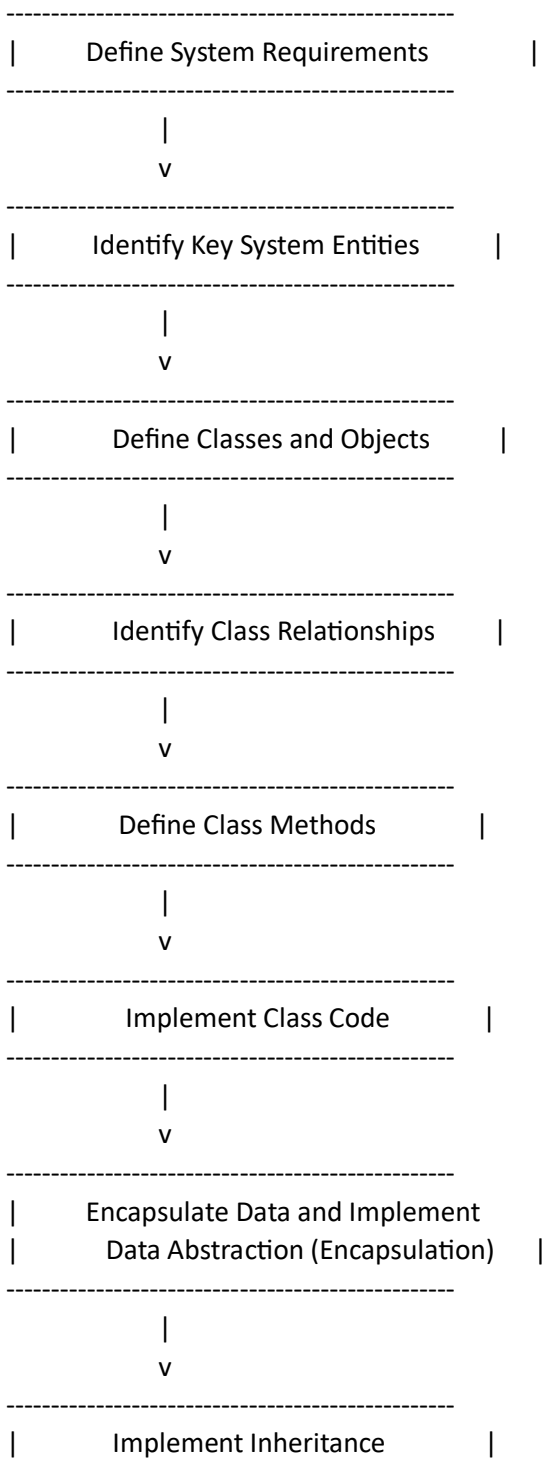
GERALD MANGERA

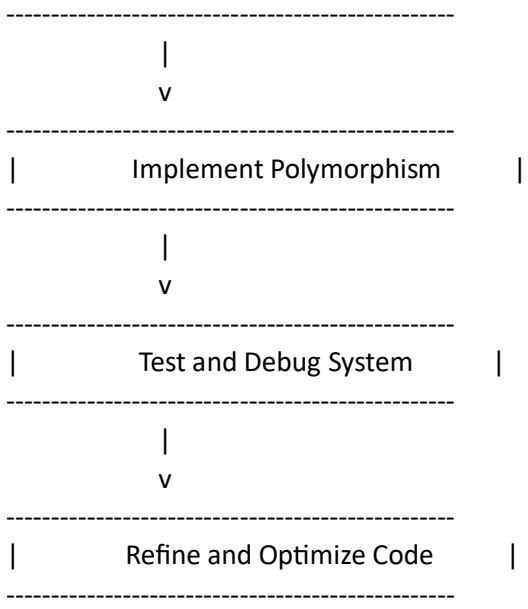
SCT121-0972/2022

OOP ASSIGNMENT

PART A

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





II. **What is the Object Modeling Techniques (OMT**

Object Modelling Techniques (OMT) is a method for visualizing and documenting the analysis and design of a system using object-oriented concepts and principles.

III. **Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP).**

Object-oriented analysis and design (OOAD) focuses on modelling and planning the structure and behaviour of a system, while object-oriented programming (OOP) involves the actual implementation of that design in executable code using a programming language.

IV. **Discuss Mian goals of UML.**

- **Standardization:**

  Provide a standardized modelling language that enables a common visual representation for system design and analysis.

- **Communication:**

  Enhance communication among stakeholders, including developers, analysts, designers, and clients, by providing a common set of visual models and notations.

- **Clarity:**

  Improve the clarity and understanding of system requirements, architecture, and behaviour through visual representation, reducing ambiguity in communication.

- **Flexibility:**

  Support a wide range of systems and application domains, making it applicable to various types of software

V. **DESCRIBE three advantages of using object oriented to develop an information system.**

- **Modularity and Reusability:**

Object-oriented programming (OOP) promotes modularity by organizing code into objects, which encapsulate data and behaviour. This modular structure enhances code readability, maintainability, and reusability. Developers can reuse existing classes and objects in new

projects, saving time and effort. This promotes a more efficient development process and reduces the likelihood of errors since proven and tested components can be reused.

- **Encapsulation and Security:**
  Encapsulation, a key principle in OOP, involves bundling data and the methods that operate on the data within a single unit, i.e., an object. This encapsulation provides a protective barrier that prevents external code from directly accessing an object's internal state. The object exposes only the necessary methods to interact with its data, enhancing security and reducing the risk of unintended interference. This helps in building more secure and robust information systems, particularly in scenarios where sensitive data handling is critical.
- **Ease of Maintenance and Scalability:**
  OOP facilitates easier maintenance of codebases. Each object is responsible for a specific functionality, making it easier to locate and fix bugs or add new features without affecting the entire system. This results in a more scalable and adaptable information system. Additionally, the use of inheritance and polymorphism in OOP allows developers to extend and modify existing classes without altering their core functionalities. This makes it easier to accommodate changes in requirements and scale the system as needed over time, leading to a more flexible and maintainable codebase.

VI. **Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept.**

- **CONSTRUCTOR**

- A constructor is a special member function of a class that is automatically called when an object of the class is created. It is used to initialize the object's data members or perform any necessary setup operations.

```
C++ Code:
#include <iostream>

class MyScore{
public:
   // Constructor
   MyScore() {
      std::cout << "Constructor called. Object created!" << std::endl;
   }
};

int main() {

   MyScore obj;

   return 0;
}
```

- **OBJECT**

An object is an instance of a class. It is a concrete realization of a class, representing a specific entity in a program with its own set of attributes (data members) and behaviors (methods).
C++ Code:

```
#include <iostream>

class Car {
public:
    std::string brand;
    int year;

    void displayInfo() {
        std::cout << "Brand: " << brand << ", Year: " << year << std::endl;
    }
};

int main() {
    // Creating an object of Car
    Car myCar;
    myCar.brand = "Mobius";
    myCar.year = 2022;

    // Calling a method on the object
    myCar.displayInfo();

    return 0;
}
```

DESTRUCTOR

A destructor is a special member function of a class that is called when an object goes out of scope or is explicitly deleted. It is used to perform cleanup operations, such as releasing resources allocated by the object.

C++ code;

```
#include <iostream>

class MyClass {
public:
    // Constructor
    MyClass() {
        std::cout << "Constructor called. Object created!" << std::endl;
    }

    // Destructor
    ~MyClass() {
        std::cout << "Destructor called. Object destroyed!" << std::endl;
    }
};

int main() {
    // Creating an object of MyClass
```

```
    MyClass obj;

    // Object goes out of scope, destructor is called automatically

    return 0;
}
```

POLYMORPHISM

Polymorphism allows objects of different classes to be treated as objects of a common base class. It includes concepts like function overloading and virtual functions.

C++ code;

```
#include <iostream>

// Base class
class Shape {
public:
    // Virtual function
    virtual void draw() {
        std::cout << "Drawing a shape." << std::endl;
    }
};

// Derived class
class Circle : public Shape {
public:
    // Override the draw function
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

int main() {
    // Using polymorphism
    Shape* shapePtr = new Circle();
    shapePtr->draw();  // Calls the draw function of Circle

    delete shapePtr;

    return 0;
}
```

- **CLASS**

  A class is a blueprint From which objects are created. It defines the properties (attributes) and behaviors (methods) that objects of the class will have.

  C++ code;

```
#include <iostream>

// Class definition
class Rectangle {
```

```
public:
    // Data members
    int length;
    int width;

    // Member function to calculate area
    int calculateArea() {
        return length * width;
    }
};

int main() {
    // Creating an object of Rectangle
    Rectangle myRectangle;
    myRectangle.length = 5;
    myRectangle.width = 3;

    // Calling a method on the object
    int area = myRectangle.calculateArea();
    std::cout << "Area of the rectangle: " << area << std::endl;

    return 0;
```

- **INHERITANCE**

Inheritance is a mechanism in which a class (subclass or derived class,child class) inherits properties and behaviours from another class (base class or superclass, parent class), allowing code reuse and the creation of a hierarchy of classes.

C++ code;

```
#include <iostream>


// Base class
class Animal {
public:
    void sound() {
        std::cout << "Animal makes a sound." << std::endl;
    }
};


// Derived class
class Dog : public Animal {
public:
    void bark() {
```

```
        std::cout << "Dog barks." << std::endl;
    }
};

int main() {
    // Using inheritance
    Dog myDog;
    myDog.sound();  // Inherited from Animal class
    myDog.bark();

    return 0;
}
```

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

- **ASSOCIATION**
  Association is a basic and general relationship between two or more classes where objects of one class are related to objects of another class. It represents a connection between classes and is often used to model relationships that are not strong enough to warrant aggregation or composition.
- **AGGREGATION**
  Aggregation is a more specialized form of association, indicating a "has-a" relationship where one class (the whole) contains another class (the part). The part can exist independently, and it can be part of multiple wholes. Aggregation is typically represented by a diamond-shaped line with an open arrowhead.
- **COMPOSITION**
  Composition is a stronger form of aggregation, indicating a "whole-part" relationship where one class (the whole) is composed of another class (the part). The part is an integral part of the whole, and it cannot exist independently. Composition is often depicted by a solid diamond-shaped line

VIII. **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]**

Abstract class and pure functions

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

// Abstract class with pure virtual functions
class Shape {
public:
    virtual double calculateArea() const = 0;
    virtual double calculatePerimeter() const = 0;
};

// Derived class for Circle
class Circle : public Shape {
private:
    double radius;

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

    double calculateArea() const override {
        return 3.14 * radius * radius;
    }

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

// Derived class for Rectangle
class Rectangle : public Shape {
private:
    double length;
    double width;

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

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

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

// Derived class for Triangle
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) {}

    double calculateArea() const override {
        double s = (side1 + side2 + side3) / 2.0;
        return sqrt(s * (s - side1) * (s - side2) * (s - side3));
    }

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

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

// Friend function to output shape details
void printDetails(const Shape& shape) {
    std::cout << "Area: " << shape.calculateArea() << std::endl;
    std::cout << "Perimeter: " << shape.calculatePerimeter() << std::endl;
}

int main() {
    Circle circle(5);
    Rectangle rectangle(4, 6);
    Triangle triangle(3, 4, 5);
    Square square(7);

    // Friend function usage
    printDetails(circle);
    printDetails(rectangle);
    printDetails(triangle);
    printDetails(square);

    return 0;
}
```

IX. Using a program written in C++, differentiate between the following. [6 Marks]

a. **Function overloading and operator overloading**
b. **Pass by value and pass by reference**
c. **Parameters and arguments**

```
#include <iostream>

// Function Overloading
int add(int a, int b) {
```

```
    return a + b;
}

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

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

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

// Pass by Value
void modifyValue(int x) {
    x += 5;
    std::cout << "Inside modifyValue: " << x << std::endl;
}

// Pass by Reference
void modifyReference(int& x) {
    x += 5;
    std::cout << "Inside modifyReference: " << x << std::endl;
}

int main() {
    // Function Overloading
    std::cout << "Function Overloading:" << std::endl;
    std::cout << "Adding integers: " << add(3, 5) << std::endl;
    std::cout << "Adding doubles: " << add(3.5, 5.7) << std::endl;

    // Operator Overloading
    std::cout << "\nOperator Overloading:" << std::endl;
    Complex a{1.0, 2.5};
    Complex b{0.5, 1.2};
    Complex result = a + b;

  std::cout << "Result of complex addition: " << result.real << " + " << result.imag <<"i" <<
std::endl;

    // Pass by Value
```

```
        std::cout << "\nPass by Value:" << std::endl;
        int value = 10;
        modifyValue(value);
        std::cout << "Outside modifyValue: " << value << std::endl;

        // Pass by Reference
        std::cout << "\nPass by Reference:" << std::endl;
        int reference = 10;
        modifyReference(reference);
        std::cout << "Outside modifyReference: " << reference << std::endl;

        return 0;
    }
```

X.

```
#include <iostream>


class CalculateG {
private:
    double gravity = -9.81;
    double fallingTime = 30;
    double initialVelocity = 0.0;
    double finalVelocity;
    double initialPosition = 0.0;
    double finalPosition;


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

    // Method for powering to square
    double powerSquare(double a) {
        return a * a;
    }
```

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

    // Method for printing out a result
    void outline(double result) {
        std::cout << "Result: " << result << std::endl;
    }

    // Method to compute the position and velocity
    void computePositionAndVelocity() {
        finalPosition = 0.5 * gravity * fallingTime * fallingTime + initialVelocity * fallingTime +
initialPosition;
        finalVelocity = gravity * fallingTime + initialVelocity;
    }

    // Main method
    int main() {
        // Compute the position and velocity
        computePositionAndVelocity();

        // Print out the results
        std::cout << "The object's position after " << fallingTime << " seconds is " << finalPosition << "
m." << std::endl;
        std::cout << "The object's velocity after " << fallingTime << " seconds is " << finalVelocity << "
m/s." << std::endl;

        return 0;
    }
};

int main() {
```

```
    // Create an instance of the CalculateG class
    CalculateG calculator;

    // Call the main method of the class
    calculator.main();

    return 0;
}
```

PART B

```
#include <iostream>

// Function to find the sum of even-valued terms in the Fibonacci sequence
int sumEvenFibonacci(int limit) {
    int term1 = 1, term2 = 2, nextTerm, sum = 0;

    while (term1 <= limit) {
        if (term1 % 2 == 0) {
            sum += term1;
        }

        nextTerm = term1 + term2;
        term1 = term2;
        term2 = nextTerm;
    }

    return sum;
}

int main() {
    int limit = 4000000;
```

```
    // Find and print the sum of even-valued terms
    int result = sumEvenFibonacci(limit);

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

    return 0;
}
```

QUESTION 3

```
#include <iostream>

int main() {
    const int arraySize = 15;
    int numbers[arraySize];

    // Part a: Input 15 values from the user
    std::cout << "Enter 15 integers, one at a time:" << std::endl;
    for (int i = 0; i < arraySize; ++i) {
        std::cout << "Enter value " << (i + 1) << ": ";
        std::cin >> numbers[i];
    }

    // Part b: Print the values stored in the array
    std::cout << "\nValues stored in the array:" << std::endl;
    for (int i = 0; i < arraySize; ++i) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl;

    // Part c: Check if a number is present in the array
    int searchNumber;
    std::cout << "\nEnter a number to search in the array: ";
    std::cin >> searchNumber;

    bool found = false;
    for (int i = 0; i < arraySize; ++i) {
        if (numbers[i] == searchNumber) {
            std::cout << "The number found at index " << i << std::endl;
            found = true;
            break;
        }
    }
```

```
    if (!found) {
        std::cout << "Number not found in this array." << std::endl;
    }

    // Part d: Create another array, copy elements in reverse order, and print the new array
    int reversedNumbers[arraySize];
    for (int i = 0; i < arraySize; ++i) {
        reversedNumbers[i] = numbers[arraySize - i - 1];
    }

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

    // Part e: Get the sum and product of all elements
    int sum = 0;
    long long product = 1;

    for (int i = 0; i < arraySize; ++i) {
        sum += numbers[i];
        product *= numbers[i];
    }

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

    return 0;
}
```