GODWIN KAMAU

SCT121-0942/2022

DIT: OOP Assignment

Part A:

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

I.

II. Object Modeling Techniques (OMT) is a set of visual modeling techniques and tools for capturing and expressing the analysis, design, and implementation of software systems.

III. OOAD is a software engineering methodology that involves using object oriented concepts to design and implement software systems while OOP is a programming paradigm that revolves around the concept of objects. It involves the modelling of real world objects as software objects with properties and methods that represent the behaviour of those objects.

IV. UML's main goals are to provide a standardized, visual, and flexible modeling language that facilitates communication, visualization, and specification of software systems throughout the development life cycle**.**

V. Modularity and Reusability:

Modularity: OOP allows developers to encapsulate data and behavior within classes. Each class represents a module or component, and the internal details are hidden from the outside. This modularity enhances code organization and maintenance.

Reusability: OOP promotes code reuse through the concept of inheritance. Developers can create new classes by inheriting attributes and behaviors from existing classes. This not only reduces redundancy but also allows the incorporation of proven and tested code into new parts of the system.

Ease of Maintenance and Flexibility:

Ease of Maintenance: OOP's encapsulation and modularity make it easier to maintain and update the software. Changes to one class do not necessarily affect

other parts of the system, provided that the external interface remains unchanged. This isolation simplifies debugging and updates.

Flexibility: OOP supports polymorphism, allowing objects to be treated uniformly based on their common interface. This flexibility enables developers to extend or modify the system without affecting existing code. New classes can be added, and existing ones can be modified or replaced without disrupting the overall structure.

Improved Analysis, Design, and Modeling:

Improved Analysis: OOP facilitates a natural mapping of real-world entities into software objects. This makes the analysis and understanding of the problem domain more intuitive. Object-oriented analysis helps to identify and model entities, their attributes, and their interactions in a way that closely mirrors the actual system.

Improved Design: The use of classes, objects, and relationships in OOP supports a more structured and clear design process. Class diagrams, object diagrams, and other UML diagrams used in OOP provide visual representations that aid in the communication and documentation of the system design.

Improved Modeling: OOP aligns well with the modeling of complex systems. The ability to represent entities, relationships, and behaviors through objects and classes helps create a more accurate and understandable model of the information system. This, in turn, leads to a better-managed development process.

Vi

a. **Constructor:**A constructor is a special method in a class that is automatically called when an object of that class is created. It is used to initialize the object's state and set up any initial values or configurations.

Sample code

```
public class Car {
   private String model;
   private int year;

   // Constructor
   public Car(String model, int year) {
      this.model = model;
```

```
        this.year = year;
    }

    // Other methods and properties...
}

// Creating an object and calling the constructor
Car myCar = new Car("Toyota Camry", 2022);
```

b. Object: An object is an instance of a class. It is a runtime entity with state and behavior defined by its class. Objects represent real-world entities and are created based on the class blueprint.

Sample code

```
public class Person {
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Other methods and properties...
}

// Creating an object and using it
```

```
Person person1 = new Person("John Doe", 25);

System.out.println("Name: " + person1.getName() + ", Age: " + person1.getAge());
```

C. **Destructor:** In Java, there is no explicit destructor like in some other languages. Java uses automatic garbage collection to reclaim memory occupied by objects that are no longer in use. Developers do not need to explicitly release resources or memory.

D. **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables a single interface to represent different types or forms .

Sample code

```
public interface Shape {
   void draw();
}

public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Circle");
   }
}

public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Square");
   }
```

```
}

// Using polymorphism
Shape shape1 = new Circle();
Shape shape2 = new Square();

shape1.draw();  // Output: Drawing a Circle
shape2.draw();  // Output: Drawing a Square
```

E. **Class:**A class is a blueprint for creating objects. It defines a data structure and methods that operate on that data. Objects are instances of classes.

Sample code

```
public class Dog {
   private String name;
   private int age;

   // Constructor
   public Dog(String name, int age) {
      this.name = name;
      this.age = age;
   }

   // Other methods and properties...
}
```

F. **Inheritance:**Inheritance is a mechanism in which a new class inherits properties and behaviors from an existing class . It promotes code reuse and supports the creation of a hierarchy of classes.

Sample code

```
public class Animal {
   protected String species;

   // Constructor
   public Animal(String species) {
      this.species = species;
   }

   public void makeSound() {
      System.out.println("Animal sound");
   }
}

public class Cat extends Animal {
   // Constructor
   public Cat() {
      super("Cat");
   }

   // Override method
   @Override
   public void makeSound() {
      System.out.println("Meow");
```

```
    }
}
```

```
// Using inheritance
Animal myCat = new Cat();
myCat.makeSound();  // Output: Meow
```

Vii. Association: Association is a general relationship between two or more classes or objects. It describes how objects from different classes are connected and interact with each other.

Aggregation: Aggregation is a specialized form of association where one class represents a whole and another class represents a part. It implies a relationship where the part can exist independently of the whole.

Composition: Composition is a stronger form of aggregation where one class represents a whole and another class represents a part, but the part cannot exist independently of the whole. If the whole is deleted, the part is also deleted.

Viii. A class diagram is a type of UML (Unified Modeling Language) diagram that represents the static structure of a system by showing classes, their attributes, methods, and the relationships between them. It provides a visual representation of the essential elements in a system and how they interact with each other.

Class diagrams are commonly used during the design and analysis phases of software development.They provide a blueprint for the system's structure, helping developers and stakeholders understand the organization of classes and their relationships.

The steps

Identify Classes:

Identify the main classes in your system. These are the key entities or objects that are relevant to your system.

Identify Attributes:

For each class, identify its attributes (properties or characteristics). These represent the data that each object of the class will contain.

Identify Methods:

Determine the methods (behaviors or functions) associated with each class. These represent the operations that can be performed on objects of the class.

Define Relationships:

Identify the relationships between classes. Common relationships include association, aggregation, and composition. Use lines with arrows to indicate the direction of the relationship.

Cardinality and Multiplicity:

Specify the cardinality and multiplicity of associations. Cardinality defines the number of instances that participate in a relationship, while multiplicity defines the number of objects an object can be associated with.

Draw the Diagram:

Use a diagramming tool or paper and pencil to draw the class diagram. Place the classes, attributes, and methods in the appropriate locations. Connect the classes with lines to represent relationships.

IX.

CODE

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

// Abstract class Shape
class Shape {
public:
    // Pure virtual functions for area and perimeter
    virtual double calculateArea() const = 0;
    virtual double calculatePerimeter() const = 0;
};

// Derived class Circle (Single Inheritance)
class Circle : public Shape {
private:
    double radius;

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

    // Method overriding for area and perimeter calculation
    double calculateArea() const override {
        return 3.14 * radius * radius;
    }

    double calculatePerimeter() const override {
        return 2 * 3.14 * radius;
```

```
    }
};

// Derived class Rectangle (Single Inheritance)
class Rectangle : public Shape {
private:
    double length;
    double width;

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

    // Method overriding for area and perimeter calculation
    double calculateArea() const override {
        return length * width;
    }

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

// Derived class Triangle (Single Inheritance)
class Triangle : public Shape {
private:
    double side1;
```

```
    double side2;
    double side3;

public:
    Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

    // Method overriding for area and perimeter calculation
    double calculateArea() const override {
        // Using Heron's formula for area of a triangle
        double s = (side1 + side2 + side3) / 2;
        return sqrt(s * (s - side1) * (s - side2) * (s - side3));
    }

    double calculatePerimeter() const override {
        return side1 + side2 + side3;
    }
};

// Derived class Square (Hierarchical Inheritance)
class Square : public Rectangle {
public:
    Square(double side) : Rectangle(side, side) {}
};

// Friend function for printing information about a shape
void printShapeInfo(const Shape& shape) {
```

```
    std::cout << "Area: " << shape.calculateArea() << ", Perimeter: " <<
shape.calculatePerimeter() << std::endl;
}

int main() {
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);
    Triangle triangle(3.0, 4.0, 5.0);
    Square square(7.0);

    // Using friend function to print information about each shape
    printShapeInfo(circle);
    printShapeInfo(rectangle);
    printShapeInfo(triangle);
    printShapeInfo(square);

    return 0;
}
```

Explanation of OOP concepts used:

a. Inheritance (Single Inheritance, Multiple Inheritance, Hierarchical Inheritance):

Circle, Rectangle, Triangle inherit from the abstract class Shape, demonstrating single inheritance.

Square inherits from Rectangle, illustrating hierarchical inheritance.

b. Friend Functions:

The printShapeInfo function is declared as a friend function to access the private members of the Shape class.

c. Method Overloading and Method Overriding:

Method overriding is used in the calculateArea and calculatePerimeter methods of derived classes to provide specific implementations.

Method overloading is not explicitly used in this example, but it is a technique where multiple functions have the same name but differ in parameter types or the number of parameters.

d. Late Binding and Early Binding:

Late binding (runtime polymorphism) is achieved through the use of virtual functions in the Shape class. The actual function to be called is determined at runtime.

Early binding (compile-time polymorphism) refers to the static binding that occurs when a function call is resolved at compile time. It is not explicitly demonstrated in this example.

e. Abstract Class and Pure Functions:

The Shape class is an abstract class with pure virtual functions (calculateArea and calculatePerimeter). Objects of abstract classes cannot be instantiated, and these pure virtual functions must be implemented by the derived classes.

X. Function Overloading and Operator Overloading:

Function Overloading:

In function overloading, multiple functions in the same scope have the same name but different parameters.

The compiler distinguishes between the functions based on the number or types of parameters.

Example:

```
#include <iostream>

// Function Overloading
int add(int a, int b) {
   return a + b;
}

double add(double a, double b) {
   return a + b;
}

int main() {
   std::cout << add(3, 5) << std::endl;       // Calls the int version
   std::cout << add(3.5, 2.7) << std::endl;   // Calls the double version
   return 0;
}
```

Operator Overloading:

Operator overloading allows you to define how an operator behaves when applied to user-defined types.

It involves defining special member functions like operator+, operator-, etc.

Example:

```
#include <iostream>

// Operator Overloading
class Complex {
private:
   double real;
   double imag;

public:
   Complex(double r, double i) : real(r), imag(i) {}

   Complex operator+(const Complex& other) const {
      return Complex(real + other.real, imag + other.imag);
   }

   void display() const {
      std::cout << real << " + " << imag << "i" << std::endl;
   }
};

int main() {
   Complex a(3.0, 2.0);
   Complex b(1.5, 4.5);
```

```
    Complex result = a + b;  // Calls the overloaded operator+
    result.display();
    return 0;
}
```

Pass by Value and Pass by Reference:

Pass by Value:

In pass by value, the actual value of the variable is passed to the function.

Changes made to the parameter inside the function do not affect the original variable.

Example:

```
#include <iostream>

void increment(int x) {
    x++;
}

int main() {
    int num = 5;
    increment(num);  // Pass by value
    std::cout << num << std::endl;  // Output: 5 (unchanged)
    return 0;
```

```
}
```

Pass by Reference:

In pass by reference, the memory address (reference) of the variable is passed to the function.

Changes made to the parameter inside the function affect the original variable.

Example:

```
#include <iostream>

void incrementByReference(int& x) {
   x++;
}

int main() {
   int num = 5;
   incrementByReference(num);  // Pass by reference
   std::cout << num << std::endl;  // Output: 6 (changed)
   return 0;
}
```

Parameters and Arguments:

Parameters:

Parameters are the variables declared in the function definition.

They act as placeholders for the values that will be passed when the function is called.

Example:

```
#include <iostream>

void addAndPrint(int a, int b) {
   std::cout << "Sum: " << a + b << std::endl;
}

int main() {
   int x = 3;
   int y = 5;
   addAndPrint(x, y);  // x and y are parameters
   return 0;
}
```

Arguments:

Arguments are the actual values passed to a function when it is called.

They correspond to the parameters in the function definition.

Example:

```
#include <iostream>


void addAndPrint(int a, int b) {
   std::cout << "Sum: " << a + b << std::endl;
}#include <iostream>


int main() {
   const int limit = 4000000;


   int fib1 = 1;
   int fib2 = 2;
   int evenSum = 0;


   while (fib2 <= limit) {
      if (fib2 % 2 == 0) {
         evenSum += fib2;
      }


      // Generate the next Fibonacci term
      int nextTerm = fib1 + fib2;
      fib1 = fib2;
      fib2 = nextTerm;
   }
```

```
    std::cout << "The sum of even-valued terms in the Fibonacci sequence up to " <<
limit << " is: " << evenSum << std::endl;


    return 0;

}




int main() {

    int x = 3;

    int y = 5;

    addAndPrint(x, y);  // x and y are arguments

    return 0;

}
```

PART B