like name and location.
    - **Relationships**:
        - Library has an **aggregation** relationship with Book (a library contains multiple books).
        - Book has an **association** relationship with Library (books are associated with a library).

Here's a simplified class diagram for our example:



Q10: Given that you are creating area and perimeter calculator using C++, to computer area and perimeter of various shaped like Circles, Rectangle, Triangle and Square, use well written code to explain and implement the calculator using the following OOP concepts. a. Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance) [10 Marks] b. Friend functions [5

Marks] c. Method overloading and method overriding [10 Marks] d. Late binding and early binding [8 Marks] e. Abstract class and pure functions

```cpp
#include <iostream>
#include <cmath>

#ifndef SHAPE_H
#define SHAPE_H

#define PI 3.14159265358979323846

class Shape {
public:
    virtual double getArea() const = 0;
    virtual double getPerimeter() const = 0;
};
class Circle : public Shape {
private:
    double radius;

public:
    explicit Circle(double r) : radius(r) {}

    double getArea() const override {
        return PI * std::pow(radius, 2);
```

```
    }

    double getPerimeter() const override {
        return 2 * PI * radius;
    }
};

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w) : length(l), width(w) {}

    double getArea() const override {
        return length * width;
    }

    double getPerimeter() const override {
        return 2 * (length + width);
    }
};

class Triangle : public Shape {
```

```
private:
    double base;
    double height;

public:
    Triangle(double b, double h) : base(b), height(h) {}

    double getArea() const override {
        return 0.5 * base * height;
    }

    double getPerimeter() const override {
        return 3 * base;
    }
};

class Square : public Rectangle {
public:
    explicit Square(double side) : Rectangle(side, side) {}
};

#endif

int main() {
    double radius, length, width, base, height;
```

```
    std::cout << "Enter the radius of the circle: ";
    std::cin >> radius;
    Circle circle(radius);

    std::cout << "Enter the length and width of the rectangle: ";
    std::cin >> length >> width;
    Rectangle rectangle(length, width);

    std::cout << "Enter the base and height of the triangle: ";
    std::cin >> base >> height;
    Triangle triangle(base, height);

    std::cout << "Enter the side length of the square: ";
    std::cin >> length;
    Square square(length);

    std::cout << "\nCircle: Area = " << circle.getArea() << ", Perimeter = " <<
circle.getPerimeter() << std::endl;
    std::cout << "Rectangle: Area = " << rectangle.getArea() << ", Perimeter = " <<
rectangle.getPerimeter() << std::endl;
    std::cout << "Triangle: Area = " << triangle.getArea() << ", Perimeter = " <<
triangle.getPerimeter() << std::endl;
    std::cout << "Square: Area = " << square.getArea() << ", Perimeter = " <<
square.getPerimeter() << std::endl;

    return 0;
```

```
}
```

Q11:

```
import java.lang.Math;

public class CalculateG {

    public double multi(double a, double b) {
        return a * b;
    }

    public double powerToSquare(double value) {
        return Math.pow(value, 2);
    }

    public double summation(double a, double b) {
        return a + b;
    }

    public void outline(double result) {
        System.out.println("The result is: " + result);
    }
```

```
    public void main() {
        double gravity = -9.81;
        double fallingTime = 30.0;
        double initialVelocity = 0.0;
        double finalVelocity;
        double initialPosition = 0.0;
        double finalPosition;


        finalPosition = 0.5 * gravity * powerToSquare(fallingTime) + multi(initialVelocity,
fallingTime) + initialPosition;
        finalVelocity = multi(gravity, fallingTime) + initialVelocity;


        outline(finalPosition);


        outline(finalVelocity);
    }


    public static void main(String[] args) {
        CalculateG calculator = new CalculateG();
        calculator.main();
```

```
  }
}
```

PART B:

1.

```
#include <iostream>
using namespace std;

int evenFibSum(int limit) {
    if (limit < 2) return 0;
    long long int ef1 = 0, ef2 = 2;
    long long int sum = ef1 + ef2;
    while (ef2 <= limit) {
        long long int ef3 = 4 * ef2 + ef1;
        if (ef3 > limit) break;
        ef1 = ef2;
        ef2 = ef3;
        sum += ef2;
    }
    return sum;
}

int main() {
    int limit = 4000000;
    cout << evenFibSum(limit);
```

```
    return 0;
}
```

Q2.
```
#include <iostream>
using namespace std;

int main() {
    int n, num, digit, rev = 0;

    cout << "Enter a positive number: ";
    cin >> num;

    n = num;
    do {
        digit = num % 10;
        rev = (rev * 10) + digit;
        num = num / 10;
    } while (num != 0);

    cout << "The reverse of the number is: " << rev << endl;

    if (n == rev)
        cout << "The number is a palindrome.";
    else
        cout << "The number is not a palindrome.";
```

```
    return 0;
}
```

Q3. a)

```
#include <iostream>
using namespace std;

int main() {
    const int SIZE = 15;
    int myArray[SIZE];

    cout << "Enter " << SIZE << " integers:\n";
    for (int i = 0; i < SIZE; ++i) {
        cout << "Enter value " << (i + 1) << ": ";
        cin >> myArray[i];
    }

    cout << "\nThe values stored in the array are:\n";
    for (int i = 0; i < SIZE; ++i) {
        cout << myArray[i] << " ";
    }

    return 0;
}
```