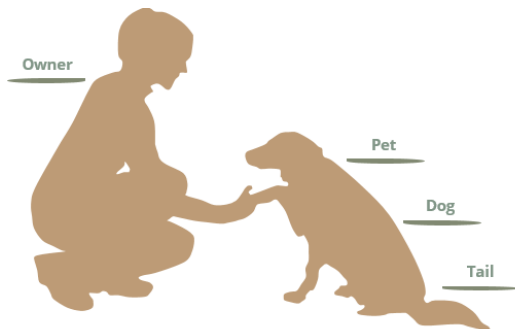


Notes ESE

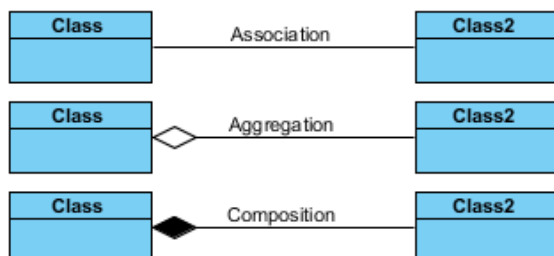
📅 Exam Date	@15/01/2024 9:00
📁 Module	Object Oriented Programming and Modelling
☰ Units	1 2 3 4
📄 Code	COMP 20043
⚙️ Status	In progress
📅 Semester	Fall 23
📄 Type	Final

1. Differentiate between Aggregation and Composition with example (Unit 4 : Slides 6-15).

Aggregation and Composition are both types of associations between classes in object-oriented programming.

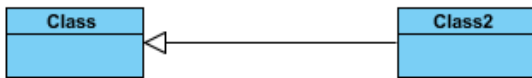


- owners feed pets, pets please owners (**association**)
- a tail is a part of both dogs and cats (**aggregation / composition**)
- a cat is a kind of pet (**inheritance / generalization**)



- owners feed pets, pets please owners (**association**)
- a tail is a part of both dogs and cats (**aggregation / composition**)
- a cat is a kind of pet (**inheritance / generalization**)

generalization

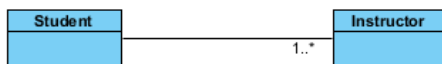


Association with multiplicity

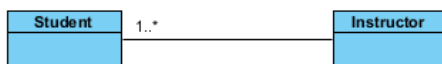
Association is a relationship between two or more objects or classes. This relationship is described as "has-a" or "uses-a". For instance, a car has an engine, or a student uses a book. Therefore, the car is associated with the engine, and the student is associated with the book.

The multiplicity of the association, which describes how many instances of one class are associated with one instance of the other class, can also be indicated at each end of the line.

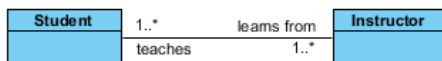
A single student can associate with multiple teachers:



The example indicates that every Instructor has one or more Students:

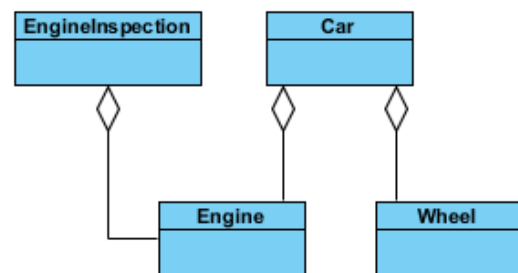


We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

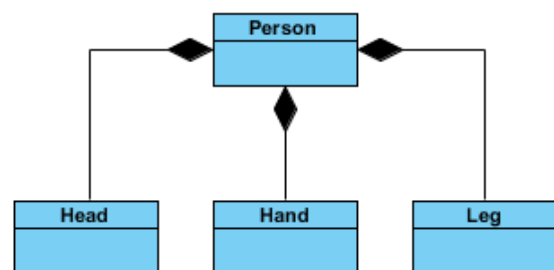


- **Aggregation** means a relationship where the child can exist even without the parent. For example: Class (parent) and Student (child). Even if the Class is gone, the Students still exist.
- **Composition** means a relationship where the child can't exist without the parent. For example: House (parent) and Room (child). Rooms can't exist without a House.

For example, a Car (class) can have multiple instances of Engine (class) and Wheel (class), which signifies that a Car is composed of Engines and Wheels. However, both Engine and Wheel can exist independently even if the Car is destroyed, so this is an example of Aggregation.



Composition represents a stronger "whole-part" relationship where one class is composed of another class, and the lifetime of the composed class depends on the lifetime of the composing class. For example, a Person's existence directly affects the existence of the Head, Hand, and Leg. This shows a strong lifecycle dependency between these classes.



2. Explain Activity Diagram with example.

An Activity Diagram is a type of diagram in the Unified Modeling Language (UML), which is used to represent workflows of stepwise activities and actions that are involved in a process or a part of a system. It is particularly useful in visualizing the

flow of control in a system, modeling the process, and depicting the sequence from one operation to the next.

Activity Diagrams are highly versatile and can describe a wide range of behaviors, from the workflow within a single use case to the high-level flow of business processes. They cover both sequential and concurrent activities, providing support for choice, iteration, and concurrency, which means they can represent complex conditional or repetitive activities as well as parallel activities.

The main elements of an Activity Diagram include activity states, transitions, decision nodes, and fork/join nodes. Activity states represent the performance of an action or a sub-activity. Transitions are represented by arrows that show the order of execution of activities. Decision nodes are used to represent conditional branching, where different outcomes are possible based on a certain condition. Fork nodes are used to split the flow into multiple concurrent flows, and join nodes are used to synchronize multiple concurrent flows.

In essence, Activity Diagrams graphically illustrate the procedural logic, control and data flow of a system. They are a powerful tool for visualizing and understanding the dynamic behavior of a system, aiding in effective system design and analysis.

3. Describe the steps for creating use case Diagram (Unit3: Slides 8-9).

Follow this guide to create a use case diagram:

1. Define the system's scope and boundaries: Determine what your system is designed to do and establish its limits. This will help focus the use case diagram on the necessary functionalities and actors.
2. Identify the actors: Make a list of all users and external systems that will interact with your system. Consider individuals, groups, organizations, or other systems that either use your system or are impacted by it.
3. Define the use cases: Identify the different functionalities, processes, or services your system provides to meet the needs of the actors. Summarize the purpose of each use case with a short, descriptive name.
4. Draw the actors and use cases: Use a diagramming tool or software to visually represent the actors and use cases. Typically, actors are shown as stick figures,

people icons, or labeled rectangles (for external systems), and use cases are represented as ovals or rounded rectangles with their names inside.

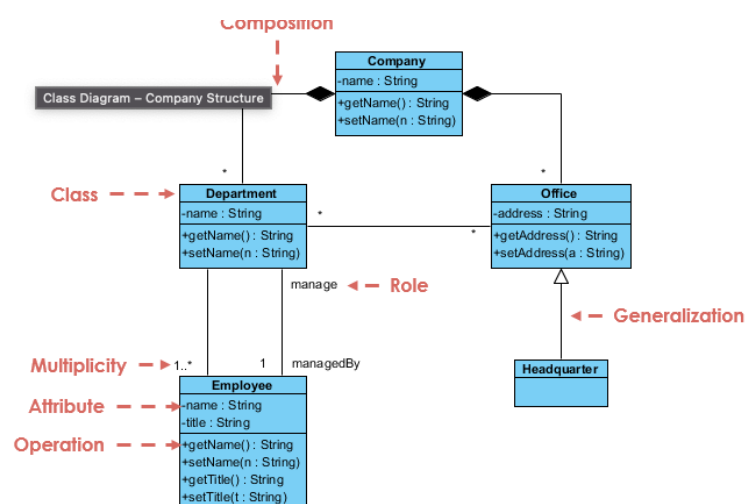
5. Establish relationships: Connect the actors and use cases with solid lines to illustrate associations. For "includes" and "extends" relationships between use cases, use dashed arrows labeled with "«include»" and "«extend»" respectively.
6. Review and refine: Review your use case diagram carefully for accuracy and clarity. Make any necessary adjustments and ensure it effectively communicates the system's requirements and interactions.

4. Discuss Unified Modeling Language (UML) and any three diagrams (Unit 2: slides 14-17).

A model is our interpretation of our perception of reality. It helps us understand, define, or simulate a system or concept by breaking complex structures into simpler ones.

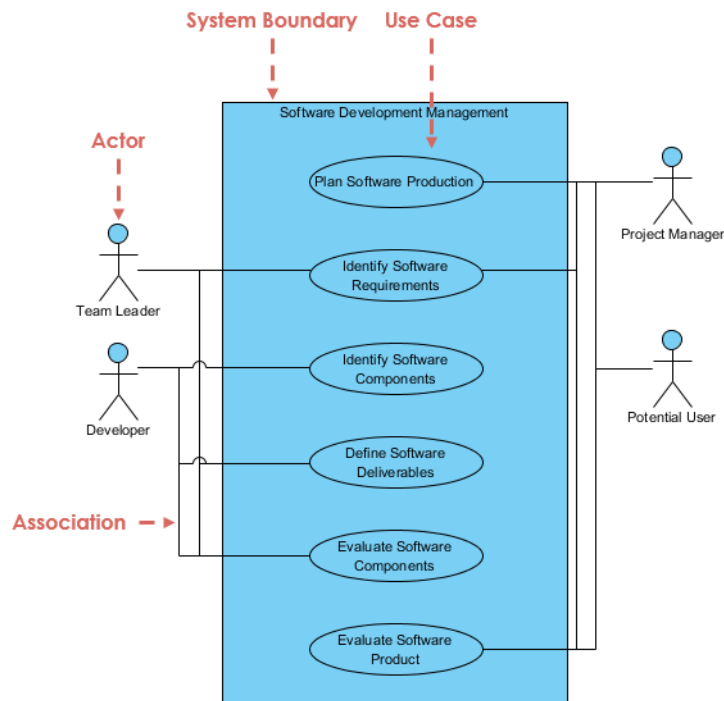
Unified Modeling Language (UML) provides a standard way to visualize and document a system's design. Common UML diagrams: Class Diagram, Activity Diagram, Sequence Diagram, and Use Case Diagram.

1. The Class Diagram represents the system's static structure, illustrating classes, their attributes, operations, and relationships.



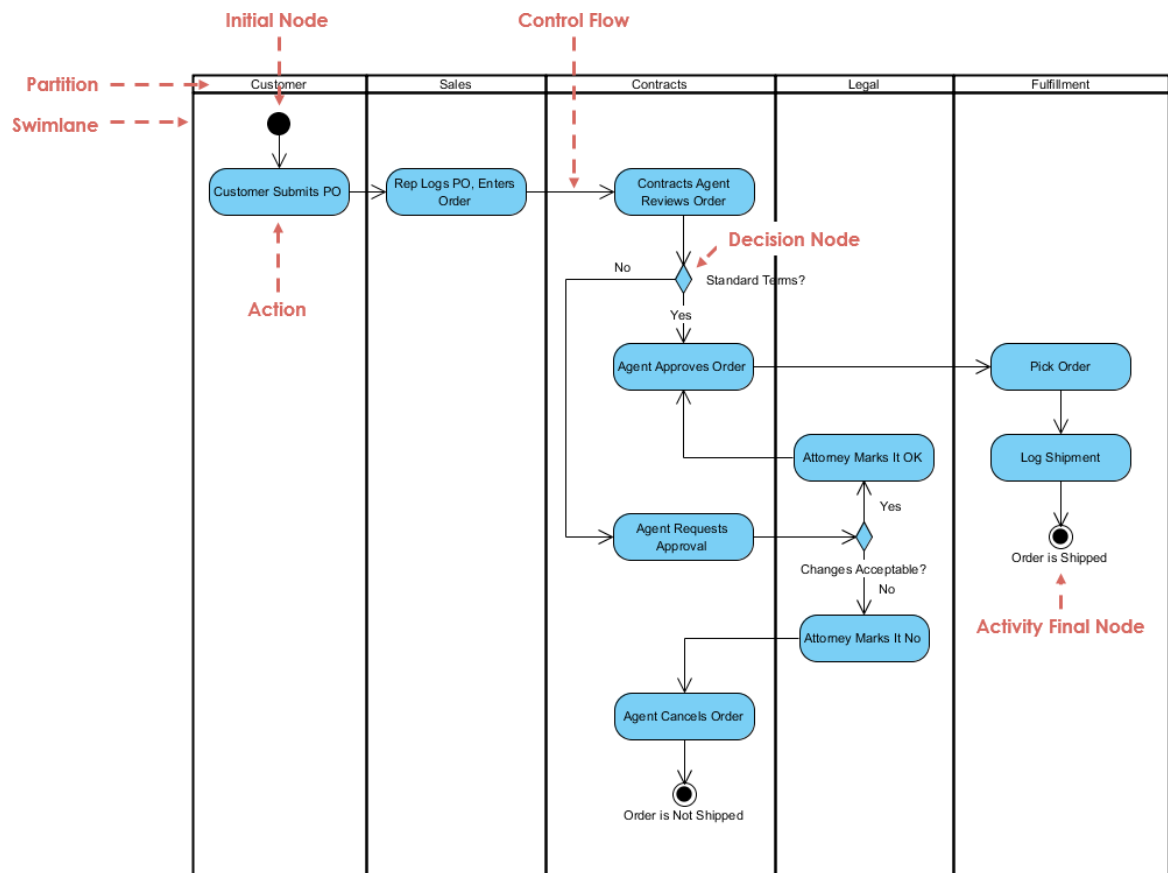
2. The Use Case

Diagram describes the system's functionality from a user's point of view, identifying actors and their interactions with the system.

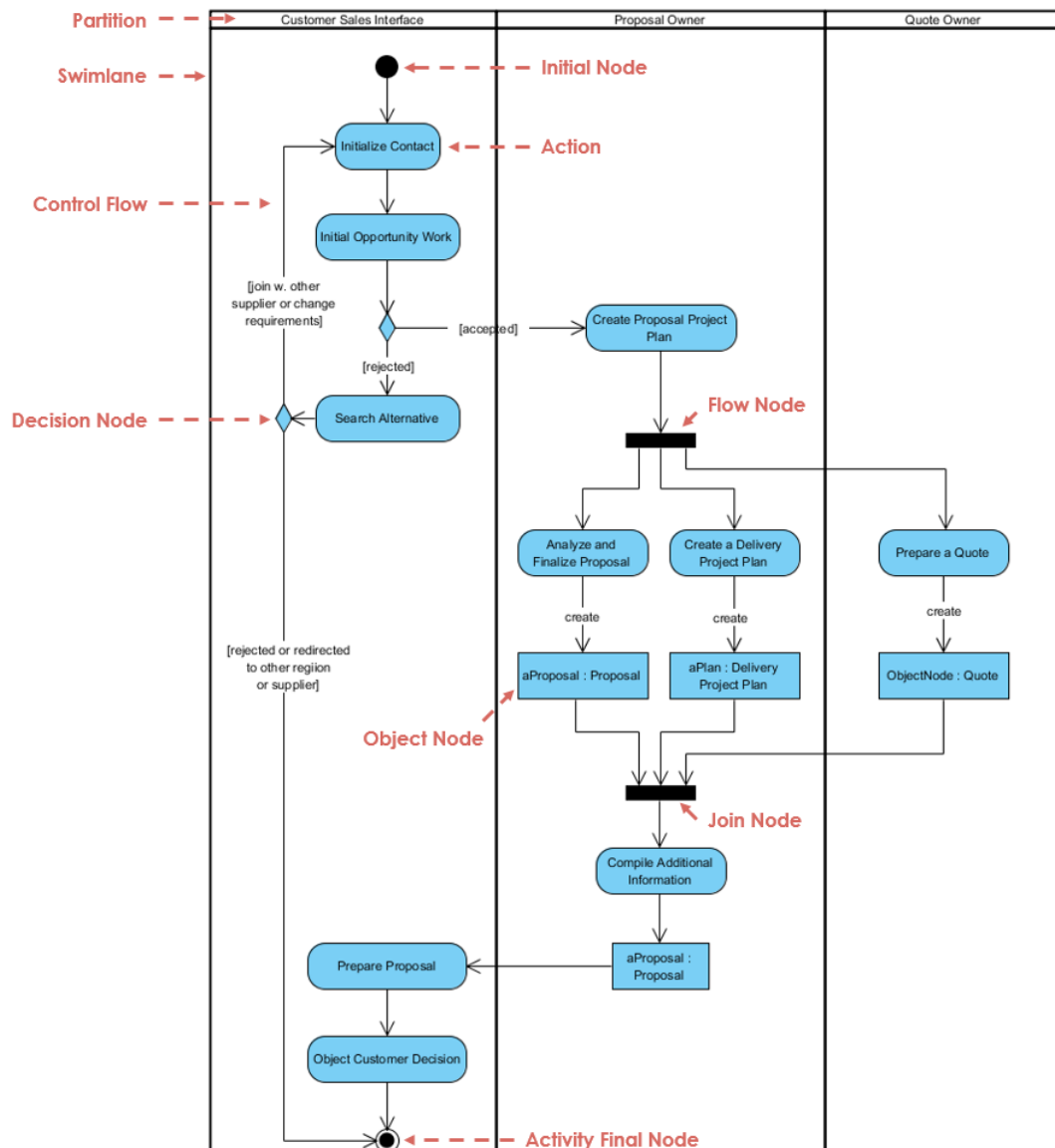


3. Activity Diagram: This UML diagram depicts a process or operation's workflow. It shows a sequence of activities, each representing a process step. Arrows represent control flow, demonstrating the path from one activity to the next. It also illustrates decision points and parallel processing, providing a comprehensive process view. They are useful for illustrating complex processes, mapping software algorithms, and understanding system control flow.

example :



Another example :



5. You have been tasked with designing a software system for a car rental company. The system should allow customers to rent cars, view available cars, and return cars. The system should also allow employees to manage reservations, view rental history, and add or remove cars from the fleet. Discuss the use of UML and various modelling techniques to design the system.

1. **Use Case Diagram:** This would help identify the different actors (customers, employees) and their interactions with the system. For example, customers will have use cases such as "View Cars", "Rent Car", "Return Car", while employees might have use cases like "Manage Reservations", "View Rental History", "Add/Remove Cars".
2. **Class Diagram:** This would define the classes and their relationships. For instance, classes could include "Customer", "Employee", "Car", "Reservation", and "RentalHistory". The relationships between these classes are illustrated in the class diagram.
3. **Activity Diagram:** This would be used to model the flow of activities involved in renting and returning cars. For instance, it could show the steps a customer takes to rent a car or the process an employee follows to manage reservations.
4. **Sequence Diagram:** This diagram could be used to show the interactions between different system components (like the user interface, database, and backend services) when a particular process occurs, such as when a customer rents a car.

-
6. **Create a class named Mobile having 4 Instance variables Model, price , weight, color. The class should have a method Display() which should display all the attributes of the class. The class should also have a one parameterized constructor to initialize all the attributes with values passed from main method. Create a main class that contains the main method. Create 3 objects of the class Mobile using an array. Initialize all the objects. Display the details of the mobile with the lowest price.**

```
class MobileESE_PQ {  
    String model, color;  
    double price, weight;  
  
    MobileESE_PQ(String model, String color, double price, do  
        this.model = model;  
        this.color = color;
```

```

        this.price = price;
        this.weight = weight;
    }

    void Display() {
        System.out.println("Model: " + model);
        System.out.println("Color: " + color);
        System.out.println("Price: " + price);
        System.out.println("Weight: " + weight);
    }
}

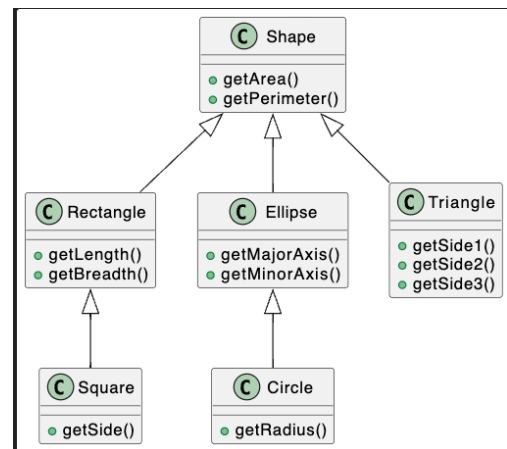
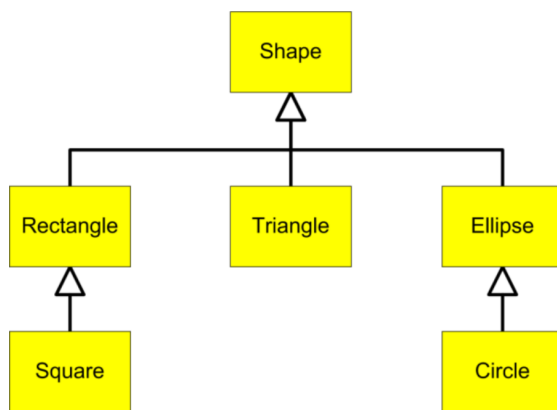
class MainMobile {
    public static void main(String[] args) {
        MobileESE_PQ[] NewMobile = new MobileESE_PQ[3];
        NewMobile[0] = new MobileESE_PQ("Samsung", "Black", 50000);
        NewMobile[1] = new MobileESE_PQ("Nokia", "Green", 20400);
        NewMobile[2] = new MobileESE_PQ("Apple", "Red", 80660);

        int index = 0;
        for (int i = 1; i < NewMobile.length; i++) {
            if (NewMobile[0].price < NewMobile[i].price) {
                index = i;
            }
        }
        System.out.println("Details of the mobile with the low price is:");
        NewMobile[index].Display();
    }
}

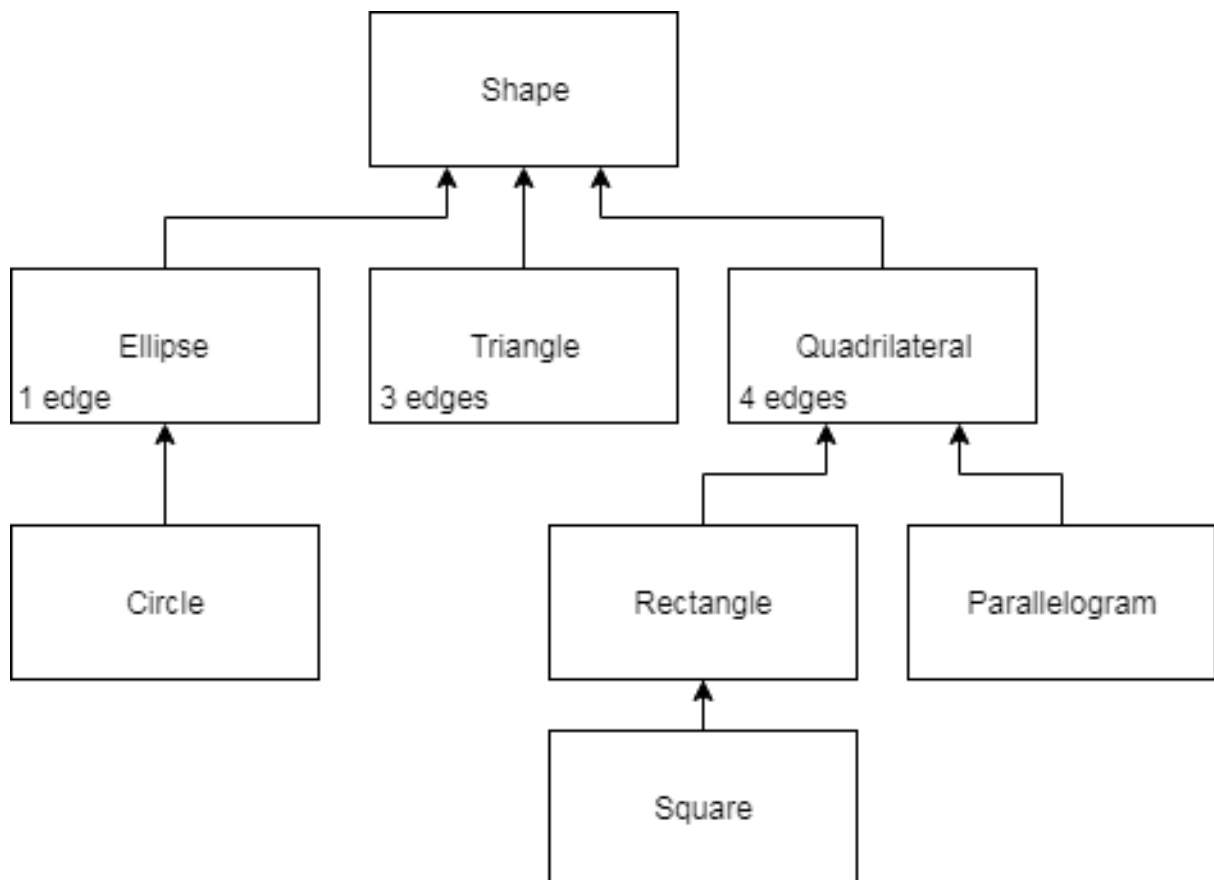
```

7. Create a diagram and justify an inheritance hierarchy that could be used to represent the following classes : shape, rectangle, square, circle, ellipse, triangle

Note: You can solve this without using the methods implied in the classes.



To help you understand why



8. Create a diagram and justify inheritance hierarchy that could be used to represent the following classes with appropriate attributes and operations:

person, employee, full-time employee, manager,

Implement the above inheritance hierarchy using JAVA. Write the code for each class and include the attributes and necessary methods as given below:

1. Person class:

- Variables: id and name
- Methods: void inputDetails() to assign some values to variables. void displayDetails() – to display the values of the variables.

1. Employee class:

- void getsalary() prints “employee earns salary”

1. fulltimeemployee class:

- void paidleave() prints “can take one month paid leave”

1. manager class:

- void getbonus() prints “ gets bonus every month”

1. student class:

- void getmarks() prints “ gets marks for the modules”

Additionally, create a **main** method in a Main class to demonstrate the behavior of the different person types. Inside the **main** method. Create an instance of the **manager** class And Create an instance of the **student** class. Call the methods of these two classes using the objects created.

```
class PersonESE_PQ {
    int id;
    String name;

    void inputDetails(int id, String name) {
        this.id = id;
        this.name = name;
    }

    void displayDetails() {
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
    }
}
```

```

    }
}

class Employee extends PersonESE_PQ {
    void getSalary() {
        System.out.println("Employee earns salary");
    }
}

class FullTimeEmployee extends Employee {
    void paidLeave() {
        System.out.println("Can take one month paid leave");
    }
}

class Manager extends Employee {
    void getBonus() {
        System.out.println("Gets bonus every month");
    }
}

class Student extends PersonESE_PQ {
    void getMarks() {
        System.out.println("Gets marks for the modules");
    }
}

class Main {
    public static void main(String[] args) {
        Manager m = new Manager();
        Student s = new Student();
        m.inputDetails(1, "Taher");
        s.inputDetails(2, "Test");
        m.displayDetails();
        m.getSalary();
        m.getBonus();
        s.displayDetails();
        s.getMarks();
    }
}

```

```
}  
}
```

9. Using the correct syntax in java, define a class called Vehicle that contains 2 data members: vehicleId and vehicleType, and 2 member functions: void input() to assign input values and void display() to display the values. Appropriate data types are to be chosen. Using the correct syntax in java, show how a subclass called myCar can be derived from the class vehicle to include additional data members carModel, carBrand. Appropriate data types are to be chosen.

```
class Vehicle_ESE {  
    int vehicleId;  
    String vehicleType;  
  
    void input(int vehicleId, String vehicleType) {  
        this.vehicleId = vehicleId;  
        this.vehicleType = vehicleType;  
    }  
  
    void display() {  
        System.out.println("Vehicle ID: " + vehicleId);  
        System.out.println("Vehicle Type: " + vehicleType);  
    }  
}  
  
class myCar_ESE extends Vehicle_ESE {  
    String carModel;  
    String carBrand;  
  
    void input(int vehicleId, String vehicleType, String carM  
        super.input(vehicleId, vehicleType);  
        this.carModel = carModel;  
        this.carBrand = carBrand;  
    }  
}
```

```

        void display() {
            super.display();
            System.out.println("Car Model: " + carModel);
            System.out.println("Car Brand: " + carBrand);
        }
    }

    class VehicleMain_ESE {
        public static void main(String[] args) {
            myCar_ESE car = new myCar_ESE();
            car.input(1, "Car", "Model 3", "Tesla");
            car.display();
        }
    }

```

10. Using the correct syntax in java, define a class called Book that contains three data members: bookId, title, and author, and two member functions: void input() to assign input values and void display() to display the values. Appropriate data types are to be chosen. Using the correct syntax in Java, show how a subclass called AudioBook can be derived from the class Book to include additional data members length and narrator. Appropriate data types are to be chosen.

```

class Book_ESE_PQ10 {
    int bookId;
    String title;
    String author;

    void input(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }

    void display() {
        System.out.println("Book ID: " + bookId);
    }
}

```

```

        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
    }
}

class AudioBook_ESE_PQ10 extends Book_ESE_PQ10 {
    int length;
    String narrator;

    void input(int bookId, String title, String author, int l
        super.input(bookId, title, author);
        this.length = length;
        this.narrator = narrator;
    }

    void display() {
        super.display();
        System.out.println("Length: " + length);
        System.out.println("Narrator: " + narrator);
    }
}

class MainBook_ESE {
    public static void main(String[] args) {
        AudioBook_ESE_PQ10 book = new AudioBook_ESE_PQ10();
        book.input(1, "The Alchemist", "Paulo Coelho", 3, "Je
        book.display();
    }
}

```

11. Using JAVA program, Define a class testResult with 5 instance variables namely moduleName, test1, test2, test3 , total and avg and 4 methods namely assign() to assign values to the instance variables, calcTotal() to calculate the total test score, calcAvg() to calculate the average test scores and display() to display the content of all the instance variables. Define a Main class with main method. Create an array of

class testResult to store the results of 3 students. Assign values to all the 3 objects using assign() method and display the values of all variables using display() method.

```
class TestResult_ESE {

    String moduleName;
    double test1, test2, test3, total, avg;

    void assign(String moduleName, double test1, double test2,
        double test3) {
        this.moduleName = moduleName;
        this.test1 = test1;
        this.test2 = test2;
        this.test3 = test3;
    }

    void calcTotal() {
        total = test1 + test2 + test3;
    }

    void calcAvg() {
        avg = total / 3;
    }

    void display() {
        System.out.println("Module Name: " + moduleName);
        System.out.println("Test 1: " + test1);
        System.out.println("Test 2: " + test2);
        System.out.println("Test 3: " + test3);
        System.out.println("Total: " + total);
        System.out.println("Average: " + avg);
    }
}

class testResultMain_ESE {
    public static void main(String[] args) {
        TestResult_ESE[] results = new TestResult_ESE[3];
        results[0] = new TestResult_ESE();
    }
}
```

```

        results[1] = new TestResult_ESE();
        results[2] = new TestResult_ESE();

        results[0].assign("Maths", 17, 21, 30);
        results[1].assign("Physics", 13, 23, 33);
        results[2].assign("Psychology", 27, 20, 50);

        results[0].calcTotal();
        results[1].calcTotal();
        results[2].calcTotal();

        results[0].calcAvg();
        results[1].calcAvg();
        results[2].calcAvg();

        results[0].display();
        results[1].display();
        results[2].display();
    }
}

```

12. Create a class called Employee with instance variables name, id, and salary, as well as methods to set and get the values of these variables. Then, create a subclass called Manager that adds additional instance variables bonus and teamSize and methods to set and get their values, as well as a method to calculate Bonus and total salary of the manager. Bonus = 200 OMR if teamSize is more than 50 else bonus is 100 OMR . Total salary is salary + bonus. Create a Main class, create object of Manager class and call all the methods

```

class Employee_ESE {
    String name;
    int id;

```

```

    double salary;

    void setName(String name) {
        this.name = name;
    }

    void setID(int id) {
        this.id = id;
    }

    void setSalary(double salary) {
        this.salary = salary;
    }

    String getName() {
        return name;
    }

    int getID() {
        return id;
    }

    double getSalary() {
        return salary;
    }
}

class Manager_ESE extends Employee_ESE {
    double bonus;
    int teamSize;

    void setBonus(double bonus) {
        this.bonus = bonus;
    }

    void setTeamSize(int teamSize) {
        this.teamSize = teamSize;
    }
}

```

```

    double getBonus() {
        return bonus;
    }

    int getTeamSize() {
        return teamSize;
    }

    double calcBonus() {
        if (teamSize > 50) {
            return 200;
        } else {
            return 100;
        }
    }

    double calcTotalSalary() {
        return salary + calcBonus();
    }
}

class Employee_Main_ESE {
    public static void main(String[] args) {
        Manager_ESE manager = new Manager_ESE();
        manager.setName("Taher");
        manager.setID(123333);
        manager.setSalary(1099);
        manager.setTeamSize(51);

        System.out.println("Name: " + manager.getName());
        System.out.println("ID: " + manager.getID());
        System.out.println("Salary: " + manager.getSalary());
        System.out.println("Team Size: " + manager.getTeamSize());
        System.out.println("Bonus: " + manager.calcBonus());
        System.out.println("Total Salary: " + manager.calcTotalSalary());
    }
}

```

13. Create a diagram that shows inheritance hierarchy that could be used to represent the following classes with appropriate attributes and operations: person, doctor, patient, general practitioner, and specialist.

- Person (Parent class)
 - Doctor (Child class)
 - General Practitioner (Grandchild class)
 - Specialist (Grandchild class)
 - Patient (Child class)

14. Create a diagram to justify inheritance hierarchy that could be used to represent the following classes with appropriate attributes and operations: person, part-time teacher, full-time teacher, hod,

- Person (Parent class)
 - Attributes: name, age, gender
 - Operations: speak(), walk(), eat()
 - Part-time Teacher (Child class)
 - Attributes: subject, hoursWorked
 - Operations: teach(), gradePapers()
 - Full-time Teacher (Child class)
 - Attributes: subject
 - Operations: teach(), gradePapers(), attendMeetings()
 - HOD (Grandchild class)
 - Attributes: department
 - Operations: manageDepartment()

15. Differentiate between different Relationship types in a use case diagram.

In a use case diagram, three main types of relationships exist:

1. **Association:** This is a communication link between an actor and a use case. This shows how an actor participates in a use case.
2. **Include:** This is a directed relationship between two use cases. It is used to show that the behavior of a use case (the base use case) includes the behavior of another use case (the inclusion use case). The base use case explicitly incorporates the behavior of the inclusion use case at some point in its flow of events.
3. **Extend:** This is a directed relationship that specifies how and when the behavior defined in usually optional extending use case can be inserted into the behavior defined in the extended use case. The extension takes place at specific extension points defined in the extended use case.
4. **Generalization**



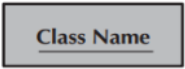




Generalization in Actors: This is a relationship between actors where one actor (the child actor) inherits the roles, responsibilities, and attributes of another actor (the parent actor). This is common in situations where some actors share a significant amount of common behavior but have some differences that set them apart. For instance, in a school system, a "Student" actor could be a generalization of "Undergraduate" and "Graduate" actors. Both undergraduates and graduates are students, but they have unique behaviors and attributes that need to be modeled separately.


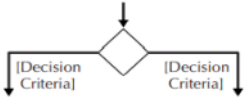
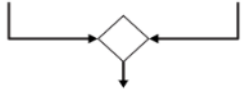


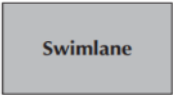
Generalization in Use Cases: This is a relationship between use cases where one use case (the child use case) inherits the behaviors and scenarios of another use case (the parent use case). Just like with actors, this is common when use cases share common behaviors but also have unique behaviors that need to be modeled separately. For example, a "Purchase Item" use case could be a generalization of "Purchase Book" and "Purchase Game" use cases. Both purchasing a book and purchasing a game involve the general behavior of purchasing an item, but they may also have unique behaviors that need to be captured.

16. Discuss any five elements of a activity diagram and identify the notation used for each of the discussed elements.

- **Activity States:** These are the fundamental units of activity or action in an activity diagram. Every activity state in an activity diagram represents a specific action that needs to be performed. They are represented as rounded rectangles.
- **Transitions:** Transitions are represented as arrows that move from one activity state to another. They show the sequence of execution of activities, i.e., once an action is executed, what will be the next action or step.
- **Decision Nodes:** Decision nodes are used to represent a test condition that helps ensure that the flow of control moves in the right direction. Based on the condition, the control will move to different parts of the diagram. These nodes are represented as diamonds.
- **Fork Nodes:** Fork nodes are used when a single flow of control (or sequence of activities) splits into multiple parallel flows. This allows for several activities to be executed in parallel. They are represented as a horizontal or vertical bar (depending on the orientation of the diagram) with one incoming arrow (representing the single incoming control flow) and multiple outgoing arrows (each representing a parallel flow of control).
- **Join Nodes:** Join nodes are represented as horizontal or vertical bars and are used to synchronize concurrent flows. They provide a synchronization mechanism in scenarios where the system's flow of control is split into multiple concurrent flows of control at a fork node. A join node has multiple incoming arrows (each representing a concurrent flow of control) and one outgoing arrow (representing the single outgoing control flow).
- **Start Nodes:** These are the starting point of any process in the activity diagram. They are represented by a filled circle.
- **End Nodes:** These are used to illustrate the termination of an activity or process. They are represented by a filled circle surrounded by a hollow circle.
- **Swimlanes:** Swimlanes are used to partition the activity diagram into different areas representing different entities (like classes, subsystems, or actors)

responsible for certain sets of activities. They are represented as rectangles containing a set of activities.

An action: <ul style="list-style-type: none"> ■ Is a simple, nondecomposable piece of behavior. ■ Is labeled by its name. 	
An activity: <ul style="list-style-type: none"> ■ Is used to represent a set of actions. ■ Is labeled by its name. 	
An object node: <ul style="list-style-type: none"> ■ Is used to represent an object that is connected to a set of object flows. ■ Is labeled by its class name. 	
A control flow: <ul style="list-style-type: none"> ■ Shows the sequence of execution. 	
An object flow: <ul style="list-style-type: none"> ■ Shows the flow of an object from one activity (or action) to another activity (or action). 	
An initial node: <ul style="list-style-type: none"> ■ Portrays the beginning of a set of actions or activities. 	
A final-activity node: <ul style="list-style-type: none"> ■ Is used to stop all control flows and object flows in an activity (or action). 	

A final-flow node: <ul style="list-style-type: none"> ■ Is used to stop a specific control flow or object flow. 	
A decision node: <ul style="list-style-type: none"> ■ Is used to represent a test condition to ensure that the control flow or object flow only goes down one path. ■ Is labeled with the decision criteria to continue down the specific path. 	
A merge node: <ul style="list-style-type: none"> ■ Is used to bring back together different decision paths that were created using a decision node. 	
A Fork node: <ul style="list-style-type: none"> Is used to split behavior into a set of parallel or concurrent flows of activities (or actions). 	
A Join node: <ul style="list-style-type: none"> Is used to bring back together a set of parallel or concurrent flows of activities (or actions). 	
A Swimlane: <ul style="list-style-type: none"> Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action). Is labeled with the name of the individual or object responsible. 	

17. Create and justify an inheritance hierarchy that could be used to represent the following classes: Animal, Mammal, Bird, Reptile, Dog, Eagle, Snake.

Implement the above inheritance hierarchy using Java. Write the code for each class and include the necessary methods as described below:

1. **Animal** class:

- Method: **void move()** - prints "Animal is moving."
- Method: **void ()** - prints "Animal is sleeping."

2. **Amphibian** class:

- Method: **void habitat()** – prints “ lives in land and in water”

1. **Mammal** class :

- Method: **void giveBirth()** - prints "Mammal is giving birth."

2. **Bird** class :

- Method: **void layeggs()** - prints "Birds lay eggs."

3. **Reptile** class

- Method: **void crawl()** - prints "Reptile is crawling."

4. **Dog** class

- Method: **void petanimal()** - prints "Dog is a good pet animal."

5. **Eagle** class (inherits from **Bird**):

- Method: **void fly()** - prints "Eagle flies very high."

6. **Snake** class (inherits from **Reptile**):

- Method: **void bite()** - prints "Snake bite is dangerous."

Additionally, create a **main** method in a Main class to demonstrate the behavior of the different animal types. Inside the **main** method . Create an instance of the **Dog** class And Create an instance of the **snake** class. Call the methods of these two classes using the objects created.

```

class Animal_ESE {
    void move() {
        System.out.println("Animal is moving.");
    }

    void sleep() {
        System.out.println("Animal is sleeping.");
    }
}

class Amphibian_ESE extends Animal_ESE {
    void habitat() {
        System.out.println("Lives in land and in water");
    }
}

class Mammal_ESE extends Animal_ESE {
    void giveBirth() {
        System.out.println("Mammal is giving birth.");
    }
}

class Bird_ESE extends Animal_ESE {
    void layEggs() {
        System.out.println("Birds lay eggs.");
    }
}

class Reptile_ESE extends Animal_ESE {
    void crawl() {
        System.out.println("Reptile is crawling.");
    }
}

class Dog_ESE extends Mammal_ESE {
    void petAnimal() {
        System.out.println("Dog is a good pet animal.");
    }
}

```

```

}

class Eagle_ESE extends Bird_ESE {
    void fly() {
        System.out.println("Eagle flies very high.");
    }
}

class Snake_ESE extends Reptile_ESE {
    void bite() {
        System.out.println("Snake bite is dangerous.");
    }
}

class Main_ESE {
    public static void main(String[] args) {
        Dog_ESE dog = new Dog_ESE();
        Snake_ESE snake = new Snake_ESE();
        dog.move();
        dog.sleep();
        dog.giveBirth();
        dog.petAnimal();
        snake.move();
        snake.sleep();
        snake.crawl();
        snake.bite();
    }
}

```