NAME: BRAVIL MAHWA

REG NO: SCT121-0946/2022

Part A:

Question one

i. Creating a System Using OOP Principles:

```
Identify Objects        Define Classes

      |                       |
      v                       v

Establish             Define Methods
Relationships

      |                     v
      v
 Implement
Model Behavior        Functionality

      |                     |
      v                     v

Encapsulate           Create Instances
Data and Code

      |                     v
      v
```

Interact

Refine the Model Between

Objects

ii. Object Modeling Techniques (OMT):

Object Modeling Techniques (OMT) is a method for visualizing and documenting an information system. It includes graphical notations for modeling objects, classes, relationships, and processes. It was developed by James Rumbaugh and is part of the broader Unified Modeling Language (UML).

iii. OOAD vs OOP:

- Object-Oriented Analysis and Design (OOAD) is a broader process that involves analyzing and designing a system using object-oriented principles. It encompasses both analysis (understanding the problem domain) and design (creating a solution). OOP, on the other hand, specifically refers to the programming paradigm using objects.

iv. Main Goals of UML:

Unified Modeling Language (UML) aims to provide a standardized way to visualize the design of a system. The main goals include:

1. Visualization: Create a visual model of the system.

2. Specification: Define and specify the architecture and components.

3. Construction: Facilitate the construction and documentation of the software system.

v. Advantages of Using OOP:

Three advantages of using object-oriented programming for information systems development include:

1. Modularity: Encapsulation allows for modular code, making it easier to manage and maintain.

2. Reusability: Objects can be reused in different parts of the system, reducing redundancy.

3. Flexibility and Scalability: OOP supports scalability, enabling the system to adapt to changing requirements.

vi. Terms in OOP:

a. Constructor:

- Explanation: A constructor is a special method that is called when an object is instantiated. It initializes the object's state.

- Java Code:

```
public class MyClass {
   public MyClass() {
   }
}
```

b. Object:

- Explanation: An object is an instance of a class, encapsulating data and behavior.

- Java Code:

```
MyClass myObject = new MyClass();
```

c. Destructor:

- Explanation: Java doesn't have explicit destructors. Garbage collection automatically handles memory deallocation.

- Java Code: Not applicable.

d. Polymorphism:

- Explanation: Polymorphism allows objects of different types to be treated as objects of a common type.

- Java Code:

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

public class Circle implements Shape {
```

```
    public void draw() {
    }
}

public class Square implements Shape {
    public void draw() {
    }
}
```

e. Class:

- Explanation: A class is a blueprint for creating objects. It defines attributes and methods that the objects will have.

- Java Code:

```
public class MyClass {

}
```

f. Inheritance:

- Explanation: Inheritance allows a class (subclass) to inherit attributes and methods from another class (superclass).

- Java Code:

```
public class Animal {
    void eat() {
    }
}

public class Dog extends Animal {
    void bark() {

    }
```

}

vii. Types of Associations in OOP:

1. Association: A simple relationship where objects are associated but not dependent on each other.
2. Aggregation: A specialized form of association where one object contains another object.
3. Composition: A strong form of aggregation where the child object is part of the parent object's life cycle.

viii. Class Diagram

In object-oriented programming (OOP), a class diagram is a visual representation of the classes, relationships, and structures within a system. It illustrates the static aspects of a system, showing the classes, their attributes, methods, and how they interact.

Class diagrams are used during the design phase of software development to model the structure of a system. They provide a blueprint for developers and other stakeholders to understand the relationships and interactions between different components of the system.

Steps to Draw a Class Diagram:

1. Identify Classes:
   - Customer
   - Account
   - Transaction

2. Add Attributes and Methods:

   - Customer Class:
     - Attributes: CustomerID, Name, Address
     - Methods: OpenAccount(), CloseAccount()

- Account Class:
  - Attributes: AccountNumber, Balance
  - Methods: Deposit(), Withdraw()

- Transaction Class:
  - Attributes: TransactionID, Date, Amount
  - Methods: RecordTransaction()

3. Establish Relationships:

- Connect classes with lines to represent relationships. For example:
  - Customer has a relationship with Account (association).
  - Transaction has a relationship with Account (association).

4. Multiplicity:

- Indicate the multiplicity of associations, such as "1" for one-to-one and "*" for many.

5. Inheritance:

- If there are any hierarchical relationships, indicate them using a solid line with an arrowhead pointing to the superclass.

6. Dependencies:

- Add dependencies (dashed lines) between classes when one class uses another without being part of its structure.

Area and Perimeter Calculator in C++

};

```
class Square : public Shape {
private:
    double side;

public:
    Square(double s) : side(s) {}

    double calculateArea() const override {
        return side * side;
    }

    double calculatePerimeter() const override {
        return 4 * side;
    }
};

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



    Shape* shapes[] = {&circle, &rectangle, &triangle, &square};

    for (const auto& shape : shapes) {
        std::cout << "Area: " << shape->calculateArea() << ", Perimeter: " << shape->calculatePerimeter() << std::endl;
    }

    return 0;
}
```

OOP Concepts Implementation

a. Inheritance (Single, Multiple, Hierarchical

- Single Inheritance: Each shape class (Circle, Rectangle, Triangle, Square) inherits directly from the base class `Shape`.
- Multiple Inheritance: There's no explicit multiple inheritance in this example.
- Hierarchical Inheritance: The `Shape` class is a common base class for all shapes, forming a hierarchical inheritance structure.

b. Friend Functions

There are no friend functions explicitly used in this example. Friend functions allow external functions to access private members of a class.

c. Method Overloading and Method Overriding

- Method Overloading: This is demonstrated in the `Shape` class, where `calculateArea` and `calculatePerimeter` are overloaded for each derived shape class.
- Method Overriding:Each derived class provides its own implementation of `calculateArea` and `calculatePerimeter`, overriding the virtual functions in the base class.

d. Late Binding and Early Binding

- Late Binding (Dynamic Binding):
  - Achieved through the use of virtual functions (`calculateArea` and `calculatePerimeter`).
  - The actual function to be called is determined at runtime.
  - Example:

```
Shape* shapes[] = {&circle, &rectangle, &triangle, &square};
```

```
for (const auto& shape : shapes) {
    std::cout << "Area: " << shape->calculateArea() << ", Perimeter: " << shape->calculatePerimeter() << std::endl;
}
```

- The appropriate `calculateArea` and `calculatePerimeter` functions for each shape are determined dynamically at runtime.

- Early Binding (Static Binding):
  - The compiler knows at compile-time which function will be called for non-virtual functions.
  - Example:

```
double result = square.calculateArea();
```

  - The compiler knows at compile-time that `calculateArea` of the `Square` class will be called.

e. Abstract Class and Pure Functions

- Abstract Class:
  - The `Shape` class is abstract, as it contains pure virtual functions (`calculateArea` and `calculatePerimeter`).
  - Objects of abstract classes cannot be instantiated.
  - Example:

```
class Shape {
public:
    virtual double calculateArea() const = 0;
    virtual double calculatePerimeter() const = 0;
};
```

- Pure Functions:

  - The pure virtual functions in the `Shape` class are pure functions, as they are declared with `= 0` and must be implemented by derived classes.

  - Example:

```
    virtual double calculateArea() const = 0;

    virtual double calculatePerimeter() const = 0;
```

viii. Sure, let's go through each differentiation using examples in C++:

a. Function Overloading and Operator Overloading

Function Overloading:

Function overloading refers to defining multiple functions in the same scope with the same name but different parameter types or a different number of parameters. The compiler decides which function to call based on the number and types of arguments provided during the function call.

Example of Function Overloading in C++:

```
#include <iostream>

int sum(int a, int b) {
    return a + b;
}

double sum(double a, double b) {
    return a + b;
}
```

```
int main() {
    std::cout << "Sum of integers: " << sum(5, 3) << std::endl;
    std::cout << "Sum of doubles: " << sum(2.5, 3.7) << std::endl;

    return 0;
}
```

Operator Overloading:

Operator overloading involves defining how operators work for user-defined types. This allows you to use operators with objects of a class in a meaningful way.

Example of Operator Overloading in C++:

```
#include <iostream>

class Complex {
public:
    double real;
    double imaginary;

    Complex operator+(const Complex& other) {
        Complex result;
        result.real = this->real + other.real;
        result.imaginary = this->imaginary + other.imaginary;
        return result;
    }
};

int main() {
```

```
    Complex a{2.0, 3.0};
    Complex b{1.5, 2.5};

    Complex result = a + b;

    std::cout << "Sum of Complex Numbers: " << result.real << " + " << result.imaginary << "i" << std::endl;

    return 0;
}
```

b. Pass by Value and Pass by Reference

Pass by Value:

Passing by value involves passing the actual value of a variable to a function. Changes made to the parameter inside the function do not affect the original variable.

Example of Pass by Value in C++:

```
#include <iostream>

void modifyValue(int x) {
    x = 10;
}

int main() {
    int value = 5;

    modifyValue(value);

    std::cout << "Original Value: " << value << std::endl;
```

```
    return 0;
}
```

Pass by Reference:

Passing by reference involves passing a reference (memory address) of a variable to a function. Changes made to the parameter inside the function affect the original variable.

Example of Pass by Reference in C++:

```
#include <iostream>

void modifyReference(int& x) {
    x = 10;
}

int main() {
    int value = 5;

    modifyReference(value);

    std::cout << "Modified Value: " << value << std::endl;

    return 0;
}
```

c. Parameters and Arguments

Parameters:

Parameters are variables listed in a function's definition. They act as placeholders for the actual values that will be passed to the function during a function call.

Example of Parameters in C++:

```
#include <iostream>

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

int main() {
    int x = 3;
    int y = 4;

    printSum(x, y); // x and y are parameters

    return 0;
}
```

Arguments:

Arguments are the actual values passed to a function during a function call. They correspond to the parameters defined in the function's signature.

Example of Arguments in C++:

```
#include <iostream>
```

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

int main() {
    int x = 3;
    int y = 4;

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

    return 0;
}
```

V. Calculator G

```
public class CalculateG {

    private double gravity = -9.81; // Earth's gravity in m/s^2
    private double fallingTime = 30;
    private double initialVelocity = 0.0;
    private double finalVelocity;
    private double initialPosition = 0.0;
    private double finalPosition;

    // Method to compute position and velocity
    public void calculatePositionAndVelocity() {
        // Add the formulas for position and velocity
        finalPosition = 0.5 * gravity * Math.pow(fallingTime, 2) + initialVelocity * fallingTime + initialPosition;
        finalVelocity = initialVelocity + gravity * fallingTime;

        // Output the result
```

```
    System.out.println("The object's position after " + fallingTime + " seconds is " + finalPosition + " meters.");
    System.out.println("The object's velocity after " + fallingTime + " seconds is " + finalVelocity + " m/s.");
}

// Method for multiplication
public double multiply(double a, double b) {
    return a * b;
}

// Method for powering to square
public double powerToSquare(double a) {
    return Math.pow(a, 2);
}

// Method for summation
public double sum(double a, double b) {
    return a + b;
}

// Method for printing out a result
public void outline(double result) {
    System.out.println("Result: " + result);
}

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

    // Compute the position and velocity of an object
    calculator.calculatePositionAndVelocity();

    // Example usage of other methods
    double multiplicationResult = calculator.multiply(2.0, 3.0);
```

```
        double squareResult = calculator.powerToSquare(4.0);
        double summationResult = calculator.sum(5.0, 7.0);

        // Print out the results
        calculator.outline(multiplicationResult);
        calculator.outline(squareResult);
        calculator.outline(summationResult);
    }
}
```

PART B

QUESTION ONE

```
#include <iostream>

int main() {
    const int limit = 4000000;

    int firstTerm = 1;
    int secondTerm = 2;
    int nextTerm = firstTerm + secondTerm;
    int evenSum = 2;

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

        firstTerm = secondTerm;
        secondTerm = nextTerm;
```

```
        nextTerm = firstTerm + secondTerm;
    }

    std::cout << "Sum of even-valued terms in the Fibonacci sequence not exceeding four million: " << evenSum <<
std::endl;

    return 0;
}
```

QUESTION THREE

```
#include <iostream>

int main() {
    const int size = 15;
    int values[size];

    std::cout << "Enter 15 integer values:\n";
    for (int i = 0; i < size; ++i) {
        std::cout << "Value " << i + 1 << ": ";
        std::cin >> values[i];
    }
    std::cout << "\nValues stored in the array:\n";
    for (int i = 0; i < size; ++i) {
        std::cout << values[i] << " ";
    }
    std::cout << "\n";

    int numberToFind;
    std::cout << "\nEnter a number to find in the array: ";
    std::cin >> numberToFind;
```

```
bool numberFound = false;
for (int i = 0; i < size; ++i) {
    if (values[i] == numberToFind) {
        std::cout << "The number found at index " << i << "\n";
        numberFound = true;
        break;
    }
}

if (!numberFound) {
    std::cout << "Number not found in this array\n";
}



int reversedArray[size];
for (int i = 0; i < size; ++i) {
    reversedArray[i] = values[size - i - 1];
}



std::cout << "\nElements of the new array in reverse order:\n";
for (int i = 0; i < size; ++i) {
    std::cout << reversedArray[i] << " ";
}
std::cout << "\n";

int sum = 0;
long long product = 1;

for (int i = 0; i < size; ++i) {
    sum += values[i];
```

```
        product *= values[i];
    }

   t
    std::cout << "\nSum of all elements: " << sum << "\n";
    std::cout << "Product of all elements: " << product << "\n";

    return 0;
}
```