**DIT: OOP Assignment**

Student’s Name

**Dut Achol Deng-SCT121-0853/2022**

Institutional Affiliation

**Jomo Kenyatta University of Agriculture and Technology**

Instructor’s Name

**John Kimotho**

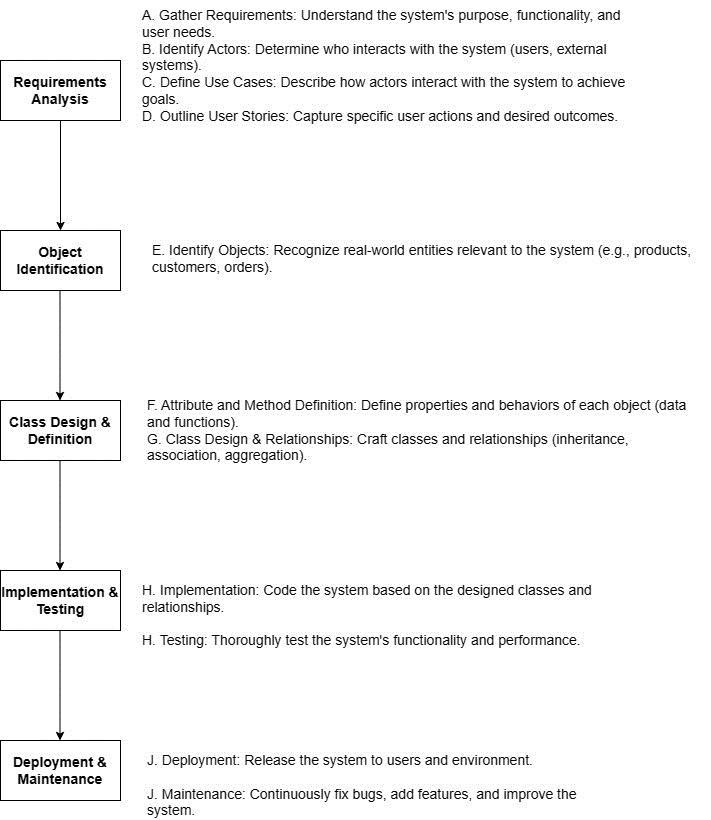
Course

**Object Oriented Programming**

Date

**14-12-2023**

**Part A: Instructions for part A: answer all the Questions in this section.**

1. **Using a well labeled diagram, explain the steps of creating a system using OOP principles. [4 Marks]**
2. **What is the Object Modeling Techniques (OMT). [1 Marks]**

Object Modeling Techniques (OMT) is a method for modeling and designing object-oriented systems.

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

Object-Oriented Analysis and Design (OOAD) is a comprehensive methodology covering system analysis and design, while Object-Oriented Programming (OOP) is the coding phase focusing on implementing software using objects.

Object-Oriented Analysis and Design (OOAD) involves the entire software development lifecycle, while Object-Oriented Programming (OOP) is a programming paradigm specifically addressing the implementation phase through object-centric coding

**Discuss Main goals of UML. [2 Marks]**

1. **Standardization:**

UML aims to provide a standardized modeling language for visualizing, specifying, constructing, and documenting software systems.

1. **Communication:**

Facilitates effective communication among stakeholders by offering a common visual representation of system architecture, fostering better understanding and collaboration.

Top of Form

1. **DESCRIBE three advantages of using object oriented to develop an information system. [3Marks]**
2. **Modularity:** Object-oriented development enables modular design, enhancing code organization and maintenance.
3. **Reusability:** Promotes code reuse through class inheritance and polymorphism, reducing development time.
4. **Encapsulation:** Hides internal details, enhancing security and facilitating easier updates without affecting other parts of the system
5. **Briefly explain the following terms as used in object-oriented programming. Write a sample C++ code to illustrate the implementation of each concept. [12 Marks]**
   1. **Constructor**
   2. **object**
   3. **Destructor**
   4. **polymorphism**
   5. **class**
   6. **Inheritance**

**a. Constructor:**

A constructor is a special member function in a class that is automatically called when an object is created. It initializes the object's state.

#include <iostream>

class MyClass {

public:

// Constructor

MyClass(int initialValue) {

value = initialValue;

std::cout << "Constructor called with value: " << value << std::endl;

}

private:

int value;

};

int main() {

// Creating an object of MyClass

MyClass myObject(10);

return 0;

}

**b. Object:**

An object is an instance of a class. It encapsulates data and behavior defined by the class.

#include <iostream>

class MyClass {

public:

void display() {

std::cout << "Hello from MyClass" << std::endl;

}

};

int main() {

// Creating an object of MyClass

MyClass myObject;

myObject.display();

return 0;

}

**c. Destructor:**

In C++, a destructor is a special member function with the same name as the class but preceded by a tilde (~). It is called automatically when an object goes out of scope to free the memory space.

#include <iostream>

class MyClass {

public:

// Destructor

~MyClass() {

std::cout << "Destructor called" << std::endl;

}

};

int main() {

// Object goes out of scope, destructor called

MyClass myObject;

return 0;

}

**d. Polymorphism**:

Polymorphism allows objects of different types to be treated as objects of a common type. It can be achieved through method overriding.

#include <iostream>

class Animal {

public:

virtual void makeSound() {

std::cout << "Some generic sound" << std::endl;

}

};

class Dog : public Animal {

public:

void makeSound() override {

std::cout << "Woof! Woof!" << std::endl;

}

};

int main() {

// Using polymorphism

Animal\* myDog = new Dog();

myDog->makeSound(); // Output: Woof! Woof!

delete myDog;

return 0;

}

**e. Class:**

A class is a blueprint for creating objects. It defines attributes (data) and methods (functions) that the objects will have.

#include <iostream>

class Car {

public:

// Methods

void start() {

std::cout << "Engine started." << std::endl;

}

};

int main() {

// Creating an object of Car

Car myCar;

myCar.start();

return 0;

}

**f. Inheritance:**

Inheritance allows a class to inherit attributes and behaviors from another class. It promotes code reuse and supports the creation of a class hierarchy.

#include <iostream>

class Vehicle {

public:

// Methods

void start() {

std::cout << "Vehicle started." << std::endl;

}

};

class Car : public Vehicle {

// Additional fields or methods specific to Car

};

int main() {

// Using inheritance

Car myCar;

myCar.start(); // Output: Vehicle started.

return 0;

}

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

**1. Association**:

- Represents a generic relationship between objects. Objects are loosely connected and can exist independently.

**2. Aggregation:**

- Denotes a "whole-part" relationship where objects are associated, but one object (whole) can exist without the other (part).

**3. Composition:**

- Stronger than aggregation, indicating a "whole-part" relationship where the part has no independent existence and is solely owned by the whole.

**Vii. 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 visual representation in UML depicting classes, attributes, methods, and relationships within a system. It is used for system modeling and design.

Steps to draw:

1. Identify classes.

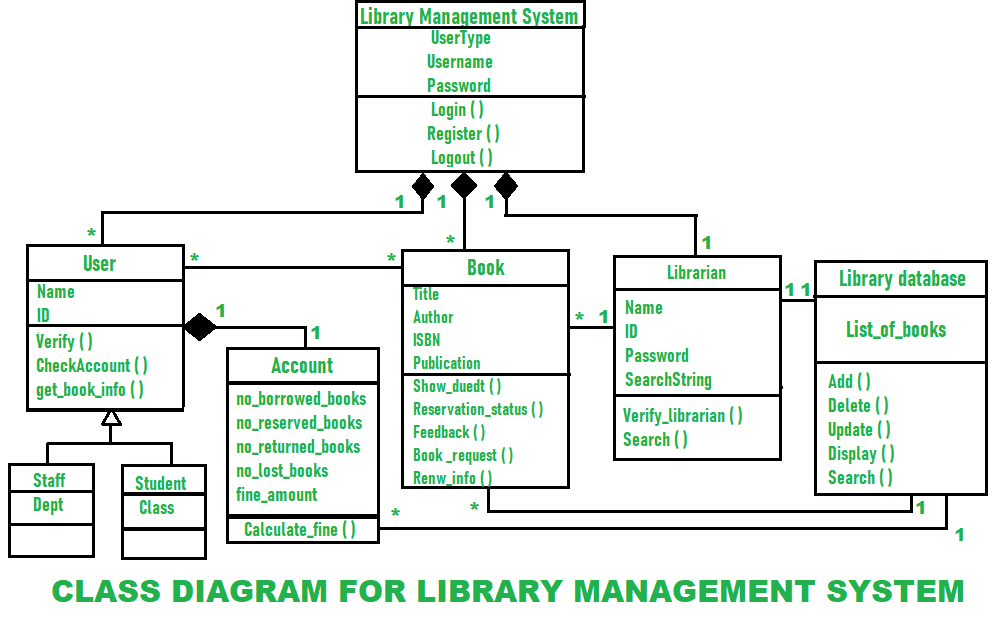
2. Specify attributes and methods.

3. Define relationships (association, aggregation, composition).

4. Add multiplicities.

5. Include additional details (visibility, types).

Example: Consider a Library Management System with classes like Book, User, and Librarian, Stuff, Student, Account, and Library Database illustrating their attributes and associations.



1. **Given that you are creating area and perimeter calculator using C++, to compute 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.**

**The code**

#include <iostream>

#include <cmath>

// Abstract class Shape

class Shape {

public:

// Pure virtual functions

virtual double area() const = 0;

virtual double perimeter() const = 0;

virtual ~Shape() {}

};

// Circle class derived from Shape

class Circle : public Shape {

private:

double radius;

public:

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

// Overridden functions

double area() const override {

return 3.14159265358979323846 \* radius \* radius;

}

double perimeter() const override {

return 2 \* 3.14159265358979323846 \* radius;

}

};

// Rectangle class derived from Shape

class Rectangle : public Shape {

private:

double length;

double width;

public:

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

// Overridden functions

double area() const override {

return length \* width;

}

double perimeter() const override {

return 2 \* (length + width);

}

};

// Triangle class derived 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) {}

// Overridden functions

double area() const override {

// Using Heron's formula

double s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

double perimeter() const override {

return side1 + side2 + side3;

}

};

// Square class derived from Rectangle (Single Inheritance)

class Square : public Rectangle {

public:

Square(double side) : Rectangle(side, side) {}

};

// Multiple Inheritance

class Cuboid : public Rectangle, public Shape {

private:

double height;

public:

Cuboid(double l, double w, double h) : Rectangle(l, w), height(h) {}

// Overridden function

double area() const override {

return 2 \* (length \* width + length \* height + width \* height);

}

// Perimeter is not applicable for a 3D shape, so it's not implemented.

};

// Hierarchical Inheritance

class EquilateralTriangle : public Triangle {

public:

EquilateralTriangle(double side) : Triangle(side, side, side) {}

};

// Friend function for displaying details

void displayDetails(const Shape& shape) {

std::cout << "Area: " << shape.area() << std::endl;

std::cout << "Perimeter: " << shape.perimeter() << std::endl;

}

int main() {

// Example usage

Circle circle(5);

Rectangle rectangle(4, 6);

Triangle triangle(3, 4, 5);

Square square(4);

Cuboid cuboid(3, 4, 5);

EquilateralTriangle eqTriangle(5);

// Displaying details using friend function

std::cout << "Circle details:" << std::endl;

displayDetails(circle);

std::cout << "\nRectangle details:" << std::endl;

displayDetails(rectangle);

std::cout << "\nTriangle details:" << std::endl;

displayDetails(triangle);

std::cout << "\nSquare details:" << std::endl;

displayDetails(square);

std::cout << "\nCuboid details:" << std::endl;

displayDetails(cuboid);

std::cout << "\nEquilateral Triangle details:" << std::endl;

displayDetails(eqTriangle);

return 0;

}

**Explanation of OOP Concepts Used**:

* 1. **Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance) [10 Marks]**

Single Inheritance: Square is derived from Rectangle.

Multiple Inheritance: Cuboid inherits from both Rectangle and Shape.

Hierarchical Inheritance: EquilateralTriangle and Triangle both inherit from Shape.

* 1. **Friend functions [5 Marks]**

displayDetails is a friend function that can access the private members of the Shape class.

* 1. **Method overloading and method overriding [10 Marks]**

Area and perimeter functions are overloaded in different derived classes.

They are overridden in the respective derived classes to provide specific implementations.

* 1. **Late binding and early binding [8 Marks]**

Late Binding (Dynamic Binding): Achieved through virtual functions (area and perimeter) and the use of pointers/references to the base class (Shape).

Early Binding (Static Binding): Occurs when the object is known at compile time.

* 1. **Abstract class and pure functions [6 Marks]**

Shape is an abstract class with pure virtual functions (area and perimeter), making it an abstract class. The derived classes must implement these functions.

1. **Using a program written in C++, differentiate between the following. [6 Marks]**
   1. **Function overloading and operator overloading**
   2. **Pass by value and pass by reference**
   3. **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 {

private:

double real;

double imag;

public:

Complex(double r, double i) : real(r), imag(i) {}

// Overloading the + operator

Complex operator+(const Complex& other) {

return Complex(real + other.real, imag + other.imag);

}

// Display function

void display() {

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

}

};

// Pass by Value

void incrementValue(int x) {

x++;

}

// Pass by Reference

void incrementReference(int& x) {

x++;

}

int main() {

// Function Overloading

std::cout << "Function Overloading:" << std::endl;

std::cout << "Adding integers: " << add(3, 4) << std::endl;

std::cout << "Adding doubles: " << add(3.5, 4.2) << std::endl;

// Operator Overloading

std::cout << "\nOperator Overloading:" << std::endl;

Complex c1(2.0, 3.0);

Complex c2(1.5, 2.5);

Complex result = c1 + c2;

std::cout << "Result of addition: ";

result.display();

// Pass by Value

std::cout << "\nPass by Value:" << std::endl;

int value = 5;

incrementValue(value);

std::cout << "Original value: 5, Incremented value (pass by value): " << value << std::endl;

// Pass by Reference

std::cout << "\nPass by Reference:" << std::endl;

int reference = 5;

incrementReference(reference);

std::cout << "Original value: 5, Incremented value (pass by reference): " << reference << std::endl;

return 0;

}

**Explanation:**

**a. Function Overloading and Operator Overloading:**

Function overloading is demonstrated with the add function, which can add integers or doubles.

Operator overloading is shown with the Complex class, overloading the + operator.

**b. Pass by Value and Pass by Reference**:

**Pass by Value:** The incrementValue function increments a value passed by value, but the original variable remains unchanged.

**Pass by Reference:** The incrementReference function increments a value passed by reference, directly modifying the original variable.

**c. Parameters and Arguments:**

**Parameters:** Represented by variables in the function definition (int a, int b in add function).

**Arguments:** Actual values passed to the function when it is called (3, 4, 3.5, 4.2 in the example calls).

***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*) fallingTime = 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∗𝑎𝑡2 +𝑣𝑖𝑡+𝑥𝑖 𝑣(𝑡)=𝑎𝑡+𝑣𝑖

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.

**Solution to the provided code above**.

#include <iostream>

class CalculateG { // created class called calculateG

public:

double multi(double a, double b) {

return a \* b;

}

double square(double a) {

return a \* a;

}

double summation(double a, double b) {

return a + b;

}

void outline(double result) {

std::cout << "The result is: " << result << std::endl;

}

void calculatePositionAndVelocity() {

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

double fallingTime = 30.0;

double initialVelocity = 0.0;

double finalVelocity;

double initialPosition = 0.0;

double finalPosition;

// Calculate position and velocity

finalPosition = 0.5 \* gravity \* square(fallingTime) + initialVelocity \* fallingTime + initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

// Output the results

outline(finalPosition);

outline(finalVelocity);

}

};

int main() {

CalculateG calculator;

// Compute the position and velocity of an object

calculator.calculatePositionAndVelocity();

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.

**Did Question 1 and 3**

**Solution**

#include <iostream>

// Function to find the sum of even-valued terms in the Fibonacci sequence

int sumEvenFibonacci(int limit) {

int term1 = 1;

int term2 = 2;

int sum = 0;

while (term2 <= limit) {

if (term2 % 2 == 0) {

sum += term2;

}

// Generating the next Fibonacci terms

int nextTerm = term1 + term2;

term1 = term2;

term2 = nextTerm;

}

return sum;

}

int main() {

int limit = 4000000;

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;

}

**Explanation:**

1. The **sumEvenFibonacci** function takes a limit as an argument and calculates the sum of all even-valued terms in the Fibonacci sequence up to that limit.
2. The program uses a while loop to generate Fibonacci terms and checks if each term is even. If it is, it adds the term to the sum.
3. The loop continues until the generated term exceeds the specified limit.
4. The **main** function sets the limit to four million and prints the result.

**Question Two: [15 marks]**

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 C++ 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 🡪

**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.

1. Print the values stored in the array on screen.
2. 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”
3. 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
4. Get the sum and product of all elements of your array. Print product and the sum each on its own line.

**Solution**

#include <iostream>

int main() {

const int size = 15;

int originalArray[size];

int reversedArray[size];

// Part (a): Input values into the array and print them

std::cout << "Enter 15 integer values:" << std::endl;

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

std::cout << "Enter value " << i + 1 << ": ";

std::cin >> originalArray[i];

}

std::cout << "Values stored in the array:" << std::endl;

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

std::cout << originalArray[i] << " ";

}

std::cout << std::endl;

// Part (b): Check if a number is present in the array

int searchNumber;

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

std::cin >> searchNumber;

bool numberFound = false;

int foundIndex = -1;

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

if (originalArray[i] == searchNumber) {

numberFound = true;

foundIndex = i;

break;

}

}

if (numberFound) {

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

} else {

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

}

// Part (c): Create a reversed array and print its elements

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

reversedArray[i] = originalArray[size - i - 1];

}

std::cout << "Values of the reversed array:" << std::endl;

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

std::cout << reversedArray[i] << " ";

}

std::cout << std::endl;

// Part (d): Calculate sum and product of array elements

int sum = 0;

long long product = 1; // Using long long to handle potential overflow for product

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

sum += originalArray[i];

product \*= originalArray[i];

}

std::cout << "Sum of array elements: " << sum << std::endl;

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

return 0;

}

**Explanation:**

- The program uses two arrays (`originalArray` and `reversedArray`) to store the input values and the reversed values.

- It checks if a user-entered number is present in the array and prints the result.

- It then creates a reversed array and prints its elements.

- Finally, it calculates and prints the sum and product of the array elements.

**References**

C++ Tutorial. (n.d.). <https://www.w3schools.com/cpp/default.asp>.

*Draw.io - free flowchart maker and diagrams online*. Flowchart Maker & Online Diagram Software. (n.d.). <https://app.diagrams.net/#G1kZsiR0yOe0pAyP64liH3VYOb9n3u9kl3>.