Multi-story Carpark Report

Joel Adams (joa38)

Joa38
joa38@aber.ac.uk
StudentID: 180005309
April 28, 2019

Contents

1 Introduction

This document details the implementation of the multi-story car park application as outlined in the assignment. Please note that this folder also contains an external library called *JSON.simple* for working with *.JSON* files.

The following document consists of three sections;

Design This section encompasses the design, implementation and relationship of each of the classes

Testing Displays each function working correctly and a test table as specified in the brief

Evaluation More of a general description of how the assignment was solved. It also encompasses the use of flair in the assignment, difficulty, what remains to be done and what I think I should be awarded. This is also an opportunity to describe what I have leaned doing this assignment.

The assignment proved to be more complex than originally predicted but it meets all the requirements as specified in the brief

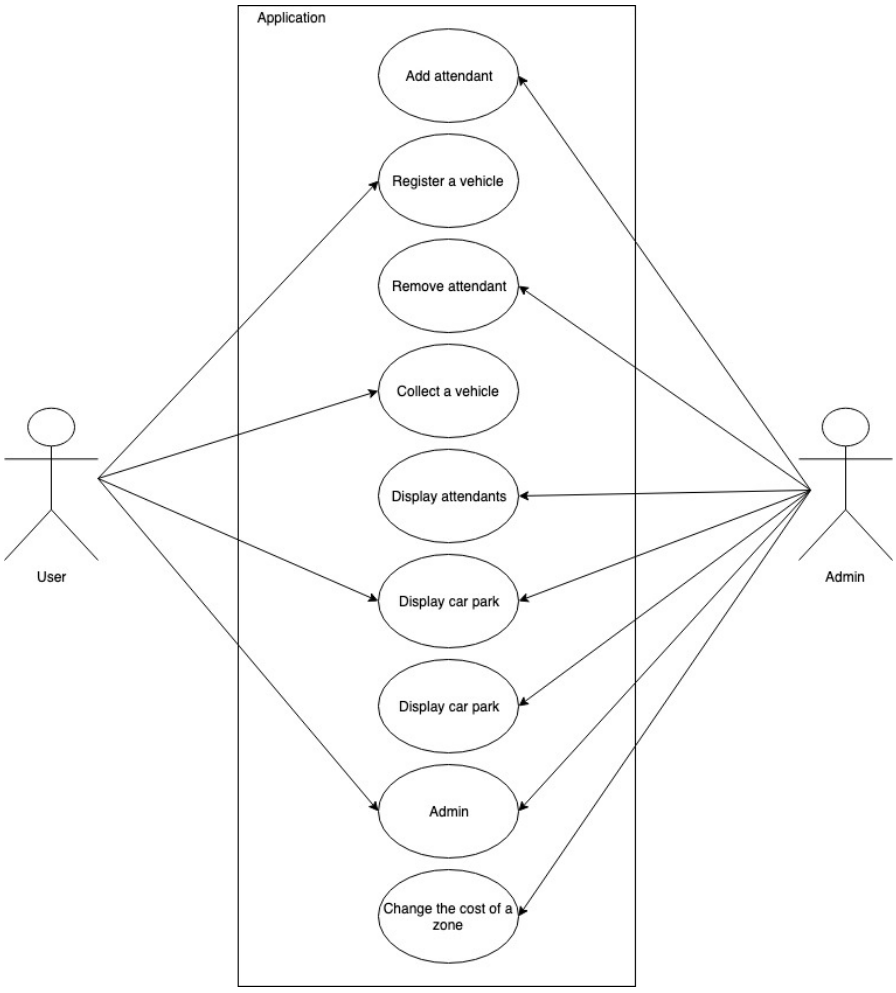

Figure 1: Use-case diagram

2 Design

2.1 Class design

Application is the **top level** class where the end user interacts with the different options available. They can either; *register a vehicle*, *collect a vehicle*, *show the car park* or *access the Admin class*. The class also calls the load methods for the **Attendants**, **ZoneCost** and **Database**.

The Application class has a relationship with the **Scanner**, **Admin** 2.1, **Zone Cost** 2.1, **Attendants** 2.1 and the **Database** 2.1.

Admin is another **top level** class that can only be accessed from the **Application** 2.1 if the password "*1234Admin*" is used. This stops a regular user from accessing these options and seeing all the data about each space in the car park. This class allows the admin to *Add/remove attendants*, *Display attendants / car park* or *change the cost of a zone*.

The Admin class has a relationship with the **Scanner**, **Zone Cost** 2.1, **Attendants** 2.1 and the **Database** 2.1.

Database relies heavily on the use of a package called *JSON.simple*. This allows the use of a *.JSON* file which has been a vast improvement over using a simple *.txt* file as it makes for a better function for loading the car park and saving it. The database is in charge of managing the car park and loading it into an **ArrayList** which makes the functions simpler as they only need to use a **for each** loop to iterate over it.

The Database class has a relationship with the **Scanner**, **ZoneCost** 2.1 and **Attendants** 2.1.

Attendants is a class that contains a **ArrayList** of the attendants that are loaded in from a *.JSON* file. I used a *.JSON* file as it would allow each attendant to be a separate object and to have a *name*, *space*, *zone* and a *boolean* that returns if the attendant is occupied

The Attendants class has a relationship with the **Admin** 2.1 and the **Database** 2.1.

ZoneCost stores and displays the different cost of each zone. I chose to store the data in a *.properties* file as it allowed me to have better control over what the user could do to the file. As seen there is no implementation of a *save* function as they change the cost of 1 zone at a time and that loads the file again using *changeZone()*.

The ZoneCost class has a relationship with the **Application** 2.1, **Admin** 2.1 and **Database** 2.1.

Vehicle is a simple class that consists of two instance variables for storing the *licence plate* and the *vehicle type*. When setting the type of vehicle a switch statement is used and it returns a string stating its type or "N/A", if the type isn't 1-5 or N/A then it returns a message stating that something went wrong. This is a very basic class and inheritance could be used in the future to fix it.

The Vehicle class has a relationship with the **Zone** 2.1.

Zone & zones This consists of a **abstract class** called **Zone**. It is in charge of storing the data about every space in the car park but is called inside each seperate zone e.g. *(Zone1() creates a new space in zone 1)*. This class is used as a template for different zones and is future proof as each new zone created relies on implementation of two *abstract methods* called *getZone()* and *checkVehicles()* *(To understand these methods please refer to the JavaDoc)*.

The **Zone** class is the parent class of; **Zone1**, **Zone2**, **Zone3**, **Zone4** and **Zone5**. It has a relationship with the **Database** 2.1.

2.2 Most complex algorithm

The most complex algorithm in this program is located in the **Database** class 2.1 and is called **checkParkingReceipt**.

Puseudo code:

```
public boolean checkParkingReceipt(String receipt) {
   boolean isReceiptCorrect = false;
   foreach (space) {
      if (receipt == s.getReceipt()) {
         setSpace = "N/A"
         // get the time in millis and get current time
         cost = costOfZone * timeInSpace;
         if (user == disabled) {
         cost / 2
         }
         boolean paid = false;
         while (!paid) {
            enter money
            if (cost == 0) {
               paid = true;
            }
         }
      }
      // make the attendant free again
      isReceiptCorrect = true;
      break;
   }
   return isReceiptCorrect;
}
```

2.3 UML Class Diagram


Attendants
- name: String
- AttendantRequired: boolean
- space: String
- zone: String
- attendants: ArrayList<Attendants>
+ Attendants()
+ Attendants(String)
+ Attendants(String, boolean, String, String)
+ isAttendantRequired(): boolean
+ setAttendantRequired(boolean): void
+ getName(): String
+ getSpace(): String
+ getZone(): String
+ setZone(String): void
+ setSpace(String): void
+ addAttendant(String): void
+ callAttendant(): String
+ setAttendant(String, String): void
+ makeAttendantFree(String, String): void
+ removeAttendant(String): void
+ load(): void
+ save(): void
+ printAttendantsAdmin(): void
+ printAttendants(): void
+ toString(): String
Application
- scan: Scanner
- admin: Admin
- zoneCost: ZoneCost
- attendants: Attendants
- database: Database
- Application()
- printMenu(): void
- runMenu(): void
- registerVehicle(): void
- CollectVehicle(): void
- printAll(): void
- admin(): void
- leaveCarPark(): void
- printVehicleType(): void
- attendantParking(): void
+ psvm()
Admin
- scan: Scanner
- attendant: Attendants
- zoneCost: ZoneCost
- database: Database
Admin(ZoneCost, Database, Attendants, Scanner)
+ runAdmin(): void
- printAdminMenu(): void
- addAttendant(): void
- removeAttendant(): void
- displayAttendants(): void
- displayCarPark(): void
- changeZone(): void
0..1
1..1
1..1
1..1
1..1
0..1
0..1
0..1
0..1<b>
0..1
0..1
ZoneCost
- pro: Properties
- zone: String
- cost: float
- zones: ArrayList<ZoneCost>
+ ZoneCost()
- ZoneCost(String, float)
getTimeCost(long, String): float
+ getCost(): float
+ getZone(): String
+ changeZone(String, String): void
- correctDecimalCost(double): float
- getTimeInHours(long): float
- getZoneCost(String): float
+ load(): void
pay(String, float): float
+ printZoneCost(): void
+ toString(): String
Database
- zoneCost: ZoneCost
- scan: Scanner
- attendants: Attendants
- spaces: ArrayList<Zone>
- zones: String
+ Database(ZoneCost, Attendants, Scanner)
+ save(): void
+ load(): void
+ searchForSpace(String, String): void
+ inputSpace(String, String, String, String): void
- setSpaces(Zone, String, String): void
+ checkParkingReceipt(String): boolean
- parkingReceipt(): String
+ toString(): String
+ printCarPark(): String
0..1
1..1
Vehicle
- licencePlate: String
- vehicleType: String
+ Vehicle()
+ Vehicle(String, String)
+ getVehicleType(): String
+ getVehicleTypeAsNumber(): String
+ getLicencePlate(): String
- setVehicleType(String): void
+ toString(): String
1..*
0..1
1..*
{abstract}
Zone
- parkingSpace: String
- spaceOccupied: boolean
- timeReserved: Calendar
- noTimeReserved: String
- parkingReceipt: String
- vehicle: Vehicle
Zone()
Zone(String, boolean, String, String, String, String)
+ getZone(): String
+ checkVehicles(String): boolean
+ getParkingSpace(): String
+ isSpaceOccupied(): boolean
+ getTimeReserved(): String
+ getTimeInMillis(): long
+ getParkingReceipt(): String
+ getVehicleType(): String
+ getLicencePlate(): String
+ setSpaceOccupied(String): void
+ setTimeReserved(String): void
+ setParkingReceipt(String): void
+ setVehicle(Vehicle): void
+ toString(): String
Zone1
+ Zone1()
+ Zone1(String, boolean, String, String,String,String)
+ getZone(): String
+ checkVehicles(String): boolean
1..*
Zone2
+ Zone2()
+ Zone2(String, boolean, String, String,String,String)
+ getZone(): String
+ checkVehicles(String): boolean
1..*
Zone3
+ Zone3()
+ Zone3(String, boolean, String, String,String,String)
+ getZone(): String
+ checkVehicles(String): boolean
1..*
Zone4
+ Zone4()
+ Zone4(String, boolean, String, String,String,String)
+ getZone(): String
+ checkVehicles(String): boolean
1..*
Zone5
+ Zone5()
+ Zone5(String, boolean, String, String,String,String)
+ getZone(): String
+ checkVehicles(String): boolean

3 Testing

3.1 Test table

For information on the requirements please go to the Test Table Key 3.1 on page 7, screen shots 3.3 on page 7

ID	Requirement	Description	Inputs	Expected outputs	Pass/ Fail	Comments
A1.1	FR1	Checking attendants can park only	1-3	N/A	P	
			4-5	Screen shot SS1 is displayed	P	
A1.2	FR1	Checking if the user has selected to park or attendants	y or Y	Screen shot SS2 is displayed	P	
			n or N	Screen shot SS3 is displayed	P	
			Other input	Screen shot SS4 is displayed	P	
A1.3	FR1	Attendants search for space or use system	S or R	Screen shot SS5 is displayed	P	
			Other input	Recursive call to function	P	
A2.1	FR2	Enter parking receipt	A parking receipt in car park	Requests user to pay for vehicle	P	
			A parking receipt not in car park	Tells user that parking receipt doesn't exist and recursive call	P	
A2.2	FR2	Check if user is disabled	D	Cost of vehicle cut in half	P	
			Other input	Screen shot SS6 is displayed	P	
A2.3	FR2	User pays for vehicle	Positive input e.g. 200 units	Screen shot SS7 is displayed	P	
			Negative input e.g. -100	Price increases by the amount	P	
A2.3	FR2	Leaving car park in 15mins	Presses a key before 15min	Screen shot SS8 is displayed	P	
			Presses key after 15min	Screen shot SS9 is displayed	P	

			Presses key on 15min	Screen shot SS8 is displayed	P	
A3.1	FR3	Display the car park	User displays car park	Screen shot SS10 is displayed	P	
A4.1	FR4	User enters password credentials	1234Admin	Enters the Admin class	P	
			Another password	Prints Invalid password	P	
A5.1	FR5	Add attendant	New attendant name	Adds attendant to pool of attendants	P	
A5.2	FR5	Remove attendant using their name	Correct name input e.g. Bob Ross	Screen shot SS11 is displayed	P	
			Incorrect name input	No attendant removed and nothing displayed	P	
A6.1	FR6	Display attendants in car par	N/A	Displays full details about attendants	P	
A6.2	FR6	Display everything in car park	N/A	Displays full information on car park, attendants and zone cost	P	The print statements from this can be very convoluted
A7.1	FR7	Change the cost of a zone	Enters correct name of zone e.g. Zone1	Screen shot SS12 is displayed	P	Can be very tricky to spell name correctly
			Enters incorrect name of zone e.g. Zon1	Screen shot SS13 is displayed	P	

Figure 2: Test table

Functional Requirement	Description
FR1	Registering of a vehicle
FR2	Collection of a vehicle
FR3	Display car park
FR4	Admin
FR5	add/remove Attendant
FR6	display car park/attendant
FR7	Change the cost of a zone

Table 1: Test table key

3.2 Discussion

In **A1.1** I used the inputs 1-3 and 4-5 as the attendants are only allowed to park the vehicle if the input is 1-3 (vehicle type) and they can't park the vehicle if it's 4-5 *(please see screen shot SS1 3)*.

In **A2.3** I also have the input *Presses a key on 15min* as there is a possiblity that the user ends up leaving the car park at exactly 15mins after paying for the parking ticket.

3.3 Screen shots

This section regards the **Test table** 3.1 on page 5.

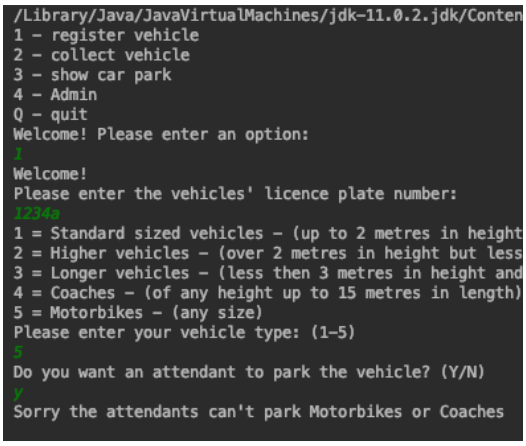

Figure 3: SS1

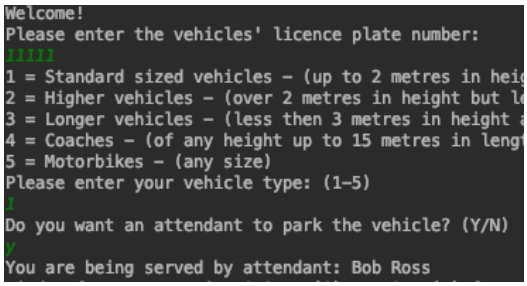

Figure 4: SS2

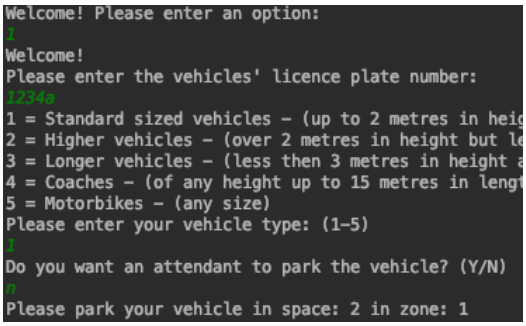

Figure 5: SS3

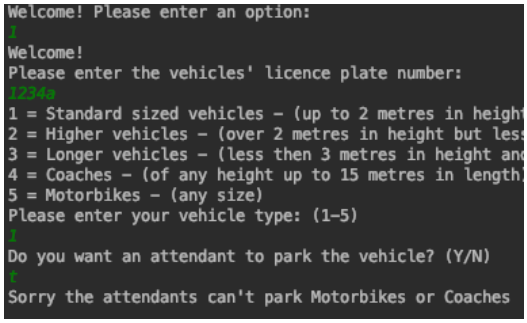

Figure 6: SS4

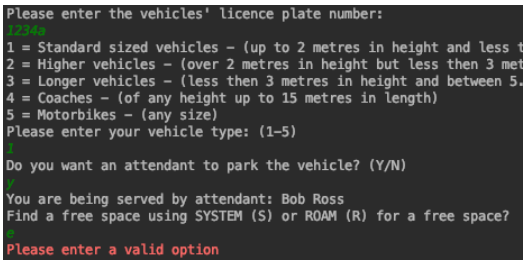

Figure 7: SS5

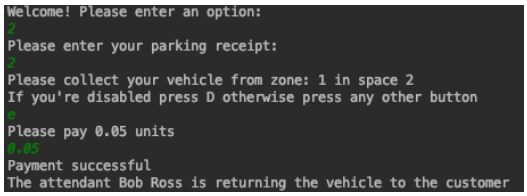

Figure 8: SS6

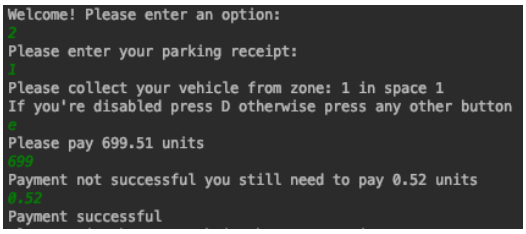

Figure 9: SS7

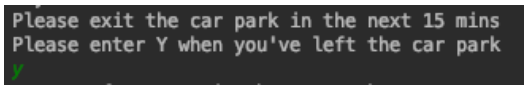

Figure 10: SS8

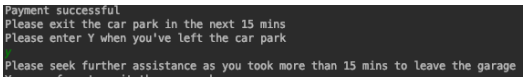

Figure 11: SS9

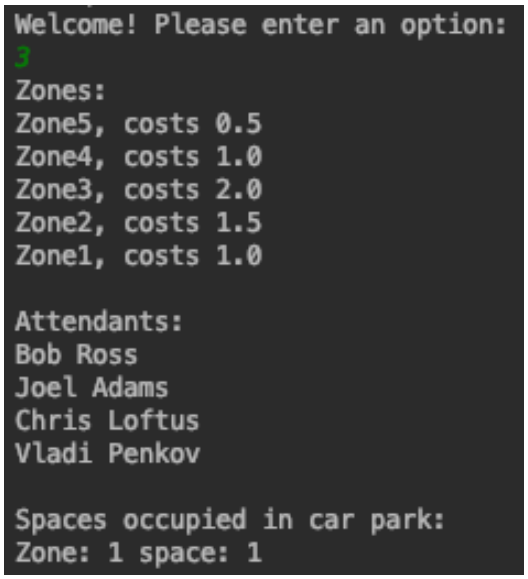

Figure 12: SS10

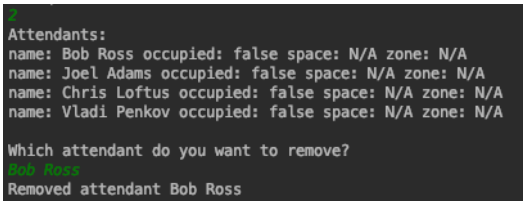

Figure 13: SS11

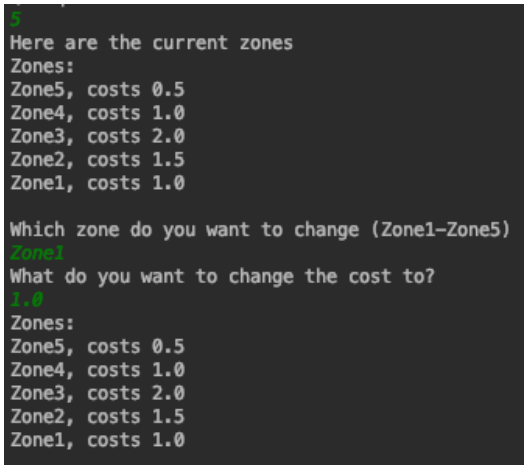

Figure 14: SS12

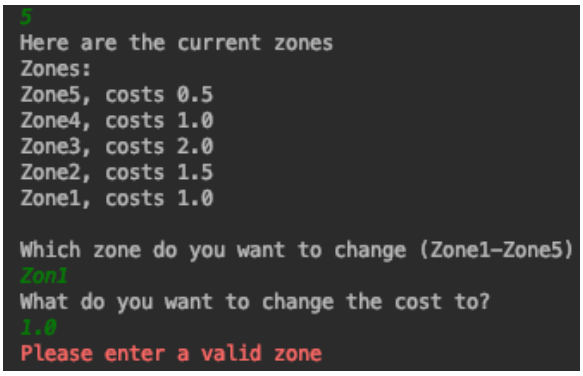

Figure 15: SS13

4 Evaluation

4.1 Solving the assignment

I began by thoroughly reading though the brief highlighting any key words or phrases. The key information was then used to create a checklist of funtional requirements and methods that needed to be implemented (seen below).

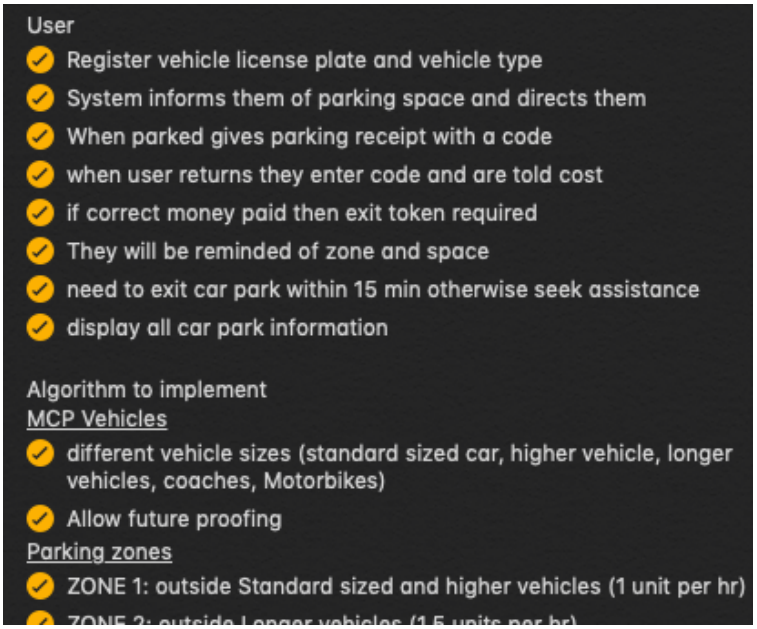

Figure 16: Check list of Functional Requirements

I then began to draw a UML diagram of the rough design of the program and started creating the classes. My original UML diagram has changed drastically over the course of the assignment due to implementation of abstract classes so please see my final UML diagram 2.3 for more information.

When I originally wrote the classes **Database** 2.1 and **Attendants** 2.1 I used plain text files to store the data which proved to be unreliable and easily corruptable so I opted for the use of *.JSON* file type as it proved to be more convinient and smarter. The **ZoneCost** 2.1 could have been stored in a *.JSON* file but it was more efficient to use a *.properties* file as it was easier to code and allowed for a *load()* method with polymorphism.

4.2 Flair

This assignment contains 2 things that I would classify as flair; *.JSON* files and *.properties* files. The use of these file types in the assignment allowed me to use polymorphism which substantially reduced the amount of code in the *.load()* methods of the classes.

4.3 Difficulty

The assignment originally proved to be quite difficult as the more I read into to the brief the more confused I became but this issue was easily resloved using the check list 16 and a UML diagram. I would have also wanted to use **JavaFX** but I have found it too confusing to implement in the short time I have left to work on the assignment. This has been more challenging than I had originally planned for but in hindsight it has been a useful learning experience.

4.4 What remains to be done?

I believe that I have satisfied the main functional requirements 3.1 of the assignment but there are some small bugs present in the programming. Sometimes when exiting a function back into the *switch* statement it takes an empty input but sadly I was unable to resolve this issue.

4.5 What did I learn?

I have learned how to use *.JSON* and *.properties* files which I have never learned about prior to this. This was done using an external package called *JSON.simple* and it was my first time implementing a library. I also used stackoverflow.com when I was lost trying to figure out how to implement anything.

During my last assignment I was unhappy with the final formatting of the report so I have written this entire assignment in **LaTeX** using **TeXstudio**. This is the first time I'm using LaTeX and it has proven to be a large challenge but I can now use it for future reports as I understand the basic commands.

4.6 What mark should I be awarded?

I believe that I should be awarded **75%** as I have provided documentation that follows the brief, my implementation of the general code is robust and the use of abstract classes for inheritance is good. The reason why I'm not giving myself a higher percentage is because I don't believe that I have achieved all the possible flair marks.