## Kevin Cadena, Sofia Garcia

## Date April 5, 2024

CS 3331 – Advanced Object-Oriented Programming – Term Spring 2024

## Instructor - Bhanukiran Gurijala, Ph.D.

## Project Part 1

# **Program Explanation**

In this program, we created the User, Person, Admin, Car, Sedan, SUV, Hatchback, Pickup, Log, RunShop, IssueTicket, loadAllCars, and CSVManager classes.

The Car class is the superclass, the Sedan, SUV, Hatchback, and Pickup all inherit from the Car class. The Log class oversees logging in user activity in the Log.txt file whenever called. The IssueTicket creates an IssueTicket object and is used to update the new issued\_tickets.csv file that keeps track of the cars ordered for each user. The loadAllCars oversees loading the Car objects instantiated to an array list for processing. The CSVManager class oversees reading and writing from the CSV’s as well as updating the whole CSV once the user exits.

We tackled a few problems by using the principles of object programming. First, in the Car class what we did is call the CSVManager to read the CSV file, and the instantiate Car objects for each car that was read. This allowed processing each to be much simpler, because then we can use the classes that inherit form the Car classes to get the attributes if needed. The same process was done with Person, and User. The IssueTicket was another problem. If the user logs off, and we only instantiate an array list in the system, that information will be lost if the program closes. Therefore, we made a new class IssueTicket that generates ticket objects that are placed in the issue\_ticket file, so that the user can access their tickets even after they log off.

The user gets 3 tries to log in. After that the system locks them out. We created a switch case statement to handle the main menu, and each time the user selects the main menu the action is logged using the Log.log command from the Log class. A couple of helper methods were created, which help with checking user login, printing all the cars, and checking to see if the user can purchase a car. When the user exists, the system calls the updateCSV() from the CSVManager in order to update the CSV with the new information that was stored temporarily by the array list. This process shows that the problems were broken down into smaller ones, each class having its own separate function so that the RunShop() does not get too long.

# **What did I learn?**

One of the most important things that we learned is breaking down a big component (or a requirement, problem) into smaller components. Very quickly we realized that implemented everything with only that classes that were asked was doable, but would make the RunShop() exceedingly large. This is why we chose to make more classes, that have only one responsibility, such as handling file processing, handling logging data, handling object creation, and handling tickets for the customer. This way we can break down the program to subsections and make it more readable and maintainable if new features want to be added.

The solution can be improved further by making a few more classes that oversee handling the array list conversions, and main menu operations. The main menu is a big, one mostly because it must handle user input and calls the appropriate functions and classes. Moving the main menu to a new class would make the RunShop() smaller and make things easier to refactor/update if needed. Also, as an optional feature we could make a GUI using Jframe for better user experience rather than text based.

Another idea that we thought about to solve the problem was make methods in the RunShop that call other classes and handle all the operations. These methods would only be called in the main method in the RunShop class so that readability is improved. This way if someone wants to edit our code, then they could just go the method that was called and trace it there, making refactoring easier and improving code maintainability.

We used the full two weeks to complete this lab. We honestly would have appreciated more time, but in 2 weeks we were able to finish the lab, although some optimizations could have been made with more time.

# **Solution Design**

In developing the program, a modular approach was used, with each class assigned a specific responsibility to enhance maintainability and ensure separation of duties. The primary data structure used was Array Lists, which was selected for their dynamic sizing. Array lists is used for managing users, cars, and transactions. The program was designed under key assumptions, including consistent CSV file formats being uniform to ensure that the CSVManager class works as intended. These assumptions allowed for a focused implementation of core functionalities like user logins, car displays, purchases, and logging actions.

# **Testing**

Each member tested the code separately to ensure that the code was tested thoroughly. Each element of the main menu was tested, included buying cars until the user cannot buy anymore and signing out. The CSV files were checked upon signing out to see if the parameters were appropriately updated. The issue\_ticket.csv file was also checked to see if the tickets were appropriately updated and handled from the code. Incorrect user input was input on purpose by each team member to ensure that the system handles the input correctly, by providing a warning message and looping back to the main menu or the corresponding menu appropriately.

For this testing, we primarily used black-box testing focusing on the user experience. This was determined because the end product is what the user sees, therefore we prioritized black-box testing. If the black-box testing shows that the menu system is unresponsive, locks the user out on certain operations or doesn’t work as intended, then we know that the code is bad, because the end product is what matters more. However, some white-box testing was performed to ensure that the internal codebase was working as correctly.

Our solution was tested enough times to ensure that each part of the menu system handles exceptions, input, and reading/writing to the CSV files correctly. The testing practices can be improved substantially by including white-box testing to examine if the logic of some of the functions work as intended, especially considered edge cases that might not happen often from user input.

The test cases used were simple.

* 1. The first test case used was going through each of the main menu options one by one with the correct input and signing out.
  2. The second test case was going through each of the menus but with the incorrect input to check if the system handles unexpected input correctly.
  3. The third case was failing to log in after 3 times to ensure that the log in system works as intended.
  4. The fourth case was buying a car until the car can’t be purchased anymore.
  5. The fifth case was buying a car without enough money.
  6. The sixth case was accessing the issued tickets.
  7. The seventh case was logging off, restarting the code, and accessing the tickets under the correct user.

The test cases showed that the system was robust in terms of handling incorrect user input, incorrect file inputs/file format, and performed well in the menu system. However, the testing did show that the CSVManager was not reading files appropriately as it was reading the first line in the file and returning error. The issue was that a string was being converted to an integer and falling under the number exception case. The solution was to skip the first line to ensure we don’t have incorrect type conversions.

# **Test results**

The following test cases were used:

1. The first test case used was going through each of the main menu options one by one with the correct input and signing out.
2. The second test case was going through each of the menus but with the incorrect input to check if the system handles unexpected input correctly.
3. The third case was failing to log in after 3 times to ensure that the log in system works as intended.
4. The fourth case was buying a car until the car can’t be purchased anymore.
5. The fifth case was buying a car without enough money.
6. The sixth case was accessing the issued tickets.
7. The seventh case was logging off, restarting the code, and accessing the tickets under the correct user.

Test 1 console output:

Username: neo

Password: L8GKCj$A

Welcome neo

Miner Cars main menu

Please input the corresponding number for the menu you wish to go to:

1. Display all cars.

2. Filter Cars (used / new)

1) New

(Print info)

2) Used

(Print info)

3) Go back

3. Purchase a car

4. View Tickets

5. Sign out

1

Displaying all cars...

//due to the large number of outputs this was omitted.

Miner Cars main menu

Please input the corresponding number for the menu you wish to go to:

1. Display all cars.

2. Filter Cars (used / new)

1) New

(Print info)

2) Used

(Print info)

3) Go back

3. Purchase a car

4. View Tickets

5. Sign out

2

Filter options:

1) New

2) Used

3) Go back

1

New Cars:

//due to the large number of new cars this was omitted.

Miner Cars main menu

Please input the corresponding number for the menu you wish to go to:

1. Display all cars.

2. Filter Cars (used / new)

1) New

(Print info)

2) Used

(Print info)

3) Go back

3. Purchase a car

4. View Tickets

5. Sign out

2

Filter options:

1) New

2) Used

3) Go back

2

Used Cars:

//due to the large number of used cars this was omitted.

Miner Cars main menu

Please input the corresponding number for the menu you wish to go to:

1. Display all cars.

2. Filter Cars (used / new)

1) New

(Print info)

2) Used

(Print info)

3) Go back

3. Purchase a car

4. View Tickets

5. Sign out

3

Enter ID of car:

1

Unable to purchase the car

Miner Cars main menu

Please input the corresponding number for the menu you wish to go to:

1. Display all cars.

2. Filter Cars (used / new)

1) New

(Print info)

2) Used

(Print info)

3) Go back

3. Purchase a car

4. View Tickets

5. Sign out

4

Viewing tickets...

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 2

Username: neo

Car Type: Sedan

Model: Toyota Camry

Year: 2024

Color: Silver

Car ID: 3

Username: neo

Car Type: SUV

Model: Toyota RAV4

Year: 2024

Color: Green

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 6

Username: neo

Car Type: SUV

Model: Ford Explorer

Year: 2024

Color: White

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 2

Username: neo

Car Type: Sedan

Model: Toyota Camry

Year: 2024

Color: Silver

Car ID: 3

Username: neo

Car Type: SUV

Model: Toyota RAV4

Year: 2024

Color: Green

Car ID: 7

Username: neo

Car Type: Pickup

Model: Ford F-150

Year: 2024

Color: Black

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Car ID: 1

Username: neo

Car Type: Hatchback

Model: Honda Fit

Year: 2024

Color: Yellow

Miner Cars main menu

Please input the corresponding number for the menu you wish to go to:

1. Display all cars.

2. Filter Cars (used / new)

1) New

(Print info)

2) Used

(Print info)

3) Go back

3. Purchase a car

4. View Tickets

5. Sign out

5

Signed out

Text document of test results:  
<https://docs.google.com/document/d/e/2PACX-1vQrTX6uhBuNXy9NZ7YEDq6hp5tuUOzRL4kJ8cB32RgtxqwKkzq_gNIGfvu932KZGwH7m6nMpuZdyvci/pub>

# **Code Review**

Code Review was performed separately by each team member. Each member was assigned half of the classes to review and provide feedback or comments to the other member as to how it could be improved or if the logic in the code is ok for final deliverable. In our team, each team member is assigned a code review section that they did not work on, to avoid reviewing code that was implemented by the member. Feedback was given through a quick meeting with notes attached.

For the code review checklist, each section was carefully analyzed to determine if it met expectations. The following was conducted for the Implementation section:

**Does this code do what it’s supposed to do?**  
Each class was reviewed to check if the returned statements matched what was needed for the requirements.

**Can it be simplified?**Each class was reviewed as to whether the logic can be simplified by removing dead code and refactored to methods to simplify the logic if necessary

**Is the code dynamic or hardcoded?**

Each class was reviewed to check whether variables were implemented to replace hardcoded values/integers

**Is the code maintainable?**  
Each class was reviewed to check whether the appropriate methods, and inherited attributes was properly implemented for maintainability. Furthermore, special emphasis was placed to make each class dynamic, so maintainability is easier if and update to the system must be made.

The following was conducted for the Logic section:  
  
**Cases where code does not behave as expected/intended?**The code was reviewed to check whether try {} and catch {} were successfully implemented where unexpected inputs can happen. This was intended to handle unexpected input from the user.

**Test cases where it may fail?**During code review each reviewer set test cases in the main menu to determine if the unexpected input would brick or crash the system.

The following was conducted for the Readability/Style:

**Easy to read/understand?**

During code review the reviewer was tasked to trace the code for understanding and change/simplify logic so that code because more readable. Javadoc comments were analyzed to determine if they convey the responsibility of the class.

**What parts can be modified or adjusted?**

During code review parts in the code that could be done in a class or method were noted and implemented later as a separate class or method. This process was done to ensure future maintainability of the system.

**Is the structure appropriate?**

The code was compared to the UML diagrams provided to ensure consistency in the structure.

**Does it follow the appropriate language style?**

The code reviewed was analyzed for the specific Javadoc comments and use of inheritance techniques to fall in line with the object programming design language.

**Is the code well documented?**

During code review, the generated Javadoc was analyzed to ensure that each method was well documented along with its attributes and parameters.

The following was conducted for Performance:

**What is the code complexity?**

The code complexity was analyzed during code review by checking for nested loop statements and other functions calls that required nested loops. The time complexity of any method or class was found to be greater than O(n^2) then the code would be refactored to reduce the time complexity back to the minimum baseline.

**How does the complexity change with various inputs?**

The code complexity increases linearly as the input grows, so it has a O(n) complexity. This was checked during code review to ensure that there were no nested loops that could lead to excessive computation times.