CSC 8350 – Advanced Software Engineering

PARKADE “The Multi-Storey Parking Lot System” using DEVS-JAVA

Hands-on Project Final Report

Team 3

Sai Kowshik Ananthula (002638298)

Rujula Malineni (002571938)

1. PROBLEM STATEMENT

Managing a parking lot is a complicated system that involves tasks such as reserving a parking spot, managing vehicles depending on their size, giving fast service to vehicles, charging the appropriate amount, and so on. As a result, having a well-organized system that keeps track of parking spots and properly distributes them based on vehicle size is crucial. It is also necessary to guarantee that all available resources are used in accordance with request priorities, and that the vehicle is paid efficiently based on the parking space used, with no resources staying occupied for longer than the system lot allocation allows. Also, efficiently allocate and reuse the finite space for parking lot with multiple stories, organize and provide the respective slots to the oncoming customers.

2. MODELING AND SIMULATION GOALS

Our goal is to address the challenges with the parking lot reservation management system using a systematic flow. User preferences are collected and used as inputs to categorize the vehicles. These factors can be used to help distribute parking places and allocate resources more efficiently. Our goal is to ensure that no parking space is vacant at any one time, that vehicles are parked just for the time allotted, and that all available spaces are occupied and utilized effectively. In addition, we want to record for how long a vehicle has been parked so that we can rapidly calculate the required parking space fee depending on the vehicle's needs.

3. MODEL DESIGN AND DESCRIPTION

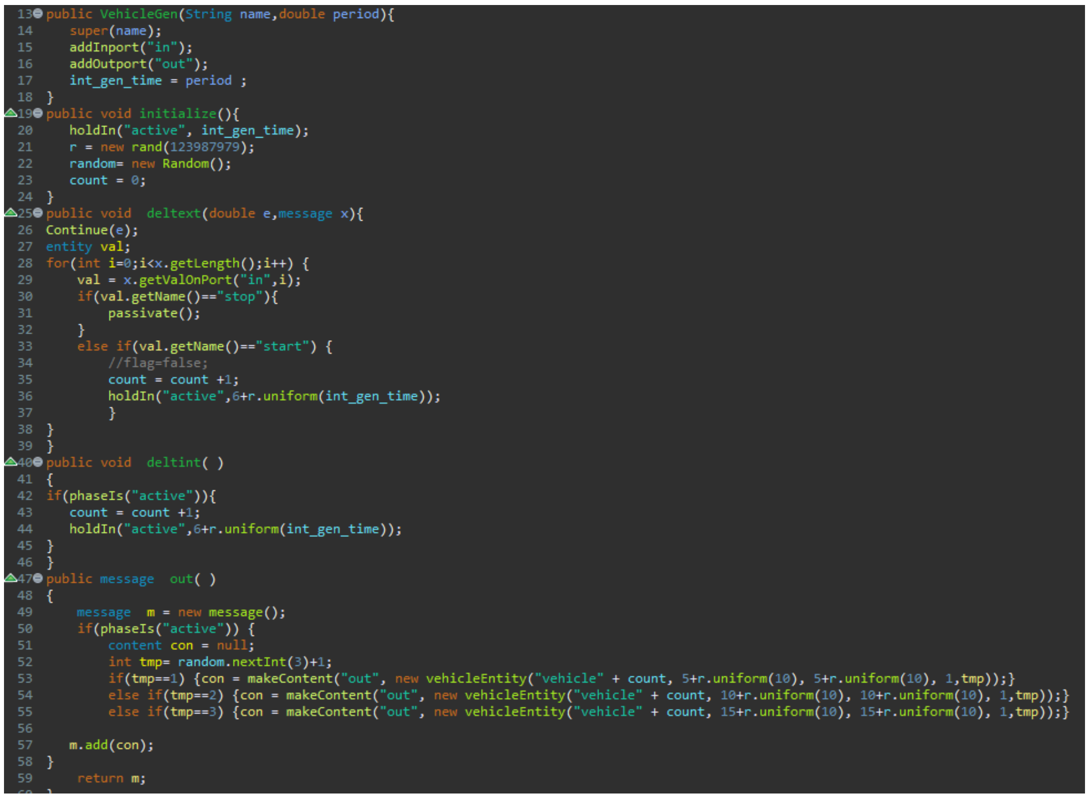
The trending online reservation system has mostly encouraged parking lot reservations. This was considered when creating this simulation model. Customers are initially given access to an online site that collects information from them, such as the type of vehicle they have (whether it's a car, motorcycle, or truck), the period of parking (how long the user plans to park the vehicle in the parking lot), and so on. This information was gathered to classify the vehicles and assign parking spaces to them. After the categories have been assigned, the requests are submitted to a parkingslotManager class, which determines which areas will be assigned to these requests. The parkingslotManager is kept in a queue to process one request at a time. The requests are subsequently forwarded to Building class, which allocates a time slot that the user requested for the incoming vehicle. A checkout counter has been considered for this purpose. Each incoming request has been given a specific time at the checkout center. As a result, we ensure that the slot bookings, slot allocation, and payment systems operate systematically, maximizing resource utilization while minimizing difficulties during slot allocation. Also, a Transducer is allocated to book keep arrival times, solved times and the total price for the vehicles.

Each atomic model shows some information when we hover over it. For suppose when we hover over parkingSpace atomic model we will find information such as current floor, current job it is doing and number of cars waiting in the queue.

4. DESIGN:

4.1 VehicleGen

VehicleGen component generates the type of vehicle as car, motorcycle, and truck. This component is initially inactive with a sigma value of 7. The VehicleGen randomly generates a new request, maintaining an active state for that frequency. The outport of VehicleGen is connected to the ParkingSlotManager. The type of vehicle if decided by the slots attribute of each vehicle.

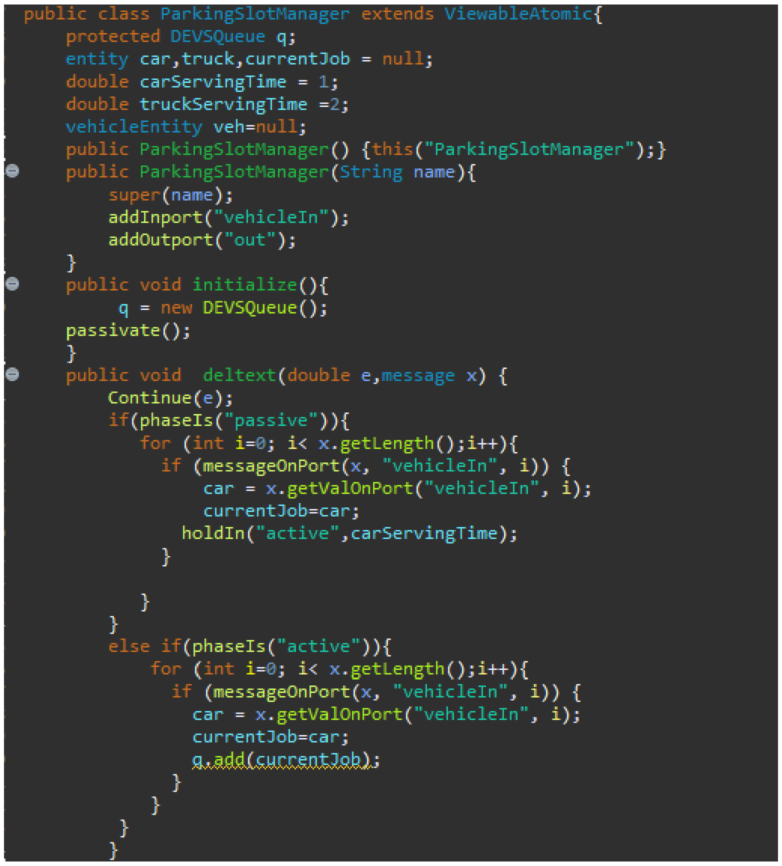


```java
public VehicleGen(String name,double period){
    super(name);
    addInport("in");
    addOutport("out");
    int_gen_time = period ;
}
public void initialize(){
    holdIn("active", int_gen_time);
    r = new rand(123987979);
    random= new Random();
    count = 0;
}
public void  deltext(double e,message x){
Continue(e);
entity val;
for(int i=0;i<x.getLength();i++) {
     val = x.getValOnPort("in",i);
     if(val.getName()=="stop"){
         passivate();
     }
     else if(val.getName()=="start") {
         //flag=false;
         count = count +1;
         holdIn("active",6+r.uniform(int_gen_time));
         }
}
}
public void  deltint( )
{
if(phaseIs("active")){
    count = count +1;
    holdIn("active",6+r.uniform(int_gen_time));
}
}
public message  out( )
{
     message  m = new message();
     if(phaseIs("active")) {
         content con = null;
         int tmp= random.nextInt(3)+1;
         if(tmp==1) {con = makeContent("out", new vehicleEntity("vehicle" + count, 5+r.uniform(10), 5+r.uniform(10), 1,tmp));}
         else if(tmp==2) {con = makeContent("out", new vehicleEntity("vehicle" + count, 10+r.uniform(10), 10+r.uniform(10), 1,tmp));}
         else if(tmp==3) {con = makeContent("out", new vehicleEntity("vehicle" + count, 15+r.uniform(10), 15+r.uniform(10), 1,tmp));}

    m.add(con);
}
     return m;
```

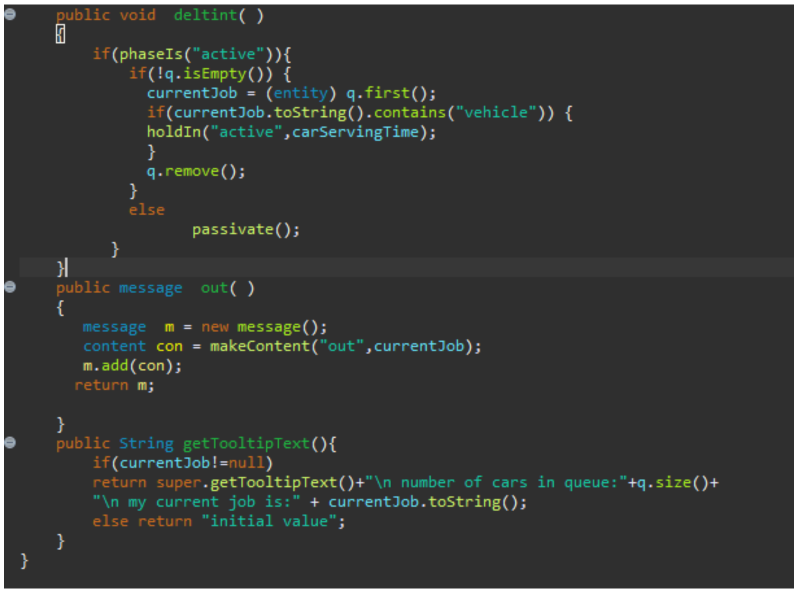
Fig.1 Code snippet for VehicleGen showing internal and external transition functions.

4.2 parkingSlotManager

The parkingslotManager component is the decision-maker of the flow. Initially, it is set to the wait state for an infinity amount of time. It gets data from the VehicleGen class. Whenever a new request is made to the in-port, it will determine if the request is made by a motorcycle, a car, or a truck depending on the request port. The request that arrived first will be served first. The parkingslotManager component has a queue for each incoming request apart from the one being processed. This queue will add all the requests when the parkingslotManager component is busy serving the other request. They are later processed in the same order in which they are received. It is changed to a busy state for a frequency of 1 to process each request. There is an out port to the parkingslotManager component called “out”. The outport “out” passes the processed request to “parking space” component. This atomic model displays the number of cars in queue and the current job it is serving when we hover on it.



```java
public class ParkingSlotManager extends ViewableAtomic{
    protected DEVSQueue q;
    entity car,truck,currentJob = null;
    double carServingTime = 1;
    double truckServingTime =2;
    vehicleEntity veh=null;
    public ParkingSlotManager() {this("ParkingSlotManager");}
    public ParkingSlotManager(String name){
        super(name);
        addInport("vehicleIn");
        addOutport("out");
    }
    public void initialize(){
         q = new DEVSQueue();
    passivate();
    }
    public void  deltext(double e,message x) {
        Continue(e);
        if(phaseIs("passive")){
           for (int i=0; i< x.getLength();i++){
             if (messageOnPort(x, "vehicleIn", i)) {
                 car = x.getValOnPort("vehicleIn", i);
                 currentJob=car;
               holdIn("active",carServingTime);
             }

           }
        }
        else if(phaseIs("active")){
            for (int i=0; i< x.getLength();i++){
              if (messageOnPort(x, "vehicleIn", i)) {
                car = x.getValOnPort("vehicleIn", i);
                currentJob=car;
                q.add(currentJob);
              }
            }
         }
        }
```

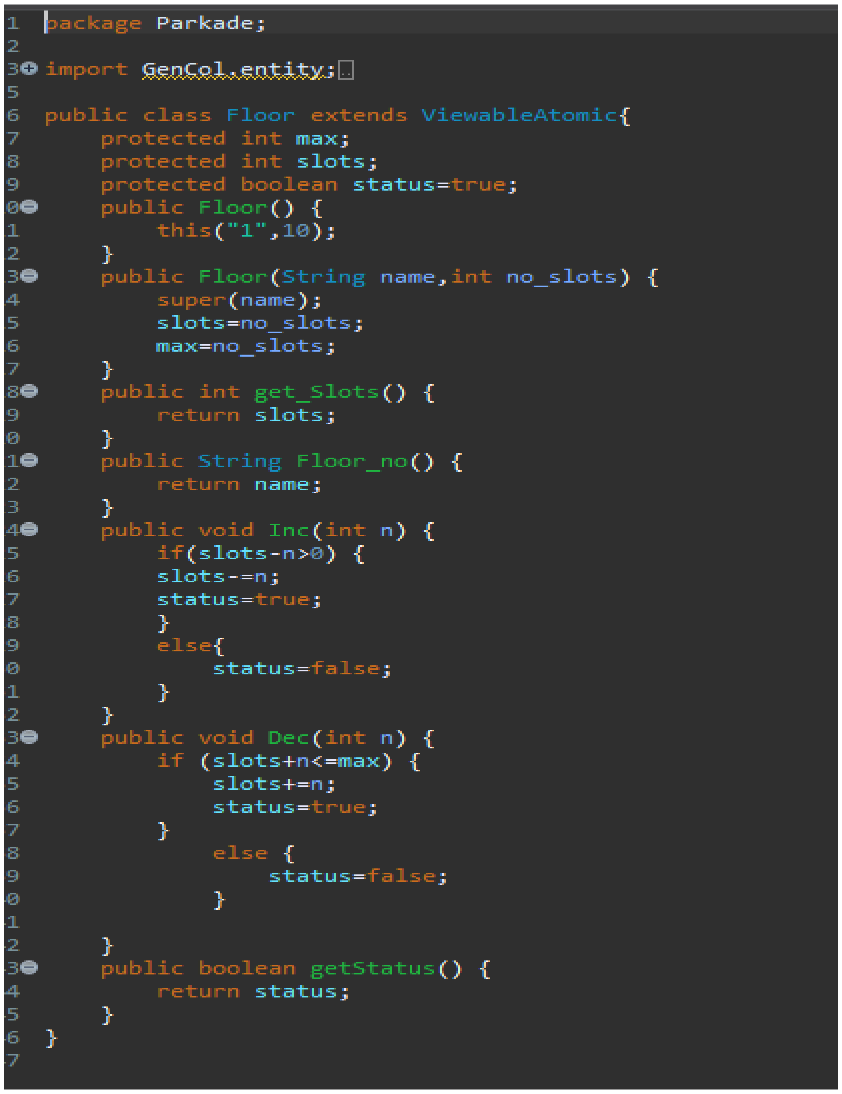


```java
    public void  deltint( )
    {
        if(phaseIs("active")){
            if(!q.isEmpty()) {
              currentJob = (entity) q.first();
              if(currentJob.toString().contains("vehicle")) {
              holdIn("active",carServingTime);
              }
              q.remove();
            }
            else
                   passivate();
          }
    }
    public message  out( )
    {
       message  m = new message();
       content con = makeContent("out",currentJob);
       m.add(con);
      return m;

    }
    public String getTooltipText(){
        if(currentJob!=null)
        return super.getTooltipText()+"\n number of cars in queue:"+q.size()+
        "\n my current job is:" + currentJob.toString();
        else return "initial value";
    }
}
```

Fig.2 Code snippet for parkingslotManager showing how the requests is added to the queue.

4.3 Floor

This class provides each floor that resides in the building as well as its attributes**.** Two methods are provided to increase and decrease the floor number based on the slot availability. This class sets the floor number, slot number and returns status as well.

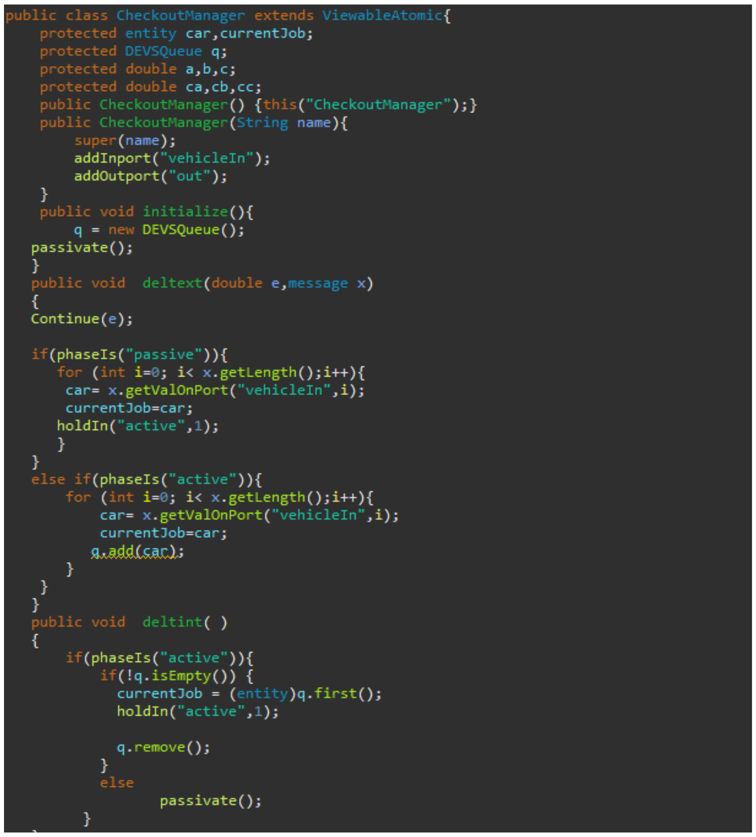


```java
package Parkade;

import GenCol.entity;

public class Floor extends ViewableAtomic{
    protected int max;
    protected int slots;
    protected boolean status=true;
    public Floor() {
        this("1",10);
    }
    public Floor(String name,int no_slots) {
        super(name);
        slots=no_slots;
        max=no_slots;
    }
    public int get_Slots() {
        return slots;
    }
    public String Floor_no() {
        return name;
    }
    public void Inc(int n) {
        if(slots-n>0) {
        slots-=n;
        status=true;
        }
        else{
            status=false;
        }
    }
    public void Dec(int n) {
        if (slots+n<=max) {
            slots+=n;
            status=true;
        }
            else {
                status=false;
            }

    }
    public boolean getStatus() {
        return status;
    }
}
```

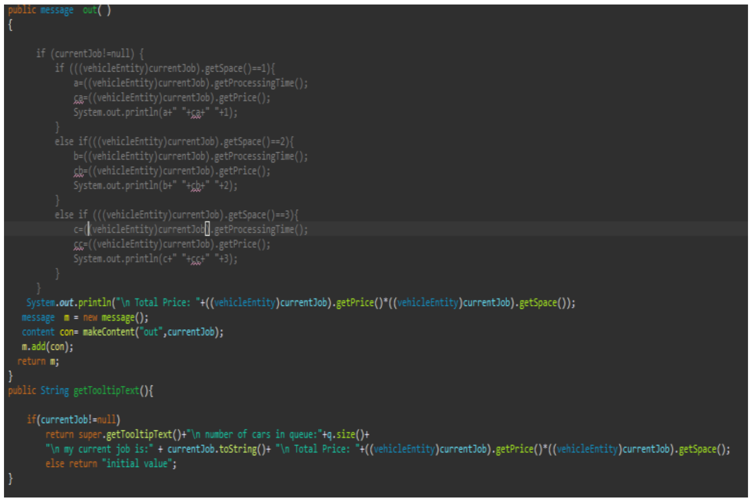
Fig.3 Code snippet for Floor class.

4.4 CheckoutManager

This class will fetch the vehicle attributes such as type of vehicle, number of slots it took, total time it spent and finally price for each time unit. Based on this data, this class provides the total amount that has to be paid by the vehicle before leaving the parking system. The commented code in the snippet is to gather experimental data and to analyze the results.



```java
public class CheckoutManager extends ViewableAtomic{
    protected entity car,currentJob;
    protected DEVSQueue q;
    protected double a,b,c;
    protected double ca,cb,cc;
    public CheckoutManager() {this("CheckoutManager");}
    public CheckoutManager(String name){
        super(name);
        addInport("vehicleIn");
        addOutport("out");
    }
    public void initialize(){
        q = new DEVSQueue();
   passivate();
   }
   public void  deltext(double e,message x)
   {
   Continue(e);

   if(phaseIs("passive")){
      for (int i=0; i< x.getLength();i++){
       car= x.getValOnPort("vehicleIn",i);
       currentJob=car;
      holdIn("active",1);
       }
   }
   else if(phaseIs("active")){
       for (int i=0; i< x.getLength();i++){
           car= x.getValOnPort("vehicleIn",i);
           currentJob=car;
          q.add(car);
       }
    }
   }
   public void  deltint( )
   {
       if(phaseIs("active")){
           if(!q.isEmpty()) {
             currentJob = (entity)q.first();
             holdIn("active",1);

             q.remove();
           }
           else
                  passivate();
         }
```

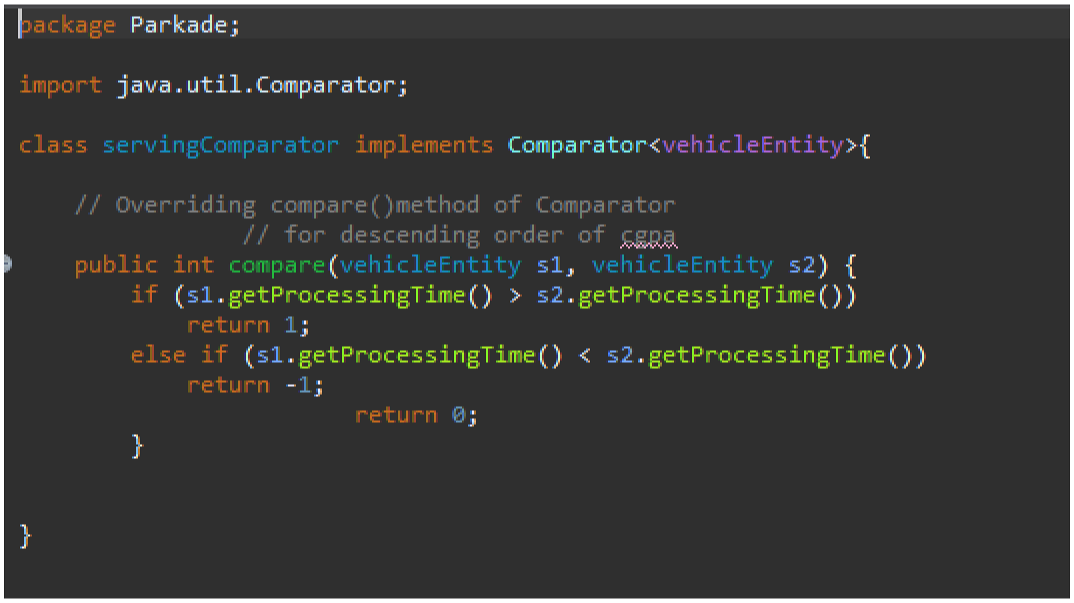


```java
public message  out( )
{

    if (currentJob!=null) {
        if (((vehicleEntity)currentJob).getSpace()==1){
            a=((vehicleEntity)currentJob).getProcessingTime();
            ca=((vehicleEntity)currentJob).getPrice();
            System.out.println(a+" "+ca+" "+1);
        }
        else if(((vehicleEntity)currentJob).getSpace()==2){
            b=((vehicleEntity)currentJob).getProcessingTime();
            cb=((vehicleEntity)currentJob).getPrice();
            System.out.println(b+" "+cb+" "+2);
        }
        else if (((vehicleEntity)currentJob).getSpace()==3){
            c=((vehicleEntity)currentJob).getProcessingTime();
            cc=((vehicleEntity)currentJob).getPrice();
            System.out.println(c+" "+cc+" "+3);
        }
    }
   System.out.println("\n Total Price: "+((vehicleEntity)currentJob).getPrice()*((vehicleEntity)currentJob).getSpace());
  message  m = new message();
  content con= makeContent("out",currentJob);
  m.add(con);
 return m;
}
public String getTooltipText(){

   if(currentJob!=null)
      return super.getTooltipText()+"\n number of cars in queue:"+q.size()+
      "\n my current job is:" + currentJob.toString()+ "\n Total Price: "+((vehicleEntity)currentJob).getPrice()*((vehicleEntity)currentJob).getSpace();
      else return "initial value";
}
```

Fig.4 Code snippet for CheckoutManager class

4.5 servingComparator

This class acts a comparator for the Priority Queue which we have used to process the vehicle times. It compares the processing time attribute between two vehicles.

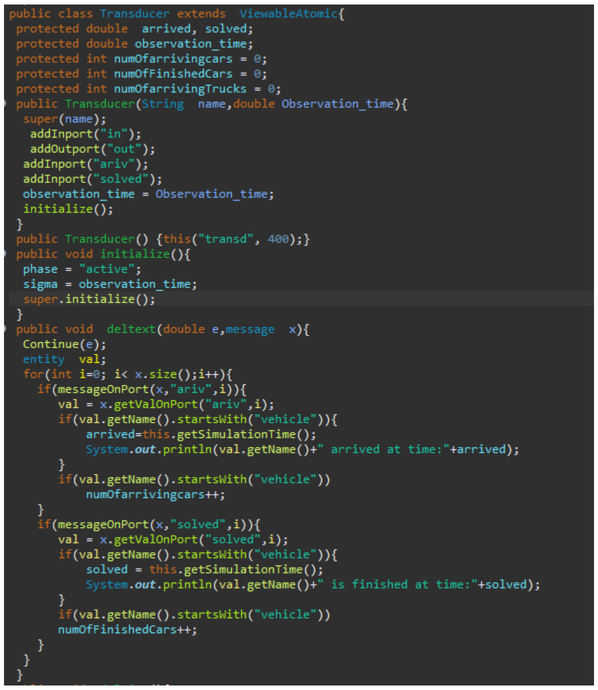


```
package Parkade;

import java.util.Comparator;

class servingComparator implements Comparator<vehicleEntity>{

    // Overriding compare()method of Comparator
                // for descending order of cgpa
    public int compare(vehicleEntity s1, vehicleEntity s2) {
        if (s1.getProcessingTime() > s2.getProcessingTime())
            return 1;
        else if (s1.getProcessingTime() < s2.getProcessingTime())
            return -1;
                        return 0;
        }


}
```

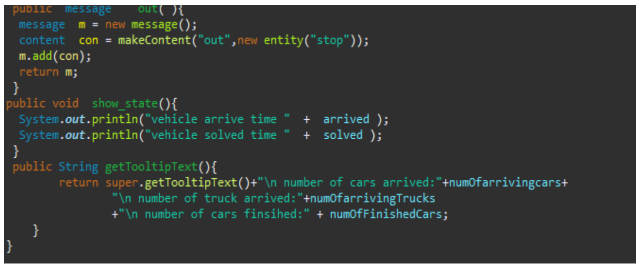
Fig5. ServingComparator class code snippet.

4.6 Transducer

This class acts as a bookkeeper to note all the vehicle arrival times as well as their solved time periods. Initially the sigma value for this class is 400 and once the sigma value becomes 0 it sends a signal to vehicleGenerator to stop sending vehicles. This indicates that the day is completed, and no more parking slots are available.



```
public class Transducer extends  ViewableAtomic{
 protected double  arrived, solved;
 protected double observation_time;
 protected int numOfarrivingcars = 0;
 protected int numOfFinishedCars = 0;
 protected int numOfarrivingTrucks = 0;
 public Transducer(String  name,double Observation_time){
  super(name);
   addInport("in");
   addOutport("out");
  addInport("ariv");
  addInport("solved");
  observation_time = Observation_time;
  initialize();
 }
 public Transducer() {this("transd", 400);}
 public void initialize(){
  phase = "active";
  sigma = observation_time;
  super.initialize();
 }
 public void  deltext(double e,message  x){
  Continue(e);
  entity  val;
  for(int i=0; i< x.size();i++){
    if(messageOnPort(x,"ariv",i)){
       val = x.getValOnPort("ariv",i);
       if(val.getName().startsWith("vehicle")){
           arrived=this.getSimulationTime();
           System.out.println(val.getName()+" arrived at time:"+arrived);
       }
       if(val.getName().startsWith("vehicle"))
           numOfarrivingcars++;
    }
    if(messageOnPort(x,"solved",i)){
       val = x.getValOnPort("solved",i);
       if(val.getName().startsWith("vehicle")){
           solved = this.getSimulationTime();
           System.out.println(val.getName()+" is finished at time:"+solved);
       }
       if(val.getName().startsWith("vehicle"))
       numOfFinishedCars++;
    }
  }
 }
```



```
 public  message    out( ){
  message  m = new message();
  content  con = makeContent("out",new entity("stop"));
  m.add(con);
  return m;
 }
public void  show_state(){
  System.out.println("vehicle arrive time "  +  arrived );
  System.out.println("vehicle solved time "  +  solved );
 }
 public String getTooltipText(){
       return super.getTooltipText()+"\n number of cars arrived:"+numOfarrivingcars+
               "\n number of truck arrived:"+numOfarrivingTrucks
               +"\n number of cars finsihed:" + numOfFinishedCars;
    }
}
```

Fig.6 Code snippet for Transducer

5. SIMULATION MODEL

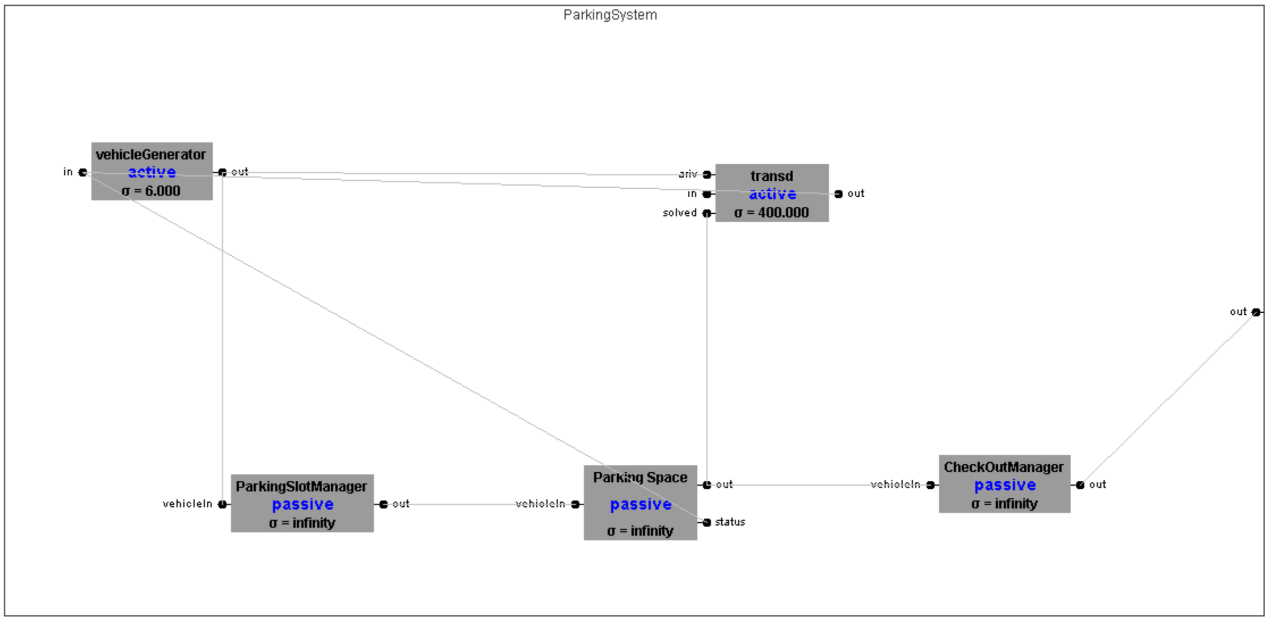


Fig.7 The Parkade Simulation model

6. RESULTS AND ANALYSIS

A correct queuing system is maintained to distribute parking spots to various cars by slotBookingSystem component. It also guarantees that all parking spaces are occupied by vehicles and that any further requests are processed by the vehicles arriving at the parking lot. A queue is also maintained at the parking exit counters to handle requests for the checkout procedure. This simulation model aids management in ensuring that no parking lot resource is wasted.

6.1 Challenges and Solutions

Several issues were encountered during the implementation of this model, including dealing with priority of the vehicles. Let us say if vehicle 1 has processing time of 30 units and vehicle 2 has processing time of 5 time units. We must process vehicle 2 first and the vehicle 1. But using DEVS Queue this cannot be achieved. So, to solve this problem, we have adopted a Priority queue. In which it uses a comparator every time when a new vehicle came in. Also, another issue is dealing with all the floors that has slots left whenever a vehicle leaves. To solve this we have written a function called addToFLoor() which makes sure that always filling is from them top floor.

6.2. Analysis

According to our initial assumptions for different days, in figure 11 our parking lot management system will receive different number of vehicles of each type. The day wise analysis of cars, trucks and executive cars are shown in the figure. Based on this estimation, we can assume the amount of profit that we receive at the end of the day as well as we can understand the traffic flow at the parking management system which helps in allocating the slots efficiently.

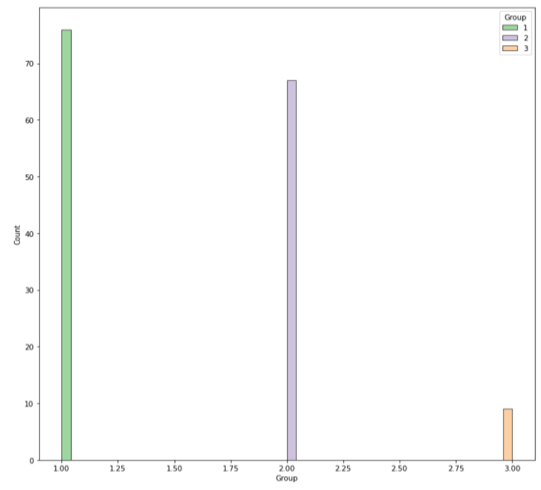


Fig.8 Bar graph showing the frequency of types of vehicles.

From the above graph, we can see that motorcycle frequency is higher than cars and truck frequency at a given point of time.

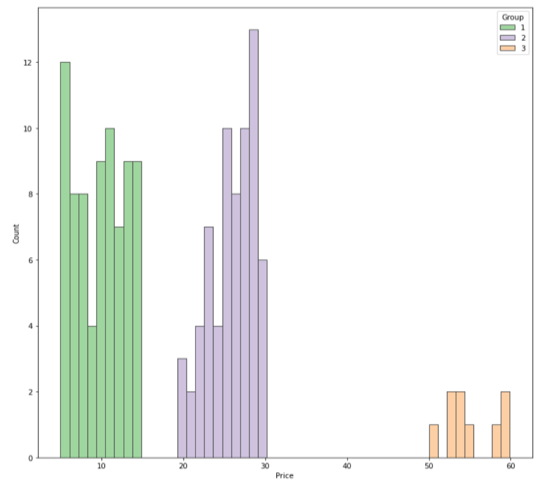


Fig.9 Frequency vs Price graph

From the above graph, we can say that price for trucks parking is higher than motorcycles.

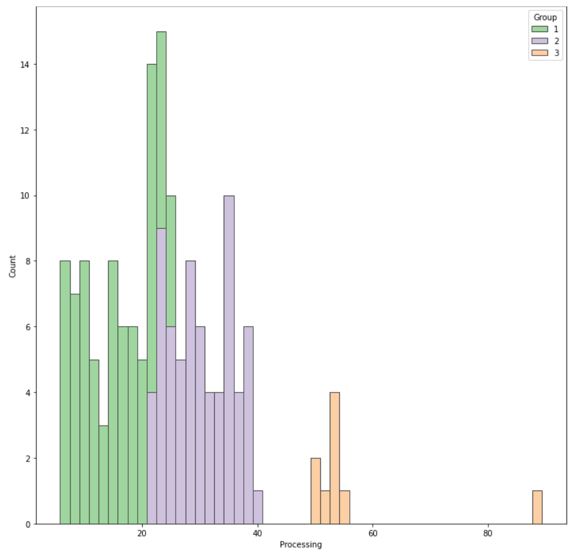


Fig.10 frequency vs Processing time graph

From the above graph, we can clearly say that processing time for both cars and motorcycles is almost same. But for the truck its is twice that of a car or a motorcycle.

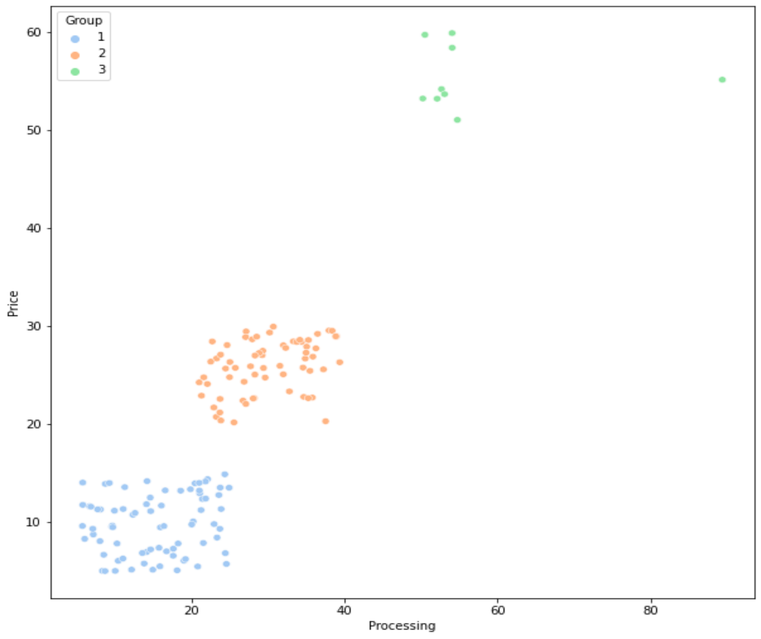


Fig11. Price vs Processing

From the scatter plot above the profit coming from trucks is huge even though the trucks count is low, but it takes more hours and more parking space than other two vehicle types.

7. CONCLUSION

The Parkade simulation model helps us with an economic approach to eliminate long waiting time to avail a parking spot by maintaining a queued system based on first come first serve approach. It also ensures that all the slots are occupied at any given time based on the arrival of vehicles. We also made sure to maintain a queued system at two parking exits to have a balanced system functionality. In this way, we attain an optimal stimulation model to maintain a multi-storey parking lot system efficiently.