50.003: Elements of Software Construction
Problem Set 1
Tan Ze Xin Dylan (1004385)

Question 1:
A railway network consists of several tracks, junction points and trains. A train moves along the track until it requires changing the track to reach destination. The changes of track occur only through a junction point. Each train is initially positioned at a junction point and its destination is a different junction point. Hence, while a train approaches its destination junction point, it does not need to change tracks any more. Each track can have at most one train at a time to avoid collision, but all tracks in the railway network are bi-directional. Design a set of APIs (application programming interface) for a safe (i.e. collision free) system. No coding is required, the APIs and their purpose will suffice.

```
public class Train {
   String trainID;
   double travelSpeed;
   Junction finalDestination;
   List<Junction> travelRoute; // Contains list of junctions
   Track currentTrack;
   Track nextTrack;
   double currentPosition; // Distance from the start of currentTrack
   double distanceLeft; // Distance left to end of currentTrack

   public double getDistanceLeft(Track currentTrack, double travelSpeed, double
currentPosition){
      // gets the distance left to the end of the currentTrack
      return distanceLeft;
   }
   public void getCurrentTrack(){
      // gets the current track that the train is on
   }
   public void findNextTrack(){
      // Updates nextTrack based on travelRoute and currentPosition
   }
   public void updateParams(){
      // Updates the all the attributes
   }
}
public class Track {
   String trackID;
   double length;
}
public class Junction {
   String junctionID;
   boolean checkOccupied;
   double x,y;
   List<Double> position = Arrays.asList(x,y);
```

```
    public boolean checkOccupied(){
        // returns True if occupied, otherwise False.
        return checkOccupied;
    }
}
public class Scheduler {
    // Contains a list of all the trains
    List<Train> allTrains;
    String reason;
    // Linked list to store potential collisions
    List<List<String>> potentialCollisionList = new LinkedList<>();

    public void updateByTime(){
        // use moveTrain() method to move all trains
        // uses getUpdates() and checkPotentialCollision() method every few seconds
        // use trainPriority() method to schedule train.
    }
    public void getUpdates(){
        // The method loops through all the Trains in the allTrains list
        // Invokes the UpdateParams() method in Train class
    }
    public void checkPotentialCollision(){
        // Check for potential collisions based on speed, next track and immediate destination
        // if next trackIDs for 2 trains are the same, add to collision list
        // if immediate destination for 2 trains are the same and distance left for at least
        // one train is lower than a specified amount, add to collision list
        // Appends a List containing both potentially colliding trains with the reason,
        // to the potentialCollisionList.
        // If reason is due to immediate destination being the same,
        // append the train that is closer first
    }
    public void stopTrain(){
        // Sends a signal to top moving the train as soon as possible
    }
    public void moveTrain(){
        // Sends signal to move the train
    }
    public void trainPriority(){
        // For each potential collision in the potentialCollisionList,
        // Check for the reason for potential collision.
        // if reason is due to same next track,
        // randomly assign a train to stop and a train to move, using the stopTrain() or
moveTrain() methods
        // if reason is due to immediate destination being the same,
        // use stopTrain() method to stop the 2nd train.
    }
}
```

Question 2:
Consider the railway network from previous exercise. Assume tracks of different type – broad gauge, meter gauge and narrow gauge. Each junction point is further divided into individual establishments that exclusively handle a specific track (i.e. meter, broad or narrow). Similarly, trains for meter gauge track is different from trains for broad gauge and narrow gauge. Narrow gauge trains are not powerful to run longer than the distance between two junction points. At each junction point, therefore, its engine is changed. <u>Refine your set of APIs to capture this system.</u>

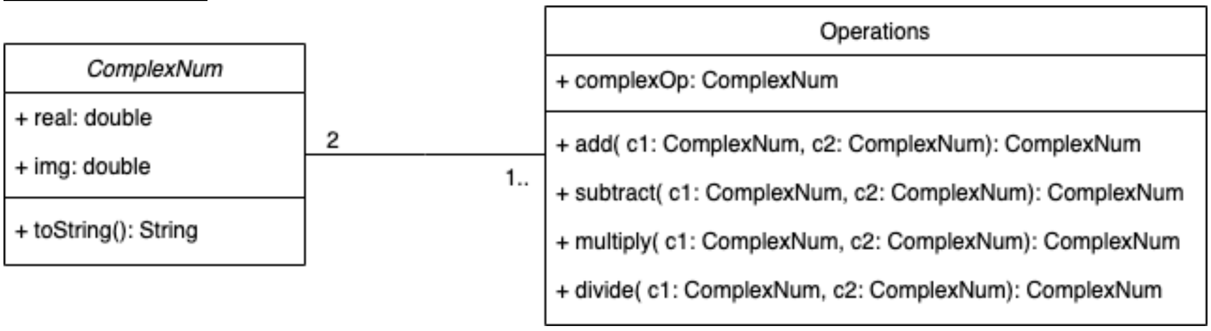
```
public class Train {
   String trainID;
   double travelSpeed;
   Junction finalDestination;
   List<Junction> travelRoute; // Contains list of junctions
   Track currentTrack;
   Track nextTrack;
   double currentPosition; // Distance from the start of currentTrack
   double distanceLeft; // Distance left to end of currentTrack
   // Qns 2
   String trainType;
   public double getDistanceLeft(Track currentTrack, double travelSpeed, double
currentPosition){
      // gets the distance left to the end of the currentTrack
      return distanceLeft;
   }
   public void getCurrentTrack(){
      // gets the current track that the train is on
   }
   public void findNextTrack(){
      // Updates nextTrack based on travelRoute and currentPosition
   }
   public void updateParams(){
      // Updates the all the attributes
   }
   // Qns 2
   public void changeEngine(String trainType){
      // If train type is narrow gauge, changes engine when a junction point is reached
   }
}

public class Track {
   String trackID;
   double length;
   // Qns 2
   String trackType;
}
```

```
public class Junction {
   String junctionID;
   boolean checkOccupied;
   double x,y;
   List<Double> position = Arrays.asList(x,y);
   public boolean checkOccupied(){
      // returns True if occupied, otherwise False.
      return checkOccupied;
   }
   // Qns 2
   public void changeTrack(){
      // checks for trainType and changes the trackType accordingly
   }
}

public class Scheduler {
   // Contains a list of all the trains
   List<Train> allTrains;
   String reason;
   // Linked list to store potential collisions
   List<List<String>> potentialCollisionList = new LinkedList<>();

   public void updateByTime(){
      // use moveTrain() method to move all trains
      // uses getUpdates() and checkPotentialCollision() method every few seconds
      // use trainPriority() method to schedule train.
      // invoke changeTrack() method to change tracks at the junctions as required by
the trains
   }

   public void getUpdates(){
      // The method loops through all the Trains in the allTrains list
      // Invokes the UpdateParams() method in Train class
   }

   public void checkPotentialCollision(){
      // Check for potential collisions based on speed, next track and immediate destination
      // if next trackIDs for 2 trains are the same, add to collision list
      // if immediate destination for 2 trains are the same and distance left for at least
      // one train is lower than a specified amount, add to collision list
      // Appends a List containing both potentially colliding trainIDs with the reason,
      // to the potentialCollisionList.
      // If reason is due to immediate destination being the same,
      // append the train that is closer first
   }
   public void stopTrain(){
      // Sends a signal to top moving the train as soon as possible
   }
```

```
    public void moveTrain(){
        // Sends signal to move the train
    }
    public void trainPriority(){
        // For each potential collision in the potentialCollisionList,
        // Check for the reason for potential collision.
        // if reason is due to same next track,
        // randomly assign a train to stop and a train to move, using the stopTrain() or
moveTrain() methods
        // if reason is due to immediate destination being the same,
        // use stopTrain() method to stop the 2nd train.
    }
}
```

Question 3:
Design and implement a program that supports accepting two complex numbers from the user; adding, subtracting, multiplying, and/dividing them; and reporting each result to the user. Show both the class diagram and the actual code.

Class Diagram

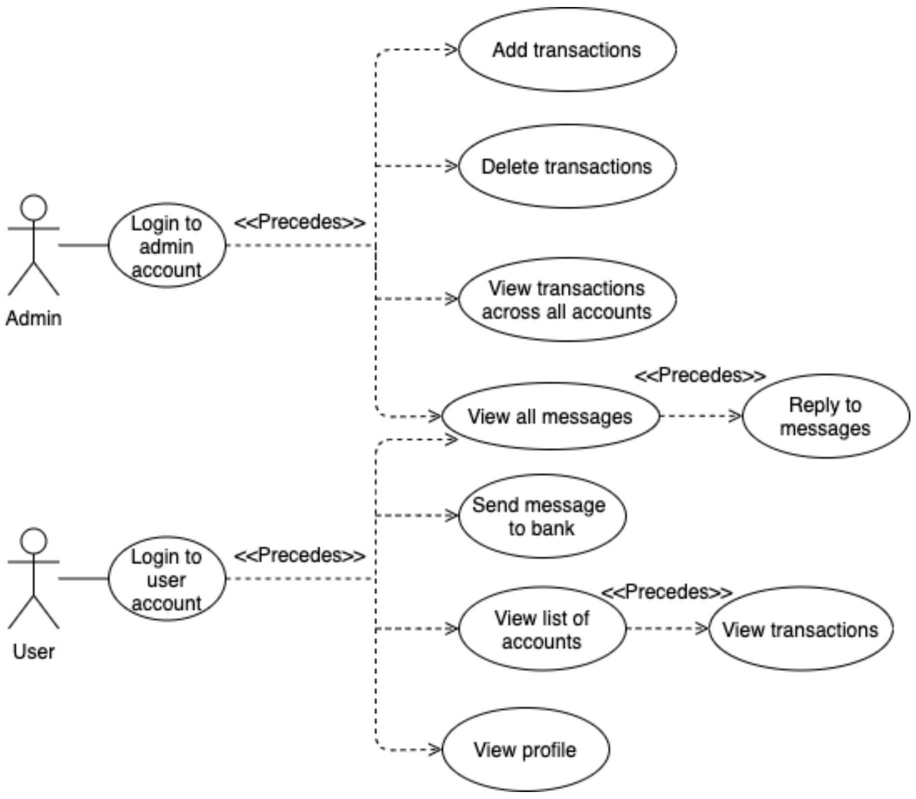


ComplexNum
+ real: double
+ img: double
+ toString(): String
2
1..
Operations
+ complexOp: ComplexNum
+ add(c1: ComplexNum, c2: ComplexNum): ComplexNum
+ subtract(c1: ComplexNum, c2: ComplexNum): ComplexNum
+ multiply(c1: ComplexNum, c2: ComplexNum): ComplexNum
+ divide(c1: ComplexNum, c2: ComplexNum): ComplexNum

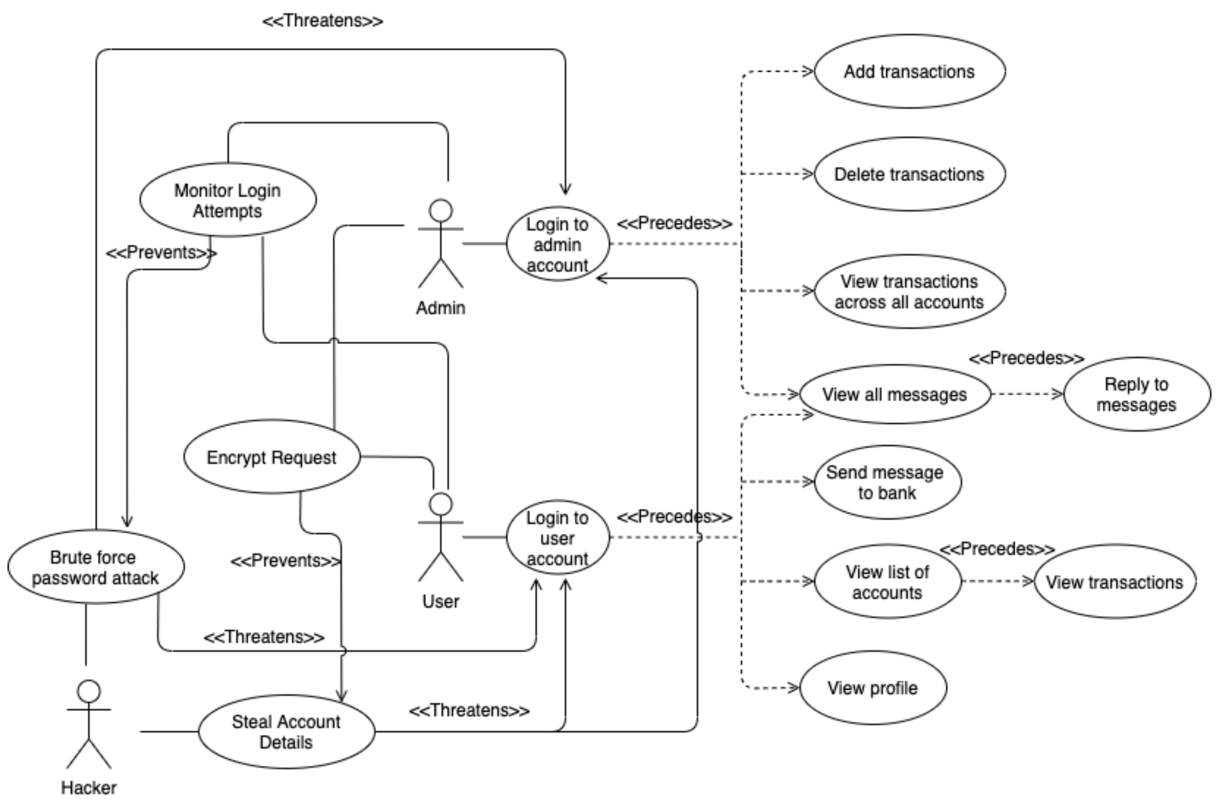
Actual Code

```
public class MyClass {
   public static void main(String args[]) {
      ComplexNum c1 = new ComplexNum(20, -4);
      ComplexNum c2 = new ComplexNum(3, 2);
      ComplexNum temp = new ComplexNum();

      System.out.println("First Complex Num: " + c1);
      System.out.println("Second Complex Num: " + c2);

      // Addition
      temp = Operations.add(c1, c2);
      System.out.println(temp);

      // Subtraction
      temp = Operations.subtract(c1, c2);
      System.out.println(temp);

      // Multiplication
      temp = Operations.multiply(c1, c2);
      System.out.println(temp);

      // Division
      temp = Operations.divide(c1, c2);
      System.out.println(temp);
   }
}

public class ComplexNum {
   double real, img;

   ComplexNum(){
      new ComplexNum(0,0);
   }

   ComplexNum(double real, double img){
        this.real = real;
        this.img = img;
    }

    public String toString(){
        if (this.img < 0){
            return this.real + " - " + (-this.img) + "i";
        }
        else if (this.img > 0) {
            return this.real + " + " + this.img + "i";
        }
        else return this.real + "";
    }
}

public class Operations {
    static ComplexNum complexOp = new ComplexNum();

    public static ComplexNum add(ComplexNum c1, ComplexNum c2)
    {
        complexOp.real = c1.real + c2.real;
        complexOp.img = c1.img + c2.img;

        return complexOp;
    }
    public static ComplexNum subtract(ComplexNum c1, ComplexNum c2)
    {
        complexOp.real = c1.real - c2.real;
        complexOp.img = c1.img - c2.img;

        return complexOp;
    }
    public static ComplexNum multiply(ComplexNum c1, ComplexNum c2)
    {
        // (a+bi)(c+di) = (ac-bd) + (ad+bc)i
        complexOp.real = (c1.real*c2.real)-(c1.img*c2.img);
        complexOp.img = (c1.real*c2.img) + (c1.img*c2.real);

        return complexOp;
    }

public static ComplexNum divide(ComplexNum c1, ComplexNum c2)
    {
        // multiply by the conjugate
        double denominator;
        // (a+bi)(a-bi) = a^2 + b^2
        denominator = Math.pow(c2.real,2) + Math.pow(c2.img,2);
        complexOp.real = ((c1.real*c2.real)+(c1.img*c2.img))/denominator;
        complexOp.img = (-(c1.real*c2.img)+(c1.img*c2.real))/denominator;
        return complexOp;
    }
}
```

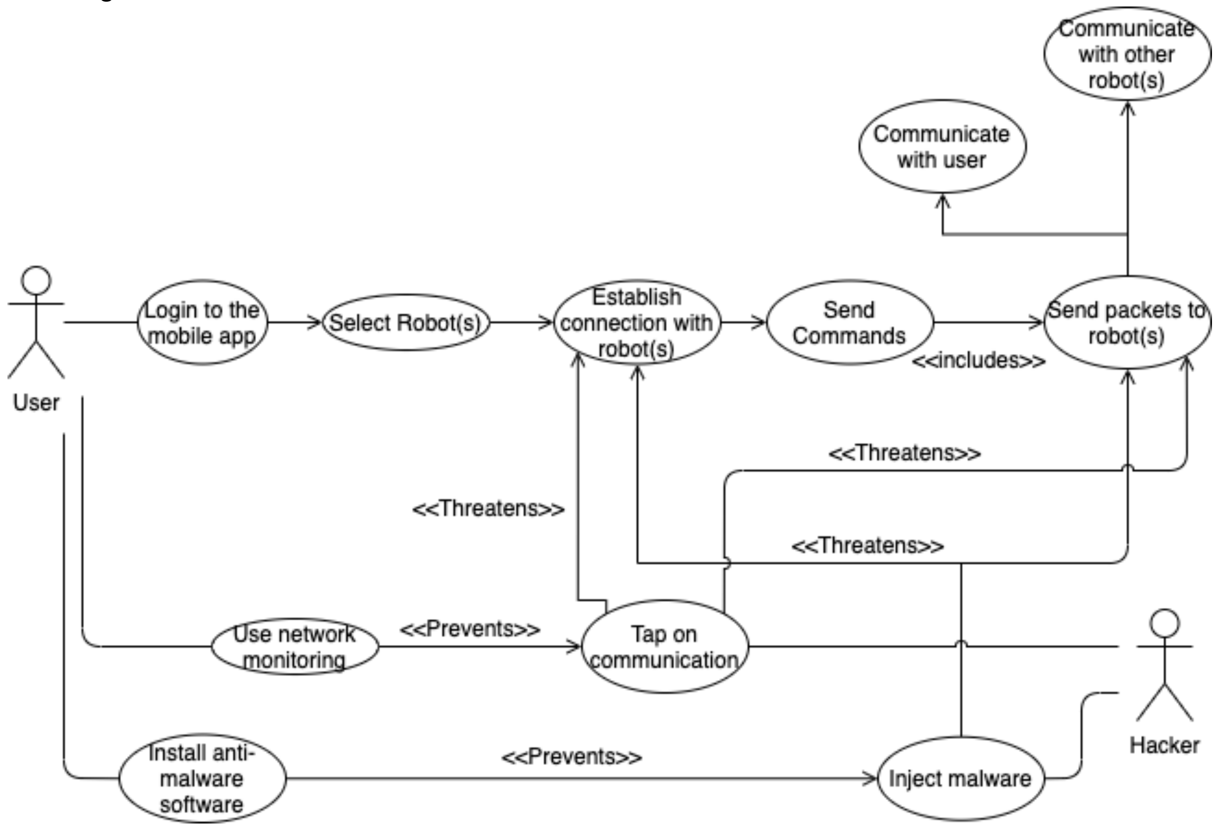
Question 4:
Draw a user case diagram for KBO (the online banking system discussed in the class). Use only “precedes” relationship.



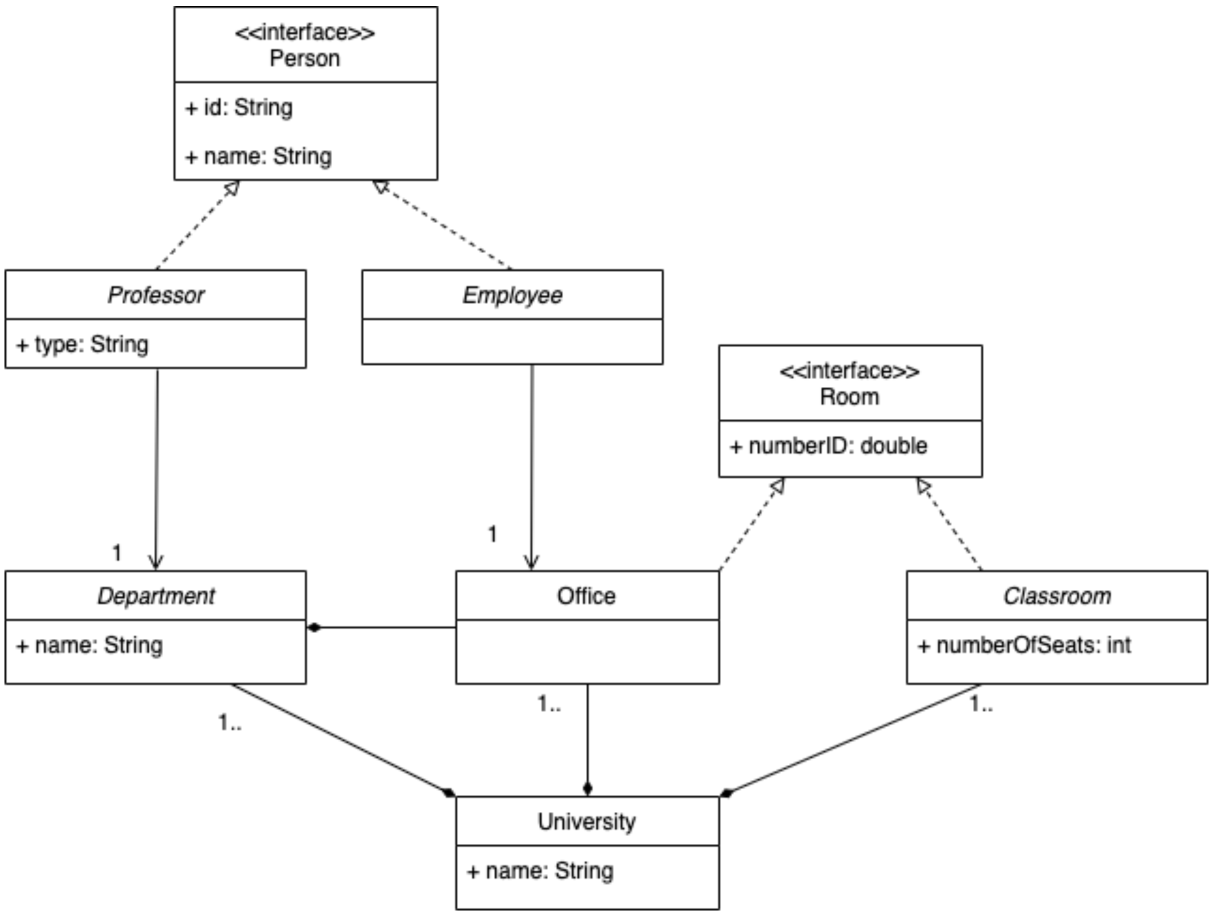
Question 5:
Augment the use case diagram for KBO with at least two misuse cases and additional use cases to prevent the misuse.



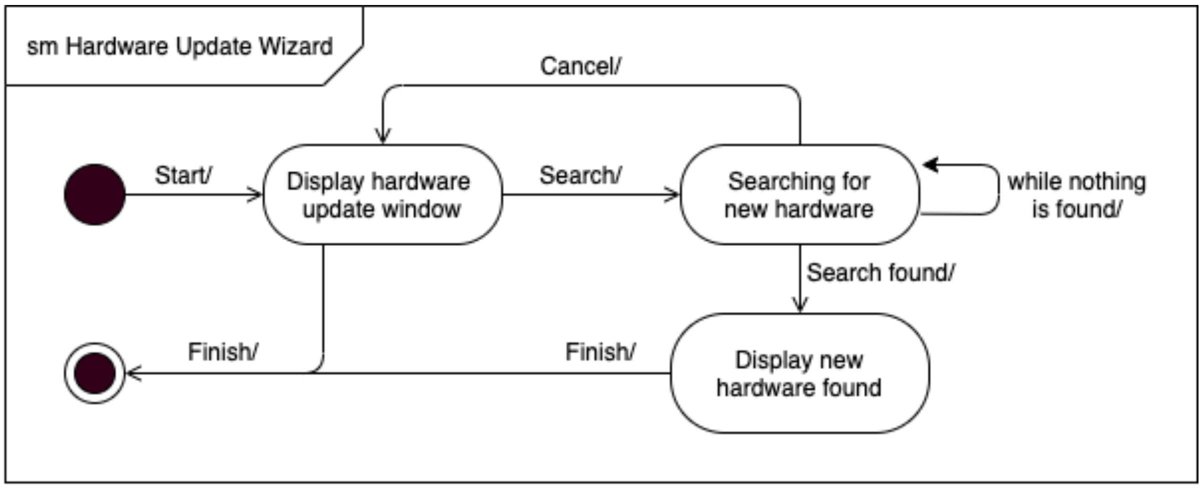
Question 6:
Consider a swarm of robots where a user can control any robot via mobile. Each robot can communicate to the user or to another robot. A hacker can affect the system by injecting a malware in the mobile or tapping on the communication between robots (e.g. modifying or delaying packets sent through the network). Draw the combined use&misuse case diagram for the system. Integrate anti---malware and network monitoring as security solutions for blocking attackers.



Question 7:
Draw a class diagram for the following scenario. In a university there are different classrooms, offices and departments. A department has a name and it contains many offices. A person working at the university has a unique ID and can be a professor or an employee. • A professor can be a full, associate or assistant professor and he/she is enrolled in one department. • Offices and classrooms have a number ID, and a classroom has a number of seats. • Every employee works in an office.



Question 8:
A hardware update wizard can be in three states as follows: 1. Displaying a hardware update window. 2. Searching for new hardware. 3. Displaying new hardware found. The wizard starts by displaying a hardware update window. While displaying this window, the user can press a "Search" button to cause the wizard to start searching for new hardware, or the user can press a "Finish" button to leave the wizard. While the wizard is searching for new hardware, the user may cancel the search at any time. If the user cancels the search, the wizard displays the hardware update window again. When the wizard has completed searching for new hardware, it displays the new hardware found. Draw a <u>state machine diagram</u> that represents the function of the hardware update wizard just described.


sm Hardware Update Wizard
Cancel/
Start/
Display hardware update window
Search/
Searching for new hardware
while nothing is found/
Search found/
Finish/
Finish/
Display new hardware found