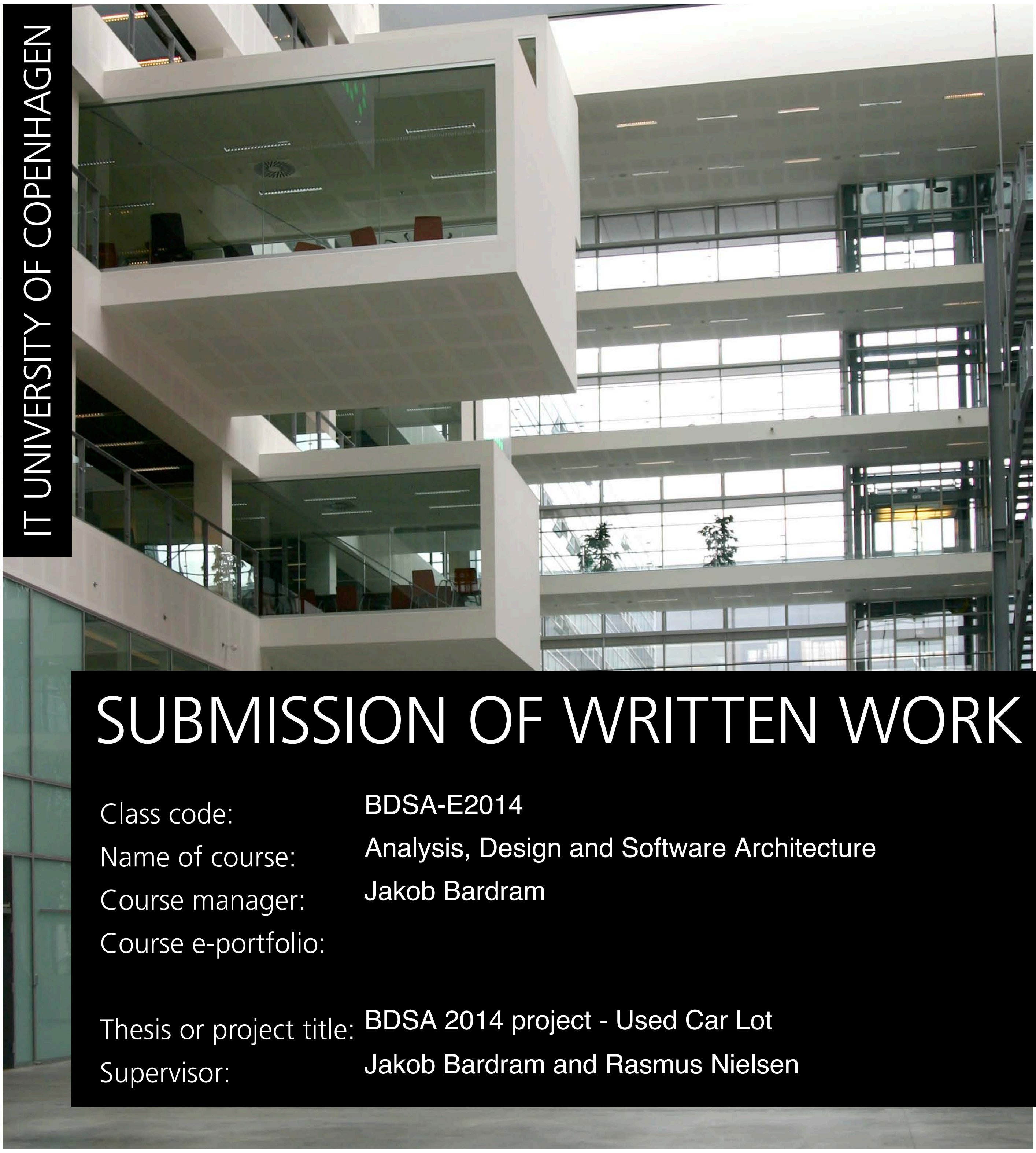


SUBMISSION OF WRITTEN WORK

Class code:	BDSA-E2014
Name of course:	Analysis, Design and Software Architecture
Course manager:	Jakob Bardram
Course e-portfolio:	
Thesis or project title:	BDSA 2014 project - Used Car Lot
Supervisor:	Jakob Bardram and Rasmus Nielsen

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Fischer-Nielsen	06/05-93	afin@itu.dk
2.	Anders Wind Steffensen	10/02/93	awis@itu.dk
3.	Christopher Niklas Blundell	10/08-94	cnbl@itu.dk
4.	Jacob Stenum Czepluch	10/10-90	jstc@itu.dk
5.	Mikael Lindemann Jepsen	17/02-92	mlin@itu.dk
6.	Pierre Popino Mandas	28/10-94	ppma@itu.dk
7.			@itu.dk

IT UNIVERSITY OF COPENHAGEN

BDSA 2014

Requirements Analysis, Software Design, and Test Documentation

DRIVEIT

Anders Fischer-Nielsen - afin@itu.dk
Anders Wind Steffensen - awis@itu.dk
Christopher Blundell - cnbl@itu.dk
Jacob Stenum Czepluch - jstc@itu.dk
Mikael Lindemann Jepsen - mlin@itu.dk
Pierre Mandas - ppma@itu.dk

December 17, 2014

RevisionHistory

Date	Ver.	Description	Ref.	Author
17/11-14	0.1	Initial draft. Setting up document structure		awis
25/11-14	0.2	Initial design and analysis in RAD and SDD.		All
05/12-14	0.3	Updates to sub system decomp. & persis. storage.		afin
12/12-14	0.4	Updates to SCRUM, testing & ODD.		afin & awis
15/12-14	0.5	Extended every chapter, added missing chapters.		All
16/12-14	0.6	Completed the final read-through and editing of grammar.		All
17/12-14	1.0	Finished the last bits of everything.		All

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Purpose of the System

The system must provide a way to manage sales of used cars for a used car dealership.

The used car lot has a number of employees that need to register cars, sales and customers. Potential customers should be able to view cars and notify the company of their interest in a car.

This functionality must be implemented using a Windows application and a web client. The system must also support persistent saving of data in the system.

1.2 Scope of the System

The scope of the system is to deliver a functional piece of software which allows employees at a used car dealership to fulfil their daily tasks managing cars, customers, sales, and other employees.

The system must support creating, reading, updating, and deleting data types of the car dealership provided the user of the given sub system has the right authorisation. These include used cars, customers, employees, sales, comments on cars, and requests for contact from an interested customer. Cars can have several pictures attached to them which should be available to users of the system. The data of the system should be stored persistently in a hosted database.

1.3 Non-Scope of the System

It is not in the scope of this system to provide a fully tested, error-free application. Developing a system without any errors is therefore outside the scope of this project.

Nor is it in the scope to have all business logic complete. The system should have room for expansion and should therefore not be seen as a completely finished product, but more as a nearly finished release.

Some functionality will be shown, but might not be functional. This functionality will then be achievable to implement in a future release. This also means that any handling of receipts and transactions is not supported.

The system should not support remote patching by the developers. If an update for the system is released, it should not be required that users are able to download and install this patch in the system automatically, and a reinstall might be required.

The system will not support users other than the ones specified in this document - the employees and customers at the used car lot. The system is not supposed to be designed for general customer and sales management.

1.4 Objectives and Success Criteria of the Project

The main objective of the project is to fulfil the previously mentioned scope of the system. The system must support the users' daily tasks in an efficient manner.

The project can be deemed a success when every use case has been fulfilled inside the given requirements detailed in this document.

Another success criteria is having functional tests of the system within the limits of the scope. The development of the system must use the agile *SCRUM* software development methodology.

Part I

Requirement Analysis Document

CHAPTER 2

Proposed System

2.1 Overview

This *Requirement Analysis Document* describes the requirements of the system, from now on referred to as the `DriveIT System`. This document describes the following:

- General Requirements
- Initial Analysis Objects
- Target Environments
- Nonfunctional Requirements
- Scenarios
- Use Cases
- Domain Object Models
- Dynamic Models

This *Requirement Analysis Document* will elaborate on what requirements and required functionality the `DriveIT System` must support.

2.2 Functional requirements

General Functional Requirements

- The system must be able to support different user roles, more specifically a customer-, employee- and administrator role.
- The Windows client must only be accessible to employees and administrators.

- It must be possible for customers to browse cars with different filters and criteria.
- A customer must be able to create and sign in to a customer account on the web client.
- A customer must be able to find contact information for employees without signing in.
- A customer must be able to request contact from an employee, if the customer is signed into the system.
- A customer must be able to comment on the cars in the system, if signed in.
- A customer must be able to edit contact information for his/her account.
- An employee must be able create, read, update, and delete all cars in the system.
- It must be possible for an employee to create, read and delete several images of a car.
- An employee must be able to read, update and delete a list of requests from customers wanting to be contacted regarding a car.
- An employee must be able to create a account on behalf of a customer calling the dealership without an existing customer account.
- An employee must be able to create, read, update and delete all sales of a car.
- A customer must be able to see all cars he has purchased.
- An administrator must be able to create, read, update, and delete employees.
- It must be possible to browse through the images of a car.
- An employee must be able to get more information about a car using the *Car Query API*[1].
- A sold car must be visible in the system for another week after it has been sold.

[1] `http://www.carqueryapi.com/`

- A customer must be able to create, read, and delete all contact requests he/she has made.
- Profile pictures of customers and employees should be provided using their *Gravatar*[2] avatar.

The group have chosen to design and develop a comment system instead of a rating system for cars.

Optional Functional Requirements

The following optional requirements must be fulfilled.

- The system must be deployed/hosted on Microsoft Azure.
- The system must supports showing several images of a car in an image gallery.
- (*Partially*) The system must have a form for a customer to request contact by an employee or let the customer contact the dealership lot by clicking a car.

2.3 Initial Analysis Objects

- **Customer:**
  A customer is a person interested in buying a car. The customer can only use the web client. A customer object will hold contact information about a given customer e.g. name, email, and phone.
- **Employee:**
  An employee is a person working for the used car dealership. The employee uses the Windows client. The employee object will hold information about a given employee e.g. name, email, job title, and phone. Employees can be administrators, meaning that they are able to create, update, and delete employees.
- **Car:**
  The car object is a car for sale at the used car dealership. The car object will hold relevant information about itself e.g. manufacturer, model, year, colour, price, etc. The object also has an attribute indicating whether it is sold or not.

[2] `https://en.gravatar.com/`

- **Sale:**
  A sale consists of a car, an employee, and a customer. It uses some different information from each of these objects to make a sale. From the customer's perspective, it is called an order.
- **Comment:**
  A comment consists of a customer, a title and a description. A car object can have several comments.
- **Contact Request:**
  A contact request will be created when a customer wants to be contacted by an employee regarding a given car. It contains the car in question, the requesting customer and an optional employee when the request is being dealt with.

2.4 Target Environments

Different parts of the `DriveIT System` will run in separate environments:

- The `DriveIT Web API` and `DriveIT Web Client` will be hosted at *Microsoft Azure*.
- The `DriveIT Windows Client` will run on a Windows computer inside the used car dealership, which has the *.NET Framework 4.5* installed.

2.5 Non-Functional Requirements

Usability

The `DriveIT Windows Client` must use the *Design Guidelines of Microsoft*[3] such that it is intuitive for users who are used to the Windows platform.

Furthermore the UI must be minimalistic and the most frequently used features should be placed in view no further than two mouse clicks away from the start-up page. Some training of employees may be required prior to using the client.

[3]`http://msdn.microsoft.com/library/windows/apps/hh781237.aspx`

The `DriveIT Web Client` should follow modern mobile-first website design thus making the site intuitive and easy to use for everyday users. Furthermore simplicity should be paramount, and viewing information about cars should be quick and linear (e.g. not interrupt the users current task).

Reliability

For the `DriveIT Web Client` and `DriveIT Web API` we use *Microsoft Azure* to deploy the system. Due to this, we are not responsible for server maintenance and down time.
Microsoft Azure is a reliable and fast deployment environment, saving development from the troubles of setting up and hosting a deployment and server environment.

The three systems should have sufficient exception handling and not crash the entire system, but instead provide an error message.

Performance

The system must be responsive and fast to use. Most user actions must not freeze the user interface, therefore giving the UI a responsive feel. The start-up time for the `DriveIT Windows Client` must be less than 3 seconds for a modern computer.

Furthermore the `DriveIT Web Client` must be fast to use and all pages should be loaded in less than 2 seconds for a 10mbit/1mbit connection excluding browser rendering time, for a modern browser.

Maintainability

The systems must be easy to maintain and extend which must be achieved through the use of good code structure using interfaces, design patterns, and extensive documentation. Changes in the backend of the `DriveIT System` requires a reinstall of the `DriveIT Web Client` and `Windows Client`.

Portability

The functionality of the `DriveIT` clients should be easy to port to other clients. This must be achieved by implementing the data access of the system using an open Web API with documented functionality, therefore allowing other clients to access the information. The `DriveIT Web Client` should be usable on mobile devices too.

Implementation

The different parts of the system must be implemented in different ways. It must have a main `DriveIT Web API` that must take care of persistence and communication between the `DriveIT Windows Client` and the `DriveIT Web Client`. The system must mainly be implemented using the programming language C♯. Implementation in a popular language such as C♯ assures supportability and extensibility. Using the *.NET framework* offers a lot of developer tools making development more efficient.

Operations

The system should mainly be maintained by the developers of the `DriveIT System`. Training is required for employees so that they are able to use the system on a daily basis and solve minor issues that might occur on a day to day basis.

CHAPTER **3**

System Models

3.1 Use Case Models

Scenarios

Six specific scenarios have been chosen for detailed specification. These are the scenarios that are the most important to get a good understanding of the more complicated features of the system. A list of the scenarios not further specified can be seen in appendix 9.2

The scenarios that chosen to focus on are shown below. These scenarios were chosen since they represent some of the paramount parts of the system functionality, thus analysing these further will ensure that the main logic of the system is handled correctly.

Scenario: Sell a Used Car to a Customer

Actors: `Bob:Employee, Alice:Customer`

Flow:

1. Alice has chosen to get contacted by an employee of the used car lot about a nice Mercedes Benz she has seen on the website.

2. Bob, the employee, sees on the `DriveIT Windows Client` that Alice wants to get contacted about a given car.

3. Bob uses the `DriveIT Windows Client` to get contact information about Alice and to get some information about the car and gives Alice a call.

4. Bob and Alice talk on the phone and Bob succeeds in convincing Alice to buy the car.
5. Bob uses the system to make a sale with the car and Alice as the customer. He then sends a bill via email to Alice.

Scenario: Employee Unsuccessfully Tries to Contact a Customer.

Actors: `Robert:Employee, Joe:Customer`

Flow:

1. Robert sees that Joe wishes to be contacted about a BMW.
2. Robert finds the info Joe has registered about himself in order to find his phone number.
3. Robert calls Joe, but Joe does not answer the call.
4. Robert looks in Joe's provided contact information to find his e-mail address.
5. Robert sends an e-mail to Joe using an external e-mail system.

Scenario: Customer Wishes to Find a Car

Actors: `Jane:Customer`

Flow:

1. Jane, recently divorced, just had her tricycle stolen. With her regular commute of 36 miles each way, she spontaneously decides to purchase a car.
2. Jane faintly recalls the midnight infomercial she viewed the previous night about the used car dealership.
3. After removing the heap of dirty clothes from her chair and finding her laptop, she swiftly types in the URL of the web page of the used car dealership.
4. Upon entering the website, she is prompted with a landing page, where she can see some of the most recently added cars. She can also go to a more advanced view where she can filter cars by her criteria.
5. She spots the ′filter by fuel type′ option and proceeds to locate the nearest blunt object to break her piggy bank.
6. After counting her pennies, nickels, and dimes she sets the ′filter by fuel type′ option to ′Diesel′.
7. Jane continues to click the ′Search′ button and sees a list of cars.
8. Jane scrolls through the cars, looking at the specifications with her needs in mind, and ends up falling in love with the 2008 Mini Clubman Cooper.

Scenario: Get Contacted by Employee

Actors: `Alice:Customer, Bob:Employee`

Flow:

1. Alice has been browsing the site of the used car dealership for a while and has found a car that she really likes.

2. Since Alice is not in a hurry to get contacted, she decides that the wishes to be contacted by an employee of the dealership.
3. Alice presses the button named "Request to Get Contacted".
4. Alice sees that the site where she requested to be contacted now says that a contact request has been sent.

Scenario: Put a Used Car up for Sale

Actors: `Bob:Employee`

Flow:

1. The used car dealership has just purchased a used car.
2. Bob needs to put the newly purchased car up for sale, so that the customers of the car dealership can see and hopefully buy it.
3. Bob knows the make and model of the car, but not any other information. He uses the `DriveIT Windows Client` to find additional information about the car.
4. Bob has also taken photos of the car, and uploads these to the page of the car.
5. Bob has put all the information and photos he knows into the system, and saves the information.
6. Bob double-checks that the car is saved and put up for sale.

Scenario: Customer Comments on a Car

Actors: `John:Customer`

Flow:

1. John is browsing for cars, and while browsing he finds a Toyoto identical to one he once had himself.
2. Since John was very fond of the car he wants to leave a comment, letting other people know that it is a good car.
3. John is already signed in, and is therefore able to comment right away.
4. He writes fills in the title and description fields of the site. He then creates the comment.
5. John is now able to see his comment, and can delete or edit it if he so desires.

Use Case Model

The use case model (fig. 3.1) shows which actors can perform which use-cases.

The unregistered customer can browse cars, see a detailed view of a specific car, and browse employees.
A registered customer can do the same as the unregistered customer, but has access to modifying his or her user profile, as well as requesting to get contacted by an employee and the ability to comment on cars that are up for sale. They also have the ability to see current contact requests that they have made and see all their previous orders.

An employee can browse cars, view details about cars, browse employees, modify, create, and view the list of customers who wants to be contacted regarding cars for sale, sell cars, and create, modify, and remove existing cars for sale on the web page.

The administrator is a special kind of employee who can access everything an employee can, as well as creating, modifying and removing other employees, and delete customer accounts.

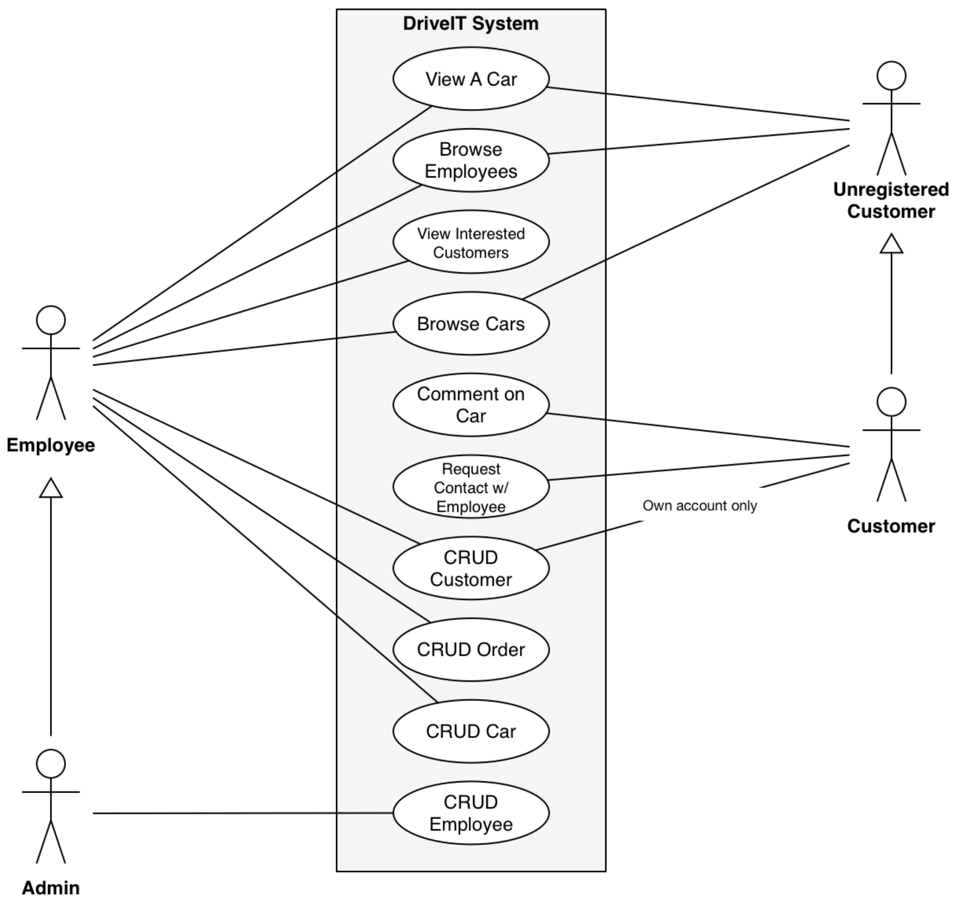


Figure 3.1 – The use case model - `Customer` inherits the unregistred customers cases, and the `Administrator` inherits from `Employee`

3.2 Use Cases

Due to time constraints only five use cases are explained in detail. The selected use cases are the most critical for the user requirements and are important to support for the system to fulfil the users' requirements.

UC-1 : CreateSale

Description: The `Employee` creates an order when a `Customer` has decided to buy a specific car.

Actors: `Employee, Customer`

Preconditions:

- The `Customer` and the `Employee` has agreed upon a deal regarding the purchase of a specific car.
- The `Employee` has the `DriveIT Windows Client` available with an Internet connection.

Postconditions:

- A `Sale` has been successfully created and is persisted in the database.
- The `DriveIT Web Client` now shows that the car is sold, but leaves it available to view in the system for another week.

Main success scenario:

1. The `Employee` opens the 'Sales' view.
2. The `Employee` presses the 'Create' button and selects the `Car`, `Customer`, and fills in the `Price` agreed upon.
3. When all the requires fields to make a valid `Sale` is filled, the `Employee` clicks the 'Create Sale' button.

Exception Cases:

- If the `Customer` does not already exist in the system, then the `Employee` performs the use case `"Create Customer"`.

Frequency: Daily

UC-2 : Create User Account

Description: The `Customer` registers on the `DriveIT Web Client`

Actors: `Customer`

Preconditions:

- The `DriveIT Web Client` is initialised and the device initialising it is connected to the Internet.
- The `Customer` is on the front page of the `DriveIT Web Client`.

Postconditions:

- The `Customer` has registered an account in the system.
- The `Customer` is able to log in to the system.

Main success scenario:

1. The `Customer` wishes to register an account for the `DriveIT Web Client`.

2. The `Customer` selects 'Register' in the main view of the `DriveIT Web Client`.
3. The `Customer` fills in his/her e-mail address, first name, last name, phone, and a password. He/she then selects the 'Register' button to save the account.
4. If the input is valid, the account is persisted and the `Customer` is logged in.

Exception Cases:

- If the phone number is not put in correctly the user will be notified about this.
- If the password is not put in correctly the user will be notified about this.

Frequency: Hourly

UC-3 : Create Car

Description: An `Employee` uses the `DriveIT Windows Client` to create a new car and thereby making it available for sale.

Actors: `Employee`

Preconditions:

- The `Employee` at least has the manufacturer, model, and price of the car plus at least one picture.
- The `Employee` is signed in to the `DriveIT Windows Client` and is connected to the Internet.

Postconditions:

- The `Car` is persisted in the database.
- The `Car` is available for sale on the `DriveIT Web Client`.

Main success scenario:

1. The `Employee` chooses the 'Car view' and chooses the 'Create' button to persist the car and therby making available on the `DriveIT Web Client`.
2. The `Employee` fills out the available data about the car.
3. The `Employee` checks that the data is correct and submits the car.

Exception Cases:

- If the phone number is not put in correctly the user will be notified about this.
- If the password is not put in correctly the user will be notified about this.

Frequency: A couple of times a week

Extensions:

- Given manufacturer and model, the `Employee` can choose to use the `Car Query` button to retrieve data about the given `Car`.

UC-4 : Contact Interested Customer

Description: An `Employee` contacts a `Customer` who has requested to get contacted.

Actors: `Employee, Customer`

Preconditions:

- The `DriveIT Windows Client` is online and the `Employee` is signed in to the system.
- There exists at least one `Contact Request` in the `DriveIT System`.

Postconditions:

- The `Customer` who requested to get contacted has been contacted by the `Employee`.

Main success scenario:

1. The `Employee` navigates to the view of `Contact Requests`.
2. The `Employee` chooses the oldest non contacted `Contact Request`, and opens a detailed view of the request.
3. The `Employee` notes the `Car` ID and `Customer` ID and navigates to the '`Car` view' to find the `Car` with the correct ID.
4. The `Employee` navigates to the '`Customer` view' and finds the `Customer` information.
5. The `Employee` uses a phone or an external email to contact the `Customer`.

Exception Cases:

- If the `Customer` does not answer the phone the `Employee` contacts the `Customer` through email.

Frequency: Daily

Extensions:

- Depending on the outcome of the communication between the two, the `Employee` either deletes the `Contact Request` or converts it into a `Sale`.

UC-5 : Request to Get Contacted by Employee

Description: A `Customer` wants to get contacted by an `Employee` regarding a specific car.

Actors: `Employee, Customer`

Preconditions:

- The `Customer` has established connection to the Internet.
- The `Customer` has found a specific car he wants to get contacted regarding and has loaded the detailed view of the car on the `DriveIT Web Client`.
- The `Customer` is signed in to the system.

Postcondition:

- The `Customer` has either been contacted regarding a car or has regret that he wants to get contacted and has deleted the request.

Main success scenario:

1. The `Customer` clicks on the 'Request to get contacted' button.
2. The `Customer` gets notified that he has requested to get contacted regarding the specific car.

Frequency: Daily

Extensions:

- If the `Customer` regrets that he wants to get contacted, he can follow a link on the detailed view for the specific car to see an overview of all his requests and then delete the given request.

3.3 Domain Object Models

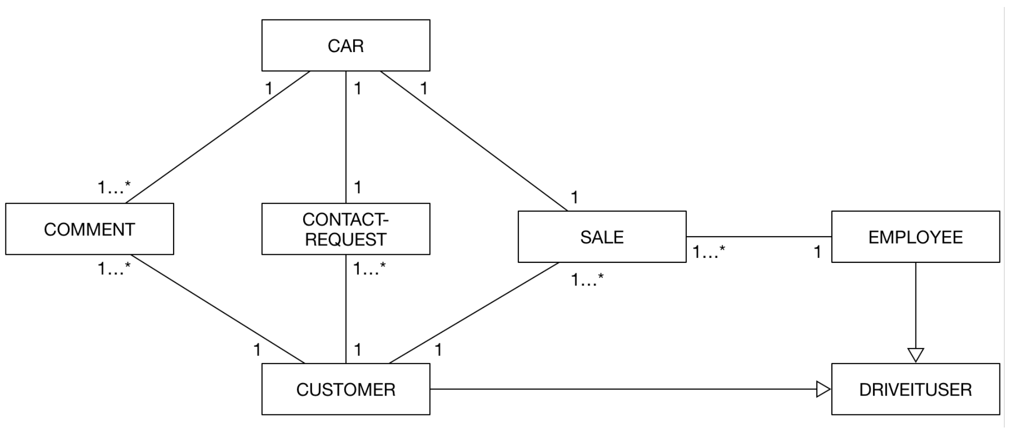


Figure 3.2 – The Domain Object Model of the `DriveIT System`.

The `Car` object is a given used car that the car lot has purchased. A `Car` can have many or no `Comments`. A `Comment` has only one `Customer`, but one `Customer` can create many `Comments`.

A `Customer` can create many `Contact Requests`, but a given `Contact Request` can only come from one `Customer`.

When a `Car` is purchased a `Sale` is created. A `Sale` can only have one `Customer`, one `Car` and one `Employee`.

One `Customer` can have many sales (if he/she buys many cars). An `Employee` can likewise have sold many cars.
An `Employee` represents an employee working at the used car lot, selling `Car`s to `Customer`s. The employee is referenced in a `Sale` when selling a `Car`.

3.4 Dynamic Models

Use Case Sequence Diagrams

CreateSale Sequence Diagram

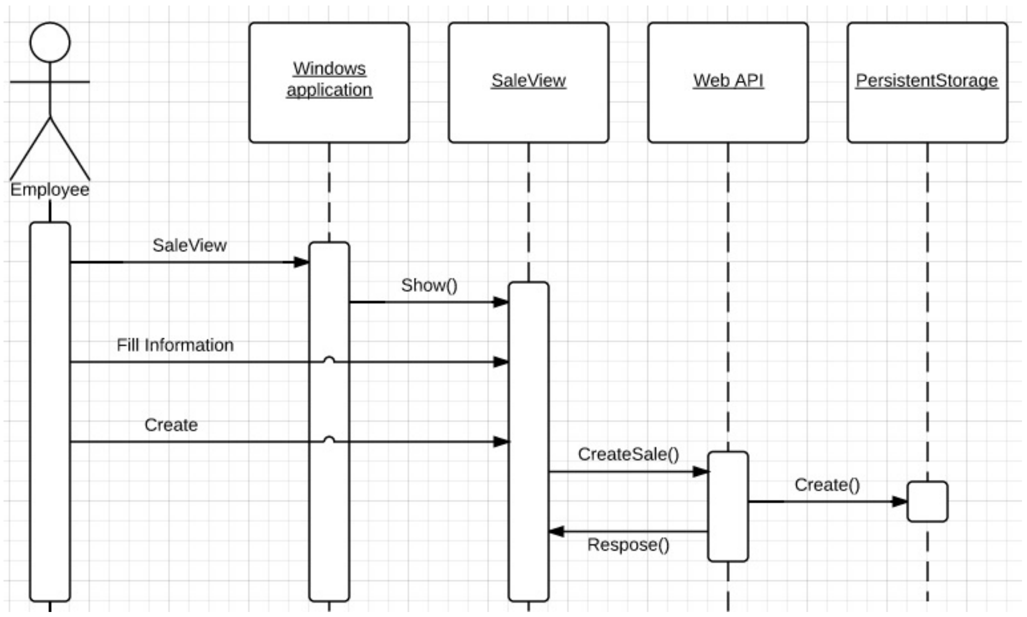


Figure 3.3 – Sequence diagram of the CreateSale use case.

Sequence diagram 3.3 is made from the use case `CreateSale UC-1`. *Sec. 3.2*.

In this use case, the employee starts with opening the `DriveIT Windows Client`, and thereafter navigates to the `Sale` view. In the `Sale` view, the employee fills in the information necessary to create a `Sale`, and after the `Employee` has finished filling in the information, he or she clicks on the "Create" button.

The sale will then be created and saved in the persistent storage. The `Employee` will then be prompted with a pop-up window, telling if the save was a success or an error has occurred.

CreateUserAccount Sequence Diagram

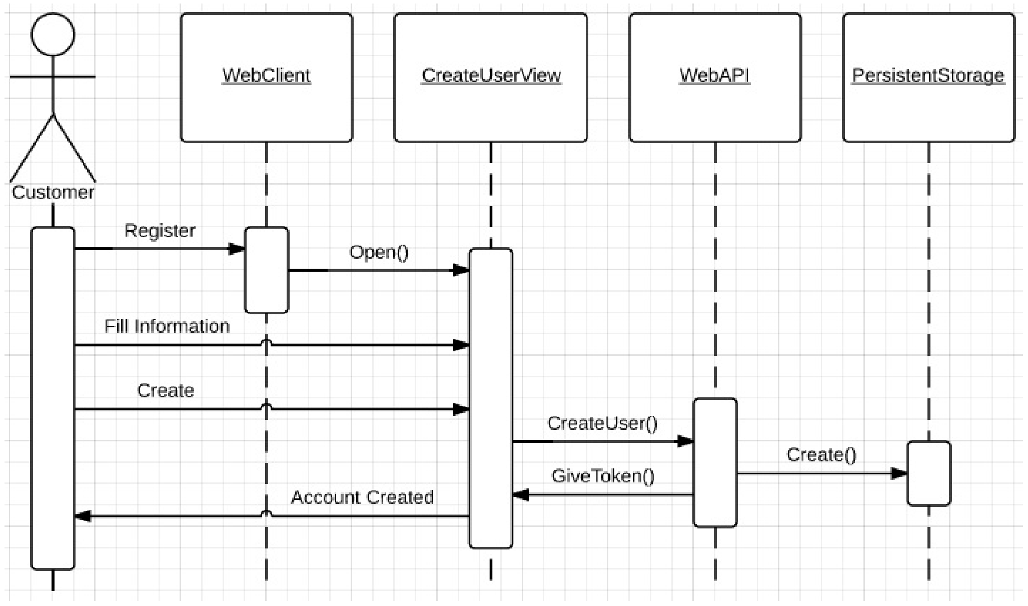


Figure 3.4 – Sequence diagram of the CreateUserAccount use case.

Sequence diagram 3.4 is made from the use case `CreateUserAccount UC-2`. *Sec. 3.2.*

In this use case, the non-registered `Customer` has navigated to the `DriveIT Web Client`, clicks "Register", where after the customer is directed to the `CreateUserView`.

In this view, the `Customer` will be filling in the necessary information, and `Customer` he or she then clicks the "Create" button. The `Customer` account will then be created and saved in the persistent storage. The `Customer` will then be redirected to the `DriveIT Web Client` front page and is logged in.

ContactInterestedCustomer Sequence Diagram

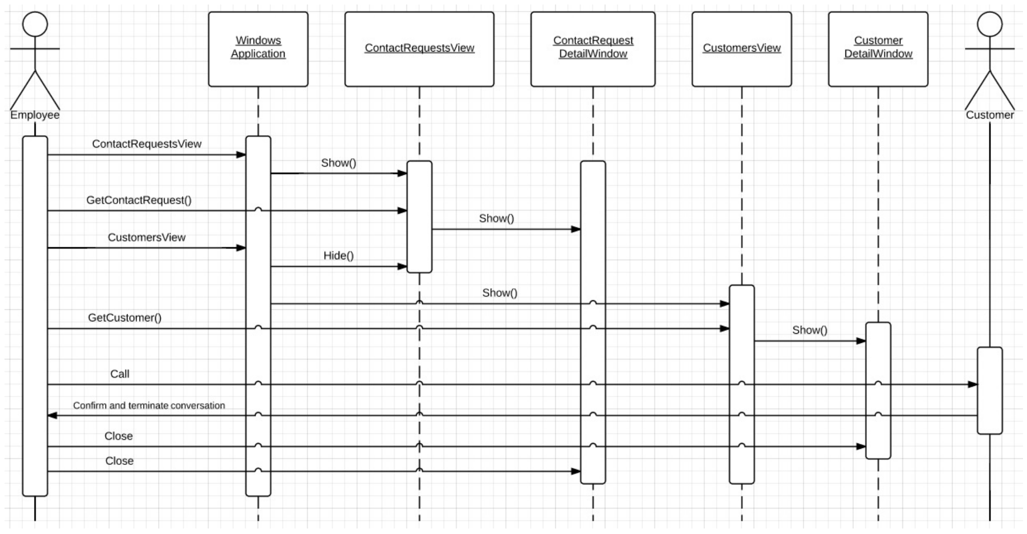


Figure 3.5 – Sequence diagram of the ContactInterestedCustomer use case.

Sequence diagram 3.5 is made from the use case `ContactInterested Customer UC-4` *Sec. 3.2*.

In this case, the `Employee` starts by opening the `DriveIT Windows Client`. The `Employee` then navigates to the `ContactRequestView`, where there will be shown a list of `Contact Requests` by different `Customers`.

The `Employee` picks one of the `Contact Request` and a more detailed view of the `Contact Request` will be shown. The `Employee` navigates to the `CustomerView`, finds the specific `Customer` and finds his information by clicking on the `Customer`.

RequestEmployeeContact Sequence Diagram

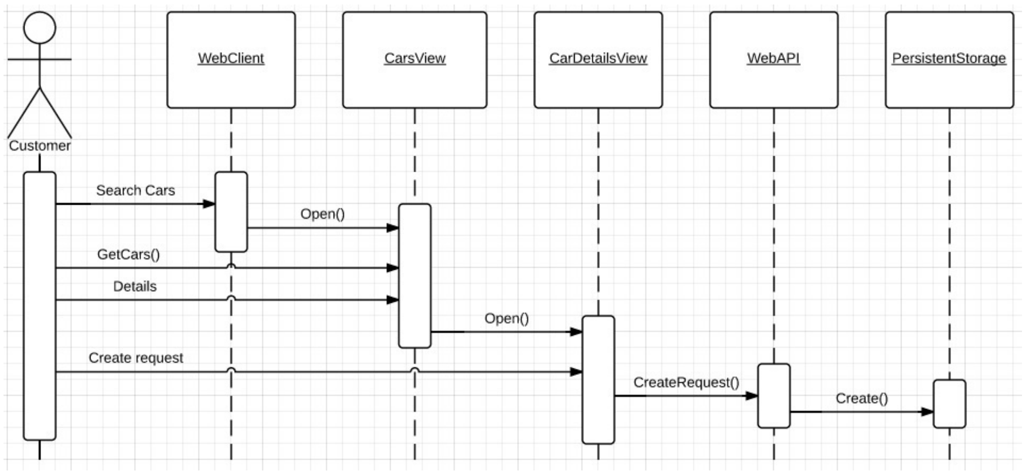


Figure 3.6 – Sequence diagram of the RequestEmployeeContact use case.

Sequence diagram 3.6 is made from the use case `RequestEmployee Contact UC-5`. *Sec. 3.2.*

In this case, the `Customer` is signed in and has navigated to the `DriveIT Web Client` and then clicks "Find Cars", whereupon the "CarView" will be show.

The `Customer` will then search for the specific `Car` that he or she is trying to find. The `Customer` will then navigate to the details of the `Car`, by clicking "Details". Then the `Customer` creates a request by clicking "Request to get contacted". The `Contact Request` will then be created and saved in the persistent storage.

State Diagrams

Car State Diagram

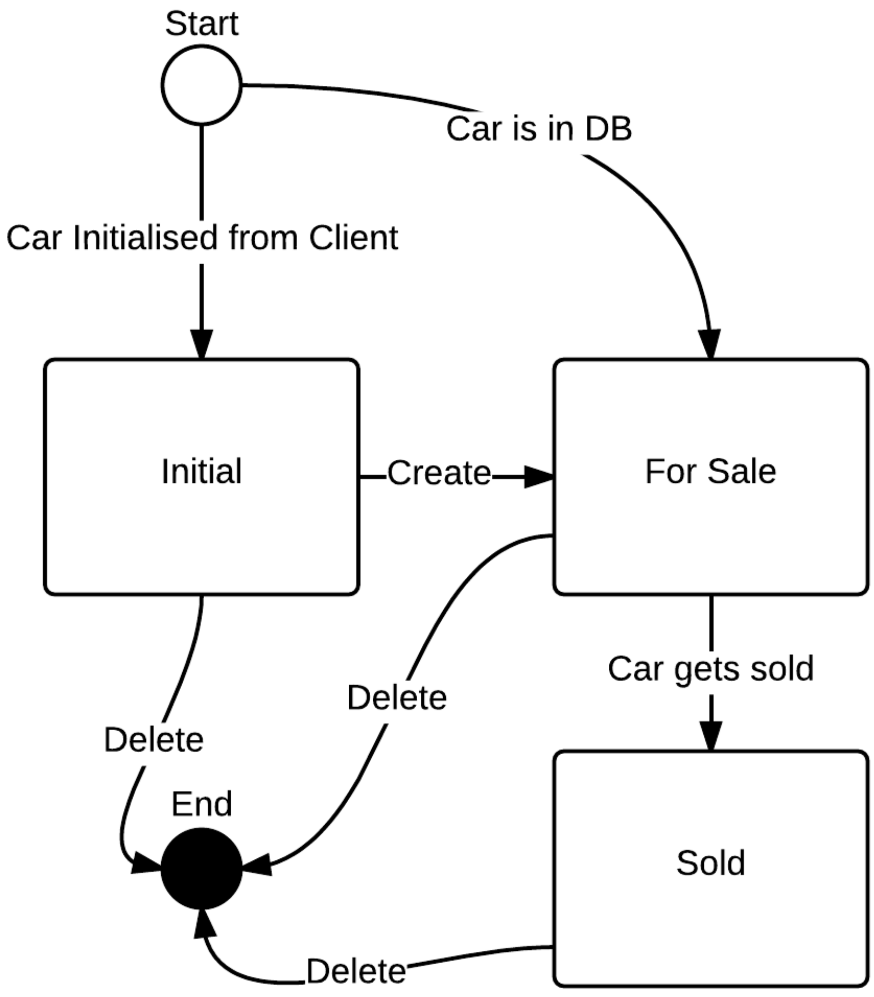


Figure 3.7 – State diagram of the Car object

Fig. 3.7 shows the states a `Car` can be in. It must always be possible to delete the `Car`, as shown in the diagram, but to be sold the `Car` must first enter the *For Sale* state.

In the *For Sale* and *Sold* state the `Car` should be visible from the `DriveIT Web Client` with a label indicating which of the two states it is in. The `Car` should not be visible on the `DriveIT Web Client` if sold more than a week ago. The `Car` should be visible from the `DriveIT Windows Client` in all of its states.

User Interfaces

Windows Client

The user interface was developed to have a native Windows look and feel from the beginning of the project. This means that anything superfluous has been removed, and that the content of the `DriveIT Windows Client` is in focus. The `Employee` needs to finish his/her task as quickly and efficiently as possible, which extra unnecessary user interface elements would hinder. Motion has also been introduced in the interface in the form of user interface transitions when navigating menus. This all follows the Design Guidelines of Microsoft[1].

The interface could have been more refined and made easier to use. The input fields for creating and updating entities could have fetched and parsed data dynamically, depending on the user context.

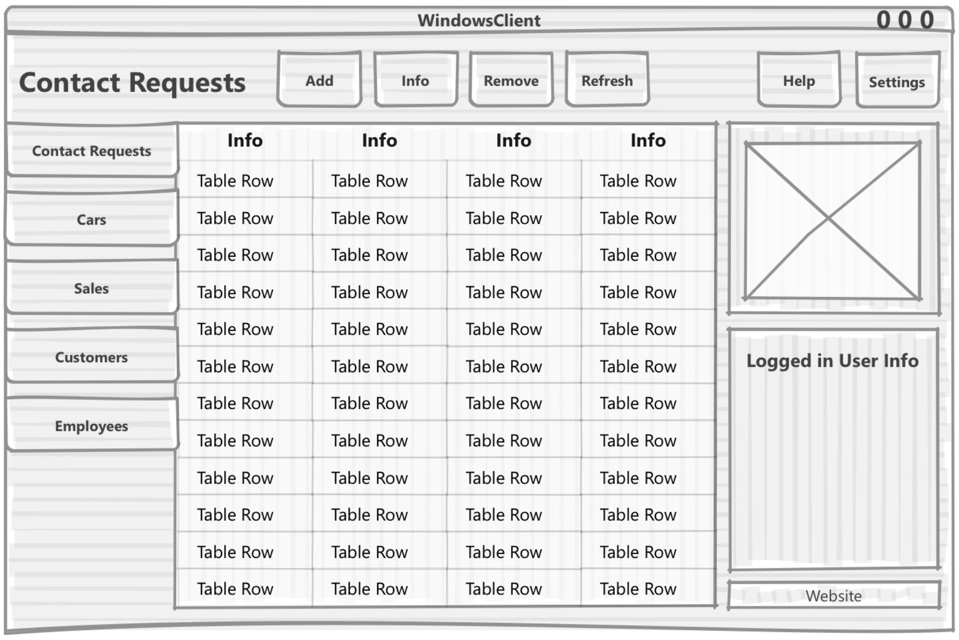
When creating a `Sale` the "Car Id" field could present more meaningful selection options for the `Employee` e.g. model names with an attached ID. Currently the user has to know the ID in advance.
This goes for all entities.

Prototyping

A prototype was sketched and refined over the course of the project.
The final prototype can be seen below.

Main Window:

[1]`http://msdn.microsoft.com/library/windows/apps/hh781237.aspx`



Info	Info	Info	Info
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row
Table Row	Table Row	Table Row	Table Row

Figure 3.8 – The Sketch of the Main Window of the Windows Client.

Entity Windows:

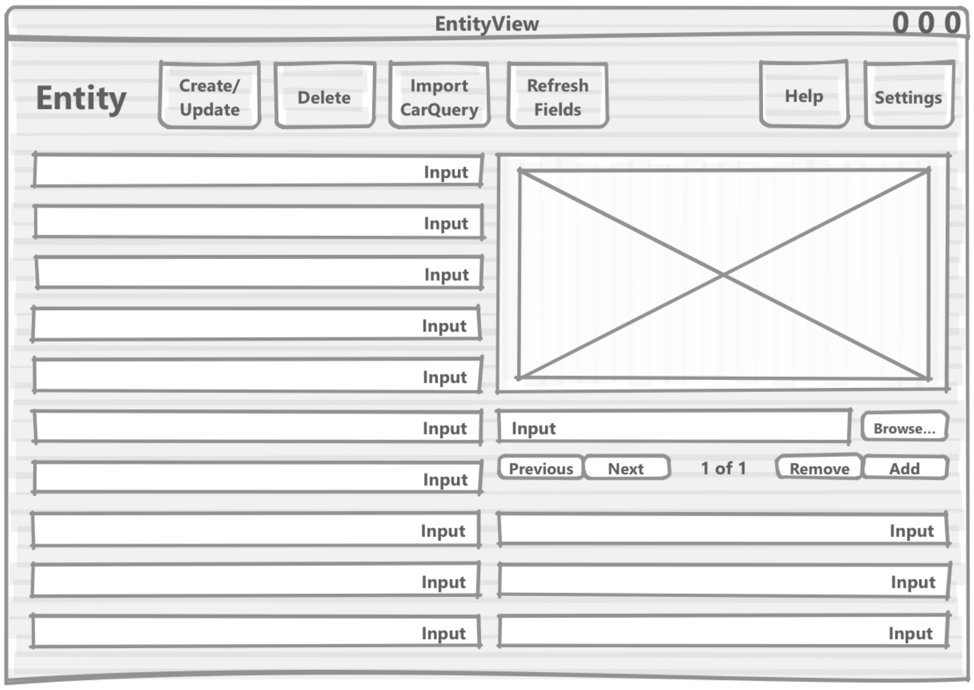


Figure 3.9 – The Sketch of the Entity View of the Windows Client.

Part II

System Design Document

CHAPTER 4

Proposed Software Architecture

4.1 Overview

This Software Design Document describes how the `DriveIT System` has been built. This document describes the following:

- Subsystem Decomposition
- Hardware and Software Mapping
- Persistent Data Management
- Access Control
- Global Software Control
- Boundary Conditions &
- Persistent Data Management

The `DriveIT System` is primarily composed of five subsystems: A web client for customers, a Windows client for employees. Furthermore a REST[1] web API functioning as a connection between the clients and the data-storage, a `CarQuery` subsystem for fetching extra data about cars, and a subsystem for storing and fetching data in a Microsoft SQL Database.

[1]`http://en.wikipedia.org/wiki/REST`

4.2 Design Goals

Powerful and Easy Usability

The goal with the usability of the `DriveIT Windows Client` is to give employees a powerful and easy to use tool to perform their work tasks. The *Gestalt Principles*[2] are excellent examples on how to make a clean and simple user interface. By using the *Gestalt Principles*, we are able to make a clear distinction of which parts of the user interface that belongs to each other. This makes it easier for the user to interact with the user interface. To give the `Employee`s a powerful tool, the most frequently used tasks must be achievable quickly, but with possible extensions to allow more advanced options. This is making it fast to use the interface, while still maintaining enough complexity to be in appliance with the requirements.

For the usability of the `DriveIT Web Client` the goal is to have a website which is simple and provides the `Customer` with only a few actions. This will decrease the possibility of confusion and therefore hopefully keep the users on the site. *Gestalt Principles* should be used in the making of the *User Interface* (henceforth UI) such that it is easy for the user to see which functions map together.

High Reliability

It is desired to create a reliable program that will prevent the user from losing progress made in the system due to unexpected events e.g. a crash. By synchronising with the *Microsoft Azure* server after every *CRUD*[3] action and not just at start up and shut down, we ensure that a minimum amount of data is lost, if a crash should occur. Should the `DriveIT Windows Client` loose connection to the server a message must be shown to the user. All CRUD actions will fail until a connection is re established. This is a design choice which is chosen due to the time limit of the project.
Force restarting is not acceptable and most exceptions must be caught and handled during runtime. By doing the above mentioned, data loss will be kept to a minimum by limiting it to the user's current activity, should a failure occur. Due to time constraints the exception handling of both `DriveIT Clients` are not implemented in such a manner, that the error message presents enough details about the error that occurred.

[2]`http://en.wikipedia.org/wiki/Gestalt_psychology`
[3]`http://en.wikipedia.org/wiki/Create,_read,_update_and_delete`

Strong Architecture with Focus on Extensibility

When rolling out future updates for the `DriveIT Systems`, a full reinstallation will be necessary. This is due to time constraints and because this feature is not part of the scope. It would be desirable to have a patch system, since this makes it less resource demanding to update the system in the future. It would also make it easier for the `Employee`s to just accept and update instead of manually having to uninstall the application and then reinstall it.

Good Documentation and Testing of the Most Important Subsystems

The `DriveIT System`s must be tested before release, but in a limited way, as a full unit, integration, and acceptance test are out of the scope due to the time frame set for the development of the system. Focus has therefore been on testing key components which are central and critical parts of our system, while also working as a template for future tests.

Documentation is a central part of the `DriveIT System`, as it is important to document how the system works as a whole and also how the modules and subsystems work separately. By making this documentation it will be easier to extend the system by developers that have no knowledge about the `DriveIT System`.

High Portability

To make sure that the system is easy to extend and scale it will be built around a web API which makes it easy to communicate with the system. This allows other developers to make applications for other platforms i.e. mobile devices by using the `DriveIT Web API`.

Easy and Efficient Implementation

The fact that the system will be developed and maintained in a popular and broadly used programming language, C♯, assures a relatively painless implementation process. Since C♯ uses the .NET platform there are a lot of well designed and highly maintained tools available, minimizing the development time and increasing the quality of the software.

Security as a Secondary Priority

It is not a part of the scope of this project to take security into consideration. However security has to some extent been considered, since authorization is required to access to certain parts of the `DriveIT System`. Using the *ASP.NET Identity*[4] framework, passwords are hashed when they reach the `DriveIT Web API`, though not until then. Because *SSL*[5] is out of scope data sent between the `DriveIT Systems` are not encrypted.

4.3 Subsystem Decomposition

The `DriveIT System` can be split into five main subsystems - the `DriveIT Windows Client`, `CarQuery`, `DriveIT Web Client`, `DriveIT Web API`, and `Persistent Storage` subsystems. These subsystems are described below.

The DriveIT System

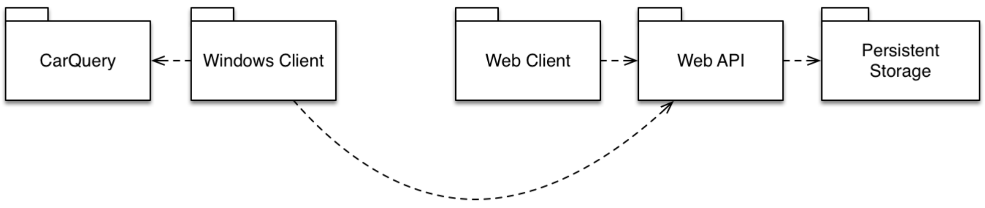


Figure 4.1 – Subsystems of the `DriveIT System`.

Persistent Storage Subsystem

The Persistent Storage Subsystem manages storing and retrieving entity objects using the *Entity Framework* and its serialisation functionality.
The serialised entities are stored in a *Microsoft SQL Database* which is hosted on *Microsoft Azure*. The subsystem provides the `DriveIT Web API` with deserialised entities, which uses the entities to provide the

[4]`http://www.asp.net/identity`
[5]`http://en.wikipedia.org/wiki/Secure_Sockets_Layer`

backend for other subsystems.
The subsystem supports retrieving all Entities of a given type, a specific entity using its unique ID, and retrieving an entity on its relation to other entities. The Persistent Storage Subsystem consists of two classes and an interface (see figure 4.2). `EntityStorage` is the repository working on top of the *Entity Framework* database layer, `DriveITContext`, providing the `IPersistentStorage` interface.

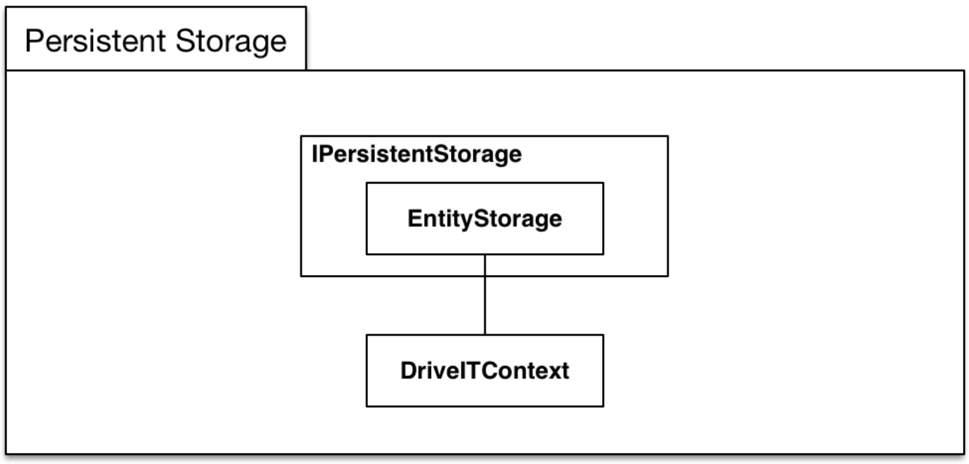


Figure 4.2 – Classes in the Persistent Storage Subsystem.

Web API

The `DriveIT Web API`, which is built on *ASP.NET Web API*[6], provides public communication with the `Persistent Storage` subsystem. This is done using specific URL routes. Furthermore it enforces authorisation so unauthorized users, and users with different user roles, only have limited access to the data of the `DriveIT System`.
Every table in the `Persistent Storage` subsystem must therefore be supported by the web API, although not every table is available to every user.
The `DriveIT Web API` is comprised of a series of modules for serialising the Model Entities into `JSON`[7] or `XML`[8] and transferring these via a `REST` interface accessible using `HTTP`.
These modules are built into *ASP.NET Web API*, and are not handled in

[6]`http://www.asp.net/web-api`
[7]`http://en.wikipedia.org/wiki/JSON`
[8]`http://en.wikipedia.org/wiki/XML`

our code.

The Web API consists of several controllers which uses a static class to translate `DTO's`[9] into entities and vice versa. It is also the controllers that check for special cases, e.g. when a customer wants to update or delete a comment, and it is not allowed to change other `Customers`' comments.

Windows Client

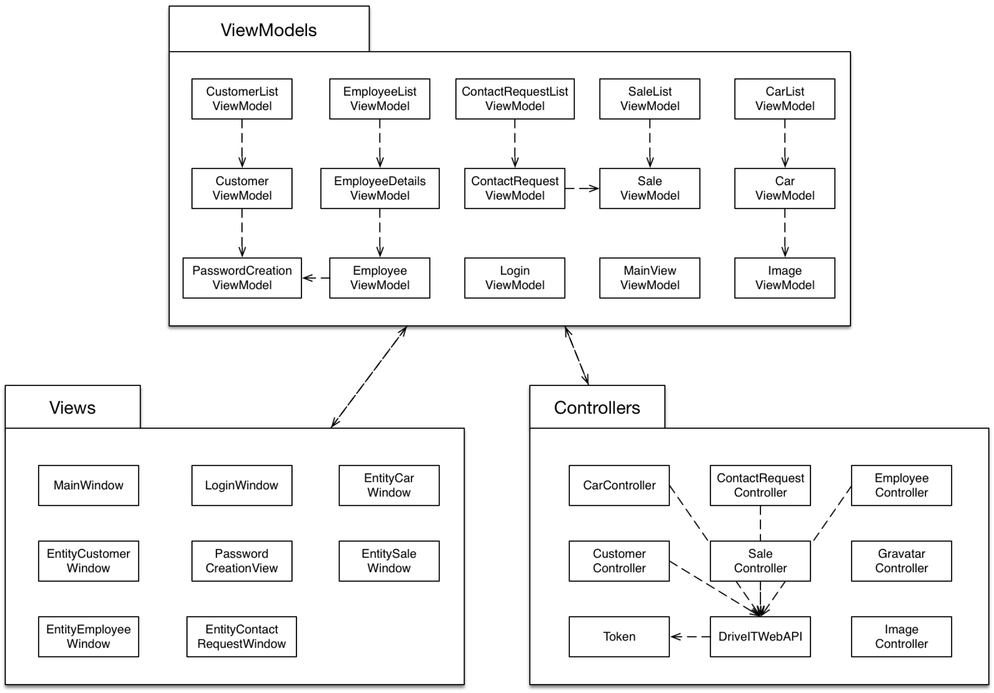


Figure 4.3 – The Code Map of the Windows Client

The `DriveIT Windows Client` is used by the `Employees`, to manipulate data in the `DriveIT System`. Depending on their role, `Employee` or `Administrator`, they can *CRUD* all entities except for creating `ContactRequests`.

The `Employee` signs in to the client with his or her email and password, and can then use the user interface to manipulate data in the `DriveIT System`.

The subsystem is made out of three main components: The `Views`, the `ViewModels` and the `Controllers`.

[9]`http://en.wikipedia.org/wiki/Data_transfer_object`

The `Views` package contains the user interface written in Windows Presentation Foundation (henceforth WPF). The `ViewModels` package contains the `ViewModels` which hold data bound to the `Views` following Microsoft's *Model View ViewModel* architectural pattern[10].

`ViewModels` for entities are often created in the form of a `EntityListViewModel` and a `EntityViewModel`, but a few extra `ViewModels` exists for additional `Views` such as the `LoginViewModel` and the `PasswordCreationViewModel`.

The `Controllers` package contains the `Controllers` which transforms data and sends and receives HTTP requests to and from the `DriveIT Web API`. Images are uploaded to *Azure Blob Storage*. To provide `Employees` and `Customers` with profile pictures, a `GravatarController` exists to convert email strings into *Gravatar* profile URLs.

Web Client

The `DriveIT Web Client` is accessible through the Internet, where it is being used by non-registered and registered `Customers`. The `DriveIT Web Client` is being used to search for `Cars`, that are up for sale in the `DriveIT System`.

The `DriveIT Web Client` allows the `Customer` to *CRUD* `Comments` if signed in. Otherwise `Comments` can only be read.

It is also possible to *create*, *read* and *delete* `ContactRequests` and to read `Orders` and `Employees`. A `Customer` is also able to *read* and *update* information regarding their own account.

It is possible to create an account on the web page with a non registered email.

The `DriveIT Web Client` is based on the *ASP.NET MVC* framework[11], that is implementing the *Model View Controller* design pattern (see section 6.3).

The *MVC* subsystem would usually consist of three components: *Models*, *Views* and *Controllers*. However, since the `DriveIt Web Client` directly references the controllers from the `DriveIT Web API`, the `MVC` subsystem uses the same models as the `DriveIT Web API`. A regular *ASP.NET MVC* project, would have its own models for persistence. The *MvcControllers* create instances of the corresponding *ApiControllers* and use these to retrieve the objects from the `DriveIT Web API` and serve them to the *Views* as *Models*. These *Models* are then used in the views to represent the different DTO's.

[10]`http://en.wikipedia.org/wiki/Model_View_ViewModel`
[11]`http://www.asp.net/mvc`

CarQuery

This subsystem is a collection of classes that enable the the `DriveIT Windows Client` to fill out missing information about cars, using the *CarQuery API*.
The subsystem is used by an `Employee` when creating or updating a `Car`. The subsystem consists of a *JSON* deserialiser that communicates with the *CarQuery API* and another class that receives a `Car` object and fills out the missing attributes using the deserialised *JSON* data.

An `Employee` fills out the known information about the `Car` and the `CarQuery` subsystem then creates its query on the *CarQuery API*.

Only the `CarQuery` static class is publicly accessible. As seen in figure 4.4 it uses a `JSONWrapper` to receive data from the *CarQuery API* and deserialises it into a `Trims` object which is an array of `Trim` objects holding the actual `Car` data.

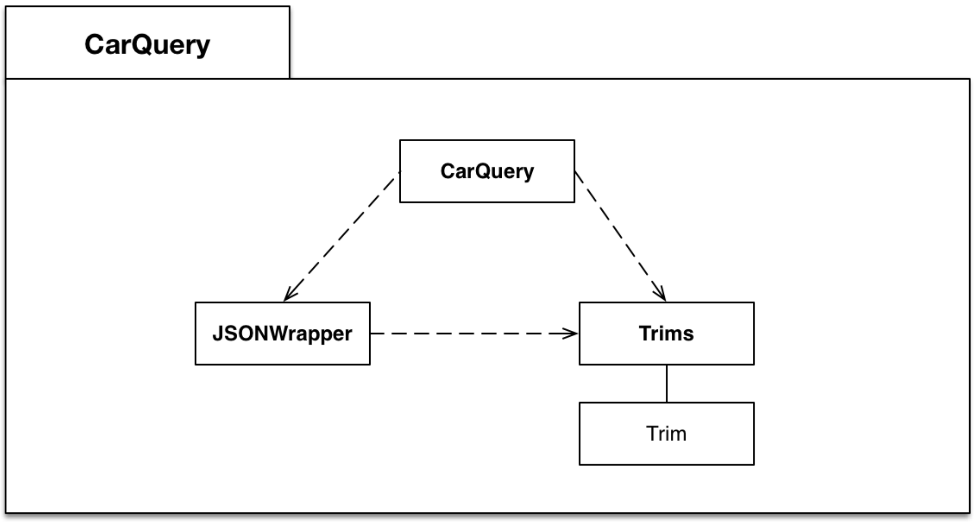


Figure 4.4 – Classes of the `CarQuery` Subsystem.

4.4 Hardware and Software Mapping

As figure 4.5 shows, we want to run the `DriveIT` subsystems on the following hardware:

The `Persistent Storage`, `DriveIT Web API`, and `DriveIT Web Client` subsystems will be hosted on a web server running

ASP.NET. We have chosen *Microsoft Azure* as our host.

The `DriveIT Web API` uses the `Persistent Storage` subsystem internally to contact the *Microsoft SQL Server* that runs behind the *Entity Framework*.

The `DriveIT Web Client` is accessed through a modern web browser using HTTP. The `DriveIT Web Client` uses the `DriveIT Web API` internally, and as they are located in the same project, they can be hosted on the same website at *Microsoft Azure*.

The `DriveIT Windows Client` runs on the computers of the used car dealership using the *.NET Framework 4.5*. It furthermore uses the `CarQuery` subsystem, which helps `Employee`s fill out missing information about `Car`s. The `DriveIT Windows Client` communicates with the `DriveIT Web API` subsystem through HTTP. The content of the HTTP-requests and responses are *JSON* objects when data transfer is needed.

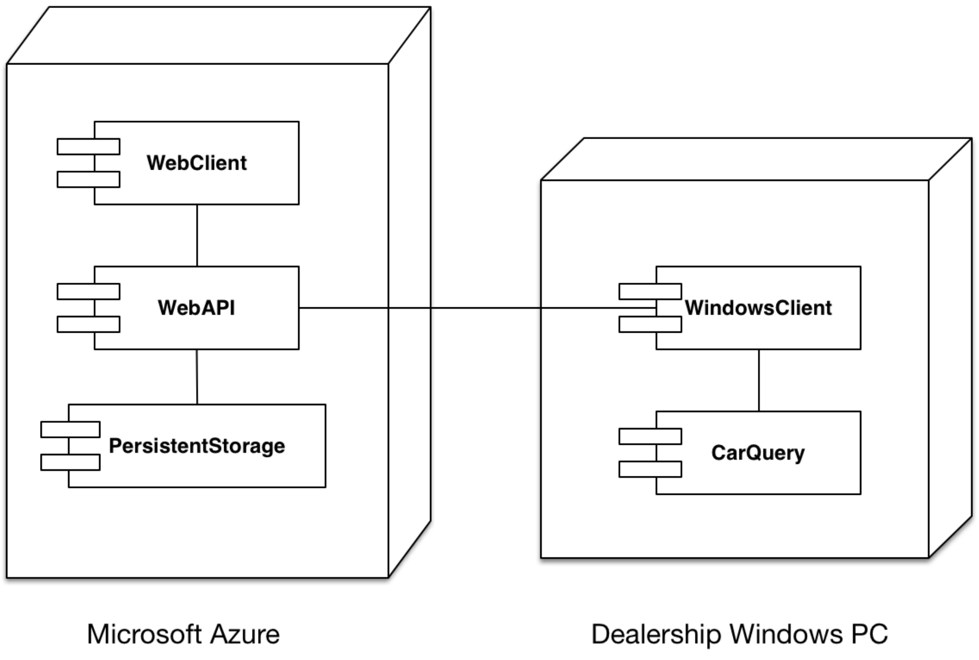


Figure 4.5 – Hardware software mapping of the entire system.

4.5 Persistent Data Management

Data of all entities in the `DriveIT System` are stored persistently to avoid data loss if the system crashes, and to ensure that users can expect to resume work from the point where they last left. The `Persistent Storage` subsystem stores this data using a *DBMS*[12] hosted online - more specifically in a Microsoft SQL database.

It stores the following entities:

- **Car:** Representing a used car. `Car`s are created, read, modified, and deleted by subsystems in the `DriveIT System`.
  `Car`s have many attributes, all describing the properties of the used car. `Car`s are referenced in many other entities in the storage system, but do not reference other entities themselves.
- **Customer:** Representing a potential buyer of car(s). `Customer`s are read by some parts of the system, but only modified and deleted in certain authorized instances.
  `Customer`s have few attributes describing the person acting as the customer. `Customer`s are referenced in some entities in the storage system, but do not reference other entities themselves.
- **Employee:** Representing a salesman of cars. `Employee`s are read by some parts of the system, but only modified and deleted in certain authorized instances.
  `Employee`s have few attributes describing the person selling cars. `Employee`s are referenced in some entities in the system, but do not reference other entities themselves.
- **Sale:** Representing a sale of a car. `Sale`s are only created, read, modified, and deleted in certain authorized instances.
  `Sale`s have few attributes describing the specific sale, but mainly reference other entities. A given `Sale` references an `Employee`, a `Car` and a `Customer`.
- **Comment:** Representing a comment on a car. `Comment`s are read by some parts of the system, but only modified and deleted in certain authorized instances.
  `Comment`s have few attributes describing the specific comment, but mainly reference other entities. A given `Comment` references a `Car` and a `Customer`.

[12] `http://en.wikipedia.org/wiki/DBMS`

- **ContactRequest:** Representing a customer's request for contact about a car which can be followed up by an employee. `ContactRequest`s are only created, read, updated, and deleted in certain authorized instances.
  `ContactRequest`s have one attribute describing the specific request, but mainly reference other entities. A given `ContactRequest` references a `Car`, a `Customer` and optionally an `Employee`.

Figure 4.6 shows all entities with their respective attributes.

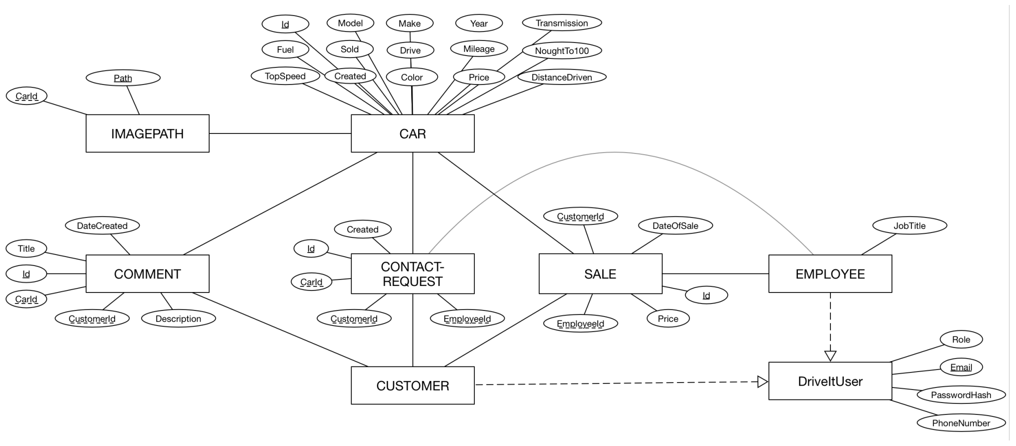


Figure 4.6 – The Persistent Data Model.

4.6 Access Control and Security

Due to the fact that the `DriveIT System` is available for many users, it is necessary to keep track of which users may have access to specific resources in the system. To keep track of which users have access to which resources, it is necessary to check each user at login, through authentication, to confirm which resources they can access. Figure 4.7 shows which functionality the different actor types have access to.

In the `DriveIT System` a non-registered customer is restricted to browse `Car`s with comments, get a list of `Employee`s and create an account on the homepage.

The `Customer` is able to access the same information as a non-registered customer, but is also able to look through in `Sale`s which he or she has been part of, and able to create, update or delete his or her own `Comment`s

Actors \ Entities	Car	Comment	ContactRequest	Customer	Employee	Sale
Non-registered Customer	read	read		create	read	
Customer	read	create read update delete	create read delete	read update	read	read
Employee	create read update delete	(read)	read update delete	create read update	read	create read update delete
Administrator	create read update delete	(read) (delete)	read update delete	create read update delete	create read update delete	create read update delete

Figure 4.7 – Access Matrix for the `DriveIT System`

on `Cars`.

The `Employee` is able to create, read, update, and delete `Cars`. The `Employee` is also able to create a new `Customer`, get information about a specific `Customer` and update a `Customer` by request. The `Employee` can furthermore create a `Sale`, read all `Sale`, update, and delete a `Sale`. The `Employee` can read a list of other `Employees`.

An administrator is a special kind of `Employee`, who can do everything the `Employee` can, but in addition create, update and delete other `Employees` and administrators. `Customer` accounts can be deleted by the administrator role.

The `DriveIT Web API` supports that `Employees` and administrators can read `Comments` on `Cars`. It furthermore supports that administrators can delete comments. The `DriveIT Windows Client` does not support these features.

There is a bug in the `DriveIT Web Client` which means that `Employees` and administrators can create `Comments` and `ContactRequests`. They cannot manage other `Customer's` `Comments` and `ContactRequests` though. Information about `Employees` and administrators cannot be changed through the `DriveIT Web Client`, as it was not meant to support this functionality.

4.7 Global Software Control

The flow of the `DriveIT System` would be a *Procedure-driven control*[13], and the reason for this, is that the program will follow specific procedures, depending on how the `Customer/Employee` is interacting with the system. An exception to the flow model is the `DriveIT Windows Client`. This subsystem uses an event-driven control flow for the data-binding between the `View` and `ViewModel`s.

By using the *Procedure-driven control* approach, some concurrency and synchronisation problems may occur. If two `Employee`s update the same entity which were read at the same time, then the last update will override the first. Due to the scope of the project, such problems are not handled in the `Client`s.

Upon starting the system, the `Customer` will be presented with the main view of the system. It is not required to log into the system to use it. The `Customer` can freely navigate through the database searching for cars given the search criteria. Only when the `Customer` wishes to be contacted about a `Car`, a login will be prompted.

This differs a bit for the `Employee` as their usage of the system is different. An `Employee` has tasks which require a higher access level, that are not available for a `Customer`. Therefore an `Employee` will log in upon starting up the system, and hereby verify the authorization level.

For logging in, the `LoginViewModel` of our `ViewModel Subsystem` will use the information entered into our login view to authenticate the information, and if the authentication through the `EmployeeController` of the `Controller Subsystem` is a success, the `Persistence Subsystem` will access the information belonging to the `Customer/Employee`, and our `ViewModel Subsystem` will then observe the `Persistence Subsystem` and update its state, whereafter the `DriveITMainView`, from our `View Subsystem`, will be initialized and show the data received from the database.

[13]Object-Oriented Sofware Engineering - 3rd Edition - 2009 - page 275

The `Customer` can now interact with the system in multiple ways. For instance, the `Customer` can view the cars they are interested in. This task is handled by the `Controller Subsystem`, which updates the state of the `Model Subsystem` which is then reflected in the `DriveITMainView`, which will register the activity of the `Customer`, update the state of the Model and then load it into the DriveITSystemView. In the background, new activity will be stored in the database, as the `Customer` is still using the system. If a change is commited, the Model will update its own state, whereafter the `DriveITMainView` will be updated by the `ViewModel Subsystem`.

On the other hand an `Employee` will have the main task of handling requests for contact by a `Customer`. Anybody with `Employee` authentication, will have a shared 'forum', where they can view a list of `Users` who are interested in cars/contact from an `Employee`. An `Employee` can now drag one of these requests into their own workfield and hereby claim a task. The same way as the task for a `Customer` above, the task will be handled by the `Controller Subsystem`, which updates the state of the `ViewModel Subsystem` which is then reflected in the `DriveITMainView`. This will register the activity of the `Employee`, update the state of the Model and then load it into the `DriveITMainView`. In the background, new activity will be stored in the database, as the `Employee` is still using the system. If a change is commited, the Model will update its own state, whereafter the `DriveITMainView` will be updated by the `ViewController Subsystem`.

4.8 Boundary Conditions

Service Boundary Condition

For the `DriveIT Web Client` and the `DriveIT Windows Client` to be initiated the `DriveIT Web API` must have been initialised otherwise all functionality will be unavailable.

Furthermore the `DriveIT Web API` must run constantly to allow the clients to persist and retrieve data.

Subsystem Boundary Condition

DriveIT Web Client

On initialisation of the `DriveIT Web Client` all controllers are configured to map to certain routes. Furthermore *OWIN*[14] configured for the `DriveIT Web Client`. This is all done automatically by *ASP.NET* and *OWIN*.

Every time a page is loaded the data shown in the view is read from the `DriveIT Web API`, thereby synchronising data.

Exceptions which are not caught and handled, will display a generic status message, but will not crash the server.

No actions are currently performed at shutdown.

DriveIT Windows Client

On initialisation of the `DriveIT Windows Client` an `Employee` must sign in with a valid account in the `LoginView`. This initialises the static class `DriveITWebAPI` such that a connection to the `DriveIT Web API` can be established when needed.

To synchronise with the `DriveIT Web API`, the `DriveIT Windows Client` updates data each time a list-`View` changes.

The `DriveIT Windows Client` handles exceptions by displaying an error message to the user in the `View` where the error originated. This is displayed in a status label, mentioning which action failed.

On shutdown the objects of the `DriveIT Windows Client` are disposed by the *CLR*[15]. No further actions are performed.

[14] `http://owin.org`

[15] `http://msdn.microsoft.com/en-us/library/8bs2ecf4%28v=vs.110%29.aspx`

CHAPTER 5

Subsystem Services

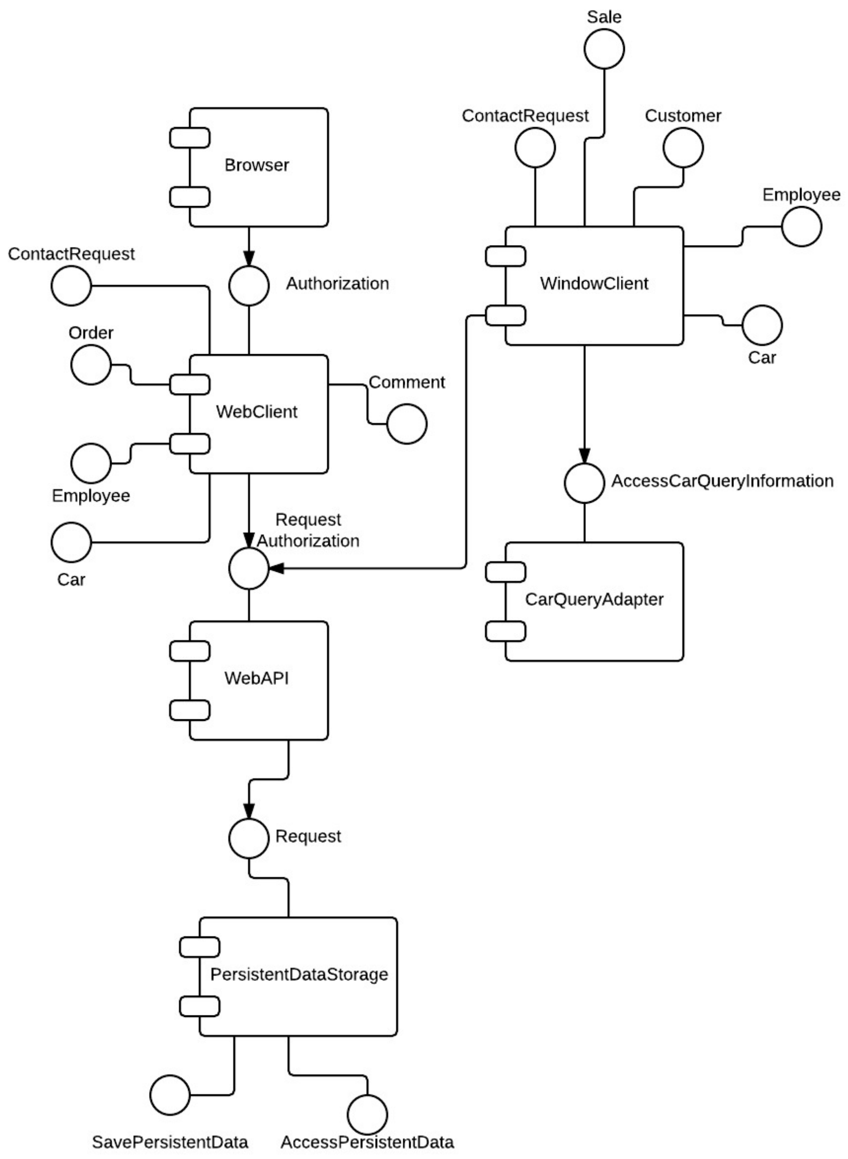


Figure 5.1 – Services provided by the `DriveIT` subsystems

The above figure shows how the different services of the `DriveIT System` interact with each other.

The five main subsystems provide ways to contact each other. The main communication happens through the authorisation hooks of the systems, which is illustrated on the figure above. Data access is also provided by the different services, so that entities can be transferred between services.

Part III

Object Design Document

CHAPTER 6

Object Design and Patterns

6.1 Object Design

Reuse

DriveIT Windows Client Controllers

To perform the create, read, update, and delete functionality the `DriveIT Windows Client` uses a number of controllers. These controllers each handle the different kinds of entities which the client supports. By reusing code the client becomes easier to test, maintain and less error prone. The class `DriveITWebAPI` is created with this purpose in mind. This class encapsulate all HTTP request code to contact the `DriveIT Web API` and uses generics to allow all of the controllers to reuse the methods. Therefore all the entity controllers know nothing of the `DriveIT Web API` or the code to contact it, and with a few adjustments the static `DriveITWebAPI` class could be reused in any other project involving a REST based web API.

Persistent Storage

The `EntitiyStorage` class is very specific for the `DriveIT System` since it only deals with creating, reading, updating and deleting `DriveIT` entities. It therefore does not provide any major re usability for other purposes than this.

Adding re-usability could be accomplished by using generics, supported by the C♯ language, which would allow the system to create, read, update and delete any input entity, provided it was supported by the given `DbContext`. This would allow for greater extensibility and easier code maintenance.

CarQuery

The `JSONWrapper` of the `CarQuery` subsystem is written using generics, which enables using any object and URL as input for getting and deserialising a serialised JSON object, provided that the properties of the input object match the received JSON data. The class is only used by the `CarQuery` subsystem, though. This functionality could be used by other classes in the system for communicating with e.g. the `DriveIT Web API`. Having a generic method for dealing with filling out properties instead of a specific `Car`-only method would also have benefited reuse of the system.

Encapsulation

Encapsulation has been of high priority during development of the `DriveIT System`.

Keeping functionality of different subsystems encapsulated has helped debugging and refactoring. Extensibility is also made easier in the future.

Persistent Storage

Encapsulation has been used in the `Persistent Storage` subsystem in methods dealing with retrieving, editing and deleting entities. Methods generally have few side effects and do not keep references to other objects after performing their task.

A design choice in implementing the methods were to keep a `DriveITContext` in memory for a short amount of time. This ensures that the context fulfil its purpose in dealing with the entities where after it is disposed.

Updating entities requires copying all attributes of the entity which is done in separate methods without side effects.

CarQuery

The functionality of the `CarQuery` subsystem is implemented using few static classes with few methods dealing with small tasks. The classes have been made static due to the fact that no initialisation is required.

The subsystem is used in short periods of time and has one purpose: getting `Car` data. Therefore no object needs to be kept in memory, data should just be deserialised and added to a `Car`. The entire `CarQuery` subsystem is therefore made up of static functionality, since no data needs to be set up before using the sub system i.e. by using a constructor.

Web API

Because the `DriveIT Web API` is supposed to provide an external REST-interface, it has been important not to expose the internal types of the `Persistent Storage` subsystem. Due to this only *DTO*'s are exchanged between the `DriveIT Web API` and the clients.

This has also made it possible to change details about the storage without having to change the *DTO* and therefore functionality in the clients. An example is the `Car` entity. In the first draft of the database the property `Sold` was maintained by the `Employee`s of the car dealership using `DriveIT`. In a later revision it was decided to calculate the value from data in the database. If a `Sale` entity references the `Car`, it is sold. This could happen without breaking the functionality in i.e. the `DriveIT Windows Client`.

Web Client

Initially the `DriveIT Web Client` was using the `DriveIT Web API` through *HTTP*-requests. However, after some consideration it was clear that it did not make much sense, not to use the controllers from the `DriveIT Web API` as controllers for the `DriveIT Web Client` as well. This was due to the fact that it is common to use *ASP.NET MVC* projects as *REST API*s for other systems to use. Since both the `DriveIT Web API` and `DriveIT Web Client` is to be deployed on *Microsoft Azure* it makes the most sense to have the implementation of both in the same project.

Inheritance

Persistent Storage

The `DriveITContext` extends an `IdentityDbContext` which provides the built in functionality for handling user logins and different user roles. This is used for the sign in and access control functionality of the `DriveIT System`. The `IdentityDbContext` in turn extends a `DbContext` which enables `Entity Framework` to save the user entities, along with every other `DriveIT` entity, in the underlying Microsoft SQL database.

In the `Persistent Storage` subsystem we have two kinds of concrete user types. The `Customer` and the `Employee`. Both of these inherits from an abstract `DriveITUser` which inherits from `IdentityUser` which is a base for users in ASP.NET Identity. `DriveITUser` adds a few common properties to the user-model, which means that both `Customer`s and `Employee`s have first and last names.

6.2 Interfaces

Persistent Storage

The `PersistentStorage` subsystem provides an interface `IPersistentStorage`, which gives access to creating, reading, updating and deleting `DriveIT` entities.

Implementers, Extenders, and Users

The implementer of `IPersistentStorage`, `EntityStorage`, uses an instance of `DriveITContext` to accomplish its tasks. The main purpose of the `DriveITContext` is to support the functionality of `EntityStorage`. Other implementers are free to support the functionality of the methods of the `IPersistenStorage` in other ways.

Visibility

Every method in the `IPersistentStorage` is public to users. The methods in the interface are expected to be implemented, and users of the interface should not expect otherwise. The `EntityStorage` implementation has few private methods exclusively for internal use, since their purpose is very specific to the functionality of the implementation.

6.3 Design Patterns

Model View ViewModel Pattern

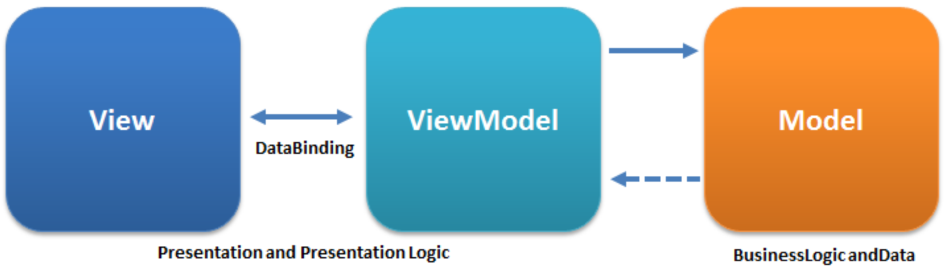


Figure 6.1 – The Model View ViewModel Pattern

The `DriveIT Windows Client` uses the *Model View ViewModel*[1] (hereafter *MVVM*) architectural pattern which tries to ensure a clear separation between the *Model* and the *View*. The pattern derives from both the *Model View Controller* pattern and the *Presentation Model* design pattern. Therefore it has some of the same attributes. By using the *MVVM* pattern we created a more testable and easier expandable application.

Furthermore we used *Microsoft Blend*'s[2] built in *Actions*[3] to encapsulate calls and commands from the `View` such that `View` elements would be sent to the `ViewModel`. These actions are implemented to ensure low coupling.

A built-in part of *MVVM* is a version of the *Observer-Pattern*[4]. This is done by creating data-bindings between the `View` and their `ViewModel`. In *WPF* this is achieved by having the `ViewModels` implement the interface `INotifyPropertyChanged` which will notify the `View` when the `ViewModel` has changed. By using this interface the `ViewModel` knows nothing of the `View` and cannot directly change its attributes.

[1]`http://en.wikipedia.org/wiki/Model_View_ViewModel`
[2]`http://en.wikipedia.org/wiki/Microsoft_Blend`
[3]`http://blogs.msdn.com/b/expression/archive/2009/03/23/an-introduction-to-behaviors-triggers-and-actions.aspx`
[4]`http://en.wikipedia.org/wiki/Observer_pattern`

Adapter Pattern

The *Adapter pattern* allows the developer to encapsulate methods and properties of an object in an "Adapter" class. Since the `DriveIT Windows Client` uses the *MVVM* design pattern, and the `DriveIT Web API` uses *DTO*s to send and receive data, and *DTO*s are only meant to transfer data, an Adapter Pattern fits perfectly. Almost all single entitiy `ViewModel`s in the `DriveIT.WindowsClient.ViewModels` namespace function as adapters for their corresponding *DTO*. E.g the `CarViewModel` class is an adapter for the `CarDto` class.

This implementation provides the `DriveIT Windows Client` with an easy way to manipulate data, while still retaining the data structure such that entities can be created, read, updated, and deleted over the `DriveIT Web API` without the having to convert client classes to API classes.

Model View Controller Pattern

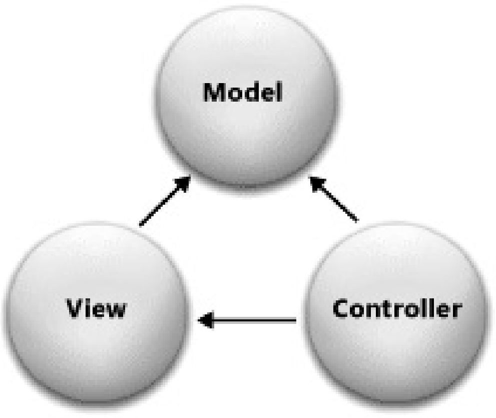


Figure 6.2 – The Model View Controller Pattern

The `DriveIT Web Client` uses the *Model View Controller*[5] architectural pattern (henceforth *MVC*), which separates the business logic, called *Model*, and the input control, called *Controller*, from the display logic, called *View*.

The *Model* is the part of the application that will be handling the logic for the application data. The *Model*s are being used to store data from the

[5]`http://msdn.microsoft.com/en-us/library/dd381412`

database. The *View* is the part of the application that will be handling the representation of the data. Mostly this data will be given by the *Model*. The *Controller* is the part of the application that will be handling the user interaction with the `DriveIT Web Client`. The *Controller*s read the user input from the *View*s and updates the state of the *Model*.

The *MVC* pattern allows for separation of the *Model*, *View*, and *Controller* classes, such that their individual purposes are clearly distinct from each other. This means that different developers are able to work on each of the individual parts of the *MVC* pattern, e.g we are able to work on the *View* without depending on the business logic or the input.

By making this distinction between different parts of the *MVC* pattern, it is easier to test the `DriveIT Web Client`, and retain a good data structure throughout the development of the `DriveIT Web Client`.

Façade Pattern

The *Façade design pattern*[6] is used several times in the `DriveIT System`. The `Persistent Storage` subsystem uses a façade pattern to hide the internal subsystems that accomplish the functionality defined in the `IPersistentStorage` interface.

The `DriveITContext` in the subsystem is used by the implementer of the `IPersistentStorage` interface, `EntityStorage`, but is hidden for users of the interface.

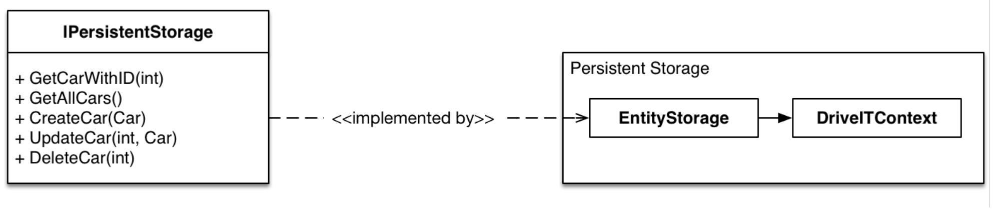


Figure 6.3 – The Facade Pattern of IPersistentStorage.

[6]`http://en.wikipedia.org/wiki/Facade_pattern`

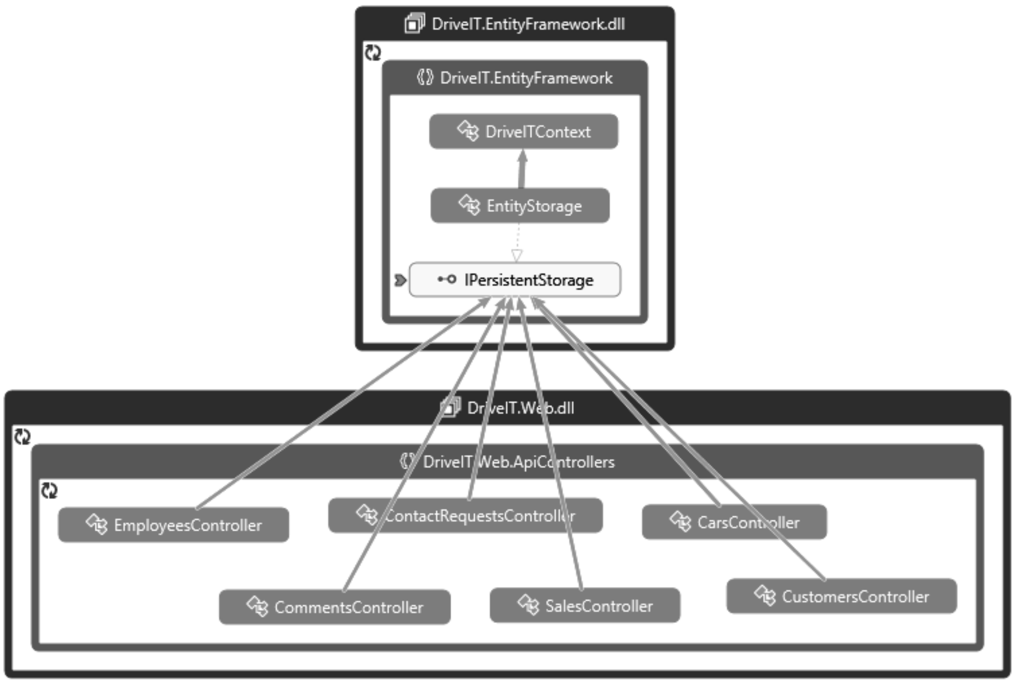


Figure 6.4 – The Facade Pattern Used by the `DriveIT Web API`.

Part IV

Test Document

CHAPTER 7

Test Plan and Results

7.1 Overall Test Approach

One of the non-functional requirements and design goals of the `DriveIT` system was to test critical sections of the system.

This has been accomplished by using a bottom-up testing approach. The deeper layers of the system providing data have been unit tested. Other subsystems of the `DriveIT System` building on top of these unit testes subsystems have been integration or system tested.

Trusting that the lower layers of the system function correctly enables that these can be used directly for integration testing.

7.2 Unit Testing

PersistentStorage Unit Testing

A subset of the functionality of the `PersistentStorage` subsystem has primarily been unit tested.

As mentioned earlier the system supports create, read, update & delete functionality of the entities of the `DriveIT System`. This functionality is very similar for the different entities, which is why testing has mainly been done for a specific entity type, namely `Sale`.

Focusing the testing on the underlying frameworks used in the classes does not make sense, since trusting that the developers of the *Entity Framework* have done extensive testing allows testing the logic in the `PersistentStorage` subsystem instead, which saves time and allows focusing on testing specific `DriveIT System` logic in depth.

Isolating the `PersistentStorage` logic requires setting up a mock[1] of the `DbContext` and `DbSet` classes of *Entity Framework* to be able to test

[1] `http://en.wikipedia.org/wiki/Mock_object`

that the logic calls the right methods in these classes. The *Moq*[2] framework provides relatively easy mocking of these classes.

A list of `Sale` objects is used as the input to a `Mock` object that is set up to provide the same functionality as the `DbSet` and `DbContext` classes. Tests can then be run on these classes.
The *Moq* framework provides methods to verify that a method has been run on the `Mock`, thereby confirming that the method performed the correct action.

The *Moq* framework has no ability to mock extension methods, which turned out to make testing a great deal harder. Since the implementation of `IPersistentStorage` relies on the extension methods in the *.NET Framework* to include certain results when retrieving entities, this proved to be too hard to test.

If testing was to be thorough, static methods of the *Entity Framework* would have to be implemented by the `DriveIT` developers, which was outside of the scope of this project.
Therefore some functionality of the system could not be tested.

DriveIT Web API Unit Testing

The `DriveIT Web API` consists of several `APIController`s, which are responsible for one entity each. The following `Controllers` have been tested:

- CarsController
- CommentsController
- ContactRequestsController
- CustomersController
- EmployeesController
- SalesController

The `AccountController` has not been tested because of time constraints and due to the fact that it was mostly auto generated by Visual Studio.

[2]`https://github.com/Moq/moq4`

The controllers have been tested with branch coverage, except for the places mentioned in the section below.

As mentioned earlier, one important thing to remember when testing is to test the implemented logic, and not the framework surrounding it. Therefore we have tested the `DriveIT Web API` using mocks of the repository and by calling the methods directly instead of using *HTTP* requests. Some parts of the `Controllers` cannot be tested this way, which is the parts where the `Controllers` use the `ModelState` to check the request for validity, but since this is a built in part of *ASP.NET Web API* it is assumed that these checks work as expected.

Methods without parameters, `GET` methods, are tested to see if they return the right data. Other methods are tested with valid and invalid input to see that the `Controllers` handle this data correctly.

All unit tests should succeed after every extension or refactoring of the `DriveIT Web API`.

The test suite is a collection of test drivers, used to test the functionality of the `Controllers`. Mocks have been used for tests stubs for mocking `IPersistentStorage`. The tests use a mocked IPersistentStorage which has been setup to support the required functionality using the *Moq* framework. The unit tests of the `DriveIT Web API` are located in the `DriveIT.Web.Tests.ApiControllers` namespace.

In the current version of `DriveIT Web API` the tests have completed with success. This does not mean that the `DriveIT Web API` is bug-free, but it provides more confidence that the controllers work as intended.

DriveIT Windows Client Unit Testing

By using the *MVVM* architectural pattern, the application is highly testable since the business logic is separated from the *View* and therefore does not require any actions done in the *View*. Due to time limitations not all `ViewModels` have been tested, but the image gallery functions of `CarViewModel` and the `PasswordCreationViewModel` have been tested and work as an example of how one could extend the test suite to cover the entire `ViewModel` subsystem.

For the tests of the two components described above, whitebox testing with branch coverage has been used, e.g. the `NextImage` functionality in the `CarViewModel` has been whitebox tested with branch coverage.

This is done by the test methods `NextPictureWithOnePicture`, `NextPictureWithTwoPicture`, `NextPictureWithThreePicture`. Here there are three different scenarios, either the image gallery index does not change, the index swaps between two different values, or index increments until it hits the last index and then returns to the first.

Another example of white box testing with branch coverage is visible in the `PasswordCreationViewModel`. Here the `CheckPassword` method is tested. Since the method contains a number of if-statements, the testing must make sure that every if-statement is entered. For this a test method is created for each of the if-statements and therefore we reach branch coverage on that method.

All tests in the `DriveIT Windows Client` pass the component unit-tests.

7.3 Integration Testing

When implementing a system which is based on having a separation of application and storage, integration testing is of utmost importance. In this system we have both done both automatic and manual integration testing.

The `DriveIT Windows Client` has a fully automatic integration test suite of the CRUD functionality for the entities `Car`, `ContactRequest`, `Employee`, `Customer` and `Sale`. The tests are not properly formed and could be improved in many ways, but the results can be used to indicate if there are problems with the integration with the `DriveIT Web API`. Since the `DriveIT Windows Client` is able to switch between the local database and *Micrososft Azure*, the testing can also be set up to test both of these environments.
The `DriveIT Web Client` does not contain automatic integration tests, but extensive testing based on the functionality which is provided to the `DriveIT Web Client` by the `DriveIT Web API` has been executed manually.

The most significant methods ensuring the functionality of the `CarQuery` class have primarily been integration tested.
Retrieving information as *JSON* and transferring this data to *DTO* objects has been tested to ensure that this functionality does not have any errors.

Testing that data is received and that this data is correct is tested by requesting a known data set, e.g. `"make=ford"`, and checking that the received data is of the anticipated type.

Boundary testing is performed by sending a malformed `CarDto` object as a parameter to the class and checking that the expected exception is thrown.

7.4 System Testing

The `DriveIT System` has been under continuous testing throughout development.

As soon as the working prototype had been developed, the different components of the system were put together and system tested. By having a bottom-up test approach the system tests were mainly focused on during the end of development after all lower sub systems had been unit and integration tested.

Scope

The scope of the system tests was primarily focused on the main scenarios and use cases. These should be supported so that users are able to complete their tasks.

Testing every single area of the system was not in scope, since this would take a very long time that was not available. Main functionality and use cases were the focus points of the testing.

Approach

The `DriveIT Web Client` and `DriveIT Windows Client` have been tested against the specified use cases in the Requirement Analysis Document to see whether every the use cases are possible to complete inside the functional and non-functional requirements.

At the end of development the near-finished `DriveIT System` was system tested by the developers. The developers sat down and opened a release version of the `DriveIT System`.

Every requirement from the Requirement Analysis Document was tested to see whether the system fulfils the functionality specified.

Minor errors were found and fixed, e.g. long image path strings would not be persisted, because of database limitations. After fixing this bug by only using *GUIDs*[3] as file names, system tests were run again and passed successfully.

[3]`http://en.wikipedia.org/wiki/Globally_unique_identifier`

When a use case was completed in an acceptable fashion the test was deemed as passed.

Requirements

The systems were tested so that it was possible to complete every use case while staying inside the functional and non-functional requirements.

Results

Both the `DriveIT Web Client` and `DriveIT Windows Client` have passed the tests.

7.5 Acceptance Testing

Since the project has no real product owner, acceptance testing was done at each *SCRUM* sprint by the team members. At the end of the last sprint, *ref. 9.3*, two acceptance tests were made, one for a potential employee of the used car dealership who used the `DriveIT Web Client` and the `DriveIT Windows Client`, and another for a potential customer who used the `DriveIT Web Client`.

Employee - Acceptance Testing

The acceptance testing for employees are based on the scenarios: `Sell a Used Car to a Customer` *Sec. 3.1*, `Put a New Used Car Up for Sale` *Sec. 3.1*. To test these scenarios a bachelor student of Digital Media and Design at the IT University of Copenhagen was found. This student is not the perfect representative of a normal car salesman, but the feedback still provides some valid information.

Feedback

A couple of the notes taken from the session can be seen below:

- The connection between the website and `DriveIT Windows Client` works fine.
- When creating a new entity, the message indicating that the next time you press the "Create/Update" button suddenly *updates* the entity is too subtle.

- Data samples of the different properties in the `EntityViews` could be provided.
- When converting a `ContactRequest` into a `Sale` it is unclear if the `ContactRequest` gets deleted or if the `Employee` must do it himself.

Overall the acceptance testing was successful and showed that the system works as intended.

Customer - Acceptance Testing

The acceptance testing for customers are based on the scenarios and use case: `Create Account` *Sec. 3.2*, `Comment on a Car` *Sec. 3.1*, `Customer Wishes to Find a Car` *Sec. 3.1*. To test these scenarios a bachelor student in Software Development at the IT University of Copenhagen was found. This student is not the perfect representative of an everyday customer, but the feedback still provides some valid information.

Feedback

Here are a couple of the notes we took from the session.

- The website is easy to understand.
- Units are not provided on properties - this causes some confusion.
- Password creation is difficult since password requirements are not provided to begin with.

Overall the acceptance testing was successful and showed that the system works as intended.

Part V

SCRUM Documentation

CHAPTER 8

SCRUM

8.1 SCRUM Organisations

The team had an introductory meeting at the very beginning of the project. This meeting was not a *SCRUM* meeting, but rather a meeting to make a cooperation agreement, in compliance with the available time and expectations of the team members. At this meeting the first proper SCRUM meeting was scheduled and took place a few days later.

Subsequent to the first *SCRUM* meeting, the *SCRUM* process was generally followed. Stand-up meetings took place every morning at the scheduled time. At the sprint start meeting, tasks were assigned, sprints were planned, and at each sprint end meeting, sprints were evaluated and finished work was showcased.
Using the *SCRUM* planning tool in *Visual Studio Online* greatly benefit the team, since no post-its etc. were needed for planning tasks and backlog items, though a whiteboard still came in handy.

The team members each took turns acting as SCRUM master.
During the project the different members were involved in different areas of work, not only focusing on one aspect of the project. Still, each team member had a main area of focus.

Sprints were each a week long, began on Wednesdays and ended on Tuesdays.
This gave the team the ability to work over weekends, where some members had more time to work. The retrospective meetings and the sprint planning meetings could be dealt with over the course of one day, which had the benefit that members could get straight to work the next day.

Presentations of the finished work and sprint retrospection usually happened in one meeting. This happened quite naturally since discussions of how the work had gone over the course of the sprint usually started after showing off the work.

8.2 Sprints

After each sprint the team sat down for a sprint end meeting. We followed the procedure outlined in the *SCRUM* documentation[1] on how to carry out the meeting.

After reviewing the sprint retrospections some issues required more work to fix during the development of the system.

Writing the documentation in LaTeX from the beginning turned out to be harder for some group members, since they did not have prior in depth experience with the syntax.

Staying on top of updating the documentation while developing the code base of the `DriveIT System` was harder than first estimated. Some major revisions had to be made to the documentation near the end of the project, since they had not been updated during development.

Finding a fitting amount of work for the team members took some time, especially due to the time team members had available for working on the project exclusively changed over the course of the project. In general time estimates were correct, and the team did not face any major issues regarding the workload until the very end of the project.

Using Git as a version control system was easy and fit the team well. Branching was used extensively and helped avoiding overwriting other team members' work.

Stand-up meetings worked well, and beginning a day's work was made easier by knowing which team members did what.

The full summaries of the Sprint End meetings, including the following Sprint retrospections, can be seen in Appendix 9.3.

8.3 Burndown Charts

The team made effort of trying to estimate every task they added and worked on during each sprint.

This enabled *Visual Studio Online* to create graphical burndown charts of the completed work. These provided a nice overview of how the sprint went at every retrospection.

[1] `http://en.wikipedia.org/wiki/Scrum_(software_development)`

Sprint 1

The burndown chart after sprint one, figure 8.1, reflects how the team stayed very close to within the ideal trend.

Team members were not assigned an extraordinary amount of tasks and they completed the tasks within the set time frame.

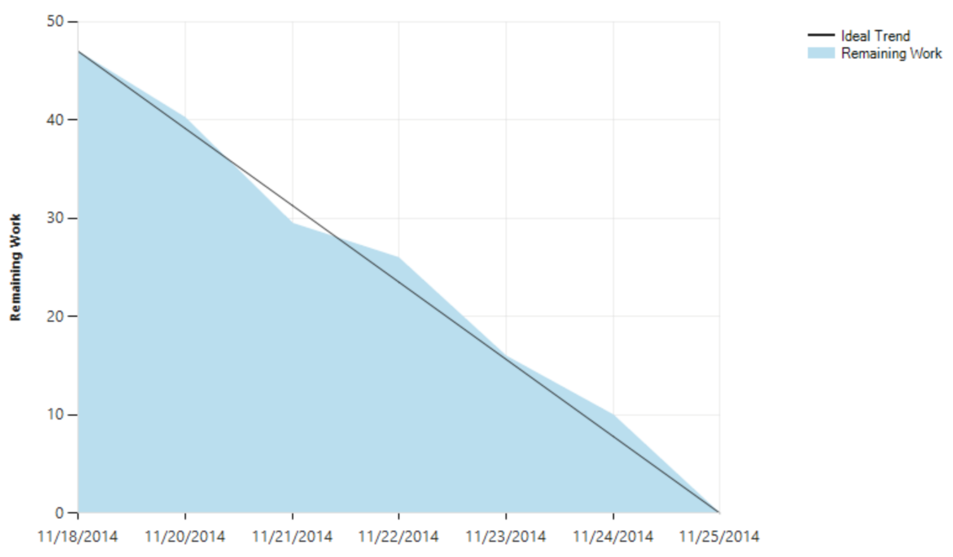


Figure 8.1 – Burndown Chart of Sprint 1.

Sprint 2

The burndown chart after sprint two figure 8.2 reflects how the team changed tactics.

It was decided at the sprint planning meeting after sprint one to have an aggressive amount of work assigned to every team member, in order to complete a larger amount of work.

Team members discussed not being able to complete *every* task and focus on having a working prototype ready for the end of the sprint.

Note the total amount of hours at the top of the chart and the hump near the end. Also note that the team did not complete every assignment, but got the main tasks out of the way.

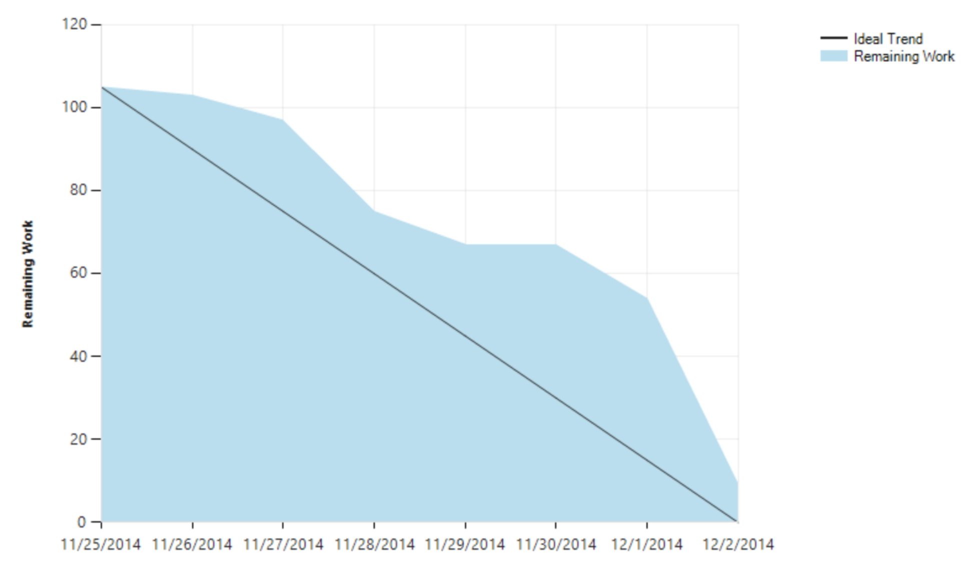


Figure 8.2 – Burndown Chart of Sprint 2.

Sprint 3

The burndown chart after sprint three, figure 8.3, reflects how the team found a reasonable amount of work.

After having finished a working prototype the workload shifted to a more appropriate amount despite team members working more. This reflects that the team found out how much work they could complete in a given amount of time.

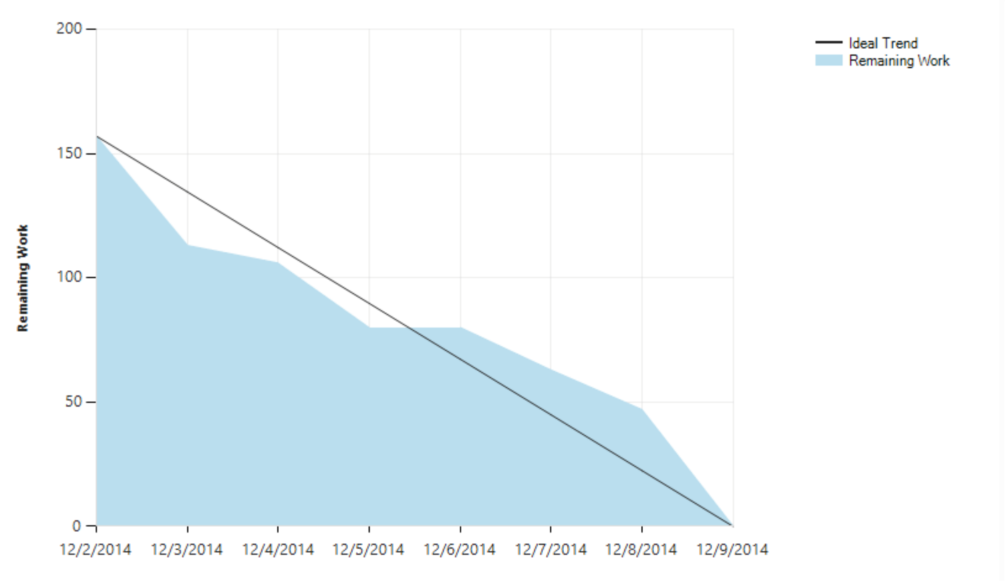


Figure 8.3 – Burndown Chart of Sprint 3.

Sprint 4

The burndown chart after sprint four, figure 8.4, reflects how the team didn't use the online SCRUM tool and instead worked on items they felt needed work to be finished. The burndown chart reflects this in that it does not follow the expected graph. All work was finished, but sometimes not put into the system.

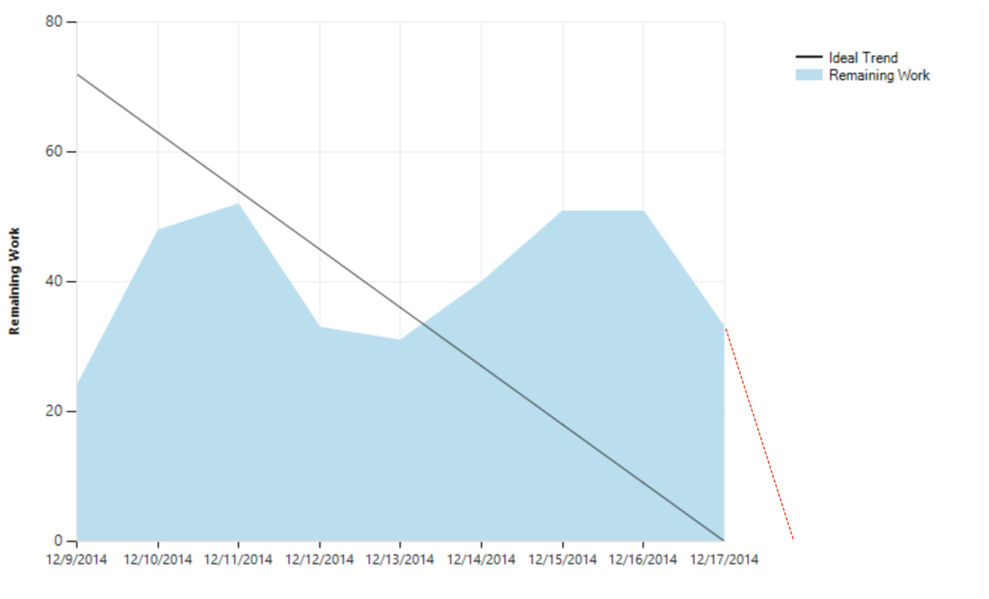


Figure 8.4 – Burndown Chart of Sprint 4.

8.4 Review and Retrospective

Generally the retrospective meetings were finished very quickly.

The team worked quite well and did not face any major issues during the sprints.

The main issues found were the work estimates, where some tasks took longer or less time than expected and some team members allocated either too much or too little to themselves.

Focusing on the documentation in the first sprint gave the team a solid foundation for the ensuing programming sprints, which was a great benefit.

Using *SCRUM* proved to be very effective in planning the work. Having shorter sprints helped keep spirits up for team members, and ensured that everyone did not drown in work. The team was always able to see the end of the next sprint and how much work was left.

Using an iterative approach was a benefit and the team appreciated the ability to change course during the project.

Having a waterfall-like approach would not have given the team the same abilities to do this, and unexpected problems that were discovered during the project might not have been as easy to fix.

The fact that the team was relatively small and therefore well-suited for the *SCRUM* methodology helped a great deal as well. Had the team been larger other issues might have surfaced, which the waterfall approach would have handled better.

At the end of the project the team got a bit more relaxed assigning tasks before starting work. This resulted in some team members shifting focus, and working on tasks other than ones assigned to them. This resulted in a lack of overview, and some tasks getting pushed to end of the project.

This goes to show that sticking to the *SCRUM* principles pays off, which the team forgot in the end, and even though *SCRUM* seems as a more relaxed way of organising a team, you still need to follow the principles.

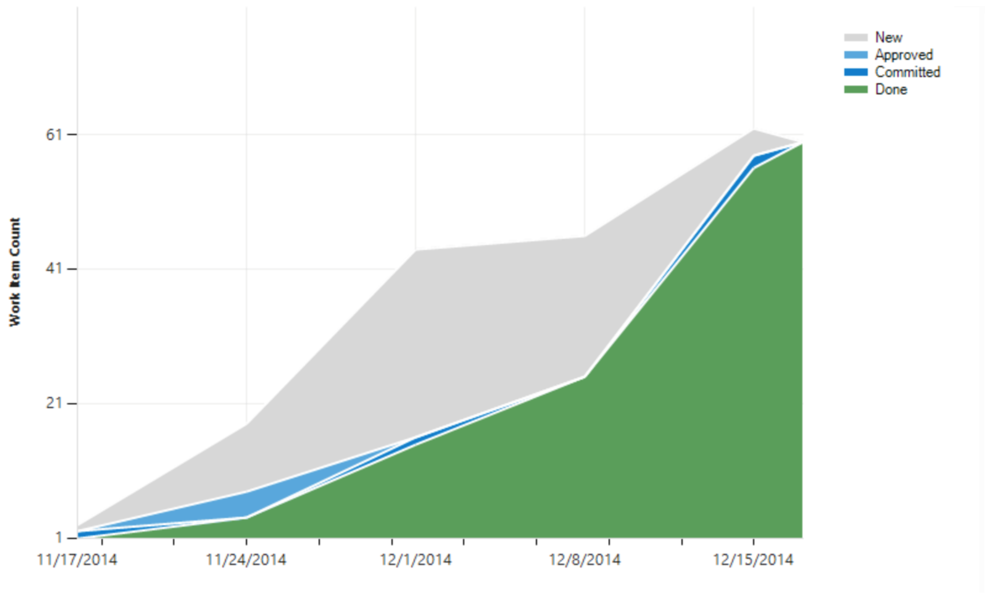


Figure 8.5 – Total work completed graph.

Figure 8.5 shows the graph of completed work. The graph reflects the fact that more work has been completed later in the time frame of the project. Furthermore it is visible that the use of the tags *Approved* and *Committed* has only been used minimally.

The project as an entirety has been a great success, and the *SCRUM* methodology has proven a most valuable tool.

Part VI

Appendices List

Chapter 9

Appendices

9.1 Appendix 1 - Distribution of Work

Report

- Introduction
  - Purpose of the System
    awis, afin
  - Scope of the System
    awis, afin
  - Non-Scope of the System
    awis, afin
  - Objectives and Success Criteria of the Project
    awis, afin
- Requirement Analysis Document
  - Proposed System
    - Overview
      afin
    - Functional Requirements
      jstc
    - Initial Analysis Objects
      jstc
    - Target Environments
      mlin
    - Non-Functional Requirements
      awis, jstc
  - System Models

    - Use Case Models
      jstc, (all)
    - Use Cases
      jstc, (all)
    - Domain Object Models
      afin
    - Dynamic Models
      ppma, (afin, awis)

- System Design Document
  - Proposed Software Architecture
    - Overview
      afin
    - Design Goals
      awis, jstc
    - Subsystem Decomposition
      all
    - Hardware and Software Mapping
      mlin
    - Persistent Data Management
      afin
    - Access Control and Security
      ppma, mlin
    - Global Software Control
      awis, cnbl
    - Boundary Conditions
      awis, cnbl
  - Subsystem Services
    ppma
- Object Design Document
  - Object Design and Patterns
    - Object Design
      all
    - Interfaces
      afin
    - Design Patterns
      afin, awis, cnbl, mlin, ppma

- Test Document
  - Test Plan and Results
    - Overall Test Approach
      afin
    - Unit Testing
      afin, awis, mlin
    - Integration Testing
      awis
    - System Testing
      afin, (all)
    - Acceptance Testings
      awis, (afin)
- SCRUM Documentation
  - SCRUM
    - SCRUM Organisations
      afin
    - Sprints
      afin
    - Burndown Charts
      afin
    - Review and Retrospective
      afin

Code

- DriveIT.CarQuery
  afin
- DriveIT.CarQuery.Tests
  afin
- DriveIT.EntityFramework
  afin, mlin
- DriveIT.EntityFramework.Tests
  afin
- DriveIT.Models
  mlin

- DriveIT.Web.Api*
  mlin
- DriveIT.Web.Mvc* + DriveIT.Web.Views jstc, ppma
- DriveIT.Web.Tests
  mlin
- DriveIT.WindowsClient.Controllers
  awis
- DriveIT.WindowsClient.ViewModels
  awis, cnbl
- DriveIT.WindowsClient.Views
  awis, cnbl, (afin)
- DriveIT.WindowsClient.Tests
  awis

9.2 Appendix for unspecified use case models

List of use cases not specified further.

- Contact interested customer
- Update car information
- Remove car from car lot database
- Search for cars that are up for sale
- Search for customers
- View orders
- Register a calling customer in the system
- Customer wants to delete account
- Customer wants to see cars with some specified specification
- A new employee is hired and must be added to the system
- An employee stops working at DriveIT and must be deleted from the system
- Information about an employee must be updated.

9.3 Appendix for Sprint Retrospections

The main points to consider during Sprint Retrospections were *"What Could Be Improved?"*, *"What Went Well?"* and *"Presentation of Work"*.

Sprint 1: Retrospection

Facilitated by Jacob Stenum Czepluch (SCRUM master)

What Could Be Improved?

- Mikael would like to be able to see stuff in VS Online. He had no access.
- LaTeX is horrible, no one likes using it.
- TeX templates are weird and we spent time figuring them out.
- Markdown would make (at least Fischer's) life easier.
- LaTeX conventions need to be specified and **used**. This takes time.
- Wind does not want to be the "tex-guy". Everyone needs to LaTeX their own stuff.

What Went Well?

- Work went well.
- People managed their time well.
- The workload will increase when coding starts.
- Facebook group worked really well.
- Tasks and SCRUM interface online works well.
- Graphical online interface gives a nice overview.
- Git works, branching is going to be interesting.

Presentation of Work

We have no product owner, so the group therefore pretended to be the group owner while also being the developers of the product.

All group members have checked the work together, and everything looks the way it should.
Minor improvements have either been made or planned and added to the next sprint.

Sprint 2: Retrospection

Facilitated by Anders Wind Steffensen (SCRUM master)

What Could Be Improved?

- Everyone would like a bit more time, but still felt they finished the most important work.
- Git branching is okay, some people still don't quite get it, but we have not had any major conflicts.
- We forgot the documentation a bit this time, focusing more on code.
- Some group members underestimated the time they thought they would spend to finish a task.
- We need to remember to update the SCRUM online tool so that the burndown chart is correct.
- Visual Studio can be a bitch. There's too much black magic.

What Went Well?

- Work went well. We managed to accomplish the major goals (working prototype).
- People managed their time well.
- Facebook group still works really well.
- Git is awesome when it works.

Presentation of Work

We have no product owner, so the group therefore pretended to be the group owner while also being the developer of the product.

All group members have checked the work together, and everything looks the way it should.
Minor improvements have either been made or planned and added to the next sprint, with mostly the same people assigned.
Work people didn't have time to finish have been added to the next sprint.

Sprint 3: Retrospection

Facilitated by Anders Fischer-Nielsen (SCRUM master)

What Could Be Improved?

- We need to stay on top of updating the SCRUM tool (VS Online).

- We need a better delegation of work. Some group members relaxed a bit.

    Better time estimates could fix this.

- The placement of big tasks (that affect other parts of the product) should be changed so that these happen at the beginning of the sprint. That should free up other team members so that they are less stressed by the end of a sprint.
- We need to get better at attending the stand up meetings.
- Noone should be working at the SCRUM meetings. NO CODING!

What Went Well?

- The group is somewhat following the time frame for the product. We're not *too* far behind so far.
- Time estimates are generally okay, everything considered.
- The communication of the group is fine. The Facebook group is working beautifully.

Presentation of Work

We have no product owner, so the group therefore pretended to be the group owner while also being the developer of the product.

Some group members have presented their finished work and eveything that people finished in time looks good.
Some tasks were not completed, and did not work, when we tried to show them off. Minor improvements have either been planned and added to the next sprint, with mostly the same people assigned.
The work group members didn't have time to finish have been added to the next sprint.

Sprint 4: Retrospection

Facilitated by Anders Fischer-Nielsen (SCRUM master)

What Could Be Improved?

- Assigning tasks completely failed this sprint. Group members worked on whatever they felt was missing.
- We need a better delegation of work. Since members had no work assigned, roles were never assigned.

What Went Well?

- Considering no tasks were delegated all the work was completed without major issues.
- Work was finished in time in a satisfactory manner. Of course there were some things that could have been fixed post-project, but they are outside of the scope.

Presentation of Work
We have no product owner, so the group therefore pretended to be the group owner while also being the developer of the product.

The team went through the finished work together, trying to find any missing requirements. No major issues were found. The work was deemed satisfactory (shippable) considering the time frame.

9.4 Appendix for Github Repository

All code is available on the private Github Repository: `https://github.com/andersfischernielsen/DriveIT`. Invitations to join the repository will be sent to Jakob Bardram and Rasmus Nielsen.