THE ELEVATOR PROJECT

Tom Elmaleh (24454)

Dorset College
Data Structures & Algorithms

LECTURER NAME: William Hanks	
STUDENT NAME: Tom Elmaleh	**STUDENT ID: 24454**
PROGRAMME: Semester Abroad	**STAGE/YEAR: 3**
MODULE NAME: Data Structures & Algorithms	
ASSIGNMENT NO. & TITLE: CA Project	
GROUP (Names & Student IDs): Individual Project	
• All assignments must be submitted through Moodle by the agreed submission date. • Only submit a signed hardcopy to your lecturer if requested and you must arrange to hand it directly to the Lecturer and must include this cover sheet. • No submissions will be accepted at Reception. • Late assignments will incur a penalty unless a learner has documented personal mitigating circumstances or if they have been granted an extension. All such documentation must be submitted at least 7 days prior to the submission date. For late submission the assessment grade will be reduced by 10% for each day that the assessment is late. After day five, the assignment will not be accepted.	Date Received: (Lecturer use only)
MODE OF SUBMISSION: SOFTCOPY ☒ HARDCOPY ☐	
COMPONENTS OF SUBMISSION: **(e.g. no and type of pieces submitted, no of pages in a report, disk included?)**	

DECLARATION: I declare that:

- *By uploading this work to Moodle I automatically declare that this work is entirely my own and that I have acknowledged all materials and sources used in its preparation;*
- *I have not copied in part or whole or otherwise plagiarised the work of anyone else and have not knowingly allowed others to plagiarise my work in this way;*
- *I understand that plagiarism is a serious offence and that I am bound by Dorset College policy on Academic Integrity. I understand that I may be penalised if I have violated the policy in any way;*
- *This assignment has not been submitted for any other course or module at Dorset College or any other institution, without authorisation by the relevant lecturer(s);*
- *I have read and abided by all of the requirements set down for this assignment.*

SIGNATURE* Tom Elmaleh DATE 09/12/2021

(* if this is a group assignment, each member of the group must sign).

ASSIGNEMENT SOLUTION

Table des matières

Introduction

My topic was to create an elevator through object-oriented programming by using different data structures. This elevator is supposed to consider the number of passengers inside. For example, it won't retrieve a passenger if it has reached its full capacity (6 passengers). The other condition I imposed was that I would try to optimise its movement as much as I can. Every algorithm for an elevator has at least one flaw, I used two of them. I added to both algorithms some code to make it more optimised (capacity, less motion). At first, I started with a First Come First Serve algorithm (explained here [1]) combined with a process to serve multiple requests in parallel. This algorithm optimises the elevator motions the least, especially for a sequence of up and down (main flaw). Therefore, I decided to use a second algorithm called the LOOK algorithm ([2]). It's the most optimised algorithm and it's used for real elevator system. In both processes, I have also added an innovation. My elevator remembers the request of a passenger if he can't get inside because of a full capacity (in terms of passenger). The elevator will take care of his request once possible given the capacity and the passenger concerned won't have to call the elevator again in the meantime.

Explanation of the algorithm

1. Classes

- Passenger

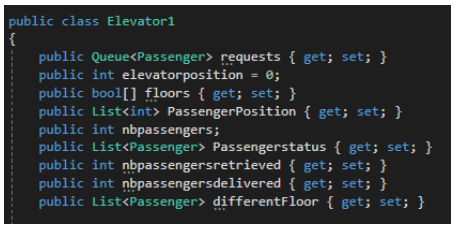
This class permits to store the important information related to each passenger of the elevator whether he's inside it or not.
A passenger has 5 different attributes:

1. The initial position (int) of the passenger before he enters inside the elevator.
2. The floor to which the passenger wants to go (requestedfloor).
3. The direction indicating if the passenger wants to go up or down.
4. Insideelevator which indicates if the passenger is inside the elevator.
5. Requestadded indicates if the passenger's request was treated. It is used for a specific case related to the capacity and the elevator position: (elevatorpos =initialpos and nbpassengers > capacity).

A passenger is created with initialpos, requestedfloor and direction.

- Elevator1

For this class, the elevator's movement is based on the First Come First Serve algorithm



The screenshot depicts all the attributes of this class. Many of them are used in Elevator2 (in blue)

I will now describe each of them :

1. Queue<Passenger> requests contains the requests of each passenger. It's used to process them.
2. List<int> PassengerPosition contains the initial position of the passengers who are not at the same position as the elevator. It permits to get the number of passengers retrieved and not retrieved at a specific floor. As it's a list of int, I can use Contains(int) to compare with elevatorposition (in Up()).
3. List<Passenger> Passengerstatus stores each passenger of the elevator. It's used to determine how many passengers were delivered at a specific floor and to add a passenger's request to requests(in case requestadded=false). The list is updated given how many passengers were delivered, retrieved, and not retrieved by using the methods UpdateNbPassengers() and Search().
4. List<Passenger> differentFloor : Stores the passengers who are not at the same position as the elevator. It's used to find out which passengers were really retrieved and not retrieved at a specific floor by the elevator in the method Search().
5. elevatorposition : The actual position of the elevator. It is set by default to the floor 0.
6. Nbpassengers : Number of passengers inside the elevator
7. nbpassengersretrieved : Number of passengers retrieved by the lift at a specific floor
8. nbpassengersdelivered : Number of passengers delivered at a specific floor by the lift.
9. Bool [] floors indicates for each floor whether the elevator should stop (true) or not (false).

Once the elevator is created ([3]) in Program.cs ([4]). We call the method Request(Passenger) ([5]) to add every request to the queue requests. Then as shown in Program.cs, we call the method Move() which ables the elevator to move based on the direction.

```
public void Move()
{
    string direction = "";
    Passenger passenger;
    while (requests.Count != 0)    // if they are no more requests the elevat
    {
        direction = requests.Peek().direction;
        while (requests.Count != 0 && direction == requests.Peek().direction)
        {
            passenger = requests.Dequeue();
            Process(passenger);
        }

        if (direction == "up")
        {
            Up();
        }

        if (direction == "down")
        {
            Down();
        }
    }
}
```

The 1st while loop implies that the elevator will stop moving once, they are no more requests. Then we define the direction of the elevator by the most recent request. The 2nd loop implies that while the direction of each request is the same, we take them off and we Process them ([6]). Given the direction the elevator goes ever up or down (permits to serve multiple requests in parallel in most cases). The methods Up() and Down() work in the same way. The only difference is that floors is read in the order for Up() and in reverse for Down(). In the appendix there is an explanation of Up [(7)] and a screenshot of Down([8]). Up() and Down() call the other methods (all of them are explained in the appendix).

- Elevator 2

For this class, the elevator's movement is based on the LOOK algorithm which is more optimised than the first one. The attributes of Elevator2 are almost the same as those of the Elevator1 ([11]).The constructor is the same as the one from Elevator1 except we initialize the queue goingdown and the boolean arrays up and down. The method with the most changes is Move ([12]). The methods

Process and Search are slightly changed. There is one more parameter : bool [ ] direction (can be either up or down when these methods are called), in the code direction replaces floors. Another small change is made in methods Up() and Down() : floors is respectively replaced by [] up and [] down. No changes are made in the methods UpdateNbPassengers(pos) and Request(passenger).

2. Data structures :

I used a boolean array as the number of floors for an elevator doesn't change and it's the easiest way to know if the elevator needs to stop there or not (true). I tried at first to use a sorted list of passengers but it was too difficult to process the requests.

I decided to use a queue to treat the requests because I want to treat the most recent request for each passenger which corresponds to the First-In First-Out principle related to a queue. As the queue considers the most recent request, when I copy some information about the passengers on lists it is already sorted.

I used 3 different lists to copy information about passengers as it's easy to remove the information or to add the information when something is done on the request of a passenger.

3. Showcase :

Full capacity is considered and my innovation works. It is shown here ([13]). The class Elevator2 has a better result than the one from Elevator1 in terms of optimisation of motions. To prove this, I used several requests (10) ordered as a sequence of up and down motions. We can see that they are much more moves for Elevator1 than for Elevator2 because of the LOOK algorithm. The requests are shown in the screenshot below :

```
Elevator2 elevator = new Elevator2();
elevator.Request(new Passenger(0,4,"up"));
elevator.Request(new Passenger(5,3,"down"));
elevator.Request(new Passenger(0,3,"up"));
elevator.Request(new Passenger(6,2,"down"));
elevator.Request(new Passenger(3,6,"up"));
elevator.Request(new Passenger(6,0,"down"));
elevator.Request(new Passenger(2,6,"up"));
elevator.Request(new Passenger(6,2,"down"));
elevator.Request(new Passenger(2,6,"up"));
elevator.Request(new Passenger(6,0,"down"));
elevator.Move();
```

Elevator 1 : 15 moves Screenshot here ([14])

Elevator 2 : 8 moves

```
Elevator went from floor 0 to floor 2 to get 2 passenger(s) with 2 passengers inside

Elevator went from floor 2 to floor 3 to get 1 passenger(s) with 4 passengers inside
It also delivered 1 passenger(s). It now has 4 passenger(s)

Elevator went up from floor 3 to floor number 4 and delivered 1 passenger(s) with 4 passenger(s) inside.
It now has 3 passenger(s) inside

Elevator went up from floor 4 to floor number 6 and delivered 3 passenger(s) with 3 passenger(s) inside.
It now has 0 passenger(s) inside

Elevator went from floor 6 to floor 5 to get 1 passenger(s) with 4 passengers inside

Elevator went down from floor 5 to floor number 3 and delivered 1 passenger(s) with 5 passenger(s) inside
It now has 4 passenger(s) inside

Elevator went down from floor 3 to floor number 2 and delivered 2 passenger(s) with 4 passenger(s) inside
It now has 2 passenger(s) inside

Elevator went down from floor 2 to floor number 0 and delivered 2 passenger(s) with 2 passenger(s) inside
It now has 0 passenger(s) inside
```

References

I didn't use/copy any code from the internet. I coded all the methods by myself.

The only reference I have is a youtube video which described different kinds of algorithm made to move an elevator.

Link to the video : https://www.youtube.com/watch?v=siqiJAJWUVg&list=WL&index=19

This video illustrates many points related to the elevator :

- the requirements of an Elevator System
- the general objects (not in terms of code) necessary to design an elevator (I only used 3 of them : passenger, elevator, and floors). The video says that its unnecessary to use a Passenger class but I used one.
- 4 different algorithms that we can use for an elevator and the flaws related to them. I used two of them : First Come First Serve and LOOK

There wasn't any code in the video. The only things I learned in terms of code was to use boolean arrays (up,down,floors) to check if the elevator needs to stop and to use a priority queue (with parameter floorrequested and direction) to process the request and consider the priority. I couldn't implement the priority queue so I used a queue.

I will describe some differences and common points with 2 codes :

Link to 1st code : https://gist.github.com/dmjio/3430920

There are two classes : Elevator and ElevatorStatus. There isn't a class for the passenger in this code.

This code uses also a boolean array to check if the elevator needs to stop. This code doesn't use a queue to process the requests. It also doesn't take account of the capacity of the elevator.

There is also a method dedicated to the request of the passenger called FloorPress but the method is different from my code as I didn't consider the status of the elevator. I used its position.

The methods Ascend and Descend look like Up and Down() methods from my code. However, they don't take account of passengers calling the elevator who are at the same floor as the elevator.

This code doesn't take account of the case mentioned above while mine does. The elevator only treats requests one by one if the passenger is at the same floor as him.

Link to 2nd code : https://dotnetfiddle.net/UwHreh

They are 3 classes (Rider, Elevator, Control System) and one interface (`ElevatorControlSystem`) in this code.

The class Rider has some common points with Passenger in terms of attributes (Originating and Destination floor).

The class Direction permits to deduce the direction based on the originating and destination floor. I didn't include anything to determine the direction. I supposed the passenger had to indicate when calling the elevator (in the class Passenger the attribute direction).

The method Step permits to move the elevator and it doesn't have anything in common with my method Move().

The objects used in this code are more complex and harder to understand for me. I cannot say whether this elevator will be optimised in terms of movement, priority of requests and capacity.

It seems more optimised than the first code. The user can create the number of requests, floors and elevators.

I used this link to explain the LOOK algorithm in the appendix :
https://en.wikipedia.org/wiki/LOOK_algorithm

Questions

1.a) The most sublime algorithm of my code is the method Up(). This algorithm is iterative because the programs starts with a loop for which is an iterative structure. There is also a while loop for one specific case. All the methods called in Up() also use a for loop. I chose to go with this approach as I had to check every single floor of the elevator and these floors were represented by an array. Using a recursive approach would have been difficult to read the array.

b) At first, there is a for loop of complexity O(n)(with n, the size of floors) in the method. Then they are 3 possible cases for each I in the loop :

- The 1st if case is validated : The global complexity is O(m) (complexity of the method UpdateNbPassengers(i), m corresponds to the size of the list PassengerStatus. The size of PassengerStatus decreases when this method is used in some cases).
- The 2nd if case is validated : The global complexity is O(l) * O(m)*O(k) = O(jmk).
  O(j) is the complexity of the while loop : j is the size of the list PassengerPosition (it also decreases in the while loop if PassengersPosition.Contains(i)).
  O(k) is the complexity of the method Search : k is the size of the list differentFloor (it also decreases if either retrieved>0 or not retreived>0).
  O(m) is the complexity of the method UpdateNbPassengers(i)
- The third case : No if cases are validated so the complexity is O(1).

I also want to specify that you can't go into 1st , 2nd and 3rd cases at the same time for each index.

The complexity of the method Up() is the same for both classes Elevator1 and Elevator2.

So, the worst-case (all 3 cases are validated at least for one index in the for loop) complexity of this method is :

O(n) * O(jmk) * O(m) * O(1) = O(n*m^2*l*k)

The average (1st and 3rd case are validated) complexity is:

O(n) * O(m) *O(1) = O(nm)

The best-case (only 3rd case is validated) complexity is :

O(n) * O(1) = O(n)

2. a) The most sublime algorithm in my code is the method Up(). If I was forbidden to use the array floors, I would have used a list as it can work like an array. This wouldn't have any effect on my code.

Appendix

1. First Come First Serve algorithm : Requests are served in the order they arrive. The main flaw is the extra up and down motions of the elevator to serve the requests. This algorithm alone can't serve multiple requests in parallel. It is the considered as one of the less optimised algorithm for an elevator in terms of motion.

2. LOOK Algorithm :
   The elevator will look at all the floors to check if it needs to stop at a specific floor. It will start by checking if there are requests to go up. If it's the case, the elevator will go up and once they are no more requests implying to go up. It will stop and check if they are requests for the down direction. If that's the case the elevator will go down and once all requests implying to go down are fulfilled, it will stop. In my code the algorithm is repeated until they are no more requests. This algorithm alone can serve multiple requests in parallel. It has a flaw is linked to the priority of the requests. Example : if the first and the third request imply to go up while the second request implies to go down, the first and the third request will be fulfilled before the second request. For 3 requests, it's a minor flaw but for a bigger number of requests, it's a more important one.

3. Constructor for Elevator1

```
public Elevator1()
{
    Passengerstatus = new List<Passenger>();
    floors = new bool[7];
    PassengerPosition = new List<int>();
    requests = new Queue<Passenger>();
    nbpassengers = 0;
    differentFloor = new List<Passenger>();
}
```

4. Program.cs

```
static void Main(string[] args)
{
    Elevator2 elevator = new Elevator2();
    elevator.Request(new Passenger(0, 4, "up"));
    elevator.Request(new Passenger(0, 5, "up"));
    elevator.Request(new Passenger(0, 3, "up"));
    elevator.Request(new Passenger(0, 6, "up"));
    elevator.Request(new Passenger(0, 1, "up"));
    elevator.Request(new Passenger(0, 2, "up"));
    elevator.Request(new Passenger(4, 6, "up"));
    elevator.Request(new Passenger(4, 0, "down"));
    elevator.Request(new Passenger(4, 2, "down"));
    elevator.Request(new Passenger(6, 0, "down"));
    elevator.Move();
}
```

5. Requests

```
public void Request(Passenger passenger)
{
    requests.Enqueue(passenger);
}
```

6. Process : permits to deal with the passenger's request

```
public void Process(Passenger passenger)
{
    if (passenger.requestedfloor != passenger.initialposition)
    {
        if (elevatorposition == passenger.initialposition)
        {

            if (nbpassengers <= 5)
            {
                nbpassengers++;
                floors[passenger.requestedfloor] = true;
                passenger.insideelevator = true;
            }
            else
            {
                passenger.insideelevator = false;
                passenger.requestadded = false;
            }
        }
        else
        {
            passenger.insideelevator = false;
            PassengerPosition.Add(passenger.initialposition);
            differentFloor.Add(passenger);
        }
        Passengerstatus.Add(passenger);
    }                                                    // we add the
}
```

There are 2 main cases in Process :

i. The elevator is at the same position as the passenger and :
   - Full capacity is not reached : The requested floor is set to true in floors so that the elevator will stop there. We also set insideelevator to true as the passenger is inside the elevator. Nbpassengers is increased.
   - Full capacity is reached : The passenger will not be inside the elevator, so his request can't be fulfilled (requestadded=false)

ii. The elevator is not at the same position as the passenger :
He is not inside the elevator. We add his initial position to the list <int>PassengerPosition and the passenger to the list <Passenger> differentFloor.
Once these cases are dealt , we add the passenger to the list <Passenger> PassengerStatus.

7. Up()

When reading the values inside floors (in order) there are two possible cases (each screenshot describes one of them) :

```
public void Up()
{
    int NotRetreived = 0;
    bool display = true;
    for (int i = 0; i < floors.Length; i++)                          // we read from 0 to 6th floor
    {
        nbpassengersretrieved = 0;
        if (floors[i] == true && PassengerPosition.Contains(i) == false)       // Case in which the elevator has to stop at the floor i and no passengers need to get inside the elevator
        {
            UpdateNbPassengers(i);
            Console.WriteLine($"\nElevator went up from floor {elevatorposition} to floor number {i} and delivered {nbpassengersdelivered} passenger(s) with {nbpassengers} passengers inside ");
            nbpassengers -= nbpassengersdelivered;
            Console.WriteLine($"It now has {nbpassengers} passenger(s) inside ");
            floors[i] = false;                                                          // The elevator doesn't need to stop there again
            elevatorposition = i;                                                       // The elevator position is now i
        }
```

i. Screenshot above : The elevator stops at floor i and no passengers needs to get inside the lift at this floor. Then we use UpdateNbPassengers (i) to update nbpassengers delivered (explained here [9]). Finally, we change the elevator position to i and floors[i] is set to false.

```
if (PassengerPosition.Contains(i) == true)                                          // Case in which the elevator reached the initial position of a passenger (we retrieve it by taking account of the capacity), passengers
{
    display = true;
    nbpassengersretrieved = 0;
    while (PassengerPosition.Count != 0 && PassengerPosition.Contains(i) == true)                    // while they are still people at this floor orr PassengerPosition is not empty we count the number of passengers retreived ar
    {
        if (nbpassengers <= 5)
        {
            PassengerPosition.Remove(i);                        // We have to remove it from PassengerPosition so that it won't get taken into account again since the passenger has been retreived
            nbpassengersretrieved++;
            nbpassengers++;
        }
        else
        {
            PassengerPosition.Remove(i);                    // We have to remove it from PassengerPosition because any passenger related to this position will be removed as shown in the method Search
            NotRetreived++;
        }
    }
    Search(i, NotRetreived);                            // We use Search to deal with passengers retreived and not retreived
    if (nbpassengersretrieved > 0)                                                                                  // If some passengers were retrieved the elevator will move
    {
        Console.WriteLine($"\nElevator went from floor {elevatorposition} to floor {i} to get {nbpassengersretrieved} passenger(s) with {nbpassengers - nbpassengersretrieved} passengers inside ");
        elevatorposition = i;
        display = false;
    }
    UpdateNbPassengers(i);                                      // We update nbpassengersdelivered
    if (nbpassengersdelivered > 0 && floors[i] == true)                                                                                 // If some passengers have to be delivered at this floor
    {
        nbpassengers -= nbpassengersdelivered;
        if (display == false) { Console.WriteLine($"It also delivered {nbpassengersdelivered} passenger(s). It now has {nbpassengers} passenger(s)"); }
        if (display == true)
        {
            Console.WriteLine($"\nElevator went from floor {elevatorposition} to floor {i} and delivered {nbpassengersdelivered} passenger(s) with {nbpassengers + nbpassengersdelivered} passengers inside. It now has {nbpassengers}");
            elevatorposition = i;
        }
        floors[i] = false;
    }
```

ii. Screenshot above : The elevator has reached the initial position of some passenger(s). The while loop implies that we count the number of passengers retrieved and not retrieved given the capacity. We also remove the passengers retrieved (the request is fulfilled) and not retrieved (linked to the method Search) from PassengerPosition. The loop keeps going if they are still people at this floor and PassengerPosition is not empty.

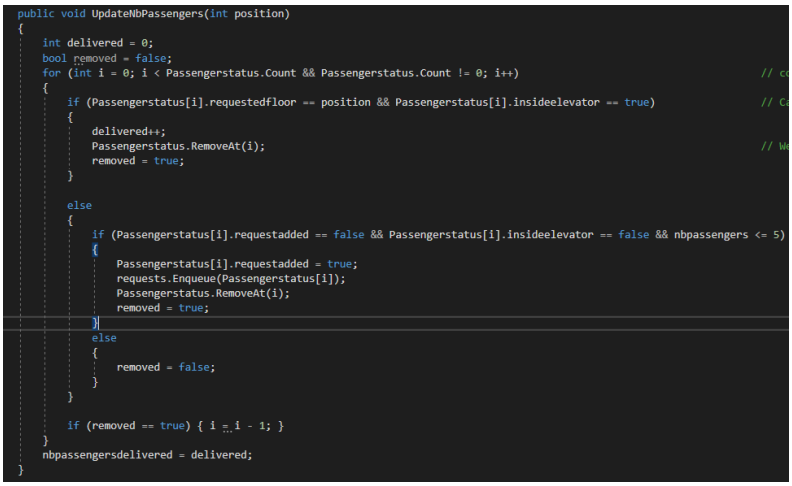
Then we use the method Search to deal with the passengers retrieved and not retrieved (explained here [10]). If (nbpassengersretrieved>0), we move the elevator, retrieve the passenger(s) and change its position. We update nbpassengersdelivered through the method UpdateNbPassengers(i). If nbpassengersdelivered>0 and the elevator must stop there to deliver the passengers. We can also deliver the passengers and move the elevator if it hasn't moved before (display=true).

8. Down()

```
public void Down()
{
    int NotRetreived = 0;
    bool display = true;
    for (int i = floors.Length - 1; i >= 0; i--)                        // we read from the 6th floor to 0
    {
        nbpassengersretrieved = 0;
        if (floors[i] == true && PassengerPosition.Contains(i) == false)            // Case in which the elevator has to stop at the floor i and no passengers get inside the elevator
        {
            UpdateNbPassengers(i);
            Console.WriteLine($"\nElevator went down from floor {elevatorposition} to floor number {i} and delivered {nbpassengersdelivered} passenger(s) with {nbpassengers} passengers inside");
            nbpassengers -= nbpassengersdelivered;
            Console.WriteLine($"It now has {nbpassengers} passenger(s) inside ");
            elevatorposition = i;
            floors[i] = false;
        }
```

```
if (PassengerPosition.Contains(i) == true)                  // Case in which the elevator reached the initial position of a passenger (we retrieve it by taking account of the capacity), passengers can also be delivered
{
    display = true;
    while (PassengerPosition.Count != 0 && PassengerPosition.Contains(i) == true)                    // while they are still people at this floor or PassengerPosition is not empty we count the number of passengers retreived and
    {
        if (nbpassengers <= 5)                                                                  // We check if full capacity is reached
        {
            PassengerPosition.Remove(i);                                                        // We have to remove it from PassengerPosition so that it won't get taken into account again since the passenger has been retreived
            nbpassengersretrieved++;
            nbpassengers++;
        }
        else
        {
            PassengerPosition.Remove(i);                                                        // We have to remove it from PassengerPosition because any passenger related to this position will be removed as shown in the method S
            NotRetreived++;
        }
    }
    Search(i, NotRetreived);                                                    // We use Search to deal with passengers retreived and not retreived
    if (nbpassengersretrieved > 0)
    {
        Console.WriteLine($"\nElevator went from floor {elevatorposition} to floor {i} to get {nbpassengersretrieved} passenger(s) with {nbpassengers - nbpassengersretrieved} passengers inside ");
        elevatorposition = i;
        display = false;
    }
    UpdateNbPassengers(i);                                      // We update nbpassengers delivered
    if (nbpassengersdelivered > 0 && floors[i] == true)                                                                                 // If some passengers have to be delivered at this floor
    {
        nbpassengers -= nbpassengersdelivered;
        if (display == false) { Console.WriteLine($"It also delivered {nbpassengersdelivered} passenger(s). It now has {nbpassengers} passenger(s)"); }
        if (display == true)
        {
            Console.WriteLine($"\nElevator went from floor {elevatorposition} to floor {i} and delivered {nbpassengersdelivered} passenger(s) with {nbpassengers + nbpassengersdelivered} passengers inside. It now has {nbpassengers}");
            elevatorposition = i;
        }
        floors[i] = false;
    }
```

9. UpdateNbPassengers(i) : Updates the number of passengers delivered at a specific floor(position). It's also used to add a passenger who wasn't in requests because the elevator was at full capacity (requestadded=false)



```
public void UpdateNbPassengers(int position)
{
    int delivered = 0;
    bool removed = false;
    for (int i = 0; i < Passengerstatus.Count && Passengerstatus.Count != 0; i++)
    {
        if (Passengerstatus[i].requestedfloor == position && Passengerstatus[i].insideelevator == true)
        {
            delivered++;
            Passengerstatus.RemoveAt(i);
            removed = true;
        }

        else
        {
            if (Passengerstatus[i].requestadded == false && Passengerstatus[i].insideelevator == false && nbpassengers <= 5)
            {
                Passengerstatus[i].requestadded = true;
                requests.Enqueue(Passengerstatus[i]);
                Passengerstatus.RemoveAt(i);
                removed = true;
            }
            else
            {
                removed = false;
            }
        }

        if (removed == true) { i = i - 1; }
    }
    nbpassengersdelivered = delivered;
}
```

When reading PassengerStatus there are 2 main cases (2 if) :

I. The passenger is inside the elevator and the elevator has reached his requested floor. We don't need to check requestadded as it's used only to add a passenger to requests. Delivered is incremented and the passenger is removed from Passengerstatus as his request is fulfilled.

II. The elevator is not at full capacity, the passenger is not inside the elevator and his request wasn't considered because of full capacity. We set requestadded to true as it's the default value for the passenger and this value can be changed in Process(passenger). We add the passenger to the queue requests and we remove him from Passengerstatus as he will be added again.

The boolean removed permits to avoid an ArgumentOutOfRangeException for the index i. The modifications of i are made in case we remove a passenger from the list.

10. Search : Method to process what should be done with the passengers retrieved and not retrieved. We start with the passengers retrieved as differentFloor is sorted from the most recent request to the least recent with (pos!=elevatorposition). This ensures that the elevator won't fulfil the wrong request.

```
public void Search(int initialpos, int notretreived)
{
    int count = 0;
    int index = 0;
    if (nbpassengersretrieved > 0)
    {
        while (differentFloor.Count != 0 && count != nbpassengersretrieved)
        {
            if (differentFloor[index].initialposition == initialpos && count != nbpassengersretrieved)
            {
                Passenger pass = differentFloor[index];
                differentFloor.Remove(pass);
                Passengerstatus.Remove(pass);
                pass.insideelevator = true;
                Passengerstatus.Add(pass);
                floors[pass.requestedfloor] = true;
                count++;
                if (index > 0) { index--; }
            }
            else { index++; }
        }
    }
```

a) Retrieved > 0 (Screenshot above) : In the 1st while loop, the condition implies that if there is a passenger in differentFloor who's initial position is equal to initialpos and count!=nbpassengersretrieved, this passenger has been retrieved by the elevator. He is removed from the lists differentFloor (no longer necessary as he is in the elevator) and PassengerStatus. Insideelevator is set to true. We add the passenger again to Passengerstatus

with the modification on insideelevator so that the passenger can now be delivered through UpdateNbPassengers(pos) (case 1). We also set his requestedfloor to true in floors so that the elevator will stop there.

```
count = 0;
index = 0;
if (notretreived > 0)                                                                    // If so
{
    while (differentFloor.Count != 0 && count != notretreived)
    {
        if (differentFloor[index].initialposition == initialpos && count != notretreived)
        {
            Passenger pass = differentFloor[index];
            differentFloor.Remove(pass);                                  // We remove them from diff
            Passengerstatus.Remove(pass);                                 // We do the same for Passe
            requests.Enqueue(pass);                                       // We add the passenger aga
            count++;
            if (index > 0) { index--; }
        }                                                                                // The
        else { index++; }
    }
}
}
```

b) Not retrieved >0 (Screenshot above) : In the 2nd while loop, the condition implies that if there is a passenger in differentFloor who's initial position is equal to initialpos and count!=notretrieved, this passenger wasn't retrieved by the elevator(as the passenger retrieved were processed before). We remove the passenger from differentFloor (he might be added again) and PassengerStatus (he will be added again). We add the passenger again to the queue requests as he wasn't retrieved so that his request will be processed later. As a result, the passengers who weren't retrieved won't have to call the elevator again.

11. Attributes of Elevator2

```
public class Elevator2
{
    public Queue<Passenger> requests { get; set; }
    public Queue<Passenger> goingdown { get; set; }
    public int elevatorposition = 0;
    public bool[] up { get; set; }
    public bool[] down { get; set; }
    List<int> PassengerPosition { get; set; }
    public int nbpassengers;
    public List<Passenger> Passengerstatus { get; set; }
    public int nbpassengersretrieved { get; set; }
    public int nbpassengersdelivered { get; set; }
    public List<Passenger> differentFloor { get; set; }
```

The only attributes that differ from those of class Elevator1 are :

1) Bool [ ] up : indicates for each floor whether the elevator should stop (true) or not (false) when the elevator is going up.
2) Bool [ ] down : down [] indicates for each floor whether the elevator should stop (true) or not (false) when the elevator is going down
3) Queue<Passenger> goingdown : permits to store the requests for the passengers who want to go down

12. Move() Elevator2 :

```
public void Move()
{
    Passenger passenger = null;
    string direction = "up";
    while (requests.Count != 0 )
    {
        Console.WriteLine(requests.Count);
        Console.WriteLine(goingdown.Count);
        while (requests.Count != 0)
        {
            passenger = requests.Dequeue();
            if (passenger.direction == "up")
            {
                Process(passenger, up);
            }

            else
            {
                goingdown.Enqueue(passenger);
            }
        }
        if (direction == "up")
        {
            Up();
            direction = "down";
        }
        if (direction == "down")
        {
            while (goingdown.Count != 0)
            {
                Process(goingdown.Dequeue(), down);
            }
            Down();
            direction = "up";
        }
    }
}
```

At the beginning the direction of the elevator is set to up so that the elevator will start by going up.

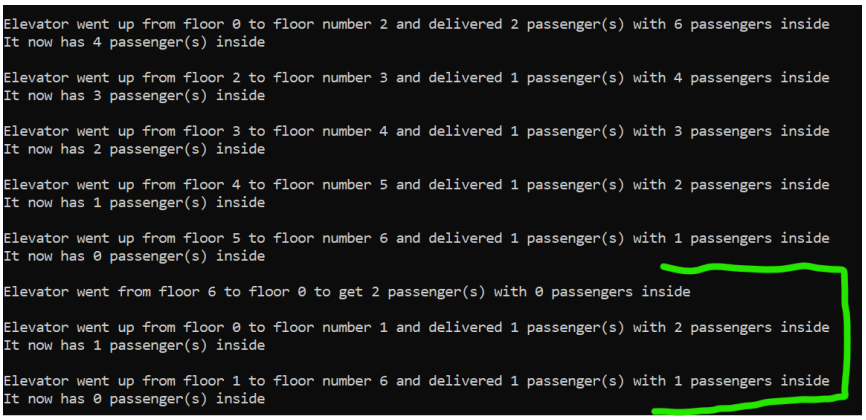
The 1st loop implies that the elevator will stop moving if they are no more passengers in requests. We only need to check requests as in the method, the elevator will go constantly up and down (goingdown will be empty after the process) each time the 1st loop is executed. Also, each passenger can be added again only at first in the queue requests.

The 2nd while loop is used to have zero elements in requests which contains all the requests at the beginning. The passengers going up are processed and those going down are added to the queue goingdown. The elevator moves given direction and then direction is set to the opposite. In the case direction is down we use a while loop to process every request of the queue goingdown.

13. Full capacity and innovation

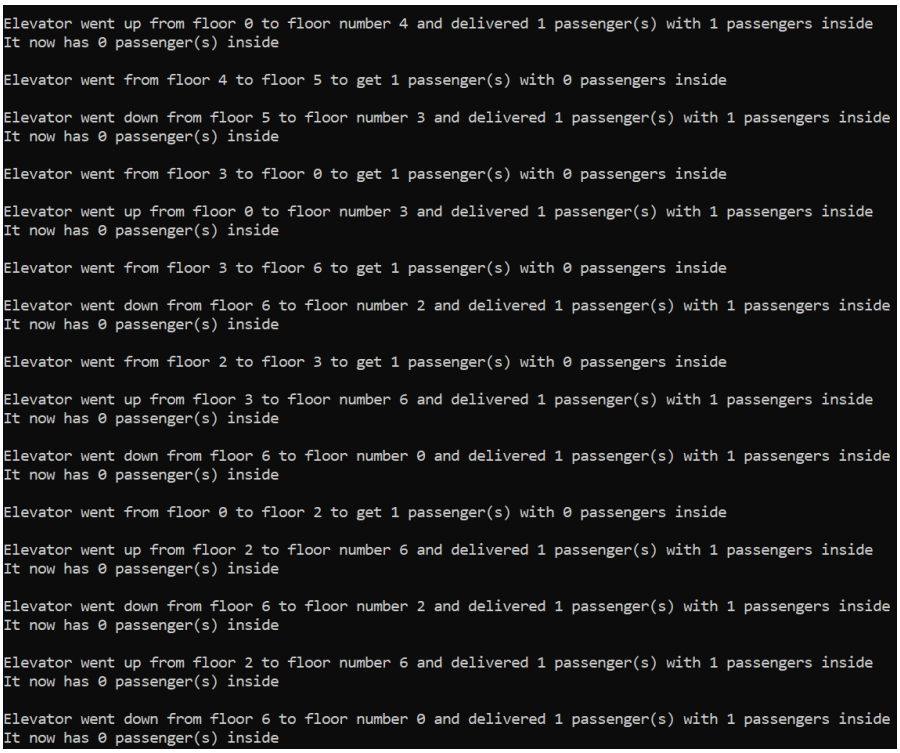
```
Elevator1 elevator = new Elevator1();
elevator.Request(new Passenger(0,2,"up"));
elevator.Request(new Passenger(0,2,"up"));
elevator.Request(new Passenger(0,6,"up"));
elevator.Request(new Passenger(0,3,"up"));
elevator.Request(new Passenger(0,4,"up"));
elevator.Request(new Passenger(0,5,"up"));
elevator.Request(new Passenger(0,6,"up"));
elevator.Request(new Passenger(0,1,"up"));
elevator.Move();
```

There are 8 passengers requesting to go up from floor 0. The last two passenger's requests (framed in red screenshot above) will be treated at the end (in green screenshot below) as the elevator reaches full capacity for the previous request. These 2 passengers won't have to call again the elevator (innovation)



The screenshot above concerns the Elevator1 and the result for the Elevator2 is the same. This shows that both classes Elevator1 and Elevator2 create an elevator able to serve multiple requests in parallel.

14. Elevator 1 : result with 15 moves



As we can see the screenshot above shows the main flaw of the First Come First Serve algorithm for Elevator1 which is the extra motions of up and down. In fact, once the direction changes, the elevator will process the next most recent request and go in the opposite direction so the elevator will move much more than for the algorithm used on the class Elevator2. In this specifical case, the elevator created from the class Elevator1 doesn't serve multiple requests at the same time.