|  |  |
| --- | --- |
| A screen shot of a computer  Description automatically generated  Lift simulation  Control Assessment | Abstract  In this report I will be delighting you with the development of my lift simulation solution for the problem. Where, the problem is to develop the most cost-efficient lift simulation compared to the base case.  690064952 150510  Data Structures and Algorithms |

Table of Contents

[General Description 2](#_Toc38834943)

[Data Structures and Algorithms 3](#_Toc38834944)

[Classes and Objects 3](#_Toc38834945)

[Queue 3](#_Toc38834946)

[Linked List 3](#_Toc38834947)

[Array 3](#_Toc38834948)

[List 4](#_Toc38834949)

[Hashing 4](#_Toc38834950)

[Merge Sort 4](#_Toc38834951)

[Shortest Path Algorithm 4](#_Toc38834952)

[Divide and Conquer Algorithm 4](#_Toc38834953)

[Weekly Log 6](#_Toc38834954)

[Performance Analysis 13](#_Toc38834955)

[Base Case Analysis 13](#_Toc38834956)

[Advance Case Analysis 13](#_Toc38834957)

[Comparsion 14](#_Toc38834958)

[References 15](#_Toc38834959)

[Video 15](#_Toc38834960)

[Soruce Code 16](#_Toc38834961)

[Mechanical Lift System (Base Case) 16](#_Toc38834962)

[Model Package 16](#_Toc38834963)

[Panel Package 33](#_Toc38834964)

[API Package 45](#_Toc38834965)

[Main Package 46](#_Toc38834966)

[Graph Python File 51](#_Toc38834967)

[Improved Lift System (Advance Case) 52](#_Toc38834968)

[Model Package 52](#_Toc38834969)

[Panel Package 70](#_Toc38834970)

[API Package 82](#_Toc38834971)

[Main Package 84](#_Toc38834972)

[Graph Python File 89](#_Toc38834973)

Lift Simulation

# General Description

The mechanical lift system (base case) contains an algorithm that only permits the lift changing direction (e.g. up or down) after it reaches its maximum floor it can travel to. To clarify this in more detail, assume there is a building implemented with the mechanical lift system for 10 floors and a lift is traveling up and currently in floor 5. Let’s suppose there is a passenger who is waiting to board the lift at floor 4. Rather than the lift going straight to floor 4 by changing direction to down, thanks to the mechanical lift system the lift will first go to floor 10 then change direction to down and travel to floor 4. This implies that the mechanical lift system is extremely inefficient. From the example we can see that rather than performing 1 move the mechanical lift system will take 11 moves to get to the passenger.

Therefore, I am going to share my approach to implementing a more efficient lift system (advance case). To improve efficiency, I used multiple lifts and shortest path algorithm, also, I tried to implement lift(s) being disturbed for sets of floors (like the image on the cover page, but it didn’t work for passengers over 1000).

My rationale is that the cost will always be 3 or below in average.

I believe my lift system proposal is better than the mechanical lift system, since my proposal is 6 times more efficient than the mechanical lift system in average.

# Data Structures and Algorithms

## Classes and Objects

I am using Java program language to implement my lift system and Java is an object ordinated program. Since I am planning to have multiple lifts, implementing a lift class and initialising the lift class as objects of the number of lifts I need.

Furthermore, object-oriented programming methods make code more maintainable. Identifying the source of errors is easier because objects are self-contained. Also, because objects contain both data and methods that act on data, objects can be thought of as self-contained black boxes. This feature makes it easy to reuse code in new systems. Messages provide a predefined interface to an object's data and functionality. With this interface, an object can be used in any context. Finally, object-oriented programs are also scalable. As an object's interface provides a road map for reusing the object in new software and provides all the information needed to replace the object without affecting other code. This way aging code can be replaced with faster algorithms and newer technology.

## Queue

The main advantage of using queue is that it is a dynamic data structure, since it doesn’t have a fixed length of size. I used a queue data structure to store the passenger who is waiting to board a lift. This is because a queue structure follows a special structure as first in and first out. This use of structure makes my simulation more realistic, since passenger who is waiting to board first will board first and who is waiting to board last will board last. Furthermore, A queue allows for O(1) insertion from the end and O(1)deletion from the front.

## Linked List

The main advantage of using linked list is that it is a dynamic data structure, since it doesn’t have a fixed length of size. Also, this data structure stores elements in a sequential structure. Using a linked list to store the queue data structure of the passenger who is waiting to board. Linked list is a perfect storage structure to control waiting passenger. For example, with a linked list I can add new passenger to the end of the linked list. Similarly, when the full becomes full then the passenger who boarded the lift last will alight the lift and add to the front of the linked list, since they are returning passenger who wasn’t able to board. The concept of over following of values cannot be arising in link list. Since, Memory can be physically released using the free function for latter allocation of nodes. The chances of memory wastage are minimum because memories are allocated dynamically as the requirement of the user.

## Array

I stored the floor objects and lift objects in an array, since I know size of the storage structure. Also, the time complexity of insertion, deletion and find has O(1). In addition, it is better and convenient way of storing the data of same datatype with same size. It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays. Furthermore, Iterating the arrays using their index is faster compared to any other methods like linked list etc.

## List

The arrival floor (passenger’s source floor) is stored in a list. This is because the lift will move to the floor, which is in the arrival floor, since it notify the lift that it is waiting to be boarded. List is a dynamic structure meaning the length of the list isn’t pre-determine, so the chances of memory wastage are minimum because memories are allocated dynamically as the requirement of the user. This means that the list can be resized when a new element is added, and non-unwanted memory space is pre-allocated.

## Hashing

I used a hashing method to store the passenger objects. Hashing provides a more reliable and flexible method of data retrieval than any other data structure. In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches and sets. It contains unique element. There would be no null value and no null key. It is synchronized. I used this since the hashing method will returns the hash code of a non-null argument.

## Merge Sort

I used to merge sort to sort the arrival floor inserted by the passengers. The advantages to merge sort are it is always fast. Even in its worst case its runtime is O(nlogn). It is also stable.

Since this divide the elements in an array into two sub-arrays of n/2 elements of each sub-arrays and sort the two sub-arrays recursively using merge sort. Finally, merging the two sorted sub-arrays of size n/2 to produce the sorted arrays of n elements.

## Shortest Path Algorithm

I written an algorithm that finds the shortest path for the lift to travel. This algorithm is very important, because thanks to this algorithm the cost of my lift system is a scale factor of 0.01 of the mechanical lift systems. Even in its worst case its runtime is O(n).

## Divide and Conquer Algorithm

This divide and conquer algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm work by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. A typical Divide and Conquer algorithm solve a problem using following three steps.

* Divide: Break the problem into sub-problems of same type.
* Conquer: Recursively solve these sub-problems.
* Combine: Combine the solution sub-problems.

The most recognizable benefit of the divide and conquer paradigm is that it allows us to solve difficult problem, such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it. However, with the divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would. It also uses memory caches effectively. When the sub problems become simple enough, they can be solved within a cache, without having to access the slower main memory.

# Weekly Log

|  |  |  |  |
| --- | --- | --- | --- |
|  | Date and Time | Progress | Hours |
| 2 | 22th January  15:00 – 16:30 | Introduction   * Researching Java and Python to decide which programming language to use when implementing the lift simulation. * Deciding which programming style to write the lift simulation. For example, object-oriented, functional or scripted programming. * Analysing the specification: implementing two applications, where one is a base case, and another is an advance case. * Fully understanding what the client wants before starting the plan. * Finally, I have decided to create the lift simulations in Java as an object-oriented program. This is because the simulation will be well structured and easier to maintain/test in the future. | 1.5 |
| 3 | 27th January  11:00 – 17:00 | First implementing a base case that satisfies the minimum requirement.  Planning   * Designing a simple use case: to physical see how the client will interact with the application. As a result of humans being unpredictable, and I need a know where/how the application relies on the user, so it can be controlled/validated. Luckily, from the use case I see that the user only interacts with the menu, when selecting the building’s floor, passenger volume and lift’s weight. Therefore, the simulation runs independently. * Object-orientated programming: understand the scenario as objects. I know to have 4 main classes that are the backbones of the lift simulation. There 4 main classes are: Lift, Passenger, Floor and Building. * Listing all the attributes that is associated with each of the 4 main classes. For example, Lift class will have height, width, passengers, etc. * Designing a UML class diagram including all the attributes and behaviours of the classes. Also, thinking about the data that needs to be shared with other classes. This is the prevent any non-needed information to be send around the Java project. * As well as deciding if there is any relationship between all the classes, such as “is-a”, “has-a” and etc. * Relationship: Building has a Lift. Building has Floors. Floor has Passengers. Lift has Passengers. | 6 |
|  | 28th January  11:00 – 15:00 | Implementation   * I decided to create two different simulation for each case, where I will have a thread running in the background as a back-end simulation and another thread to display the changes in the back-end simulation to the front-end simulation. * Therefore, I appended my UML class diagram to contain a “model” package as well as a “panel” package. Where the “model” package will contain all the classes involved with back-end simulation. Similarly, “panel” package will contain all the classes involved with front-end simulation. * As well as deciding if there is any relationship between all the classes, such as “is-a”, “has-a” and etc.   Setting up the project   * I will be using IntelliJ to develop a program in Java that runs a lift simulation. * Creating a new project called “LiftSimulationBaseCase”. * Inside the “src” (source) folder I created a package “com. simulation.lift”. * Creating the following packages: main, model and panel. | 4 |
|  | 29th January  14:00 – 18:00 | Start Implementing the “model” package   * This package will contain all the classes that will perform the back-end simulation.   Implementation of the Lift class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are floor, maxFloor, minFloor, maxWeight, height, width, name, liftMode, liftDirection, passengers, arriveFloors and departFloors. * Implementing getter/setter methods for the Lift class that will be edited or required by other classes in the project. The getter methods are getHeight, getWidth, getMaxFloor, getMinFLoor, getName and getMaxMoveDistance. The setter methods are setArriveFloor. | 4 |
| 4 | 3rd February  11:00 – 17:00 | Implementation of the Lift class   * I came to the conclusion of using enum state machine for control the states of a lift. * Initialisation an enum class containing the following states: up, down, open, close, wait, board, alight and full. I have a “full” state as to control the lift when its capacity becomes full. * Implementing the states in a method called “liftOperation”. By using a switch statement, I can implement each state as different cases. Creating the following sub-methods to help with this implementation: boardPassenger, alight, getUpDownMove, getUpMove and getDownMove. | 6 |
|  | 4rd February  11:00 – 15:00 | Implementation of the Passenger class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are weight, sourceFloor and destinationFloor. * Implementing getter methods for the Lift class that will be edited or required by other classes in the project. The getter methods are getWeight, getSourceFloor and DestinationFloor. * I also created a method called “hashCode”, which hash the Passenger object to a hash table and maps it to a unique hash code. This unify the Passenger objects and enable us to have order of 1, when retrieving the passenger in most cases. In addition, each hash table’s index is a linked list, when there is more than one element. | 4 |
|  | 5rd February  14:00 – 18:00 | Implementation of the Floor class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are floorNo, height, width and passengers. * Implementing getter methods for the Lift class that will be edited or required by other classes in the project. The getter methods are getPassenger. * Creating a method that adds next passengers to the end of the LinkList like a queue of FIFO, called “addNewPassenger”. * Similarly, creating a method that removes the first passenger in the queue after their boarded the lift, called “removePassenger”. * Finally, implementing a method that returns first to the queue when the lift is full and cannot board it, called “addReturmPassenger”. | 4 |
| 5 | 10th February  11:00 – 17:00 | Implementation of the Building class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are maxFloor, minFloor, liftWeight, liftHeight, liftWidth, floorHeight, floorWidth, and floors. * Implementing getter/setter methods for the Lift class that will be edited or required by other classes in the project. The getter methods are getLift, getFloors, getMaxFloor, getFloorHeight and get liftWidth. * Also, implementing a method that initialises the floor objects as well as the lift objects. Since we declared in the UML diagram that building has floors and building has a lift. | 6 |
|  | 11th February  11:00 – 13:00 | Testing the “model” package   * Checking if the classes were implemented correctly. * Checking if the all the attributes and behaviours have been implemented correctly. * Checking is the relationship is handled correctly. * Checking for any syntax, logical and run-time errors. | 2 |
|  | 12th February  14:00 – 17:00 | Refactoring the “model” package   * Removing any redundant code. * Refactoring the implemented code. * Reducing similar/repeated code. * Testing if everything works as it supposed to. | 3 |
| 6 | 17th February  11:00 – 17:00 | Start Implementing the “panel” package   * This package will contain all the classes that will perform the front-end simulation.   Implementation of the LiftPanel class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are x, y, width, height, backgroundColor and lift. * Implementing getter methods for the Lift class that will be edited or required by other classes in the project. The getter methods are getLift and getX. * Also, implementing a method that will animate the life moving up in the building called “moveUp”. * Similarly, implementing a “moveDown” method that will animate the lift moving down in the building. | 6 |
|  | 18th February  11:00 – 15:00 | Implementation of the LiftPanel class   * In addition, the “openDoor” function animate the lift of the door opening in the building to pick/drop passengers. * Like “openDoor”, I implemented another method that animate the lift of the door closing in the building for the lift to move up/down. The method is called “closeDoor”. * Finally, a “paintComponent” method that will draw the lift panel inside the building panel, which will be visual for the client. | 4 |
|  | 19th February  14:00 – 18:00 | Implementation of the PassengerPanel class   * While implementing the PassengerPanel class I realised that there isn’t point in creating an entirely new class that will display the passenger. This is because I know that passengers are only being displayed in the floor. Also, to prevent tight coupling I just merge the passenger GUI class and floor GUI class to together into one. | 4 |
| 7 | 24th February  11:00 – 17:00 | Implementation of the FloorPanel class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are x, y, width, height, backgroundColor, floor, list of passengers, move passenger and passenger state. * Implementing getter methods for the Lift class that will be edited or required by other classes in the project. The getter methods are getFloor. * I came to the conclusion of using enum state machine for control the states of a passenger. * Initialisation an enum class containing the following states: board, alight and wait. Where, the ‘wait’ state is used to indicate the system that the passenger is waiting in the floor to be boarded. * Implementing a “flashFloor” method that draws the floor panel with any existing passengers in the building panel. | 6 |
|  | 25th February  11:00 – 15:00 | Implementation of the FloorPanel class   * Also, implementing “flashBoardFloor” function, which draws the passenger moving toward the lift to present passengers boarding a lift. * Similarly, “flashAlightFloor” draws the passengers moving away from the lift and existing the floor panel. * Finally, a “paintComponent” method that will draw the floor panel inside the building panel, which will be visual for the client. | 4 |
|  | 26th February  14:00 – 18:00 | Implementation of the BuildingPanel class   * Implementing the attributes mention in the UML Lift diagram in the constructors. The attributes are x, y, width, height, backgroundColor, building, lift, list of floors, number of passengers, maxFloor, maxLift, liftWeight and isAuto state. | 4 |
| 8 | 2nd March  11:00 – 17:00 | Implementation of the BuildingPanel class   * A “createLift” method initialise life panel objects which will be presented in the building. * Similarly, “createFloors” initialise floor panel object which will be presented in the building. * In addition, implementing a method that initialises the floor panel objects as well as the lift planel objects. Since we declared in the UML diagram that building has floors and building has a lift. * Finally, a “paintComponent” method that will draw the building panel inside the building panel, which will be visual for the client. | 6 |
|  | 3rd March  11:00 – 13:00 | Testing the “panel” package   * Checking if the classes were implemented correctly. * Checking if the all the attributes and behaviours have been implemented correctly. * Checking is the relationship is handled correctly. * Checking for any syntax, logical and run-time errors. | 2 |
|  | 4th March  14:00 – 17:00 | Refactoring the “panel” package   * Removing any redundant code. * Refactoring the implemented code. * Reducing similar/repeated code. * Testing if everything works as it supposed to. | 3 |
| 9 | 9th March  11:00 – 19:00 | To link the contain in the model package and the panel package I need to common interface to share the back and front-end data.  Therefore, I create a new package called “api” that will contain the interface class to share the data between front and back end.  Implementation of the LiftListener interface class   * Declaring “upMoves” method, which moves the lift up depending on certain number of moves. * Similarly, “downMoves” method, which moves the lift down depending on certain number of moves. * In addition, “getBoardPassenger” gets the passenger who are boarding a lift. * Similarly, the method “alightPassenger” removes the passengers who are alighting from a lift. * “openDoor” method opens the door of a lift. * Similarly, “closeDoor” method closes the door of a lift. * I also declared “started” function, which indicates the lift has started. * Finally, I declared “stopped” function, which indicates the lift has stopped.   Implementing the LiftListener interface class in the BuildingPlanel class. So, the BuildingPanel class controls the threads and synchronizing the back and front end of the program for it can be mentioned. As a result, is will allow the same data being processed at front and back end. Furthermore, using object waitFor and notify allow the threads to wait and notify, so the front and back end processes are synchronised. | 8 |
|  | 10th March  11:00 – 15:00 | Testing the “api” package and the whole project   * Checking if the classes were implemented correctly. * Checking if the all the attributes and behaviours have been implemented correctly. * Checking is the relationship is handled correctly. * Checking for any syntax, logical and run-time errors. | 4 |
|  | 11th March  13:00 – 18:00 | Refactoring the “api” package and the whole project   * Removing any redundant code. * Refactoring the implemented code. * Reducing similar/repeated code. * Testing if everything works as it supposed to.   Refactoring all the project panel’s dimension to make the GUI more appealing.  Similarly, checking if what is represented in the back end is represent in the front end.  For example,   * the axis (0,0) stats on the top left and not on the bottom left of the Java Swing frame. * the floors are represented in the correct order. Where, the floors start from the bottom and not the other way around. * the passengers are displayed on the correct source floor and alight on the correct destination floor. * the number of moves the lift perform is correct. | 5 |
| 10 | 16th March  11:00 – 19:00 | Finally, implementing an advance case of the lift simulation project.   * The advance case is similar to the base case, where the functionality is the same, but unlike the base case this is aimed to maximise the requirement.   Planning   * Similar to the base case planning. * To maximise the requirement, I added multiple lifts, and distrusting the list of lifts to set of floors.   Setting up the project   * I will be using IntelliJ to develop a program in Java that runs a lift simulation. * Creating a new project called “LiftSimulationAdvanceCase”. * Inside the “src” (source) folder I created a package “com. simulation.lift”. * Same as the base case, creating the following packages: main, model and panel.   Start Implementing the “model” package   * This package will contain all the classes that will perform the back-end simulation.   Implementation of the Lift class   * Same as the class from the base case, but with an algorithm that finds the shortest path while still functionating as a lift. Also, the backbone of both classes is the same. | 6 |
|  | 17th March  11:00 – 15:00 | Implementation of the Passenger class   * This is the same as the base case.   Implementation of the FloorPanel class   * This is the same as the base case.   Implementation of the Building class   * Initialising multiple lift objects in an array.   Start Implementing the “panel” package   * This package will contain all the classes that will perform the front-end simulation.   Implementation of the LiftPanel class   * This is the same as the base case.   Implementation of the FloorPanel class   * This is the same as the base case. | 4 |
|  | 18th March  13:00 – 18:00 | Implementation of the BuildingPanel class   * Initialising multiple lift panel objects in an array. * Assigning individual threads for each life, so it performs independently.   Testing the whole project   * Checking if the classes were implemented correctly. * Checking if the all the attributes and behaviours have been implemented correctly. * Checking is the relationship is handled correctly. * Checking for any syntax, logical and run-time errors.   Refactoring the whole project   * Removing any redundant code. * Refactoring the implemented code. * Reducing similar/repeated code. * Testing if everything works as it supposed to. | 5 |
|  | 7th April  11:00 – 17:00 | Reaching how to draw graph in Java   * Since I extra time thanks to the extension, I decided to make the program draw a graph to represent the performance analysis. * Trying to draw the 3d graph. * Tried Java3d – failed since it isn’t supported for Java 8+. * Tried JavaFx – failed since not being compatible with MacOS. * Tried Jzy3D – failed since I can’t use it in IntelliJ. | 6 |
|  | 9th April  11:00 – 15:00 | Drawing graph in Java   * Solution using Matplotlib in Python to draw the graph. * Researching how to run Python program in Java. * One solution is trying Jython, however it wasn’t compatible with MacOs. | 4 |
|  | 20th April  14:00 – 18:00 | Solution using Java’s in-built library process and runtime, where I create a python runtime environment for the .py file to be executed.  I used a python dictionary structure to store sets of data in Jason form.  These data will be used to plot a 4D graph on matplotlib. Where the graph we present cost against lift capacity, number of passengers and number of floors. | 4 |

# Performance Analysis

## Base Case Analysis

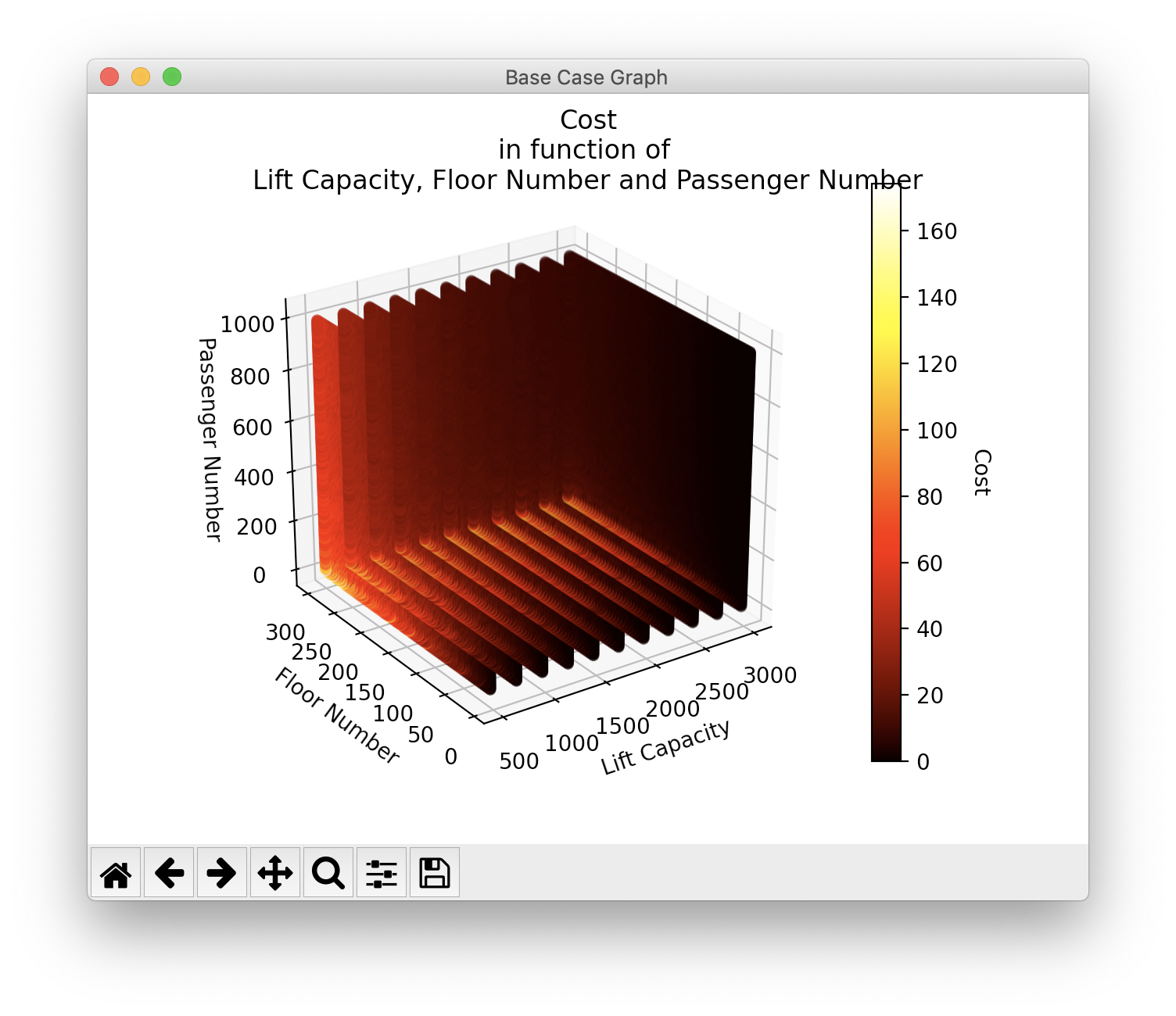
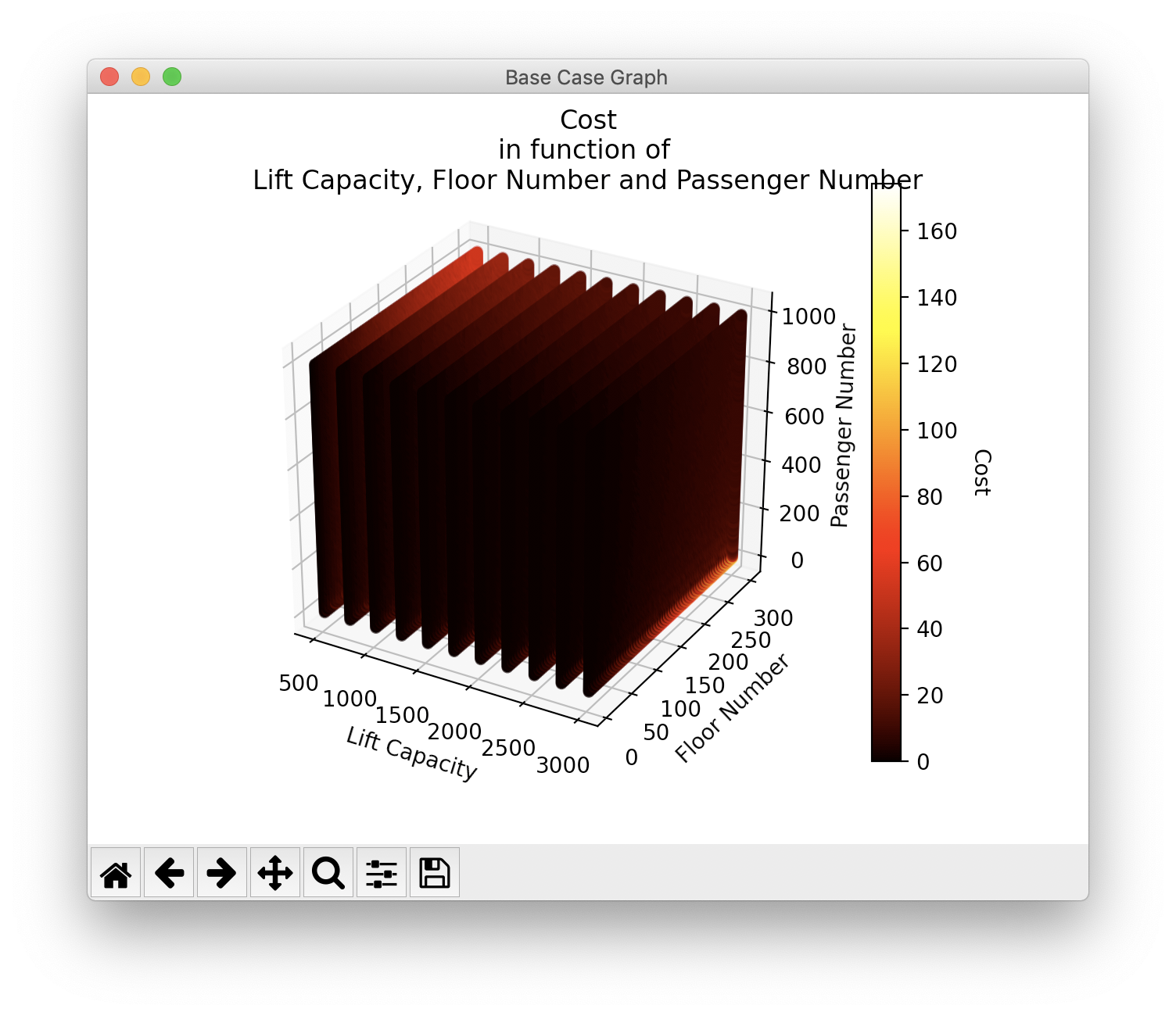


Figure 1

(a)

(b)



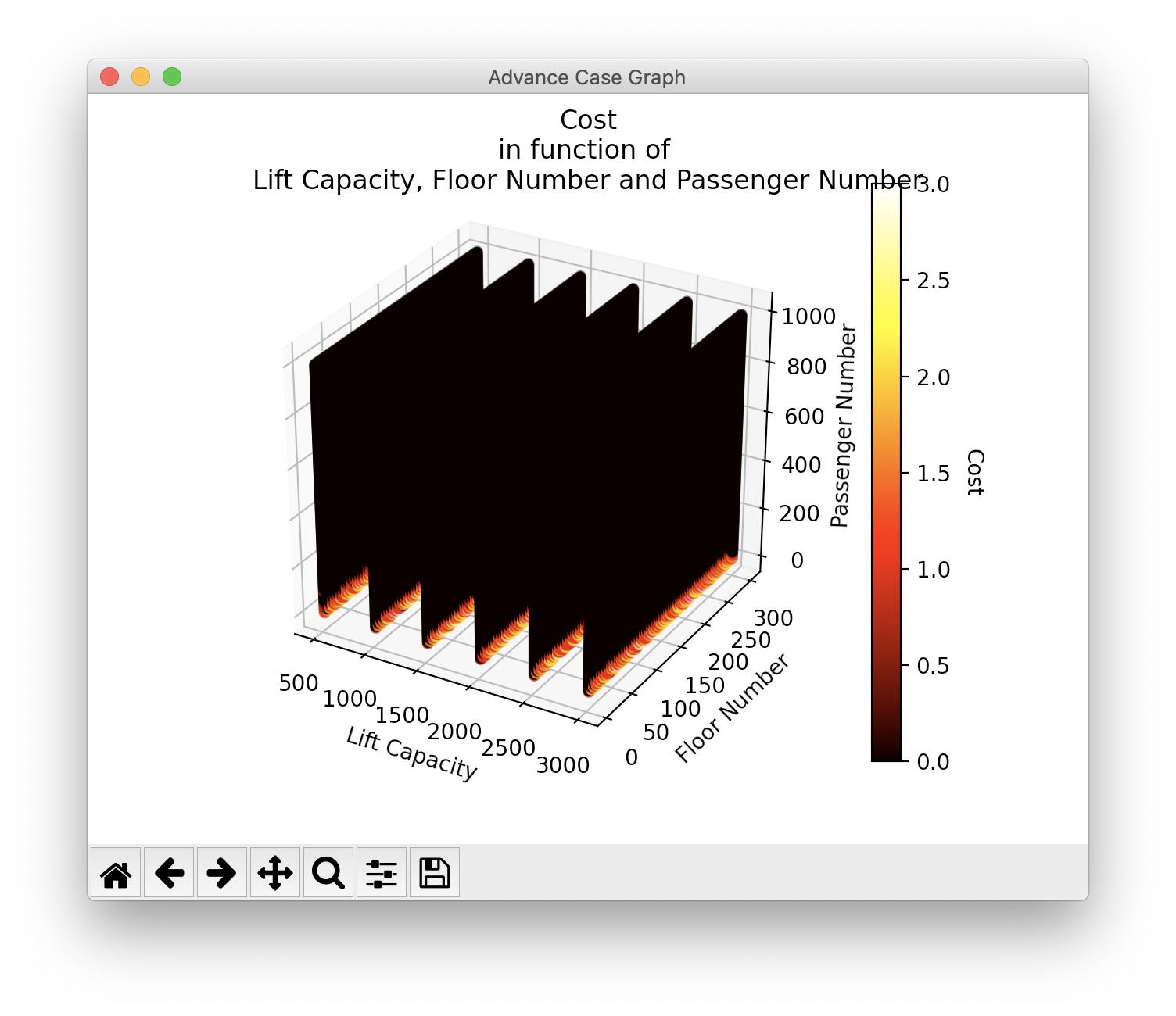
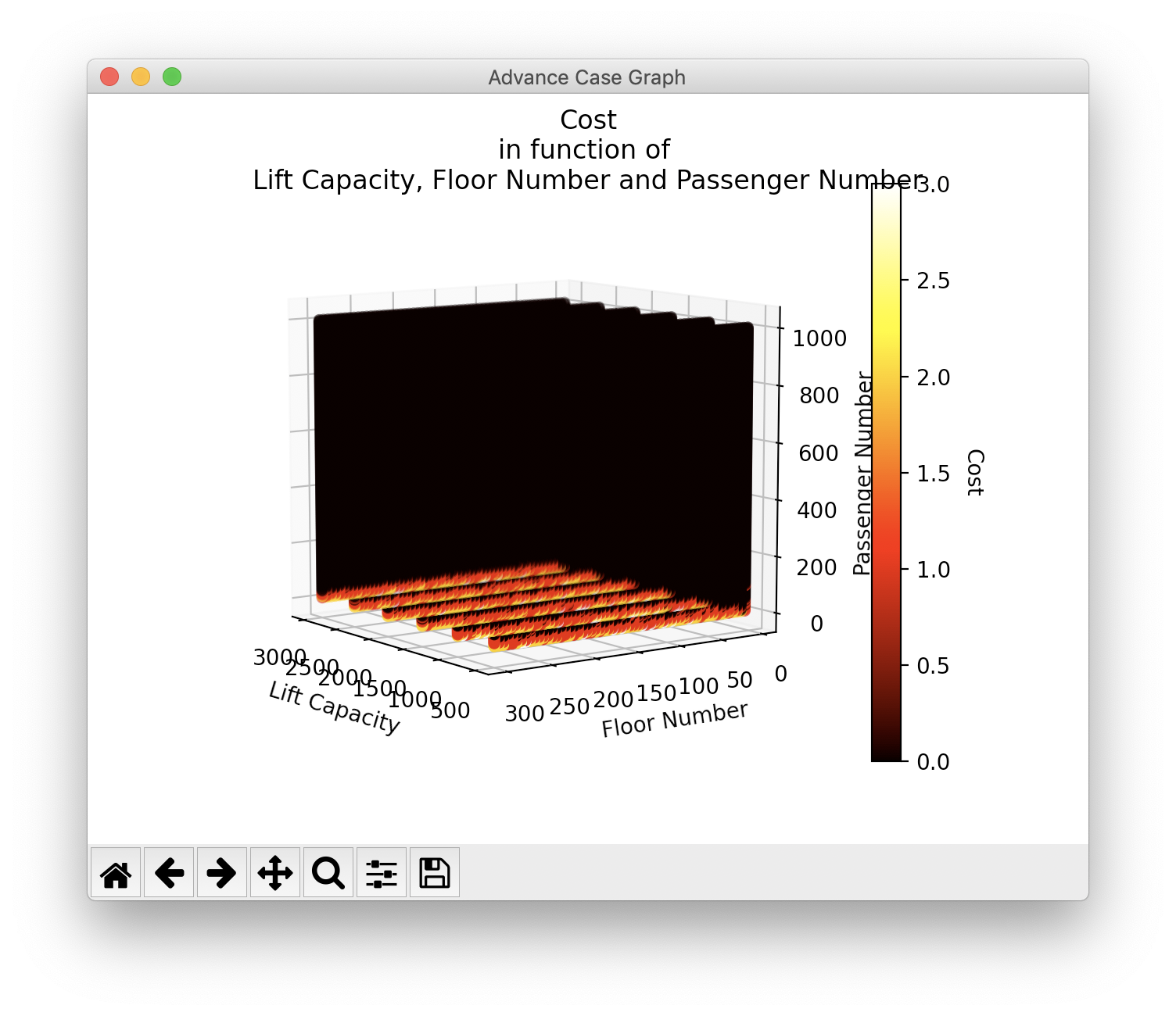
The performance of the mechanical lift system is presented in figure 1. Figure 1 represent a 4D graph of cost per lift against lift capacity, floor number and passenger. From figure 1 (a) we can see that the highest cost is 170. However, from the scatter graph we can see that the high distribution of cost is less than 20. In situation like the scenario presented in figure 1 (b), where the cost is 170 when the lift capacity is 500, floor number is 300 and finally 5 passengers. This makes sense as in a worse case the maximum cost will be 600.

## Advance Case Analysis

Figure 2

(a)

(b)



The performance of the advance lift system is presented in figure 2. Figure 2 represent a 4D graph of cost per lift against lift capacity, floor number and passenger. From figure 2 (a) we can see that the highest cost is 3. However, from the scatter graph we can see that the high distribution of cost is less than 20. In situation like the scenario presented in figure 2 (b), where the cost is 3 when the lift capacity is 500, floor number is 300 and finally 5 passengers.

## Comparsion

From both figures we can see that figure 2 is dramatically more efficient then figure 1. During the worst case of the advance lift system is 50+ times more efficient then the worst case of the mechanical lift system. Furthermore, on average the advance lift system is infinitely for efficient than the average mechanical lift system. Therefore, I can state that I have successfully implemented a system that is more cost efficient then the mechanical lift system.

# References

Mathplotlib () Pyplot tutorial, Available at: https://matplotlib.org/tutorials/introductory/pyplot.html (Accessed: 08/04/2020).

Tutorials Point () Java Tutorial, Available at: https://www.tutorialspoint.com/java/index.htm (Accessed: 22/01/2020).

Tutorials Point () SWING Tutorial, Available at: https://www.tutorialspoint.com/swing/index.htm (Accessed: 22/01/2020).

# Video

Private video - https://www.youtube.com/watch?v=PFTWkxvpKRo&feature=youtu.be

# Soruce Code

## Mechanical Lift System (Base Case)

### Model Package

#### Lift Class

package com.simulation.lift.model;  
  
import com.simulation.lift.api.LiftListener;  
  
import java.util.\*;  
import java.util.stream.Collectors;  
  
public class Lift {  
 */\*\*  
 \* Enum presenting the states of a lift.  
 \*/* public enum LiftMode {  
 *UP*,  
 *DOWN*,  
 *OPEN*,  
 *CLOSE*,  
 *WAIT*,  
 *BOARED*,  
 *ALIGHT*,  
 *FULL* }  
  
 private int floor;  
  
 private final int maxFloor;  
 private final int minFloor;  
 private int maxWeight;  
 private final int height;  
 private final int width;  
 private final String name;  
  
 private int totalBoardPassengers;  
 private int totalAlightPassengers;  
 private int totalMoves;  
  
 private LiftMode liftMode;  
 private LiftMode liftDirection;  
  
 private final List<Passenger> passengers;  
 private final static List<Integer> *arriveFloors* = Collections.*synchronizedList*(new ArrayList<>());  
 private final List<Integer> departFloors;  
  
 private LiftListener listener;  
  
 private boolean isRunning;  
  
 private Object object = new Object();  
  
 private final boolean isAuto;  
 private final boolean isBackEnd;  
  
 */\*\*  
 \* Constructs an instance of the object containing maxFloor, minFloor,  
 \* maxWeight, height, width and name arguments.  
 \*  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *minFloor minimum floor number  
 \** ***@param*** *maxWeight maximum capacity  
 \** ***@param*** *height height of a lift  
 \** ***@param*** *width width of a lift  
 \** ***@param*** *name name of the lift  
 \** ***@param*** *isAuto is the program auto  
 \** ***@param*** *isBackEnd is the program only for back-end  
 \*/* public Lift(int maxFloor, int minFloor, int maxWeight, int height, int width, String name, boolean isAuto, boolean isBackEnd) {  
 this.floor = minFloor;  
  
 this.maxFloor = maxFloor;  
 this.minFloor = minFloor;  
 this.maxWeight = maxWeight;  
 this.height = height;  
 this.width = width;  
  
 this.name = name;  
  
 this.isAuto = isAuto;  
 this.isBackEnd = isBackEnd;  
  
 this.liftMode = LiftMode.*WAIT*;  
 this.liftDirection = LiftMode.*WAIT*;  
  
 if(!this.isAuto) {  
 this.liftMode = LiftMode.*UP*;  
 }  
  
 this.passengers = new ArrayList<>();  
 this.departFloors = new ArrayList<>();  
 }  
  
 */\*\*  
 \* Method sets a listener object to the Lift class.  
 \*  
 \** ***@param*** *listener listener object  
 \*/* public void setListener(LiftListener listener) {  
 this.listener = listener;  
 }  
  
 public void setLiftMode(LiftMode mode) {  
 this.liftMode = mode;  
 }  
  
 */\*\*  
 \* Method gets the height of a lift.  
 \*  
 \** ***@return*** *lift's height  
 \*/* public int getHeight() {  
 return height;  
 }  
  
 */\*\*  
 \* Method gets the width of a lift.  
 \*  
 \** ***@return*** *lift's width  
 \*/* public int getWidth() {  
 return width;  
 }  
  
 */\*\*  
 \* Method gets the maximum floor the lift can travel to.  
 \*  
 \** ***@return*** *lift's maximum floor  
 \*/* public int getMaxFloor() {  
 return maxFloor;  
 }  
  
 */\*\*  
 \* Method gets the minimum floor the lift can travel to.  
 \*  
 \** ***@return*** *lift's minimum floor  
 \*/* public int getMinFloor() {  
 return minFloor;  
 }  
  
 */\*\*  
 \* Method gets the name of the lift.  
 \*  
 \** ***@return*** *lift's name  
 \*/* public String getName() {  
 return this.name;  
 }  
  
 */\*\*  
 \* Method gets the total number of passengers, who have boarded the lift.  
 \*  
 \** ***@return*** *total boarded passengers  
 \*/* public int getTotalBoardPassengers() {  
 return totalBoardPassengers;  
 }  
  
 */\*\*  
 \* Method gets the total number of passengers, who have alighted from the lift.  
 \*  
 \** ***@return*** *total alighted passengers  
 \*/* public int getTotalAlightPassengers() {  
 return totalAlightPassengers;  
 }  
  
 */\*\*  
 \* Method gets the total moves completed by the lift.  
 \*  
 \** ***@return*** *total moves  
 \*/* public int getTotalMoves() {  
 return totalMoves;  
 }  
  
 */\*\*  
 \* Method to start the lift simulation in the back-end.  
 \*/* public void start() {  
 this.isRunning = true;  
 liftOperation();  
 }  
  
 */\*\*  
 \* Method to stop the lift simulation in the back-end.  
 \*/* public void stop() {  
 this.isRunning = false;  
 synchronized (object) {  
 object.notifyAll();  
 }  
 }  
  
 public int getFloor() {  
 return floor;  
 }  
  
 */\*\*  
 \* Method gets the maximum distance between the maxFloor and minFloor.  
 \*  
 \** ***@return*** *maximum distance  
 \*/* public int getMaxMoveDistance() {  
 return this.maxFloor - (this.minFloor - 1);  
 }  
  
 */\*\*  
 \* Method boards passengers to the lift.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \** ***@return*** *whether or not the passenger boarded  
 \*/* private boolean boardPassenger(Passenger passenger){  
 if (this.maxWeight - passenger.getWeight() < 0 ) {  
 return true;  
 }  
  
 if (this.passengers.add(passenger)) {  
 this.maxWeight -= passenger.getWeight();  
 if (!this.departFloors.contains(passenger.getDestinationFloor())) {  
 this.departFloors.add(passenger.getDestinationFloor());  
 this.departFloors.sort(Comparator.*naturalOrder*());  
 }  
 }  
 return false;  
 }  
  
 */\*\*  
 \* Method returns a list of alighting passenger from the lift.  
 \*  
 \** ***@return*** *list of passengers  
 \*/* private List<Passenger> alight() {  
  
 List<Passenger> alightPassenger = this.passengers.stream()  
 .filter(p->this.floor == p.getDestinationFloor())  
 .map(p -> {  
 this.maxWeight += p.getWeight();  
 return p;  
 })  
 .collect(Collectors.*toList*());  
  
 this.passengers.removeAll(alightPassenger);  
  
 return alightPassenger;  
 }  
  
 */\*\*  
 \* Method sets the arrival floors of passengers.  
 \*  
 \** ***@param*** *floorNO floor number  
 \*/* public void setArriveFloor(int floorNO) {  
 synchronized (*arriveFloors*) {  
 if (!*arriveFloors*.contains(floorNO)) {  
 *arriveFloors*.add(Integer.*valueOf*(floorNO));  
 *arriveFloors*.sort(Comparator.*naturalOrder*());  
 }  
 }  
  
 synchronized (object) {  
 object.notifyAll();  
 }  
 }  
  
 */\*\*  
 \* Method controls the states of the lift.  
 \*/* private void liftOperation() {  
 //Start Lift  
 this.listener.started(this.name);  
  
 boolean isWaitUp = false;  
 boolean isWaitDown = false;  
 boolean isFull = false;  
  
 while(isRunning) {  
 System.*out*.println("Thread "+this.name+ " LiftMode: "+this.liftMode.name());  
  
 try {  
 switch (this.liftMode) {  
 case *FULL*:  
 int move = getUpDownMove(liftDirection, this.floor);  
 if(move == -1){  
 liftDirection = liftDirection == LiftMode.*UP* ? LiftMode.*DOWN* : LiftMode.*UP*;  
 }  
 else if (liftDirection == LiftMode.*UP*) {  
 isFull = false;  
 this.listener.upMoves(this.name, this.floor, move);  
  
 this.floor += move;  
 *arriveFloors*.remove(Integer.*valueOf*(this.floor));  
 this.totalMoves += move;  
 this.liftMode = LiftMode.*OPEN*;  
 } else {  
 isFull = false;  
 this.listener.downMoves(this.name, this.floor, move);  
  
 this.floor -= move;  
 *arriveFloors*.remove(Integer.*valueOf*(this.floor));  
 this.totalMoves += move;  
 this.liftMode = LiftMode.*OPEN*;  
 }  
 break;  
  
 case *WAIT*:  
 synchronized (object) {  
 try {  
 object.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 this.liftMode = LiftMode.*UP*;  
 isWaitDown = false;  
 isWaitUp = false;  
 break;  
  
 case *UP*:  
 if (this.floor == this.maxFloor) {  
 this.liftMode = LiftMode.*DOWN*;  
 isWaitUp = true;  
 } else {  
 move = getUpMove(this.floor);  
 if (move == -1) {  
 if(passengers.isEmpty() && totalBoardPassengers == totalAlightPassengers && (*arriveFloors*.isEmpty() && departFloors.isEmpty())) {  
 if(!this.isAuto) {  
 stop();  
 break;  
 }  
 this.liftMode = LiftMode.*WAIT*;  
 }  
 else {  
 move = maxFloor - floor;  
 this.listener.upMoves(this.name, this.floor, move);  
 this.floor += move;  
 this.totalMoves += move;  
  
 this.liftMode = LiftMode.*DOWN*;  
 isWaitUp = true;  
 }  
  
 } else {  
 this.listener.upMoves(this.name, this.floor, move);  
  
 this.floor += move;  
 this.totalMoves += move;  
 liftDirection = LiftMode.*UP*;  
 this.liftMode = LiftMode.*OPEN*;  
 }  
 }  
  
 break;  
  
 case *DOWN*:  
 if (this.floor == this.minFloor) {  
 this.liftMode = LiftMode.*UP*;  
 isWaitDown = true;  
 } else {  
 move = getDownMove(this.floor);  
 if (move == -1) {  
 if(passengers.isEmpty() && totalBoardPassengers == totalAlightPassengers && (*arriveFloors*.isEmpty() && departFloors.isEmpty())) {  
 if(!this.isAuto) {  
 stop();  
 break;  
 }  
 this.liftMode = LiftMode.*WAIT*;  
 }  
 else {  
 move = floor - minFloor;  
 this.listener.downMoves(this.name, this.floor, move);  
 this.floor -= move;  
 this.totalMoves += move;  
  
 this.liftMode = LiftMode.*UP*;  
 isWaitDown = true;  
 }  
  
 } else {  
 this.listener.downMoves(this.name, this.floor, move);  
  
 this.floor -= move;  
 this.totalMoves += move;  
 liftDirection = LiftMode.*DOWN*;  
 this.liftMode = LiftMode.*OPEN*;  
 }  
 }  
  
 break;  
  
 case *OPEN*:  
 isWaitDown = false;  
 isWaitUp = false;  
  
 this.listener.openDoor(this.name, this.floor);  
  
 this.liftMode = LiftMode.*ALIGHT*;  
 break;  
  
 case *ALIGHT*:  
 //Get off passenger  
 for(Passenger passenger : alight()) {  
 this.totalAlightPassengers++;  
 this.listener.alightPassenger(this.name, this.floor, passenger);  
 if(!isBackEnd) {  
 *waitFor*();  
 }  
 }  
  
 departFloors.remove(Integer.*valueOf*(this.floor));  
 this.liftMode = LiftMode.*BOARED*;  
 break;  
  
 case *BOARED*:  
 //Get in passenger  
 Passenger ps = this.listener.getBoardPassenger(this.name, this.floor, null, false);  
 while(ps != null) {  
 isFull = boardPassenger(ps);  
 ps = this.listener.getBoardPassenger(this.name, this.floor, ps, isFull);  
 this.totalBoardPassengers++;  
 if(isFull) {  
 this.totalBoardPassengers--;  
 }  
 }  
 this.liftMode = LiftMode.*CLOSE*;  
 break;  
  
 case *CLOSE*:  
 this.listener.closeDoor(this.name, this.floor);  
  
 if (isFull) {  
 this.liftMode = LiftMode.*FULL*;  
 } else {  
 this.liftMode = liftDirection;  
 }  
  
 isWaitDown = false;  
 isWaitUp = false;  
 }  
 } catch(Throwable t) {  
 t.printStackTrace();  
 }  
  
 if (isWaitUp && isWaitDown) {  
 this.liftMode = LiftMode.*WAIT*;  
 }  
 }  
  
 //Stop Lift  
 this.listener.stopped(this.name);  
 }  
  
 */\*\*  
 \* Method used to delay the simulation to be viewable by humans.  
 \*/* public static void waitFor() {  
 try {  
 Thread.*sleep*(15);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Method gets the number of moves lift have to perform for passengers to alight.  
 \* This is because the lift is full, so it needs to empty lift.  
 \*  
 \** ***@param*** *mode lift's state  
 \** ***@param*** *floorNo current floor  
 \** ***@return*** *moves  
 \*/* private int getUpDownMove(LiftMode mode, int floorNo) {  
  
 if (mode == LiftMode.*UP*) {  
 for (int i = 0; i < this.departFloors.size(); i++) {  
 int dep = this.departFloors.get(i).intValue();  
 if (dep > floorNo) {  
 return dep - floorNo;  
 }  
 }  
 } else {  
 for (int i = 1; i <= this.departFloors.size(); i++) {  
 int dep = this.departFloors.get(this.departFloors.size() -i).intValue();  
 if (dep < floorNo) {  
 return floorNo - dep;  
 }  
 }  
 }  
 return -1;  
 }  
  
 */\*\*  
 \* Method gets the number of up moves lift have to perform.  
 \*  
 \** ***@param*** *floorNo current floor  
 \** ***@return*** *moves  
 \*/* private int getUpMove(int floorNo) {  
 int arr = 0;  
 for (int i = 0; i < *arriveFloors*.size(); i++) {  
 if (*arriveFloors*.get(i) >= floorNo) {  
 arr = *arriveFloors*.get(i);  
 break;  
 }  
 }  
  
 int dep = 0;  
 for (int i = 0; i < this.departFloors.size(); i++) {  
 if (this.departFloors.get(i) >= floorNo) {  
 dep = this.departFloors.get(i);  
 break;  
 }  
 }  
  
 if (arr == 0 && dep == 0) {  
 return -1;  
 } else if (arr == 0) {  
 return dep - floorNo;  
 } else if (dep == 0) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return arr - floorNo;  
 } else if (dep >= arr) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return arr - floorNo;  
 } else {  
 return dep - floorNo;  
 }  
 }  
  
 */\*\*  
 \* Method gets the number of down moves lift have to perform.  
 \*  
 \** ***@param*** *floorNo current floor  
 \** ***@return*** *moves  
 \*/* private int getDownMove(int floorNo) {  
 int arr = 0;  
 for (int i = *arriveFloors*.size(); i > 0; i--) {  
 if (*arriveFloors*.get(i-1) <= floorNo) {  
 arr = *arriveFloors*.get(i-1);  
 break;  
 }  
 }  
  
 int dep = 0;  
 for (int i = this.departFloors.size(); i > 0; i--) {  
 if (this.departFloors.get(i-1) <= floorNo) {  
 dep = this.departFloors.get(i-1);  
 break;  
 }  
 }  
  
 if (arr == 0 && dep == 0) {  
 return -1;  
 } else if (arr == 0) {  
 return floorNo - dep;  
 } else if (dep == 0) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return floorNo - arr;  
 } else if (dep >= arr) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return floorNo - arr;  
 } else {  
 return floorNo - dep;  
 }  
 }  
}

#### Passenger class

package com.simulation.lift.model;  
  
import java.util.Objects;  
  
public class Passenger {  
 private final int weight;  
 private final int sourceFloor;  
 private final int destinationFloor;  
  
 */\*\*  
 \* Constructs an instance of the object containing  
 \* weight, sourceFloor and destinationFloor arguments.  
 \*  
 \** ***@param*** *weight weight of a passenger  
 \** ***@param*** *sourceFloor source floor of the passenger  
 \** ***@param*** *destinationFloor destination floor of the passenger  
 \*/* public Passenger(int weight, int sourceFloor, int destinationFloor) {  
 this.weight = weight;  
  
 this.sourceFloor = sourceFloor;  
 this.destinationFloor = destinationFloor;  
 }  
  
 */\*\*  
 \* Method get the weight of the passenger.  
 \*  
 \** ***@return*** *passenger's weight  
 \*/* public int getWeight() {  
 return weight;  
 }  
  
 */\*\*  
 \* Method gets the source floor of the passenger.  
 \*  
 \** ***@return*** *passenger's source floor  
 \*/* public int getSourceFloor() {  
 return sourceFloor;  
 }  
  
 */\*\*  
 \* Method gets the destination floor of the passenger.  
 \*  
 \** ***@return*** *passenger's destination floor  
 \*/* public int getDestinationFloor() {  
 return destinationFloor;  
 }  
  
 */\*\*  
 \* Methods checks whether if a given passenger object exists.  
 \*  
 \** ***@param*** *o object parameter  
 \** ***@return*** *isEqual  
 \*/* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Passenger passenger = (Passenger) o;  
 return weight == passenger.weight &&  
 sourceFloor == passenger.sourceFloor &&  
 destinationFloor == passenger.destinationFloor;  
 }  
  
 */\*\*  
 \* Method returns the hash code of a non-null argument and 0 for a null argument.  
 \*  
 \** ***@return*** *hash code  
 \*/* @Override  
 public int hashCode() {  
 return Objects.*hash*(weight, sourceFloor, destinationFloor);  
 }  
  
 */\*\*  
 \* Method combines some of the Passenger's attributes into a String.  
 \*  
 \** ***@return*** *passenger's attributes as String  
 \*/* @Override  
 public String toString() {  
 return "Passenger{" +  
 "weight=" + weight +  
 ", sourceFloor=" + sourceFloor +  
 ", destinationFloor=" + destinationFloor +  
 '}';  
 }  
}

#### Floor Class

package com.simulation.lift.model;  
  
import java.util.Collections;  
import java.util.LinkedList;  
import java.util.List;  
  
public class Floor {  
 private final int floorNo;  
 private final int height;  
 private final int width;  
 private final LinkedList<Passenger> passengers;  
  
 */\*\*  
 \* Constructs an instance of the object containing floorNo,  
 \* height and width arguments.  
 \*  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *height height of a floor  
 \** ***@param*** *width width of a floor  
 \*/* public Floor(int floorNo, int height, int width) {  
 this.floorNo = floorNo;  
 this.height = height;  
 this.width = width;  
 this.passengers = new LinkedList<>();  
 }  
  
 public int getFloorNo() {  
 return floorNo;  
 }  
  
 */\*\*  
 \* Method adding passenger to the floor, who is waiting to aboard a lift.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \*/* public void addNewPassenger(Passenger passenger) {  
 synchronized (this.passengers) {  
 this.passengers.addLast(passenger);  
 }  
 }  
  
 */\*\*  
 \* Method returning passenger to the floor, who is waiting to aboard a lift again.  
 \* This because the passenger was not successful the last time.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \*/* public void addReturnPassenger(Passenger passenger) {  
 synchronized (this.passengers) {  
 this.passengers.addFirst(passenger);  
 }  
 }  
  
 */\*\*  
 \* Method gets the list of Passengers' object from the Floor class.  
 \*  
 \** ***@return*** *list of passengers  
 \*/* public List<Passenger> getPassengers() {  
 return Collections.*unmodifiableList*(this.passengers);  
 }  
  
 */\*\*  
 \* Method removes the first passenger in the queue to board the lift.  
 \*  
 \** ***@return*** *Passenger object  
 \*/* public Passenger removePassenger() {  
 synchronized (this.passengers) {  
 if (!this.passengers.isEmpty()) {  
 return this.passengers.removeFirst();  
 }  
 }  
 return null;  
 }  
}

#### Building Class

package com.simulation.lift.model;  
  
import com.simulation.lift.api.LiftListener;  
  
import java.util.Arrays;  
import java.util.Random;  
  
public class Building implements LiftListener {  
  
 private final int maxFloor;  
 private final int liftWeight;  
 private final int liftHeight;  
 private final int liftWidth;  
 private final int floorHeight;  
 private final int floorWidth;  
  
 private final boolean isAuto;  
 private final boolean isBackEnd;  
  
 private Lift lift;  
 private Floor[] floors;  
  
 */\*\*  
 \* Constructs an instance of the object containing maxFloor, floorHeight,  
 \* floorWidth, maxLift, liftHeight, liftWidth, liftWeight and isAuto arguments.  
 \*  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *floorHeight height of a floor  
 \** ***@param*** *floorWidth width of a floor  
 \** ***@param*** *liftHeight height of a lift  
 \** ***@param*** *liftWidth width of a lift  
 \** ***@param*** *liftWeight maximum lift capacity  
 \** ***@param*** *isAuto is the program auto  
 \*/* public Building(int maxFloor, int floorHeight, int floorWidth, int liftHeight, int liftWidth, int liftWeight, boolean isAuto) {  
 this.maxFloor = maxFloor;  
 this.liftWeight = liftWeight;  
 this.floors = new Floor[maxFloor];  
 this.liftHeight = liftHeight;  
 this.liftWidth = liftWidth;  
 this.floorHeight = floorHeight;  
 this.floorWidth = floorWidth;  
  
 this.isAuto = isAuto;  
 this.isBackEnd = false;  
  
 init();  
 }  
  
 */\*\*  
 \* Constructs an instance of the object containing maxFloor, floorHeight,  
 \* floorWidth, maxLift, liftHeight, liftWidth, liftWeight, passengerNo and isAuto arguments.  
 \*  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *floorHeight height of a floor  
 \** ***@param*** *floorWidth width of a floor  
 \** ***@param*** *liftHeight height of a lift  
 \** ***@param*** *liftWidth width of a lift  
 \** ***@param*** *liftWeight maximum lift capacity  
 \** ***@param*** *passengerNo number of passenger  
 \** ***@param*** *isAuto is the program auto  
 \*/* public Building(int maxFloor, int floorHeight, int floorWidth, int liftHeight, int liftWidth, int liftWeight, int passengerNo, boolean isAuto) {  
 this.maxFloor = maxFloor;  
 this.liftWeight = liftWeight;  
 this.floors = new Floor[maxFloor];  
 this.liftHeight = liftHeight;  
 this.liftWidth = liftWidth;  
 this.floorHeight = floorHeight;  
 this.floorWidth = floorWidth;  
  
 this.isAuto = isAuto;  
 this.isBackEnd = true;  
  
 init();  
  
 setPassengerFloor(passengerNo);  
 }  
  
 */\*\*  
 \* Method starts the project, by initialising the threads.  
 \*/* public void start() {  
 //starting the lifts' threads.  
 this.lift.setListener(this);  
 this.lift.setLiftMode(Lift.LiftMode.*UP*);  
 this.lift.start();  
 }  
  
 */\*\*  
 \* Method gets the list of Lifts' object from the Building class.  
 \*  
 \** ***@return*** *list of lifts  
 \*/* public Lift getLift() {  
 return lift;  
 }  
  
 */\*\*  
 \* Method gets the list of Floors' object from the Building class.  
 \*  
 \** ***@return*** *list of floors  
 \*/* public Floor[] getFloors() {  
 return floors;  
 }  
  
 */\*\*  
 \* Method gets the building's maximum number floors.  
 \*  
 \** ***@return*** *maximum floor number  
 \*/* public int getMaxFloor() {  
 return maxFloor;  
 }  
  
 */\*\*  
 \* Method gets the building's floor height.  
 \*  
 \** ***@return*** *height of a floor  
 \*/* public int getFloorHeight() {  
 return floorHeight;  
 }  
  
 */\*\*  
 \* Method gets the building's lift width.  
 \*  
 \** ***@return*** *width of a lift  
 \*/* public int getLiftWidth() {  
 return liftWidth;  
 }  
  
 */\*\*  
 \* Method calculate the total passengers who have boarded the lifts.  
 \*  
 \** ***@return*** *total boarded passengers  
 \*/* public int calculateTotalBoardPassengers() {  
 return this.lift.getTotalBoardPassengers();  
 }  
  
 */\*\*  
 \* Method calculate the total passengers who have alighted the lifts.  
 \*  
 \** ***@return*** *total alighted passengers  
 \*/* public int calculateTotalAlightPassengers() {  
 return this.lift.getTotalAlightPassengers();  
 }  
  
 */\*\*  
 \* Method calculate the total moves performed by the lifts.  
 \*  
 \** ***@return*** *total lift's moves  
 \*/* private int calculateTotalLiftMoves() {  
 return this.lift.getTotalMoves();  
 }  
  
 */\*\*  
 \* Method calculate the total cost of the program.  
 \*  
 \** ***@return*** *total cost  
 \*/* public int calculateCost() {  
 if(calculateTotalAlightPassengers() == 0) {  
 return 0;  
 }  
 return calculateTotalLiftMoves() / calculateTotalAlightPassengers();  
 }  
  
 */\*\*  
 \* Method sets the passenger's source and destination floor.  
 \*  
 \** ***@param*** *PassengerNo number of passengers  
 \*/* private void setPassengerFloor(int PassengerNo) {  
 Random random = new Random();  
 for (int i = 0; i < PassengerNo; i++) {  
 int sourceFloor;  
 do {  
 sourceFloor = random.nextInt(this.maxFloor) + 1;  
 } while (sourceFloor < 1 || sourceFloor > this.maxFloor);  
  
 int destinationFloor;  
 do {  
 destinationFloor = random.nextInt(this.maxFloor)+ 1;  
 } while(destinationFloor == sourceFloor || destinationFloor < 1 || destinationFloor > this.maxFloor);  
  
 if(this.floors[sourceFloor - 1].getPassengers().size() < PassengerNo) {  
  
 int weight;  
 do {  
 weight = random.nextInt(120) + 50;  
 } while (weight > this.liftWeight);  
  
 this.floors[sourceFloor - 1].addNewPassenger(new Passenger(weight, sourceFloor, destinationFloor));  
  
 System.*out*.println("Source "+sourceFloor);  
  
 this.lift.setArriveFloor(sourceFloor);  
 }  
 }  
 }  
  
 */\*\*  
 \* Method initialise the content of the class in the construct.  
 \*/* private void init() {  
 for (int i = 0; i < this.maxFloor; i++) {  
 this.floors[i] = new Floor(i+1, this.floorHeight, this.floorWidth);  
 }  
 this.lift = new Lift(this.maxFloor, 1, this.liftWeight, this.liftHeight, this.liftWidth, "Lift-"+(1), this.isAuto, this.isBackEnd);  
 }  
  
 */\*\*  
 \* Method moves the lift up by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void upMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: %s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : upMoves : "+s);  
 }  
  
 */\*\*  
 \* Method moves the lift down by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void downMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: $s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : downMoves : "+ s);  
 }  
  
 */\*\*  
 \* Method adds the passengers who are boarding a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *lastPassenger previous passenger  
 \** ***@param*** *isFull is the lift full  
 \** ***@return*** *passenger who could not board  
 \*/* @Override  
 public Passenger getBoardPassenger(String liftName, int floorNo, Passenger lastPassenger, boolean isFull) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, lastPassenger != null ? lastPassenger.toString(): null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : "+s);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : isFull: "+isFull);  
  
 Floor floor = findFloor(floorNo);  
 if(!isFull) {  
 Passenger passenger = floor.removePassenger();  
 return passenger;  
 }  
  
 Lift lift = findLiftByName(liftName);  
 floor.addReturnPassenger(lastPassenger);  
 lift.setArriveFloor(floorNo);  
 return null;  
 }  
  
 */\*\*  
 \* Method removes the passenger who are alighting from a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *passenger alighting passenger  
 \*/* @Override  
 public void alightPassenger(String liftName, int floorNo, Passenger passenger) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, passenger != null ? passenger.toString() : null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : alightPassenger : "+s);  
 }  
  
 */\*\*  
 \* Method to open the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void openDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : openDoor :"+s);  
 }  
  
 */\*\*  
 \* Method to close the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void closeDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : closeDoor :"+s);  
 }  
  
 */\*\*  
 \* Method to display the lift's thread has initialised in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void started(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : started : "+liftName);  
 }  
  
 */\*\*  
 \* Method to display the lift's tread has terminated in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void stopped(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : stopped : "+liftName);  
 }  
  
  
 private Lift findLiftByName(String liftName) {  
 return Arrays.*asList*(this.lift).stream().filter(l -> l.getName().equals(liftName)).findAny().get();  
 }  
  
 private Floor findFloor(int floorNo) {  
 for(Floor floor : this.floors){  
 if(floor.getFloorNo() == floorNo){  
 return floor;  
 }  
 }  
 return null;  
 }  
}

### Panel Package

#### LiftPanel Class

package com.simulation.lift.panel;  
  
import com.simulation.lift.model.Lift;  
  
import javax.swing.\*;  
import java.awt.\*;  
  
public class LiftPanel extends JPanel {  
 private final int x;  
 private int y;  
 private final int width;  
 private final int height;  
 private int doorWidth;  
 private final Color backgroundColor;  
 private final Lift lift;  
  
 private int ty;  
  
 */\*\*  
 \* Constructs an instance of the object containing x, y, width, height,  
 \* backgroundColor and lift arguments.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *height panel's height  
 \** ***@param*** *backgroundColor panel's background colour  
 \** ***@param*** *lift Lift object  
 \*/* public LiftPanel(int x, int y, int width, int height, Color backgroundColor, Lift lift) {  
 this.x = x;  
 this.y = y;  
 this.width = width;  
 this.height = height;  
 this.doorWidth = width / 2;  
 this.backgroundColor = backgroundColor;  
 this.lift = lift;  
  
 setLayout(null);  
 setBounds(x, y, width, height \* lift.getMaxMoveDistance());  
 repaint();  
 }  
  
 */\*\*  
 \* Method gets the Lift object from this class.  
 \*  
 \** ***@return*** *Lift object  
 \*/* public Lift getLift() {  
 return lift;  
 }  
  
 */\*\*  
 \* Method paints the lift moving up.  
 \*  
 \** ***@param*** *move number of moves  
 \*/* public void moveUp(int move) {  
 if (move > 0 ) {  
 int count = 0;  
 int tmp = this.ty;  
 while (count < this.lift.getHeight() \* move) {  
 repaint();  
 this.ty += 5;  
 count += 5;  
 Lift.*waitFor*();  
 }  
 this.ty = tmp + this.lift.getHeight() \* move;  
 }  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the lift moving down.  
 \*  
 \** ***@param*** *move number of moves  
 \*/* public void moveDown(int move) {  
 int count = 0;  
 int tmp = this.ty;  
 while(count < this.lift.getHeight()\*move) {  
 repaint();  
 Lift.*waitFor*();  
 this.ty -= 5;  
 count += 5;  
 }  
 this.ty = tmp - this.lift.getHeight()\*move;  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the door of a lift opening.  
 \*/* public void openDoor() {  
 this.doorWidth = width / 2;  
 while(this.doorWidth > 0) {  
 this.doorWidth-=5;  
 repaint();  
 Lift.*waitFor*();  
 }  
 this.doorWidth = 0;  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the door of a lift closing.  
 \*/* public void closeDoor() {  
 while(this.doorWidth < width / 2) {  
 this.doorWidth+=5;  
 repaint();  
 Lift.*waitFor*();  
 }  
 this.doorWidth = width / 2;  
 repaint();  
 }  
  
 */\*\*  
 \* Method gets the x-axis of the lift.  
 \*  
 \** ***@return*** *x-axis  
 \*/* public int getX() {  
 return x;  
 }  
  
 */\*\*  
 \* Method paints the panel of a LiftPanel object.  
 \*  
 \** ***@param*** *g Graphics object  
 \*/* @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 setBackground(backgroundColor);  
  
 int tw = this.lift.getWidth() - 2;  
 int ty1 = (this.lift.getMaxFloor() - this.lift.getMinFloor()) \* this.lift.getHeight();  
 g.setColor(Color.*RED*);  
 g.drawRect(0, ty1 - this.ty, tw, this.lift.getHeight() - (50 / this.lift.getMaxFloor()));  
 g.setColor(Color.*LIGHT\_GRAY*);  
 g.fillRect(1, ty1-this.ty, this.doorWidth - 1, this.lift.getHeight() - (50 / this.lift.getMaxFloor()));  
 g.fillRect(tw - this.doorWidth + 1, ty1-this.ty, this.doorWidth - 1, this.lift.getHeight() - (50 / this.lift.getMaxFloor()));  
 }  
}

#### FloorPanel Class

package com.simulation.lift.panel;  
  
import com.simulation.lift.model.Floor;  
import com.simulation.lift.model.Lift;  
import com.simulation.lift.model.Passenger;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
public class FloorPanel extends JPanel {  
 */\*\*  
 \* Enum presenting the states of a passenger.  
 \*/* private enum PP {  
 *WAIT*,  
 *BOARD*,  
 *ALIGHT* }  
  
 private final int x;  
 private final int y;  
 private final int width;  
 private final int height;  
 private final Color backgroundColor;  
 private final Floor floor;  
  
 private List<Passenger> passengers;  
 private Passenger movePassenger;  
 private PP pp;  
  
 */\*\*  
 \* Constructs an instance of the object containing x, y, width, height,  
 \* backgroundColor and floor arguments.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *height panel's height  
 \** ***@param*** *backgroundColor panel's background colour  
 \** ***@param*** *floor floor number  
 \*/* public FloorPanel(int x, int y, int width, int height, Color backgroundColor, Floor floor) {  
 this.x = x;  
 this.y = y;  
 this.width = width;  
 this.height = height;  
 this.backgroundColor = backgroundColor;  
 this.floor = floor;  
 this.passengers = new ArrayList<>();  
 this.pp = PP.*WAIT*;  
  
 setLayout(null);  
 setBounds(x, y, width, height);  
 repaint();  
 }  
  
 */\*\*  
 \* Method gets the floor number.  
 \*  
 \** ***@return*** *floor number  
 \*/* public Floor getFloor() {  
 return floor;  
 }  
  
 */\*\*  
 \* Method paints the passengers who are waiting on the floor.  
 \*/* public void flashFloor() {  
 this.passengers = new ArrayList<>(this.floor.getPassengers());  
 this.pp = PP.*WAIT*;  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the passenger who is boarding a lift.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \*/* public void flashBoardFloor(Passenger passenger) {  
 this.movePassenger = passenger;  
 int count = 3;  
 while(count > 0) {  
 this.pp = PP.*BOARD*;  
 repaint();  
 Lift.*waitFor*();  
 count--;  
 }  
 Lift.*waitFor*();  
 flashFloor();  
 }  
  
 */\*\*  
 \* Method paints the passenger who is alighting from a lift.  
 \** ***@param*** *passenger Passenger object  
 \*/* public void flashAlightFloor(Passenger passenger) {  
 this.movePassenger = passenger;  
 int count = 3;  
 while(count > 0) {  
 this.pp = PP.*ALIGHT*;  
 repaint();  
 Lift.*waitFor*();  
 count--;  
 }  
 Lift.*waitFor*();  
 flashFloor();  
 }  
  
 */\*\*  
 \* Method paints moving passengers to/from lifts.  
 \*  
 \** ***@param*** *g Graphics object  
 \** ***@param*** *x x axis move  
 \*/* private void drawMovePassengers(Graphics g, int x) {  
 g.fillRect(x, this.height / 2, 20, this.height / 2);  
 g.setFont(new Font("SansSerif", Font.*PLAIN*, 10));  
 g.drawString("W: " + this.movePassenger.getWeight(), x - 2, this.height / 2 - 18);  
 g.drawString("S: " + (this.movePassenger.getSourceFloor()), x - 2, this.height / 2 - 10);  
 g.drawString("D: " + (this.movePassenger.getDestinationFloor()), x - 2, this.height / 2 - 20);  
 }  
  
 */\*\*  
 \* Method paints the PassengerPanel objects.  
 \*  
 \** ***@param*** *g Graphics object  
 \*/* @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 setBackground(backgroundColor);  
  
 int count = 0;  
 int tx = this.width \* 3 / 4 - 25;  
 while (tx > (this.width / 4) && count < this.passengers.size()) {  
 Passenger p = this.passengers.get(count);  
 g.fillRect(tx, this.height / 2, 20, this.height /2);  
 g.setFont(new Font("SansSerif", Font.*PLAIN*, 10));  
 g.drawString("W: " + p.getWeight(), tx - 2, this.height / 2 - 18);  
 g.drawString("S: " + (p.getSourceFloor()), tx - 2, this.height / 2 - 10);  
 g.drawString("D: " + (p.getDestinationFloor()), tx - 2, this.height / 2 - 2);  
 tx -= 30;  
 count++;  
 }  
  
 if (this.pp != PP.*WAIT*) {  
 if (this.pp == PP.*BOARD*) {  
 tx = this.width \* 3 / 4;  
 while (tx < this.width) {  
 drawMovePassengers(g, tx);  
 tx += 5;  
 }  
 } else if (this.pp == PP.*ALIGHT*) {  
 tx = this.width/4 - 20;  
 while (tx > 0) {  
 drawMovePassengers(g, tx);  
 tx -= 5;  
 }  
 }  
 }  
 }  
  
}

#### BuildingPanel Class

package com.simulation.lift.panel;  
  
import com.simulation.lift.api.LiftListener;  
import com.simulation.lift.model.Building;  
import com.simulation.lift.model.Floor;  
import com.simulation.lift.model.Lift;  
import com.simulation.lift.model.Passenger;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.util.Arrays;  
import java.util.Random;  
  
public class BuildingPanel extends JPanel implements LiftListener {  
 private final int x;  
 private final int y;  
 private final int width;  
 private final int height;  
 private final Color backgroundColor;  
 private final Building building;  
  
 private LiftPanel liftPanel;  
 private final FloorPanel[] floorPanels;  
  
 private Random random;  
 private Timer passengerTimer;  
 private Timer timer;  
 private int refreshCounter;  
 private int timeElapsedInSecs;  
 private final int passengerVolume;  
 private final int maxFloor;  
 private final int liftWeight;  
  
 private final boolean isAuto;  
  
 */\*\*  
 \* Constructs an instance of the object containing x, y, width, height,  
 \* backgroundColor, maxFloor, maxLift, liftWeight and passengerVolume arguments.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *height panel's height  
 \** ***@param*** *backgroundColor panel's background colour  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *liftWeight maximum lift capacity  
 \** ***@param*** *passengerVolume volume of passengers  
 \** ***@param*** *isAuto is the program auto  
 \*/* public BuildingPanel(int x, int y, int width, int height, Color backgroundColor, int maxFloor, int liftWeight, int passengerVolume, boolean isAuto) {  
 this.isAuto = isAuto;  
  
 this.x = x;  
 this.y = y;  
 this.width = width;  
 this.height = height - 20;  
 this.backgroundColor = backgroundColor;  
 this.building = new Building( maxFloor, this.height / maxFloor, width/2, this.height / maxFloor,width / 2, liftWeight, this.isAuto);  
  
 this.floorPanels = new FloorPanel[maxFloor];  
  
 this.passengerVolume = passengerVolume;  
 this.refreshCounter = 0;  
 this.timeElapsedInSecs = 0;  
 this.maxFloor = maxFloor;  
 this.liftWeight = liftWeight;  
  
 this.random = new Random();  
  
 setLayout(null);  
 setBounds(x, y, width, height);  
 repaint();  
 init();  
  
 if(this.isAuto) {  
 this.passengerTimer = new Timer(100, (e) -> {  
 this.refreshCounter += 100;  
 if (this.refreshCounter / 1000 == 1) {  
 this.refreshCounter = 0;  
 this.timeElapsedInSecs++;  
 if (this.timeElapsedInSecs % 5 == 0) {  
 generateNewPassengers();  
 }  
 }  
 });  
 }  
 else {  
 setPassengerFloor(passengerVolume);  
 }  
  
 this.timer = new Timer(100, (e) -> {  
 repaint();  
 });  
 }  
  
 */\*\*  
 \* Method starts the project, by initialising the threads.  
 \*/* public void start() {  
 if(this.isAuto) {  
 //starting the timer's thread.  
 Runnable passengerRunnable = () -> this.passengerTimer.start();  
 Thread passengerThread = new Thread(passengerRunnable);  
 passengerThread.setDaemon(true);  
 passengerThread.setName("PassengerTimer");  
 passengerThread.start();  
 }  
  
 //starting the cost updater thread.  
 Runnable runnable = () -> this.timer.start();  
 Thread thread = new Thread(runnable);  
 thread.setDaemon(true);  
 thread.setName("Timer");  
 thread.start();  
  
 //starting the lifts' threads.  
 Lift lift = this.liftPanel.getLift();  
 runnable = () -> lift.start();  
 thread = new Thread(runnable);  
 thread.setDaemon(true);  
 thread.setName(lift.getName());  
 thread.start();  
 }  
  
 */\*\*  
 \* Method stops the project, by terminate the the threads.  
 \*/* public void stop() {  
 if(this.isAuto) {  
 this.passengerTimer.stop();  
 }  
 this.timer.stop();  
 this.liftPanel.getLift().stop();  
 }  
  
 */\*\*  
 \* Method initialise the content of the class in the construct.  
 \*/* private void init() {  
 int x = 5;  
 int y = 5;  
 int w = this.width/2;  
 int h = this.height / this.maxFloor;  
 createFloors(x, y + 20, w, Color.*ORANGE*);  
 createLift(x + w, y + 20, h, Color.*DARK\_GRAY*);  
 }  
  
 */\*\*  
 \* Method draws the lift panels on top of the building panel.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *height panel's height  
 \** ***@param*** *color panel's background colour  
 \*/* private void createLift(int x, int y, int height, Color color) {  
 int w1 = this.building.getLiftWidth();  
  
 this.liftPanel = new LiftPanel(x, y, w1, height, color, this.building.getLift());  
 this.liftPanel.getLift().setListener(this);  
 add(this.liftPanel);  
 }  
  
 */\*\*  
 \* Method draws the floor panels on top of the building panel.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *color panel's background colour  
 \*/* private void createFloors(int x, int y, int width, Color color) {  
 int h1 = this.building.getFloorHeight();  
  
 for (int i = 1; i <= this.building.getMaxFloor(); i++) {  
 this.floorPanels[this.building.getMaxFloor() - i] = new FloorPanel(x, y, width, h1-(50/this.maxFloor), color, this.building.getFloors()[this.building.getMaxFloor()-i]);  
 add(this.floorPanels[this.building.getMaxFloor()-i]);  
 y += h1;  
 }  
 }  
  
 */\*\*  
 \* Method paints the panel of this class.  
 \*  
 \** ***@param*** *g Graphic object  
 \*/* @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 setBackground(backgroundColor);  
  
 g.setColor(Color.*BLUE*);  
 g.drawRect(5, 2, this.width / 8, 20);  
 g.drawRect(this.width / 8 + 5, 2, this.width / 4, 20);  
 g.drawRect(this.width / 8 + this.width / 4 + 5, 2, this.width / 8, 20);  
  
 g.setColor(Color.*BLACK*);  
 g.drawString("Alight", 40, 17);  
 g.drawString("Waiting Passengers", 150, 17);  
 g.drawString("Board", 355, 17);  
  
 g.setColor(Color.*RED*);  
 g.drawRect(this.width / 2 + 10, 2, 135, 20);  
 g.drawRect(this.width / 2 + 150, 2, 135, 20);  
  
 g.setColor(Color.*BLUE*);  
 g.drawString("Total Board: " + this.building.calculateTotalBoardPassengers(), this.width / 2 + 15, 17);  
 g.drawString("Total Alight: " + this.building.calculateTotalAlightPassengers(), this.width / 2 + 155, 17);  
  
 g.setColor(Color.*BLUE*);  
 g.drawRect(this.width / 2 + 300, 2, 115, 20);  
  
 g.setColor(Color.*red*);  
 g.drawString("Total Cost: " + this.building.calculateCost(), this.width / 2 + 310, 17);  
 }  
  
 */\*\*  
 \* Method moves the lift up by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void upMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: %s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : upMoves : "+s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.moveUp(moveTo);  
 }  
  
 */\*\*  
 \* Method moves the lift down by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void downMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: $s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : downMoves : "+ s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.moveDown(moveTo);  
 }  
  
 */\*\*  
 \* Method adds the passengers who are boarding a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *lastPassenger previous passenger  
 \** ***@param*** *isFull is the lift full  
 \** ***@return*** *passenger who could not board  
 \*/* @Override  
 public Passenger getBoardPassenger(String liftName, int floorNo, Passenger lastPassenger, boolean isFull) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, lastPassenger != null ? lastPassenger.toString(): null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : "+s);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : isFull: "+isFull);  
  
 FloorPanel floorPanel = findFloorPannel(floorNo);  
 if(!isFull) {  
 Passenger passenger = floorPanel.getFloor().removePassenger();  
 if(passenger == null) {  
 return null;  
 }  
 floorPanel.flashBoardFloor(passenger);  
 return passenger;  
 }  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 floorPanel.getFloor().addReturnPassenger(lastPassenger);  
 liftPanel.getLift().setArriveFloor(floorNo);  
 floorPanel.flashFloor();  
 return null;  
 }  
  
 */\*\*  
 \* Method removes the passenger who are alighting from a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *passenger alighting passenger  
 \*/* @Override  
 public void alightPassenger(String liftName, int floorNo, Passenger passenger) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, passenger != null ? passenger.toString() : null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : alightPassenger : "+s);  
  
 FloorPanel floorPanel = findFloorPannel(floorNo);  
 floorPanel.flashAlightFloor(passenger);  
 }  
  
 */\*\*  
 \* Method to open the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void openDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : openDoor :"+s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.openDoor();  
 }  
  
 */\*\*  
 \* Method to close the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void closeDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : closeDoor :"+s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.closeDoor();  
  
 }  
  
 */\*\*  
 \* Method to display the lift's thread has initialised in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void started(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : started : "+liftName);  
 }  
  
 */\*\*  
 \* Method to display the lift's tread has terminated in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void stopped(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : stopped : "+liftName);  
 }  
  
 */\*\*  
 \* Method randomly allocate source and destination floors to the new passengers.  
 \*/* private synchronized void generateNewPassengers() {  
 System.*out*.println("Source aaaaaaaaa");  
 int numOfPassenger = random.nextInt(this.passengerVolume) + 1;  
  
 setPassengerFloor(numOfPassenger);  
 }  
  
 */\*\*  
 \* Method sets the passenger's source and destination floor.  
 \*  
 \** ***@param*** *PassengerNo number of passengers  
 \*/* private void setPassengerFloor(int PassengerNo) {  
 for (int i = 0; i < PassengerNo; i++) {  
 int sourceFloor;  
 do {  
 sourceFloor = this.random.nextInt(this.maxFloor) + 1;  
 } while (sourceFloor < 1 || sourceFloor > this.maxFloor);  
  
 int destinationFloor;  
 do {  
 destinationFloor = this.random.nextInt(this.maxFloor)+ 1;  
 } while(destinationFloor == sourceFloor || destinationFloor < 1 || destinationFloor > this.maxFloor);  
  
 Floor floor = this.floorPanels[sourceFloor-1].getFloor();  
  
 if(floor.getPassengers().size() < this.passengerVolume) {  
  
 int weight;  
 do {  
 weight = this.random.nextInt(120) + 50;  
 } while (weight > this.liftWeight);  
  
 floor.addNewPassenger(new Passenger(weight, sourceFloor, destinationFloor));  
 this.floorPanels[sourceFloor-1].flashFloor();  
  
 System.*out*.println("Source "+sourceFloor);  
  
 this.liftPanel.getLift().setArriveFloor(sourceFloor);  
 }  
 }  
 }  
  
 */\*\*  
 \* Method used to finds a given lift by it's name.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@return*** *LiftPanel object  
 \*/* private LiftPanel findLiftByName(String liftName) {  
 return Arrays.*asList*(this.liftPanel).stream().filter(l -> l.getLift().getName().equals(liftName)).findAny().get();  
 }  
  
 */\*\*  
 \* Method used to find a given floor by a number.  
 \*  
 \** ***@param*** *floorNo floor number  
 \** ***@return*** *FloorPanel object  
 \*/* private FloorPanel findFloorPannel(int floorNo) {  
 return this.floorPanels[floorNo-1];  
 }  
  
 private LiftPanel findLiftPannel(int liftNo) {  
 return this.liftPanel;  
 }  
  
}

#### SimulationPanel Class

package com.simulation.lift.panel;  
  
import javax.swing.\*;  
import java.awt.\*;  
  
public class SimulationPanel extends JPanel {  
 private BuildingPanel building;  
  
 */\*\*  
 \* Constructs an instance of the object containing maxBuildingFloor,  
 \* liftWeight, passengerVolume and liftNo arguments.  
 \*  
 \** ***@param*** *maxBuildingFloor max floor of the building  
 \** ***@param*** *liftWeight lift weight capacity  
 \** ***@param*** *passengerVolume volume of incoming passenger at each refreshing rate  
 \** ***@param*** *isAuto is the program auto  
 \*/* public SimulationPanel(int maxBuildingFloor, int liftWeight, int passengerVolume, Boolean isAuto) {  
 setLayout(null);  
 setPreferredSize(new Dimension(855, 630));  
  
 this.building = new BuildingPanel(0, 0,850, 620, Color.*WHITE*, maxBuildingFloor, liftWeight, passengerVolume, isAuto);  
 //starting all the treads to run in the program.  
 this.building.start();  
  
 add(building);  
 }  
  
 */\*\*  
 \* Method stops all the treads in the BuildPanel class the stop the program.  
 \*/* public void stop() {  
 building.stop();  
 }  
}

### API Package

#### LiftListener Interface

package com.simulation.lift.api;  
  
import com.simulation.lift.model.Passenger;  
  
public interface LiftListener {  
 */\*\*  
 \* Method moves the lift up depending on certain number of moves.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* void upMoves(String liftName, int fromFloorNo, int moveTo);  
  
 */\*\*  
 \* Method moves the lift down depending on certain number of moves.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* void downMoves(String liftName, int fromFloorNo, int moveTo);  
  
 */\*\*  
 \* Method gets the passenger who are boarding a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *lastPassenger previous passenger  
 \** ***@param*** *isFull is the lift full  
 \** ***@return*** *Passenger object  
 \*/* Passenger getBoardPassenger(String liftName, int floorNo, Passenger lastPassenger, boolean isFull);  
  
 */\*\*  
 \* Method removes the passengers who are alighting from a lift.  
 \*  
 \** ***@param*** *liftName name of the life  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *passenger Passenger object  
 \*/* void alightPassenger(String liftName, int floorNo, Passenger passenger);  
  
 */\*\*  
 \* Method opens the door of a lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \** ***@param*** *floorNo floor number  
 \*/* void openDoor(String liftName, int floorNo);  
  
 */\*\*  
 \* Method closes the door of a lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \** ***@param*** *floorNo floor number  
 \*/* void closeDoor(String liftName, int floorNo);  
  
 */\*\*  
 \* Method starts the lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \*/* void started(String liftName);  
  
 */\*\*  
 \* Method stops the lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \*/* void stopped(String liftName);  
}

### Main Package

#### Menu Class

package com.simulation.lift.main;  
  
import com.simulation.lift.panel.SimulationPanel;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.WindowAdapter;  
import java.awt.event.WindowEvent;  
import java.io.\*;  
import java.util.Hashtable;  
  
import static javax.swing.JOptionPane.*showMessageDialog*;  
  
public class Menu extends JPanel {  
  
 */\*\*  
 \* Constructs an instance of the object with no arguments.  
 \*/* public Menu() {  
 initializeMenu();  
 }  
  
 */\*\*  
 \* Method initialize label in the Menu.  
 \*  
 \** ***@param*** *y y-axis of the label  
 \** ***@param*** *width width of the label  
 \** ***@param*** *message message to display in the label  
 \*/* private void initializeLabel(int x, int y, int width, String message) {  
 JPanel panel = new JPanel();  
 panel.setBounds(x, y, width, 20);  
 add(panel);  
 panel.setLayout(null);  
  
 JLabel label = new JLabel(message);  
 label.setBounds(0, 0, width, 20);  
 panel.add(label);  
 }  
  
 */\*\*  
 \* Method initialize user input form in the Menu.  
 \*/* private void initializeInput() {  
 JPanel panel = new JPanel();  
 panel.setBounds(120, -5, 50, 30);  
 add(panel);  
  
 JTextField floorNo = new JTextField(3);  
 panel.add(floorNo);  
  
 JPanel panel1 = new JPanel();  
 panel1.setBounds(250, 33, 100, 30);  
 add(panel1);  
  
 JTextField passengerNo = new JTextField(6);  
 panel1.add(passengerNo);  
  
 JPanel panel2 = new JPanel();  
 panel2.setBounds(160, 60, 190, 60);  
 add(panel2);  
  
 JSlider passengerVolume = new JSlider();  
 passengerVolume.setMajorTickSpacing(50);  
 passengerVolume.setPaintTicks(true);  
 passengerVolume.setPaintTrack(true);  
 passengerVolume.setPaintLabels(true);  
 Hashtable position = new Hashtable();  
 position.put(0, new JLabel("low"));  
 position.put(50, new JLabel("medium"));  
 position.put(100, new JLabel("high"));  
 passengerVolume.setLabelTable(position);  
 passengerVolume.setEnabled(false);  
 panel2.add(passengerVolume);  
  
 JPanel panel3 = new JPanel();  
 panel3.setBounds(110, 120, 80, 30);  
 add(panel3);  
  
 String[] liftWeightList = {"500", "750", "1000", "1250", "1500", "1750", "2000", "2250", "2500", "2750", "3000"};  
 JComboBox liftWeight = new JComboBox(liftWeightList);  
 panel3.add(liftWeight);  
  
 JPanel panel4 = new JPanel();  
 panel4.setBounds(200, 155, 80, 30);  
 add(panel4);  
  
 JPanel panel5 = new JPanel();  
 panel5.setBounds(30, 32, 50, 30);  
 add(panel5);  
  
 JRadioButton isAuto = new JRadioButton();  
 isAuto.addActionListener(e -> {  
 if(isAuto.isSelected()) {  
 passengerVolume.setEnabled(true);  
 passengerNo.setText("");  
 passengerNo.setEditable(false);  
 }  
 else {  
 passengerVolume.setEnabled(false);  
 passengerNo.setEditable(true);  
 }  
 });  
 panel5.add(isAuto);  
  
 JPanel panel6 = new JPanel();  
 panel6.setBounds(345, -3, 50, 30);  
 add(panel6);  
  
 JRadioButton isGraph = new JRadioButton();  
 isGraph.addActionListener(e -> {  
 if(isGraph.isSelected()) {  
 floorNo.setEditable(false);  
 isAuto.setEnabled(false);  
 passengerVolume.setEnabled(false);  
 passengerNo.setText("");  
 passengerNo.setEditable(false);  
 liftWeight.setEnabled(false);  
 }  
 else {  
 floorNo.setEditable(true);  
 isAuto.setEnabled(true);  
 passengerVolume.setEnabled(true);  
 passengerNo.setEditable(true);  
 liftWeight.setEnabled(true);  
 }  
 });  
 panel6.add(isGraph);  
  
  
 JButton submitButton = new JButton("Submit");  
 submitButton.addActionListener(e -> {  
 if (isGraph.isSelected()) {  
 try{  
// String prg = "cost\_dict = {}\n" +  
// "\n" +  
// "def get\_value\_by\_key(obj: dict, key: str):\n" +  
// " try:\n" +  
// " return obj[key]\n" +  
// " except:\n" +  
// " return None\n" +  
// "\n" +  
// "def update\_dict(lift\_capacity, floor\_number, passenger\_number, cost):\n" +  
// "\n" +  
// " lift\_capacity\_obj = get\_value\_by\_key(cost\_dict, lift\_capacity)\n" +  
// " if not lift\_capacity\_obj:\n" +  
// " lift\_capacity\_obj = {lift\_capacity: {}}\n" +  
// " cost\_dict.update(lift\_capacity\_obj)\n" +  
// " lift\_capacity\_obj = cost\_dict[lift\_capacity]\n" +  
// "\n" +  
// " floor\_number\_obj = get\_value\_by\_key(lift\_capacity\_obj, floor\_number)\n" +  
// " if not floor\_number\_obj:\n" +  
// " floor\_number\_obj = {floor\_number: {}}\n" +  
// " lift\_capacity\_obj.update(floor\_number\_obj)\n" +  
// " floor\_number\_obj = lift\_capacity\_obj[floor\_number]\n" +  
// "\n" +  
// " passenger\_number\_obj = get\_value\_by\_key(floor\_number\_obj, passenger\_number)\n" +  
// " if not passenger\_number\_obj:\n" +  
// " passenger\_number\_obj = {passenger\_number: {\"cost\": cost}}\n" +  
// " floor\_number\_obj.update(passenger\_number\_obj)\n";  
//  
// for(int weightCounter = 500; weightCounter <= 3000; weightCounter+=250) {  
// for(int floorCounter = 5; floorCounter <= 300; floorCounter+=5) {  
// for(int passengerCounter = 5; passengerCounter <= 1000; passengerCounter+=5) {  
// Building building = new Building(floorCounter, 620 / floorCounter, 1, 1, 1, weightCounter, passengerCounter, false);  
// building.start();  
// prg = prg + "\n" +  
// "update\_dict(" + weightCounter + ", " + floorCounter + ", " + passengerCounter + ", " + building.calculateCost() + ")\n";  
// }  
// }  
// }  
//  
// prg = prg + "\nprint(cost\_dict)\n";  
//  
// System.out.println(prg);  
//  
// BufferedWriter out = new BufferedWriter(new FileWriter("src/com/simulation/lift/api/CostDict.py"));  
// out.write(prg);  
// out.close();  
  
 Process p = Runtime.*getRuntime*().exec("python3 src/GraphPanel.py");  
 BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));  
 String ret = in.readLine();  
 System.*out*.println("value is : "+ret);  
 }catch(Exception a){  
 a.printStackTrace();  
 }  
 } else if (checkValidity(floorNo, 3, 600)) {  
 if(isAuto.isSelected()) {  
 if (passengerVolume.getValue() >= 0 && passengerVolume.getValue() <= 33) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(floorNo.getText()) / 2, true);  
 }  
 if (passengerVolume.getValue() > 33 && passengerVolume.getValue() < 67) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(floorNo.getText()), true);  
 }  
 if (passengerVolume.getValue() >= 67 && passengerVolume.getValue() <= 100) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(floorNo.getText()) \* 2, true);  
 }  
 }  
 else {  
 if(checkValidity(passengerNo, 1, Integer.*MAX\_VALUE*)) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(passengerNo.getText()),false);  
 }  
 }  
 }  
 });  
 panel4.add(submitButton);  
 }  
  
 */\*\*  
 \* Method initialize the Menu.  
 \*/* private void initializeMenu() {  
 setLayout(null);  
 setPreferredSize(new Dimension(500, 195));  
  
 initializeLabel(5, 5, 120, "Number of Floors:");  
 initializeLabel(5, 40, 35, "Auto:");  
 initializeLabel(150, 40, 120, "Total Passenger:");  
 initializeLabel(5, 80, 150, "Volume of Passengers:");  
 initializeLabel(5, 130, 105, "Max Lift Weight:");  
 initializeLabel(275, 5, 85, "Draw Graph:");  
  
 initializeInput();  
 }  
  
 */\*\*  
 \* Method validate the user input entered in the Menu.  
 \*  
 \** ***@param*** *input user input  
 \** ***@param*** *min minimum accepted value  
 \** ***@param*** *max maximum accepted value  
 \** ***@return*** *is the user input valid  
 \*/* private boolean checkValidity(JTextField input, int min, int max) {  
 try {  
 Integer.*parseInt*(input.getText());  
 } catch (NumberFormatException e) {  
 *showMessageDialog*(null, "The input must be a number.", "Error", 0);  
 return false;  
 }  
 if (Integer.*parseInt*(input.getText()) < min || Integer.*parseInt*(input.getText()) > max) {  
 *showMessageDialog*(null, "The input must be between " + min + " and " + max + ".", "Error", 0);  
 return false;  
 }  
 return true;  
 }  
  
 */\*\*  
 \* Method to start the simulation.  
 \*  
 \** ***@param*** *maxBuildingFloor max floor of the building  
 \** ***@param*** *liftWeight lift weight capacity  
 \** ***@param*** *passengerVolume volume of incoming passenger at each refreshing rate  
 \** ***@param*** *isAuto is the program auto  
 \*/* private void startLiftSimulation(int maxBuildingFloor, int liftWeight, int passengerVolume, boolean isAuto) {  
 JFrame jf = new JFrame();  
 SimulationPanel simulationPanel = new SimulationPanel(maxBuildingFloor, liftWeight, passengerVolume, isAuto);  
 jf.add(simulationPanel);  
 jf.setTitle("Lift Simulator");  
 jf.pack();  
 jf.setLocation(250,75);  
 jf.setVisible(true);  
// jf.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  
 jf.addWindowListener(new WindowAdapter() {  
 @Override  
 public void windowClosing(WindowEvent e) {  
 //stops the thread when the frame is closed.  
 simulationPanel.stop();  
 System.*out*.println("Closed");  
 e.getWindow().dispose();  
 }  
 });  
 jf.setResizable(false);  
 }  
}

#### Main Class

package com.simulation.lift.main;  
  
import javax.swing.\*;  
  
public class Main {  
  
 public static void main(String[] args) {  
 JFrame jf = new JFrame();  
 jf.add(new Menu());  
 jf.setTitle("Main Menu");  
 jf.pack();  
 jf.setLocation(400,100);  
 jf.setVisible(true);  
 jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 jf.setResizable(false);  
 }  
}

### Graph Python File

#### GraphPanel Module

import matplotlib.pyplot as plt  
from com.simulation.lift.api.CostDict import cost\_dict  
  
fig = plt.figure()  
fig.canvas.set\_window\_title('Base Case Graph')  
fig.suptitle("""Cost  
in function of   
Lift Capacity, Floor Number and Passenger Number""")  
  
ax = fig.add\_subplot(111, projection='3d')  
  
x = []  
y = []  
z = []  
c = []  
  
for lift\_capacity in cost\_dict:  
 for floor\_number in cost\_dict[lift\_capacity]:  
 for passenger\_number in cost\_dict[lift\_capacity][floor\_number]:  
 x.append(lift\_capacity)  
 y.append(floor\_number)  
 z.append(passenger\_number)  
 c.append(cost\_dict[lift\_capacity][floor\_number][passenger\_number]["cost"])  
  
img = ax.scatter(x, y, z, c=c, cmap=plt.hot())  
cbar = fig.colorbar(img)  
cbar.ax.get\_yaxis().labelpad = 15; cbar.ax.set\_ylabel("Cost", rotation = 270);  
ax.set\_xlabel('Lift Capacity')  
ax.set\_ylabel('Floor Number')  
ax.set\_zlabel('Passenger Number')  
  
plt.show()

## Improved Lift System (Advance Case)

### Model Package

#### Lift Class

package com.simulation.lift.model;  
  
import com.simulation.lift.api.LiftListener;  
  
import java.util.\*;  
import java.util.stream.Collectors;  
  
public class Lift {  
 */\*\*  
 \* Enum presenting the states of a lift.  
 \*/* public enum LiftMode {  
 *UP*,  
 *DOWN*,  
 *OPEN*,  
 *CLOSE*,  
 *WAIT*,  
 *BOARED*,  
 *ALIGHT*,  
 *FULL* }  
  
 private int floor;  
  
 private final int maxFloor;  
 private final int minFloor;  
 private int maxWeight;  
 private final int height;  
 private final int width;  
 private final String name;  
  
 private int totalBoardPassengers;  
 private int totalAlightPassengers;  
 private int totalMoves;  
  
 private LiftMode liftMode;  
 private LiftMode liftDirection;  
  
 private final List<Passenger> passengers;  
 private final static List<Integer> *arriveFloors* = Collections.*synchronizedList*(new ArrayList<>());  
 private final List<Integer> departFloors;  
  
 private LiftListener listener;  
  
 private boolean isRunning;  
  
 private Object object = new Object();  
  
 private final boolean isAuto;  
 private final boolean isBackEnd;  
  
 */\*\*  
 \* Constructs an instance of the object containing maxFloor, minFloor,  
 \* maxWeight, height, width and name arguments.  
 \*  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *minFloor minimum floor number  
 \** ***@param*** *maxWeight maximum capacity  
 \** ***@param*** *height height of a lift  
 \** ***@param*** *width width of a lift  
 \** ***@param*** *name name of the lift  
 \** ***@param*** *isAuto is the program auto  
 \** ***@param*** *isBackEnd is the program only for back-end  
 \*/* public Lift(int maxFloor, int minFloor, int maxWeight, int height, int width, String name, boolean isAuto, boolean isBackEnd) {  
 this.floor = minFloor;  
  
 this.maxFloor = maxFloor;  
 this.minFloor = minFloor;  
 this.maxWeight = maxWeight;  
 this.height = height;  
 this.width = width;  
  
 this.name = name;  
  
 this.isAuto = isAuto;  
 this.isBackEnd = isBackEnd;  
  
 this.liftMode = LiftMode.*WAIT*;  
 this.liftDirection = LiftMode.*WAIT*;  
  
 if(!this.isAuto) {  
 this.liftMode = LiftMode.*UP*;  
 }  
  
 this.passengers = new ArrayList<>();  
 this.departFloors = new ArrayList<>();  
 }  
  
 */\*\*  
 \* Method sets a listener object to the Lift class.  
 \*  
 \** ***@param*** *listener listener object  
 \*/* public void setListener(LiftListener listener) {  
 this.listener = listener;  
 }  
  
 public void setLiftMode(LiftMode mode) {  
 this.liftMode = mode;  
 }  
  
 */\*\*  
 \* Method gets the height of a lift.  
 \*  
 \** ***@return*** *lift's height  
 \*/* public int getHeight() {  
 return height;  
 }  
  
 */\*\*  
 \* Method gets the width of a lift.  
 \*  
 \** ***@return*** *lift's width  
 \*/* public int getWidth() {  
 return width;  
 }  
  
 */\*\*  
 \* Method gets the maximum floor the lift can travel to.  
 \*  
 \** ***@return*** *lift's maximum floor  
 \*/* public int getMaxFloor() {  
 return maxFloor;  
 }  
  
 */\*\*  
 \* Method gets the minimum floor the lift can travel to.  
 \*  
 \** ***@return*** *lift's minimum floor  
 \*/* public int getMinFloor() {  
 return minFloor;  
 }  
  
 */\*\*  
 \* Method gets the name of the lift.  
 \*  
 \** ***@return*** *lift's name  
 \*/* public String getName() {  
 return this.name;  
 }  
  
 */\*\*  
 \* Method gets the total number of passengers, who have boarded the lift.  
 \*  
 \** ***@return*** *total boarded passengers  
 \*/* public int getTotalBoardPassengers() {  
 return totalBoardPassengers;  
 }  
  
 */\*\*  
 \* Method gets the total number of passengers, who have alighted from the lift.  
 \*  
 \** ***@return*** *total alighted passengers  
 \*/* public int getTotalAlightPassengers() {  
 return totalAlightPassengers;  
 }  
  
 */\*\*  
 \* Method gets the total moves completed by the lift.  
 \*  
 \** ***@return*** *total moves  
 \*/* public int getTotalMoves() {  
 return totalMoves;  
 }  
  
 */\*\*  
 \* Method to start the lift simulation in the back-end.  
 \*/* public void start() {  
 this.isRunning = true;  
 liftOperation();  
 }  
  
 */\*\*  
 \* Method to stop the lift simulation in the back-end.  
 \*/* public void stop() {  
 this.isRunning = false;  
 synchronized (object) {  
 object.notifyAll();  
 }  
 }  
  
 public int getFloor() {  
 return floor;  
 }  
  
 */\*\*  
 \* Method gets the maximum distance between the maxFloor and minFloor.  
 \*  
 \** ***@return*** *maximum distance  
 \*/* public int getMaxMoveDistance() {  
 return this.maxFloor - (this.minFloor - 1);  
 }  
  
 */\*\*  
 \* Method boards passengers to the lift.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \** ***@return*** *whether or not the passenger boarded  
 \*/* private boolean boardPassenger(Passenger passenger){  
 if (this.maxWeight - passenger.getWeight() < 0 ) {  
 return true;  
 }  
  
 if (this.passengers.add(passenger)) {  
 this.maxWeight -= passenger.getWeight();  
 if (!this.departFloors.contains(passenger.getDestinationFloor())) {  
 this.departFloors.add(passenger.getDestinationFloor());  
 this.departFloors.sort(Comparator.*naturalOrder*());  
 }  
 }  
 return false;  
 }  
  
 */\*\*  
 \* Method returns a list of alighting passenger from the lift.  
 \*  
 \** ***@return*** *list of passengers  
 \*/* private List<Passenger> alight() {  
  
 List<Passenger> alightPassenger = this.passengers.stream()  
 .filter(p->this.floor == p.getDestinationFloor())  
 .map(p -> {  
 this.maxWeight += p.getWeight();  
 return p;  
 })  
 .collect(Collectors.*toList*());  
  
 this.passengers.removeAll(alightPassenger);  
  
 return alightPassenger;  
 }  
  
 */\*\*  
 \* Method sets the arrival floors of passengers.  
 \*  
 \** ***@param*** *floorNO floor number  
 \*/* public void setArriveFloor(int floorNO) {  
 synchronized (*arriveFloors*) {  
 if (!*arriveFloors*.contains(floorNO)) {  
 *arriveFloors*.add(Integer.*valueOf*(floorNO));  
 *arriveFloors*.sort(Comparator.*naturalOrder*());  
 }  
 }  
  
 synchronized (object) {  
 object.notifyAll();  
 }  
 }  
  
 */\*\*  
 \* Method controls the states of the lift.  
 \*/* private void liftOperation() {  
 //Start Lift  
 this.listener.started(this.name);  
  
 boolean isWaitUp = false;  
 boolean isWaitDown = false;  
 boolean isFull = false;  
  
 while(isRunning) {  
 System.*out*.println("Thread "+this.name+ " LiftMode: "+this.liftMode.name());  
  
 try {  
 switch (this.liftMode) {  
 case *FULL*:  
 int move = getUpDownMove(liftDirection, this.floor);  
 if(move == -1){  
 liftDirection = liftDirection == LiftMode.*UP* ? LiftMode.*DOWN* : LiftMode.*UP*;  
 }  
 else if (liftDirection == LiftMode.*UP*) {  
 isFull = false;  
 this.listener.upMoves(this.name, this.floor, move);  
  
 this.floor += move;  
 *arriveFloors*.remove(Integer.*valueOf*(this.floor));  
 this.totalMoves += move;  
 this.liftMode = LiftMode.*OPEN*;  
 } else {  
 isFull = false;  
 this.listener.downMoves(this.name, this.floor, move);  
  
 this.floor -= move;  
 *arriveFloors*.remove(Integer.*valueOf*(this.floor));  
 this.totalMoves += move;  
 this.liftMode = LiftMode.*OPEN*;  
 }  
 break;  
  
 case *WAIT*:  
 synchronized (object) {  
 try {  
 object.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 this.liftMode = LiftMode.*UP*;  
 isWaitDown = false;  
 isWaitUp = false;  
 break;  
  
 case *UP*:  
 if (this.floor == this.maxFloor) {  
 this.liftMode = LiftMode.*DOWN*;  
 isWaitUp = true;  
  
 } else {  
 move = getUpMove(this.floor);  
 if (move == -1) {  
 if(this.passengers.isEmpty() && totalBoardPassengers == totalAlightPassengers && *arriveFloors*.isEmpty() && departFloors.isEmpty()) {  
 if(!this.isAuto) {  
 stop();  
 break;  
 }  
 this.liftMode = LiftMode.*WAIT*;  
 }  
 else {  
 this.liftMode = LiftMode.*DOWN*;  
 isWaitUp = true;  
 }  
 } else {  
 this.listener.upMoves(this.name, this.floor, move);  
  
 this.floor += move;  
 this.totalMoves += move;  
 liftDirection = LiftMode.*UP*;  
 this.liftMode = LiftMode.*OPEN*;  
 }  
 }  
 break;  
  
 case *DOWN*:  
 if (this.floor == this.minFloor) {  
 this.liftMode = LiftMode.*UP*;  
 isWaitDown = true;  
  
 } else {  
 move = getDownMove(this.floor);  
 if (move == -1) {  
 if(this.passengers.isEmpty() && totalBoardPassengers == totalAlightPassengers && *arriveFloors*.isEmpty()&& departFloors.isEmpty()) {  
 if(!this.isAuto) {  
 stop();  
 break;  
 }  
 this.liftMode = LiftMode.*WAIT*;  
 }  
 else {  
 this.liftMode = LiftMode.*UP*;  
 isWaitDown = true;  
 }  
 } else {  
 this.listener.downMoves(this.name, this.floor, move);  
  
 this.floor -= move;  
 this.totalMoves += move;  
 liftDirection = LiftMode.*DOWN*;  
 this.liftMode = LiftMode.*OPEN*;  
 }  
 }  
 break;  
  
 case *OPEN*:  
 isWaitDown = false;  
 isWaitUp = false;  
  
 this.listener.openDoor(this.name, this.floor);  
  
 this.liftMode = LiftMode.*ALIGHT*;  
 break;  
  
 case *ALIGHT*:  
 //Get off passenger  
 for(Passenger passenger : alight()) {  
 this.totalAlightPassengers++;  
 this.listener.alightPassenger(this.name, this.floor, passenger);  
 if(!isBackEnd) {  
 *waitFor*();  
 }  
 }  
  
 departFloors.remove(Integer.*valueOf*(this.floor));  
 this.liftMode = LiftMode.*BOARED*;  
 break;  
  
 case *BOARED*:  
 //Get in passenger  
 Passenger ps = this.listener.getBoardPassenger(this.name, this.floor, null, false);  
 while(ps != null) {  
 isFull = boardPassenger(ps);  
 ps = this.listener.getBoardPassenger(this.name, this.floor, ps, isFull);  
 this.totalBoardPassengers++;  
 if(isFull) {  
 this.totalBoardPassengers--;  
 }  
 }  
 this.liftMode = LiftMode.*CLOSE*;  
 break;  
  
 case *CLOSE*:  
 this.listener.closeDoor(this.name, this.floor);  
  
 if (isFull) {  
 this.liftMode = LiftMode.*FULL*;  
 } else {  
 this.liftMode = liftDirection;  
 }  
  
 isWaitDown = false;  
 isWaitUp = false;  
 }  
 } catch(Throwable t) {  
 t.printStackTrace();  
 }  
  
 if (isWaitUp && isWaitDown) {  
 this.liftMode = LiftMode.*WAIT*;  
 }  
 }  
  
 //Stop Lift  
 this.listener.stopped(this.name);  
 }  
  
 */\*\*  
 \* Method used to delay the simulation to be viewable by humans.  
 \*/* public static void waitFor() {  
 try {  
 Thread.*sleep*(15);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Method gets the number of moves lift have to perform for passengers to alight.  
 \* This is because the lift is full, so it needs to empty lift.  
 \*  
 \** ***@param*** *mode lift's state  
 \** ***@param*** *floorNo current floor  
 \** ***@return*** *moves  
 \*/* private int getUpDownMove(LiftMode mode, int floorNo) {  
  
 if (mode == LiftMode.*UP*) {  
 for (int i = 0; i < this.departFloors.size(); i++) {  
 int dep = this.departFloors.get(i).intValue();  
 if (dep > floorNo) {  
 return dep - floorNo;  
 }  
 }  
 } else {  
 for (int i = 1; i <= this.departFloors.size(); i++) {  
 int dep = this.departFloors.get(this.departFloors.size() -i).intValue();  
 if (dep < floorNo) {  
 return floorNo - dep;  
 }  
 }  
 }  
 return -1;  
 }  
  
 */\*\*  
 \* Method gets the number of up moves lift have to perform.  
 \*  
 \** ***@param*** *floorNo current floor  
 \** ***@return*** *moves  
 \*/* private int getUpMove(int floorNo) {  
 int arr = 0;  
 for (int i = 0; i < *arriveFloors*.size(); i++) {  
 if (*arriveFloors*.get(i) >= floorNo) {  
 arr = *arriveFloors*.get(i);  
 break;  
 }  
 }  
  
 int dep = 0;  
 for (int i = 0; i < this.departFloors.size(); i++) {  
 if (this.departFloors.get(i) >= floorNo) {  
 dep = this.departFloors.get(i);  
 break;  
 }  
 }  
  
 if (arr == 0 && dep == 0) {  
 return -1;  
 } else if (arr == 0) {  
 return dep - floorNo;  
 } else if (dep == 0) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return arr - floorNo;  
 } else if (dep >= arr) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return arr - floorNo;  
 } else {  
 return dep - floorNo;  
 }  
 }  
  
 */\*\*  
 \* Method gets the number of down moves lift have to perform.  
 \*  
 \** ***@param*** *floorNo current floor  
 \** ***@return*** *moves  
 \*/* private int getDownMove(int floorNo) {  
 int arr = 0;  
 for (int i = *arriveFloors*.size(); i > 0; i--) {  
 if (*arriveFloors*.get(i-1) <= floorNo) {  
 arr = *arriveFloors*.get(i-1);  
 break;  
 }  
 }  
  
 int dep = 0;  
 for (int i = this.departFloors.size(); i > 0; i--) {  
 if (this.departFloors.get(i-1) <= floorNo) {  
 dep = this.departFloors.get(i-1);  
 break;  
 }  
 }  
  
 if (arr == 0 && dep == 0) {  
 return -1;  
 } else if (arr == 0) {  
 return floorNo - dep;  
 } else if (dep == 0) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return floorNo - arr;  
 } else if (dep >= arr) {  
 *arriveFloors*.remove(Integer.*valueOf*(arr));  
 return floorNo - arr;  
 } else {  
 return floorNo - dep;  
 }  
 }  
}

#### Passenger Class

package com.simulation.lift.model;  
  
import java.util.Objects;  
  
public class Passenger {  
 private final int weight;  
 private final int sourceFloor;  
 private final int destinationFloor;  
  
 */\*\*  
 \* Constructs an instance of the object containing  
 \* weight, sourceFloor and destinationFloor arguments.  
 \*  
 \** ***@param*** *weight weight of a passenger  
 \** ***@param*** *sourceFloor source floor of the passenger  
 \** ***@param*** *destinationFloor destination floor of the passenger  
 \*/* public Passenger(int weight, int sourceFloor, int destinationFloor) {  
 this.weight = weight;  
  
 this.sourceFloor = sourceFloor;  
 this.destinationFloor = destinationFloor;  
 }  
  
 */\*\*  
 \* Method get the weight of the passenger.  
 \*  
 \** ***@return*** *passenger's weight  
 \*/* public int getWeight() {  
 return weight;  
 }  
  
 */\*\*  
 \* Method gets the source floor of the passenger.  
 \*  
 \** ***@return*** *passenger's source floor  
 \*/* public int getSourceFloor() {  
 return sourceFloor;  
 }  
  
 */\*\*  
 \* Method gets the destination floor of the passenger.  
 \*  
 \** ***@return*** *passenger's destination floor  
 \*/* public int getDestinationFloor() {  
 return destinationFloor;  
 }  
  
 */\*\*  
 \* Methods checks whether if a given passenger object exists.  
 \*  
 \** ***@param*** *o object parameter  
 \** ***@return*** *isEqual  
 \*/* @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
 Passenger passenger = (Passenger) o;  
 return weight == passenger.weight &&  
 sourceFloor == passenger.sourceFloor &&  
 destinationFloor == passenger.destinationFloor;  
 }  
  
 */\*\*  
 \* Method returns the hash code of a non-null argument and 0 for a null argument.  
 \*  
 \** ***@return*** *hash code  
 \*/* @Override  
 public int hashCode() {  
 return Objects.*hash*(weight, sourceFloor, destinationFloor);  
 }  
  
 */\*\*  
 \* Method combines some of the Passenger's attributes into a String.  
 \*  
 \** ***@return*** *passenger's attributes as String  
 \*/* @Override  
 public String toString() {  
 return "Passenger{" +  
 "weight=" + weight +  
 ", sourceFloor=" + sourceFloor +  
 ", destinationFloor=" + destinationFloor +  
 '}';  
 }  
}

#### Floor Class

package com.simulation.lift.model;  
  
import java.util.Collections;  
import java.util.LinkedList;  
import java.util.List;  
  
public class Floor {  
 private final int floorNo;  
 private final int height;  
 private final int width;  
 private final LinkedList<Passenger> passengers;  
  
 */\*\*  
 \* Constructs an instance of the object containing floorNo,  
 \* height and width arguments.  
 \*  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *height height of a floor  
 \** ***@param*** *width width of a floor  
 \*/* public Floor(int floorNo, int height, int width) {  
 this.floorNo = floorNo;  
 this.height = height;  
 this.width = width;  
 this.passengers = new LinkedList<>();  
 }  
  
 public int getFloorNo() {  
 return floorNo;  
 }  
  
 */\*\*  
 \* Method adding passenger to the floor, who is waiting to aboard a lift.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \*/* public void addNewPassenger(Passenger passenger) {  
 synchronized (this.passengers) {  
 this.passengers.addLast(passenger);  
 }  
 }  
  
 */\*\*  
 \* Method returning passenger to the floor, who is waiting to aboard a lift again.  
 \* This because the passenger was not successful the last time.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \*/* public void addReturnPassenger(Passenger passenger) {  
 synchronized (this.passengers) {  
 this.passengers.addFirst(passenger);  
 }  
 }  
  
 */\*\*  
 \* Method gets the list of Passengers' object from the Floor class.  
 \*  
 \** ***@return*** *list of passengers  
 \*/* public List<Passenger> getPassengers() {  
 return Collections.*unmodifiableList*(this.passengers);  
 }  
  
 */\*\*  
 \* Method removes the first passenger in the queue to board the lift.  
 \*  
 \** ***@return*** *Passenger object  
 \*/* public Passenger removePassenger() {  
 synchronized (this.passengers) {  
 if (!this.passengers.isEmpty()) {  
 return this.passengers.removeFirst();  
 }  
 }  
 return null;  
 }  
}

#### Building Class

package com.simulation.lift.model;  
  
import com.simulation.lift.api.LiftListener;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.Random;  
import java.util.concurrent.CompletableFuture;  
  
public class Building implements LiftListener {  
  
 private final int maxFloor;  
 private final int maxLift;  
 private final int liftWeight;  
 private final int liftHeight;  
 private final int liftWidth;  
 private final int floorHeight;  
 private final int floorWidth;  
  
 private final boolean isAuto;  
 private final boolean isBackEnd;  
  
 private Lift[] lifts;  
 private Floor[] floors;  
  
 */\*\*  
 \* Constructs an instance of the object containing maxFloor, floorHeight,  
 \* floorWidth, maxLift, liftHeight, liftWidth, liftWeight and isAuto arguments.  
 \*  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *floorHeight height of a floor  
 \** ***@param*** *floorWidth width of a floor  
 \** ***@param*** *liftHeight height of a lift  
 \** ***@param*** *liftWidth width of a lift  
 \** ***@param*** *liftWeight maximum lift capacity  
 \** ***@param*** *isAuto is the program auto  
 \*/* public Building(int maxFloor, int floorHeight, int floorWidth, int maxLift, int liftHeight, int liftWidth, int liftWeight, boolean isAuto) {  
 this.maxFloor = maxFloor;  
 this.maxLift = maxLift;  
 this.liftWeight = liftWeight;  
 this.lifts = new Lift[maxLift];  
 this.floors = new Floor[maxFloor];  
 this.liftHeight = liftHeight;  
 this.liftWidth = liftWidth;  
 this.floorHeight = floorHeight;  
 this.floorWidth = floorWidth;  
  
 this.isAuto = isAuto;  
 this.isBackEnd = false;  
  
 init();  
 }  
  
 */\*\*  
 \* Constructs an instance of the object containing maxFloor, floorHeight,  
 \* floorWidth, maxLift, liftHeight, liftWidth, liftWeight, passengerNo and isAuto arguments.  
 \*  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *floorHeight height of a floor  
 \** ***@param*** *floorWidth width of a floor  
 \** ***@param*** *liftHeight height of a lift  
 \** ***@param*** *liftWidth width of a lift  
 \** ***@param*** *liftWeight maximum lift capacity  
 \** ***@param*** *passengerNo number of passenger  
 \** ***@param*** *isAuto is the program auto  
 \*/* public Building(int maxFloor, int floorHeight, int floorWidth, int maxLift, int liftHeight, int liftWidth, int liftWeight, int passengerNo, boolean isAuto) {  
 this.maxFloor = maxFloor;  
 this.maxLift = maxLift;  
 this.liftWeight = liftWeight;  
 this.lifts = new Lift[maxLift];  
 this.floors = new Floor[maxFloor];  
 this.liftHeight = liftHeight;  
 this.liftWidth = liftWidth;  
 this.floorHeight = floorHeight;  
 this.floorWidth = floorWidth;  
  
 this.isAuto = isAuto;  
 this.isBackEnd = true;  
  
 init();  
  
 setPassengerFloor(passengerNo);  
 }  
  
 */\*\*  
 \* Method starts the project, by initialising the threads.  
 \** ***@return*** *\*/* public CompletableFuture<Void> start() {  
 //starting the lifts' threads.  
 List<CompletableFuture<Void>> completableFutures = new ArrayList<>();  
 for (Lift lift : this.lifts) {  
 lift.setListener(this);  
 lift.setLiftMode(Lift.LiftMode.*UP*);  
  
 completableFutures.add(CompletableFuture.*runAsync*(lift::start));  
 }  
  
 return CompletableFuture.*allOf*(completableFutures.toArray(new CompletableFuture[completableFutures.size()]));  
 }  
  
 */\*\*  
 \* Method gets the list of Lifts' object from the Building class.  
 \*  
 \** ***@return*** *list of lifts  
 \*/* public Lift[] getLifts() {  
 return lifts;  
 }  
  
 */\*\*  
 \* Method gets the list of Floors' object from the Building class.  
 \*  
 \** ***@return*** *list of floors  
 \*/* public Floor[] getFloors() {  
 return floors;  
 }  
  
 */\*\*  
 \* Method gets the building's maximum number floors.  
 \*  
 \** ***@return*** *maximum floor number  
 \*/* public int getMaxFloor() {  
 return maxFloor;  
 }  
  
 */\*\*  
 \* Method gets the building's maximum number of lifts.  
 \*  
 \** ***@return*** *maximum lift number  
 \*/* public int getMaxLift() {  
 return maxLift;  
 }  
  
 */\*\*  
 \* Method gets the building's floor height.  
 \*  
 \** ***@return*** *height of a floor  
 \*/* public int getFloorHeight() {  
 return floorHeight;  
 }  
  
 */\*\*  
 \* Method gets the building's lift width.  
 \*  
 \** ***@return*** *width of a lift  
 \*/* public int getLiftWidth() {  
 return liftWidth;  
 }  
  
 */\*\*  
 \* Method calculate the total passengers who have boarded the lifts.  
 \*  
 \** ***@return*** *total boarded passengers  
 \*/* public int calculateTotalBoardPassengers() {  
 int totalPassengers = 0;  
 for(Lift lift : this.lifts) {  
 totalPassengers += lift.getTotalBoardPassengers();  
 }  
 return totalPassengers;  
 }  
  
 */\*\*  
 \* Method calculate the total passengers who have alighted the lifts.  
 \*  
 \** ***@return*** *total alighted passengers  
 \*/* public int calculateTotalAlightPassengers() {  
 int totalPassengers = 0;  
 for(Lift lift : this.lifts) {  
 totalPassengers += lift.getTotalAlightPassengers();  
 }  
 return totalPassengers;  
 }  
  
 */\*\*  
 \* Method calculate the total moves performed by the lifts.  
 \*  
 \** ***@return*** *total lift's moves  
 \*/* private int calculateTotalLiftMoves() {  
 int totalMoves = 0;  
 for(Lift lift : this.lifts) {  
 totalMoves += lift.getTotalMoves();  
 }  
 return totalMoves;  
 }  
  
 */\*\*  
 \* Method calculate the total cost of the program.  
 \*  
 \** ***@return*** *total cost  
 \*/* public double calculateCost() {  
 if(calculateTotalAlightPassengers() == 0) {  
 return 0;  
 }  
 return calculateTotalLiftMoves() / (calculateTotalAlightPassengers() \* this.maxLift);  
 }  
  
 */\*\*  
 \* Method sets the passenger's source and destination floor.  
 \*  
 \** ***@param*** *PassengerNo number of passengers  
 \*/* private void setPassengerFloor(int PassengerNo) {  
 Random random = new Random();  
 for (int i = 0; i < PassengerNo; i++) {  
 int sourceFloor;  
 do {  
 sourceFloor = random.nextInt(this.maxFloor) + 1;  
 } while (sourceFloor < 1 || sourceFloor > this.maxFloor);  
  
 int liftNo;  
 do {  
 liftNo = random.nextInt(this.maxLift) + 1;  
 } while (liftNo < 1 || liftNo > this.maxLift);  
  
 int destinationFloor;  
 do {  
 destinationFloor = random.nextInt(this.maxFloor)+ 1;  
 } while(destinationFloor == sourceFloor || destinationFloor < 1 || destinationFloor > this.maxFloor);  
  
 if(this.floors[sourceFloor - 1].getPassengers().size() < PassengerNo) {  
  
 int weight;  
 do {  
 weight = random.nextInt(120) + 50;  
 } while (weight > this.liftWeight);  
  
 this.floors[sourceFloor - 1].addNewPassenger(new Passenger(weight, sourceFloor, destinationFloor));  
  
 System.*out*.println("Source "+sourceFloor);  
  
 this.lifts[liftNo-1].setArriveFloor(sourceFloor);  
 }  
 }  
 }  
  
 */\*\*  
 \* Method initialise the content of the class in the construct.  
 \*/* private void init() {  
 for (int i = 0; i < this.maxFloor; i++) {  
 this.floors[i] = new Floor(i+1, this.floorHeight, this.floorWidth);  
 }  
  
 for (int i = 0; i < this.maxLift; i++) {  
 this.lifts[i] = new Lift(this.maxFloor, 1, this.liftWeight, this.liftHeight, this.liftWidth, "Lift-"+(i+1), this.isAuto, this.isBackEnd);  
 }  
 }  
  
 */\*\*  
 \* Method moves the lift up by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void upMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: %s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : upMoves : "+s);  
 }  
  
 */\*\*  
 \* Method moves the lift down by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void downMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: $s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : downMoves : "+ s);  
 }  
  
 */\*\*  
 \* Method adds the passengers who are boarding a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *lastPassenger previous passenger  
 \** ***@param*** *isFull is the lift full  
 \** ***@return*** *passenger who could not board  
 \*/* @Override  
 public Passenger getBoardPassenger(String liftName, int floorNo, Passenger lastPassenger, boolean isFull) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, lastPassenger != null ? lastPassenger.toString(): null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : "+s);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : isFull: "+isFull);  
  
 Floor floor = findFloor(floorNo);  
 if(!isFull) {  
 Passenger passenger = floor.removePassenger();  
 return passenger;  
 }  
  
 Lift lift = findLiftByName(liftName);  
 floor.addReturnPassenger(lastPassenger);  
 lift.setArriveFloor(floorNo);  
 return null;  
 }  
  
 */\*\*  
 \* Method removes the passenger who are alighting from a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *passenger alighting passenger  
 \*/* @Override  
 public void alightPassenger(String liftName, int floorNo, Passenger passenger) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, passenger != null ? passenger.toString() : null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : alightPassenger : "+s);  
 }  
  
 */\*\*  
 \* Method to open the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void openDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : openDoor :"+s);  
 }  
  
 */\*\*  
 \* Method to close the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void closeDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : closeDoor :"+s);  
 }  
  
 */\*\*  
 \* Method to display the lift's thread has initialised in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void started(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : started : "+liftName);  
 }  
  
 */\*\*  
 \* Method to display the lift's tread has terminated in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void stopped(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : stopped : "+liftName);  
 }  
  
  
 private Lift findLiftByName(String liftName) {  
 return Arrays.*asList*(this.lifts).stream().filter(l -> l.getName().equals(liftName)).findAny().get();  
 }  
  
 private Floor findFloor(int floorNo) {  
// return Arrays.asList(this.floors).stream().filter(f -> f.getFloorNo() == (floorNo - 1)).findAny().get();  
 for(Floor floor : this.floors){  
 if(floor.getFloorNo() == floorNo){  
 return floor;  
 }  
 }  
 return null;  
 }  
}

### Panel Package

#### LiftPanel Class

package com.simulation.lift.panel;  
  
import com.simulation.lift.model.Lift;  
  
import javax.swing.\*;  
import java.awt.\*;  
  
public class LiftPanel extends JPanel {  
 private final int x;  
 private int y;  
 private final int width;  
 private final int height;  
 private int doorWidth;  
 private final Color backgroundColor;  
 private final Lift lift;  
  
 private int ty;  
  
 */\*\*  
 \* Constructs an instance of the object containing x, y, width, height,  
 \* backgroundColor and lift arguments.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *height panel's height  
 \** ***@param*** *backgroundColor panel's background colour  
 \** ***@param*** *lift Lift object  
 \*/* public LiftPanel(int x, int y, int width, int height, Color backgroundColor, Lift lift) {  
 this.x = x;  
 this.y = y;  
 this.width = width;  
 this.height = height;  
 this.doorWidth = width / 2;  
 this.backgroundColor = backgroundColor;  
 this.lift = lift;  
  
 setLayout(null);  
 setBounds(x, y, width, height \* lift.getMaxMoveDistance());  
 repaint();  
 }  
  
 */\*\*  
 \* Method gets the Lift object from this class.  
 \*  
 \** ***@return*** *Lift object  
 \*/* public Lift getLift() {  
 return lift;  
 }  
  
 */\*\*  
 \* Method paints the lift moving up.  
 \*  
 \** ***@param*** *move number of moves  
 \*/* public void moveUp(int move) {  
 if (move > 0 ) {  
 int count = 0;  
 int tmp = this.ty;  
 while (count < this.lift.getHeight() \* move) {  
 repaint();  
 this.ty += 5;  
 count += 5;  
 Lift.*waitFor*();  
 }  
 this.ty = tmp + this.lift.getHeight() \* move;  
 }  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the lift moving down.  
 \*  
 \** ***@param*** *move number of moves  
 \*/* public void moveDown(int move) {  
 int count = 0;  
 int tmp = this.ty;  
 while(count < this.lift.getHeight()\*move) {  
 repaint();  
 Lift.*waitFor*();  
 this.ty -= 5;  
 count += 5;  
 }  
 this.ty = tmp - this.lift.getHeight()\*move;  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the door of a lift opening.  
 \*/* public void openDoor() {  
 this.doorWidth = width / 2;  
 while(this.doorWidth > 0) {  
 this.doorWidth-=5;  
 repaint();  
 Lift.*waitFor*();  
 }  
 this.doorWidth = 0;  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the door of a lift closing.  
 \*/* public void closeDoor() {  
 while(this.doorWidth < width / 2) {  
 this.doorWidth+=5;  
 repaint();  
 Lift.*waitFor*();  
 }  
 this.doorWidth = width / 2;  
 repaint();  
 }  
  
 */\*\*  
 \* Method gets the x-axis of the lift.  
 \*  
 \** ***@return*** *x-axis  
 \*/* @Override  
 public int getX() {  
 return x;  
 }  
  
 */\*\*  
 \* Method paints the panel of a LiftPanel object.  
 \*  
 \** ***@param*** *g Graphics object  
 \*/* @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 setBackground(backgroundColor);  
  
 int tw = this.lift.getWidth() - 2;  
 int ty1 = (this.lift.getMaxFloor() - this.lift.getMinFloor()) \* this.lift.getHeight();  
 g.setColor(Color.*RED*);  
 g.drawRect(0, ty1 - this.ty, tw, this.lift.getHeight() - (50 / this.lift.getMaxFloor()));  
 g.setColor(Color.*LIGHT\_GRAY*);  
 g.fillRect(1, ty1-this.ty, this.doorWidth - 1, this.lift.getHeight() - (50 / this.lift.getMaxFloor()));  
 g.fillRect(tw - this.doorWidth + 1, ty1-this.ty, this.doorWidth - 1, this.lift.getHeight() - (50 / this.lift.getMaxFloor()));  
 }  
}

#### FloorPanel Class

package com.simulation.lift.panel;  
  
import com.simulation.lift.model.Floor;  
import com.simulation.lift.model.Lift;  
import com.simulation.lift.model.Passenger;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
public class FloorPanel extends JPanel {  
 */\*\*  
 \* Enum presenting the states of a passenger.  
 \*/* private enum PP {  
 *WAIT*,  
 *BOARD*,  
 *ALIGHT* }  
  
 private final int x;  
 private final int y;  
 private final int width;  
 private final int height;  
 private final Color backgroundColor;  
 private final Floor floor;  
  
 private List<Passenger> passengers;  
 private Passenger movePassenger;  
 private PP pp;  
  
 */\*\*  
 \* Constructs an instance of the object containing x, y, width, height,  
 \* backgroundColor and floor arguments.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *height panel's height  
 \** ***@param*** *backgroundColor panel's background colour  
 \** ***@param*** *floor floor number  
 \*/* public FloorPanel(int x, int y, int width, int height, Color backgroundColor, Floor floor) {  
 this.x = x;  
 this.y = y;  
 this.width = width;  
 this.height = height;  
 this.backgroundColor = backgroundColor;  
 this.floor = floor;  
 this.passengers = new ArrayList<>();  
 this.pp = PP.*WAIT*;  
  
 setLayout(null);  
 setBounds(x, y, width, height);  
 repaint();  
 }  
  
 */\*\*  
 \* Method gets the floor number.  
 \*  
 \** ***@return*** *floor number  
 \*/* public Floor getFloor() {  
 return floor;  
 }  
  
 */\*\*  
 \* Method paints the passengers who are waiting on the floor.  
 \*/* public void flashFloor() {  
 this.passengers = new ArrayList<>(this.floor.getPassengers());  
 this.pp = PP.*WAIT*;  
 repaint();  
 }  
  
 */\*\*  
 \* Method paints the passenger who is boarding a lift.  
 \*  
 \** ***@param*** *passenger Passenger object  
 \*/* public void flashBoardFloor(Passenger passenger) {  
 this.movePassenger = passenger;  
 int count = 3;  
 while(count > 0) {  
 this.pp = PP.*BOARD*;  
 repaint();  
 Lift.*waitFor*();  
 count--;  
 }  
 Lift.*waitFor*();  
 flashFloor();  
 }  
  
 */\*\*  
 \* Method paints the passenger who is alighting from a lift.  
 \** ***@param*** *passenger Passenger object  
 \*/* public void flashAlightFloor(Passenger passenger) {  
 this.movePassenger = passenger;  
 int count = 3;  
 while(count > 0) {  
 this.pp = PP.*ALIGHT*;  
 repaint();  
 Lift.*waitFor*();  
 count--;  
 }  
 Lift.*waitFor*();  
 flashFloor();  
 }  
  
 */\*\*  
 \* Method paints moving passengers to/from lifts.  
 \*  
 \** ***@param*** *g Graphics object  
 \** ***@param*** *x x axis move  
 \*/* private void drawMovePassengers(Graphics g, int x) {  
 g.fillRect(x, this.height / 2, 20, this.height / 2);  
 g.setFont(new Font("SansSerif", Font.*PLAIN*, 10));  
 g.drawString("W: " + this.movePassenger.getWeight(), x - 2, this.height / 2 - 18);  
 g.drawString("S: " + (this.movePassenger.getSourceFloor()), x - 2, this.height / 2 - 10);  
 g.drawString("D: " + (this.movePassenger.getDestinationFloor()), x - 2, this.height / 2 - 20);  
 }  
  
 */\*\*  
 \* Method paints the PassengerPanel objects.  
 \*  
 \** ***@param*** *g Graphics object  
 \*/* @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 setBackground(backgroundColor);  
  
 int count = 0;  
 int tx = this.width \* 3 / 4 - 25;  
 while (tx > (this.width / 4) && count < this.passengers.size()) {  
 Passenger p = this.passengers.get(count);  
 g.fillRect(tx, this.height / 2, 20, this.height /2);  
 g.setFont(new Font("SansSerif", Font.*PLAIN*, 10));  
 g.drawString("W: " + p.getWeight(), tx - 2, this.height / 2 - 18);  
 g.drawString("S: " + (p.getSourceFloor()), tx - 2, this.height / 2 - 10);  
 g.drawString("D: " + (p.getDestinationFloor()), tx - 2, this.height / 2 - 2);  
 tx -= 30;  
 count++;  
 }  
  
 if (this.pp != PP.*WAIT*) {  
 if (this.pp == PP.*BOARD*) {  
 tx = this.width \* 3 / 4;  
 while (tx < this.width) {  
 drawMovePassengers(g, tx);  
 tx += 5;  
 }  
 } else if (this.pp == PP.*ALIGHT*) {  
 tx = this.width/4 - 20;  
 while (tx > 0) {  
 drawMovePassengers(g, tx);  
 tx -= 5;  
 }  
 }  
 }  
 }  
  
}

#### BuildingPanel class

package com.simulation.lift.panel;  
  
import com.simulation.lift.api.LiftListener;  
import com.simulation.lift.model.Building;  
import com.simulation.lift.model.Floor;  
import com.simulation.lift.model.Lift;  
import com.simulation.lift.model.Passenger;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.util.Arrays;  
import java.util.Random;  
  
public class BuildingPanel extends JPanel implements LiftListener {  
 private final int x;  
 private final int y;  
 private final int width;  
 private final int height;  
 private final Color backgroundColor;  
 private final Building building;  
  
 private final LiftPanel[] liftPanels;  
 private final FloorPanel[] floorPanels;  
  
 private Random random;  
 private Timer passengerTimer;  
 private Timer timer;  
 private int refreshCounter;  
 private int timeElapsedInSecs;  
 private final int passengerVolume;  
 private final int maxFloor;  
 private final int maxLift;  
 private final int liftWeight;  
  
 private final boolean isAuto;  
  
 */\*\*  
 \* Constructs an instance of the object containing x, y, width, height,  
 \* backgroundColor, maxFloor, maxLift, liftWeight and passengerVolume arguments.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *height panel's height  
 \** ***@param*** *backgroundColor panel's background colour  
 \** ***@param*** *maxFloor maximum floor number  
 \** ***@param*** *liftWeight maximum lift capacity  
 \** ***@param*** *passengerVolume volume of passengers  
 \** ***@param*** *isAuto is the program auto  
 \*/* public BuildingPanel(int x, int y, int width, int height, Color backgroundColor, int maxFloor, int maxLift, int liftWeight, int passengerVolume, boolean isAuto) {  
 this.isAuto = isAuto;  
  
 this.x = x;  
 this.y = y;  
 this.width = width;  
 this.height = height - 20;  
 this.backgroundColor = backgroundColor;  
 this.building = new Building( maxFloor, this.height / maxFloor, width/2, maxLift, this.height / maxFloor,width/(2\*maxLift), liftWeight, this.isAuto);  
  
 this.floorPanels = new FloorPanel[maxFloor];  
 this.liftPanels = new LiftPanel[maxLift];  
  
 this.passengerVolume = passengerVolume;  
 this.refreshCounter = 0;  
 this.timeElapsedInSecs = 0;  
 this.maxFloor = maxFloor;  
 this.maxLift = maxLift;  
 this.liftWeight = liftWeight;  
  
 this.random = new Random();  
  
 setLayout(null);  
 setBounds(x, y, width, height);  
 repaint();  
 init();  
  
 if(this.isAuto) {  
 this.passengerTimer = new Timer(100, (e) -> {  
 this.refreshCounter += 100;  
 if (this.refreshCounter / 1000 == 1) {  
 this.refreshCounter = 0;  
 this.timeElapsedInSecs++;  
 if (this.timeElapsedInSecs % 5 == 0) {  
 generateNewPassengers();  
 }  
 }  
 });  
 }  
 else {  
 setPassengerFloor(passengerVolume);  
 }  
  
 this.timer = new Timer(100, (e) -> {  
 repaint();  
 });  
 }  
  
 */\*\*  
 \* Method starts the project, by initialising the threads.  
 \*/* public void start() {  
 if(this.isAuto) {  
 //starting the timer's thread.  
 Runnable passengerRunnable = () -> this.passengerTimer.start();  
 Thread passengerThread = new Thread(passengerRunnable);  
 passengerThread.setDaemon(true);  
 passengerThread.setName("PassengerTimer");  
 passengerThread.start();  
 }  
  
 //starting the cost updater thread.  
 Runnable runnable = () -> this.timer.start();  
 Thread thread = new Thread(runnable);  
 thread.setDaemon(true);  
 thread.setName("Timer");  
 thread.start();  
  
 //starting the lifts' threads.  
 for (int i = 0; i < this.liftPanels.length; i++) {  
 Lift lift = this.liftPanels[i].getLift();  
 runnable = () -> lift.start();  
 thread = new Thread(runnable);  
 thread.setDaemon(true);  
 thread.setName(lift.getName());  
 thread.start();  
 }  
 }  
  
 */\*\*  
 \* Method stops the project, by terminate the the threads.  
 \*/* public void stop() {  
 if(this.isAuto) {  
 this.passengerTimer.stop();  
 }  
 this.timer.stop();  
 for (int i = 0; i < this.liftPanels.length; i++) {  
 this.liftPanels[i].getLift().stop();  
 }  
 }  
  
 */\*\*  
 \* Method initialise the content of the class in the construct.  
 \*/* private void init() {  
 int x = 5;  
 int y = 5;  
 int w = this.width/2;  
 int h = this.height / this.maxFloor;  
 createFloors(x, y + 20, w, Color.*ORANGE*);  
 createLifts(x + w, y + 20, h, Color.*DARK\_GRAY*);  
 }  
  
 */\*\*  
 \* Method draws the lift panels on top of the building panel.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *height panel's height  
 \** ***@param*** *color panel's background colour  
 \*/* private void createLifts(int x, int y, int height, Color color) {  
 int w1 = this.building.getLiftWidth();  
  
 for (int i = 1; i <= this.building.getMaxLift(); i++) {  
 LiftPanel liftPanel = new LiftPanel(x, y, w1-(50/this.maxFloor), height, color, this.building.getLifts()[this.building.getMaxLift()-i]);  
 liftPanel.getLift().setListener(this);  
 this.liftPanels[this.building.getMaxLift() - i] = liftPanel;  
 add(liftPanel);  
 x += w1;  
 }  
 }  
  
 */\*\*  
 \* Method draws the floor panels on top of the building panel.  
 \*  
 \** ***@param*** *x panel's starting x value  
 \** ***@param*** *y panel's starting y value  
 \** ***@param*** *width panel's width  
 \** ***@param*** *color panel's background colour  
 \*/* private void createFloors(int x, int y, int width, Color color) {  
 int h1 = this.building.getFloorHeight();  
  
 for (int i = 1; i <= this.building.getMaxFloor(); i++) {  
 this.floorPanels[this.building.getMaxFloor() - i] = new FloorPanel(x, y, width, h1-(50/this.maxFloor), color, this.building.getFloors()[this.building.getMaxFloor()-i]);  
 add(this.floorPanels[this.building.getMaxFloor()-i]);  
 y += h1;  
 }  
 }  
  
 */\*\*  
 \* Method paints the panel of this class.  
 \*  
 \** ***@param*** *g Graphic object  
 \*/* @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 setBackground(backgroundColor);  
  
 g.setColor(Color.*BLUE*);  
 g.drawRect(5, 2, this.width / 8, 20);  
 g.drawRect(this.width / 8 + 5, 2, this.width / 4, 20);  
 g.drawRect(this.width / 8 + this.width / 4 + 5, 2, this.width / 8, 20);  
  
 g.setColor(Color.*BLACK*);  
 g.drawString("Alight", 40, 17);  
 g.drawString("Waiting Passengers", 150, 17);  
 g.drawString("Board", 355, 17);  
  
 g.setColor(Color.*RED*);  
 g.drawRect(this.width / 2 + 10, 2, 135, 20);  
 g.drawRect(this.width / 2 + 150, 2, 135, 20);  
  
 g.setColor(Color.*BLUE*);  
 g.drawString("Total Board: " + this.building.calculateTotalBoardPassengers(), this.width / 2 + 15, 17);  
 g.drawString("Total Alight: " + this.building.calculateTotalAlightPassengers(), this.width / 2 + 155, 17);  
  
 g.setColor(Color.*BLUE*);  
 g.drawRect(this.width / 2 + 300, 2, 115, 20);  
  
 g.setColor(Color.*red*);  
 g.drawString("Total Cost: " + this.building.calculateCost(), this.width / 2 + 310, 17);  
 }  
  
 */\*\*  
 \* Method moves the lift up by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void upMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: %s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : upMoves : "+s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.moveUp(moveTo);  
 }  
  
 */\*\*  
 \* Method moves the lift down by certain floors.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* @Override  
 public void downMoves(String liftName, int fromFloorNo, int moveTo) {  
 String s = String.*format*(" liftName: $s, fromFloorNo: %s, moveTo: %s", liftName, fromFloorNo, moveTo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : downMoves : "+ s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.moveDown(moveTo);  
 }  
  
 */\*\*  
 \* Method adds the passengers who are boarding a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *lastPassenger previous passenger  
 \** ***@param*** *isFull is the lift full  
 \** ***@return*** *passenger who could not board  
 \*/* @Override  
 public Passenger getBoardPassenger(String liftName, int floorNo, Passenger lastPassenger, boolean isFull) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, lastPassenger != null ? lastPassenger.toString(): null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : "+s);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : getBoardPassenger : isFull: "+isFull);  
  
 FloorPanel floorPanel = findFloorPannel(floorNo);  
 if(!isFull) {  
 Passenger passenger = floorPanel.getFloor().removePassenger();  
 if(passenger == null) {  
 return null;  
 }  
 floorPanel.flashBoardFloor(passenger);  
 return passenger;  
 }  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 floorPanel.getFloor().addReturnPassenger(lastPassenger);  
 liftPanel.getLift().setArriveFloor(floorNo);  
 floorPanel.flashFloor();  
 return null;  
 }  
  
 */\*\*  
 \* Method removes the passenger who are alighting from a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *passenger alighting passenger  
 \*/* @Override  
 public void alightPassenger(String liftName, int floorNo, Passenger passenger) {  
 String s = String.*format*(" liftName: %s, floorNo: %s, lastPassenger: %s", liftName, floorNo, passenger != null ? passenger.toString() : null);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : alightPassenger : "+s);  
  
 FloorPanel floorPanel = findFloorPannel(floorNo);  
 floorPanel.flashAlightFloor(passenger);  
 }  
  
 */\*\*  
 \* Method to open the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void openDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : openDoor :"+s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.openDoor();  
 }  
  
 */\*\*  
 \* Method to close the lift's door.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \*/* @Override  
 public void closeDoor(String liftName, int floorNo) {  
 String s = String.*format*(" liftName: %s, floorNo: %s", liftName, floorNo);  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : closeDoor :"+s);  
  
 LiftPanel liftPanel = findLiftByName(liftName);  
 liftPanel.closeDoor();  
 }  
  
 */\*\*  
 \* Method to display the lift's thread has initialised in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void started(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : started : "+liftName);  
 }  
  
 */\*\*  
 \* Method to display the lift's tread has terminated in the console.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \*/* @Override  
 public void stopped(String liftName) {  
 System.*out*.println("\*\*\*\*\*\*\*\*\* : stopped : "+liftName);  
 }  
  
 */\*\*  
 \* Method randomly allocate source and destination floors to the new passengers.  
 \*/* private synchronized void generateNewPassengers() {  
 System.*out*.println("Source aaaaaaaaa");  
 int numOfPassenger = random.nextInt(this.passengerVolume) + 1;  
  
 setPassengerFloor(numOfPassenger);  
 }  
  
 */\*\*  
 \* Method sets the passenger's source and destination floor.  
 \*  
 \** ***@param*** *PassengerNo number of passengers  
 \*/* private void setPassengerFloor(int PassengerNo) {  
 for (int i = 0; i < PassengerNo; i++) {  
 int sourceFloor;  
 do {  
 sourceFloor = this.random.nextInt(this.maxFloor) + 1;  
 } while (sourceFloor < 1 || sourceFloor > this.maxFloor);  
  
 int liftNo;  
 do {  
 liftNo = this.random.nextInt(this.maxLift) + 1;  
 } while (liftNo < 1 || liftNo > this.maxLift);  
  
 int destinationFloor;  
 do {  
 destinationFloor = this.random.nextInt(this.maxFloor)+ 1;  
 } while(destinationFloor == sourceFloor || destinationFloor < 1 || destinationFloor > this.maxFloor);  
  
 Floor floor = this.floorPanels[sourceFloor-1].getFloor();  
  
 if(floor.getPassengers().size() < this.passengerVolume) {  
  
 int weight;  
 do {  
 weight = this.random.nextInt(120) + 50;  
 } while (weight > this.liftWeight);  
  
 floor.addNewPassenger(new Passenger(weight, sourceFloor, destinationFloor));  
 this.floorPanels[sourceFloor-1].flashFloor();  
  
 System.*out*.println("Source "+sourceFloor);  
  
 this.liftPanels[liftNo-1].getLift().setArriveFloor(sourceFloor);  
 }  
 }  
 }  
  
 */\*\*  
 \* Method used to finds a given lift by it's name.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@return*** *LiftPanel object  
 \*/* private LiftPanel findLiftByName(String liftName) {  
 return Arrays.*asList*(this.liftPanels).stream().filter(l -> l.getLift().getName().equals(liftName)).findAny().get();  
 }  
  
 */\*\*  
 \* Method used to find a given floor by a number.  
 \*  
 \** ***@param*** *floorNo floor number  
 \** ***@return*** *FloorPanel object  
 \*/* private FloorPanel findFloorPannel(int floorNo) {  
 return this.floorPanels[floorNo-1];  
 }  
  
 private LiftPanel findLiftPannel(int liftNo) {  
 return this.liftPanels[liftNo-1];  
 }  
  
}

#### SimulationPanel Class

package com.simulation.lift.panel;  
  
import javax.swing.\*;  
import java.awt.\*;  
  
public class SimulationPanel extends JPanel {  
 private BuildingPanel building;  
  
 */\*\*  
 \* Constructs an instance of the object containing maxBuildingFloor,  
 \* liftWeight, passengerVolume and liftNo arguments.  
 \*  
 \** ***@param*** *maxBuildingFloor max floor of the building  
 \** ***@param*** *liftWeight lift weight capacity  
 \** ***@param*** *passengerVolume volume of incoming passenger at each refreshing rate  
 \** ***@param*** *isAuto is the program auto  
 \*/* public SimulationPanel(int maxBuildingFloor, int liftWeight, int passengerVolume, int liftNo, Boolean isAuto) {  
 setLayout(null);  
 setPreferredSize(new Dimension(855, 630));  
  
 this.building = new BuildingPanel(0, 0,850, 620, Color.*WHITE*, maxBuildingFloor, liftNo, liftWeight, passengerVolume, isAuto);  
 //starting all the treads to run in the program.  
 this.building.start();  
  
 add(building);  
 }  
  
 */\*\*  
 \* Method stops all the treads in the BuildPanel class the stop the program.  
 \*/* public void stop() {  
 building.stop();  
 }  
}

### API Package

#### LiftListener Interface

package com.simulation.lift.api;  
  
import com.simulation.lift.model.Passenger;  
  
public interface LiftListener {  
 */\*\*  
 \* Method moves the lift up depending on certain number of moves.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* void upMoves(String liftName, int fromFloorNo, int moveTo);  
  
 */\*\*  
 \* Method moves the lift down depending on certain number of moves.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *fromFloorNo source floor  
 \** ***@param*** *moveTo destination floor  
 \*/* void downMoves(String liftName, int fromFloorNo, int moveTo);  
  
 */\*\*  
 \* Method gets the passenger who are boarding a lift.  
 \*  
 \** ***@param*** *liftName name of the lift  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *lastPassenger previous passenger  
 \** ***@param*** *isFull is the lift full  
 \** ***@return*** *Passenger object  
 \*/* Passenger getBoardPassenger(String liftName, int floorNo, Passenger lastPassenger, boolean isFull);  
  
 */\*\*  
 \* Method removes the passengers who are alighting from a lift.  
 \*  
 \** ***@param*** *liftName name of the life  
 \** ***@param*** *floorNo floor number  
 \** ***@param*** *passenger Passenger object  
 \*/* void alightPassenger(String liftName, int floorNo, Passenger passenger);  
  
 */\*\*  
 \* Method opens the door of a lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \** ***@param*** *floorNo floor number  
 \*/* void openDoor(String liftName, int floorNo);  
  
 */\*\*  
 \* Method closes the door of a lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \** ***@param*** *floorNo floor number  
 \*/* void closeDoor(String liftName, int floorNo);  
  
 */\*\*  
 \* Method starts the lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \*/* void started(String liftName);  
  
 */\*\*  
 \* Method stops the lift.  
 \*  
 \** ***@param*** *liftName lift name  
 \*/* void stopped(String liftName);  
}

### Main Package

#### Menu Class

package com.simulation.lift.main;  
  
import com.simulation.lift.model.Building;  
import com.simulation.lift.panel.SimulationPanel;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.WindowAdapter;  
import java.awt.event.WindowEvent;  
import java.io.\*;  
import java.util.Hashtable;  
  
import static javax.swing.JOptionPane.*showMessageDialog*;  
  
public class Menu extends JPanel {  
  
 */\*\*  
 \* Constructs an instance of the object with no arguments.  
 \*/* public Menu() {  
 initializeMenu();  
 }  
  
 */\*\*  
 \* Method initialize label in the Menu.  
 \*  
 \** ***@param*** *y y-axis of the label  
 \** ***@param*** *width width of the label  
 \** ***@param*** *message message to display in the label  
 \*/* private void initializeLabel(int x, int y, int width, String message) {  
 JPanel panel = new JPanel();  
 panel.setBounds(x, y, width, 20);  
 add(panel);  
 panel.setLayout(null);  
  
 JLabel label = new JLabel(message);  
 label.setBounds(0, 0, width, 20);  
 panel.add(label);  
 }  
  
 */\*\*  
 \* Method initialize user input form in the Menu.  
 \*/* private void initializeInput() {  
 JPanel panel = new JPanel();  
 panel.setBounds(120, -5, 50, 30);  
 add(panel);  
  
 JTextField floorNo = new JTextField(3);  
 panel.add(floorNo);  
  
 JPanel panel1 = new JPanel();  
 panel1.setBounds(250, 33, 100, 30);  
 add(panel1);  
  
 JTextField passengerNo = new JTextField(6);  
 panel1.add(passengerNo);  
  
 JPanel panel2 = new JPanel();  
 panel2.setBounds(160, 60, 190, 60);  
 add(panel2);  
  
 JSlider passengerVolume = new JSlider();  
 passengerVolume.setMajorTickSpacing(50);  
 passengerVolume.setPaintTicks(true);  
 passengerVolume.setPaintTrack(true);  
 passengerVolume.setPaintLabels(true);  
 Hashtable position = new Hashtable();  
 position.put(0, new JLabel("low"));  
 position.put(50, new JLabel("medium"));  
 position.put(100, new JLabel("high"));  
 passengerVolume.setLabelTable(position);  
 passengerVolume.setEnabled(false);  
 panel2.add(passengerVolume);  
  
 JPanel panel3 = new JPanel();  
 panel3.setBounds(110, 120, 80, 30);  
 add(panel3);  
  
 String[] liftWeightList = {"500", "750", "1000", "1250", "1500", "1750", "2000", "2250", "2500", "2750", "3000"};  
 JComboBox liftWeight = new JComboBox(liftWeightList);  
 panel3.add(liftWeight);  
  
 JPanel panel4 = new JPanel();  
 panel4.setBounds(200, 155, 80, 30);  
 add(panel4);  
  
 JPanel panel5 = new JPanel();  
 panel5.setBounds(30, 32, 50, 30);  
 add(panel5);  
  
 JRadioButton isAuto = new JRadioButton();  
 isAuto.addActionListener(e -> {  
 if(isAuto.isSelected()) {  
 passengerVolume.setEnabled(true);  
 passengerNo.setText("");  
 passengerNo.setEditable(false);  
 }  
 else {  
 passengerVolume.setEnabled(false);  
 passengerNo.setEditable(true);  
 }  
 });  
 panel5.add(isAuto);  
  
 JPanel panel6 = new JPanel();  
 panel6.setBounds(345, -3, 50, 30);  
 add(panel6);  
  
 JRadioButton isGraph = new JRadioButton();  
 isGraph.addActionListener(e -> {  
 if(isGraph.isSelected()) {  
 floorNo.setEditable(false);  
 isAuto.setEnabled(false);  
 passengerVolume.setEnabled(false);  
 passengerNo.setText("");  
 passengerNo.setEditable(false);  
 liftWeight.setEnabled(false);  
 }  
 else {  
 floorNo.setEditable(true);  
 isAuto.setEnabled(true);  
 passengerVolume.setEnabled(true);  
 passengerNo.setEditable(true);  
 liftWeight.setEnabled(true);  
 }  
 });  
 panel6.add(isGraph);  
  
  
 JButton submitButton = new JButton("Submit");  
 submitButton.addActionListener(e -> {  
 if (isGraph.isSelected()) {  
 try{  
 String prg = "cost\_dict = {}\n" +  
 "\n" +  
 "def get\_value\_by\_key(obj: dict, key: str):\n" +  
 " try:\n" +  
 " return obj[key]\n" +  
 " except:\n" +  
 " return None\n" +  
 "\n" +  
 "def update\_dict(lift\_capacity, floor\_number, passenger\_number, cost):\n" +  
 "\n" +  
 " lift\_capacity\_obj = get\_value\_by\_key(cost\_dict, lift\_capacity)\n" +  
 " if not lift\_capacity\_obj:\n" +  
 " lift\_capacity\_obj = {lift\_capacity: {}}\n" +  
 " cost\_dict.update(lift\_capacity\_obj)\n" +  
 " lift\_capacity\_obj = cost\_dict[lift\_capacity]\n" +  
 "\n" +  
 " floor\_number\_obj = get\_value\_by\_key(lift\_capacity\_obj, floor\_number)\n" +  
 " if not floor\_number\_obj:\n" +  
 " floor\_number\_obj = {floor\_number: {}}\n" +  
 " lift\_capacity\_obj.update(floor\_number\_obj)\n" +  
 " floor\_number\_obj = lift\_capacity\_obj[floor\_number]\n" +  
 "\n" +  
 " passenger\_number\_obj = get\_value\_by\_key(floor\_number\_obj, passenger\_number)\n" +  
 " if not passenger\_number\_obj:\n" +  
 " passenger\_number\_obj = {passenger\_number: {\"cost\": cost}}\n" +  
 " floor\_number\_obj.update(passenger\_number\_obj)\n";  
  
 for(int weightCounter = 500; weightCounter <= 3000; weightCounter+=250) {  
 for(int floorCounter = 5; floorCounter <= 300; floorCounter+=5) {  
 for(int passengerCounter = 5; passengerCounter <= 1000; passengerCounter+=5) {  
 Building building = new Building(floorCounter, 620 / floorCounter, 1, floorCounter / 3, 1,1, weightCounter, passengerCounter, false);  
 building.start().get();  
  
 prg = prg + "\n" +  
 "update\_dict(" + weightCounter + ", " + floorCounter + ", " + passengerCounter + ", " + building.calculateCost() + ")\n";  
 }  
 }  
 }  
  
 prg = prg + "\nprint(cost\_dict)\n";  
  
 System.*out*.println(prg);  
  
 BufferedWriter out = new BufferedWriter(new FileWriter("src/com/simulation/lift/api/CostDict.py"));  
 out.write(prg);  
 out.close();  
  
 Process p = Runtime.*getRuntime*().exec("python3 src/GraphPanel.py");  
 BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));  
 String ret = in.readLine();  
 System.*out*.println("value is : "+ret);  
 }catch(Exception a){  
 a.printStackTrace();  
 }  
 } else if (checkValidity(floorNo, 3, 600)) {  
 if(isAuto.isSelected()) {  
 if (passengerVolume.getValue() >= 0 && passengerVolume.getValue() <= 33) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(floorNo.getText()) / 2, Integer.*parseInt*(floorNo.getText()) / 3, true);  
 }  
 if (passengerVolume.getValue() > 33 && passengerVolume.getValue() < 67) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*(floorNo.getText()) / 3, true);  
 }  
 if (passengerVolume.getValue() >= 67 && passengerVolume.getValue() <= 100) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(floorNo.getText()) \* 2, Integer.*parseInt*(floorNo.getText()) / 3, true);  
 }  
 }  
 else {  
 if(checkValidity(passengerNo, 1, Integer.*MAX\_VALUE*)) {  
 startLiftSimulation(Integer.*parseInt*(floorNo.getText()), Integer.*parseInt*((String) liftWeight.getSelectedItem()), Integer.*parseInt*(passengerNo.getText()), Integer.*parseInt*(floorNo.getText()) / 3, false);  
 }  
 }  
 }  
 });  
 panel4.add(submitButton);  
 }  
  
 */\*\*  
 \* Method initialize the Menu.  
 \*/* private void initializeMenu() {  
 setLayout(null);  
 setPreferredSize(new Dimension(500, 195));  
  
 initializeLabel(5, 5, 120, "Number of Floors:");  
 initializeLabel(5, 40, 35, "Auto:");  
 initializeLabel(150, 40, 120, "Total Passenger:");  
 initializeLabel(5, 80, 150, "Volume of Passengers:");  
 initializeLabel(5, 130, 105, "Max Lift Weight:");  
 initializeLabel(275, 5, 85, "Draw Graph:");  
  
 initializeInput();  
 }  
  
 */\*\*  
 \* Method validate the user input entered in the Menu.  
 \*  
 \** ***@param*** *input user input  
 \** ***@param*** *min minimum accepted value  
 \** ***@param*** *max maximum accepted value  
 \** ***@return*** *is the user input valid  
 \*/* private boolean checkValidity(JTextField input, int min, int max) {  
 try {  
 Integer.*parseInt*(input.getText());  
 } catch (NumberFormatException e) {  
 *showMessageDialog*(null, "The input must be a number.", "Error", 0);  
 return false;  
 }  
 if (Integer.*parseInt*(input.getText()) < min || Integer.*parseInt*(input.getText()) > max) {  
 *showMessageDialog*(null, "The input must be between " + min + " and " + max + ".", "Error", 0);  
 return false;  
 }  
 return true;  
 }  
  
 */\*\*  
 \* Method to start the simulation.  
 \*  
 \** ***@param*** *maxBuildingFloor max floor of the building  
 \** ***@param*** *liftWeight lift weight capacity  
 \** ***@param*** *passengerVolume volume of incoming passenger at each refreshing rate  
 \** ***@param*** *isAuto is the program auto  
 \*/* private void startLiftSimulation(int maxBuildingFloor, int liftWeight, int passengerVolume, int liftNo, boolean isAuto) {  
 JFrame jf = new JFrame();  
 SimulationPanel simulationPanel = new SimulationPanel(maxBuildingFloor, liftWeight, passengerVolume, liftNo, isAuto);  
 jf.add(simulationPanel);  
 jf.setTitle("Lift Simulator");  
 jf.pack();  
 jf.setLocation(250,75);  
 jf.setVisible(true);  
// jf.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  
 jf.addWindowListener(new WindowAdapter() {  
 @Override  
 public void windowClosing(WindowEvent e) {  
 simulationPanel.stop();  
 System.*out*.println("Closed");  
 e.getWindow().dispose();  
 }  
 });  
 jf.setResizable(false);  
 }  
}

#### MAin Class

package com.simulation.lift.main;  
  
import javax.swing.\*;  
  
public class Main {  
  
 public static void main(String[] args) {  
 JFrame jf = new JFrame();  
 jf.add(new Menu());  
 jf.setTitle("Main Menu");  
 jf.pack();  
 jf.setLocation(400,100);  
 jf.setVisible(true);  
 jf.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 jf.setResizable(false);  
 }  
}

### Graph Python File

#### GraphPanel Module

import matplotlib.pyplot as plt  
from com.simulation.lift.api.CostDict import cost\_dict  
  
fig = plt.figure()  
fig.canvas.set\_window\_title('Base Case Graph')  
fig.suptitle("""Cost  
in function of   
Lift Capacity, Floor Number and Passenger Number""")  
  
ax = fig.add\_subplot(111, projection='3d')  
  
x = []  
y = []  
z = []  
c = []  
  
for lift\_capacity in cost\_dict:  
 for floor\_number in cost\_dict[lift\_capacity]:  
 for passenger\_number in cost\_dict[lift\_capacity][floor\_number]:  
 x.append(lift\_capacity)  
 y.append(floor\_number)  
 z.append(passenger\_number)  
 c.append(cost\_dict[lift\_capacity][floor\_number][passenger\_number]["cost"])  
  
img = ax.scatter(x, y, z, c=c, cmap=plt.hot())  
cbar = fig.colorbar(img)  
cbar.ax.get\_yaxis().labelpad = 15; cbar.ax.set\_ylabel("Cost", rotation = 270);  
ax.set\_xlabel('Lift Capacity')  
ax.set\_ylabel('Floor Number')  
ax.set\_zlabel('Passenger Number')  
  
plt.show()