VDM SPECIFICATION DOCUMENT

Formal Methods in Software Engineering (SE-313)

Department of Software
Engineering, NED
University of Engineering
&
Technology, Karachi

Roll No	SE-21077
Name	Ahmed Gala
Batch	2021
Year	2023-2024
Department	Software Engineering

ELEVATOR SYSTEM

Scope of the system:

The scope of the Elevator System encompasses user-initiated requests for calling the elevator and selecting the destination floor. Users can effortlessly request the elevator and specify the floor they wish to reach. The system, in turn, is designed to respond to user requests by evaluating whether they meet the required conditions. Subsequently, the system signals the elevator to move either upward or downward, depending on its current position and the designated destination floor.

The hardware component of the system autonomously executes the physical movement of the elevator and communicates its actions to the software. The system receives feedback from the hardware, recording any changes in the elevator's position. Furthermore, the software instructs the elevator to continue moving or come to a halt based on the received feedback.

Additionally, the system oversees the control of elevator movement, monitors its trajectory, and manages door operations. Specifically, when the elevator comes to a stop, the doors are programmed to open, and during movement, they remain closed. The constraint on the system is that the number of floors ranges from 0 to 15. The primary focus of the system is on essential functionalities, with a particular emphasis on seamless user interaction and efficient elevator management within the building.

4+1 Architectural View Model:

The 4+1 architectural view model, also known as Kruchten's architectural model, is used for describing the architecture of software-intensive systems, based on the use of multiple, concurrent views. The views address specific concerns of stakeholders, such as designers, developers, and end-users, providing a comprehensive understanding of the system's structure, behaviour, deployment, and development. The four views of the model are:

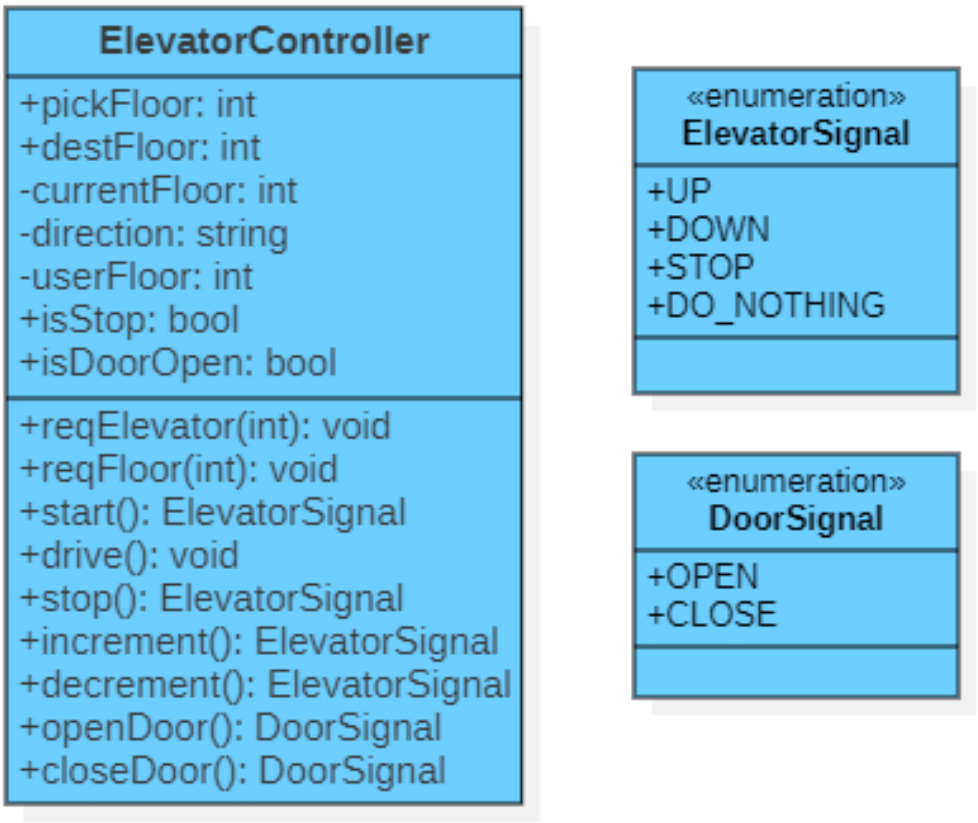
- Logical view.
- Development view.
- Process view
- Physical view.

In addition to that, it includes an additional perspective that encompasses selected scenarios or use cases to illustrate the architecture serving as the 'plus one' view.

Following are the UML diagrams for each of the views.

Logical View:

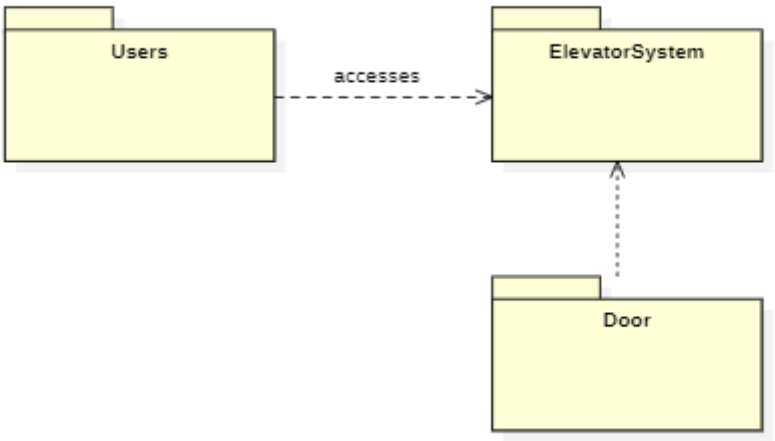
The **Class Diagram** is used to represent the logical view of the Elevator System as it represents the high-level organization of the system in terms of classes and their relationships. It focuses on the static structure of the software.



The ElevatorController class shows all of its attributes and methods. In it, there are two different signal outputs, ElevatorSignal and DoorSignal which are not standard UML types such as an integer, therefore they had to be represented separately. A type that consists of a small number of named values is known as an *enumerated type,* and so this is the standard way of marking a UML class as an enumerated type.

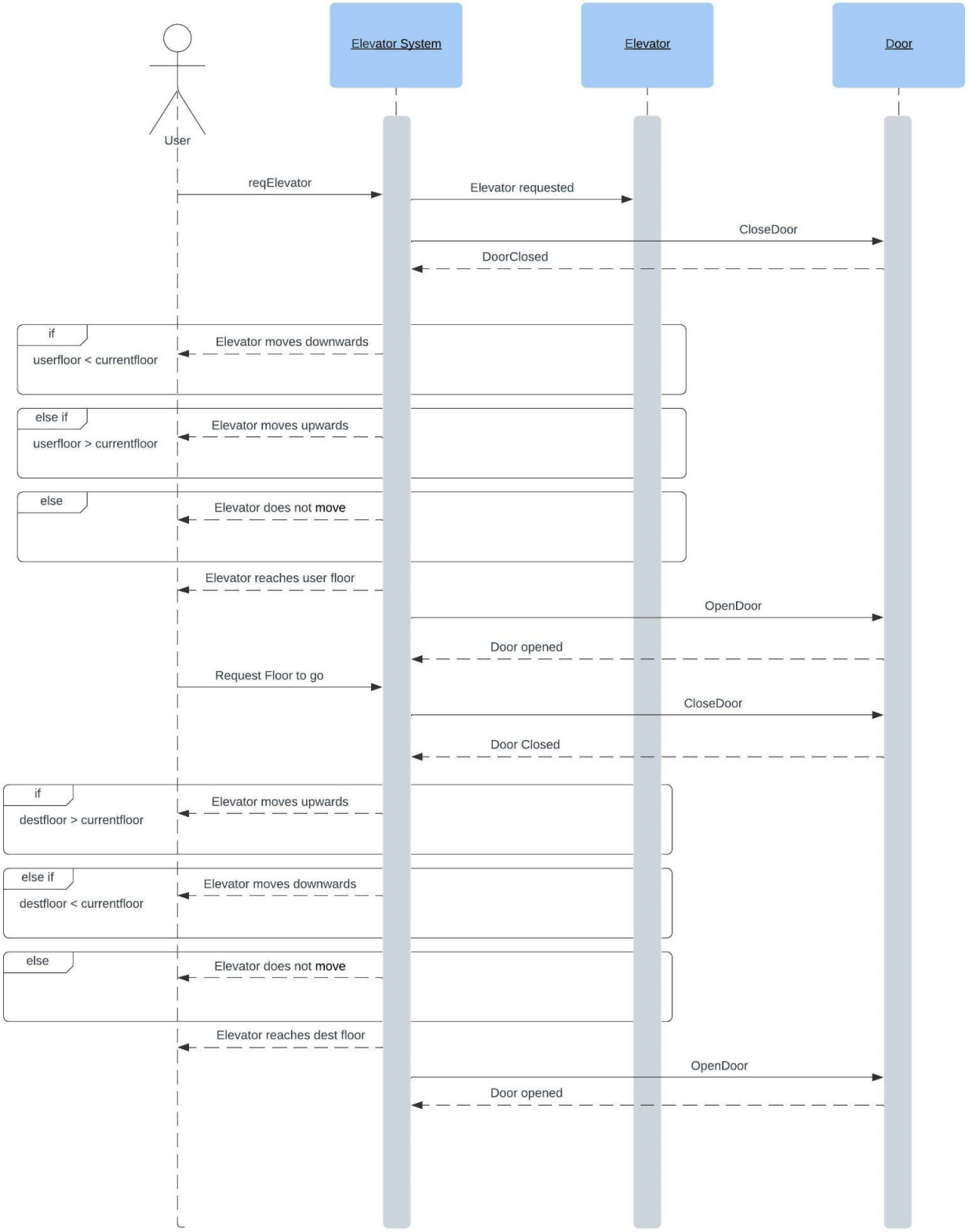
Development View:

To illustrate the development view of the elevator system, a **Package diagram** is used. They focus on organizing and depicting the high-level structure of software modules or components during the development phase. Package diagrams show how various elements are grouped into packages, illustrating the dependencies and relationships between them.



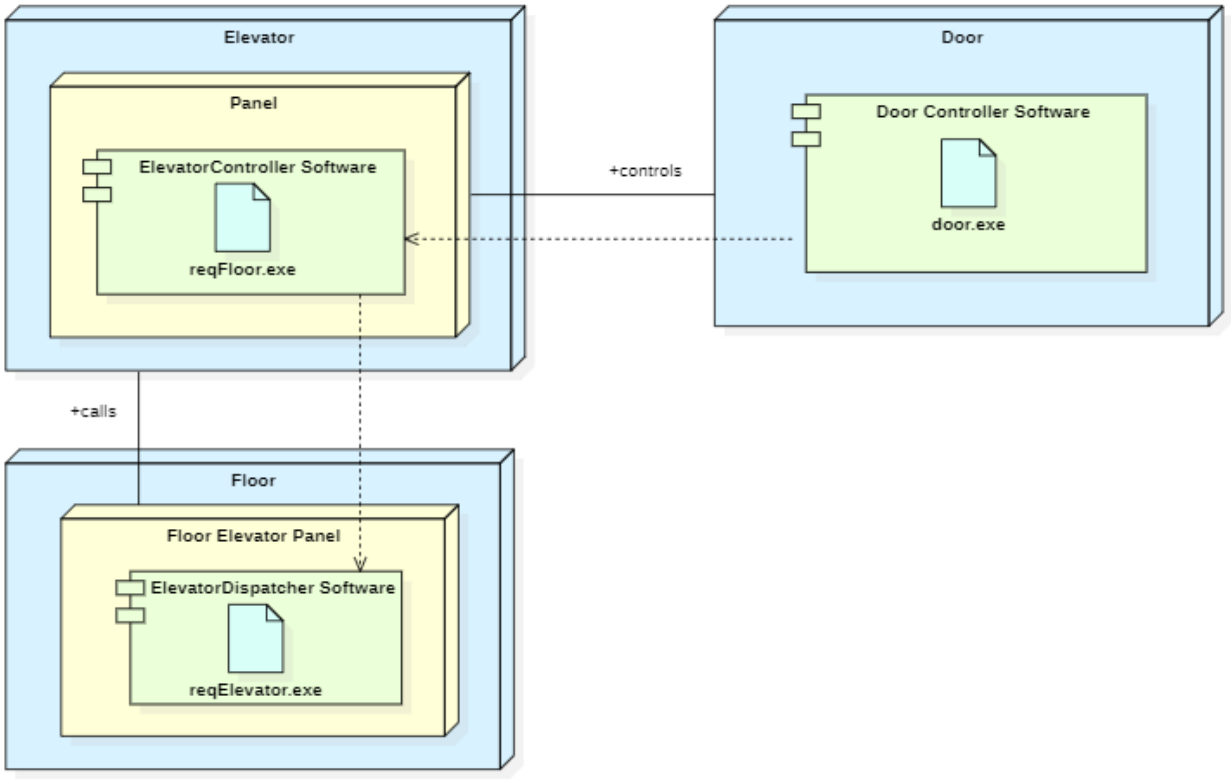
Process view:

Process view can be illustrated through the **Sequence diagram**. Sequence diagrams can represent the dynamic aspects of the system, showcasing the runtime behavior and the sequence of activities during specific scenarios or use cases.


Elevator System
Elevator
Door
User
reqElevator
Elevator requested
CloseDoor
DoorClosed
if
userfloor < currentfloor
Elevator moves downwards
else if
userfloor > currentfloor
Elevator moves upwards
else
Elevator does not move
Elevator reaches user floor
OpenDoor
Door opened
Request Floor to go
CloseDoor
Door Closed
if
destfloor > currentfloor
Elevator moves upwards
else if
destfloor < currentfloor
Elevator moves downwards
else
Elevator does not move
Elevator reaches dest floor
OpenDoor
Door opened

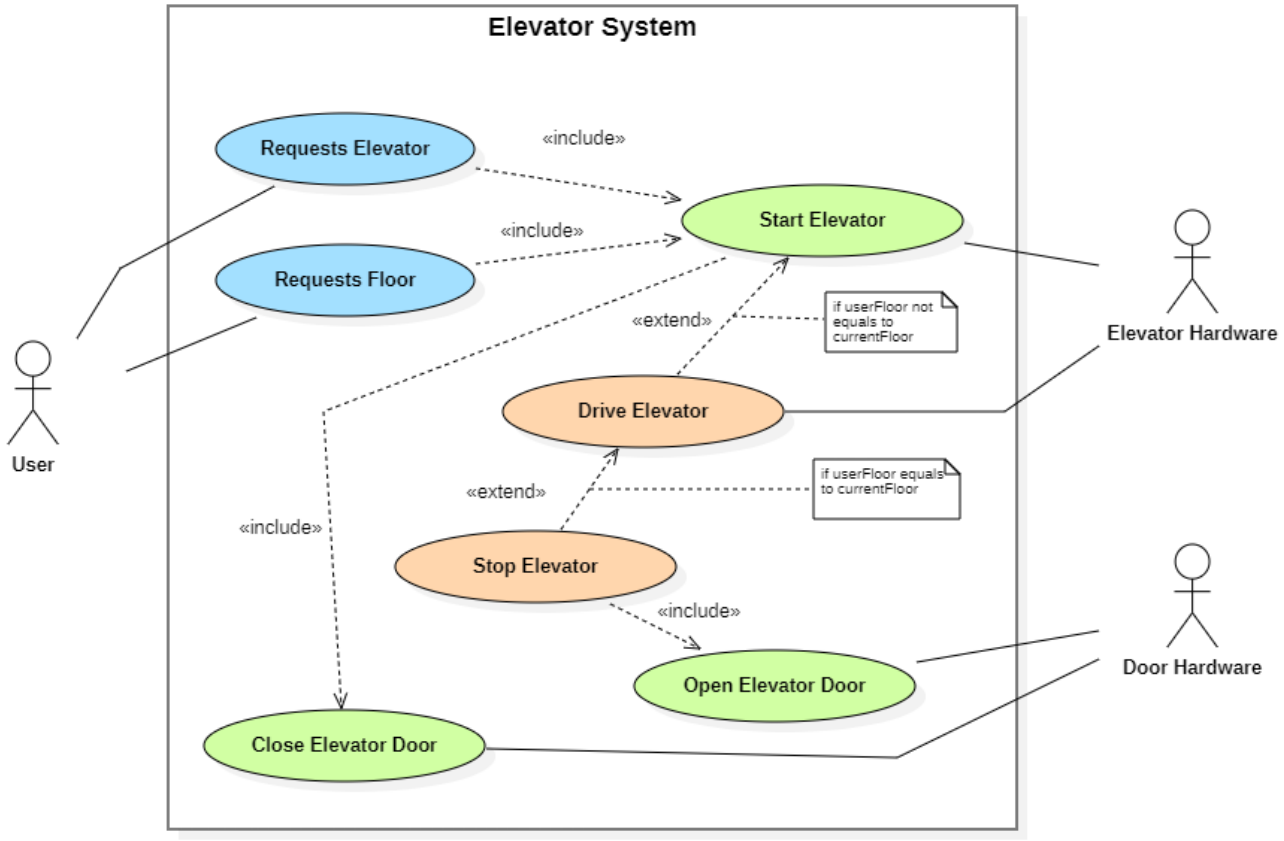
Physical View:

Deployment Diagram is used to depict the physical view as it illustrates the physical deployment of software components on hardware, including servers, nodes, and networks. It provides insights into the system's distribution and scalability.



Scenarios (or Use Cases) View:

Use Case diagram presents a set of scenarios or use cases that describe interactions between users and the system. This view is centred around capturing functional requirements from the end-users' perspective.



The complete VDM-SL of the Elevator System:

types
ElevatorSignal = <UP> | <DOWN> | <STOP> | <DO_NOTHING>
DoorSignal = <OPEN> | <CLOSE>
String = Char*

values

--the elevator system has 0 to 15 floors.

MAX: ℕ = 15
MIN: ℕ = 0

state *Elevator* **of**

pickFloor : ℕ
destFloor : ℕ
currentFloor : ℕ
direction : String
userFloor : ℕ
isStop : 𝔹
isDoorOpen : 𝔹

-- all floor values must be in the range
inv mk-*Elevator* (*p, d, c, u*) ≜ (*inRange*(*p*) ∧ *inRange*(*d*) ∧ *inRange*(*c*) ∧ *inRange*(*u*))

-- the floor values shall be zero and isDoorOpen and isStop shall be false when the system is initialized
init mk-*Elevator* (*p, d, c, u, d, s*) ≜ *p* = 0 ∧ *d* = 0 ∧ *c* = 0 ∧ *u* = 0 ∧ *d* = FALSE ∧ *s* = FALSE

end

functions
inRange(*val* : ℕ) *result* : 𝔹
pre TRUE
post *result* ⇔ *MIN* ≤ *val* ≤ *MAX*

operations

-- an operation that requests the elevator to reach the floor at which the user requested the elevator

reqElevator(*floor* : ℕ)
ext wr *pickFloor* : ℕ
rd *userFloor* : ℕ
pre *inRange*(*floor*)
post *pickFloor* = *floor* ∧ *userFloor* = *floor* ∧ *start*()

-- an operation that records the floor user wishes to go to and initiates the request to the Elevator System

reqFloor(*floor* : ℕ)
ext wr *destFloor* : ℕ
rd *userFloor* : ℕ
pre *inRange*(*floor*)

post *destFloor* = *floor* ∧ *userFloor* = *floor* ∧ *closeDoor*() ∧ *start*()

-- an operation that controls the movement of the elevator by instructing the hardware

start() *signalOut* : *ElevatorSignal*
ext rd *userFloor* : ℕ
rd *currentFloor* : ℕ
rd *direction* : *String*
rd *isStop* : B
rd *isDoorOpen* : B
pre isStop = TRUE ∧ *isDoorOpen* = FALSE
post (*userFloor* = *currentFloor* ∧ *openDoor*() ∧ *signalOut* = <DO_NOTHING>
∨ *userFloor* < *currentFloor* ∧ *direction* = *'down'* ∧ *drive*() ∧ *signalOut* = <DOWN>
∨ *userFloor* > *currentFloor* ∧ *direction* = *'up'* ∧ *drive*() ∧ *signalOut* = <UP>)

-- an operation that manages the logic of the movement of the elevator

drive()
ext rd *currentFloor* : ℕ
rd *userFloor* : ℕ
rd *direction* : *String*
rd *isStop* : B
pre *userFloor* ≠ *currentFloor*
post *isStop* = FALSE ∧
(*direction* = *'up'* ∧ *increment*()
∨ *direction* = *'down'* ∧ *decrement*()
∨ *userFloor* = *currentFloor* ∧ *stop*())

-- an operation that instructs the hardware to stop the elevator when required

stop() *signalOut* : *ElevatorSignal*
ext rd *currentFloor* : ℕ
rd *userFloor* : ℕ
rd *isStop* : B
pre *isStop* = FALSE ∧ *userFloor* = *currentFloor*
post *isStop* = TRUE ∧ *openDoor*() ∧ *signalOut* = <STOP>

-- an operation that records an increase in floor number by 1.

increment() *signalOut* : *ElevatorSignal*
ext wr *currentFloor* : ℕ
rd *userFloor* : ℕ

pre *userFloor* > *currentFloor*
post *currentFloor* = $\overline{currentFloor}$ + *1* ∧ *signalOut* = <UP>

-- an operation that records a decrease in floor number by 1.

decrement() *signalOut* : *ElevatorSignal*
ext wr *currentFloor* : ℕ
rd *userFloor* : ℕ
pre *userFloor* < *currentFloor*
post *currentFloor* = $\overline{currentFloor}$ - *1* ∧ *signalOut* = <DOWN>

-- an operation that opens the elevator door by instructing the hardware

openDoor() *signalOut* : *DoorSignal*
ext wr *isDoorOpen* : B
rd *isStop* : B
pre *isDoorOpen* = FALSE ∧ *isStop* = TRUE
post *isDoorOpen* = TRUE ∧ *signalOut* = <OPEN>

-- an operation that closes the elevator door by instructing the hardware

closeDoor() *signalOut* : *DoorSignal*
ext wr *isDoorOpen* : B
pre TRUE
post *isDoorOpen* = FALSE ∧ *signalOut* = <CLOSE>

Java Implementation of the VDM Specification:

Translating ElevatorSignal type into Java:

ElevatorSignal = <UP> | <DOWN> | <STOP> | <DO_NOTHING>

Java Implementation:

```
class ElevatorSignal {

    //one variable that will be used
```

```
    private int value;


    //initializing objects inside the class with variables names that
correspond to VDM-SL
    public static final ElevatorSignal UP = new ElevatorSignal(0);
    public static final ElevatorSignal DOWN = new ElevatorSignal(1);
    public static final ElevatorSignal STOP = new ElevatorSignal(2);
    public static final ElevatorSignal DO_NOTHING = new
ElevatorSignal(3);

    //constructor to set the value of the variable value
    private ElevatorSignal(int v){
        value = v;
    }

    public String toString(){

        switch (value) {
            case 0:
                return "SIGNAL: UP";

            case 1:
                return "SIGNAL: DOWN";

            case 2:
                return "SIGNAL: STOP";

            default:
                return "SIGNAL: DO_NOTHING";
        }
    }

};
```

Translating DoorSignal type into Java:

DoorSignal = <OPEN> | <CLOSE>

Java Implementation:

```
class DoorSignal {

    private int value;

    public static final DoorSignal OPEN = new DoorSignal(0);
    public static final DoorSignal CLOSE = new DoorSignal(1);

    private DoorSignal(int x){
        value = x;
    }

    public String toString(){

        switch (value) {
            case 0:
                return "SIGNAL: OPEN";

            default:
                return "SIGNAL: CLOSE";
        }
    }

};
```

Translating values into Java:

values
--the elevator system has 0 to 15 floors.
MAX: ℕ = 15
MIN: ℕ = 0

Java Implementation:

```
public static final int MAX = 15; //represents highest floor
public static final int MIN = 0; //represents lowest floor
```

Translating state in Java:

state *ElevatorController* **of**

pickFloor : ℕ
destFloor : ℕ
currentFloor : ℕ
direction : String
userFloor : ℕ
isStop : $\mathbb{B}$
isDoorOpen : B

end

Java Implementation:

```
public class ElevatorController {


       //----x-x-x-x-x-x-x-x- VDM STATE -x-x-x-x-x-x-x--x-x--//

//all state variables are private variables which can be accessed only
within the class.

    //Natural number in VDM is converted to int in JAVA
    private int pickFloor;
    private int destFloor;
    private int currentFloor;
    private String direction;
    private int userFloor;
    private boolean isStop;
    private boolean isDoorOpen;

}
```

Translating Initialization clause in Java:

init mk-*ElevatorController* $(p, d, c, u, d, s) \triangleq p = 0 \wedge d = 0 \wedge c = 0 \wedge u = 0 \wedge d = \text{FALSE} \wedge s = \text{FALSE}$

Java Implementation

```
    public ElevatorController() {
        this.pickFloor = 0;
```

```
        this.destFloor = 0;
        this.currentFloor = 0;
        this.direction = ""; // Add a default value for String
        this.userFloor = 0;
        this.isStop = true;
        this.isDoorOpen = false;

        VDM.invTest(this);
    }
```

Translating Invariant clause in Java:

inv mk-*ElevatorController* $(p, d, c, u) \triangleq (inRange(p) \wedge inRange(d) \wedge inRange(c) \wedge inRange(u))$

Java Implementation

```
    public boolean inv (){
        return (inRange(pickFloor) && inRange(destFloor)
        && inRange(userFloor) && inRange(currentFloor));
    }
```

Here, the invariant function is defined that exactly translates the VDM specification.

```
public class ElevatorController implements InvariantCheck {

//other code

}

interface InvariantCheck{
    public boolean inv();
}
```

The above code shows the implementation of the interface and how it is used with the class.

Translating pre-condition in Java:

eg.
pre *isStop* = FALSE ∧ *userFloor* = *currentFloor*

Java Implementation

```
public class VDM {

    public static void preTest (boolean exp){
        if(!exp){
            throw new VDMException("Precondition not Satisfied.");
        }
    }
}
```

A VDM Class is made that has the function preTest that tests the precondition, and throws a VDMException if the precondition is violated.

Whenever the precondition has to be tested it will be called like this:

```
Eg.
VDM.preTest(isStop == false && userFloor == currentFloor);
```

Application of invariant check in Java:

Java Implementation

```
public static void invTest (ElevatorController elevator) {

        boolean invariant = elevator.inv(); //this function checks
whether the invariant is satisfied and then returns a boolean.
        if(!invariant){
            throw new VDMException("Invariant is violated!");
        }

    }
```

A VDM Class is made that has the function invTest that tests the invariant condition and throws a VDMException if the precondition is violated.

Whenever the invariant has to be tested it will be called like this:

```
VDM.invTest();
```

Application of VDM Exception in Java:

Java Implementation

```
class VDMException extends RuntimeException {

    public VDMException(String message) {
        super(message);
    }
}
```

This will throw a custom VDMException whenever precondition or invariant is violated.

Translation Functions in Java:

functions
inRange(*val* : ℕ) *result* : B
pre TRUE
post *result* ⇔*MIN* ≤ *val* ≤ *MAX*

Java Implementation:

```
public boolean inRange (int value){
        return ((value>=MIN) && (value<=MAX));
    }
```

Translation reqElevator operation in Java:

operations

reqElevator(*floor* : ℕ)
ext wr *pickFloor* : ℕ
rd *userFloor* : ℕ
pre *inRange*(*floor*)
post *pickFloor* = *floor* ∧ *userFloor* = *floor* ∧ *start*()

Java Implementation:

```
public void reqElevator (int floor){

    System.out.println("\nElevator is Requested");

    VDM.preTest(inRange(floor)); //applying preconditon check using VDM
class that throws an exception

    System.out.println("\nYou are at Floor Number: " + floor);
    System.out.println("Elevator is at Floor Number: " + currentFloor);

    pickFloor = floor;
    userFloor = floor;
    closeDoor();
    start();

    VDM.invTest(this);

}
```

Translation reqFloor operation in Java:

reqFloor(*floor* : ℕ)
ext wr *destFloor* : ℕ
rd *userFloor* : ℕ

pre *inRange*(*floor*)
post *destFloor* = *floor* ∧ *userFloor* = *floor* ∧ *closeDoor*() ∧ *start*()

Java Implementation:

```
public void reqFloor (int floor){

    System.out.println("\nFloor " + floor +  " is Requested");

    VDM.preTest(inRange(floor)); //applying preconditon check using VDM
class that throws an exception

    destFloor = floor;
    userFloor = floor;
    closeDoor();
    start();

    VDM.invTest(this);

}
```

Translation start operation in Java:

start() *signalOut* : *ElevatorSignal*
ext rd *userFloor* : ℕ
rd *currentFloor* : ℕ
rd *direction* : *String*
rd *isStop* : *B*
rd *isDoorOpen* : *B*
pre isStop = TRUE ∧ *isDoorOpen* = FALSE
post (*userFloor* = *currentFloor* ∧ *openDoor*() ∧ *signalOut* = <DO_NOTHING>
∨ *userFloor* < *currentFloor* ∧ *direction* = *'down'* ∧ *drive*() ∧ *signalOut* = <DOWN>
∨ *userFloor* > *currentFloor* ∧ *direction* = *'up'* ∧ *drive*() ∧ *signalOut* = <UP>)

Java Implementation:

```
public ElevatorSignal start (){

    VDM.preTest(isDoorOpen == false && isStop == true);

    ElevatorSignal signalOut = ElevatorSignal.DO_NOTHING;

    if(userFloor == currentFloor){
        System.out.println("\nElevator is at the same floor");
```

```
            openDoor();
            signalOut = ElevatorSignal.DO_NOTHING;
            System.out.println(signalOut.toString());
        }

        if(userFloor < currentFloor){
            System.out.println("\nElevator will move downwards");
            direction = "down";
            drive();
            signalOut = ElevatorSignal.DOWN;
        }

        if(userFloor > currentFloor){
            System.out.println("\nElevator will move upwards");
            direction = "up";
            drive();
            signalOut = ElevatorSignal.UP;
        }

        //invariant check not to be done in operations where there are no
write operations.

        return signalOut;
    }
```

Translation drive operation in Java:

drive()
ext rd *currentFloor* : ℕ
    rd *userFloor* : ℕ
    rd *direction* : *String*
    rd *isStop* : $\mathbb{B}$
pre *userFloor* ≠ *currentFloor*
post *isStop* = FALSE ∧
    (*direction* = *'up'* ∧ *increment*()
      ∨ *direction* = *'down'* ∧ *decrement*()
      ∨ *userFloor* = *currentFloor* ∧ *stop*())

Java Implementation:

```
public void drive(){

    VDM.preTest(userFloor != currentFloor);

    System.out.println("\nElevator has started moving");

    isStop = false;

    while(userFloor != currentFloor){ //drive until they are not equal
        System.out.println("\nFloor - " + currentFloor );

        if(direction == "up"){
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            increment();
        }

        if(direction == "down"){
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            decrement();
        }
    }

    System.out.println("\nFloor - " + currentFloor );

    System.out.println("\nThe Elevator has arrived");

    stop();  //it is called when while loop is finished as
                //at that point userFloor == currentFloor so stop

}
```

Translation of stop operation in Java:

stop() *signalOut* : *ElevatorSignal*
ext rd *currentFloor* : ℕ
rd *userFloor* : ℕ
rd *isStop* : B
pre *isStop* = FALSE ∧ *userFloor* = *currentFloor*
post *isStop* = TRUE ∧ *openDoor*() ∧ *signalOut* = <STOP>

Java Implementation:

```
public ElevatorSignal stop (){

    System.out.println("\nElevator Stopped");

    VDM.preTest(isStop == false && userFloor == currentFloor);

    isStop = true;
    openDoor();

    ElevatorSignal signalOut = ElevatorSignal.STOP;
    System.out.println(signalOut.toString());

    return signalOut;
}
```

Translation of increment operation in Java:

increment() *signalOut* : *ElevatorSignal*
ext wr *currentFloor* : ℕ
rd *userFloor* : ℕ
pre *userFloor* > *currentFloor*
post *currentFloor* = $\overline{currentFloor}$ + 1 ∧ *signalOut* = <UP>

Java Implementation:

```
public ElevatorSignal increment (){

    VDM.preTest(userFloor > currentFloor);

    currentFloor = currentFloor + 1;
    ElevatorSignal signalOut = ElevatorSignal.UP;
    System.out.println(signalOut.toString());

    VDM.invTest(this); //invariant check before returning

    return signalOut;

}
```

Translation of decrement operation in Java:

decrement() *signalOut* : *ElevatorSignal*
ext wr *currentFloor* : ℕ
rd *userFloor* : ℕ
pre *userFloor* < *currentFloor*
post *currentFloor* = $\overline{currentFloor}$ - *1* ∧ *signalOut* = <DOWN>

Java Implementation:

```
public ElevatorSignal decrement (){

    VDM.preTest(userFloor < currentFloor);

    currentFloor = currentFloor - 1;
    ElevatorSignal signalOut = ElevatorSignal.DOWN;
    System.out.println(signalOut.toString());

    VDM.invTest(this);

    return signalOut;
}
```

Translation of openDoor operation in Java:

openDoor() *signalOut* : *DoorSignal*
ext wr *isDoorOpen* : B
rd *isStop* : B
pre *isDoorOpen* = FALSE ∧ *isStop* = TRUE
post *isDoorOpen* = TRUE ∧ *signalOut* = <OPEN>

Java Implementation:

```
public DoorSignal openDoor(){

    System.out.println("\nOpening Door..\n");
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    VDM.preTest(!isDoorOpen && isStop);

    isDoorOpen = true;
    DoorSignal signalOut = DoorSignal.OPEN;
    System.out.println(signalOut.toString());

    VDM.invTest(this);

    return signalOut;
}
```

Translation of closeDoor operation in Java:

closeDoor() *signalOut* : *DoorSignal*
ext wr *isDoorOpen* : B
pre TRUE
post *isDoorOpen* = FALSE ∧ *signalOut* = <CLOSE>

Java Implementation:

```
public DoorSignal closeDoor(){  //returns a DoorSignal object

    System.out.println("\nClosing Door..\n");
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    isDoorOpen = false;
    //declare and initialize a DoorSignal output variable.
    DoorSignal signalOut = DoorSignal.CLOSE;
    System.out.println(signalOut.toString());

    VDM.invTest(this);

    return signalOut;
}
```

Testing the Java Class:

```
//TESTER CLASS

import java.util.Scanner;


public class ElevatorTester {

    //Main Function
      public static void main(String[] args){

        char choice; //for storing user input of option choice
        Scanner scanner = new Scanner(System.in);

        //try-catch block
```

```
        try {

            ElevatorController elevator = new ElevatorController();
//creating obj of the Elevator class

            do{
                System.out.println("\n\n\n\t\tElevator Sytem\n");
                System.out.println("Floor Number Range from 0 - 15\n");
                System.out.println("1: Request Elevator");
                System.out.println("2: Request Floor");
                System.out.println("3: Quit");
                System.out.println("Enter Choice 1-3\n");

                choice = scanner.next().charAt(0); //reading user input

                System.out.println(); //blank line

                try {
                    switch (choice) {
                        case '1':
                            option1(elevator, scanner);
                            break;

                        case '2':
                            option2(elevator, scanner);
                            break;

                        default:
                            break;
                    }
                } catch (VDMException e) { // to catch invariant and
precondition violations in operations
                    e.printStackTrace();
                }

            }while(choice != '3');

        } catch (Exception e) { //to catch invariant violation in
initialization of Elevator object

            System.out.println("Initialization values violate
invariant!");
```

```
            System.out.println("Set Initialization values according to
invariant!");
            System.out.println("\nPress Enter to quit");
            scanner.nextLine();
        }

        scanner.close(); //to release resources

    };


    public static void option1(ElevatorController elevator, Scanner
scanner){

        System.out.println("\nEnter Current Floor Number: ");
        int userInput = scanner.nextInt();
        elevator.reqElevator(userInput);

    };

    public static void option2(ElevatorController elevator, Scanner
scanner){

        System.out.println("\nEnter Destination Floor Number: ");
        int userInput = scanner.nextInt();
        elevator.reqFloor(userInput);

    };


}
```

Complete Java Code of the Elevator System:

ElevatorController.java

```
//----x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-TYPES IN
VDM-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x--x-x-x-x-x--
//
```

```
//ElevatorSignal and DoorSignal that are used to signal the hardware
components

class ElevatorSignal {

    //one variable that will be used
    private int value;


    //initializing objects inside the class with variables names that
correspond to VDM-SL
    public static final ElevatorSignal UP = new ElevatorSignal(0);
    public static final ElevatorSignal DOWN = new ElevatorSignal(1);
    public static final ElevatorSignal STOP = new ElevatorSignal(2);
    public static final ElevatorSignal DO_NOTHING = new
ElevatorSignal(3);

    //constructor to set the value of the variable value
    private ElevatorSignal(int v){
        value = v;
    }

    public String toString(){

        switch (value) {
            case 0:
                return "SIGNAL: UP";

            case 1:
                return "SIGNAL: DOWN";

            case 2:
                return "SIGNAL: STOP";

            default:
                return "SIGNAL: DO_NOTHING";
        }
    }

};


class DoorSignal {
```

```
    private int value;

    public static final DoorSignal OPEN = new DoorSignal(0);
    public static final DoorSignal CLOSE = new DoorSignal(1);

    private DoorSignal(int x){
        value = x;
    }

    public String toString(){

        switch (value) {
            case 0:
                return "SIGNAL: OPEN";

            default:
                return "SIGNAL: CLOSE";
        }
    }

};

interface InvariantCheck{
    public boolean inv();
}

/**
 * Elevator
 */

public class ElevatorController implements InvariantCheck {

//----x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-DECLARING
VALUES OF VDM-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x--x-
x-x-x-x--//

    public static final int MAX = 15; //represents highest floor
    public static final int MIN = 0; //represents lowest floor

//----x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x- STATE IN
VDM-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x--x-x-x-x-x--
//
```

```
//all state variables are private variables which can be accessed only
within the class.

    //Natural number in VDM is converted to int in JAVA
    private int pickFloor;
    private int destFloor;
    private int currentFloor;
    private String direction;
    private int userFloor;
    private boolean isStop;
    private boolean isDoorOpen;


    //INVARIANT FUNCTION of VDM

    //invariant is supposed to be a public function that returns a
boolean
    public boolean inv (){
        return (inRange(pickFloor) && inRange(destFloor)
        && inRange(userFloor) && inRange(currentFloor));
    }

//----x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-FUNCTIONS
IN VDM-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x--x-x-x-x-
x--//

    public boolean inRange (int value){
        return ((value>=MIN) && (value<=MAX));
    }


    //----x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-VDM
INITIALIZATION CLAUSE-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-
x-x-x--x-x-x-x-x--//

    public ElevatorController() {
        this.pickFloor = 0;
        this.destFloor = 0;
        this.currentFloor = 0;
        this.direction = ""; // Add a default value for String
        this.userFloor = 0;
        this.isStop = true;
```

```
        this.isDoorOpen = false;

        VDM.invTest(this);
    }

    //----x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-VDM
OPERATIONS-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x-x--x-x-
x-x-x--//

    public void reqElevator (int floor){

        System.out.println("\nElevator is Requested");

        VDM.preTest(inRange(floor)); //applying preconditon check using
VDM class that throws an exception

        System.out.println("\nYou are at Floor Number: " + floor);
        System.out.println("Elevator is at Floor Number: " +
currentFloor);

        pickFloor = floor;
        userFloor = floor;
        closeDoor();
        start();

        VDM.invTest(this);

    }

    public void reqFloor (int floor){

        System.out.println("\nFloor " + floor +  " is Requested");

        VDM.preTest(inRange(floor)); //applying preconditon check using
VDM class that throws an exception

        destFloor = floor;
        userFloor = floor;
        closeDoor();
        start();

        VDM.invTest(this);
```

```
    }

    public ElevatorSignal start (){

        VDM.preTest(isDoorOpen == false && isStop == true);
        System.out.println("\nElevator Started");

        ElevatorSignal signalOut = ElevatorSignal.DO_NOTHING;

        if(userFloor == currentFloor){
            System.out.println("\nElevator is at the same floor");
            openDoor();
            signalOut = ElevatorSignal.DO_NOTHING;
            System.out.println(signalOut.toString());
        }

        if(userFloor < currentFloor){
            System.out.println("\nElevator will move downwards");
            direction = "down";
            drive();
            signalOut = ElevatorSignal.DOWN;
        }

        if(userFloor > currentFloor){
            System.out.println("\nElevator will move upwards");
            direction = "up";
            drive();
            signalOut = ElevatorSignal.UP;
        }

        //invariant check not to be done in operations where there are
no write operations.

        return signalOut;
    }

    public void drive(){

        VDM.preTest(userFloor != currentFloor);

        System.out.println("\nElevator has started moving");

        isStop = false;
```

```
        while(userFloor != currentFloor){ //drive until they are not
equal
            System.out.println("\nFloor - " + currentFloor );

            if(direction == "up"){
                try {
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                increment();
            }

            if(direction == "down"){
                try {
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                decrement();
            }
        }

        System.out.println("\nFloor - " + currentFloor );

        System.out.println("\nThe Elevator has arrived");

        stop();  //it is called when while loop is finished as
                 //at that point userFloor == currentFloor so stop

    }


    public ElevatorSignal stop (){

        System.out.println("\nElevator Stopped");

        VDM.preTest(isStop == false && userFloor == currentFloor);

        isStop = true;
        openDoor();
```

```
        ElevatorSignal signalOut = ElevatorSignal.STOP;
        System.out.println(signalOut.toString());

        return signalOut;
    }


    public ElevatorSignal increment (){

        VDM.preTest(userFloor > currentFloor);

        currentFloor = currentFloor + 1;
        ElevatorSignal signalOut = ElevatorSignal.UP;
        System.out.println(signalOut.toString());

        VDM.invTest(this); //invariant check before returning

        return signalOut;

    }

    public ElevatorSignal decrement (){

        VDM.preTest(userFloor < currentFloor);

        currentFloor = currentFloor - 1;
        ElevatorSignal signalOut = ElevatorSignal.DOWN;
        System.out.println(signalOut.toString());

        VDM.invTest(this);

        return signalOut;
    }


    public DoorSignal openDoor(){

        System.out.println("\nOpening Door..\n");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }

        VDM.preTest(!isDoorOpen && isStop);

        isDoorOpen = true;
        DoorSignal signalOut = DoorSignal.OPEN;
        System.out.println(signalOut.toString());

        VDM.invTest(this);

        return signalOut;
    }

    public DoorSignal closeDoor(){  //returns a DoorSignal object

        System.out.println("\nClosing Door..\n");
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        isDoorOpen = false;
        //declare and initialize a DoorSignal output variable.
        DoorSignal signalOut = DoorSignal.CLOSE;
        System.out.println(signalOut.toString());

        VDM.invTest(this);

        return signalOut;
    }

};
```

VDM.java

```
class VDMException extends RuntimeException {

    public VDMException(String message) {
        super(message);
    }
}
```

```
public class VDM {

    public static void preTest (boolean exp){
        if(!exp){
            throw new VDMException("Precondition not Satisfied.");
        }
    }

    public static void invTest (ElevatorController elevator) {

        boolean invariant = elevator.inv(); //this function checks
whether the invariant is satisfied and then returns a boolean.
        if(!invariant){
            throw new VDMException("Invariant is violated!");
        }

    }

}
```

ElevatorTester.java

```
//TESTER CLASS

import java.util.Scanner;


public class ElevatorTester {

    //Main Function
      public static void main(String[] args){

        char choice; //for storing user input of option choice
        Scanner scanner = new Scanner(System.in);

        //try-catch block
        try {

            ElevatorController elevator = new ElevatorController();
//creating obj of the Elevator class

            do{
```

```
                System.out.println("\n\n\n\t\tElevator Sytem\n");
                System.out.println("Floor Number Range from 0 - 15\n");
                System.out.println("1: Request Elevator");
                System.out.println("2: Request Floor");
                System.out.println("3: Quit");
                System.out.println("Enter Choice 1-3\n");

                choice = scanner.next().charAt(0); //reading user input

                System.out.println(); //blank line

                try {
                    switch (choice) {
                        case '1':
                            option1(elevator, scanner);
                            break;

                        case '2':
                            option2(elevator, scanner);
                            break;

                        default:
                            break;
                    }
                } catch (VDMException e) { // to catch invariant and
precondition violations in operations
                    e.printStackTrace();
                }

            }while(choice != '3');

        } catch (Exception e) { //to catch invariant violation in
initialization of Elevator object

            System.out.println("Initialization values violate
invariant!");
            System.out.println("Set Initialization values according to
invariant!");
            System.out.println("\nPress Enter to quit");
            scanner.nextLine();
        }

        scanner.close(); //to release resources
```

```
    };


    public static void option1(ElevatorController elevator, Scanner
scanner){

        System.out.println("\nEnter Current Floor Number: ");
        int userInput = scanner.nextInt();
        elevator.reqElevator(userInput);

    };

    public static void option2(ElevatorController elevator, Scanner
scanner){

        System.out.println("\nEnter Destination Floor Number: ");
        int userInput = scanner.nextInt();
        elevator.reqFloor(userInput);

    };


}
```

Boundary Value Analysis:

Elevator System has a constraint that the floor values shall be between 0 and 15. Therefore through the boundary value analysis, the elevator system can be tested for the following test cases:

Test Case (floor number)	Expected Output	Actual Output	Status
-1	Precondition Failed	VDMException: Precondition not Satisfied. at VDM.preTest(VDM.java:12) at ElevatorController.reqElevator(ElevatorController.java:130) at ElevatorTester.option1(ElevatorTester.java:67) at ElevatorTester.main(ElevatorTester.java:34)	Passed
0	Elevator Started	Elevator Started	Passed
1	Elevator Started	Elevator Started	Passed
14	Elevator Started	Elevator Started	Passed
15	Elevator Started	Elevator Started	Passed
16	Precondition Failed	VDMException: Precondition not Satisfied. at VDM.preTest(VDM.java:12) at ElevatorController.reqElevator(ElevatorController.java:130) at ElevatorTester.option1(ElevatorTester.java:67)	Passed

		at ElevatorTester.main(ElevatorTester.java:34)	