JOHNY MAUNDU
SCT121-0945/2022

i. Creating a System Using OOP Principles:

To create a system using Object-Oriented Programming (OOP) principles:

Identify Objects: Identify the key entities in the system and represent them as objects. Objects have attributes (data) and behaviors (methods).

Define Classes: Group similar objects into classes. Classes act as blueprints for creating objects. They encapsulate attributes and methods.

Establish Relationships: Define relationships between classes, such as associations, aggregations, and compositions. This helps in modeling how objects collaborate and interact.

Encapsulation: Hide the internal details of a class and expose only what is necessary. This is achieved through access modifiers (public, private, protected).

Inheritance: Use inheritance to model the "is-a" relationship between classes. Subclasses inherit attributes and behaviors from their superclass.

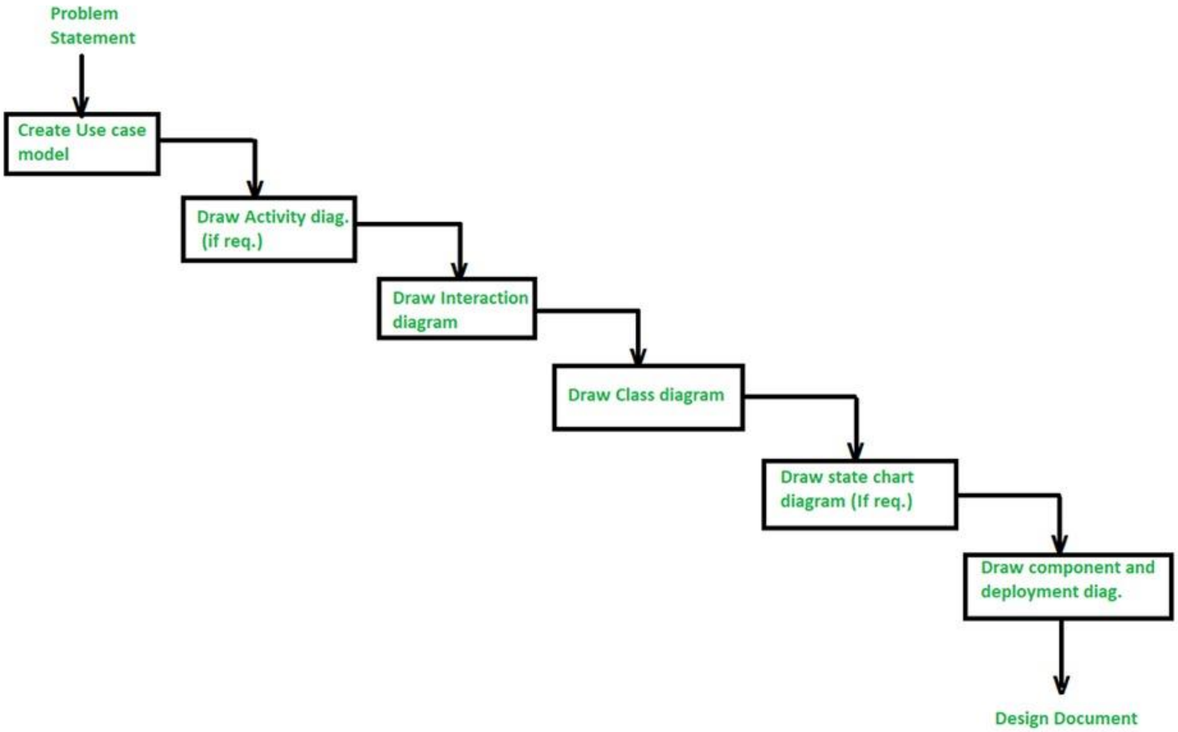
Polymorphism: Allow objects to take on multiple forms. This can be achieved through method overloading and overriding.

Abstraction: Abstract complex systems by simplifying them into manageable components. Focus on relevant details while hiding unnecessary complexity.

Modularity: Break down the system into smaller, independent modules. Each module should have a specific responsibility.

JOHNY MAUNDU
SCT121-0945/2022

OOP System Creation Diagram



ii. Object Modeling Techniques (OMT):

is a method for modeling and designing systems using object-oriented concepts. It includes techniques for identifying, defining, and specifying the objects and their relationships in a system.

iii. Object-Oriented Analysis and Design (OOAD) vs. Object Analysis and Design (OOP):

OOAD involves analyzing and designing a system from an object-oriented perspective, considering both analysis and design phases.

OOP typically refers to the broader concept of programming using object-oriented principles without specifically emphasizing the analysis and design phases.

iv. Main Goals of UML:

Standardization: Provide a standardized way to visualize, document, construct, and communicate the artifacts of a software system.

Specification: Specify the structure and behavior of the system using a visual modeling language.

Visualization: Facilitate understanding of system architecture and design through graphical representations.

JOHNY MAUNDU
SCT121-0945/2022

v. Advantages of Using Object-Oriented Programming:

Modularity: Encourages modular design, making it easier to understand, maintain, and update the code.

Reusability: Promotes reuse of code through inheritance and polymorphism, reducing redundancy.

Flexibility and Extensibility: Supports the addition of new features and modifications without affecting existing code.

vi. Explanation of Object-Oriented Programming Terms with Java Code:

a**. Constructor:** a constructor is a special method or function that is automatically called when an object is created from a class. Its primary purpose is to initialize the object's attributes or properties and perform any necessary setup for the object to be in a valid and usable state.

```
public class My Class {

   private int value;


   // Constructor

   public My Class (int initial Value) {

      this .value = initial Value;

   }


   public int get Value() {

      return value;

   }


   public static void main (String [] args) {

      // Creating an object and initializing it using the constructor

      My Class my Object = new My Class (10);


      // Accessing the value through a getter method

      System .out. print ln ("Initial value: " + my Object. Get Value ());

   }

}
```

JOHNY MAUNDU
SCT121-0945/2022

b. object: an object is an instance of a class. Objects are created based on these classes, and they represent real-world entities, concepts, or instances of a particular type.

```
public class Dog {
   String breed;
   Int age;

   public void bark () {
      System .out. print ln("Woof!");
   }

   public static void main (String [] args) {
      // Creating objects of the Dog class
      Dog my Dog = new Dog ();
      My Dog. breed = "Labrador";
      My Dog. age = 3;

      // Accessing state and behavior of the object
      System. out. print ln ("Breed: " + my Dog. breed);
      System .out. print ln ("Age: " + my Dog. age);
      My Dog. Bark ();
   }
}
```

c Destructor: is a special method or function that is automatically called when an object is no longer in use or is about to be destroyed. The primary purpose of a destructor is to release resources or perform cleanup tasks before an object is removed from memory.

```
class My Class {
public:
   // Constructor
   My Class () {
      // Initialization code here
```

```
  }


  // Destructor
  ~My Class () {
    // Cleanup code here
  }
};
```

D: Polymorphism: allows objects of different types to be treated as objects of a common base type.

```
 class Animal {
      i.      public void make Sound () {
    System. out. Print ln ("Some generic sound");
  }
}


class Dog extends Animal {
  @Override
  public void make Sound () {
    System .out. print ln("Woof!");
  }
}


public class Polymorphism Example {
  public static void main (String [] args) {
    // Polymorphism in action
    Animal my Animal = new Dog () ;  //  Upcasting
    My Anima l. make Sound () ;  // Calls Dog's overridden method
  }
}
```

JOHNY MAUNDU
SCT121-0945/2022

e. class: A class is a blueprint or template that defines the attributes (data members) and methods (functions) common to all objects of a certain kind

```
public class Car {
   // Attributes
   String brand;
   String model;
   int year;

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

   // Behavior
   public void start Engine () {
      System .out .print ln ("Engine started!");
   }

   public static void main (String [] args) {
      // Creating an object of the Car class
      Car my Car = new Car ("Toyota", "Camry", 2022);

      // Accessing attributes and invoking behavior
      System .out. print ln ("Car Details: " + my Car. brand + " " + my Car. model + " " + my Car. year);
      My Car. start Engine ();
   }
}
```

JOHNY MAUNDU
SCT121-0945/2022

F. Inheritance: allows a new class, called a derived or subclass, to inherit attributes and behaviors from an existing class, known as a base or superclass. This relationship between classes promotes code reuse and the creation of a hierarchical structure among classes**.**

```
 // Superclass

class Animal {

   void eat () {

      System. out. Print ln ("Animal is eating");

   }

}


// Subclass inheriting from Animal

class Dog extends Animal {

   void bark () {

      System .out. print ln ("Dog is barking");

   }

}

public class Inheritance Example {

   public static void main (String [] args) {

      // Creating an object of the subclass

      Dog my Dog = new Dog ();

      // Accessing methods from both superclass and subclass

      My Dog. eat (); // Inherited from Animal

      My Dog. bark (); // Specific to Dog

   }

}
```

vii. Three Types of Associations Between Objects:

Aggregation: Represents a "has-a" relationship where one object contains another but allows independent existence.

JOHNY MAUNDU
SCT121-0945/2022

Composition: Signifies a stronger "whole-part" relationship, where the part cannot exist independently of the whole.

Association: Represents a generic relationship between two objects.

viii. Class Diagram:

Definition: A class diagram is a visual representation of the classes and their relationships in a system.

Usage: Used in the analysis and design phases of software development to illustrate the structure of a system.

Steps to Draw Class Diagram:

Identify classes and their attributes.

Define relationships between classes (associations, aggregations, compositions).

Add methods and properties to classes.

Indicate visibility (public, private) and multiplicity of associations.

ix. Area and Perimeter Calculator in C++ Using OOP Concepts:

```
#include <iostream>

#include <cmath>


// Abstract class Shape

class Shape {

public:

   // Pure virtual functions for area and perimeter

   virtual double area () const = 0;

   virtual double perimeter () const = 0;

   virtual ~Shape () {}

};


// Concrete class Circle inheriting from Shape

class Circle: public Shape {

private:
```

JOHNY MAUNDU
SCT121-0945/2022

```
    double radius;

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

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

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

// Concrete class Rectangle inheriting from Shape
class Rectangle: public Shape {
private:
    double length;
    double width;

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

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

    double perimeter () const override {
        return 2 * (length + width);
```

```
    }
};

// Concrete class Triangle inheriting from Shape
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 area () 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 perimeter () const override {
        return side1 + side2 + side3;
    }
};

// Concrete class Square inheriting from Rectangle (Single Inheritance)
class Square: public Rectangle {
public:
    Square (double side) : Rectangle(side, side) {}
};
```

JOHNY MAUNDU
SCT121-0945/2022

```
// Concrete class Compound Shape inheriting from Rectangle and Circle (Multiple Inheritance)
class Compound Shape: public Rectangle, public Circle {
public:
   Compound Shape (double l, double w, double r) : Rectangle(l, w), Circle(r) {}
};

// Concrete class Polygon inheriting from Shape (Hierarchical Inheritance)
class Polygon: public Shape {
   // Implementation of Polygon class goes here...
};

// Friend function for printing details of a shape
void print Details (const Shape& shape) {
   std::cout << "Area: " << shape. Area () << std::endl;
   std::cout << "Perimeter: " << shape. Perimeter () << std::endl;
}

int main () {
   Circle circle (5.0);
   Rectangle rectangle (4.0, 6.0);
   Triangle triangle (3.0, 4.0, 5.0);
   Square square  (4.0);
   Compound Shape compound (2.0, 3.0, 1.5);

   // Friend function usage
   Print Details(circle);
   Print Details(rectangle);
   Print Details(triangle);
```

JOHNY MAUNDU
SCT121-0945/2022

```
    print Details(square);

    print Details(compound);


    return 0;

}
```

Explanation of OOP concepts used:

a. Inheritance:

Single Inheritance: Square inherits from Rectangle.

Multiple Inheritance: Compound Shape inherits from both Rectangle and Circle.

Hierarchical Inheritance: Polygon inherits from Shape, forming a hierarchy.

b. Friend Functions:

The print Details function is a friend function that can access private members of the Shape class.

c. Method Overloading and Method Overriding:

Method Overloading: Constructors in various shapes and the print Details function demonstrate method overloading.

Method Overriding: area and perimeter functions are overridden in each derived class.

d. Late Binding (Dynamic Binding) and Early Binding (Static Binding):

Late Binding: Virtual functions in the Shape class enable late binding, allowing the correct function to be called at runtime.

Early Binding: Non-virtual functions are bound at compile time.

e. Abstract Class and Pure Functions:

Shape is an abstract class with pure virtual functions (area and perimeter), making it impossible to instantiate. Subclasses provide concrete implementations for these functions.

JOHNY MAUNDU
SCT121-0945/2022

Viii a. Function overloading and operator overloading:

Function Overloading:

Function overloading refers to the ability to define multiple functions with the same name but with different parameter lists.

Example:

```
#include<iostream>

void display (int num) {
   std::cout << "Integer: " << num << std::endl;
}

void display(double num) {
   std::cout << "Double: " << num << std::endl;
}

int main() {
   display(5);
   display(3.14);

  return 0;
}
```

Operator Overloading:

Operator overloading allows you to define how operators behave for user-defined types.

Example:

JOHNY MAUNDU
SCT121-0945/2022

```
#include<iostream>


class Complex {

private:

    double real;

    double imaginary;


public:

    Complex () : real(0), imaginary(0) {}


    Complex operator + (const Complex& other) {

        Complex result;

        result.real = this->real + other.real;

        result.imaginary = this->imaginary + other.imaginary;

        return result;

    }


    void display() {

        std::cout << "Real: " << real << ", Imaginary: " << imaginary << std::endl;

    }

};


int main() {

    Complex c1, c2, result;

    // Assume values are assigned to c1 and c2

    result = c1 + c2;

    result. display();


    return 0;
```

JOHNY MAUNDU
SCT121-0945/2022

}

In the + operator is overloaded for the Complex class.

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. This means that any modifications made to the parameter inside the function do not affect the original variable.

```
#include<iostream>


void square (int num) {
   num = num * num;
}


int main() {
   int x = 5;
   square(x);
   std::cout << "Original value: " << x << std::endl; // Output: Original value: 5


   return 0;
}
```

Pass by Reference:

Passing by reference involves passing the memory address (reference) of a variable to a function. This allows the function to modify the original variable directly.

Example:

```
#include<iostream>
```

JOHNY MAUNDU
SCT121-0945/2022

```
void square (int &num) {
   num = num * num;
}


int main() {
   int x = 5;
   square(x);
   std::cout << "Modified value: " << x << std::endl; // Output: Modified value: 25


   return 0;
}
```

c. Parameters and Arguments:

Parameters:

Parameters are the variables declared in the function signature. They act as placeholders for the values that will be passed to the function when it is called.

Example:

```
#include<iostream>


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


int main() {
   int x = 10, y = 20;
   add (x, y);
```

```
    return 0;
}
```

a and b are parameters in the add function.

Arguments:

Arguments are the actual values passed to the function when it is called. They are the concrete values that are substituted for the parameters in the function call.

Example:

```
#include<iostream>

void display(int num) {
    std::cout << "Value: " << num << std::endl;
}

int main() {
    int x = 42;
    display(x);

    return 0;
}
```

6.

```
public class Calculate G {

    // Constants for Earth's gravity and falling time
    static double gravity = -9.81;
    static double falling Time = 30;
    static double initial Velocity = 0.0;
    static double initial Position = 0.0;
```

JOHNY MAUNDU
SCT121-0945/2022

```
   // Method to compute position using the formula: x(t) = 0.5 * a * t^2 + v i * t + x i

   public static double calculate Position (double acceleration, double time, double initial Velocity,
double initial Position) {

      return 0.5 * acceleration * Math. Pow (time, 2) + initial Velocity * time + initial Position;

   }


   // Method to compute velocity using the formula: v(t) = a * t + v_i

   public static double calculate Velocity (double acceleration, double time, double initial Velocity) {

      return acceleration * time + initial Velocity;

   }


   // Method for multiplication

   public static double multiply (double a, double b) {

      return a * b;

   }


   // Method for powering to square

   public static double power Square (double a) {

      return Math. Pow (a, 2);

   }


   // Method for summation

   public static double sum (double a, double b) {

      return a + b;

   }


   // Method for printing out a result

   public static void outline (String message, double result) {
```

JOHNY MAUNDU
SCT121-0945/2022

```
      System. out. Print ln (message + result);
   }

   public static void main(String[] args) {

      // Compute position and velocity
      double final Position = calculate Position (gravity, falling Time, initial Velocity, initial Position);
      double final Velocity = calculate Velocity (gravity, falling Time, initial Velocity);

      // Print out the results
      Outline ("The object's position after " + falling Time + " seconds is ", final Position + " m.");
      outline ("The object's velocity after " + falling Time + " seconds is ", final Velocity + " m/s.");
   }
}
```

Part B:

1.

```
#include <iostream>

// Function to find the sum of even-valued terms in the Fibonacci sequence
long long sum Even Fibonacci (int limit) {
   long long a = 1, b = 2, temp, sum = 0;

   while (b <= limit) {
      // Check if the current term is even
      if (b % 2 == 0) {
         sum += b;
      }

      // Generate the next Fibonacci term
```

JOHNY MAUNDU
SCT121-0945/2022

```
        temp = a + b;

        a = b;

        b = temp;

    }


    return sum;

}


int main () {

    // Define the limit (four million in this case)

    int limit = 4000000;


    // Call the function to find the sum of even-valued Fibonacci terms

    long long result = sum Even Fibonacci(limit);


    // Output the result

    std::cout << "The sum of even-valued terms in the Fibonacci sequence not exceeding "

              << limit << " is: " << result << std::
```

Question two

2

```
#include <Q Application>

#include <Q Widget>

#include <Q Line Edit>

#include <Q Push Button>

#include <Q Label>

#include <Q String>

#include <algorithm>
```

JOHNY MAUNDU
SCT121-0945/2022

```
class Palindrome Checker: public Q Widget {
   Q_OBJECT


public:
   Palindrome Checker (Q Widget *parent = nullptr): Q Widget(parent) {
      // Set up user interface elements
      Number Input = new Q Line Edit(this);
      Check Button = new Q Push Button ("Check Palindrome", this);
      Result Label = new Q Label ("", this);


      // Set up layout
      QV Box Layout *layout = new QV Box Layout(this);
      layout->add Widget (number Input);
      layout->add Widget (check Button);
      layout->add Widget (result Label);


      // Connect button click to check Palindrome function
      Connect (check Button, &Q Push Button:: clicked, this, &Palindrome Checker::check
Palindrome);
   }


private slots:
   void check Palindrome () {
      // Get the entered number as a Q String
      Q String input String = number Input->text();


      // Convert the Q String to a standard string
```

JOHNY MAUNDU
SCT121-0945/2022

```
        std::string input Std String = input String.to Std String();


        // Reverse the string
        std::string reversed Std String = input Std String;
        std::reverse(reversed Std String. begin (), reversed Std String. End ());


        // Check if the original and reversed strings are equal
        if (input Std String == reversed Std String) {
            result Label->set Text("Palindrome!");
        } else {
            Result Label->set Text ("Not a Palindrome!");
        }
    }


private:
    Q Line Edit *number Input;
    Q Push Button *check Button;
    Q Label *result Label;
};


int main (int argc, char *argv []) {
    Q Application app (argc, argv);


    Palindrome Checker window;
    window. set Window Title ("Palindrome Checker");
    window. Resize (300, 150);
    window. Show ();
```

JOHNY MAUNDU
SCT121-0945/2022

```
   return app. Exec ();
}

#include "main.moc"
```

Question three

```
#include <iostream>

int main() {
   const int SIZE = 15;

   // Part a: Take 15 values as input from the user and store in an array
   int original Array[SIZE];

   std::cout << "Enter 15 integer values:" << std::endl;
   for (int i = 0; i < SIZE; ++i) {
      std::cout << "Enter value #" << (i + 1) << ": ";
      std::cin >> original Array[i];
   }

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

   // Ask the user to enter a number
```

JOHNY MAUNDU
SCT121-0945/2022

```
int search Number;

std::cout << "\nEnter a number to search in the array: ";

std::cin >> search Number;


// Check if the number is present in the array

bool number Found = false;

int found Index = -1;


for (int i = 0; i < SIZE; ++i) {

   if (original Array[i] == search Number) {

      number Found = true;

      found Index = i;

      break;

   }

}


// Print the result

if (number Found) {

   std::cout << "The number found at index " << found Index << std::endl;

} else {

   std::cout << "Number not found in this array." << std::endl;

}


// Part c: Create another array and copy elements in reverse order

int reversed Array[SIZE];


for (int i = 0; i < SIZE; ++i) {
```

```
    reversed Array[i] = original Array [SIZE - 1 - i];
}

// Print elements of the new array
std::cout << "\nValues in the new array (reversed):" << std::endl;
for (int i = 0; i < SIZE; ++i) {
    std::cout << reversed Array[i] << " ";
}

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

for (int i = 0; i < SIZE; ++i) {
    sum += original Array[i];
    product *= original Array[i];
}

// Print sum and product
std::cout << "\
```