NAME: CHRISTINE MACIRA

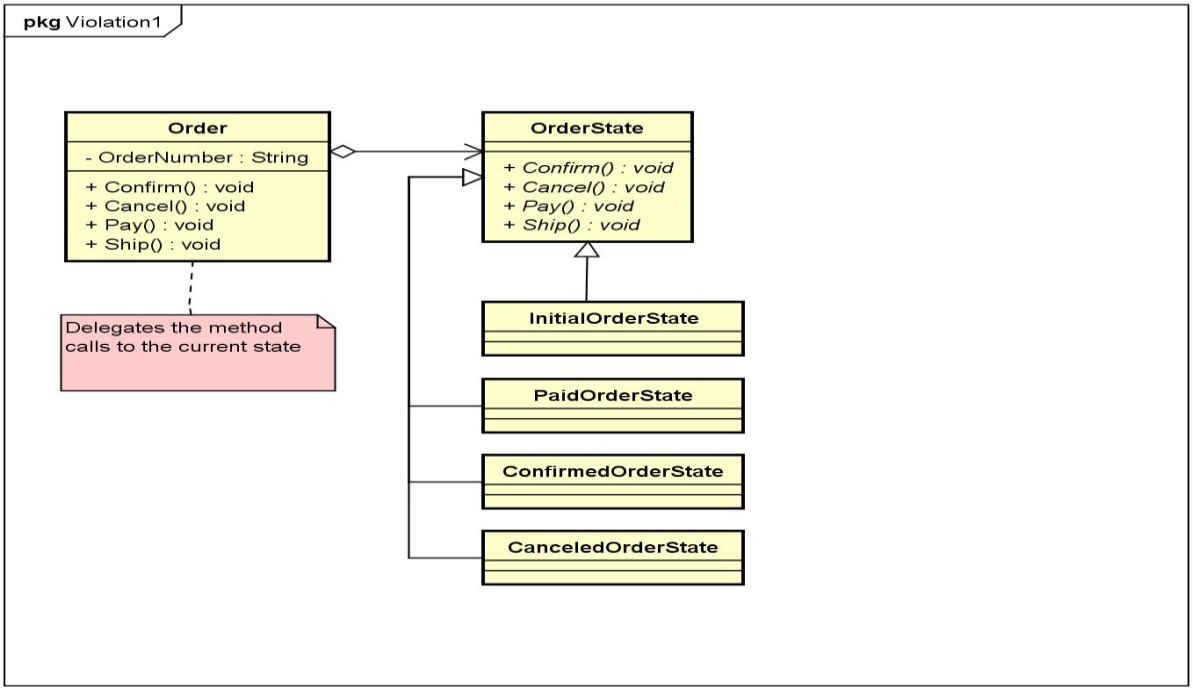
REG N.O: SCT121-0939/2022

DIT 0301: OBJECT ORIENED PROGRAMMING ASSIGNMENT.

Part A:

1. **Using a well labeled diagram, explain the steps of creating a system using OOP principles [4 marks]**

- Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on the data into a single unit known as a class.
- Inheritance allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class).
- Polymorphism allows objects of different classes to be treated as objects of a common base class. It involves method overloading and method overriding.
- Abstraction involves simplifying complex systems by modeling classes based on the essential features they share, while ignoring irrelevant details.
- Composition is the concept of combining simple objects to create more complex ones. It involves creating objects within objects.



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

- Object Modelling: This involves defining the objects in a system and their relationships.
- Dynamic Modelling: It focuses on the behaviour of objects over time. It includes state diagrams, activity diagrams.

- Functional Modelling**:** Functional modelling involves creating data flow diagrams to represent the flow of data within the system.

3. **Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP). [2 Marks]**
    - OOAD focuses on the systematic process of analyzing and designing a system from an object-oriented perspective while OOP is the actual implementation phase where the designs created during OOAD are translated into executable code.
    - OOAD typically involves the use of modeling techniques such as UML (Unified Modeling Language) to create diagrams like class diagrams, sequence diagrams, state diagrams while in OOP Programming languages that support object-oriented features, such as Java, C++, Python, and others, are used for OOP. Developers write code to instantiate classes, create objects, and define their behavior.

4. **Discuss Mian goals of UML. [2 Marks]**

- UML provides a graphical representation that allows developers and stakeholders to visually model the structure and behavior of a system.
- UML supports various levels of abstraction, allowing practitioners to model different aspects of a system, including structural, behavioral, and architectural views.
- UML promotes effective communication among stakeholders by providing a common visual language for expressing software design concepts**.**

5. **Describe three advantages of using object oriented to develop an information system. [3Marks]**
    - Modularity that facilitates easier maintenance as changes can be made to specific classes without affecting the entire system
    - Encapsulation which enhances security by controlling access to the internal state of objects reducing the risk of unintended interference.
    - Code reusability where inherited classes can reuse the attributes and methods of their parent classes, reducing the need to rewrite code and enhancing development efficiency

6. **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]**
    a. **Constructor**
    - It is a special type of method that is used to initialize objects. It is used to Initialize object's state.

```
public class Car {
   private String brand;
   private String model;
```

```
    // Constructor
    public Car (String brand, String model) {
        this. Brand = brand;
        this. Model = model;
        System.out.println("A new car is created!");
    }

    // Other methods and properties...
}

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

b. **Object**

- It is an instance of a class.

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

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

    // Other methods and properties...
}

// Creating objects (instances) of the Student class
Student student1 = new Student("Alice", 20);
Student student2 = new Student("Bob", 22);
```

c. **Destructor**

- It is a special method that is automatically invoked when an object is about to be destroyed or its memory is about to be released.

There is no specific syntax in java for destructors

d. **Polymorphism**

- It is a concept that allows objects of different types to be treated as objects of a common type. It can be through method overloading or method overriding.

Through method overloading:

```
public class Calculator {
    // Method overloading
    public int add(int)
```

e. **Class**

- It is a group of objects with mutual methods. It is a blue print for creating objects.

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

f. **Inheritance**

- It is a mechanism where an object acquires all properties and behaviour of a parent object.

```
// Parent class
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}

// Child class
class Dog extends Animal {
    void bark() {
        System.out.println("barking...");
    }
}
```

7. **Explain the three types of associations (relationships) between objects in object oriented. [6 Marks]**
    - Aggregation represents a "has-a" relationship between objects, where one object contains another object, but the contained object can exist independently of the container. It is often denoted by a diamond shape on the side of the container class.
    - Composition is a stronger form of aggregation where the lifetime of the contained object is tied to the lifetime of the container object. If the container is destroyed, the contained object is also destroyed. It is denoted by a filled diamond shape on the side of the container class.
    - Association represents a more general relationship between objects where they are connected, but the connection is more flexible. Objects in an association can exist independently of each other. It is typically represented by a line connecting the associated classes, with an optional arrow indicating the direction of the association.
8. **What do you mean by class diagram? Where it is used and also discuss the steps to draw the class diagram with any one example.**
   [6 Marks]
    - A class diagram is a type of static structure diagram in the UML that represents the structure and relationships of classes within a system.

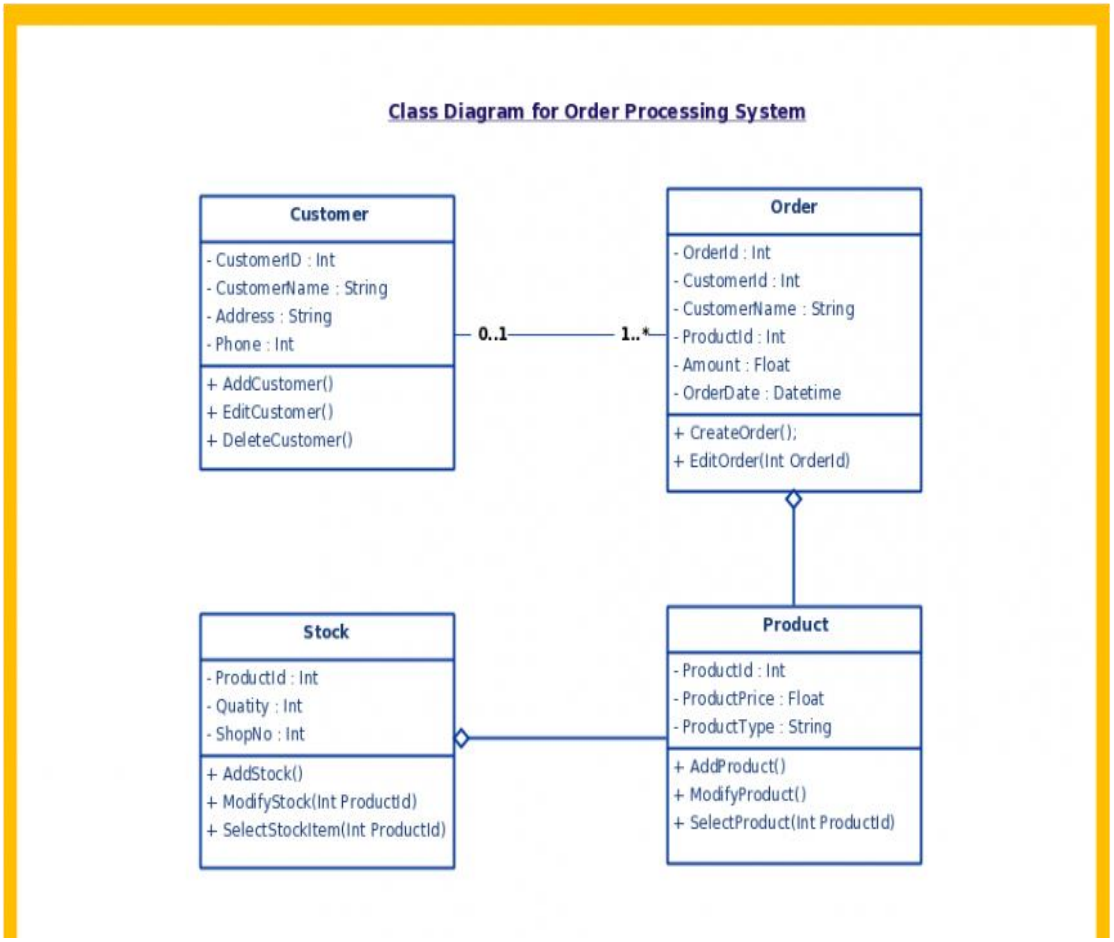
Class diagrams are widely used in;

- During the analysis and design phases of software development, class diagrams help in visualizing and designing the structure of the system.
- Class diagrams serve as documentation, providing a clear and concise representation of the static aspects of a system.

- They aid in communication between stakeholders, including developers, designers, and clients, by providing a visual representation of the system's architecture.

Steps to draw a Class Diagram:

1. Identify the main classes in your system.
2. Determine the attributes (data members) that represent the characteristics of the class.
3. Identify the methods (operations) that each class can perform.
4. Identify relationships between classes. Determine if the relationships are associations, aggregations, compositions, or generalizations.
5. Specify the multiplicity of the associations, indicating how many instances participate in the relationship.
6. Draw the diagram.



9. 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) [10Marks]**

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

using namespace std;
// Abstract class Shape
class Shape {
public:
   virtual double area() = 0;
   virtual double perimeter() = 0;
};

// Derived class Circle
class Circle : public Shape {
private:
   double radius;
public:
   Circle(double r) : radius(r) {}
   double area() override { return M_PI * radius * radius; }
   double perimeter() override { return 2 * M_PI * radius; }
};

// Derived class Rectangle
class Rectangle : public Shape {
private:
   double length, width;
public:
   Rectangle(double l, double w) : length(l), width(w) {}
   double area() override { return length * width;}
   double perimeter() override { return 2 * (length + width);}
};

// Derived class Triangle
class Triangle : public Shape {
private:
   double a, b, c;
public:
   Triangle(double side1, double side2, double side3): a(side1), b(side2), c(side3) {}
   double area() override {
      double s = (a + b + c) / 2;
      return sqrt(s * (s - a) * (s - b) * (s - c));
   }
   double perimeter() override { return a + b + c;}
};

// Derived class Square
class Square : public Rectangle {
```

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

int main() {
    // Circle example
    Circle c(5);
    cout << "Circle area: " << c.area() << endl;
    cout << "Circle perimeter: " << c.perimeter() << endl;

    // Rectangle example
    Rectangle r(4, 6);
    cout << "Rectangle area: " << r.area() << endl;
    cout << "Rectangle perimeter: " << r.perimeter() << endl;

    // Triangle example
    Triangle t(3, 4, 5);
    cout << "Triangle area: " << t.area() << endl;
    cout << "Triangle perimeter: " << t.perimeter() << endl;

    // Square example
    Square s(7);
    cout << "Square area: " << s.area() << endl;
    cout << "Square perimeter: " << s.perimeter() << endl;

    return 0;
}
```

b. Friend functions

```
class Shape {
public:
    virtual double area() const = 0;
    virtual double perimeter() const = 0;

    // Friend function declaration
    friend void printDetails(const Shape& s);
};

void printDetails(const Shape& s) {
    std:cout << "Area: " << s.area() << ", Perimeter: " << s.perimeter() << std:endl;
}
```

c. Method overloading and method overriding

```
// In the Shape class
class Shape {
public:
   // Method Overloading
   void display() const {
      std:cout << "This is a shape." << std:endl;
```

```
    }

    // Method Overriding
    virtual void showDetails() const {
        std:cout << "Details of a shape." << std:endl;
    }
};

// In the Circle class
class Circle : public Shape {
public:
    // Method Overriding
    void showDetails() const override {
        std:cout << "Details of a circle." << std:endl;
    }
};

// Usage
int main() {
    Shape shape;
    shape.display();        // Calls the display method in Shape class
    shape.showDetails();    // Calls the showDetails method in Shape class

    Circle circle;
    circle.display();       // Calls the display method in Shape class (method overloading)
    circle.showDetails();       // Calls the showDetails method in Circle class (method
overriding)

    return 0;
}
```

d. Late binding and early binding

```
     // In the Shape class
class Shape {
public:
    // Early Binding (Non-virtual function)
    void printType() const {
        std:cout << "Shape" << std:endl;
    }

    // Late Binding (Virtual function)
    virtual void displayType() const {
        std::cout << "Shape" << std::endl;
    }
};

// In the Circle class
class Circle : public Shape {
public:
    // Early Binding (Non-virtual function)
```

```
    void printType() const {
        std::cout << "Circle" << std::endl;
    }

    // Late Binding (Virtual function)
    void displayType() const override {
        std::cout << "Circle" << std::endl;
    }
};

// Usage
int main() {
    Shape shape;
    Circle circle;

    // Early Binding
    shape.printType();      // Calls printType in Shape class
    circle.printType();     // Calls printType in Circle class

    // Late Binding
    shape.displayType();    // Calls displayType in Shape class
    circle.displayType();   // Calls
```

e. Abstract class and pure functions.

```
// Abstract class (contains pure virtual functions)
class Shape {
public:
    virtual double area() const = 0;       // Pure virtual function
    virtual double perimeter() const = 0;  // Pure virtual function
};

// Concrete class (inherits from abstract class)
class Circle : public Shape {
private:
    double radius;

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

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

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

// Usage
int main() {
```

```
    // Shape shape;   // Error: Cannot create an instance of an abstract class
    Circle circle(5);
    std:cout << "Circle Area: " << circle.area() << ", Perimeter: " << circle.perimeter() <<
std:endl;

    return 0;
}
```

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

a. Function overloading and operator overloading

```
#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, imag;

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

int main() {
    // Function Overloading
    std:cout << "Sum of integers: " << add(3, 5) << std:endl;
    std:cout << "Sum of doubles: " << add(3.5, 2.7) << std:endl;

    // Operator Overloading
    Complex c1{2.0, 3.0};
    Complex c2{1.5, 2.5};
    Complex sum = c1 + c2;
    std:cout << "Sum of complex numbers: Real = " << sum.real << ", Imaginary
= " << sum.imag << std:endl;

    return 0;
}
```

b. Pass by value as pass by reference

```
#include <iostream>

// Pass by Value
void modifyValue(int x) {
   x = 10;
}

// Pass by Reference
void modifyReference(int& x) {
   x = 10;
}

int main() {
   int value = 5;
   int reference = 5;

   // Pass by Value
   modifyValue(value);
   std:cout << "Value after pass by value: " << value << std:endl;

   // Pass by Reference
   modifyReference(reference);
   std:cout << "Value after pass by reference: " << reference << std:endl;

   return 0;
}
```

c. Parameters and arguments

```
#include <iostream>

// Function with parameters
void addNumbers(int a, int b) {
   int sum = a + b;
   std:cout << "Sum: " << sum << std:endl;
}

int main() {
   int num1 = 5;
   int num2 = 7;

   // Function call with arguments
   addNumbers(num1, num2);

   return 0;
}
```

- - In the function addNumbers, a and b are parameters.
- Parameters are variables declared in the function signature to receive values during the function call.

- In the main function, num1 and num2 are arguments.
- Arguments are the actual values or expressions that are passed to a function during its call.

NOTE: To score high marks, you are required to explain each question in detail. Do good research and cite all the sources of your information. DO NOTE CITE WIKIPEDIA.

Create a new class called *CalculateG.*
Copy and paste the following initial version of the code. Note variables declaration and the types.

```
class CalculateG {
int main(){

(datatype) gravity =-9.81; // Earth's gravity in m/s^2 (datatype) falling Time =
30;

(datatype)initialVelocity = 0.0; (datatype) finalVelocity = ;

(datatype) initialPosition = 0.0; (datatype) finalPosition = ;

        // Add the formulas for position and velocity
        Cout<<"The object's position after " << fallingTime << " seconds is "
        + finalPosition + << m."<<endl;
    // Add output line for velocity (similar to position)

} }
```

Modify the example program to compute the position and velocity of an object after falling for 30 seconds, outputting the position in meters. The formula in Math notation is:

$x(t)=0.5*at^2+v_it+x_i\, v(t)=at+v_i$

Run the completed code in Eclipse (Run → Run As → Java Application). 5. Extend `datatype` class with the following code:

```
public class CalculateG {

public double multi(......){ // method for multiplication

}

// add 2 more methods for powering to square and summation (similar to
multiplication)

public void outline (......){
// method for printing out a result

}
int main () {

// compute the position and velocity of an object with defined methods and print
out the
result
```

} }

6. Create methods for multiplication, powering to square, summation and printing out a result in *CalculateG* class.

```
include <iostream>

using namespace std;

class CalculateG {

private:

    double gravity = -9.81; // Earth's gravity in m/s^2

    double fallingTime = 30;

    double initialVelocity = 0.0;

    double finalVelocity = 0.0;

    double initialPosition = 0.0;

    double finalPosition = 0.0;

public:

    void calculate() {

        // Add the formulas for position and velocity

        finalPosition = 0.5 * gravity * fallingTime * fallingTime + initialVelocity * fallingTime
+ initialPosition;

        finalVelocity = gravity * fallingTime + initialVelocity;

        cout << "The object's position after " << fallingTime << " seconds is " << finalPosition
<< " m." << endl;

        cout << "The object's velocity after " << fallingTime << " seconds is " << finalVelocity
<< " m/s." << endl;

    }

    double multiply(double a, double b) {
```

```
        return a * b;
    }
    double power(double a) {
        return a * a;
    }
    double sum(double a, double b) {
        return a + b;
    }
    void outline(string text) {
        cout << text << endl;
    }
};
int main() {
    CalculateG obj;
    obj.calculate();
    obj.outline("Multiplication: " + to_string(obj.multiply(2, 3)));
    obj.outline("Power: " + to_string(obj.power(4)));
    obj.outline("Sum: " + to_string(obj.sum(2, 3)));
    return 0;
}
```

Part B:

Instructions for part B: Do question 1 and any other one question from this section.

1. Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, write a C++ method to find the sum of all the even- valued terms.

```
#include <iostream>


int main() {
    int limit = 4000000;


    int first = 1;
    int second = 2;
    int nextTerm = 0;
    int sum = 2;  // Initializing with the sum of the first even term (2)


    while (nextTerm <= limit) {
        nextTerm = first + second;
        if (nextTerm % 2 == 0) {
            sum += nextTerm;
        }


       // Update the values for the next iteration
        first = second;
        second = nextTerm;
    }


    std:cout << "Sum of even-valued terms in the Fibonacci sequence below 4 million: " << sum
<< std:endl;
```

```
    return 0;

}
```

Question three: [15 marks]

Write a C++ program that takes 15 values of type integer as inputs from user, store the values in an array.

a) Print the values stored in the array on screen.
b) Ask user to enter a number, check if that number (entered by user) is present in array or not. If it is present print, "the number found at index (index of the number) " and the text "number not found in this array"
c) Create another array, copy all the elements from the existing array to the new array but in reverse order. Now print the elements of the new array on the screen
d) Get the sum and product of all elements of your array. Print product and the sum each on its own line.

```
#include <iostream>

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

    // a) Input values from the user
    std:cout << "Enter 15 integer values:" << std:endl;
    for (int i = 0; i < size; ++i) {
        std:cout << "Enter value #" << i + 1 << ": ";
        std:cin >> values[i];
    }

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

    // Ask the user to enter a number
    int userNumber;
    std::cout << "\nEnter a number to search in the array: ";
    std::cin >> userNumber;

    // Check if the number is present in the array
    bool found = false;
    int index;
    for (int i = 0; i < size; ++i) {
        if (values[i] == userNumber) {
            found = true;
            index = i;
            break;
```

```
        }
    }

    // Print the result
    if (found) {
        std::cout << "The number found at index " << index << std::endl;
    } else {
        std::cout << "Number not found in this array" << std::endl;
    }

    // c) Create another array and copy elements in reverse order
    int reversedArray[size];
    for (int i = 0; i < size; ++i) {
        reversedArray[i] = values[size - 1 - i];
    }

    // Print elements of the new array
    std::cout << "\nThe elements of the new array in reverse order are:" << std::endl;
    for (int i = 0; i < size; ++i) {
        std::cout << reversedArray[i] << " ";
    }
    std::cout << std::endl;

    // d) Calculate and print the sum and product of array elements
    int sum = 0;
    long long product = 1;
    for (int i = 0; i < size; ++i) {
        sum += values[i];
        product *= values[i];
    }

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

    return 0;
}
```