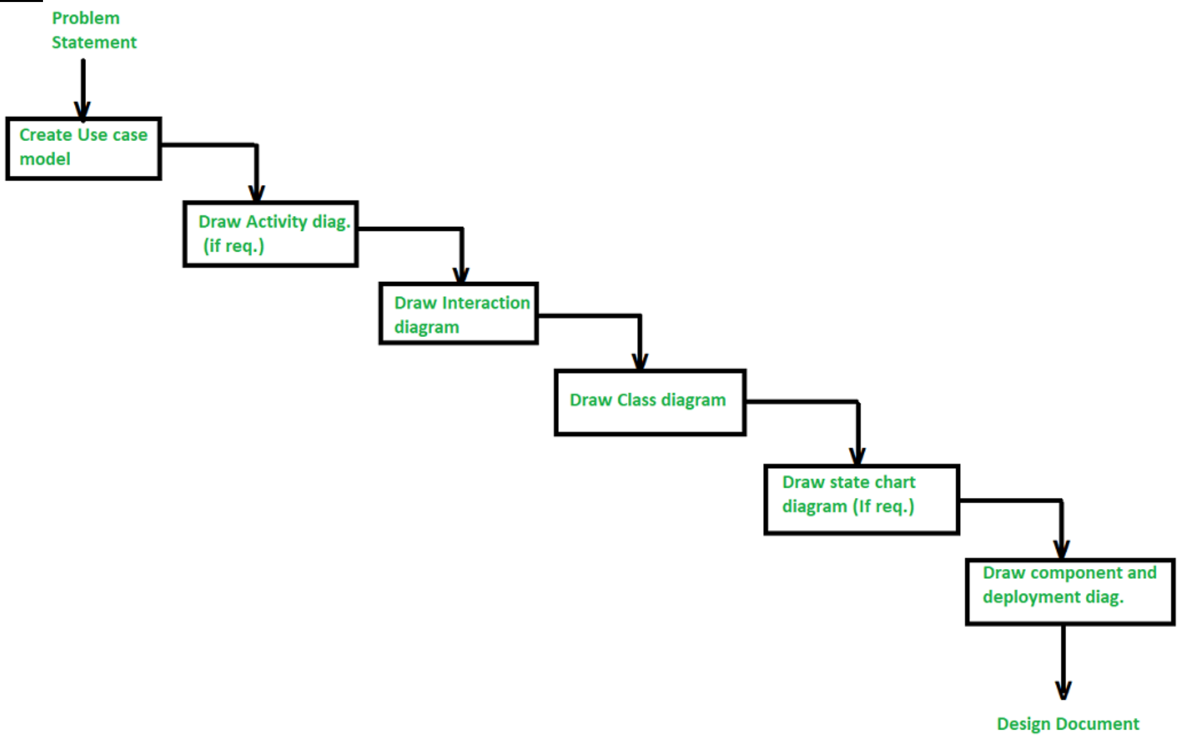
BRIAN MWANGI NJENGA   SCT121-0991/2022

PART A


Problem Statement
Create Use case model
Draw Activity diag. (if req.)
Draw Interaction diagram
Draw Class diagram
Draw state chart diagram (If req.)
Draw component and deployment diag.
Design Document

1. **Identify the actors**: The first step is to identify the actors involved in the system. Actors are the entities that interact with the system. They can be users, devices, or other systems.
2. **Create a use case diagram**: Once the actors are identified, create a use case diagram that shows the interactions between the actors and the system. A use case diagram is a visual representation of the system's functionality.
3. **Create an activity diagram**: An activity diagram shows the flow of control in the system. It demonstrates how the system responds to events and how it processes data.
4. **Create an interaction diagram**: An interaction diagram shows the communication between objects in the system. It demonstrates how objects collaborate to achieve a specific task.
5. **Create a class diagram**: A class diagram shows the structure of the system. It demonstrates the classes in the system, their attributes, and their relationships.

ii. The Object Modeling Technique (OMT) is a real world based modeling approach for software modeling and designing. It was developed as a method to develop object-oriented systems and to support object-oriented programming. It describes the static structure, dynamic behavior, and functional aspects of a system using three types of models: object model, dynamic model, and functional model.

iii. Object-oriented analysis and design (OOAD) and object analysis and design (OOP) are both software engineering methodologies that involve using object-oriented concepts to design and implement software systems. However, they differ in some aspects, such as:

- OOAD is more comprehensive and covers the entire software development life cycle, from requirements analysis to testing and maintenance. OOP is more focused on the implementation phase and the programming language features.

- OOAD uses a standardized notation, such as UML, to create diagrams that represent different aspects of a software system, such as classes, objects, relationships, states, and interactions. OOP does not have a universal notation, but relies on the syntax and semantics of the chosen programming language.

- OOAD emphasizes the use of design patterns, which are reusable solutions to common problems in software design. OOP does not explicitly use design patterns, but may implement some of them implicitly or explicitly depending on the programming language and the programmer's style.

iv. The main goals of UML are:

- To define a general-purpose modeling language that all modelers can use, regardless of the domain, platform, or methodology.

- To be nonproprietary and based on common agreement by much of the computing community, especially the leading object-oriented methods and tools.

- To be as familiar as possible to the existing users of object-oriented methods, by using notation from OMT, Booch, Objectory, and other leading methods.

- To support good practices for design, such as encapsulation, separation of concerns, and capture of requirements and design decisions.

- To be extensible and customizable, by allowing users to define their own stereotypes, tagged values, and constraints.

v. Some advantages of using object-oriented to develop an information system are:

- Reusability: Object-oriented systems can reuse existing components and design patterns, which can save time and effort in software development.

- Scalability: Object-oriented systems can handle changes in user demand and business requirements over time, by allowing new classes and objects to be added or modified without affecting the existing ones.

- Maintainability: Object-oriented systems are easier to maintain and update over time, by using modular design and encapsulation to isolate the effects of changes and reduce the complexity of the system.

VI.

a. Constructor: A constructor is a special method that is used to initialize an object when it is created. It has the same name as the class and no return type. In Java, a constructor can be used to set the initial values of the object's attributes or to perform any other initialization tasks. Here is an example of a constructor in Java:

```
public class Car {
  private String make;
  private String model;
  private int year;

  public Car(String make, String model, int year) {
    this.make = make;
    this.model = model;
    this.year = year;
  }
}
```

b. Object: An object is an instance of a class. It is created using the new keyword followed by the name of the class and any arguments required by the constructor. Objects have attributes (also known as fields) and methods that define their behavior. Here is an example of creating an object in Java:

```
Car myCar = new Car("Toyota", "Corolla", 2022);
```

c. Destructor: Unlike some other programming languages, Java does not have a destructor. Instead, Java uses a garbage collector to automatically free up memory that is no longer being used by an object.

d. Polymorphism: Polymorphism is the ability of an object to take on many forms. In Java, polymorphism is achieved through method overriding and method overloading. Method overriding allows a subclass to provide a specific implementation of a method that is already provided by its parent class. Method overloading allows multiple methods to have the same name, but different parameters. Here is an example of polymorphism in Java:

```
public class Animal {

  public void makeSound() {

    System.out.println("The animal makes a sound");

  }

}


public class Dog extends Animal {

  public void makeSound() {

    System.out.println("The dog barks");

  }

}


public class Cat extends Animal {

  public void makeSound() {

    System.out.println("The cat meows");

  }

}
```

```
public class Main {

  public static void main(String[] args) {

    Animal myAnimal = new Animal();

    Animal myDog = new Dog();

    Animal myCat = new Cat();


    myAnimal.makeSound();

    myDog.makeSound();

    myCat.makeSound();

  }

}
```

e. Class: A class is a blueprint for creating objects. It defines the attributes and methods that an object will have. In Java, a class is defined using the class keyword followed by the name of the class. Here is an example of a class in Java:

```
public class Car {

  private String make;

  private String model;

  private int year;


  public Car(String make, String model, int year) {

    this.make = make;

    this.model = model;
    this.year = year;
  }

  public String getMake() {
    return make;
  }

  public String getModel() {
    return model;
  }

  public int getYear() {
    return year;
  }
}
```

f. Inheritance: Inheritance is a mechanism in which one class acquires the properties and methods of another class. The class that is being inherited from is called the parent class or superclass, and the class that is inheriting is called the child class or subclass. In Java, inheritance is achieved using the extends keyword. Here is an example of inheritance in Java:

```
public class Vehicle {
  protected String make;
  protected String model;
  protected int year;
  public Vehicle(String make, String model, int year) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  public String getMake() {
    return make;
  }

  public String getModel() {
    return model;
  }

  public int getYear() {
    return year;
  }
}

public class Car extends Vehicle {
  private int numDoors;

  public Car(String make, String model, int year, int numDoors) {
    super(make, model, year);

    this.numDoors = numDoors;

  }


  public int getNumDoors() {

    return numDoors;

  }

}
```

VII.

- Three types of associations:

    - Association is a semantic connection between classes that indicates that objects of one class can send messages to objects of another class[1]. Associations can be bi-directional or unidirectional[2].

    - Dependency is a weaker form of association that shows that one class depends on the definitions of another class[3]. Dependencies are always unidirectional and are represented by dashed arrows.

    - Aggregation is a stronger form of association that shows the relationship between a whole and its parts[4]. Aggregations are transitive, antisymmetric, and non-cyclic. They are represented by a diamond at the end of the association line.

VIII.

- Class diagram is a static structure diagram that shows the classes, attributes, operations, and relationships of a system. Class diagrams are used to model the logical and conceptual aspects of a system, such as the domain model, the analysis model, and the design model. Class diagrams can also show the implementation details of a system, such as the source code, the database schema, and the components.

    - Steps to draw a class diagram:

        - Identify the classes and their attributes and operations based on the requirements or the existing system.

- Draw the classes as rectangles with three compartments: the top one for the class name, the middle one for the attributes, and the bottom one for the operations.

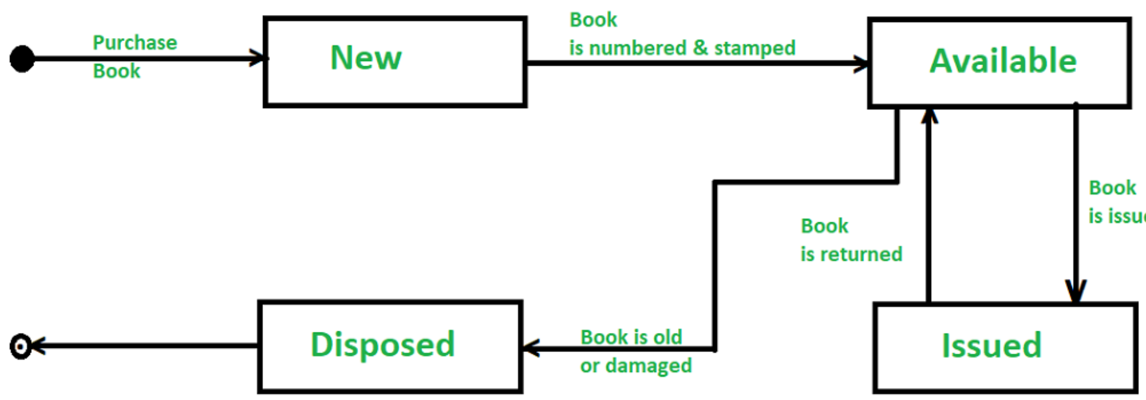
- Identify the associations and dependencies between the classes and draw them as lines connecting the classes. Use appropriate symbols and labels to indicate the type, direction, multiplicity, and role of the associations and dependencies.

- Identify the aggregations and generalizations between the classes and draw them as lines with diamonds or triangles at the end. Use appropriate labels to indicate the name and scope of the aggregations and generalizations.

- Verify the correctness and completeness of the class diagram by checking the consistency, clarity, and validity of the classes and their relationships.

- Example of a class diagram: The following class diagram shows a simplified model of a library system. It has four classes: Book, Member, Loan, and Librarian. Book has attributes such as title, author, and ISBN, and operations such as checkAvailability and reserve. Member has attributes such as name, address, and phone, and operations such as borrow and return. Loan has attributes such as dueDate and fine, and operations such as calculateFine and renew. Librarian has attributes such as name, ID, and password, and operations such as issue and receive. Book and Member have a many-to-many association called Loan, which has a role name of loanedTo and loanedBy. Book and Loan have a one-to-many aggregation called Catalog, which has a name of books. Member and Loan have a one-to-many aggregation called Record, which has a name of loans. Librarian has a one-to-many dependency on Loan, which has a name of manages. Librarian also has a generalization from Member, which has a name of staff.

Class diagram example



This is the implementation of the calculator using the OOP concepts:

```
#include <iostream>

#include <cmath>


using namespace std;


// Abstract class Shape

class Shape {

public:

   virtual double area() = 0;

   virtual double perimeter() = 0;

};


// Class Circle derived from Shape

class Circle : public Shape {

private:

   double radius;

public:

   Circle(double r) {

      radius = r;

   }

   double area() {

      return M_PI * radius * radius;
```

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

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

// Class Triangle derived from Shape
```

```
class Triangle : public Shape {

private:

   double a, b, c;

public:

   Triangle(double x, double y, double z) {

      a = x;

      b = y;

      c = z;

   }

   double area() {

      double s = (a + b + c) / 2;

      return sqrt(s * (s - a) * (s - b) * (s - c));

   }

   double perimeter() {

      return a + b + c;

   }

};


// Class Square derived from Rectangle

class Square : public Rectangle {

public:

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

};
```

```
// Friend function to display the area and perimeter of a shape

void display(Shape& s) {

    cout << "Area: " << s.area() << endl;

    cout << "Perimeter: " << s.perimeter() << endl;

}


int main() {

    Circle c(5);

    Rectangle r(4, 6);

    Triangle t(3, 4, 5);

    Square s(7);


    display(c);

    display(r);

    display(t);

    display(s);


    return 0;

}
```

a. Inheritance is a mechanism in which one class acquires the properties and methods of another class. In this program, we have used single inheritance, where a derived class inherits from a single base class. The classes Circle, Rectangle, Triangle, and Square are derived from the base class Shape.

b. Friend functions are non-member functions that have access to the private and protected members of a class. In this program, we have used a friend function display to display the area and perimeter of a shape.

c. Method overloading is a feature of OOP that allows a class to have multiple methods with the same name but different parameters. In this program, we have overloaded the constructor of the Rectangle class to accept two parameters and the constructor of the Square class to accept one parameter.

Method overriding is a feature of OOP that allows a subclass to provide a specific implementation of a method that is already provided by its parent class. In this program, we have overridden the area and perimeter methods of the Shape class in the derived classes Circle, Rectangle, Triangle, and Square.

d. Late binding is a feature of OOP that allows the selection of the appropriate method implementation at runtime based on the actual type of the object. In this program, we have used late binding to call the area and perimeter methods of the derived classes through a reference to the base class Shape.

Early binding is a feature of OOP that allows the selection of the appropriate method implementation at compile-time based on the declared type of the object.

e. Abstract class is a class that cannot be instantiated and has at least one pure virtual function. In this program, the Shape class is an abstract class because it has two pure virtual functions area and perimeter.

Pure functions are functions that do not modify the state of the object and do not have any side effects. In this program, the area and perimeter methods of the Shape class are pure functions.

viii. Here are the differences between the following:

a. Function overloading is a feature of C++ that allows a function to have multiple definitions with the same name but different parameters. The compiler selects the appropriate function to call based on the number, types, and order of the arguments passed to the function.

Operator overloading is a feature of C++ that allows an operator to have multiple meanings based on the context in which it is used. For example, the + operator can be used to add two numbers or concatenate two strings.

b. Pass by value is a method of passing arguments to a function in which a copy of the argument is made and passed to the function. Any changes made to the argument inside the function do not affect the original value of the argument.

Pass by reference is a method of passing arguments to a function in which the memory address of the argument is passed to the function. Any changes made to the argument inside the function affect the original value of the argument.

c. Parameters are the variables declared in the function definition that receive the values passed as arguments when the function is called.

Arguments are the values passed to a function when it is called. They are assigned to the parameters declared in the function definition.

6.

This is the modified version of the code to compute the position and velocity of an object after falling for 30 seconds, outputting the position in meters:

```
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 = gravity * fallingTime + initialVelocity;
    double initialPosition = 0.0;
    double finalPosition = 0.5 * gravity * Math.pow(fallingTime, 2) + initialVelocity * fallingTime +
initialPosition;

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

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


  public double power(double a) { // method for powering to square

    return Math.pow(a, 2);

  }


  public double sum(double a, double b) { // method for summation

    return a + b;

  }


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

    System.out.println("The result is " + result);

  }

}
```

In this modified version of the code, I have added the formulas for position and velocity based on the given formula in Math notation. I have also added methods for multiplication, powering to square, summation, and printing out a result. Finally, I have computed the position and velocity of an object with the defined methods and printed out the result.

You can run this code in Eclipse by selecting Run → Run As → Java Application.

PART B

1.

- The sum of all even-valued terms in the Fibonacci sequence whose values do not exceed four million can be calculated using the following C++ code:

```
#include <iostream>

using namespace std;

int main() {
   int limit = 4000000;
   int sum = 0;
   int a = 1, b = 2;
   while (b <= limit) {
      if (b % 2 == 0) {
         sum += b;
      }
      int c = a + b;
      a = b;
      b = c;
   }
   cout << "The sum of all even-valued terms in the Fibonacci sequence whose values do not exceed four million is " << sum << endl;
   return 0;
}
```

3.

- This is a C++ program that takes 15 values of type integer as inputs from the user, stores the values in an array, and performs the following operations:

a) Prints the values stored in the array on screen.

b) Asks the user to enter a number, checks if that number (entered by the user) is present in the array or not. If it is present, prints "The number found at index (index of the number)." Otherwise, prints "Number not found in this array."

c) Creates another array, copies all the elements from the existing array to the new array but in reverse order. Now prints the elements of the new array on the screen.

d) Gets the sum and product of all elements of the array. Prints the product and the sum each on its own line.

```
#include <iostream>


using namespace std;


int main() {

   int arr[15];

   cout << "Enter 15 integers:" << endl;

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

      cin >> arr[i];

   }

   cout << "The values stored in the array are:" << endl;

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

      cout << arr[i] << " ";
   }

   cout << endl;

   int num;

   cout << "Enter a number to search: ";

   cin >> num;

   bool found = false;

   int index;

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

      if (arr[i] == num) {

          found = true;

          index = i;

          break;

      }

   }

   if (found) {

      cout << "The number found at index " << index << "." << endl;

   } else {

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

   }

   int revArr[15];

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

      revArr[i] = arr[14 - i];

   }

   cout << "The values stored in the reversed array are:" << endl;
```

```
    for (int i = 0; i < 15; i++) {
        cout << revArr[i] << " ";
    }
    cout << endl;
    int sum = 0;
    int product = 1;
    for (int i = 0; i < 15; i++) {
        sum += arr[i];
        product *= arr[i];
    }
    cout << "The sum of all elements of the array is " << sum << "." << endl;
    cout << "The product of all elements of the array is " << product << "." << endl;
    return 0;
}
```