Lehrstuhl Softwaretechnik & Programmiersprachen
Fakultät Wirtschaftsinformatik & Angewandte Informatik

Report

SWT-SWL-B
Winter Semester 2021/22

Team Name

Dehom Melissa Pereira Gnassingbe
Student Number: 1234567
Degree Course/Semester: SoSySc/4

Patrick Willibald Haas
Student Number: 1234567
Degree Course/Semester: SoSySc/4

Aaron Joachim Hißting
Student Number: 1234567
Degree Course/Semester: SoSySc/4

Oleksandr Huba
Student Number: 1234567
Degree Course/Semester: SoSySc/4

Thomas Max Kretschmann
Student Number: 1870356
Degree Course/Semester: SoSySc/4

Sabir Mammadov
Student Number: 1980078
Degree Course/Semester: SoSySc/6

Supervisor: Prof. Dr. Gerald Lüttgen
Version: 14.11.2021

Report Guidelines

This report template is available in LaTeX from the SWT Lehrstuhl's information page on the VC. You are strongly encouraged to use this LaTeX template as it adheres to the formatting and structuring requirements.

Language

You are encouraged to write your report in English, but you may also write in German.

Format

Your report shall be printed on A4 paper, use a double-sided, single-spacing page format with reasonable margins (between 15mm to 30mm to the left and right), and use 12pt Computer Modern *or* Times *fonts. All pages shall be numbered, and all sections shall begin on the right-hand page.*

Structure & Content

Your report's structure shall be the one of this document. In particular, the report shall contain a title page, a table of contents, a list of figures, all sections and subsections of this document, a bibliography, an appendix with the final product backlog, and a signed Ehrenwörtliche Erklärung. Further appendices may be included as needed.

For each section of this report template, its approximate weighing on the report mark is provided, followed by a brief description of what is expected. Note that your report will be marked alongside your digital submission. Please discuss the expectations of your report with your supervisor, as well as the effort you should spend on the report versus the digital submission, if applicable.

Expected Length

The report shall be 30–50 pages of text *in length. This excludes the title page, the table of contents, the table of figures, the bibliography, all appendices and the Ehrenwörtliche Erklärung, as well as all figures, diagrams and code excerpts/listings.*

Figures & Diagrams

Each figure, diagram or code excerpt/listing/table shall be easily readable and have a number and caption that also appears in the list of figures/tables. See Figure 1 and Table 1 as examples.

Figure 1: Example figure.

Table 1: Example table

Section number	1	2	3	4	5	6
Expected. no. of pages	2–3	6–12	5–8	10–15	4–7	3–5

References

Citations shall be marked in square brackets by an alphanumeric author-year system, e.g., [SB01, Coh04] and [Knu84]. Make sure that all sources are referenced properly and all bibliography entries are complete.

Ehrenwörtliche Erklärung

All team members shall sign the Ehrenwörtliche Erklärung *(Declaration of Proper Academic Conduct) on the report's last page.*

Submission

Sign the Ehrenwörtliche Erklärung (declaration of proper academic conduct) on the last page of your hardcopy report, and submit your report as instructed on the project module's VC page.

No marks will be awarded to submissions made after the deadline.

The report has to be handed in as hardcopy, stapled (not in a folder and not in any other cover; a large stapler is available in the General Office) at the General Office (Sekretariat) of the Lehrstuhl SWT (WE5/03.013; opening hours: Tue–Fri: 9:00–11:00, and open until 12:00 midday on 09 February 2022). For large reports that cannot be stapled properly, you may ask the Sekretariat to bound the report.

Please do not forget to justify in your report all technical and non-technical aspects of your team's conduct of the software development project.

Contents

List of Figures

List of Tables

1 Project Organization

Approximate weighing on mark: *10%*
Approximate expected report length: *2–3 pages of text*

1.1 Goal of the Software

Describe the goal (purpose / advantage / measurement) of the software.

1.2 Organization of the Team

Document the software development approach employed by your team and how the work was split between the team's members. For each team member, state their main responsibilities, the artefacts principally produced by them, and the overall work time (in hours) they contributed to the artefact. Use the following table:

Table 2: Distribution of work

Name	Responsibilities	Principal Artefacts	Work Time
John Doe	Design, Architecture	Architecture Diagram, Design Principles, Design Patterns	60h
Jane Doe	Implementation, Test	Statistic Visualisation Controller & Logic, JUnit Tests	45h
⋮	⋮	⋮	⋮

1.3 Project Blast-off

Describe the activities and outcomes of the project blast-off, e.g., a stakeholder map, a context diagram, a glossary, or a project risk analysis (as taught in the module SWT-FSE-B).

2 Requirement

Approximate weighing on mark: *15%*
Approximate expected report length: *6–12 pages of text*

Document and analyze the software's functional requirements, non-functional requirements and development constraints. In particular, state whether a requirement is derived from the project brief, is an assumption made by your team, or has been added by the client. You may apply any documentation and analysis technique taught in module SWT-FSE-B or from the requirements engineering literature, including techniques based on user stories, use cases and prototyping. Properly reference and justify all employed techniques.

3 Architecture & Design

Primary textual contributors.
Thomas Kretschmann

This chapter presents the architecture and the related design of SWTcamper. First, we show which architectural model we have chosen and justify our decision. Then we will explain the implementation based on this and how individual design decisions can be derived from it. In particular, we will go into the design of the database schema and the representation of each entity. Finally, the design patterns and design principles used are presented and briefly explained.

3.1 Choice of Architecture

For the architecture of SWTcamper, we opted for the *Model-View-Controller* architecture pattern, as this encapsulates the frontend from the backend and allows the graphical user interface to be changed while the backend remains the same. This is particularly interesting in view of the fact that future SWL students should provide the application with a web front end.
In contrast, the application we built runs completely locally, although one could imagine offering the database or the *Docker container* that contains it on a remote machine.
Due to the clear division of the layers belonging to the MVC pattern (easily recognizable in 2), it was also possible to clearly distribute associated tasks to team members with ease.

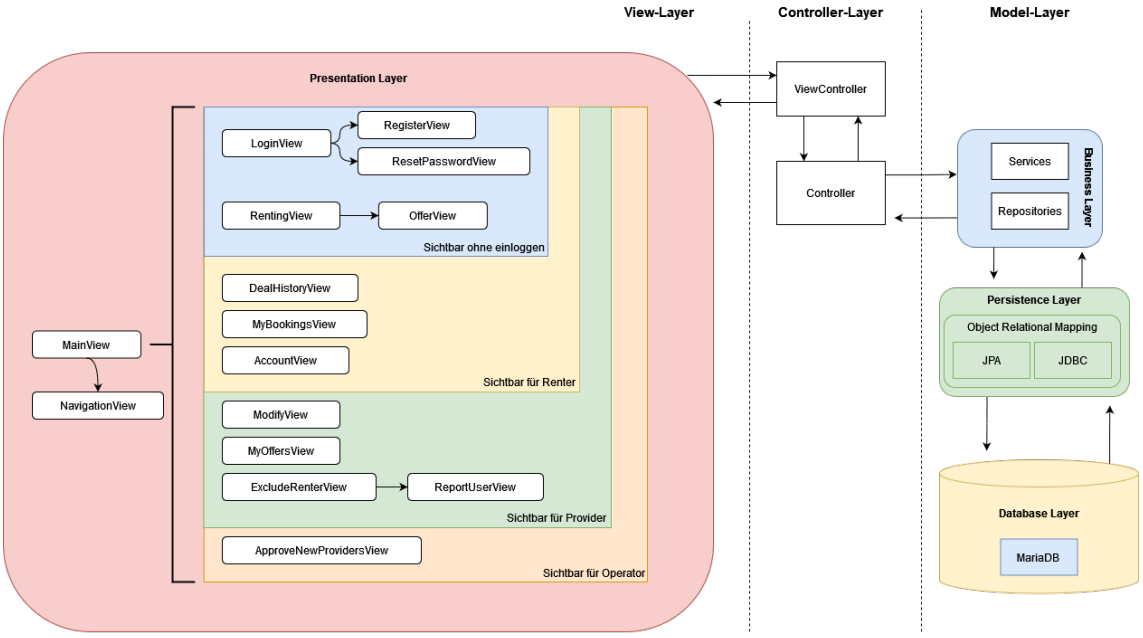

Figure 2: Architecture

3.1.1 Layered MVC in general

The Model-View-Controller pattern is an architecture or design pattern that offers flexible program design, makes it easier to change or expand it later, and allows the individual components to be reused. Applications designed according to the principles of MVC consist of three interchangeable components:

- **The model** consists of several model classes. Each represents a basic entity within the data structure used. They also provide basic data operations that are not necessarily part of the basic program logic. This includes, for example, the individual entities on the basis of which the application runs, or the services that provide the necessary values from the database and can already process them to the necessary degree.

- **View** classes provide the graphical user interface. The model classes provide the displayed data, but there is no direct connection between the two program parts. The controller informs the components of the surface display about changes to the model and they adjust the displayed content if necessary.
- **The controller** classes act as a connector between view and model components. The view forwards user actions to the controller, which executes the underlying program logic. If necessary, the logic informs individual views about changes to the model in order to enable an appropriate reaction to them.

3.1.2 JavaFX (view)

JavaFX is an open source framework used to create graphical user interfaces (GUI). In SWTcamper we used the JavaFX feature that allows to fully define the view part in FXML files. The views were only generated programmatically for dynamic program parts, like the list of available displays or the operator dashboard. The view is then created at runtime of the program using the FXML files and JavaFX. The view controllers are the controllers of the individual views, which not only take care of event handling, but also request and forward data from the 'real' controllers.

3.1.3 Spring boot (view, controller, model)

Spring Boot is an open source framework that offers many functionalities for developing a standalone application. It simplifies the software development process by reducing complexity and providing a clear structure. In our case, it is used in every layer of the MVC pattern.
In order for the framework to know how to deal with each class, there are annotations that allow Spring Boot to create the necessary environment for the classes. An example of this is the *@Service* annotation, which is placed above each of our service classes, or *@Component*, which is used above each controller.
Simply using Spring Boot doesn't guarantee quality, but it does provide a structure that makes it easier to develop quality software and avoid mistakes.

3.1.4 JPA (model)

Combined with Spring Boot, the *Java Persistence API* allows us to convert Java objects to Database objects and vice versa. Various settings can be made through annotations in the entity classes in order to implement the desired database schema automatically.

3.1.5 Docker (model)

Docker is open source software that allows us to run applications in a virtual container environment. Many software vendors already provide preconfigured images that allow their application to run in a containerized Docker environment. We used Docker to run our *MariaDB* database in it.

3.1.6 MariaDB (model)

MariaDB is an open-source relational database. We use it to store our data consistently. MariaDB also offers its own Docker image, making it a good choice for our application.

3.2 Design Implementation

In order to understand which components we need for the application and how best to proceed, we created a use case diagram (3) and an ER diagram (7) which have changed over the course of the project.

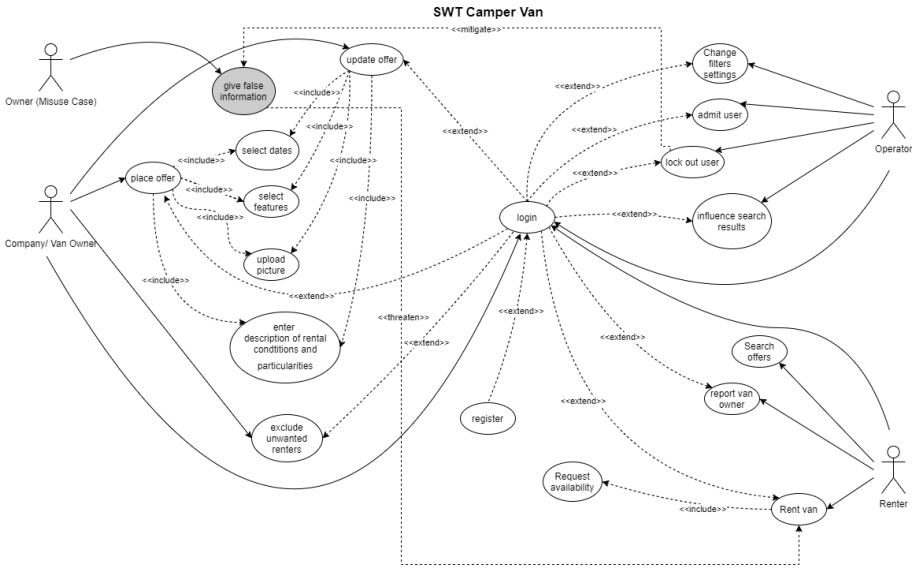

Figure 3: Use Case Diagram

Basic Structure The heart of SWTcamper is the *MainView* and its associated view controller, which is called directly from *App.java* (starting point for JavaFX applications) when the program starts. Via imports, it contains all other FXML views that are added or removed via the controller depending on the *NavigationViewController*. (trying to show in 2) When the program starts, the *RentingView* is loaded first via the *changeView()* method, which contains all available displays. To do this, the *AnchorPane* 'mainStage' in the *MainView* is first completely emptied and then filled with the *RentingView* mentioned. If the user now presses a button in the *NavigationView* (buttons are displayed depending on the *UserRole* (also taken from 2)), the *MainViewController* is instructed via its view controller to repeat the same procedure for the newly selected view. In order to differentiate between the views, a suitable string must be passed to the *changeView()* method (e.g. 'home' for the *RentingView* or 'history' for the *DealHistoryView*). Since all view controllers are already initialized when the program starts, all other view controllers must also be imported into the *MainViewController* so that a reload method can be called for each one individually.

3.3 Class diagrams

3.4 Data model & Design decisions

In order to get an overview of the entities to be implemented and how they interact, we created an ER diagram at the beginning of Sprint 0 (figure 6), which only changed slightly over the course of the project has (figure 7). In this chapter we present our entities and their structures.

Booking The Booking class is required to bind a user to an ad. To do this, it has an ID that is automatically generated by the database (through the *@Id* and *@GeneratedValue* annotations), a user who wants to rent the associated ad, the ad itself, a start date and an end date. Furthermore, there are the two boolean fields *active* and *rejected*. A booking is created with *active* = *false* and remains inactive until accepted; if it is rejected, the field *rejected*, which was also created with *false*, is set to *true*. The combination of these two values is used at several points in the application to determine states:

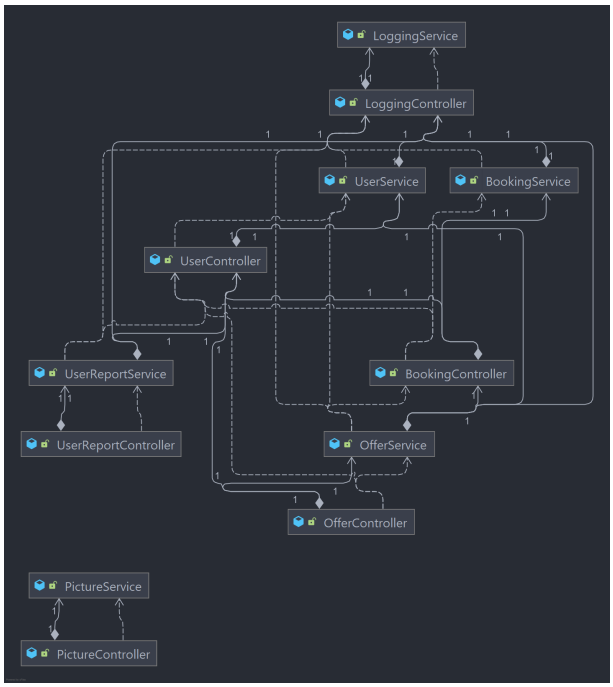

Figure 4: Class diagram of Controllers and Services

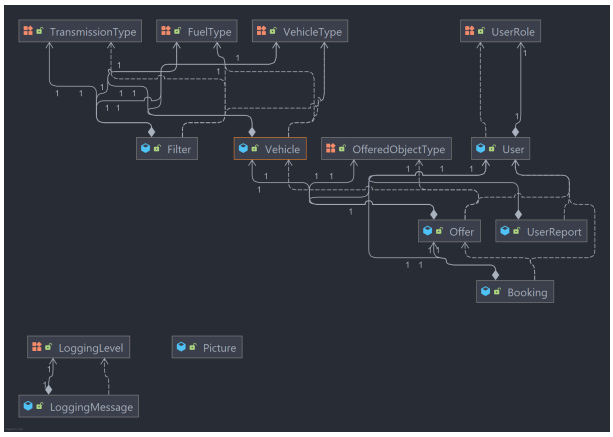

Figure 5: Class diagram of Entities

		active	
		true	false
rejected	true	Offer ended early, gets listed in offer history	/
	false	Offer can not get modified or deleted	Provider sees offer as “New request for offer” (not yet in offer history)

Filter The Filter class is the only entity class in SWTcamper that is not stored in the database. It contains all the fields from *Offer* and *Vehicle* that are required for a good search. When the search is started, a new filter instance is created and the necessary fields are set. This instance is then passed to the *OfferController*, which uses it to filter the list of all ads and returns a new list of ads that match the filter instance.

LoggingMessage Logging messages are generated by almost all services and are required to be able to track events. A list of logging messages can be viewed as an operator via the *AccountView*

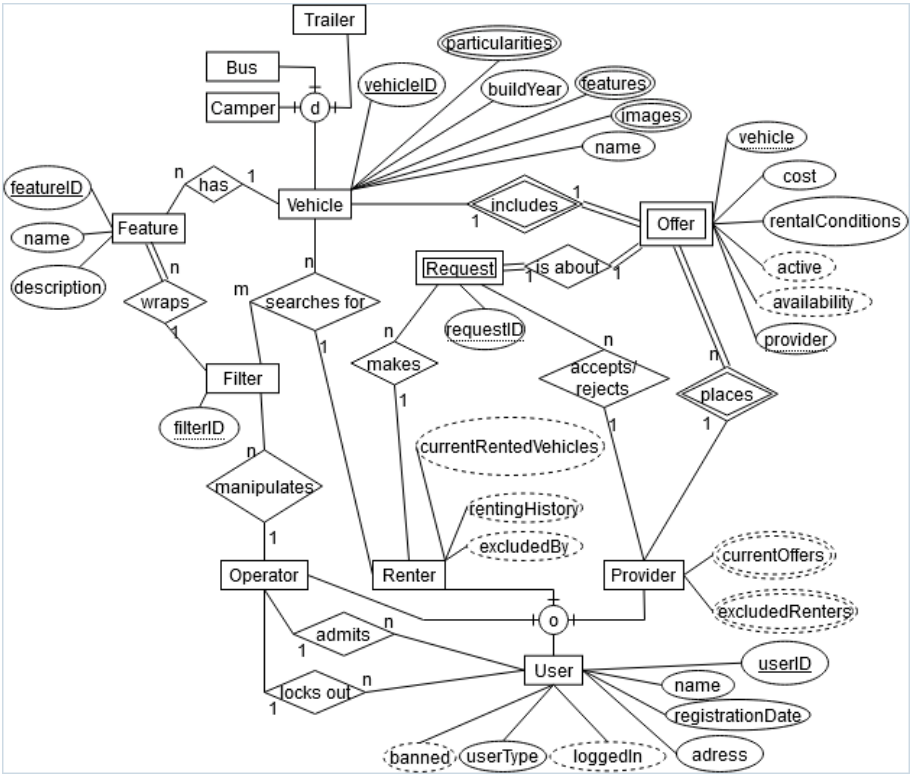

Figure 6: First draft of a Entity Relationship Diagram

and downloaded as a log file. Since all logging messages that occur are also stored in the database, each has a unique and automatically generated ID and a time stamp for which it was created. The enum *loggingLevel* indicates how important the logging message is (INFO, WARNING, ERROR) and there is also a message so that the viewer knows what it is about.

Offer The offer class was implemented first in SWTcamper together with the vehicle class. In the course of development, however, it was expanded and improved. It is needed to display offers - one of the basics of SWTcamper. Objects of this class have an ID automatically generated by the database, a user who created the offer, title, pick-up location, contact details via which the creator would like to be reached, special features of the offer (such as things that are not already covered by SWTcamper) and a price. Via *active* the creator can specify whether the offer should be publicly listed and if *promoted* is *true* the offer will be highlighted in the list; this function can only be changed by operators, but is visible to all users. Of course, each offer also contains a permanently associated vehicle object with all its values and lists for blocked dates for which the offer cannot be booked, non-overlapping bookings and rentalConditions, with which the creator specifies conditions to which the renter adheres has to hold.
The *offeredObjectType* field is no longer used in our version of SWTcamper; however, it would be possible to extend the application here.

Picture Originally it was planned that the images stored in SWTcamper for the vehicles would be completely loaded into the database; however, there were many problems finding the right data type and there were many errors that the database could not store objects of this size. So next we switched to storing a list of paths in the Vehicle object. However, this list was also too long for the database from around three images onwards.
As a solution, we have therefore created our own repository for the images. In addition to the obligatory ID, each entry contains the path to the image and the ID of the vehicle to which it is to be assigned.
Unfortunately, this approach means that images can only be stored locally and do not work in a

distributed system. However, since our application only had the requirement to run locally anyway, that was fine for this framework.

User In addition to an automatically generated ID, the User class also has attributes for the username, first and last name, email address, telephone number and password. The *userRole* attribute indicates which role the user assumes when using the application and which authorizations he has as a result (cf. view illustration in figure 2). Related to this there is also the attribute *enabled*, which is only set to *false* when a user registers as a new provider, since it was a requirement that these people should first be checked before they can create new offers. As long as a user has the role of provider and *enabled* is set to *false*, he has the same powers and options as the *userRole* Renter.
Another boolean field is *locked*, which is required to globally block a user; its application options are therefore severely limited. In contrast to this possibility, which can only be carried out by an operator, there is also one for providers with which other users can be excluded from their offers. To do this, User objects use a list *excludedRenters*, in which the IDs of the users to be excluded are stored.

UserReport UserReports are required so that one user (as of UserRole Provider) can report another. A submitted UserReport is then stored in the database (again has an automatically generated ID) and has an *active* boolean field set to *true* until the report is processed (*accepted* or *rejected*) by an operator. In order to make it easier for the decisive operator to make his decision, each UserReport contains both the username of the reporter and the reported person and a message explaining why the reporter submitted the report.

Vehicle Just like the offer class, the class for the vehicles themselves plays a very central role, which is why it was implemented first. It contains very rudimentary attributes to represent the vehicles offered; in addition to the automatically generated ID, this includes the brand, model, year of construction, length, width and height, the type of connection, the number of seats and the number of beds. Simple boolean variables were used to describe whether the vehicle has a roof tent, roof rack, bike rack, shower, toilet, kitchen unit or refrigerator. There are also separate enumerations for the type of vehicle and the type of fuel.

3.5 Design Principles

3.5.1 General Principles

Hide information Similar to what is explained in SOLID's Open-Closed-Principle (3.5.2), we have applied the 'public' and 'private' access modifiers as far as possible, so that the classes are encapsulated as much as possible. However, we really only used 'public' and 'private' and neglected e.g. 'protected', which only allows visibility within the current package. Implementing this would improve the quality of the software and could well be done in another sprint.

Loose Coupling The parts of SWTcamper's Model-, View- and Controller components are clearly seperated from each other and communicate only through interfaces.

Program to an interface, not an implementation In SWTcamper we have interfaces for our repositories for the database, for our backend controllers and for our entities. Since we didn't apply this principle from the beginning, but relatively late, we didn't build the interfaces for the entities into the rest of the code, so that they are instantiated directly and not via their interfaces (which would need casts to normal classes anyway). This task would be a good fit for another sprint and shouldn't take much time. In the meantime, the interfaces still represent a kind of 'security template'. For the first two class groups, however, we have applied it according to the principle.

High Cohesion This is pretty similar to SOLID's SRP (3.5.2); as mentioned there our classes mostly have only one purpose. Improvments could be made in the MainViewController which also includes the database scanner.

Encapsulate Logic This principle was implemented in our application primarily by the fact that the frontend controllers do not have direct access to the backend controllers, but only to the methods from the backend controllers' interfaces.

3.5.2 SOLID Design Principle

Single-Responsibility-Principle For SWTcamper we implemented contract classes and DTO objects. Through this feature we are able to display the objects of the database with additional information that can be useful to the user without modifying the objects themselves. Real changes to the objects are only carried out by saving, deleting or updating operations to ensure that we comply with them the principle of individual responsibility.
Otherwise, our classes mostly have only one purpose. An exception is the MainViewController, which, in addition to managing the other view controllers, also scans the database for changes every second.

Open-Closed-Principle By using the MVC pattern it is possible to exchange the frontend without changing the backend. Replacing the database with another one wouldn't be difficult either, as long as JPA is still able to map the objects correctly. This is made even easier by the fact that our database is made available in a container using Docker, which is easier to exchange than without. Furthermore, it would be possible to extend our existing interfaces; however, this was not explicitly valued during development. Instead, we have kept as much internals as possible private, so that classes are encapsulated from the outside as much as possible, which has worked almost completely with a few exceptions.

Liskov's Substitution-, Interface-Segregation- and Dependency-Inversion Principles Since we did not go into design principles too much when planning, we did not find any examples for these principles in our application. It is possible that SWTcamper is less suitable for this in its current form.
A possible implementation of Liskov's substitution principle would be to implement the IUser interface in the three different user classes Renter, Provider and Operator, instead of implementing it via an enumeration of UserRoles as we did. That would certainly be a point that could be improved in future sprints.

3.6 Design Patterns

3.6.1 Observer Pattern (behavioral patterns)

We would have liked to have used this pattern to implement the notifications in such a way that the database automatically sends notifications to all observers (push notification). Unfortunately, we found that MariaDB does not support such functionality. But it was possible to implement a so-called entity listener via Spring Boot annotations; This caused a thread to be started in the background, which checked for changes to the respective entity objects at regular intervals and executed the desired functionality if there was a change. However, inconveniently, in our application this caused the notification dots (the objects that the thread should change when an entity changes) to flicker in the navigation bar, so we decided to implement such a scanner ourselves, which actually worked better:
To do this, we implemented the *listenForDataBaseChanges()* method in the *MainViewController*, which fetches the necessary objects from the database and saves them in local variables. After the next fetch, it then compares these variables with the new data and executes the corresponding

functionalities if the two data sets are different differentiate. So that this is possible at regular intervals, the method was provided with *@Scheduled(fixedDelay = 1000)*, which also starts a new thread in which, in contrast to the first implementation, the time intervals can be controlled. This implementation also corresponds to the pull notification of the observer pattern.

Furthermore, we used properties in many places in the application and either bound them to others or set EventListeners on them so that dependencies adjust automatically without having to trigger the functionality separately.
The (forced by JavaFX) use of observable lists is also to be assigned to the observer pattern, since changes in the lists automatically cause a change in the objects in which they are used.

3.6.2 Singleton Pattern (creational patterns)

Since the structure of SWTcamper is designed in such a way that the *MainView* holds all other views (cf. figure 2), only one object is created at the start of the *MainViewController* and the *NavigationViewController*, which are permanently displayed and their contents are only change. However, both are also referenced in other view controllers using Spring Boot's *@Autowired*, so at this point we are not 100 percent sure how the framework handles these classes exactly.
It would also have been nice to have the database scanner explained in section 3.6.1 in its own class, so that it would only be instantiated once - independent of the *MainViewController*. However, it can be assumed that this will only happen once.

3.6.3 Factory Method (creational patterns)

We didn't use this method explicitly, but because we use the Spring Framework and provide many similar classes with the same annotations (*@Component*, *@Service*, *@Entity*, *@Repository*), they use the same interfaces and therefore also have the same standard methods , such as *initialize()*.
An explicit implementation might be possible with the ViewControllers; Due to the component-based structure of the application, however, our classes differ so much that sharing interfaces is of little use.

3.6.4 Decorator Pattern (structural patterns)

Initially, to create a new ad and to edit an ad, we had two different views and controllers. However, since the two functionalities are so similar, we have combined them in a view (and thus a controller) *ModifyView*. There is a boolean variable to distinguish between creating a new ad or editing an existing one. The decorator pattern could have been used here, which would extend the class with additional functionality at runtime. However, it is questionable whether this would bring a real advantage given the size of our application and the class.

3.6.5 Pipes and Filters (structural patterns)

We used functional programming in SWTcamper in several places, especially when processing lists (e.g. for filters), and lined up the methods so that the output of the last processing step was always the input for the next.

3.7 Summary

In SWTcamper we have applied many of the design principles we are familiar with - partly consciously and partly unconsciously. In some cases, we only implemented them when writing this report, after hearing from them again. Overall, we could definitely implement more with more time and also with higher accuracy. In addition, it would also be good in the future to go into the principles more at the beginning of the architecture and not only later on, but in retrospect we neglected that a bit too much.

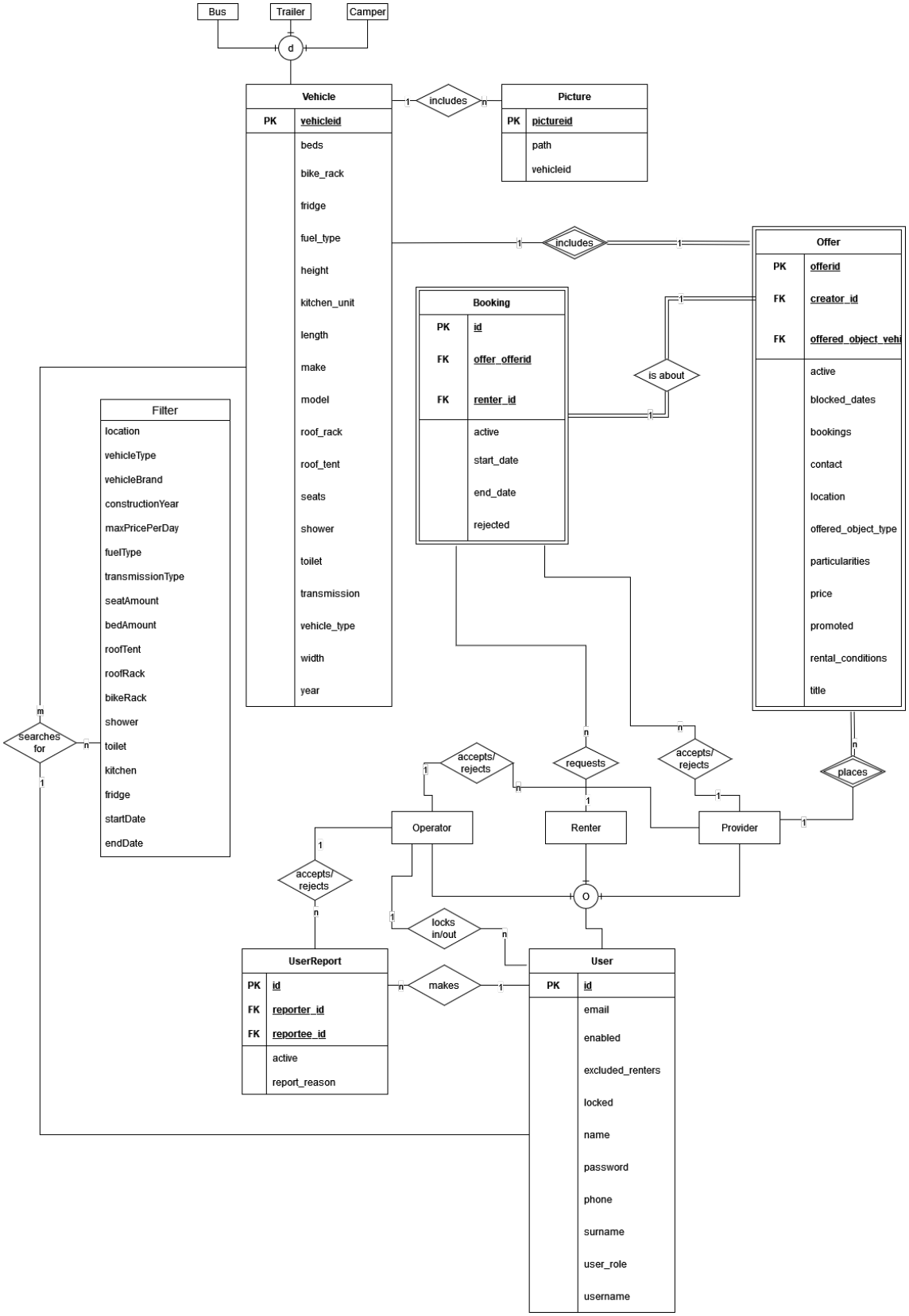

Figure 7: Our final Entity Relationship Diagram

4 Realisation

4.1 Sprint Overview

Approximate weighing on mark: *25%*
Approximate expected report length: $\frac{1}{2}$ *page of text*

Give a brief overview of each sprint, including the sprint's underlying vision.

4.2 Sprint No. 1

4.2.1 Sprint Planning

Goal Our officially stated goal for this sprint was:
In the next two weeks ("Sprint 1"), you should correct the artifacts from Sprint 0 with respect to the feedback from Prof. Lüttgen and start with implementing the first few User Stories. To do this into the xTasks project, the team first (?) needs to clean it up.

But with the experience we have after sprint 5 we would rather formulate it like this (shorter and on point):
Implement the possibility for a provider to place new and update existing offers such that they can be seen in the application.

Tasks that have been worked on

All-Offer View	User Story 3	Patrick
Update vehicle Database Connector	User Story 3	Sabir
Database Connector	User Story 2	Aaron
View Controller	User Story 2	Oleksandr
Create/Update/Delete Offer	User Story 3	Patrick/ Thomas
Lo-Fi Prototype		Oleksandr
Prioritize User Stories & Issues		Melissa
Update Vehicle View	User Story 3	Thomas
Update Vehicle View Controller	User Story 3	Thomas
Vehicle Controller	User Story 2	Aaron
Vehicle class	User Story 2	Melissa
Offer class	User Story 2	Melissa
View Place Offer	User Story 2	Patrick
Fix Architecture Diagram		Patrick
Project Plan		Melissa
Clear Project to Skeleton		Sabir
Fix Use Case Diagram		Aaron
Create dev + production branch		Sabir
Acceptance Criteria & INVEST Review		Oleksandr
Review Acceptance Criteria		Thomas

4.2.2 Noteworthy Development Aspects

In this sprint we have broken down the functionalities far too much into their individual parts (e.g. each controller or view individually named); this meant that team members could not work independently on their issues because they had to wait for others who had the same problem themselves. This 'deadlock-like' condition was also an important part of our retrospective to avoid this mistake in future sprints.
You can also see that in addition to the user stories, work was also carried out on issues that were either improvements from Sprint 0 or that are important for the project itself (such as the design of a prototype or the redesign of our project plan).
The move from the *xTasks* skeleton to *swtcamper*, as already mentioned in the (now old) Sprint Goal, has also been processed.
Unfortunately, the order in which we carried out the merge requests afterwards led to serious problems that we will also take with us into the next sprint and work on there. The better solution probably would have been to carry out all merge requests first and only then to move to *swtcamper*. We did it the other way around.

Problems that occured and how we solved them

- Docker did not run on Windows 7 → the affected team member worked on other tasks first and bought a new PC by now
- *MainController* cannot be implemented without the other Controllers → good and early conversation in the team made it possible anyway
- IntelliJ had problems with JavaFX → the help desk found out that this particular error came from the use of a wrong sub folder
- Correct usage of branches → a team member gave the others a quick introduction into Gitlab's 'Create Merge Request' from within the issue itself and also showed how to work on Merge Requests
- how to upload pictures and save pictures since the database complained about the size of the stored object? → after consultation with the client it was ok to not upload the pictures but just save their paths
- IntelliJ shows Class as not existent - even if it is → This problem is not fixed until today but since it's only from IntelliJ's linter it works anyway, but stays annoying
- 'Merge Hell' – we should have done 'clear codebase to skeleton' after merging everything, not started with everything else and merge it into new */swtcamper/* → As already stated, this problem came with us into Sprint 2 and had to be worked on further, but we could solve it and avoided it from there on mostly

4.2.3 Sprint Review

Describe the product increment produced in this sprint When starting the application, only two of the visible tabs can be clicked on and used: on the one hand, 'Login' and on the other hand, 'Rent a Camper'. The latter has not yet been implemented and the login is implemented more demonstratively than functional: there is only one 'Login' button which, after clicking on it, activates the other tabs 'My Offers' and 'Place Offer'.
All offers that have already been placed can be viewed in a list under 'My Offers'. Above that are the buttons 'Place Offer', 'View Offer', 'Update Offer' and 'Delete Offer'.

Via 'Place Offer' the user is automatically taken to the tab of the same name. Here he can enter information on the offer (title, price per day, location, availability) and on the vehicle itself (vehicle type, make, model, year of construction). For the latter, the user can also select features and upload images. At this point in time, only the title and price per day can be saved in the increment.
Using the 'Place Offer' button, the user can then return to 'My Offers' where the new offer is listed accordingly.
To change the offer, you can click on the 'Update Offer' button, which activates the tab of the same name and takes you to it. The same form can be seen here as on 'Place Offer', only this time the relevant fields are already filled out and ready to be changed. Once this is done, the 'Update Offer' button can be clicked, which brings the user back to 'My Offers'. The 'Update Offer' tab is now also deactivated again.
Via the 'View Offer' button, the user receives an info alert at this point in time, in which the relevant information (including the ID from the database) can be seen. The same function is also hidden behind a double-click on an offer.
With the last button 'Delete Offer', the selected offer is deleted from the database after a request in the form of a warning alert. If no offer is selected and the user presses one of the buttons, he will also receive an alert with the instruction to select an offer first.

Compare the achieved increment with the sprint goal and the user stories that were chosen for this sprint The sprint goal was achieved (for the most part) since new offers could be created

(and saved in the database), displayed in a list, updated and deleted. In addition, our Blast Off was updated (w.r.t. the feedback we got from the last Review Meeting).
The only thing that has not yet been fully achieved is the move to swtcamper, or has not yet been fully merged with the functionalities.

Give a brief summary on your team's retrospective, including changes to the product backlog
We learned from the customer that our menu navigation in the prototype is still too complicated (too few links), that a user should be able to have several roles at the same time and that an offer should not be deletable if it is already rented out (or communicate that to the User somehow).
We learned from the client that our sprint goal should direct more joy to the customer (which we tried to make better at the beginning of this chapter) and that quality assurance is not only possible via code, but can and should also be applied through tests/ methods, such as usability testing, before implementation.

4.3 Sprint No. 2

Approximate expected report length: *2–3 pages of text*

4.3.1 Sprint Planning

4.3.2 Noteworthy Development Aspects

4.3.3 Sprint Review

4.4 Sprint No. 3

Approximate expected report length: *2–3 pages of text*

4.4.1 Sprint Planning

4.4.2 Noteworthy Development Aspects

4.4.3 Sprint Review

4.5 Sprint No. 4

Approximate expected report length: *2–3 pages of text*

4.5.1 Sprint Planning

4.5.2 Noteworthy Development Aspects

4.5.3 Sprint Review

4.6 Sprint No. 5

Approximate expected report length: *2–3 pages of text*

4.6.1 Sprint Planning

4.6.2 Noteworthy Development Aspects

4.6.3 Sprint Review

5 Quality Assurance

Approximate weighing on mark: *15%*
Approximate expected report length: *4–7 pages of text*

Describe and justify the different quality assurance techniques that your team has applied alongside the project's conduct, including the INVEST criteria for the user stories, SMART criteria for the tasks derived from user stories, unit tests for your code, normal form criteria for the database schema, and others. Illustrate your approach to quality assurance by giving relevant examples for each employed technique. Finally, do not forget to evaluate your software's interfaces (including the GUI) and the data model.

6 Project Review

Approximate weighing on mark: *10%*
Approximate expected report length: *3–5 pages of text*

6.1 Development Process

How well did your team's development process work, and why? Did the process change between sprints? In addition, compare and contrast the SCRUM process as practised by your team to (i) 'the' textbook SCRUM process [SB01] and (ii) the other software development processes presented in module SWT-FSE-B. Could your team's development process be improved, and by which means?

6.2 Team Work

How well did your team work together? Was the distribution of work and the communication among team members effective? Was the communication with the client effective?

6.3 Lessons Learned

What would you change if you could re-start the project, regarding the employed techniques, the conduct of the project and any other matters that you consider relevant? What should stay the same?

References

[Coh04] M. Cohn. *User Stories Applied: For Agile Software Development.* Addison-Wesley, 2004.

[Knu84] D. E. Knuth. *The TEXbook.* Addison-Wesley, 1984.

[SB01] K. Schwaber and M. Beedle. *Agile Software Development with Scrum.* Prentice-Hall, 2001.

A Product Backlog

Primary textual contributors.
Sabir Mammadov

This section provides information about the user stories that have been completed in each sprint, as well as those not completed and others.

A.1 Stories Completed in Sprint 1

There was no finished story in this sprint.

A.2 Stories Completed in Sprint 2

A.2.1 Update Offer

As a Provider,
I want update an existing offer,
so that the information is up to date.

Meta-Information:

- Size: M
- Sprint: 1

Acceptance Criteria:

The story is done, when

- The provider can enter an "Edit"-Mode by clicking a button on the offer screen, which is only shown to provider accounts
- The edit button is only shown on offers belonging to the respective provider
- After editing the offer is finished, the database entry of the offer gets updated
- Providers can access the list of their offers
- Provider can delete the offer

A.3 Stories Completed in Sprint 3

A.3.1 Login

As a User,
I want login by supplying my email and my password,
so that I can rent/offer vans or administrate portal.

Meta-Information:

- Size: L
- Sprint: 3

Acceptance Criteria:

The story is done, when

- it is impossible to use functionality of portal (other than just looking through offers) without being logged in
- wrong username and/or password does not give access (user is informed if password or username is wrong)

- a user can change their password in case they've forgotten it and login with the new password
- a Renter only sees the renter GUI
- a Provider only sees the Provider GUI
- a Operator only sees the Operator GUI
- it is impossible to use provider functionalities logged in as a provider if the operator has not admitted the registration yet

A.3.2 Registration

As a User,
I want register by supplying my email and by creation my personal password plus username,
so that I can create a profile at the portal for rent/offer van activities.

Meta-Information

- Size: L
- Sprint: 3

Acceptance Criteria

The story is done, when

- only unique emails and usernames are accepted,
- only valid usernames and emails are accepted,
- only passwords with at least 5 symbols are accepted; otherwise "Password is too short" is displayed
- only usernames with at least 5 symbols are accepted; otherwise "Username is too short" is displayed
- after successful registration it is possible for the user to login in the portal
- after successful registration the user is saved in the database

A.3.3 Request

As a Renter,
I want to be able to make a request,
so that I know if the wanted vehicle is available.

Meta-Information:

- Size: M
- Sprint: 3

Acceptance Criteria:

The story is done, when

- Renter can send a request for booking
- New booking is added to the list of all bookings of the renter

A.3.4 Booking

As a Renter,
I wantto be able to make a booking,
so thatI can get the wanted vehicle

Meta-Information:

- Size: S
- Sprint: 3

Acceptance Criteria:

The story is done, when

- I get a booking confirmation

A.4 Stories Completed in Sprint 4

A.4.1 Place offer

As a Provider,
I want place an offer,
so that my vehicles can be booked by renters.

Meta-Information:

- Size: M
- Sprint: 4

Acceptance Criteria:

The story is done, when

- it is possible to enter info about a vehicle (images, features, dates when it is available, description of rental conditions, particularities) into a form, which will then be considered an offer by the system
- the offer/vehicle is entered into the database
- each offer has obligatory information. If this information is not filled-in provider cannot create the offer

A.4.2 Filter

As a Renter,
I want to filter vehicles,
so that I can find fitting vehicles easier.

Meta-Information:

- Size: M
- Sprint: 4

Acceptance Criteria:

The story is done, when

- Renter can filter for vehicle's features
- Renter can filter for availability

- Renter can filter for rental costs
- Renter can filter for rental conditions
- Returned search list is small and clearly arranged

A.4.3 Approve the booking

As a Provider,
I want approve the booking,
so that I can make the deal on the portal properly.

Meta-Information:

- Size: M
- Sprint: 4

Acceptance Criteria:

The story is done, when

- Provider receive request for renting her/his van
- Provider can agree or decline the request
- Request contains all mandatory information: date for rent, renter name and rating, amount of money etc.
- After acceptance of the request, new deal is added to the list of deals of provider

A.4.4 Booking history

As a Renter,
I want have a booking history,
so that I can track my bookings.

Meta-Information:

- Size: M
- Sprint: 4

Acceptance Criteria:

The story is done, when

- Renter can access a list of all bookings that they have including finished and active
- Renter can access each offer from past and check all info about booking

A.4.5 (Un)block provider or renter

As an Operator,
I want exclude/include back provider or renter,
so that I can administrate the portal and avoid negative deals on the portal.

Meta-Information:

- Size: M
- Sprint: 4

Acceptance Criteria:

The story is done, when

- Operator can block provider or renter after the request of the corresponding supplicant who has written a detailed complaint about the previous deal.
- Operator can unblock provider or renter.
- It is impossible for a blocked user to make more deals.

A.4.6 Exclude renters

As a Provider,
I want to be able to exclude renters,
so that I can avoid renters with a bad history of renting.

Meta-Information:

- Size: M
- Sprint: 4

Acceptance Criteria:

The story is done, when

- It is impossible for excluded renters to rent vehicles from the provider who excluded them
- It is possible to revoke the "ban" for the provider
- It is impossible to exclude an account that is already excluded
- Booking functions are not available for excluded renters

A.4.7 Operator's Influence

As an Operator,
I want be able to influence search results,
so that I can make my own vehicles more visible.

Meta-Information:

- Size: L
- Sprint: 4

Acceptance Criteria:

The story is done, when

- Operator have a list of all offers, placed on the portal.
- Operator can access each offer and overview the overall info about the deal.
- Each offer has adjustable attribute ranking named “promoted offer”.
- Operator can mark offer as “promoted offer” via a button
- After submission, the changes must be saved.
- Offers that are marked as “promoted offer” are visually highlighted in the offer list.

A.5 Stories Completed in Sprint 5

A.5.1 FAQ

As a User,
I want to read the FAQ,
so that I can follow the rules of the portal.

Meta-Information:

- Size: S
- Sprint: 5

Acceptance Criteria:

The story is done, when

- user can access the FAQ section from the main menu
- FAQ consists of all necessary info regarding rent/offer
- FAQ section shall be available for all types of users, even for those who are logged out

A.6 Not Completed Stories

A.6.1 Configure options

As an Operator,
I want configure options like for searching and filtering,
so that I can adjust them to changing needs.

Meta-Information:

- Size: L
- Sprint:

Acceptance Criteria:

The story is done, when

- Operator can access the list of all filters for search options.
- Operator can add a new filter to this list.
- Operator can delete a filter from this list.
- Operator can save the changes after the job is done.

A.7 Other Stories

B Additional Material

If needed, insert any additional material, e.g., larger diagrams or longer excerpts of source code, in this and possibly further appendices. Properly reference all appendices from the report's main part.

Ehrenwörtliche Erklärung

Alle Unterzeichner erklären hiermit, dass sie die vorliegende Arbeit (bestehend aus dem Projektbericht sowie den separat abgelieferten digitalen Werkbestandteilen) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

	Full name of Student 1
Matrikelnummer	Name
Ort, Datum	Unterschrift

	Full name of Student 2
Matrikelnummer	Name
Ort, Datum	Unterschrift

	Full name of Student 3
Matrikelnummer	Name
Ort, Datum	Unterschrift

	Full name of Student 4
Matrikelnummer	Name
Ort, Datum	Unterschrift

Full name of Student 5	
Matrikelnummer	Name
Ort, Datum	Unterschrift

Full name of Student 6	
Matrikelnummer	Name
Ort, Datum	Unterschrift