**YVETTE WAIRIMU NJUGUNA**

**SCT221-0963/2022**

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

Oobject Modelling Techniques (OMT) is an approach for visualizing and designing software systems using object-oriented concepts. It was developed by James Rumbaugh in the early 1990s. OMT comprises several modelling techniques and strategies aimed at effectively representing a system's structure and behaviour.

**Object Modelling Techniques (OMT) Components:**

1. **Object Modelling:** This involves identifying and describing objects within the system. Objects encapsulate both data (attributes) and behaviours (methods). Objects interact with each other by exchanging messages.
2. **Dynamic Modelling:** Dynamic modelling focuses on representing the behaviour of objects and their interactions over time. It includes state diagrams, activity diagrams, and event diagrams to depict the system's dynamic nature.
3. **Functional Modelling:** Functional modelling concentrates on system functionality. It involves creating data flow diagrams, process specifications, and logical data models to define the system's functions and their relationships.
4. **Object-Oriented Design:** OMT emphasizes the design phase where classes, relationships between classes, inheritance hierarchies, and collaborations are defined in detail.

OMT has been instrumental in software engineering, particularly in:

* **Software Development:** It assists in the analysis, design, and development of complex software systems, aiding in understanding system architecture and processes.
* **System Maintenance and Enhancement:** Om’s structured approach makes it easier to maintain and modify existing systems by providing a clear blueprint of the system's structure and behaviour.
* **Communication among Stakeholders:** OMT's graphical representation enables effective communication between stakeholders involved in the software development process.

"Object-Oriented Modelling and Design" by James Rumbaugh: This book is one of the primary sources where James Rumbaugh introduced and elaborated on OMT.

"The Unified Modelling Language Reference Manual" by James Rumbaugh, Ivar Jacobson, and Grady Booch: This book provides comprehensive information on the Unified Modelling Language (UML), which evolved from OMT and other methodologies.

**2 Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP). [2 Marks]**

**Object-Oriented Analysis and Design (OOAD):**

OOAD is a methodology used in software development that emphasizes the use of objects and their interactions to design a system. It comprises several stages:

1. **Analysis Phase:** In this phase, the focus is on understanding the problem domain. It involves gathering requirements, identifying objects, defining their relationships, and modelling the system's behaviour and structure using tools like UML diagrams (use case diagrams, class diagrams, sequence diagrams).
2. **Design Phase:** Once the requirements are analysed, the design phase begins, translating the analysis into a detailed design. It involves defining classes, interfaces, inheritance relationships, and collaborations between objects. Design patterns and architectural decisions are also considered in this phase.
3. **Implementation Phase:** This phase involves actual coding based on the design created during the previous phases. Object-oriented programming languages like Java, C++, or Python are used to implement the designed system.

**Object Analysis and Design (OOP):**

OOP, on the other hand, is a programming paradigm or methodology used in writing computer programs. It's more focused on the programming aspects than the entire software development process. OOP involves the following concepts:

1. **Objects and Classes:** Objects are instances of classes that encapsulate data (attributes) and behaviours (methods). Classes define the blueprint for creating objects.
2. **Inheritance:** It allows one class (subclass or derived class) to inherit properties and behaviour from another class (superclass or base class), promoting code reusability.
3. **Polymorphism:** It refers to the ability of objects to take on different forms. It includes method overriding (inherited methods being redefined in the subclass) and method overloading (multiple methods with the same name but different parameters).
4. **Encapsulation:** It refers to bundling data (attributes) and methods that operate on the data within a single unit (class), preventing direct access to the data from outside the class and ensuring data integrity.

**Comparison:**

* **OOAD vs. OOP Focus:** OOAD concentrates on the entire software development process from analysis to design to implementation. OOP focuses primarily on the programming aspect and the implementation phase.
* **Scope:** OOAD covers a broader spectrum, including requirements gathering, system modelling, architectural design, and implementation. OOP is specifically about programming and implementing solutions based on the designed architecture.
* **Abstraction Level:** OOAD operates at a higher level of abstraction, focusing on system architecture and design. OOP deals with lower-level programming constructs and implementing the designed solution.

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

**Goals of Unified Modelling Language (UML):**

1. **Standardized Modelling Notation:**
   * Goal: UML aims to provide a standardized and uniform way to visualize the design and structure of software systems.
   * Purpose: It offers a common language and notation that software developers, stakeholders, and team members can use to understand, communicate, and document various aspects of a system's architecture and behaviour.
2. **Specification and Documentation:**
   * Goal: UML intends to specify, construct, visualize, and document artifacts of a software system.
   * Purpose: UML diagrams and models serve as documentation that captures the system's architecture, design decisions, requirements, and interactions among components. These visual representations facilitate communication among stakeholders and assist in system understanding.
3. **System Behaviour and Structure Representation:**
   * Goal: UML is designed to capture both the static (structure) and dynamic (behaviour) aspects of a system.
   * Purpose: It enables the modelling of various system elements such as classes, objects, relationships, activities, interactions, and states. This comprehensive representation helps in understanding how the system works, its components, and their interactions over time.
4. **Tool for Analysis, Design, and Implementation:**
   * Goal: UML provides a set of diagrams and tools that aid in the analysis, design, and implementation of software systems.
   * Purpose: It assists software developers in conceptualizing, visualizing, and specifying the structure and behaviour of a system before implementing it. UML diagrams such as class diagrams, sequence diagrams, use case diagrams, etc., help in modelling different aspects of the system at various stages of development.
5. **Facilitating Communication and Collaboration:**
   * Goal: UML fosters effective communication and collaboration among stakeholders involved in the software development process.
   * Purpose: UML diagrams act as a common ground for discussing and conveying ideas among team members, customers, designers, and other stakeholders. It bridges the gap between technical and non-technical individuals by providing a visual representation of complex system designs.

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

1 **Encapsulation for Data Protection and Security:**

Explanation: Encapsulation refers to bundling data (attributes) and methods (behaviors) within objects, controlling access to data through well-defined interfaces (public, private, protected). It hides the internal implementation details of an object, providing data security and preventing unauthorized access or modification.

References:

Lafore, Robert. "Object-Oriented Programming in C++." Sams Publishing, 2001.

Eckel, Bruce. "Thinking in Java." Prentice Hall, 2006.

2 **Reusability through Inheritance and Polymorphism:**

Explanation: Inheritance allows the creation of new classes (subclasses) by inheriting properties and behaviors from existing classes (super classes). Polymorphism enables objects to take multiple forms, facilitating code reuse and flexibility in implementations.

References:

Gamma, Erich et al. "Design Patterns: Elements of Reusable Object-Oriented Software." Addison-Wesley, 1994.

Freeman, Eric et al. "Head First Design Patterns." O'Reilly Media, 2004.

3 Modularity **for Maintenance and Scalability:**

Explanation: OOP promotes modularity by breaking systems into smaller, manageable modules or classes. Modifications or enhancements to specific classes can be made without affecting other parts of the system, enhancing maintainability and scalability.

References:

Martin, Robert C. "Clean Code: A Handbook of Agile Software Craftsmanship." Prentice Hall, 2008.

Larman, Craig. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development." Prentice Hall, 2004.

**5 Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of the term. [10 Marks]**

**Constructor method**

* A constructor is a special method in a class used to initialize objects. It has the same name as the class and doesn't have a return type. Constructors are called automatically when an object is created.

public class Car {

private String brand;

// Constructor

public Car (String brand) {

this.brand = brand; }

public String getBrand() {

return brand; }

public static void main(String[] args) {

Car myCar = new Car("Toyota"); // Creating an object using constructor

System.out.println("Brand: " + myCar.getBrand()); }}

**object**

An object is an instance of a class. It's a runtime entity with state (attributes) and behavior (methods) defined by its class. Objects represent real-world entities and are created from classes.

public class Person {

private String name;

// Constructor

public Person(String name) {

this.name = name; }

public void greet() {

System.out.println("Hello, I am " + name); }

public static void main(String[] args) {

Person person1 = new Person("Alice"); // Creating an object 'person1'

person1.greet(); // Calling method 'greet' on object 'person1' }}

**interface**

* An interface in Java is a reference type similar to a class containing only abstract methods, constants, and default methods. It defines a contract that classes must implement.

// Interface

public interface Shape {

double calculateArea(); // Abstract method

void display(); // Another abstract method}

// Implementing interface

public class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius; }

@Override

public double calculateArea() {

return Math.PI \* radius \* radius; }

@Override

public void display() {

System.out.println("Area of circle: " + calculateArea()); }

public static void main(String[] args) {

Circle circle = new Circle(5);

circle.display(); }}

**polymorphism**

* Polymorphism refers to the ability of a method to take different forms in different classes. It allows objects of different classes to be treated as objects of a common superclass.

public class Animal {

public void makeSound() {

System.out.println("Some sound"); }}

public class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Bark!"); }}

public class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("Meow!"); }}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

Animal cat = new Cat();

dog.makeSound(); // Output: "Bark!"

cat.makeSound(); // Output: "Meow!" }}

**class**

* A class is a blueprint or template for creating objects. It defines the structure, behavior, and properties that objects of the class will have.

public class Rectangle {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width; }

public double calculateArea() {

return length \* width; }

public static void main(String[] args) {

Rectangle rect = new Rectangle(4, 5);

double area = rect.calculateArea();

System.out.println("Area of rectangle: " + area); }}

**6 *EXPLAIN* the three types of associations (relationships) between objects in object oriented. [6 Marks]**

1. **Association:**
   * Association represents a relationship between two or more objects where each object has its own lifecycle and there are no ownership or containment relationships among them. It's a general-purpose relationship between objects that can be one-to-one, one-to-many, or many-to-many.
   * For example, consider two classes: Car and Driver. They are associated because a driver can drive a car, but the car doesn’t own the driver and the driver can drive multiple cars.
2. **Aggregation:**
   * Aggregation is a specific type of association where one object (the whole) is composed of one or more other objects (parts), but the parts can exist independently of the whole.
   * It represents a "has-a" relationship. The part objects can belong to more than one whole object and can exist independently.
   * For instance, a University can have multiple Departments. Departments are parts of the University, but they can exist on their own.
3. **Composition:**
   * Composition is a stronger form of aggregation where the parts are exclusive to one whole object. The part objects are created and destroyed along with the whole object. It represents a "contains-a" relationship.
   * In a composition relationship, the lifetime of the part object is dependent on the lifetime of the whole object.
   * An example could be a House and its Rooms. Rooms are part of the house, and when the house is destroyed, the rooms cease to exist.

**7 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.**

A class diagram is a graphical representation of the structure and relationships of classes in object-oriented programming (OOP). It's part of the Unified Modeling Language (UML) and depicts the various classes in a system, their attributes, methods, and associations with other classes.

**Uses of Class Diagrams:**

* **Designing Systems:** Class diagrams help in designing the structure of software systems by defining classes and their relationships.
* **Communication Tool:** They aid in communicating the architecture and design of a system among stakeholders.
* **Code Generation:** They can serve as a basis for generating code in object-oriented programming languages like Java.

**Steps to Draw a Class Diagram in Java:**

Let's create a simple example of a banking system with classes representing **Bank**, **Account**, and **Customer**.

Step 1: Identify Classes

1. **Bank**: Represents the bank entity.
2. **Account**: Represents a bank account.
3. **Customer**: Represents a customer who holds one or multiple accounts.

Step 2: Add Attributes and Methods

* **Bank:**
  + Attributes: name, address
  + Methods: addAccount(), removeAccount()
* **Account:**
  + Attributes: accountNumber, balance
  + Methods: deposit(), withdraw(), getBalance()
* **Customer:**
  + Attributes: name, customerId
  + Methods: addAccount(), removeAccount(), getAccounts()

Step 3: Define Relationships

* **Bank** has an association with multiple **Account** instances (one-to-many).
* **Customer** has an association with multiple **Account** instances (one-to-many).

// Bank class

class Bank {

private String name;

private String address;

public void addAccount(Account account) {

// Implementation }

public void removeAccount(Account account) {

// Implementation }}

// Account class

class Account {

private int accountNumber;

private double balance;

public void deposit(double amount) {

// Implementation }

public void withdraw(double amount) {

// Implementation }

public double getBalance() {

// Implementation

return balance; }}

// Customer class

class Customer {

private String name;

private int customerId;

private List<Account> accounts;

public void addAccount(Account account) {

// Implementation }

public void removeAccount(Account account) {

// Implementation }

public List<Account> getAccounts() {

// Implementation

return accounts; }}

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

**class *CalculateG* {  
public static void main(String[] arguments){**

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

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

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

**// Add the formulas for position and velocity**

**System.out.println("The object's position after " + fallingTime + " seconds is "**

**+ finalPosition + " m.");**

**// Add output line for velocity (similar to position)} }**

class CalculateG {

public static void main(String[] arguments){

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;

// Add the formulas for position and velocity

finalPosition = 0.5 \* gravity \* Math.pow(fallingTime, 2) + initialVelocity \* fallingTime + initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

System.out.println("The object's position after " + fallingTime + " seconds is " + finalPosition + " m.");

System.out.println("The object's final velocity is " + finalVelocity + " m/s."); }}

**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∗𝑎𝑡2 +𝑣𝑖𝑡+𝑥𝑖 𝑣(𝑡)=𝑎𝑡+𝑣𝑖**

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

**public class *CalculateG* {**

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

**}**

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

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

**}  
public static void main(String[] args) {**

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

**result**

**} }**

public class CalculateG {

public static void main(String[] args) {

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

double fallingTime = 30;

double initialVelocity = 0.0;

double finalVelocity;

double initialPosition = 0.0;

double finalPosition;

// Calculate position and velocity

finalPosition = 0.5 \* gravity \* Math.pow(fallingTime, 2) + initialVelocity \* fallingTime + initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

System.out.println("The object's position after " + fallingTime + " seconds is " + finalPosition + " meters.");

System.out.println("The object's final velocity is " + finalVelocity + " m/s."); }}

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

public class CalculateG {

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

return a \* b; }

public static double powerToSquare(double a) {

return Math.pow(a, 2); }

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

return a + b; }

public static void printResult(String message, double value) {

System.out.println(message + value + " meters."); }

public static void main(String[] args) {

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

double fallingTime = 30;

double initialVelocity = 0.0;

double finalVelocity;

double initialPosition = 0.0;

double finalPosition;

// Calculate position and velocity

finalPosition = 0.5 \* multiply(gravity, powerToSquare(fallingTime)) + multiply(initialVelocity, fallingTime) + initialPosition;

finalVelocity = multiply(gravity, fallingTime) + initialVelocity;

printResult("The object's position after " + fallingTime + " seconds is ", finalPosition);

printResult("The object's final velocity is ", finalVelocity); }}

Part B

**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 Java method to find the sum of all the even- valued terms.**

public class FibonacciSum {

public static int sumEvenFibonacci(int limit) {

int sum = 0;

int first = 1;

int second = 2;

while (second <= limit) {

if (second % 2 == 0) {

sum += second; }

int next = first + second;

first = second;

second = next; }

return sum; }

public static void main(String[] args) {

int limit = 4000000;

int result = sumEvenFibonacci(limit);

System.out.println("The sum of even-valued terms in the Fibonacci sequence up to " + limit + " is: " + result); }}

**2.A palindrome number is a number that remain the same when read from behind or front ( a number that is equal to reverse of number) for example, 353 is palindrome because reverse of 353 is 353 (you see the number remains the same). But a number like 591 is not palindrome because reverse of 591 is 195 which is not equal to 591. Write Java program to check if a number entered by the user is palindrome or not. You should provide the user with a GUI interface to enter the number and display the results on the same interface.**

The interface:

**Check if a number is palindrome**

345

Enter the number

Not palindrome

Output 🡪

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

public class PalindromeCheckerGUI extends JFrame implements ActionListener {

private JTextField numberField;

private JButton checkButton;

private JLabel resultLabel;

public PalindromeCheckerGUI() {

setTitle("Palindrome Checker");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(300, 150);

setLocationRelativeTo(null);

JPanel panel = new JPanel();

panel.setLayout(new GridLayout(3, 1));

JLabel numberInputLabel = new JLabel("Enter a number:");

numberField = new JTextField();

checkButton = new JButton("Check Palindrome");

checkButton.addActionListener(this);

resultLabel = new JLabel();

panel.add(numberInputLabel);

panel.add(numberField);

panel.add(checkButton);

panel.add(resultLabel);

add(panel);

setVisible(true); }

public void actionPerformed(ActionEvent e) {

if (e.getSource() == checkButton) {

String numberStr = numberField.getText();

if (isPalindrome(numberStr)) {

resultLabel.setText(numberStr + " is a palindrome.");

} else {

resultLabel.setText(numberStr + " is not a palindrome."); } } }

private boolean isPalindrome(String str) {

int left = 0;

int right = str.length() - 1;

while (left < right) {

if (str.charAt(left) != str.charAt(right)) {

return false; }

left++;

right--; }

return true; }

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

new PalindromeCheckerGUI(); }); }}

**Question three: [15 marks]**

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

**Print the values stored in the array on screen.**

**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”**

**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**

**Get the sum and product of all elements of your array. Print product and the sum each on its own line.**

import java.util.Scanner;

public class ArrayOperations {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Taking 15 integer values as input and storing them in an array

int[] arr = new int[15];

System.out.println("Enter 15 integer values:");

for (int i = 0; i < 15; i++) {

arr[i] = scanner.nextInt(); }

// Printing the values stored in the array

System.out.println("Values stored in the array:");

for (int value : arr) {

System.out.print(value + " "); }

System.out.println();

// Asking user to enter a number to check its presence in the array

System.out.println("Enter a number to check its presence in the array:");

int numberToFind = scanner.nextInt();

boolean found = false;

int index = -1;

for (int i = 0; i < arr.length; i++) {

if (arr[i] == numberToFind) {

found = true;

index = i;

break; } }

if (found) {

System.out.println("The number found at index " + index);

} else {

System.out.println("Number not found in this array"); }

// Creating a new array with elements in reverse order

int[] reverseArray = new int[arr.length];

for (int i = 0; i < arr.length; i++) {

reverseArray[i] = arr[arr.length - 1 - i]; }

// Printing the elements of the new array in reverse order

System.out.println("Elements of the new array in reverse order:");

for (int value : reverseArray) {

System.out.print(value + " "); }

System.out.println();

// Calculating sum and product of elements in the array

int sum = 0;

int product = 1;

for (int value : arr) {

sum += value;

product \*= value; }

// Printing the sum and product of the elements

System.out.println("Sum of all elements in the array: " + sum);

System.out.println("Product of all elements in the array: " + product);

scanner.close(); }}