NORDIC MOTORHOME

Second Semester Project Report

Education: AP Computer Science — Københavns Erhvervsakademi DAT21i

Date: 2022/06/01

Teacher: Ronald Douglas Beaver

Students:

Octavian-Nicolae Roman — 2000/10/10

Jakub Mikolaj Patelski — 1997/08/23

Bartosz Sławomir Biryło — 2001/12/17

Abstract – by Octavian Roman

Our task was to devise a software system whose purpose is to satisfy the immediate administrative needs of a company named "Nordic Motorhome". The task has been undertaken using Iterative Development concepts from the Unified Process. We organized ourselves through Microsoft Azure DevOps and employed Git version control and GitHub to collaborate on source code. The system was built using Java for the backend, JavaFX and CSS for the frontend, and a Heroku-hosted MySQL "ClearDB" remote database for long-term data persistence.

Our inception phase was not atypical, consisting of business modelling and an attentive assessment of the requirements. During elaboration, we created numerous diagrams (e.g., Use Case diagram, Domain Model, SSD, etc.) and UI sketches that guided us through the construction phase. The construction phase involved the development of most of the source code. We implemented the Model-View-Controller (MVC) pattern, which facilitated the separation of logic from interface, structured our entire system, and provided a relatively low coupling and high cohesion of our classes. Going a step further, we separated the views themselves from aesthetics by means of stylesheets employing a custom flavour of CSS supported by JavaFX. The system itself has been built to support the four basic operations of persistent storage (CRUD) on real-world entities relevant to the business (rentals, motorhomes, etc). The system functionality is facilitated to the users by means of an intuitive, clutter-free, and somewhat responsive JavaFX graphical user interface. Lastly, during the transition phase, we thoroughly reviewed the system, did necessary adjustments, made a presentation, and wrote this very report. All these phases were accompanied by testing to obtain assessment of the quality of our work.

We are of the opinion that we fulfilled the goals proposed by the project. Using classes, methods, fields, control structures, iteration, data structures, and custom algorithms we created a GUI system that is versatile, dynamic, and reactive. Lastly, it should be noted that we went out of our way to surpass expectations by undertaking tasks and providing features which have not been covered yet in our education, such as password encryption or the creation of (semi-functional) executables.

Table of Contents

1. Introduction – by Octavian Roman

1.1 Purpose
The purpose of this report is to document the development of our Nordic Motorhome application. It intends to give a detailed explanation of said development during the main phases of the Unified Process.

1.2 Problem Statement
Our task is to create a software system that can handle information regarding the needs of a motorhome rental company based in Denmark. The system must be able to, at the very least, provide functionality to manage rentals, motorhomes, and customers. It is also imperative that we consider heuristic concepts such as usability guidelines or UI design rules. The solution is aimed towards the staff members of the company, who must be able to be authenticated into the system and use it with little to no assistance from third parties.

1.3 Supplementary Specification
The main non-functional requirements of the system are:

- The system should be intuitive and easy to use.
- To facilitate usage, a graphical user interface (GUI) must be implemented.
- The system should be reasonably accurate in its calculations and store data reliably.
- The user interface should be compatible with a reasonable amount of standard computer display sizes (responsiveness).
- The system should be capable of handling at the very least eight staff members, thirty-two motorhomes, eight different types of motorhomes, three seasons, and different brands, models, extras, and bed amounts.

1.4 Overview
This report will be structured in accordance with the main phases of the Unified Process, thus describing the project lifecycle in a structured manner. The main content, will, therefore, be divided into four sections:

- Inception
- Elaboration
- Construction
- Transition

2. Inception

2.1 Phase plan – by Bartosz Biryło

2.1.1 Inception phase

During this phase, we decided on our agile methodology, which was materialized as an Azure DevOps board/site that we set up and used for tracking our progress.

We divided our project's calendar into *Iterations* that each was focused on different field. We committed ourselves to follow these guidelines during the project's lifetime.

We have also prepared our git repository and decided to follow the Github flow philosophy of merging completed features into main branch.

2.1.2 Elaboration phase

At this point, we created sketches of our GUI and defined how we shall structure the architecture of our system.

This means both the code itself (compile-time, if you will) and how it will behave, and our users can interact with it(run-time).

We prepared various kinds of UML diagrams and descriptions to guide us through the development process.

2.1.3 Construction phase

Arguably the most important part of work. We implemented the architecture we agreed upon, with tests and Heroku deployment. We released every use-case separately, meaning that we focused first on making sure that the features for that use-case were perfected and well-tested.

2.1.4 Transition phase

We looked at our codebase and overall user experience in search of possible improvements. We did a small refactoring of our code to make it more readable and maintainable.

We also wrote the project report that would act as a description and an insight into How our work looked like, and what conclusions could be drawn from it.

2.2 Business Analysis – by Jakub Patelski

2.2.1 Swot analysis

By carrying out of SWOT analysis, we can see the strengths, weaknesses, opportunities, and threats that affect the performance of an organization. A SWOT analysis looks at both the internal and external factors. The internal are strengths and weakness but external

opportunities and threats. Strengths are those points where a company has a competitive advance in comparison with its competitors, but weaknesses are those points where the company has a competitive disadvantage in comparison with its competitors. Weaknesses prevent an organization from performing at its maximum level. There are areas where the business needs to improve to remain competitive. Opportunities related to favorable external factors that can give an organization a competitive advantage. Threats refer to factors that have the potential to harm an organization and its performance.
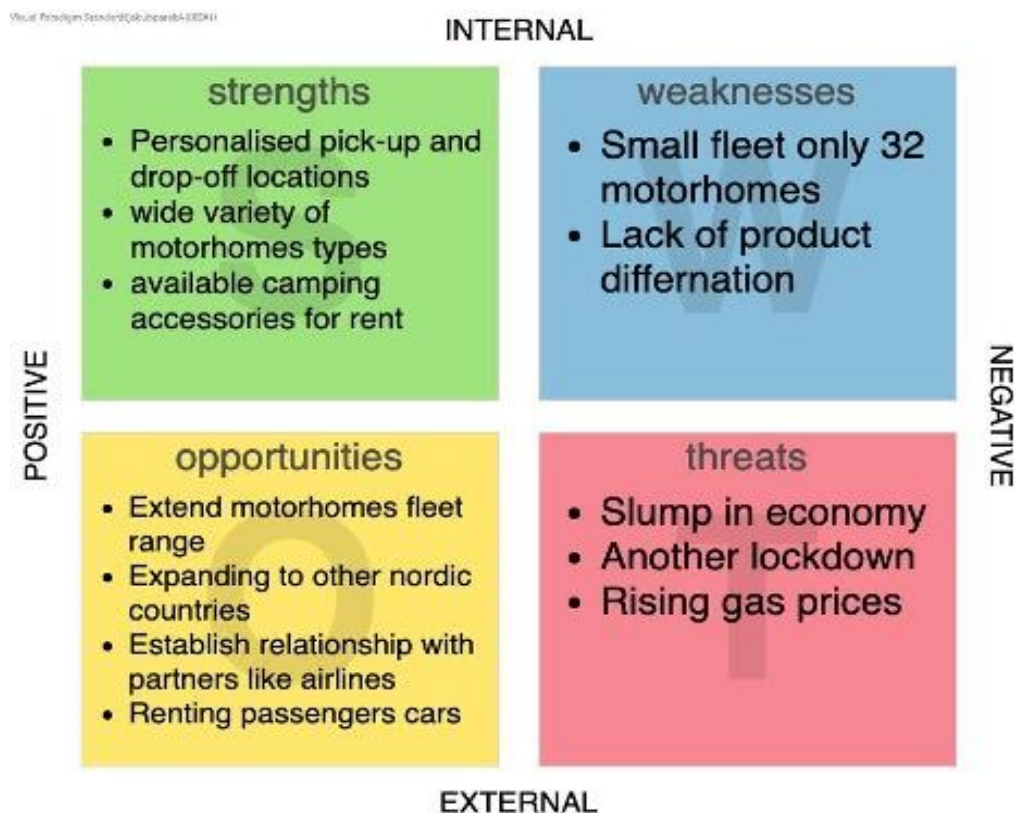
Figure 2.2.1 SWOT Analysis

2.2.1.1 Strengths
Main strength of the company is personalized pick-up and drop-off locations of its motorhomes throughout Denmark which means customers can get motorhome delivered even to their doorstep. Another strength is that the company possesses a wide variety of motorhome types that allow clients to choose between multiple bed options and brands of motorhomes. Last of the strengths is that customers can rent additional camping accessories like chairs, umbrellas, tables without buying them on their way.

2.2.1.2 Weaknesses
The weaknesses include a small fleet or motorhomes that is 32 which could be not sufficient during peak season. Another weakness is lack of product differentiation that during lockdown or season with bad weather could lead the company to go bankrupt as they do not have any other source of income, they only depend on renting motorhomes.

2.2.1.3 Opportunities

Extending motorhomes fleets seems to first natural to gain a competitive advantage as current fleet could be too small in peak season. If extending fleet will turn out to be successful company's next step to grow should be expanding business Internationally. First the company should focus on neighboring Scandinavian countries that share many similarities. [1] Also, company should focus on finding business partners related to travel industry that Nordic Motorhome could advertise its services for example airplanes. As mentioned, in weakness company does not have product differentiation they could add another product to their offer, it could be for example renting passenger cars.

2.2.1.4 Threats

After identifying strengths, weaknesses, and opportunities the last thing is threats.

The biggest threat for the company is another lockdown or travel restrictions that could affect a lower number of tourists and cause lower demand for renting motorhomes.

Other two factors are slump in economy and rising gasoline prices are related with each other. A significant rise in gasoline will raise the price of renting motorhomes and slump in the economy will occur in weaker purchasing power of people that with higher prices of renting a motorhome can lower people demand for deciding in renting motorhomes.

2.2.2 Economic feasibility study

2.2.2.1 Project Description

As our knowledge of the company is very limited and we don't have any financial statements we have decided to assume that currently the company has 50% of occupancy of their motorhomes and by our IT solution the company will be able to increase sales to 70% as they can build website on the staff version created by us.

Project name: Creating and developing new IT system for the company that will increase the occupancy of motorhomes from 50% to 70% thought a year.

2.2.2.2 Assumptions of current revenue and future after the investments.

Estimated costs- hiring 3 software developers full-time for a one-month contract. Average hourly rate for software developers in Denmark - 549 DKK.[2]

37 x 4 = 148 – 4 weeks per employee

3 X 148= 444 hours for all employees in total

444 X 549 = 243 756 DKK salaries of 3 developers

Fixed costs for creating IT system –243 756 DKK
Yearly assumption of revenue and return.

---

[1] https://www.hst.nl/en/scandinavian-market/
[2] https://www.salaryexpert.com/salary/job/software-developer-lead/denmark

Revenue before investment:

- 32 motorhomes
- 365(yearly days)
- 50% (estimated occupancy of each motorhome)
- 1100 average day price for motorhome

- 32 X 365 X 70% x 1100 = 6 424 000
- 6 424 000/ 32 = 200 750

- <u>6 424 000 DKK</u> - yearly total revenue
- <u>535 333 DKK</u> – monthly revenue
- <u>200 750 DKK</u> – yearly revenue by single motorhome
- <u>16 729 DKK</u> – monthly revenue by single motorhome

Revenue after investment:

- 32 motorhomes
- 365(yearly days)
- 70% (estimated occupancy of each motorhome)
- 1100 average day price for motorhome
- 32 X 365 X 70% x 1100 = 8 993 600
- 8 993 600 / 32 = 281 050
- <u>8 993 000 DKK</u> - yearly total revenue
- <u>749 417 DKK</u> – monthly revenue
- <u>281 050 DKK</u> – yearly revenue by single motorhome
- <u>23 420 DKK</u> – monthly revenue by single motorhome

Yearly Gross return on investment

8 993 000 - 6 424 000 = 2 569 000 DKK

Monthly Gross return on investment

2 569 000 / 12 = 214 000 DKK

2.2.2.3 Calculations

Fixed costs - 243 756 DKK

Current Revenue = 6 424 000 DKK

Expected Revenue = 8 993 000 DKK

Yearly expected Gross return on investment = 2 569 000 DKK

Monthly Gross return on investment = 214 000 DKK

NPV calculations

NPV = Today's value of the expected cash flows − Today's value of invested cash.

NPV = 2 569 000 - 243 756 = 2 325 244

Overall NPV = 2 325 244 DKK

Calculations of Return on Investment (ROI)

ROI = Overall NPV/ NPV of all costs (fixed costs in this scenario)

ROI = 2 325 244/243 756

ROI = 9.54

Yearly NPV CASH FLOW for 3 next years from investment

Year	1	2	3
Pv Cost	-243 756	0	
Pv Beefits	2569000	2569000	2569000
Yearly Npv cash flow	2 325 244	2569000	2569000
General NPV Cash Flow	2 325 244	4 894 244	7 463 244

Figure 2.2.2.3 Yearly NPV Cash Flow

Contribution margin ratio of the investment

CM ratio = Total Revenue- variable costs/ total revenue

CM ratio = = 2 569 000 – 243 756/2 569 000 = 0.95

Break event point

Break event point = Fixed costs / contribution margin ratio

Break event point = 243 756 / 0.95 = 256 585 DKK

2.2.2.4 Conclusion

The investment will cost the company 243 756 DKK while the expected yearly return on investment is expected to be 2 569 000 DKK. After calculations we will see that Net Present Value (NPV) will total 2 325 244 DKK which means that the investment is profitable as when NPV is higher than 0 the investment is considered to be profitable. The break event point is equal to 246 585 DKK that means the investment will start to be profitable when the company increase their revenue by 256 585 DKK. Every DKK company makes after that number is profit of the investment.

2.2.3 Risk Analysis

Project: Develop a new system for Nordic Motorhome

Risk type: Schedule risk

Priority (1 low … 5 critical): 4

Risk factor: Project completion will depend successful implementation software that can handle information about motorhomes, rentals and customers and enable to modify them. As the deadline is short the project can be delayed.

Probability: 50 %

Impact: Project competition will be delayed for each day that the software doesn't meet all the requirements.

Monitoring approaches: Schedule milestones reviews with developers

Contingency plan: Quickly hiring additional developer to the team.

Estimated resources: 40 000 DKK

2.2.3.1 Risk assessment matrix

	Risk	Probability[1-4], 4 highest	Impact = Loss[1-4], 4 highest	Risk or Prority = Probaility*impact (severity)
P01	Sickness of developer	2	4	8
P02	Poor time esitimation	1	2	2
P03	Software errors	1	3	3
P04	Hiring not well skilled developers	1	4	4
P05	Conflictis in the team	1	3	3

Figure 2.2.3.1 Risk Assessment Matrix

2.2.3.2 Risk Impact vs probability matrix

Impact vs Probability Matrix						
Probability		Threats				
Very High 4			PO4	PO1		
High 3						
Moderate 2			PO2			
Low 1					PO3, PO5	
Very Low 0						
Impact		Very Low 0	Low 1	Moderate 2	High 3	High 4

High Risk	Score >=6	
Moderate Risk	2 <= Score <6	
Low Risk	Score <2	

Figure 2.2.3.2 Risk Impact vs. Probability matrix

2.2.4   Stakeholder analysis

To effectively create a product and service that meets the needs of stakeholders and users, all stakeholders must be identified and included in Power/Interest Grid for Stakeholder Prioritization
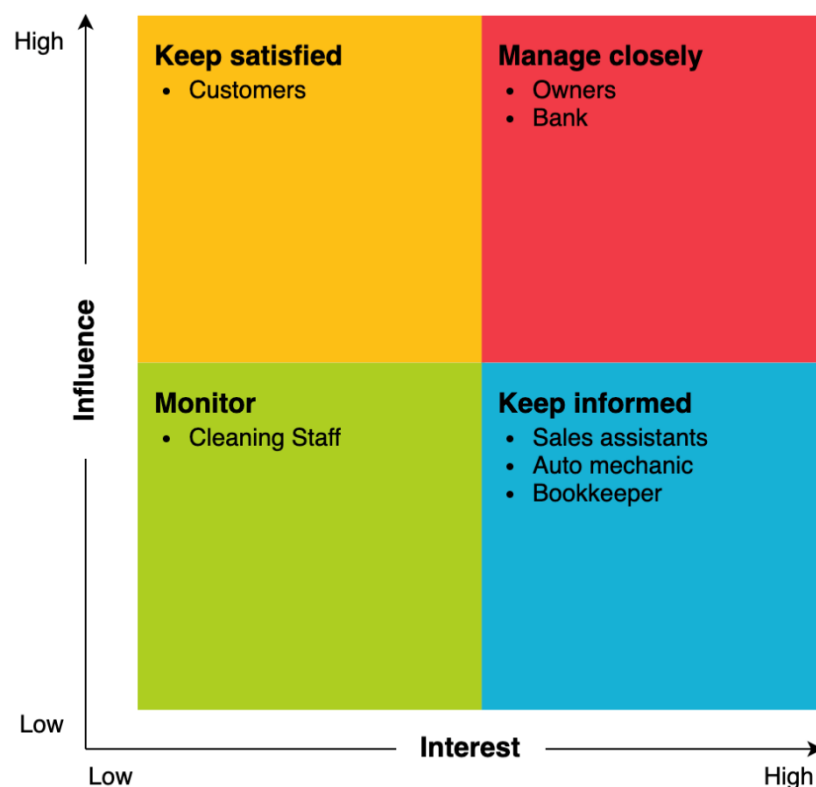

Figure 2.2.4.1 Power/Interest Grid for Stakeholder Prioritization

High power, high interest:

These are the company's most important stakeholders, and they should be prioritized. The greatest effort should be made to satisfy owners that invested their capital in the business as well as bank that needs to be ensured that the company has sufficient financial resources to pay debt.

High power, low interest:

To this group belong consumers of company's products that should be satisfied with rental to be willing to use the service again. They are low interest in the system itself.

Low power, high interest:

This group includes Sales assistants that are interested in customers' satisfaction for better sales. A bookkeeper should be informed about the financial situation of the company, its role is to collect transactions, track debits and maintain and monitor financial records. A mechanic's role is to repair all the broken motorhomes and should be informed about all the repairs in advance.

Low power, low interest:

Only cleaning staff are in this group that should be monitored but should not be involved in any issue.

3. Elaboration

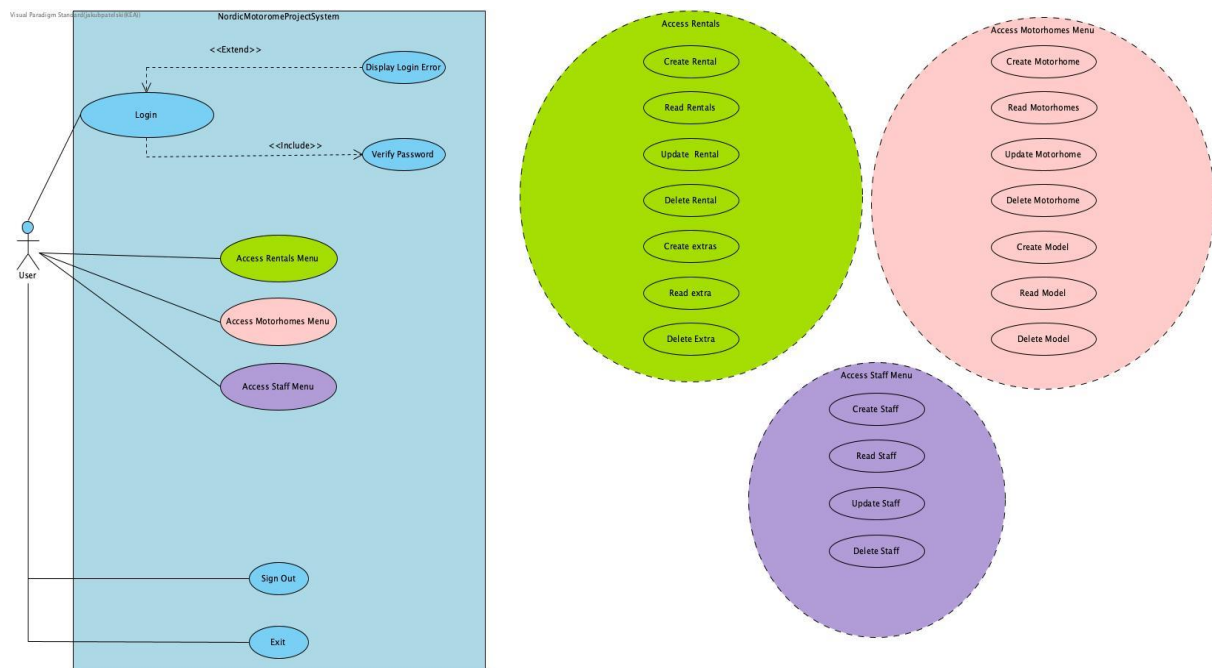## 3.1 Use Case Diagram – by Jakub Patelski

Figure 3.1.1 Use Case Diagram

The Use-case diagram describes the high-level functions and scope of a system. The diagram also identifies the interactions between the system and its actors. In our program we have one primary actor-user(admin) that can interact with all the use cases drawn in the diagram. The purpose of creating the diagram was identifying potential classes that the system requires and capturing all functions that the program should have. Our program has 3 main use-cases that imitate menu: staff, motorhomes, and rentals. All the actions take place inside the menus.
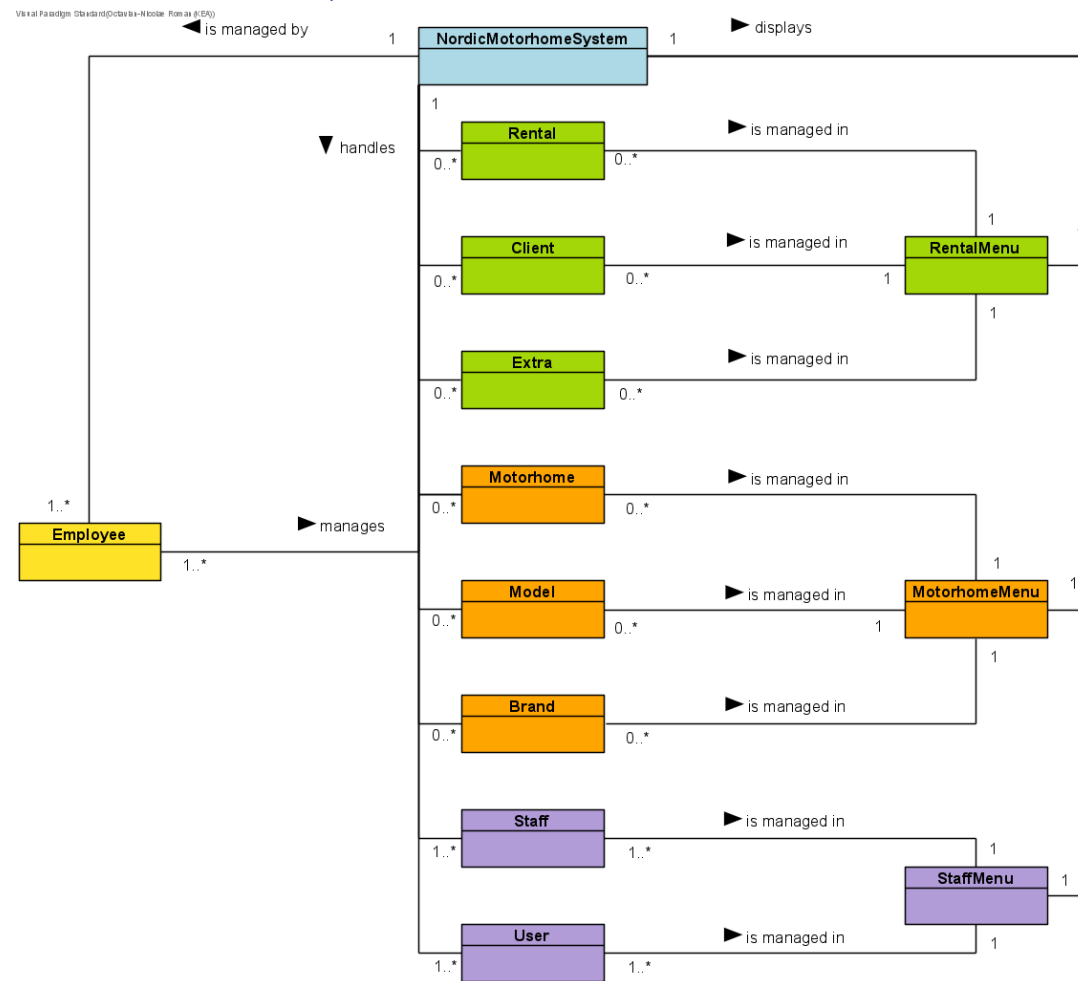
3.2 Domain Model – by Octavian Roman

Figure 3.2.1 Domain Model

This domain model is the *"visual representation of conceptual classes or real-world objects in a domain of interest" (Larman, 1997, p.128).* It is perhaps interesting to mention that there must be at least one instance of "User" (and Staff by association) because otherwise authenticating into the system would not be possible.

The diagram was most notably used as reference for our model classes representing the objects which our application handles, as seen in the java package below:
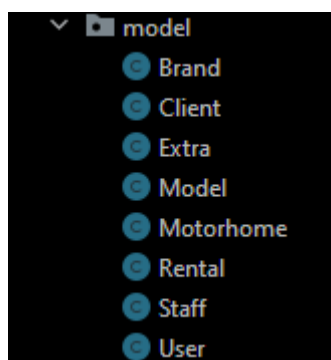

Figure 3.2.2 "model" package containing classes responsible for object modelling

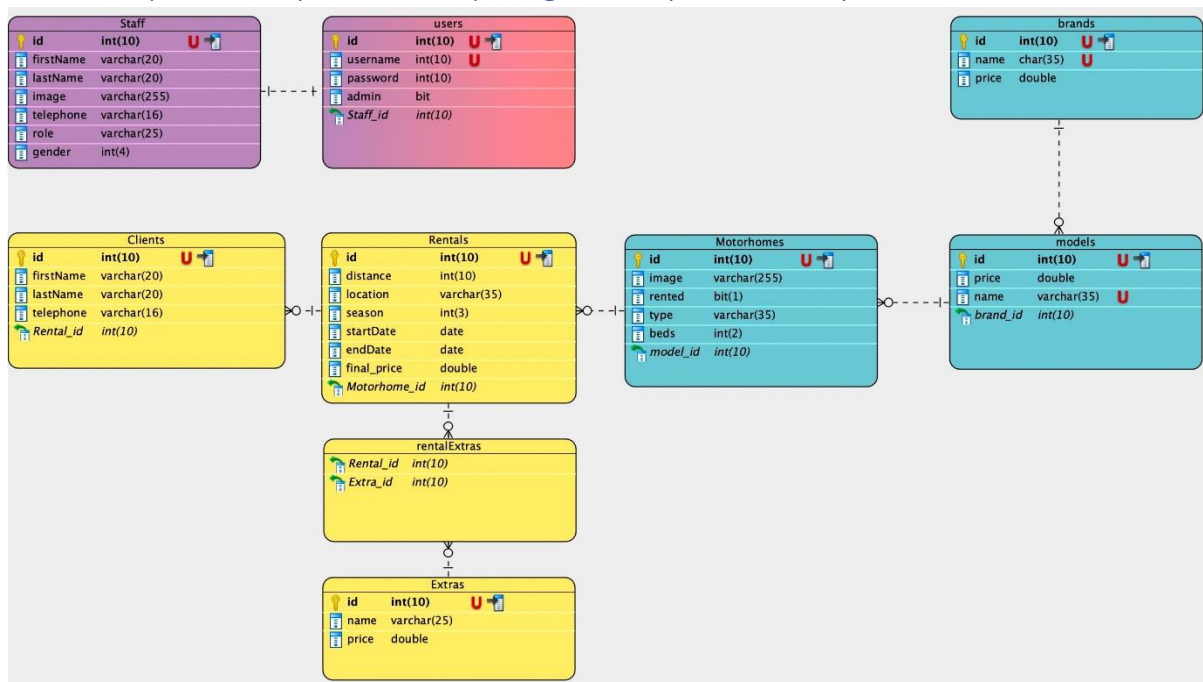## 3.3 Physical Entity Relationship Diagram – by Bartosz Biryło

Figure 3.3.1 ER Diagram. Visual Paradigm doesn't provide Enum or Boolean types so int([number of options]) and bit are used instead for gender, admin, season, rented columns.

 The physical entity relationship diagram, as its name suggests, describes relationships between different entities – concepts that can descripted using parameters or fields. Physical ER diagram differs from its logical counterpart in that it must show relational-database implementation of its entities. In our case each entity represents either a real-life object (one example being *motorhome*), or a service our business provides for either the outside clients (*Rentals* being such an example) or our own stakeholders (*Users*).

Our physical entity relationship diagram depicts the state of company's current relational database. Each entity (or table if you will) is color-coded by its dedicated use-case – yellow being managing rentals, turquoise motorhome management, purple staff management and lastly red for authentication purposes. *Users* is a somewhat special case, as it's both purple and red, meaning that it is utilized by both use-cases. Each table, apart from one, has its own unique key column. The reason why *rentalExtras* is different is because it's a *junction table*, meaning that it makes it possible to create a one-to-many relationship between *Rentals* and *Extras*.

The DDL script responsible for generating our database has been designed using this diagram as a model. *(See Appendix B for script)*

3.4 System Sequence Diagrams – by Jakub Patelski

A system sequence diagram (SSD) is a diagram that depicts the events that external actors generate, their order. An SSD shows system events for a scenario of a use case, therefore it is generated from inspection of a use case. Time proceeds downward, and events should be ordered in the same sequence as they are in the use case.
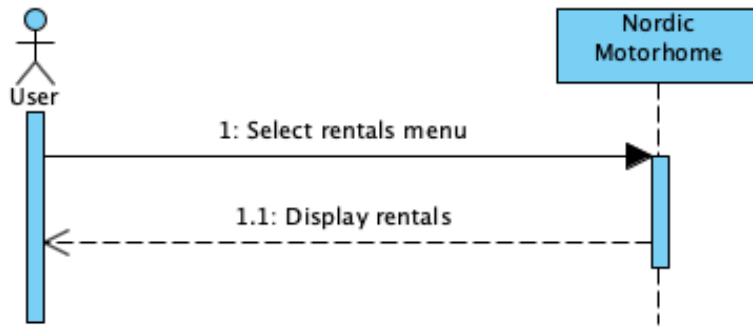
Figure 3.4.1 Access Menu rental System Sequence Diagram

This is the success scenario of Access Rental Menu. When user selects rental menu and click on it the system fetches all the rentals that have been created and displays them in GUI.

Figure 3.4.2 Add Rental System Sequence Diagram

In Add Rental first user selects add rentals button and click on rentals and click on it afterwards pop-up rentals window appears on the screen. Users enter rentals details field and press submit button, it creates a new rental that system send to database and by fetching it is again displayed to the GUI.
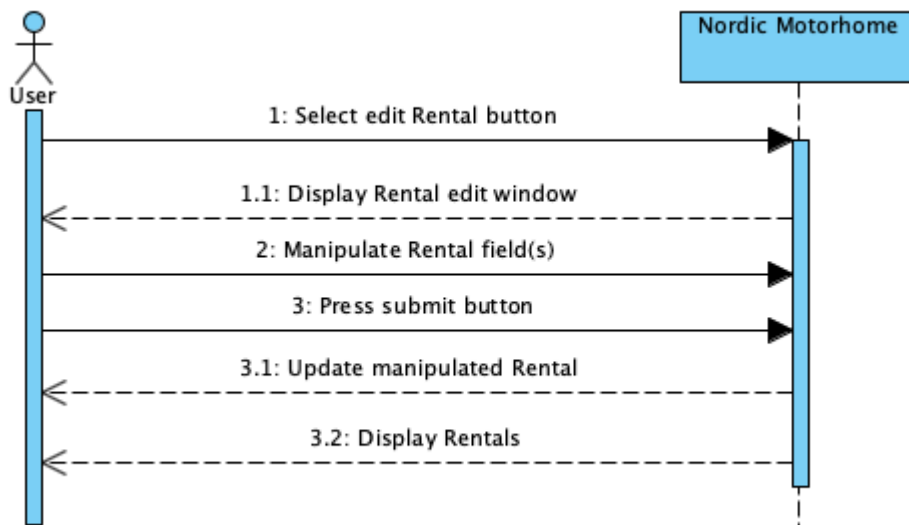
Figure 3.4.3Edit Rental System Sequence Diagram

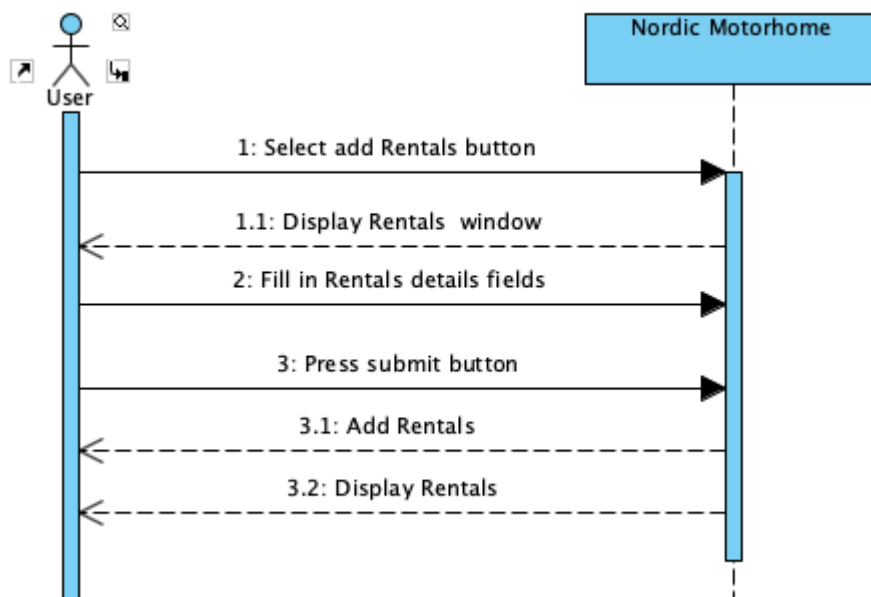In edit Rental user selects and click edit rental button then pop-up rental edit window displays with filled in fields with data stored in given rental. User manipulates some data and clicks on submit button. New data is sent to the database and current information is fetched and displayed to the GUI.

3.5 State Diagram – by Bartosz Biryło

Figure 3.5.1 State Diagram

A State Diagram, or as known in UML jargon, *UML Statechart*, models the behavior of a system by describing states of said system. By reducing the runtime architecture to finite

number of states it greatly decreases cognitive load (or our fancied term, *mental overhead*) required to understand the inner workings of the system.

In our case, we can differentiate three main areas states contribute to. Those aren't necessarily use-cases, rather abstractions that can help in guiding through our somewhat comp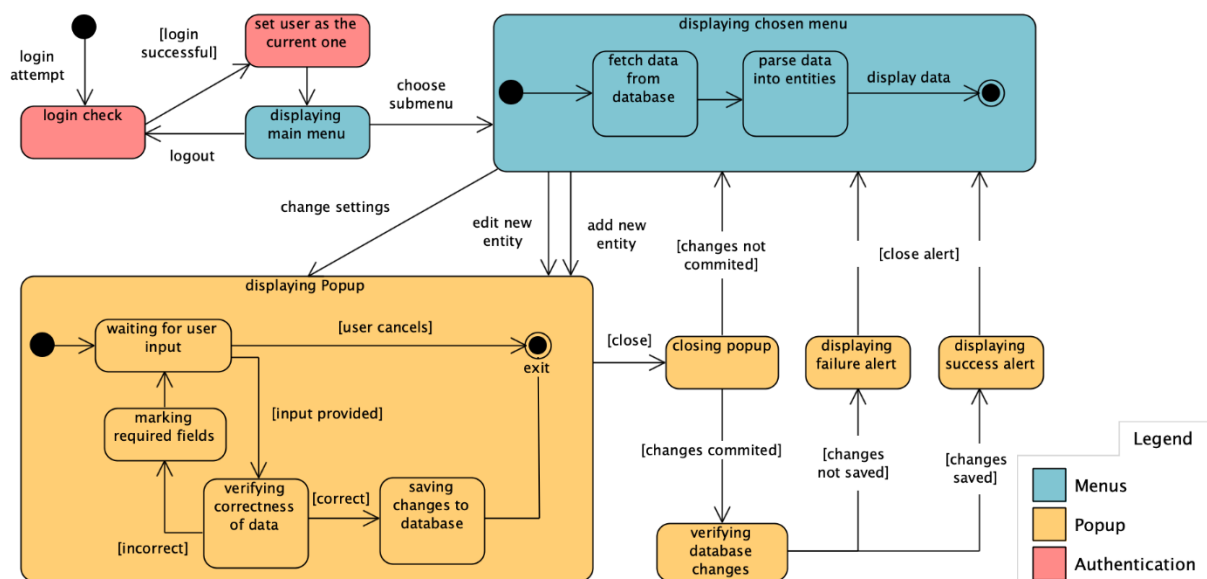lex system. Regardless of what you want to achieve through the application, logging-in is required. This is the *Authentication* part, colored red. Afterwards, main menu is displayed from which you can select a submenu – the turquoise *Menus* portion describes each state of that process. Finally, each menu allows for manipulating entities by displaying separate popups that guide users through it – procedure explained by orange section with self-explaining name, *Popup*.
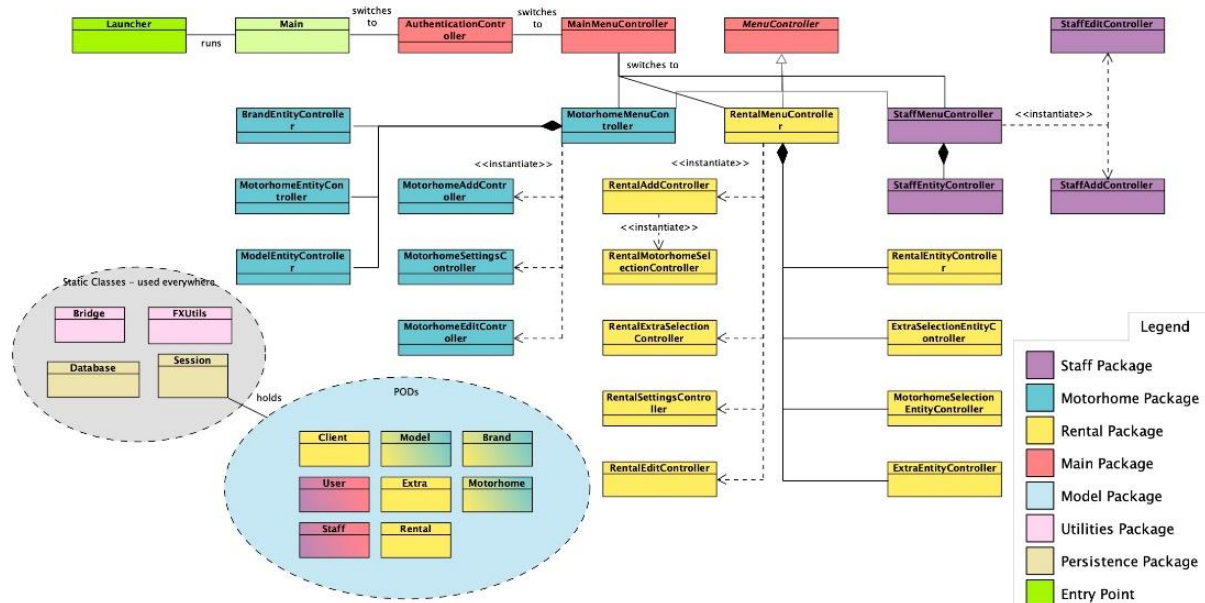
3.6 Design Class Diagram – by Bartosz Biryło

Figure 3.6.1 Design Class Diagram. As it would be practically impossible to display all classes' connections with their fields and methods, it is a simplified view that still retains the description of the system. For a diagram with all fields (connections really are complex) see Appendix.

A Design class diagram describes the code architecture of the system by depicting the layout of classes that constitute the overall codebase. Visualizing interactions and relations between classes in both compile- and run-time allows to reduce mental strain on developers and gives the ability to understand inner workings of the system without the need of delving into code implementations of classes and functions' definitions.

First, the color-coding of the diagram. Each *package* in our system has been assigned its own color to allow easier processing of provided information. Colors usually also correspond to the ones used for use-cases on some diagrams. The only exception to this rule is the green *Entry Point*, which rather than representing a package, marks the classes that program starts from. There are two classes colored this way, as due to the way JavaFX works, a second class, *Launcher*, is needed to run the proper *Main* class. The *Main* switches the application to use *AuthenticationController*, which as name suggests handles all logging-in and verification matters. This class and all the others after are using *PODs,* or *plain-old data*, a record-like objects, which definitions allow to only store data. As those objects are pure logic, meaning that they don't define *UI* in any way or directly communicate with a database, they are stored

in *Model* package, which is symbolized by light blue. Because PODs are used to transport data between our classes the diagram depicts all of them simply in their own unstructured category, colored in way that shows in what packages are they being used. They are displayed as only connected to *Session* class, since its main purpose is to hold them. This class is, similarly to PODs, bundled together with other classes, as they are used so frequently in our architecture it would otherwise make the diagram unreadable. With the main difference between them and PODs is that they are *static classes*, those are stored in two packages: pink-colored *Utilities*, which handles and provides JFX-related code, and khaki *Persistence*, which is responsible for storing the entities as either PODs or tables in a SQL database. Both of those packages are used in the previously mentioned *AuthenticationController*, which belongs to red-colored *Main* package, which hosts "core" classes, the others being *MainMenuController*, responsible for main menu, and *MenuController*, an *abstract class* that acts as a template for all other submenus.

Each use-case (excluding Authentication) is implemented by its own package with corresponding name: purple *Staff* package, turquoise *Motorhome*, and yellow *Rental*. Each of those packages has a menu controller inheriting from *MenuController*, that fetches needed data from database in form of PODs and stores *EntityControllers* created with said objects. Each of those packages also instantiates its own popup Controllers, used for adding, editing but sometimes also selecting Entities and even changing use-case specific settings.

3.7 GRASP Responsibilities – by Octavian Roman

When it comes to GRASP responsibilities, we have tried to take into consideration and implement the following three patterns:

- Controller

JavaFX facilitates the implementation of the Controller GRASP pattern by means of FXML views. FXML is a custom flavour of XML which enables us to create JavaFX views. Using this format, each view can be, and in our application is, associated with a controller class, which makes it possible to separate the UI layer from the business logic of the system. FXML is loaded as a hierarchy of objects by a JavaFX object called "FXMLLoader".

The controllers use annotations to access JavaFX elements, commonly known as "nodes". Controllers contain the logic required to handle the input system events occasioned by the nodes. Given that all views, and even some non-view (nodes that are not used for new stages) FXML files are associated with a given controller, we do not have a single bloated controller class that handles everything, but instead have twenty-four controller classes, each of which handles a very specific functionality inside of one of twenty-four FXML views. Here is our "controller" package, which has materialized due to us following the GRASP Controller pattern (alongside the MVC pattern):
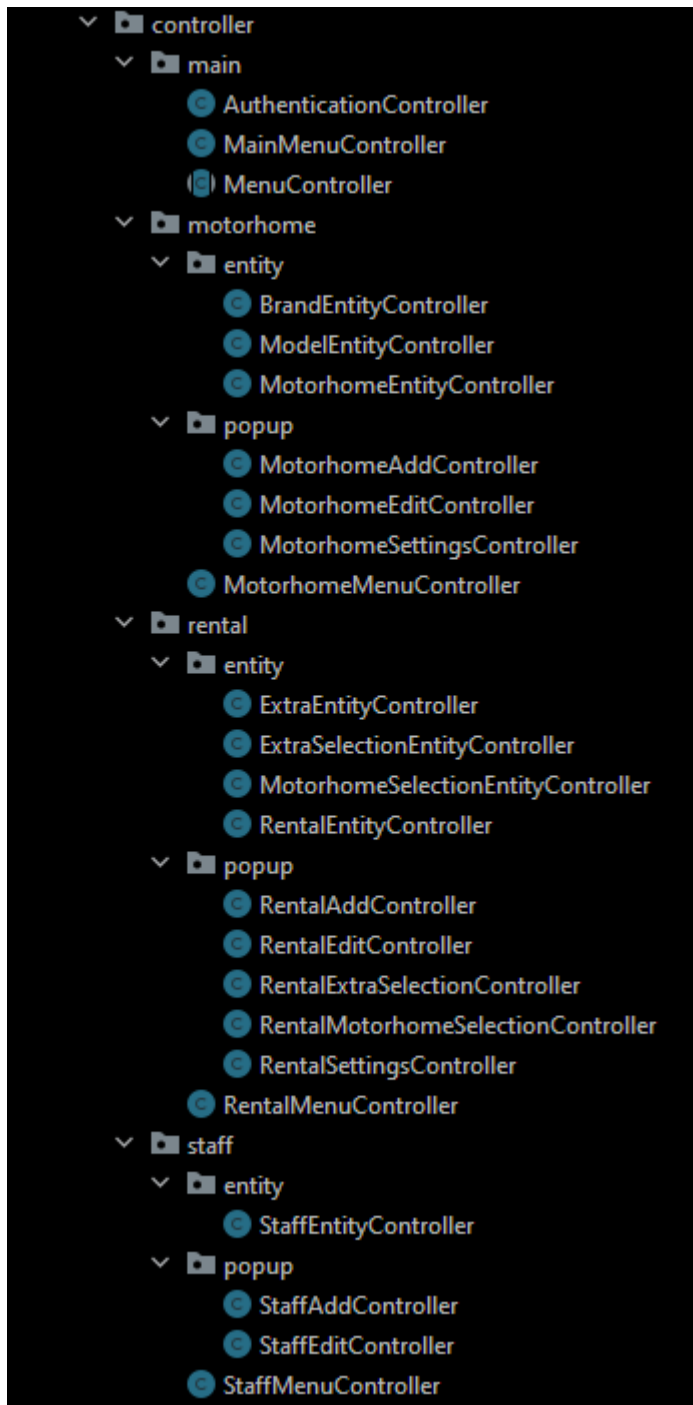
Figure 3.7.1 File structure of the "controller" package of our application

- Low coupling and high cohesion

"Coupling" can be defined as *"a measure of how strongly one element is connected to, has knowledge of, or relies on other elements" (Larman, 1997, p.229)*. Moreover, "cohesion" can be defined as *"a measure of how strongly related and focused the responsibilities of an element are" (Larman, 1997, p.232)*.

We attempt to achieve high cohesion by delegating responsibilities to many classes instead of few, ensuring that each class has a particular purpose instead of many. Thusly, our system makes use of thirty-eight classes subdivided as such:

Figure 3.7.2 Java classes with controller and model package folded as they appear in figure 3.7.1 and 3.2.2

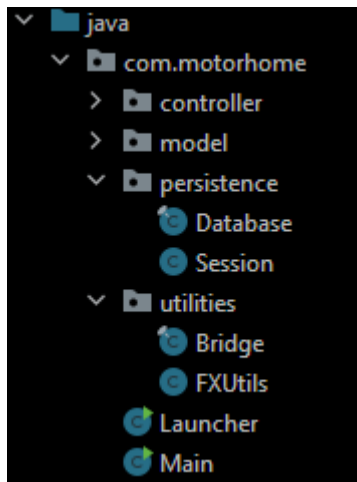We also attempt to achieve low coupling through responsibility assignment. For example, the "Database" class is responsible for connecting and disconnecting to and from our remote database. It does not rely on any other classes defined by us and focuses on that very specific purpose. Similarly, all controllers fulfil very specific responsibilities and tend not to rely on classes other than those required for utilities and persistence, albeit admittedly sometimes they trigger actions in other active controllers through the Bridge pattern.

3.8 GUI Sketches – by Octavian Roman

Lastly, it should be mentioned that we also created sketches of how we expected our graphical user interface to look after the construction was to be completed. This is not part of UML or the requirements of this project, but we have decided to nonetheless proceed to design said sketches because they are useful when it comes to counteracting decision-fatigue and they also greatly facilitate the process of creating views.

The six sketches were all created with Figma, a browser-based tool designed for this very purpose. Here is one of them:

Figure 3.8.1 Sketch of the motorhome menu during the inception phase

4. Construction

4.1 The setup – by Octavian Roman

4.1.1 Some background into JavaFX

JavaFX, (abbreviated JFX or FX) is a software platform designed to create desktop applications. Unfortunately, Oracle, the company behind the Java programming language, decided not to include JavaFX in the JDK after version 11. This entails that, for a client to run JFX, one cannot rely on the normal modern Java installation.

JFX is comprised of several packages or "modules" such as Base, Controls, Graphics, and so forth. Integrated Development Environments such as IntelliJ IDEA run JavaFX by modifying the JDK on which they execute the FX programs to include these modules as if they were still part of the mainstream Java distribution. This makes development easy and reliable, but it is rather ineffective for developing a program destined for end-users because it does not consider that most users do not run their applications in an IDE, nor do they have access to these modified JDKs or JREs. IntelliJ JavaFX projects (as initialized by the IDE) do not run if taken outside of the IDE, say, for example, in a Java artifact (JAR).

To make a JFX application run reliably as a standalone, self-contained application, one may try to somehow bundle these FX modules into the application. This is exactly what our setup achieves, although at the cost of pushing the FX platform to its limit and receiving a warning for running the application in an "unsupported" configuration on start-up. Nevertheless, the setup has been tested on multiple devices, and, unlike any other tested configuration, seems to reliably run provided that Java is installed.

To achieve this, we went out of our way to create our own custom setup. Not before trying official recommendations from the JavaFX website and numerous other sources, which were

unfruitful due to the advice offered therein being outdated, unavailable, or incomplete. It should be taken into consideration that this platform has received little support and/or attention ever since it has been dropped by Oracle, meaning that our configuration is definitely imperfect, but, nonetheless, it is the best we managed to come up with to date.

4.1.2 The setup itself

Instead of creating a "JavaFX" project in our IDE (which just so happens to be outdated and unsupported for artifact generation for anything but Java 8), we create a "Maven" project. Maven is a Java build automation tool that is going to handle the compilation of our source code. Maven builds the project in accordance with what is contained within a special XML file named "pom.xml". The setup revolves around said file, which, aside from what is to be expected, most notably includes:

- The JavaFX base, controls, FXML and graphics modules for all three major operating systems (MS Windows, macOS, Linux). With this, we are ensuring that JavaFX will have the required modules to run no matter the environment; be it a JAR, IDE or even packaged executable, FX should have the modules that it needs. This comes at a cost of a larger size in terms of memory, but these FX packages do not seem to exceed 10-20 MBs, which is inconsequential for modern machines.
- The Jupiter API and Engine for Unit Testing.
- The MySQL JDBC connector to be able to query our remote database.
- A packaging plugin named "Apache maven shade plugin". This plugin handles the JavaFX modules by removing their modularity and instead adding their contents to the classes/resources instead, essentially creating an uber-JAR. This is needed to make JavaFX run somewhat reliably in an executable format.

With all of that, we can run the application in our IDEs as an "Application" configuration, in a JAR by executing them through the terminal, and packaging applications such as "Launch4J" can be used in conjunction with aforementioned JARs to create even executables that seem to run reliably compared to any other setup known to us.

4.2 Database design – by Bartosz Biryło

As the main goal of this application is to manipulate on saved-by-user data, whatever the shape of data may be, it requires a way to persistently store it in such a way user would not lose his changes once committed. For this exact reason our application utilizes a connected SQL database to safely store data even after the app has been terminated. For the ease of deployment and various benefits of cloud storage the app currently connects to a *"ClearDB"* database deployed on a Heroku server. This allows for multiple users concurrently accessing the needed data.

The way application connects to the SQL server is straightforward. The Database class is a rather simple class compared to other parts of the application, but it is not a flaw but rather a deliberate attempt – this allows us to always have direct control over both the queries sent and results received from and to the SQL database.

The data in the database is naturally at stage 3 of normalization, since keeping the schema "clean" allows for easier data management and reduces data itself. But what really makes our schema unique compared to typical database is not the already described in ER diagram layout, but rather the fact that we use AES encryption to symmetrically encrypt users'

passwords to securely store them in our database. This ensures that passwords are safe even in case of emergency such as a data leak.

4.3 Authentication – by Octavian Roman

The entry point of the application is the class "Launcher". It's only responsible for calling onto the main method of the "Main" class. This entry point may seem redundant, but because of the inner workings of JavaFX, the Launcher class is required to be able to launch the application in a JAR or executable format.

Main inherits from the built-in "Application" class in FX. Upon calling the actual main method, the overridden "start" method brings the first view of the application into existence: "authentication", whose controller is "AuthenticationController". The user is greeted by a small window containing three visible elements:

- A field for a username
- A field for a password
- A button for submission.

The controller handles the logic behind the user input. Depending on what is inserted into the fields before submission, two things may happen:

- Error scenario: Something went wrong with the authentication. A red error message will appear at the bottom of the window detailing what the error is:
    - o "All fields are mandatory" – will appear if one of the two fields, or both, are empty.
    - o "Specified user not found" – will appear if the username provided does not match any of the usernames stored in the database.
    - o "Password does not match" – will appear if the username does match an entity stored in the database, but the provided password is incorrect.
- Success scenario: Both the username and the password match the records in our database. The authentication window is morphed into a full-screen view of the main menu.

Perhaps most interestingly we could state that, as previously mentioned, the passwords stored in the database are encrypted with the Advanced Encryption Standard (AES). Therefore, to compare the plain-text password of the user with the encrypted password in the database, the query which retrieves the information has to decrypt the password before making the comparison.

```sql
sql: "SELECT AES_DECRYPT(password, ?) AS password FROM users WHERE username = ?");
```

Figure 4.3.1 SQL SELECT statement that selects the decrypted password with a given unique username

```
password
16B 00000000   DE A8 DD 58 DC 6D 7B AD 3D 80 00 32 0E 9D F1 6D   Þ¨ÝXÜm{SHY=..2..ñm
16B 00000000   0D 78 FC B6 C8 61 40 71 80 C3 02 D3 F6 AB E3 F0   .xü¶Èa@q.Ã.Óö«ãð
16B 00000000   45 A5 9B EE D7 C7 B6 43 AF 0A 67 F2 B7 C4 84 78   E¥.î×Ç¶C¯.gò·Ä.x
<null>
16B 00000000   A9 F1 A6 88 20 BD 1F 87 8B 4A 36 CE 4F 72 B0 0A   ©ñ¦. ½...J6ÎOr°.
```

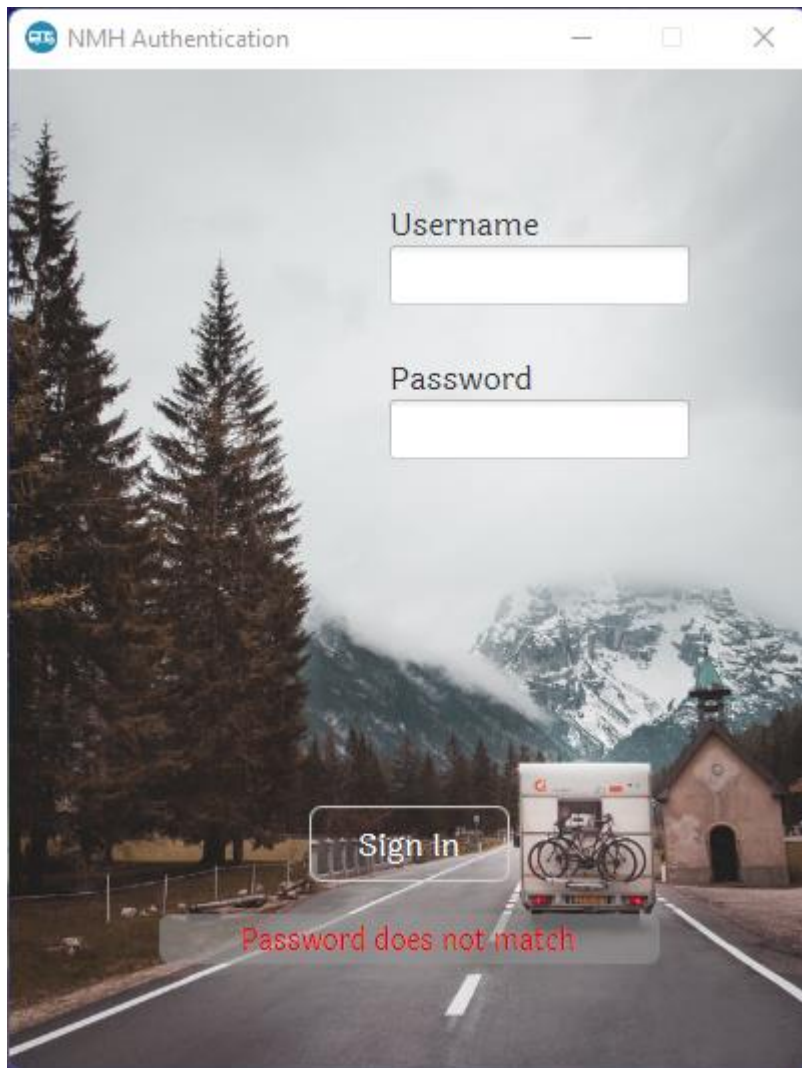*Figure 4.3.2 How the encrypted passwords look as stored in our database*

Figure 4.3.3 Authentication window with an error message at the bottom

4.4 Main Menu and GUI design – by Octavian Roman

4.4.1 Main Menu

As the user successfully logs in to the system, they are then presented with a full screen, maximized window. This window is the Main Menu, which is brought into being by the "main_menu" view whose controller is "MainMenuController".

The main menu is very simple; it only consists of a header, where the logo of the application and its name are shown to the left. To the right, an image and name belonging to the logged in user right next to a log out button is also present. Below the header, the main content of the window are three big rectangles, each designating a particular menu:

- Rentals Menu
- Motorhomes Menu
- Staff Menu

Selecting any of these menus will substitute the content of the window with that of the menu which has been picked. It should be noted that, while switching from a small authentication window to a maximized main menu window makes use of a function which changes the entire stage as the window dimensions and characteristics are not compatible, switching

between similar windows (which are coincidentally all windows except for the authentication) is facilitated by a different function which swaps the root (content) of a scene only. This is useful because changing the actual window (stage) all the time is detrimental to the application, as it causes a vast array of visual issues and resizing glitches depending on the operating system involved if the application is maximized.

```java
/**
 * Method that switched from one view to another.
 * Changes the content (scene) of a window (stage).
 * No more weird transitions.
 * @param FXMLView FXML file which represents the view to which we are switching.
 * @param stylesheet Stylesheet associated with the view.
 * @param node Node to retrieve the scene that needs to be switched. Normally you just want to pass the Node (e.g. button) triggering scene change here.
 */
public static void changeRoot(String FXMLView, String stylesheet, Node node) {
    Parent root = null;
    try {
        root = FXMLLoader.load(Objects.requireNonNull(FXUtils.class.getResource( name: "/view/" + FXMLView + ".fxml")));
    } catch (IOException e) {
        e.printStackTrace();
    }
    Scene scene = node.getScene();
    scene.getStylesheets().add(Objects.requireNonNull(FXUtils.class.getResource( name: "/stylesheets/" + stylesheet + ".css")).toExternalForm());
    scene.setRoot(root);
}
```

Figure 4.4.1.1 changeRoot() function which switches the content of a scene instead of the entire scene
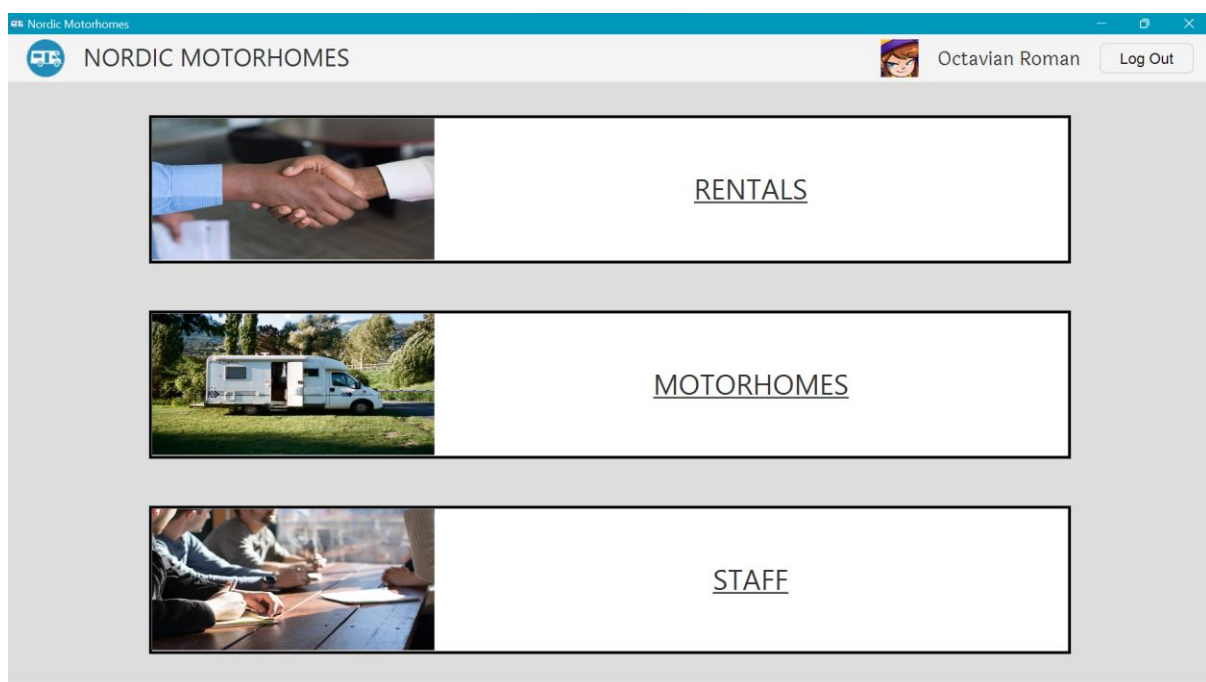
Figure 4.4.1.2 Main Menu window with its header and three buttons to access each submenu

4.4.2 Graphical User Interface Design

At this point it is appropriate to talk about the design of the GUI.

The interface has been made to take into account some of the 10 User Interface Design Heuristics described by Nielsen and Molich, such as:

- Error prevention: Error prevention mechanisms are present in the interface, for example, deleting something always prompts a confirmation, to prevent deleting data

by mistake. The image of the entity will appear as a quick visual cue to confirm that the user is indeed deleting what is intended.
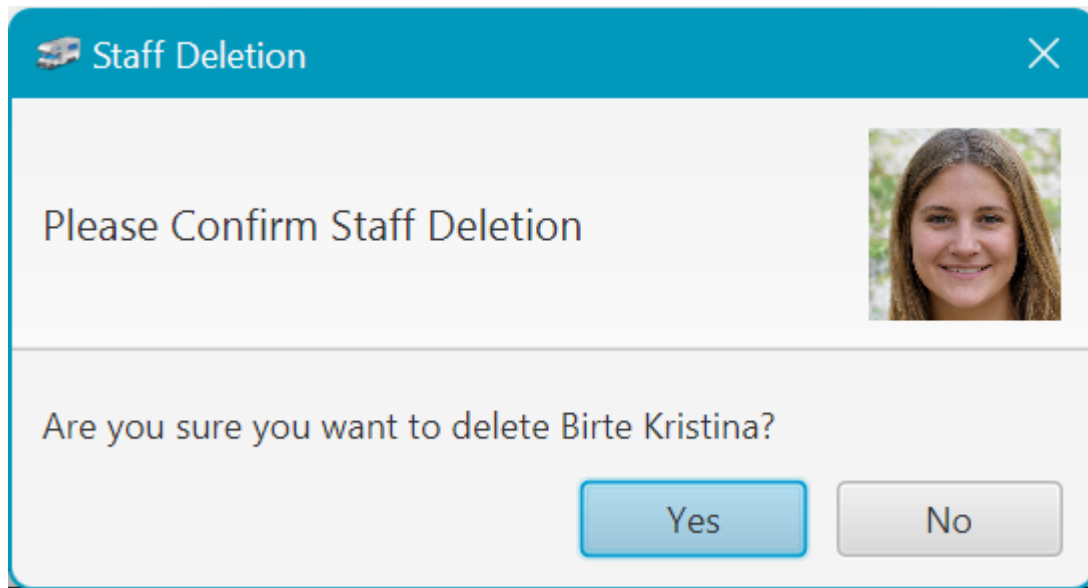

Figure 4.4.2.1 Deleting a staff member

- Consistency and standards: The interface is consistent across the entire application. The header is the same for all menus, and all main content is located below the header. All data entries are consistent in their dimensions and design, most buttons have the same dimensions, style, hover, and selection effects, etc.

- Aesthetic and minimalist design: The UI design intends to be clean and minimalist; with no clutter or over-the-top elements. The color scheme is unpretentious, with light grays dominating the interface, which are contrasted by light-blue accent colors on logos, elements of interest, hover effect for buttons, which try to achieve something similar to the styles present in the CSS library "Bootstrap".

- Help and documentation: Aside from labelling everything in sight, wherever possible, small details are included to guide the user. For example, when adding or editing a given entity, some fields may be required to successfully undergo the operation. Such requirement is shown by a red star next to the required field.

Figure 4.4.2.2 Red stars indicating which fields are required when adding a new rental to the system

Lastly, it should be mentioned that the UI is "semi" responsive. Responsive design is rather hard to implement in JavaFX, as it does not provide access to useful CSS elements such as "FlexBox". Instead, one must make do with the limited layout options provided by the platform. In our case, we are making a heavy use of "HBox" and "VBox" elements, which stand for "Horizontal Box" and "Vertical Box". We quite often nest many of these boxes to ensure visual elements remain centered and responsive to changes in other elements. This is not easy to achieve, as JavaFX mostly deals in absolute dimensions as opposed to relative ones. Nevertheless, we have managed to implement a design that should be functional, and maybe even pleasant to the eye, on most standard computer displays.

4.5 Abstract Menu Controller – by Bartosz Biryło

To create a unified template for both future and present creating of submenus and to reduce needed yet noisy boilerplate code, our application utilizes an *abstract* class called *MenuController* to ease the process of menu-creating. Currently All package submenu controllers (*StaffMenuController*, *MotorhomeMenuController*, *RentalMenuController*) reuse code thanks to this class.

The *MenuController* class consists de facto of 3 equally important parts: attributes, *final* methods and *abstract* methods. To understand how this class works and helps create cleaner code one first needs to learn all three sections of its code.

4.5.1. Attributes

```
public abstract class MenuController implements Initializable {
    @FXML protected Label usernameLabel;
    @FXML protected ImageView userImage;
    @FXML protected Button backButton;
    @FXML protected VBox entityContainer;
    @FXML protected Label entityCountLabel;
    @FXML protected HBox add;
```

Because every menu follows a standardized view, where company's logo and user informations are on top, a back-to-the-main-menu-button on bottom, and *VBox*, JFX class that acts as container for other *nodes* (JFX classes that can be displayed), in the middle, it means that a lot *@FXML* specific variables will be repeated throughout the menus. To counter that *MenuController* declares all those variables as *protected* attributes, meaning that they will get inherited by subclasses to later be let instantiated by JFX.

4.5.2 *abstract* methods

```
public abstract void fetchEntities();
public abstract void fetchEntities(String column, String order);
protected abstract void prepareToolbar();
@Override public abstract void initialize(URL url, ResourceBundle resourceBundle);
}
```

To create a uniform way of writing menus *MenuController* declares 4 *abstract* methods that need to be defined when inheriting from the class. This way the application's code is properly structured, and code can be further reduced by using these functions in *final* methods. *initialize()* can be considered special, as it is required to override by JFX's *Initializable* interface as the framework uses it to render the views.

4.5.3 *final* methods

```
// String gets changed depending on what order was last used.
// Can be used to flip order, not using boolean for clarity.
protected String currentOrder;


/**
 *Flips the order of the entities in the Scene.
 * @param field Field that is selected to flip the order (e.g. "start date")
 */
protected final void flipOrder(String field) {
    if (currentOrder.equals("ASC")) {
        fetchEntities(field, "DESC");
    } else fetchEntities(field, "ASC");
}

protected final void prepare() {
    FXUtils.setUserDetailsInHeader(usernameLabel, userImage);
    fetchEntities();
    prepareToolbar();
    backButton.setOnAction(actionEvent -> FXUtils.changeRoot( "main_menu", "main_menu", backButton));
}
```

Finally, *final* methods. The difference between them and *abstract* functions is that they not only provide *implementations* (or *definitions* from C/C++ jargon if you will) for the declared functions – simply meaning that they actually *do* something on their own – but also because they are *final*, meaning that they disallow any further *overrides*. This, combined with *abstract* methods implemented in child classes lets the *MenuController* reduce redundant lines of code such as fetching entities in every initialize() while at the same time keep shared functions consistent in subclass of every level (subclass of a subclass is also a subclass, but of level 2).

4.6 Main Entity handling – by Octavian Roman

The main goal of the application is to handle data. It may be tempting to merely insert data manually – by means of DML – for many of the relevant entities such as motorhomes, brands, or staff members. That would reduce the scope of the application significantly, as the only requirement for most entities would be to be able to display them.

Nevertheless, we believe that an application of this type should have dynamic data wherever possible. Therefore, we opted for putting serious amounts of work into ensuring that the most relevant entities to the business support the four basic data handling operations (CRUD):

- Create
- Read
- Update
- Delete

Implementing all these operations in JavaFX is a complex matter due to the fact that the platform provides no "shortcuts" or easy ways to handle complex relational data, perhaps excepting the "TableView" element, which is, in our opinion, too limited for the scope of this application.

Thus, we had to implement our own custom algorithms to make it possible to bring mere SQL rows and columns to life in a GUI application. Note that this section will only cover rental CRUD to provide continuity from the Use Cases, SSDs, and now the source code. *(If you would like to see more use cases in the form of windows of the application, check Appendix C, D and E)*

4.6.1 Read Entities – Entity Injection: The cornerstone of the application

Entity injection is perhaps the most important aspect of this application. It is the way, the algorithm, the set of instructions, through which we take given rows in a database, or "entities", and "inject" them into the GUI as JavaFX nodes that are dynamic and reactive.

The process begins as soon as the user selects a particular menu, given that we read all data by default throughout the system. JavaFX controller classes have an "Initializable" interface which, when implemented, allows us to override the "initialize" method of said interface. Initialize is very similar to a lifecycle hook in most front-end frameworks. It essentially "hooks" into the node as soon as it is initialized in the view and allows us to alter it and its child nodes. In this application we take full advantage of this feature in all controllers. For reading, when we initialize a given menu, inside of the initialize method we call a function "fetchEntities()" that handles the initial phase of the entity injection.

Here is the most complex fetch function, which gets called every time a user enters the Rental menu. This gets called by the Initialize method of the "RentalMenuController". It has been commented abundantly, to explain its inner workings in detail.

```java
/**
 * Fetch the existing Rental Entities from the database and display them in the Menu.
 * @param column Schema column which will be used to order the entities.
 * @param order String which will determine whether the order is ascending or descending: "ASC" or "DESC".
 */
@Override
public void fetchEntities(String column, String order) {
/*
 * 1. Clear container where fetching injection occurs to avoid duplication for multiple fetches.
 * 2. Clear relevant ORM ArrayLists to preserve data integrity over multiple fetches.
 * 3. Retrieve Rental, Client, Motorhome, Model and Brand entities from database and store them in ResultSet.
 * 4. Iterate over ResultSet, per iteration:
 *    a) Create objects for each entry of aforementioned entities.
 *    b) Retrieve via ID from database all Extra entities associated with the Rental object, store them all in extraResultSet.
 *    c) Instantiate an ArrayList of Extra objects.
 *    d) Iterate over extraResultSet, and for each entry, create Extra objects which are then added to the ArrayList.
 *    e) Add all objects and the Extra ArrayList to their ORM ArrayLists in persistence.Session.
 *    f) Immediately inject a new RentalEntity into the menu. This will trigger the RentalEntityController, which will handle the logic.
 * 5. Set the label displaying the entity count to the amount of Rental objects in ORM to ensure it stays updated over multiple fetches.
```

```java
 * 6. Finally, store the order that was used to fetch in order to be able to flip it on demand later.
 */
    Connection connection = Database.getConnection();
    try {
        // 1. Clear container where fetching injection occurs to avoid duplication for multiple fetches.
        entityContainer.getChildren().clear();
        // 2. Clear relevant ORM ArrayLists to preserve data integrity over multiple fetches.
        Session.rentalEntityList.clear();
        Session.clientEntityList.clear();
        Session.motorhomeEntityList.clear();
        Session.modelEntityList.clear();
        Session.brandEntityList.clear();
        Session.rentalExtrasCollectionList.clear();
        // 3. Retrieve Rental, Client, Motorhome, Model and Brand entities from database and store them in ResultSet.
        PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                "SELECT * FROM rentals " +
                        "JOIN clients ON rentals.id = clients.rental_id " +
                        "JOIN motorhomes ON rentals.motorhome_id = motorhomes.id " +
                        "JOIN models ON motorhomes.model_id = models.id " +
                        "JOIN brands ON models.brand_id = brands.id " +
                        "ORDER BY " + column + " " + order + ";");
        ResultSet resultSet = preparedStatement.executeQuery();
        // 4. Iterate over ResultSet, per iteration:
        while (resultSet.next()) {
            // a) Create objects for each entry of aforementioned entities.
            Rental rental = new Rental(
                    resultSet.getInt("rentals.id"),
                    resultSet.getInt("motorhome_id"),
                    resultSet.getInt("distance"),
                    resultSet.getString("location"),
                    resultSet.getString("season"),
                    resultSet.getDate("start_date"),
                    resultSet.getDate("end_date"),
                    resultSet.getDouble("final_price"),
                    resultSet.getString("notes")
            );
            Client client = new Client(
                    resultSet.getInt("clients.id"),
                    resultSet.getInt("rentals.id"),
                    resultSet.getString("firstName"),
                    resultSet.getString("lastName"),
                    resultSet.getString("telephone")
            );
            Motorhome motorhome = new Motorhome(
                    resultSet.getInt("motorhomes.id"),
                    resultSet.getInt("model_id"),
                    resultSet.getString("image"),
                    resultSet.getBoolean("rented"),
                    resultSet.getString("type"),
                    resultSet.getInt("beds")
            );
            Model model = new Model(
                    resultSet.getInt("models.id"),
                    resultSet.getInt("brand_id"),
                    resultSet.getString("models.name"),
                    resultSet.getDouble("models.price")
            );
            Brand brand = new Brand(
                    resultSet.getInt("brands.id"),
                    resultSet.getString("brands.name"),
                    resultSet.getDouble("brands.price")
            );
            // b) Retrieve via ID from database all Extra entities associated with the Rental object, store them all in extraResultSet.
            preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                    "SELECT * FROM rentalextras " +
                            "JOIN extras ON extra_id = extras.id " +
                            "WHERE rental_id = ?");
            preparedStatement.setInt(1, resultSet.getInt("rentals.id"));
            ResultSet extraResultSet = preparedStatement.executeQuery();
            // c) Instantiate an ArrayList of Extra objects.
            ArrayList<Extra> extraArrayList = new ArrayList<>();
            // d) Iterate over extraResultSet, and for each entry, create Extra objects which are then added to the ArrayList.
            while (extraResultSet.next()) {
                Extra extra = new Extra(
                        extraResultSet.getInt("id"),
                        extraResultSet.getString("name"),
                        extraResultSet.getDouble("price")
                );
                extraArrayList.add(extra);
            }
            // e) Add all objects and the Extra ArrayList to their ORM ArrayLists in persistence.Session.
            Session.rentalEntityList.add(rental);
            Session.clientEntityList.add(client);
```

```
        Session.motorhomeEntityList.add(motorhome);
        Session.modelEntityList.add(model);
        Session.brandEntityList.add(brand);
        Session.rentalExtrasCollectionList.add(extraArrayList);
        // f) Immediately inject a new RentalEntity into the menu. This will trigger the RentalEntityController, which will handle the logic
        FXUtils.injectEntity("rental_entity", entityContainer);
    }
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    Database.closeConnection(connection);
    // 5. Set the label displaying the entity count to the amount of Rental objects in ORM to ensure it stays updated over multiple fetches.
    // Needed because this gets called after adding or deleting, which means that the count may have changed since the last time it was called.
    entityCountLabel.setText(Session.rentalEntityList.size() + " Items");
    // 6. Finally, store the order that was used to fetch in order to be able to flip it on demand later.
    currentOrder = order;
    }
}
```

This function handles part of the implementation of the "Read rentals" use case. Very briefly, it selects all relevant entities and puts them in a ResultSet, then it iterates over the ResultSet, and creates objects with each entity. Because extras have a one-to-many relationship between rentals and them, in other words, because a rental can have many extras, none, or just one, we need to make an inner loop to select all the extras associated with the rental, create objects for each one of them, and store them in a collection.

Once all the objects and the ArrayList of extras have been created during a cycle of the iteration, we add them to ArrayLists in a class Session dedicated to runtime memory (ORM). These collections provide persistence of data during a session, as opposed to long-term persistence provided by a MySQL database. These collections store the information of all entities so that it may be retrieved in the next stage of the injection process.

Once the ArrayLists have been loaded with the new related entities, we immediately use the following function to inject a JavaFX node into the content of a "ScrollPane" container.

```
/**
 * Inject entity into Pane. Pane can be anything, although in our case it's
always a VBox.
 * Facilitates adding as many entities as needed into a given container in
a dynamic way.
 * @param fxmlFile fxml file defining the UI element corresponding to the
entity
 * @param entityContainer Pane where the entity is to be injected
 */
public static void injectEntity(String fxmlFile, Pane entityContainer) {
    try {
        new FXMLLoader();
        Parent root =
FXMLLoader.load(Objects.requireNonNull(FXUtils.class.getResource("/view/" +
fxmlFile + ".fxml")));
        entityContainer.getChildren().add(root);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

This JavaFX node is like a view in that it has a controller with an Initializable method that allows us to perform logic as soon as it appears. It differs in that instead of using Initializable for a new view, we use it for individual FX elements.
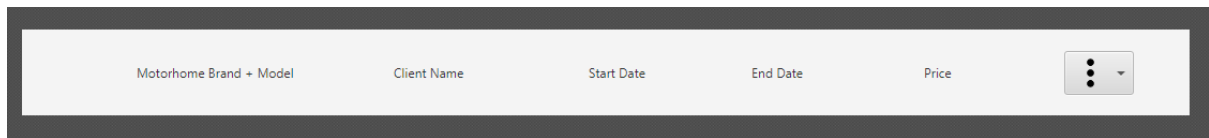
Figure 4.6.1.1 How the rental_entity view looks in SceneBuilder

As soon as the element is injected into the menu, it triggers the controller associated with it. In the controller, many things need to be done to ensure that all CRUD operations work, but for fetching all we need to do is to retrieve the data from the last elements of the ORM ArrayLists in Session and assign them to the FX nodes in the controller. To do that, we first get the index of the last element like so:

```
// This class is always going to insert the last entity inside the ArrayList.
// The same index for brands and models since they get added simultaneously.
public final int entityIndex = Session.rentalEntityList.size() - 1;
```

Then, for readability purposes we create new objects with the data stored in the ORM ArrayLists and assign the value of the properties of said objects to the FX nodes of the rental_entity view.

```java
@Override
public void initialize(URL url, ResourceBundle resourceBundle) {
    // Generate objects with retrieved data
    Rental rental = Session.rentalEntityList.get(entityIndex);
    Client client = Session.clientEntityList.get(entityIndex);
    Motorhome motorhome = Session.motorhomeEntityList.get(entityIndex);
    Brand brand = Session.brandEntityList.get(entityIndex);
    Model model = Session.modelEntityList.get(entityIndex);

    // Put data in FX nodes
    Image i = new
Image(Objects.requireNonNullElse(getClass().getResource(motorhome.getImage(
)),

getClass().getResource("/assets/motorhomes/motorhome_placeholder.png")).toE
xternalForm());
    image.setImage(i);
    motorhomeLabel.setText(brand.getName() + " " + model.getName());
    clientLabel.setText(client.getFirstName() + " " +
client.getLastName());
    startDateLabel.setText(String.valueOf(rental.getStart_date()));
    endDateLabel.setText(String.valueOf(rental.getEnd_date()));

priceLabel.setText(FXUtils.formatCurrencyValues(rental.getFinal_price()) +
" €");

    // Prepare additional views and functions to handle Update or Deletion
of the given entity
    edit.setOnAction(actionEvent -> {
        RentalEditController.entityIndex = entityIndex;
        FXUtils.popUp("rental_edit", "popup", "Edit Rental");
        Bridge.getRentalMenuController().fetchEntities("firstName", "ASC");
    });

    delete.setOnAction(actionEvent -> {
        remove(rental, motorhome, client);
        Bridge.getRentalMenuController().fetchEntities("firstName", "ASC");
    });
}
```

This might seem complicated, yet we believe it is understandable taking into consideration that we are building this algorithm "from scratch". Nevertheless, the result is satisfactory:

Figure 4.6.1.2 Rental Menu after entity injection

4.6.2 Create Entities

Creating new entities is done through the "plus" symbol seen in the figure 4.6.1.2. The button triggers a pop-up window to appear, as seen below:

Figure 4.6.2.1 New Rental Window

The user gets to fill in the fields as desired, and to pick a motorhome by either pressing the button "Pick Motorhome" or by clicking on the image. After a motorhome has been picked, the "Confirm" button is enabled, which triggers the attempt to create a new rental instance in the database with the information provided.

All of this is possible thanks to the "AddRentalController" which handles the logic behind this operation. It is the most complex "Create" controller because it also has to handle reactivity: As the user picks different motorhomes, seasons, dates, distance or extras, both the title of the window, and the prices must change to reflect the changes in real time.

The "Create" algorithm, however, works as follows for all entities:

First, a "generateEntities()" function generates all objects that are trying to be added to the system in accordance to the data provided in the fields. For rentals, it looks like this:

```java
/**
 * Generates Rental and Client entities that will be added to the database.
 * @param finalPrice Final Price needs to be computed before a new rental
can be added, and is provided here
 * @return array with Rental at index 0 and Client at index 1
 */
private Object[] generateEntities(double finalPrice) {
    String seasonEnum =  "";
    switch (seasonChoiceBox.getValue()) {
        case "Low Season" -> seasonEnum = "L";
        case "Mid Season" -> seasonEnum = "M";
        case "Peak Season" -> seasonEnum = "P";
    }
    Rental rental = new Rental(
            motorhomeId,
            Integer.parseInt(distanceField.getText().equals("") ? "0" :
distanceField.getText()),
            pickUpLocationField.getText(),
            seasonEnum,
            java.sql.Date.valueOf(startDateDatePicker.getValue()),
            java.sql.Date.valueOf(endDateDatePicker.getValue()),
            finalPrice,
            notes.getText()
    );
    Client client = new Client(
            0,
            clientFirstNameField.getText().equals("") ? null :
clientFirstNameField.getText(),
            clientLastNameField.getText(),
            clientTelephoneField.getText()
    );
    Object[] result = new Object[2];
    result[0] = rental;
    result[1] = client;
    return result;
}
```

After the entities are generated, "addRental()" can be called, which takes the generated objects, and tries to create new entities with their attribute values. Below lies the most complex "create" function of the system:

```java
/**
 * Adds rental, client and rentalExtras entities to the database.
 * 1. Compute final price.
 * 2. Generate entities with field data and computed price.
 * 3. Insert rental entity into database.
 * 4. Save newly inserted rental entity ID into local variable.
 * 5. Insert client entity into database and associate it with rental using the retrieved ID.
 * 6. Iterate over extraArrayList and add database entry for each extra present in the collection.
 * 7. Set motorhome which has been just rented "rented" boolean to true, since it is not available anymore.
 * @return true if successful, false otherwise
 */
private boolean addRental() {
    Connection connection = Database.getConnection();
    try {
        // Compute final price.
        Object[] oldEntityArray = FXUtils.retrieveMotorhomeEntities(motorhomeId);
        Model model = (Model) Objects.requireNonNull(oldEntityArray)[1];
        Brand brand = (Brand) Objects.requireNonNull(oldEntityArray)[2];
        double dailyPrice = FXUtils.computeDailyPrice(brand, model, seasonChoiceBox);
        double finalPrice = FXUtils.computeFinalPrice(
                dailyPrice,
                java.sql.Date.valueOf(startDateDatePicker.getValue()),
                java.sql.Date.valueOf(endDateDatePicker.getValue()),
                Integer.parseInt(distanceField.getText().equals("") ? "0" : distanceField.getText()),
                Session.extraSelectionList);

        // Generate entities with field data and computed price.
        Object[] newEntityArray = generateEntities(finalPrice);
        Rental rental = (Rental) newEntityArray[0];
        Client client = (Client) newEntityArray[1];

        // Insert rental entity into database.
        PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                "INSERT INTO rentals (motorhome_id, distance, season, start_date, end_date, final_price, notes) " +
                    "VALUES (?,?,?,?,?,?,?);");
        preparedStatement.setInt(1, rental.getMotorhome_id());
        preparedStatement.setInt(2, rental.getDistance());
        preparedStatement.setString(3, rental.getSeason());
        preparedStatement.setDate(4, rental.getStart_date());
        preparedStatement.setDate(5, rental.getEnd_date());
        preparedStatement.setDouble(6, rental.getFinal_price());
        preparedStatement.setString(7, rental.getNotes());
        preparedStatement.execute();

        // Save newly inserted rental entity ID into local variable.
        preparedStatement = connection.prepareStatement("SELECT LAST_INSERT_ID()");
        ResultSet resultSet = preparedStatement.executeQuery();
        resultSet.next();
        int id = resultSet.getInt(1);

        // Insert client entity into database and associate it with rental using the retrieved ID.
        preparedStatement = connection.prepareStatement(
                "INSERT INTO clients (rental_id, firstName, lastName, telephone) " +
                    "VALUES (?,?,?,?);");
        preparedStatement.setInt(1, id);
        preparedStatement.setString(2, client.getFirstName());
        preparedStatement.setString(3, client.getLastName());
        preparedStatement.setString(4, client.getTelephone());
        preparedStatement.execute();

        // Iterate over extraArrayList and associate database entry for each extra present in the collection.
        for (Extra extra : Session.extraSelectionList) {
            preparedStatement = connection.prepareStatement(
                    "INSERT INTO rentalextras (rental_id, extra_id) VALUES (?,?);");
            preparedStatement.setInt(1, id);
            preparedStatement.setInt(2, extra.getId());
            preparedStatement.execute();
        }

        // Set motorhome which has been just rented "rented" boolean to true, since it is not available anymore.
        preparedStatement = connection.prepareStatement(
                "UPDATE motorhomes SET rented = 1 WHERE id = ?");
        preparedStatement.setInt(1, rental.getMotorhome_id());
        preparedStatement.execute();
        return true;
    } catch (SQLException | NullPointerException e) {
        e.printStackTrace();
        return false;
    } finally {
        Database.closeConnection(connection);
    }
}
```

The result is that, if successful, the fetching function will be called to refresh the menu, which will make the newly created rental appear.

4.6.3 Update Entities

Updating an entity is slightly harder than adding a new one because we need to also retrieve its previous details, as seen below:

Figure 4.6.3.1

Fetching takes care of that by setting a custom event handler for each edit button in an entity's options. When we click a given "edit" button, the index of the object representing that entity in the ORM ArrayLists is passed to the edit controller, which then instantiates new objects and assigns them the values of the objects in memory with a "loadDataIntoFields()" function:

```
/**
 * Retrieve motorhome data for a given entity and load it into the fields.
 * Dynamic labels (title + price) also need to be set initially, hence the SQL statements
 * @param motorhome Motorhome object representing the entity whose data is to be loaded into the fields
 */
private void loadDataIntoFields(Rental rental, Client client, Motorhome motorhome, Model model, Brand brand, ArrayList<Extra> extraArrayList) {
    clientFirstNameField.setText(client.getFirstName());
    clientLastNameField.setText(client.getLastName());
    clientTelephoneField.setText(client.getTelephone());
    String season = "Season";
    switch (rental.getSeason()) {
        case "L" -> season = "Low Season";
        case "M" -> season = "Mid Season";
        case "P" -> season = "Peak Season";
    }
    seasonChoiceBox.setValue(season);
    pickUpLocationField.setText(rental.getLocation());
    distanceField.setText(String.valueOf(rental.getDistance()));
```

```
startDateDatePicker.setValue(rental.getStart_date().toLocalDate());
endDateDatePicker.setValue(rental.getEnd_date().toLocalDate());
// Retrieve image and create new FX image node with it
Image image = new Image(Objects.requireNonNullElse(getClass().getResource(motorhome.getImage()),
        getClass().getResource("/assets/motorhomes/motorhome_placeholder.png")).toExternalForm());
// Put node into ImageView
this.image.setImage(image);
notes.setText(rental.getNotes());
Session.extraSelectionList.clear();
Session.extraSelectionList.addAll(extraArrayList);
motorhomeId = motorhome.getId();
updateDynamicFields(model, brand);
}
```

Now that we have retrieved the previous information, we can make the desired changes. When we are satisfied, we may click on "Confirm", which will trigger the actual "Update" algorithm:

First, take the objects from the ORM, and try to assign it the values stored in the fields with the "updateEntities()" function:

```java
/**
 * Take a Rental and Client object and update their values to match the
fields of the pop-up.
 * @param rental Rental object representing the rental entity we are about
to update with editRental().
 * @param client Client object representing the rental entity we are about
to update with editRental().
 * @return An array with the Rental object at index 0 and Client object and
index 1
 */
private Object[] updateEntities(Rental rental, Client client) {
    String seasonEnum = "";
    switch (seasonChoiceBox.getValue()) {
        case "Low Season" -> seasonEnum = "L";
        case "Mid Season" -> seasonEnum = "M";
        case "Peak Season" -> seasonEnum = "P";
    }
    Object[] entityArray = FXUtils.retrieveMotorhomeEntities(motorhomeId);
    Model model = (Model) Objects.requireNonNull(entityArray)[1];
    Brand brand = (Brand) Objects.requireNonNull(entityArray)[2];
    double dailyPrice = FXUtils.computeDailyPrice(brand, model,
seasonChoiceBox);
    double finalPrice = FXUtils.computeFinalPrice(
            dailyPrice,
            java.sql.Date.valueOf(startDateDatePicker.getValue()),
            java.sql.Date.valueOf(endDateDatePicker.getValue()),
            Integer.parseInt(distanceField.getText().equals("") ? "0" :
distanceField.getText()),
            Session.extraSelectionList);

    rental.setDistance(Integer.parseInt(distanceField.getText().equals("")
? "0" : distanceField.getText()));
    rental.setLocation(pickUpLocationField.getText());
    rental.setSeason(seasonEnum);

rental.setStart_date(java.sql.Date.valueOf(startDateDatePicker.getValue()))
;

rental.setEnd_date(java.sql.Date.valueOf(endDateDatePicker.getValue()));
    rental.setFinal_price(finalPrice);
    rental.setNotes(notes.getText());

    client.setFirstName(clientFirstNameField.getText().equals("") ? null :
clientFirstNameField.getText());
    client.setLastName(clientLastNameField.getText());
    client.setTelephone(clientTelephoneField.getText());

    Object[] result = new Object[2];
    result[0] = rental;
    result[1] = client;
    return result;
}
```

Update entities returns objects with the new values. These objects can then be used in another
function which queries the database to actually update the entities:

```java
/**
 * Takes a rental and client object and updates the database entries with their attribute values.
 * @param r Rental object where the new values are stored
 * @param c Client object where the new values are stored
```

```
 * @return true if successful, false otherwise
 */
private boolean editRental(Rental r, Client c) {
    Object[] entityArray = updateEntities(r, c);
    Rental rental = (Rental) entityArray[0];
    Client client = (Client) entityArray[1];
    Connection connection = Database.getConnection();
    try {
        PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
            "UPDATE rentals SET distance = ?, location = ?, season = ?, start_date = ?, end_date = ?, final_price = ?, notes = ? " +
                "WHERE id = ?");
        preparedStatement.setInt(1, rental.getDistance());
        preparedStatement.setString(2, rental.getLocation());
        preparedStatement.setString(3, rental.getSeason());
        preparedStatement.setDate(4, rental.getStart_date());
        preparedStatement.setDate(5, rental.getEnd_date());
        preparedStatement.setDouble(6, rental.getFinal_price());
        preparedStatement.setString(7, rental.getNotes());
        preparedStatement.setInt(8, rental.getId());
        preparedStatement.execute();

        preparedStatement = connection.prepareStatement(
            "UPDATE clients SET firstName = ?, lastName = ?, telephone = ? " +
                "WHERE id = ?");
        preparedStatement.setString(1, client.getFirstName());
        preparedStatement.setString(2, client.getLastName());
        preparedStatement.setString(3, client.getTelephone());
        preparedStatement.setInt(4, client.getId());
        preparedStatement.execute();
        // Clear all previous extra associations (even the ones that are still valid, since we're adding them again)
        preparedStatement = connection.prepareStatement("DELETE FROM rentalextras WHERE rental_id = ?");
        preparedStatement.setInt(1, rental.getId());
        preparedStatement.execute();

        // Iterate over extraArrayList and associate database entry for each extra present in the collection.
        for (Extra extra : Session.extraSelectionList) {
            preparedStatement = connection.prepareStatement(
                "INSERT INTO rentalextras (rental_id, extra_id) VALUES (?,?);");
            preparedStatement.setInt(1, rental.getId());
            preparedStatement.setInt(2, extra.getId());
            preparedStatement.execute();
        }
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    } finally {
        Database.closeConnection(connection);
    }
}
```

All of this, if successful, results in the attributes of a given entity changing on demand.

4.6.4 Delete Entities

Lastly the least complex CRUD operation in our application, which is "Delete". It only needs a single method and it does not pop-up another window such as "Create" or "Add". Subsequently, it is defined in the controller of the entity as such for rentals:

```
/**
 * Remove a Rental entity from the Database.
 * Also sets the status of a Motorhome to "available"
 * @param rental Rental object representing the entity to be removed.
 * @param motorhome Motorhome object associated with the rental
 * @param client Client object associated with the rental. Deletion will cascade to it.
 */
private void remove(Rental rental, Motorhome motorhome, Client client) {
    Alert alert = FXUtils.confirmDeletion("Rental", client.getFirstName(), client.getLastName(), image);
    if (alert.getResult() == ButtonType.YES) {
        Connection connection = Database.getConnection();
        try {
            PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                "DELETE FROM rentals WHERE id = ?;");
            preparedStatement.setInt(1, rental.getId());
            preparedStatement.execute();

            preparedStatement = connection.prepareStatement(
                "UPDATE motorhomes SET rented = 0 WHERE id = ?;");
```

```
        preparedStatement.setInt(1, motorhome.getId());
        preparedStatement.execute();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        Database.closeConnection(connection);
    }
  }
}
```

Selecting "delete" merely calls this function with the appropriate objects, which takes care of the deletion operation.

4.7 Settings Menus – by Jakub Patelski

4.7.1 Read brands, models, extras

The logic behind read in Settings Menu is the same as the one described in the previous section. The program fetches the data everywhere in the same way.

```
private void fetchExtras() {
    // Establish connection
    Connection connection = Database.getConnection();
    try {
        // Prepare SQL statement
        PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                sql: "SELECT * FROM extras ORDER BY name;");
        // Execute statement and store result in a ResultSet
        ResultSet resultSet = preparedStatement.executeQuery();
        // While there are brands in the resultSet
        Session.extraEntityList.clear();
        while (resultSet.next()) {
            // Create extra object for each entity
            Extra extra = new Extra(
                    resultSet.getInt( columnLabel: "id"),
                    resultSet.getString( columnLabel: "name"),
                    resultSet.getDouble( columnLabel: "price")
            );
            // Add extra object to ArrayList for further manipulation
            Session.extraEntityList.add(extra);
            // Finally, inject the entity view into the container, triggering its controller
            FXUtils.injectEntity( fxmlFile: "extra_selection_entity", extrasContainer);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        Database.closeConnection(connection);
    }
}
```
Figure 4.7.1.1 Fetch Extras in RentalSettingsController

Here is the less complex fetch function in the whole program, which gets called every time user enters rentals settings. It selects all extras entities and puts them in a ResultSet, then it iterates over the ResultSet till the last input, and create objects with each entity. Th object is later added to ArrayList that is also in the class Session dedicated to runtime memory (ORM).

Once the ArrayLists have been loaded with the new related entities, we immediately use the following function to inject a JavaFX node into the content of a "ScrollPane" container.

When the element is injected into the settings, it triggers the controller associated with it. In fetching we always retrieve the last element of the ORM ArrayLists in Session and assign them to the FX nodes in the controller.

Then, for readability purposes we create new objects with the data stored in the ORM ArrayLists and assign the value of the properties of objects to the FX nodes of the Extras entity view.

4.7.2 Create brands, models, extras

Creating a new entities in extras is done by first clicking seetings function that displays pop-up window "rental option".
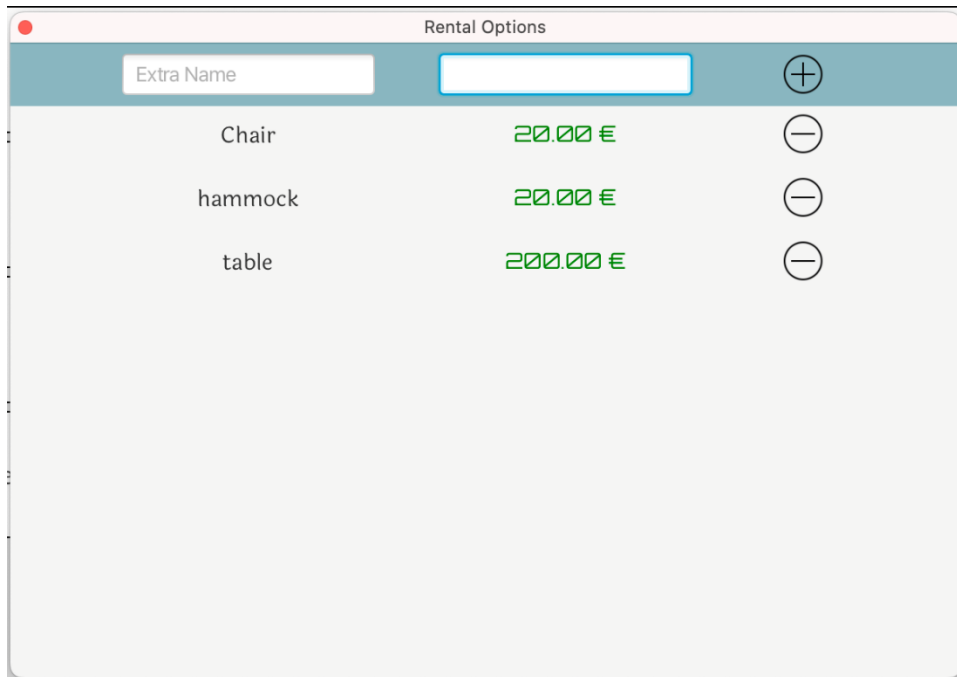

Figure 4.7.2.1 Pop-up rental options window

Then 2 fields- extra name and price/day needs to be filled to create and button with plus sign click, if one of the fields is empty Error alert pop-up to the screen with SQL error header.
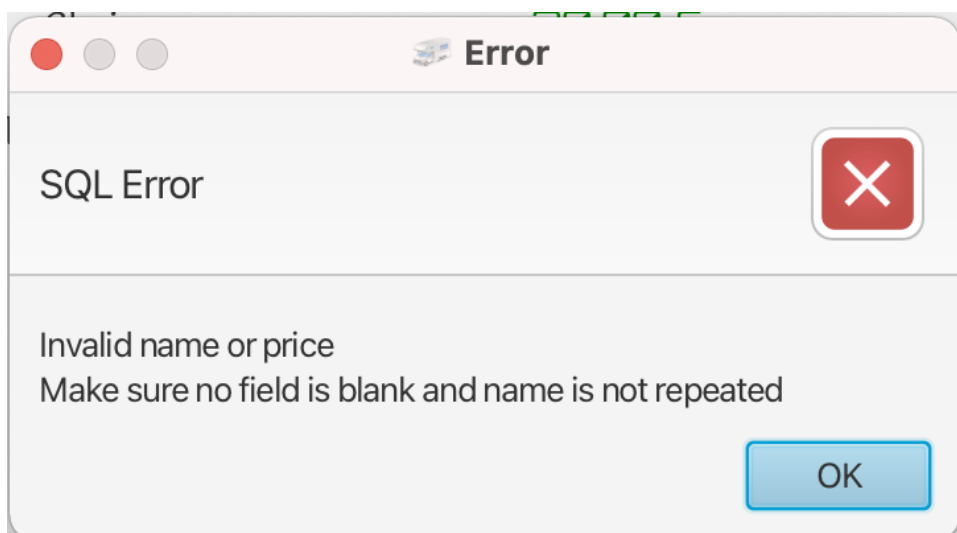
Figure 4.7.2.2 Error message

This is done by addExtra() method which handles the logic behind it.

```
private void addExtra() {
    // Establish connection
    Connection connection = Database.getConnection();
    try {
        // Prepare and execute SQL statement
        PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                sql: "INSERT INTO extras (name, price) VALUES (?,?);"
        );
        preparedStatement.setString( parameterIndex: 1, extraNameField.getText().equals("") ? null : extraNameField.getText());
        preparedStatement.setDouble( parameterIndex: 2, Double.parseDouble(extraPriceField.getText()));
        preparedStatement.execute();
        // Re-fetch the brands to reflect changes
        fetchExtras();
        // Clear fields from inserted values
        extraNameField.setText("");
        extraPriceField.setText("");
    } catch (SQLException | NumberFormatException e) {
        e.printStackTrace();
        // Print error message if something goes wrong
        FXUtils.alert(
                Alert.AlertType.ERROR,
                contextText: "Invalid name or price\nMake sure no field is blank and name is not repeated",
                title: "Error",
                header: "SQL Error",
                show: true);
    } finally {
        Database.closeConnection(connection);
    }
}
```

Figure 4.7.2.3 addExtra()

If two fields are filled and the plus sign is clicked (addExtraButton) then fetchExtras() is called again that finally displays the new entity to the GUI.

4.7.3 Delete brands, models, extras

Deleting an extra is did by clicking on minus sign(remove) that is just next to the rental extra item. When it is clicked then remove(Extra extra) is called up and Alert "Extra Deletion" pops up to the screen with yes and no button.

```
private void remove(Extra extra) {
    // Get confirmation with an alert
    Alert alert = FXUtils.confirmDeletion( entity: "Extra", nameLabel.getText());
    // If confirmation has been received
    if (alert.getResult() == ButtonType.YES) {
        Connection connection = Database.getConnection();
        try {
            // Prepare statement
            PreparedStatement preparedStatement = Objects.requireNonNull(connection).prepareStatement(
                    sql: "DELETE FROM extras WHERE id = ?"
            );
            preparedStatement.setInt( parameterIndex: 1, extra.getId());
            preparedStatement.execute();
            // If removal is successful, use the Bridge to clear and re-fetch the extras to reflect changes
            Bridge.getRentalSettingsController().fetchExtras();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            Database.closeConnection(connection);
        }
    }
}
```
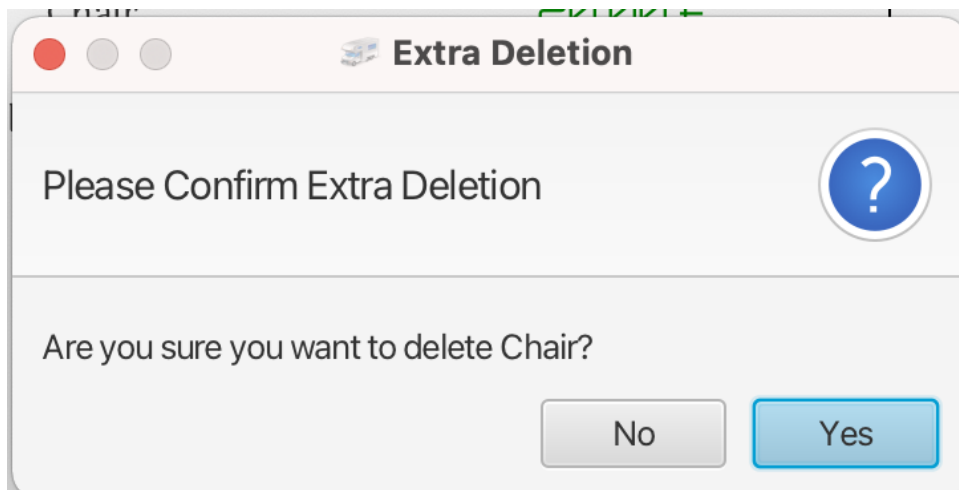
Figure 4.7.3.1 remove(Extra extra)

Figure 4.7.3.2 Extra Deletion Alert

If yes is clicked then the extras preparedStatement.setInt(1, extra.getId()) takes extras ID and prepared statement is executed that removes extra with given ID from the database. Afterwards with use of Bridge fetchExtras() is called again and clear and fetch all the extras again.

4.8 Experimental features

As mentioned in the first point of this chapter, our setup can create executables, alongside other extra features that surpass the requirements of the project. The executables are not fully functional for two reasons:

- JavaFX may glitch out in an executable form, sometimes creating visual artifacts that require a computer to restart to go away. We are not sure what causes this, but we suspect that it may be related to the fact that we are pushing JavaFX to do things it is not designed to do.
- Because we find the previous point unacceptable, we have not revamped our file handling logic to work in JAR.

Currently, when an image is picked, it's file extension is extracted, a unique name is generated in the form of a date timestamp and concatenated to the beginning of the extension, then the file itself is moved both to our "assets" folder in main, and to the "assets" folder in the compiled directory target. Then, that new image and its name can be used to display it across the system. However, in a JAR format, modifying the files of the JAR itself at runtime is hard to achieve, or even almost impossible in the case of an executable, and not a good idea for many reasons. Because of that, picking new images in executable does not work, although existing ones do show up.

To make image selection functional, we would have to revamp our file management to instead use a platform-independent directory outside of the executable itself on the user's machine. This has not been done because, even if we were to do it, we would probably still

not be capable of overcoming the first point in a timely manner and deem it unacceptable to provide a solution that may cause significant glitches or errors.

5 Transition

Our transition has not been very eventful. Taking into consideration that after each feature was implemented, we did thorough UX testing, not that many errors have surfaced. The few glitches we had were taken care of during this phase.

During said phase, we mainly focused on:

- Code refactoring
- Code clean-up
- Bug fixes
- Writing documentation
- Writing this very report
- Starting to prepare a slideshow presentation for the exam

6 Conclusion

To finalize the main section of this report our general thoughts are the following:

We are of the opinion that our software system provides an interesting solution to the problem stated herein. It appears that we have covered the most important use cases of the Nordic Motorhome System; and they have been covered in a rigorous way – without cutting many (if any) corners.

We agree on the fact that our project would be significantly worse had we not followed the guidelines of the iterative development and separated the workload into digestible chunks. As evidenced in the report, some parts of the system are rather complex and easy to get wrong. Thus, we certainly benefitted from a structured project lifecycle and from all the UML artifacts which have guided us through the construction of the system.

While there certainly are more use cases to develop and ways to further improve and optimize the solution, we seem to have managed to successfully follow the Unified Process to create what appears to be a satisfactory native application that should fulfil the immediate administrative needs of the rental company. Not only that, but we went out of our way to provide features and functionalities which may not necessarily be part of what we have been taught so far nor part of the requirements of this system.

# 7 Installation Instructions and GitHub Link – by Octavian Roman

Github Link:

https://github.com/boagroup/NordicMotorhome

Follow these steps to run the program in an IDE:

1. Clone the following repository: https://github.com/boagroup/NordicMotorhome.git
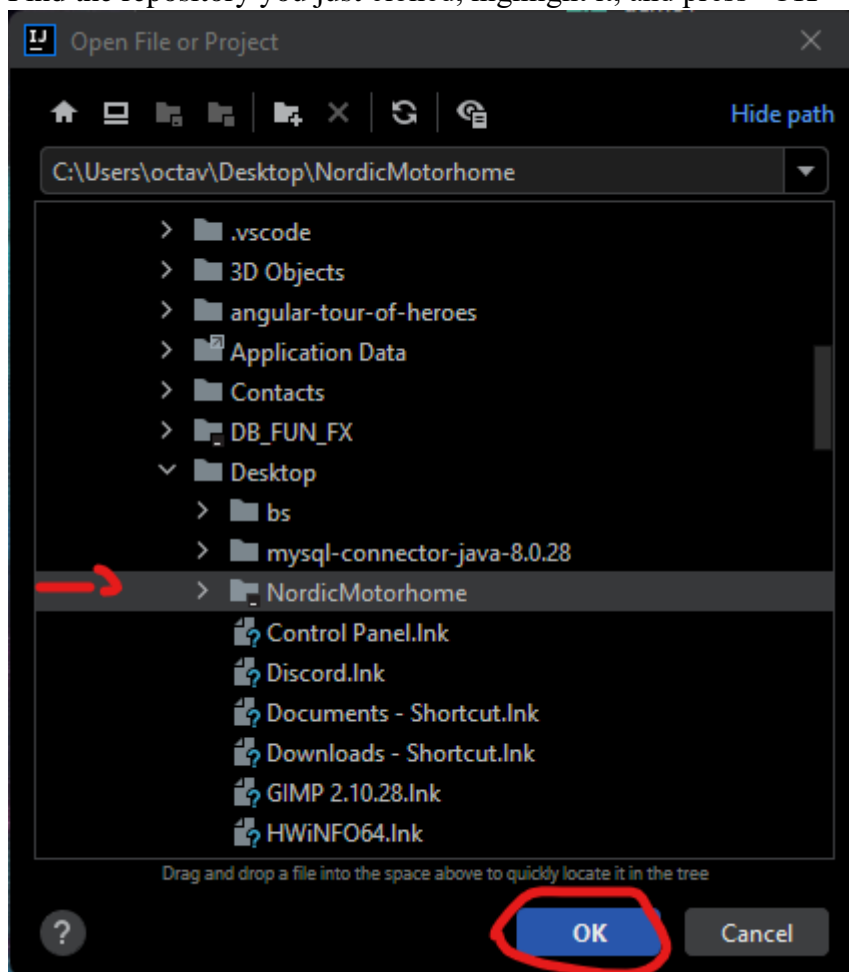   Bear in mind that this requires you to have Git installed. Use the following command
   to clone it in your desired location using a terminal:

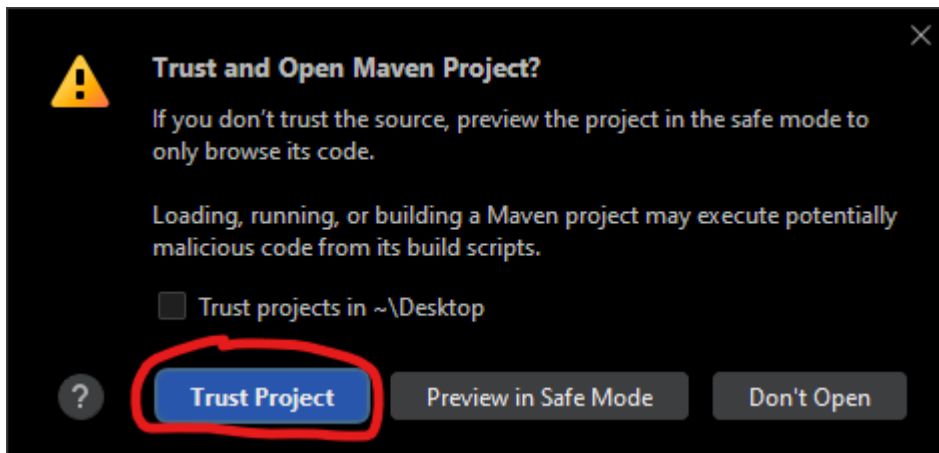   git clone https://github.com/boagroup/NordicMotorhome.git

2. Open the IDE of your choice, in our case we'll be using IntelliJ IDEA.
3. Select "Open"
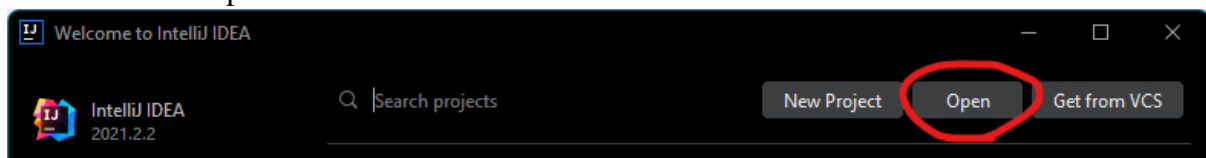
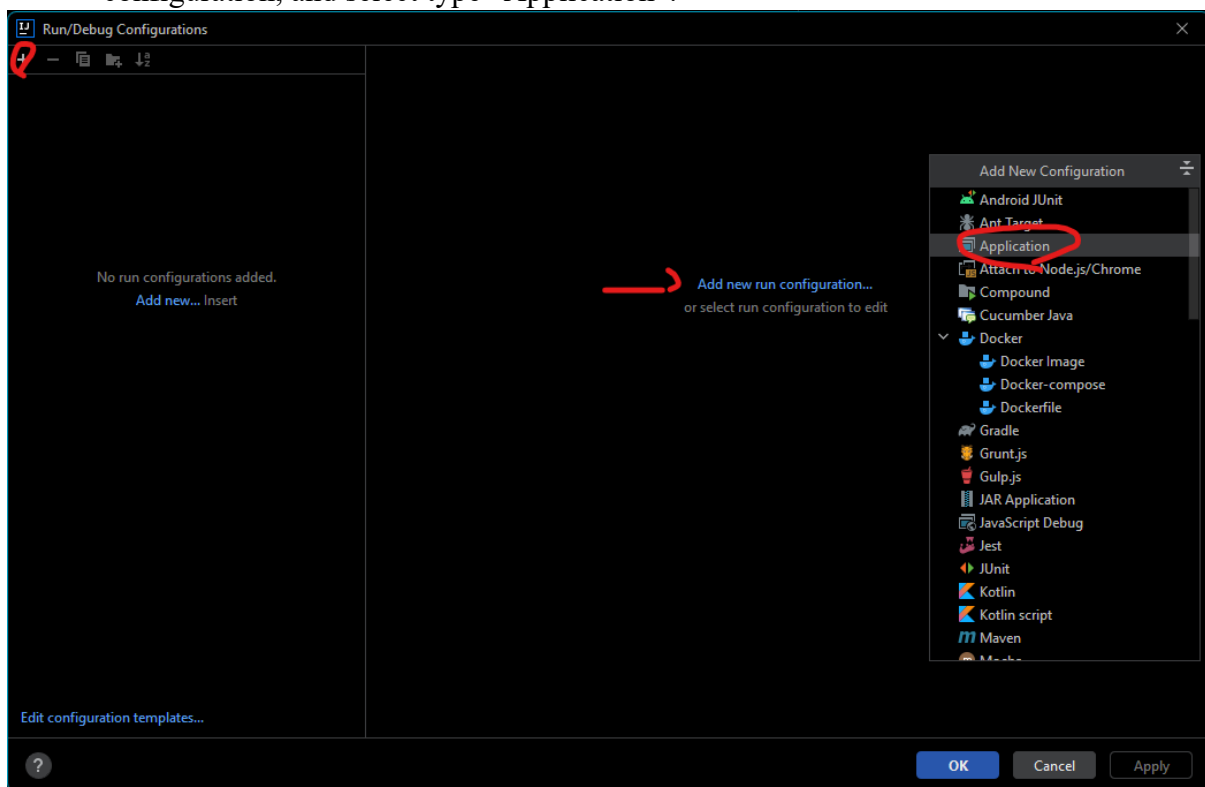4. Find the repository you just cloned, highlight it, and press "OK"
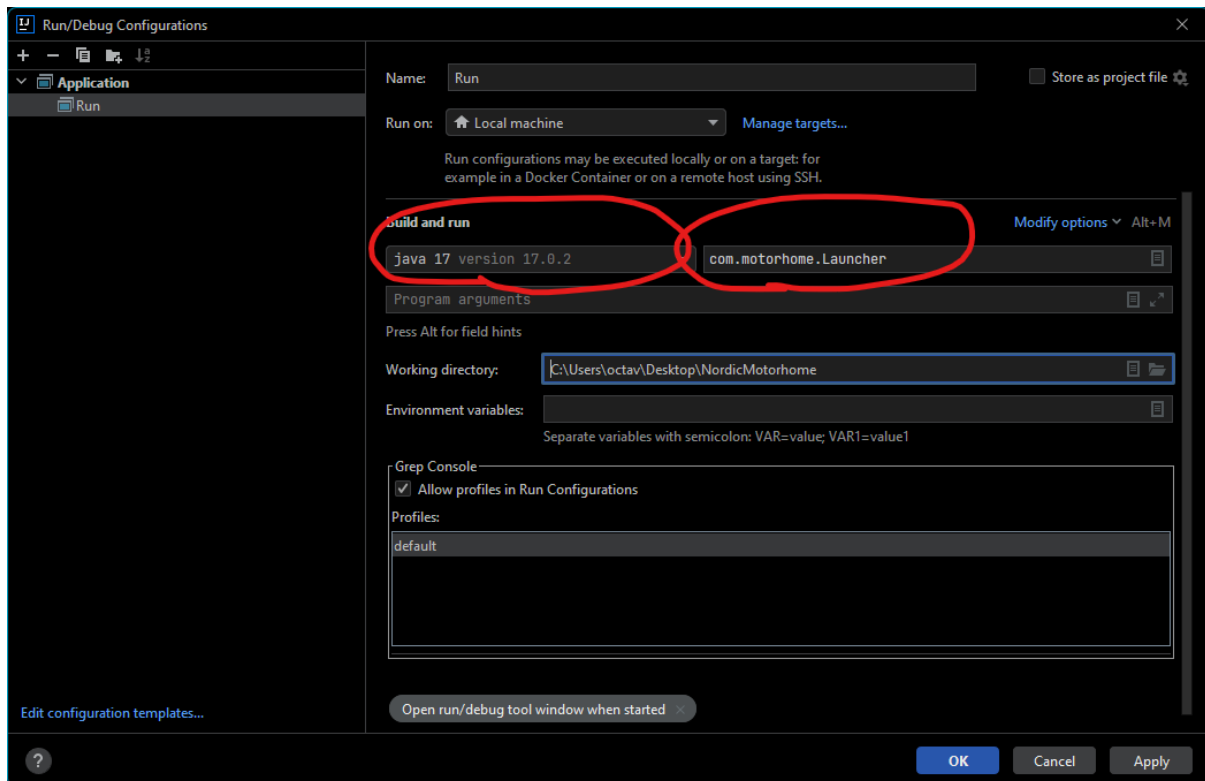

5. Select "Trust Project"

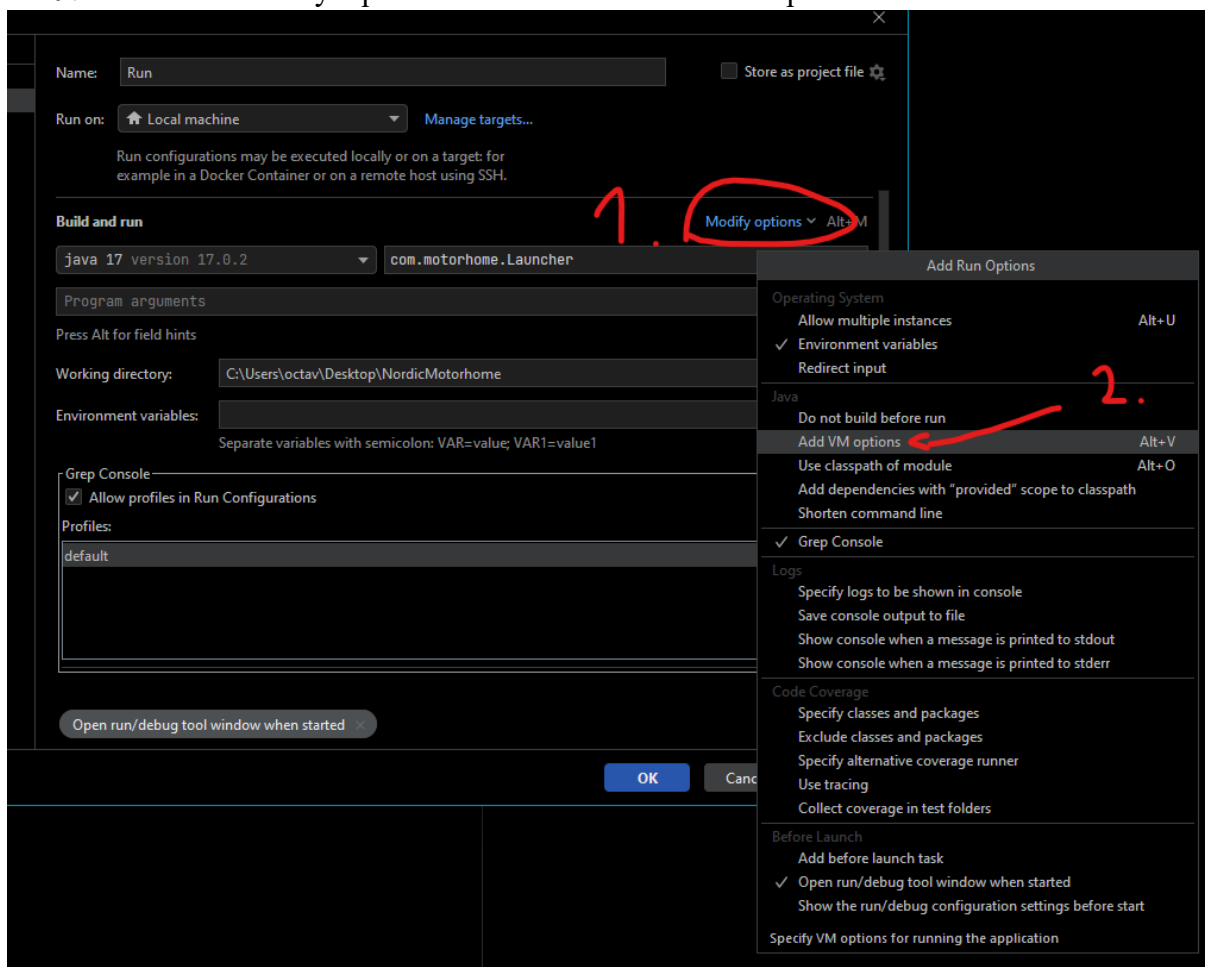6. Wait a bit until the project loads. Then, select "Add configuration at the top-right of the window:
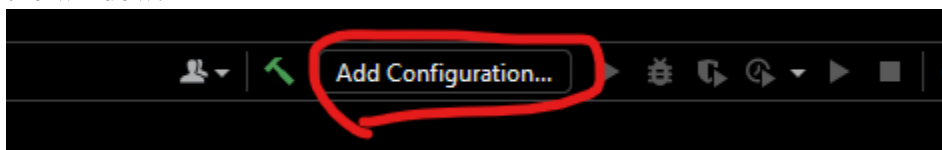
7. Click on the "plus" button or on the "Add new run configuration" link to create a new configuration, and select type "Application":
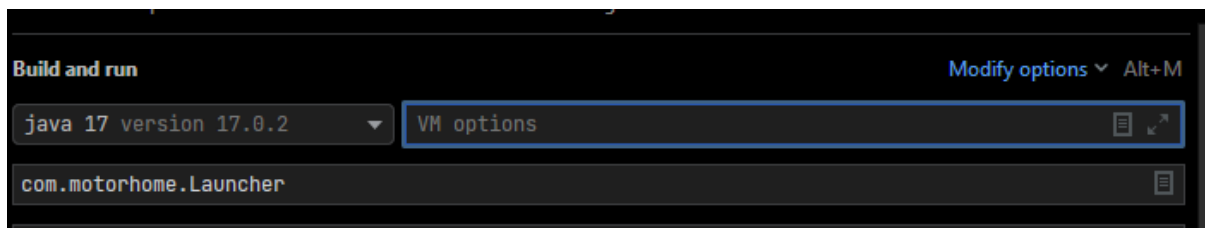

8. Select a JDK (17+ recommended), and write *com.motorhome.Launcher* as the main class:
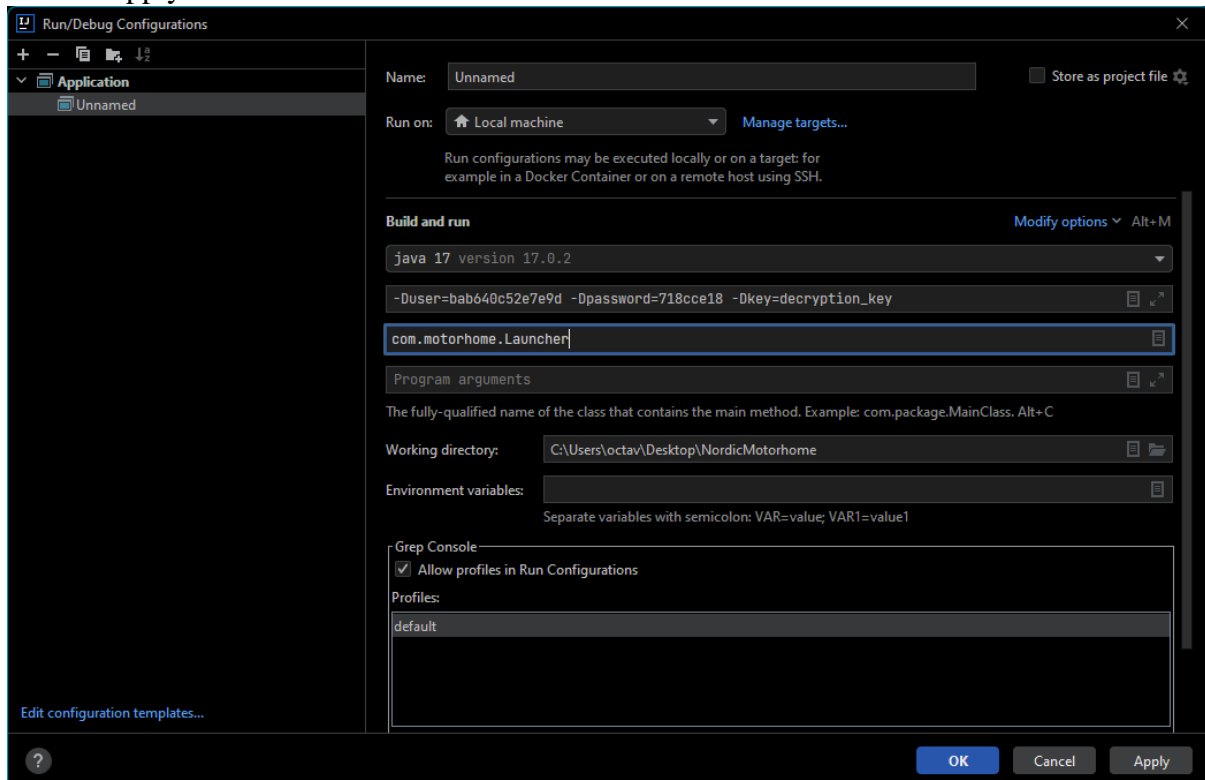
9. Click on "Modify Options" and click on "Add VM Options"

10. Copy and paste this line into the new field that appears for VM options:

-Duser=bab640c52e7e9d -Dpassword=718cce18 -Dkey=decryption_key

11. Apply and click "OK"

12. Run the configuration

If you want to try out the executables, download them from the "bin" directory in our repository. They are experimental and not all features work, therefore, use them at your own risk.

- For Windows users, just download the executable file and run it. You will probably get a warning, click on the blue text and on "Run anyway"
- For macOS or Linux, download the JAR, place it somewhere on your system, navigate to that path with your terminal, and run the following command:

java -Duser=bab640c52e7e9d -Dpassword=718cce18 -Dkey=decryption_key -jar motorhome.jar

Disclaimer: We are aware that sharing the password and decryption key is unsafe; we're doing it to ensure the project will run for the sake of this assignment. In a real-world solution, we would have to handle these passwords differently.

8 Glossary

- UP – Unified Process: an iterative and incremental software development process framework.
- UI – User Interface: space where interactions between users and the application occur
- GUI – Graphical User Interface: User interface composed by icons, menus and other visual representations or graphics
- Responsiveness: The ability of a GUI to adapt and maintain its functionality on multiple screen dimensions, sizes, aspect ratios, etc.
- Agile Methodology: A way to manage projects by breaking them in several phases
- GitHub: Provider of hosting for software development and Git version control
- UML – Unified Modelling Language: Modelling language that provides a standard way to visualize system designs in the field of Software Engineering
- Heroku: Cloud platform as a service
- ER – Entity Relationship (Diagram): A diagram depicting the physical structure of a database
- SSD – System Sequence Diagram: A UML diagram depicting the operations the messages that are being carried out
- POD – Plain-old-data
- JavaFX, JFX, FX – JavaFX, a platform for native Java application development
- JDBC – Java Database Connectivity
- SQL – Structured Query Language: A standardized programming language used to manage relational databases

9 Bibliography

- *(Larman, 1997, p.128)*
- *(Larman, 1997, p.229)*
- *(Larman, 1997, p.232)*

All from – Applying UML and Patterns, Craig Larman, Prentice Hall PTR, 1997

10 Appendix

- Appendix A: Reverse-engineered class diagram
- Appendix B: DDL Script to generate the MySQL database
- Appendix C: Selecting gender for new Staff member in the Staff Menu
- Appendix D: Selecting brand for model in Motorhome Menu Options
- Appendix E; Editing the extras of a rental in the Rental Menu