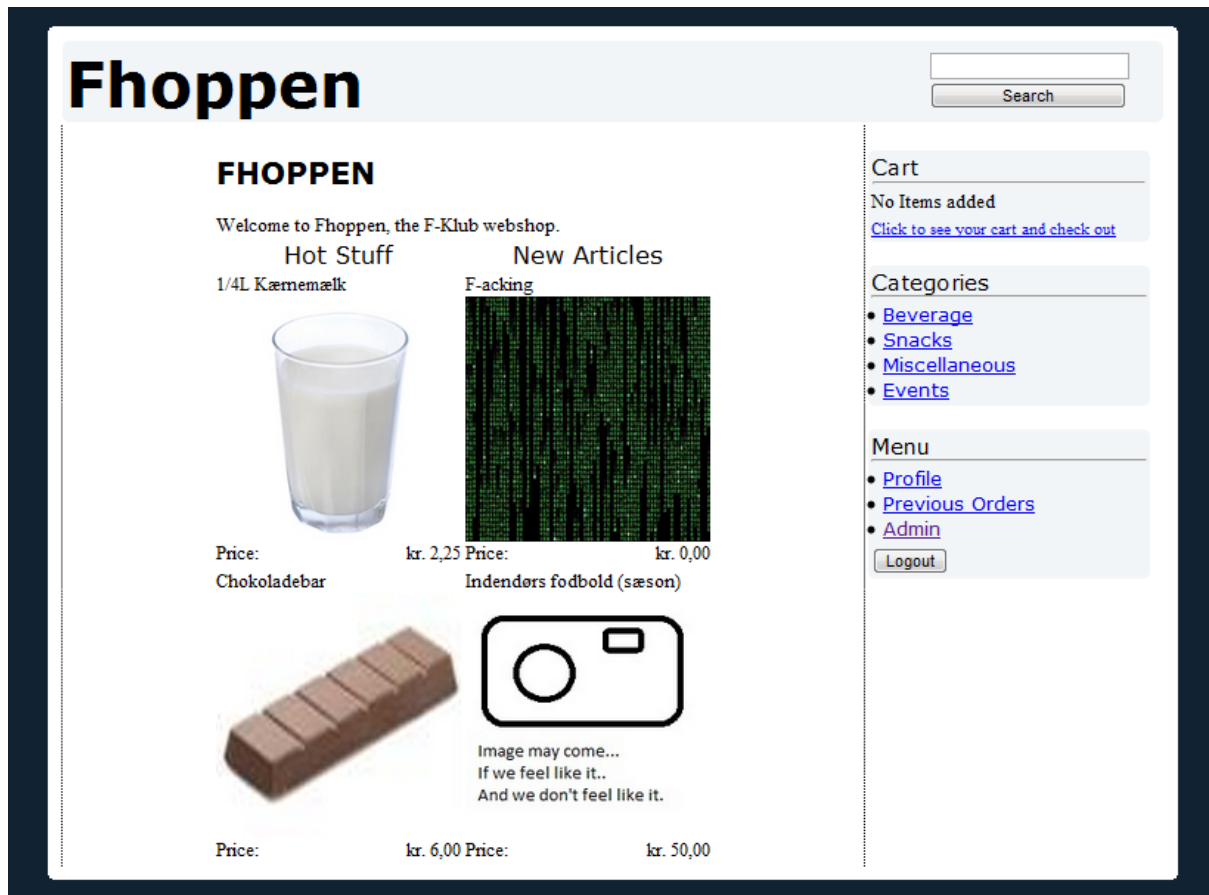


Developing FHOPPEN



a webshop for the F-Klub

Developed by s302a, Aalborg University, 2009

Title:
FHOPPEN

Project theme:
Programming

Project period:
Fall Semester 2009,
September 2nd
to December 18th

Project group:
s302a

Participants:

Nikolaj Andersen

Anders Frandsen

Sergey Gurkin

Rune Jensen

Kasper Lang

Michael Lisby

Supervisor:
Gao Cong

Print run:
8

Pages:
111

Appendices (number, type):
Blank

Department of Computer Science

Selma Lagerløfs Vej 300

DK-9220 Aalborg Ø

Telephone (+45) 9635 8080

<http://www.cs.aau.dk>

Abstract:

The goal of this semester's project has been to make a webshop. This has been done by analyzing and designing a fictional webshop, and optimizing this, by using object oriented analysis and design.

The webshop has been developed by using the ASP.NET object oriented programming language, along with Microsoft® SQL Server. The project has also focused on setting up and using a relational database, to store needed data. In the development phase, as well as the writing process of this report, the agile development method Scrum has been used. The conclusion of this report discusses the further advancement of the system and discusses some of the errors made during development.

The contents of the report are freely available, however publishing (with source) must happen only by agreement with the authors.

Preface

This report has been developed by a group of 3rd semester Software Engineering Students at Aalborg University, in the fall of 2009. It documents the development process of a webshop solution for the F-Klub and the theory behind it. To be able to fully understand the topics and dilemmas of this report, the readers are required to have a general knowledge of programming¹ and a basic understanding of the ideas presented in Object-Oriented analysis and design (OOAD)[1].

The report is made up of four major parts. First, any required theory will be explained in detail. The theory chapters of the report are based on several theory books. It will be stated in the introduction of each chapter which books it is based on, except for the programming chapter which is based on knowledge gained during the programming course of this semester, and direct citations will be noted throughout the chapters. This will be immediately followed by the analysis which will be based on the theory described in the first part. At the end of the analysis chapter the reader will be introduced to the problem statement for the project, and the system design will be explained. The third part of the report contains the description of the solution and the process of developing it. The fourth and final part of the report contains a conclusion and a discussion chapter.

Reading Instructions

While reading the report, the reader should be aware of the following:

- When mentioning *he*, the actual meaning is *he or she*, same with his or hers. Generally any person will be denoted *he*, simply to save space.
- Special abbreviations used are:
Object Oriented Analysis and Design - OOAD
Object Oriented Programming - OOP
Graphical User Interface - GUI
Entity-Relationship Model - E-R Model
- When referring to various types of literature, the number notation will be used. For example, the reference to the Object Oriented Analysis and Design book will look like [1]. This number can be checked in the literature list at the end of the report to see which book it refers to.
- All images inserted in the report will be numbered in the ascending order with a reference to the chapter, in which the image is inserted. For example, if chapter 4 contains five images, the number of the fifth image would be 4.5. All inserted images will also contain a description text.

¹This implies general structures, for- and if-statements, for instance.

-
- The appendix can also be found on a CD-ROM attached to this report, which furthermore contains the source of the implemented software and the documentation of the implemented system. Also, this report is added in pdf-format. A dump of the database in SQL-format, and an XLS-file with the same data, has also been added to the CD-ROM.
 - It is possible to access the actual website at Cassiopeia in cluster 2 on "inquiz-itzerver:25000". If this creates any problems, feel free to contact s302a@cs.aau.dk.

CONTENTS

1. Introduction	8
I Theory	9
2. System Analysis Theory	10
2.1. System Definition	10
2.2. Problem Domain	11
2.3. Application Domain	15
2.4. Evaluation	19
3. System Design Theory	20
3.1. Criteria	20
3.2. Components and Architecture	21
3.3. Model Component	22
3.4. Function Component	23
3.5. Component Connection	24
4. Programming	25
4.1. Object Oriented Programming	25
4.2. C#	25
4.3. Unit Testing	28
4.4. ASP.NET	29
5. Databases	36
5.1. History	36
5.2. Relational Database Theory	36
5.3. Structured Query Language	38
6. Agile development and Scrum	45
6.1. Agile development in software engineering	45
6.2. Scrum	46
II Analysis and Design	49
7. Analysis of the Proposed F-Klub Website	50
7.1. System Definition	50
7.2. Problem Domain	51
7.3. Application Domain	57

7.4. Analysis Summation	63
8. System Design	65
8.1. The Task	65
8.2. Criteria	65
8.3. Technical Platform	66
8.4. Architecture	66
8.5. Components	67
8.6. User Interface	74
 III Product	 76
9. Development	77
9.1. Development Phase	77
9.2. Agile Development in this Project	77
10.FHOPPEN	80
10.1Product Walkthrough	80
10.2User Interface	80
10.3Classes	81
10.4Encryption	89
10.5Database	90
10.6Architecture	92
 IV Conclusion	 94
11.Perspective	95
11.1.The FHOPPEN webshop	95
11.2Architecture	96
11.3Knowledge Gained	97
11.4.Work Process	97
12.Conclusion	99
 V Appendix	 101
A. User Interface Suggestions	102
B. SW3 Project Proposal	108

Introduction

According to [2], a shop is:

"A mercantile establishment for the retail sale of goods or services."

Earlier, a customer needed to go the physical location of the store, in order to make a purchase. To complete a purchase, an employee also had to be present to receive the payment.

The digital age has created the possibility for letting a customer interact with a shop, without having an employee present. It has even become possible for the user to access the shop from home, and order and pay for his articles without visiting the store physically.

The social club at the F-sector of Aalborg University (the F-Klub), sells various items at the IT-department Cassiopeia. This is done by the use of "Stregsystemet", an IT-system. Every member of the F-Klub has his own account, on which he can deposit money, that can be used for "Stregsystemet". This is because he can only buy items, if there are money on his account. The system is self-serviced, as the user enters his username, and an identification number (id) according to the item he wants to buy.

"Stregsystemet" works as intended, however, the F-Klub wants to add additional functions and items to this system. It would be interesting to examine, how this system could be extended, but because the current system seems primitive, as the user has to type in the id of the item he wishes to buy, it would make more sense to make a webshop instead. The reason for this would be, that the user can access the webshop from the internet, and he would not need to enter the id of the item he wants, but simply click on a few buttons, then he would have bought the item.

Therefore, the initial problem of the project is formulated as:

How does one create a functionally pleasing webshop from scratch and what does one need to know to be able to design it?

Part I

Theory

System Analysis Theory

This first chapter of the theory part will go through the basics of a system analysis. First, the basics of a system definition will be covered, including an easy way to create one. This will be followed by a brief description of how classes and objects are used in Object Oriented Analysis and Design (OOAD). The process of creating the necessary functions of a system, by looking at aspects such as usecases and actors, will be discussed in detail. Also any relevant background theory will be described, including, what a use case is and how to create one. In the end, a structure of a finalized analysis document will briefly be described and shown in detail - which will also be used as a basis for the system analysis made in this project. The chapter is based on the OOAD-textbook[1].

2.1. System Definition

When creating a software application, it is important to thoroughly analyse the current situation, in order to create an application that is adjusted to the customer's needs. For example, an application for a bank could be responsible for managing all customer transactions, but it could also be necessary to monitor stocks and exchange rates. Therefore, a system definition should be created to give a clear view of what the system will need.

A system definition is a short and exact notation that describes the system to be developed. This includes; the information it should contain and work with, a brief description of key functions, information on where the system is going to be used and which development conditions apply. The system definition should be one of the first things the development team creates when starting development on a new system, because it provides a clean and structured overview of the entire system and any possible system choices that needs to be made.

2.1.1. Application and Problem Domain

The system consists of two domains - a problem domain¹ and an application domain. The problem domain can, for instance, be a bank, a supermarket or a small, private company. Anything happening, or being prevented from happening, in this area, is said to take place in the problem domain. The idea is that the system should solve the problems in the problem domain, and make for a better working environment.

The system will be run in the application domain, which is defined as being the area where the system is managed and used. If the system for instance is an grocery store system, the application domain would be confined to the store office area. It would also stretch to the checkout counters as distributed clients. In the case of a webpage

¹The problem domain is explained in detail in section 2.2 on the facing page.

for instance, the application domain would be the server hosting the site and every computer accessing it.

These two domain definitions are important as they are used throughout both the analysis and design phase of the project. The application domain is mostly used in the design phase, whereas the problem domain is widely used in the analysis phase, as seen in the following FACTOR description.

2.1.2. FACTOR

An easy way to produce a good system definition is to use the FACTOR-criterion. It is best described as a template of what should be in the system definition. Filling out a FACTOR template allows one to easily write out a clear system definition by creating a structured text with the contents of each point. The FACTOR criterion is also a good way to create a system definition that appeals both to the customers and the developers since it incorporates aspects of both sides. The FACTOR template is made up of the following [1, p. 39-40]:

- **Functionality** - *The system functions that support the application domain tasks.*
- **Application Domain** - *Those parts of an organization that administrate, monitor or control a problem domain.*
- **Conditions** - *The conditions under which the system will be developed and used.*
- **Technology** - *Both the technology that will be used to develop the system and the technology on which the system will run.*
- **Objects** - *The main objects in the problem domain.*
- **Responsibility** - *The overall responsibility of the system in relation to its context.*

For a look on how a FACTOR template looks when finished, readers are advised to see the FACTOR of the project in section 7.1.1 on page 50.

2.2. Problem Domain

The problem domain, as explained in section 2.1.1 on the facing page, is the area or domain, where the problem(s), the system is supposed to solve, are in effect. Since it cannot describe every detail, it will instead be a model containing the elements that effect the problem primarily. The content of such a domain can be described using; classes, objects and events. This section will explain each of these, their functions and relation to each other. Finding these components in the problem domain can be difficult, but a good system definition depends on how accurately one can describe the problem domain. This makes it a very important point in one's research.

2.2.1. Objects, Events and Classes

An object is an entity in the problem domain, for instance, a blackboard. It is easy to recognize the blackboard as an object, but an object in the problem domain can also be a person or maybe even a company. The one aspect that all objects share, is that they have a unique identity in the problem domain. An object acts in the problem domain in a specific manner, be it static or dynamic.

An event, on the other hand, is something that happens involving one or more objects. This could be a salesperson selling wares to a customer, or a machine fabricating a new

Events/Classes	Customer	Salesman	Ware	Register	Shelf
Buying wares	x	x	x	x	x
Filling shelves		x	x		x
Counting register		x		x	

Table 2.1.: An event table.

sheet of metal. Normally these events would occur over time, but to simplify matters when analyzing, they are said to happen instantaneously. It also bears notice that an event can involve several objects in the problem domain. This can also be seen in the previous example where both the salesman and the customer are objects.

To simplify this, one can use an event table. This is a spreadsheet that gives the name of an event and then lists any and all objects involved in it. An example of such a table can be seen in table 2.1.

Furthermore, objects can be grouped together as a class which describes a group of object(s) that are similar to each other and any events they can partake in. For instance, four people, will have four distinct names uniquely identifying them, Bob, Markus, Tom and Daniel, which can be grouped together in the class person due to their common attribute "name". However, they can still partake in completely separate events.

As such, a class describes a collection of structure, behavioral pattern and attributes of the objects, or a single one. These are individual for each object unless there is talk of a clone.

Thus by using all three concepts; objects, events and classes, one can describe any activity occurring in the problem domain and get a structured overview of what is happening in the domain.

2.2.1.1. Finding Classes and Events

Finding classes and events, or more accurately, recognizing them in the problem domain, is not always easy. A good practice is that one should note down any and all possibilities and then sort them out later. Mostly this is done by observing the problem domain in its regular routine. The idea is to spot problems and distractions, and then design one's system to solve those.

This is most likely the best tool available, as the observer is usually new to the domain and therefore sees things from another point-of-view. One must not forget to talk to any persons who regularly visits the problem domain, for instance both salesmen and customers. This is to insure that any personal opinions is brought up by people who have spent a lot more time working in the problem domain than the developers.

Eventually one would have written down all possible objects and events, and should group them together in classes. This is also the time to weed out the non-neccesary objects or events, if any. This is an important step, as developers should not waste time working on unnecessary features.

2.2.2. Structure

To describe the relations between one's objects and classes, the term structure is used. It is a description of how everything in one's problem domain relates to each other. Correctly describing this will result in better understanding of the situation by giving a more coherent overview.

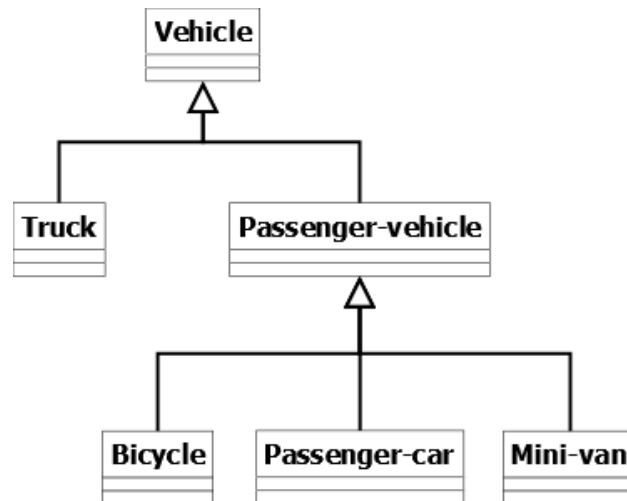


Figure 2.1.: The relations between Superclasses and subclasses.

To simplify the concept a bit, one uses only four kinds of relations, two for objects and two for classes. Understanding how these work is important to the analysis, as the design of the final system depends directly upon how those relations are defined.

2.2.2.1. Generalizations and Clusters

These first two terms are used to describe the relations between classes. A basic generalization structure consists of a subclass and a superclass. A subclass is a specialized class of the superclass, hence the superclass is a generalized class. For instance, a superclass could be named "Vehicle", specializing into the subclasses "Truck" and "Passenger-vehicle". One can see how the term "Vehicle" is a generalization of a passenger vehicle. The class "Passenger-vehicle" could in turn also be a superclass to the subclasses "Passenger-car", "Bicycle" and "Minivan". This is only an example, but it shows how the concept of generalization works.

Another way to describe the relations between the superclasses and subclasses is by using the concept of inheritance. The idea is that specialized classes inherit properties from the superclasses. Superclasses do not inherit anything from subclasses, but can inherit properties from other superclasses - effectively making it both a superclass and a subclass. This can also be seen in figure 2.1.

It is also worth noticing that multiple inheritance is possible in a generalization structure, meaning that subclasses can inherit properties from several superclasses if needed. But this does not complete the structure overview of one's classes. For that cluster are needed too.

A cluster is a collection of closely related classes that can be described under a common name. For instance, in the previous example, some of the classes can be grouped in a cluster as seen in figure 2.2 on the next page. This can also effectively divide ones problem domain into smaller subdomains. Each cluster then represents a subdomain of the problem domain. This is an effective way of gaining an overview of the entire problem domain as it can be rather large in some systems.

2.2.2.2. Aggregation and Association

Aggregation is used to express when one or more objects are a part of another object. This is seen in figure 2.3 on the following page. A computer contains several smaller

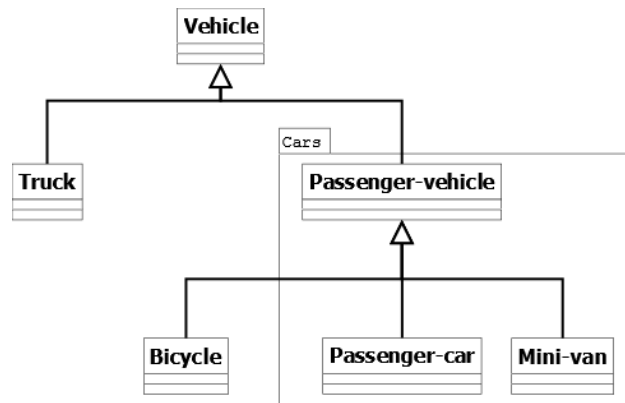


Figure 2.2.: Class diagram showing the concept of clusters.

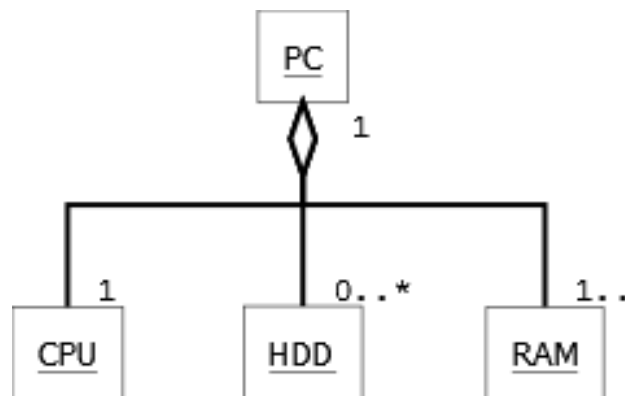


Figure 2.3.: Aggregations between objects.

parts, in this case only a harddisk drive, a RAM-module and a CPU is shown. The figure shows how each object is a part of the PC, and it also shows that in this example one computer is the minimum needed for any of the other objects to exist. This is shown by the number 1 at the computer object. The other objects are also given a number, for instance "0..*" meaning that the number of HDD's can be anything from zero to many. Notice that at least one RAM-module is required. This is to visualize that the computer *will* boot without an HDD, but without a RAM-module it will not allow for any system functionality.

The aggregation relation shows the connectivity between the objects. However, that kind of connection is not always enough to show relations between objects. For that, one also needs associations. It represents two or more objects that are related but where none are a part of the other(s). This relation does not imply any kind of hierarchy, as was the case with aggregation.

2.2.3. Behaviour

When analyzing the contents of the problem domain it is also relevant to know exactly what each object or class does. For this, one needs to analyze the behavior of the classes. This is primarily done using statecharts. Such a chart could look like the one in figure 2.4 on the next page. It shows, how a shop customer adds articles to his cart, or removes them, and then makes a checkout.

A statechart can consist of several different types of figures, each showing something

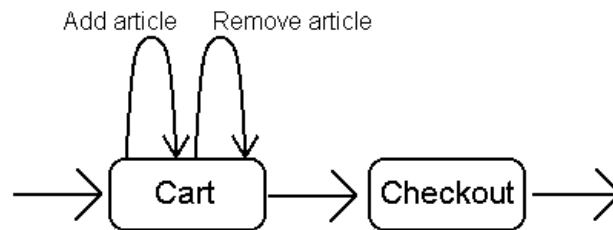


Figure 2.4.: A statechart showing the behavior of the class Cart.

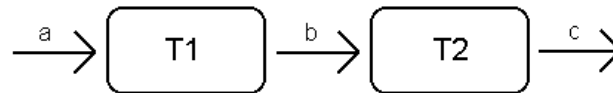


Figure 2.5.: A graphical illustration of a sequence.

different. They are:

- Sequence
- Selection
- Iteration

A sequence is a number of events that occur in succession. This effectively means that only one event can occur at a time. An example of such a figure can be seen in figure 2.5. A selection on the other hand acts a bit differently. This implies that a given event ends with one of several outcomes, effectively changing the state of the object just as in a sequence. An example can be seen in figure 2.6, where it is also seen that a selection can only happen once, unless another action returns the state of the object back to the state before the selection.

The last possibility is an iteration. This is very useful, as it shows what options the object has in its current state which does not end with a change of state. An example can be seen in figure 2.7 on the following page where "a" and "b" are both iterationable actions that the object can do. As name iteration also implies it can be performed any number of times.

It is worth mentioning that these events are the same events as found earlier in the system analysis. Normally, an event will initiate one of these actions by at least one object.

2.3. Application Domain

The application domain is the unit, which has the problems, the system is supposed to solve. The content of such a domain can be described by using actors, use cases and functions. This section will describe their purpose.

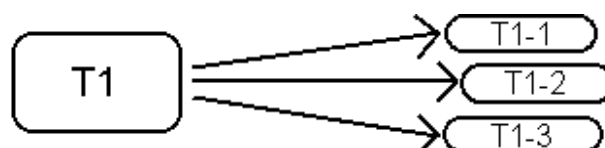


Figure 2.6.: A graphical illustration of a selection.

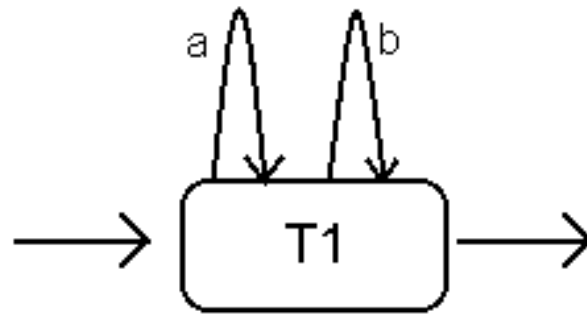


Figure 2.7.: A graphical illustration of an iteration.

2.3.1. Actors and Use Cases

After having analyzed the problem domain, one can begin to specify the basic demands for the system. In doing so, it is important to look at how the system would be used when finished. That is; what is going on in the problem domain, how is it happening and why is it happening? To find that out, one needs to look at what is called actors and use cases. Both are equally important for the system analysis. An actor, by definition, is:

Actor: An abstraction of users or other systems that interact with the target system[1, p. 119].

This means that an actor can be non-sentient, which means that other systems have to be carefully considered if they appear in the application domain. An analysis of any kind will usually result in a large amount of data, that could later be used in the design phase. To efficiently use such data, a certain level of abstraction must be maintained, in order to not waste time designing unnecessary features. To achieve such abstraction level, the concept of use cases is used:

Use case: A pattern for interaction between the system and actors in the application domain[1, p. 120].

A single use case describes a single use of the target system by an actor. Therefore, a complete set of use cases will describe every possible use of the system, within the limits of the application domain. To be able to produce such a set of use cases, the developers need to cooperate with the future users of the system. Users explain their needs and they usually have more insight and understanding of how the application domain works in real life. This means that they can give the developers some much needed ideas for the system.

2.3.1.1. Describing Actors and Use Cases

When describing an actor, it is a good idea to focus on the goal of the actor, which is the role of the actor in relation to the target system, as well as the characteristics of the actor. This could be done by listing the goal, characteristics and an example of an actor in a table.

- Goals - Describe the role, the actor should fill out in the system.
- Characteristics - Contains important points and general information about the actor.

- Example - A short description of a possible situation that describes the situation of an actor in the application domain.

A list of possible use cases can be produced by examining the application tasks of the domain. It is important to describe every use case in detail to make certain that use cases are distinct, otherwise redundancy will be an issue during design and development. A use case specification consists of:

- Use case - The use case itself, where individual actors and interactions are specified.
- Objects - The objects or actors that are involved in the use case.
- Functions - Functions that are to be developed in order to make the use case possible within the system.

2.3.2. Functions

The last aspect of the system analysis is to look at the functions performed in the problem domain during events. As with the other aspects, one needs to be able to find those functions to describe the problem domain accurately. This step is particularly important since the information obtained from these functions affect the system design phase greatly.

"A facility for making a model useful for actors." [1, p. 138]

Since the problem domain and application domain are two separate domains, there is a need for interaction between the two. This interaction is meant to work both ways - a system should be able to receive data from the problem domain, do the necessary computations in the application domain, and then return the data to the actor in the problem domain. The actor can then respond to the feedback in the problem domain and report the changes back to the application domain. This interaction is provided by functions.

A function can therefore be regarded as a operation, which takes some input data, performs a task and returns the output data. It should be noted that understanding the intent of the system, what the system should do, is not always as easy as looking at the function list. Advanced systems, a power plant management system for example, contains a large number of functions compared to a simple payroll system, thus making understanding of the system solely through the functions nearly impossible, as there are too many factors to consider.

Functions are divided into four different types. This provides developers with the ability to categorize and sort the functions and thereby understand their character easier, making the analysis more effective.

2.3.2.1. Function types

The best way to describe what a function type is, is:

"Each function type expresses a relation between the model² and the context of the system and has characteristics that help us when we define functions." [1, p. 138]

²The model of the problem domain, as mentioned in section 2.2 on page 11.

It should be noted that some of the functions of the system might not fit into a single one of those types. Instead some functions are best described as a mixture of types. The list of function types is here explained as examples from a train control system for better understanding. The four types are:

- **Update** - Activated by a problem domain event and results in a change in the state of the model.

Example: A function that receives data on locations of trains and shows them on a map.

- **Signal** - Activated by a change in the state of the model and results in a reaction in the context of the system.

Example: A function that activates an alarm when a train enters a wrong part of the tracks.

- **Read** - Activated when an actor in the application domain needs a certain piece of information to complete a task. Results in the system giving information to said actor.

Example: A function that retrieves information about a train and displays it on a control screen in the application domain.

- **Compute** - Activated when an actor in the application domain needs a certain piece of information to complete a task. In this case, the information cannot be retrieved from the system directly; rather it needs to be computed from a set of parameters given by the actor. The result is then sent to the application domain.

Example: A function that predicts the future situation on the railroad, such as the time of arrival of a train to a certain station based on its current velocity and location.

2.3.2.2. Function Analysis

The purpose of a function analysis is to determine the function requirements of the system, that is, what functions are needed to make the system fully operational. The result of this process is a complete function list that provides the developers with a wealth of information about the required functions. This list provides the developers with a tool that makes it easier to discover any mistakes in the function coverage of total system functionality.

Before the analysis can be done, the system definition and the use case analysis needs to be completed. If so, one can proceed with the function analysis and start finding functions.

Find functions The purpose of this is to find functions needed in the use cases and defining their type. The result of this should be a draft of the function list. These functions can be found by asking specific questions. For instance, what triggers a signal function or which state of the object should be updated by an update function.³

Specify complex functions Explaining exactly what a function does can be difficult, and will vary widely based on the observer's experience and understanding of the system. To make sure that functions are defined in full, one can use different tools as a help during the analysis [1, p. 145]:

³Predefined questions can be found in figures 7.2 - 7.5 in [1], but they are not necessarily the questions needed for one's system.

- A mathematical expression that shows the relation between input and output data.
- An algorithm written in pseudo code.
- Introduction of functional hierarchy directly in the function list - a complex function will be divided into several smaller functions.

It is worth noting that this definition should be as brief as possible. The goal is to identify the functions needed, not design their implementation.

2.4. Evaluation

When carrying out any of the tasks in the system analysis or design (which will follow in chapter 3 on the following page), the result is in no way a final result. Any task will most certainly be reviewed later, and the results will be changed. Failure to do this will result in developing functions which are not necessary, and possibly even miss critical functionality of the system.

System Design Theory

This chapter will go through the theory behind the design phase of a system development, starting with the criteria of the architectural structure of a system. This will be followed by a description of components, where a few will be described in details. Furthermore the relations between the components will be described at the end. The chapter is based on [1, chapters 9, 10, 12, 13, 14].

3.1. Criteria

Before defining the architectural structure of a system, it is necessary to agree on the demands to the system. These demands are called criteria, and in an optimal system, most of them would be top priority. This, however, is most often impossible, as the development phase is limited either economically or timewise, and some of the criteria are often contradicting. Therefore, the criteria must be prioritized, before being used to define the most optimal system architecture.

A list of the most used criteria can be found in [1, figure 9.1, p. 178]), but it can be edited as necessary to fit the project at hand:

- **Usable:** The system's adaptability to the organizational, work-related, and technical contexts.
- **Secure:** The precautions against unauthorized access to data and facilities.
- **Efficient:** The economical exploitation of the technical platform's facilities.
- **Correct:** The fulfillment of requirements.
- **Reliable:** The fulfillment of the required precision in function execution.
- **Maintainable:** The cost of locating and fixing system defects.
- **Testable:** The cost of ensuring that the deployed system performs as its intended function.
- **Flexible:** The cost of modifying the deployed system.
- **Comprehensible:** The effort needed to obtain a coherent understanding of the system.
- **Reusable:** The potential for using system parts in other related systems.
- **Portable:** The cost of moving the system to another technical platform.
- **Interoperable:** The cost of coupling the system to other systems.

Along with the criteria stated, some conditions also apply to a system. These conditions are either technical, organizational or human and are defined outside the analysis and design phase, but they still need to be considered when designing the architecture.

The technical conditions are demands to the system defined by the hardware to be used (if it is already decided which to use), other systems interacting with the system and patterns or components reused from earlier made systems.

The organizational conditions are demands set to the system, defined by the planned future development of the system, the work structure in the developer group and demands set by the contract.

The human conditions are demands set by the developer group. Their experience with the technical platform and with a system of the demanded type as well as their competences in system design, are factors that must be taken into consideration when defining the architecture of the system.

3.2. Components and Architecture

This section explains some architectural concepts, which can be used to ensure system comprehensibility.

3.2.1. Components

A component is a collection of several, connected structures, which responsibility is well-defined. Often, classes are grouped together in components, but components can also be grouped together in another components. This will ensure comprehensibility in the system.

3.2.2. Architecture

To ensure component comprehensibility, the structure between components needs to be clear, too. This is ensured by using an architectural pattern, which is a pre-defined template for the architecture. Two of these are the layered architecture pattern and the server-client architecture pattern, which will be described.

3.2.2.1. Layer Architecture

The layered architecture divides the components into layers, in which the components in each layer has a predefined set of layers, it is able to use operations from. These sets varies depending on the developers' decisions.

Using previously defined layers makes it easier to define what should be in each layer. Usually, an *interface layer*, a *function layer* and a *model layer*. The interface layer is responsible for reading inputs from the user or other systems, and return output to them. The function layer is responsible for complex functions, while the model layer contains the data and simpler operations. A more detailed explanation of the function and model component can be seen in sections 3.3 on the next page and 3.4 on page 23.

In the strictest setups, the layers are only able to operate on the layer just below it. This can, however, be fitted to the need of the system, but it compromises comprehensibility, as it makes some components depending on several layers, instead of only one.

The interface layer is able to operate on the function layer, which is able to operate on the model layer. This means that for any data request to be made from the function layer, a call is made through the function layer, if the architecture disallows skipping layers. An illustration hereof can be seen in figure 3.1 on the next page.

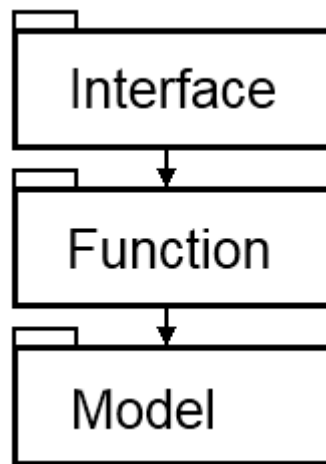


Figure 3.1.: The general component architecture structure.

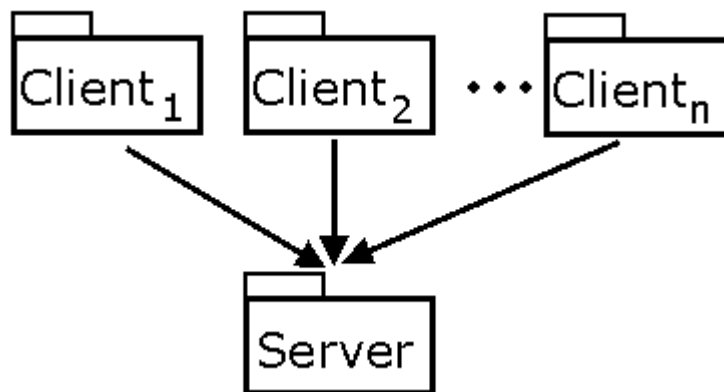


Figure 3.2.: The server-client architecture. One centralized server provides data to several, possibly geographically distributed, clients.

3.2.2.2. Server-Client Architecture

When several, geographically split users need to access the same data, the architecture needs to be considered thoroughly. The most usual way to enable data without being in the same location as the user, is to use the server-client architecture. The clients are then able to request data from the server, which returns it as needed. An illustration of the server-client architecture can be seen in figure 3.2.

3.3. Model Component

The model component in its final form is the actual class list, and their relations, for the program. This makes the design of the model component very important for successful implementation of the analyzed domain. The first draft of the model component might be revised when the function component is made. The architecture design of these two components will therefore be closely related.

The class diagram, which was the result of the analysis, along with the behavior patterns for the classes is the basis for the final model component class diagram. All classes and their relations should be analyzed to ensure that the found classes are sufficient.

The events found in the analysis should be represented in the model component. If

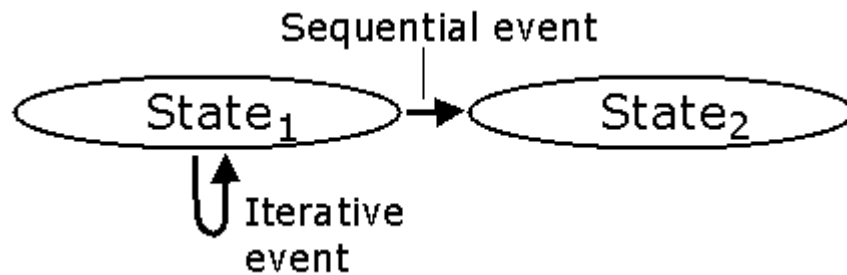


Figure 3.3.: The two different event types. The iterative events does not change the state of the object, whereas both selecting and sequential does (here, only sequential is illustrated, but the behavior of state-changing is the same).

the event occurs only in a sequence or a selection, it should be represented as an object attribute. If it on the other hand occurs as an iterative event, it should be represented as a new class, connected to the existing class. Iterative events can be found in the class behavior diagrams, as those events that originates and ends at the same state, whereas both sequential and selecting events changes the state. An illustration of the event types can be seen in figure 3.3. When the events has been represented in the model, it might be possible to make a better representation by generalizing or associating classes. It will be possible to make a generalization, if two existing classes contains the same properties, and their events are similar. Revising existing associations occurs when additional classes have been added to describe iterating events. If these events handles two classes, their association needs to be revised. The added class will have an association to the two other classes, and will also be connecting those two, instead of them using their original association.

3.4. Function Component

Functions, programming-wise, is an operation of some sort, that operates on the system. However, in this section, a function is defined as a complex operation, which operates on several classes, or an undefined number of grouped objects¹. Operations which operates on a define number of grouped objects are defined as simple operations, or just operations.

All functions and operations can be defined as a specific type. The types are the same as mentioned in section 2.3.2.1 on page 17. If the type is unclear, it is most likely possible to divide the function in fewer operations with a clearly defined type.

It is unnecessary to define all operations in the system. To explain, how a function is supposed to read an attribute from the model layer is unneeded. It might even be unnecessary to list the simplest operations, e.g. reading an attribute is so common that it is applied to almost all attributes. Such operations does not need to be listed.

The functions which then needs to be defined and explained are the complex operations, and the functions, which resides in the functions layer. They ought to be described with their types, purpose, necessary input, involved objects, the conditions, under which it is allowed to be executed, the effect of an execution, what triggers the execution and finally the object, in which the function should be placed.

¹If the classes are "aware" of each other's existance directly, they are considered grouped

3.5. Component Connection

To design a flexible and comprehensible system, the connection of the classes and components need to be thoroughly considered. The flexibility of the system is achieved by making components easily interchangeable, which is achieved by keeping the *coupling* low, and the *cohesion* high. The coupling of two given components is one of these four forms [1, p. 273]:

- **Outside coupling:** A class or component refers directly to the public properties of another class or component.
- **Inside coupling:** An operation refers directly to other, private properties in the same class.
- **Coupling from below:** A specialized class refers directly to private properties in the super class.
- **Sideways coupling:** A class refers directly to other, private properties in another class.

The couplings are ranked, with the lowest coupling at the top. The most admirable coupling therefore consists of public references to other classes and private references to own classes. This is easily achievable in most object-oriented programming languages, as it is not possible to refer to private properties of other classes.

The cohesion of a class or component describes their internal relations. For a class, high cohesion is achieved, if the attributes and object structures of the class is able to describe well-defined states of the object, along with operations that gives a functional whole, and uses each other. For a component, cohesion is achieved if the contained classes are conceptually related, their structural relations primarily are generalizations and aggregations, and operations can be executed within the component.

When connecting classes from different components, it can be done either by *aggregating classes*, *specializing a class* or *calling public operations in objects*.

If a class definition exists in one component, and a class which handles some events connected to the first class exists in another component, an aggregation can be used to connect the two classes. As the coupling is an outside coupling, the coupling is kept low by using aggregation, but the cohesion is difficult to achieve, when classes from different components are aggregated.

Specialization of classes is used when all properties in an existing class is needed in another class in another component. The coupling is kept low by specializing, as long as only public operations from the superclass is used. Protected superclass properties makes the coupling a bit higher, and using private operations and attributes, it yet again becomes a bit higher.

Operation calls are used for example when an interface component requests a read from the model layer. As only public operations are available, operation calls are considered low coupling.

Programming

In this chapter, the history of Object Oriented Programming (OOP) and the OOP language C# will be described. Furthermore, a description of unit testing and how it can be done with Microsoft® Visual Studio, will be found, along with a description of the ASP.NET technical platform.

4.1. Object Oriented Programming

The concept of OOP is to think in a rather different way, compared to some of the other, more traditional programming paradigms. As the name suggests, the concept revolves around objects, which are similar to the objects of OOAD.

An object is a unit which encapsulates state (data value), and behavior (method). An object oriented program consists of several connected objects that will communicate with each other and execute programmed operations as needed. An object oriented developer will therefore think about a program in terms of behaviors and responsibilities of objects. This allows the developers to bring everyday experience into the programming world. As with objects in analysis and design, objects in programming are based on classes. The written code is normally a class, and an instance of any class is then an object.

OOP reduces the interdependency between software components. In that way, software developers have the ability to reuse already existing components, which could have been written by other developers, without spending a lot of time reading and understanding the already existing code. This option is perfect for team development, because a developer can write a class which can be used by a second developer, who only needs to understand how to use the class. This enables developers to supplement each other, making a better program, instead of starting from scratch, every time a new project is started.

4.2. C#

C# is a programming language developed by Microsoft® and it is intended to be a simple, general purpose, object oriented programming language. C# makes use of the .NET framework, a large framework developed by Microsoft® as a combination of their technologies for developing applications. The framework is large, but the most interesting parts for this project are the .NET framework class library and ASP.NET. The .NET framework class library is an extensive library containing classes for many things, such as Windows Forms for graphical user interface (GUI), ADO.NET for database connections and the IO namespace for reading and writing files. ASP.NET is the engine responsible

for running programs written in .NET on the server side of applications for the web. A more detailed description of ASP.NET can be found in section 4.4 on page 29.

The .NET framework is designed to work with the Microsoft® Visual Studio Integrated Development Environment (IDE), and is currently in version 3.5.

4.2.1. Syntax

Some of the syntax of C# is very close to, or the same as, some other languages, such as Java. It is case sensitive and statement termination is done with a ";" as in many other programming languages. Since C# is object oriented every class must contain at least one constructor. If a constructor is not written by the developer, an empty constructor will be available. The constructor is a method which allocates memory for the object, as well as performing operations on time of initialization.

Some of the specific syntaxes of C# will be explained in this section¹.

4.2.1.1. Blocks

Blocks of code are defined using "{" and "}". A code block is a group of statements used for many things; from repeating functions using a while loop to defining methods and classes. Blocks are not considered statements in the same way as a normal statement, and should therefore not be ended with a ";".

4.2.1.2. Comments

There are usually two ways to comment in any programming language, single line and multiline. This is also true for C#. For single line comments simply put "//" in front of your comment and the compiler will ignore the rest of the line. A multiline comment can be created by starting the first line with /* and ending it with */. It is considered good practice to start all the lines in the comment with a *, it is not required by the compiler, but it makes the code look more consistent and multiline comments easier to identify. Since a comment is not a statement it does not need to end with a ";".

4.2.1.3. Variables

In C#, defining a variable is divided into two parts: declaring and initializing. A variable is declared by defining it with a type and a name, but no data. The value of the variable will be null until it has been initialized. The variable is initialized the first time it is assigned data. It is at this time that memory is allocated for the variable. Initializing a variable can be done at the same time as it is declared, but it is not a requirement. As mentioned above, a declaration includes the type of the variable. These types span from the normal **string**, **char** and **int** over different types of lists and arrays to one's own classes. Code example 4.1 on the facing page contains examples of several variable declarations. Both variables that are declared with specific types and some that are implicitly declared using the "var" keyword. This keyword can only be used when the variable is initialized at declaration.

4.2.2. Try and Catch

A way of handling exceptions in C# is to use the try-catch statement (as can be seen in [3]). This consists of a try-block and one or more catch clauses, depending on the

¹A full syntax overview can be found at <http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx>.

Code Example 4.1 Different variable declarations and initializations.

```

1 //Variables with specified types:
2
3 int a = 1; //Initialized the interger a as 1
4
5 int a;
6 a = 1; //Same as above, declared the integer a, then initialized as 1
7
8 int a = 2;
9 int b = a - 1; //b = 1
10
11 //Variables with implicit declaration:
12
13 var text = "Hello World" //Initialized the string text as Hello World
14
15 var text; //This will not compile

```

number of exceptions that should be caught, within a specific function. The try block contains the code that needs to be executed. If the execution of this code makes an exception, the code in the catch block will be executed, if not, the catch block will be omitted from execution.

Code Example 4.2 A try-catch statement.

```

1 int n = 6;
2 int m;
3
4 try
5 {
6     n = m*2;
7 }
8
9 catch (Exception EX)
10 {
11     Console.WriteLine(EX.Message);
12 }

```

Code example 4.2 shows a try-catch statement. The integer n is declared and assigned the value 6, and another integer m is declared, but no value is assigned. Inside the try-block, a calculation of $m*2$ is assigned to n . The example will result in an `NullReferenceException` being thrown, as m has no value assigned. This exception is caught by the catch block, which writes the error message of the exception in the debug console.

4.2.3. Type conversion

Type conversion is used when one wants to change a value from one type to another. This can be useful if one wants to take advantage of the properties of another type. In code example 4.3 it is shown how one can use the type conversion to discard the decimal value of a float.

Code Example 4.3 A type conversion.

```

1 float p = 2.4;
2
3 m = (int) p;
4
5 return m; //Returns 2

```

4.3. Unit Testing

When writing code, it is important to ensure that the code performs correctly. A good way to do this is by creating unit tests. During this project the unit testing framework provided in Visual Studio 2009® will be used. The idea of unit testing, is to take each method one by one, and test if it does what it is supposed to do. Consider the following situation:

A math class that contains, among other things, a method for multiplying two integers, has been written. The test for that one method, could look like the one in code example 4.4.

Code Example 4.4 A test of a multiply function.

```
1 myMultiplyTest()
2 {
3     myMath M = new myMath();
4     Assert.IsTrue(M.myMultiply(2,2) == 4);
5 }
```

The example is somewhat trivial, but it shows the basic principle of unit testing. The method is called, and given known parameters, and then tested against a pre-calculated correct result. It should be noted that while using tests help to ensure the functionality of the methods, it only tests for the conditions, it has been given. In the example, the method of multiplying two numbers with each other was tested. Since the result is easily calculated without the use of the method, it is possible to test the method. However, this does not prove that the method will return the correct result for any given parameters, only the ones tested. There is no way to test a method under all conditions, without mathematically proving it, but this is normally not a problem, only something that needs to be kept in mind when debugging.

4.3.1. The Assert Object

In code example 4.4, the Assert object is used to make the actual test. This object is a part of the Unit Testing framework and contains the functions to return the result of the test. It provides a number of methods to make the testing faster for the user, and handles the error reporting. The following is a list of the test methods of the Assert object taken from the documentation of the Unit Testing framework.²

- **AreEqual** - Verifies that specified values are equal.
- **AreNotEqual** - Verifies that specified values are not equal.
- **AreNotSame** - Verifies that specified object variables refer to different objects.
- **AreSame** - Verifies that specified object variables refer to the same object.
- **Fail** - Fails an assertion without checking any conditions.
- **Inconclusive** - Indicates that an assertion cannot be proven true or false. Also used to indicate an assertion that has not yet been implemented.
- **IsFalse** - Verifies that a specified condition is false.
- **IsInstanceOfType** - Verifies that a specified object is an instance of a specified type.

²at <http://tiny.cc/unittestassert>

- **IsNotInstanceOfType** - Verifies that a specified object is not an instance of a specified type.
- **IsNotNull** - Verifies that a specified object is not a null reference.
- **IsNull** - Verifies that a specified object is null.
- **IsTrue** - Verifies that a specified condition is true.

4.3.2. Writing Tests

If tests are to be written for existing code, Microsoft® Visual Studio provides a wizard that creates much of the code for the tests, along with comments telling what to do, in order to implement the tests correctly. This greatly reduces the time it takes to write tests, but only when the code already exists. An example can be seen in code examples 4.5, 4.6 on page 32 and 4.7 on page 33³. The first two examples shows the written code, and the auto-generated code for the test. Following the instructions in the comments of the code, a developer would end up with a test like the one in the last code example, 4.7 on page 33.

Code Example 4.5 The myMath class

```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace Math
7 {
8     class MyMath
9     {
10
11         public int myMultiply(int a, int b)
12         {
13             return a * b;
14         }
15
16         public int myAdd(int a, int b)
17         {
18             return a + b;
19         }
20     }
21 }

```

4.4. ASP.NET

ASP.NET is a web application framework which was produced by Microsoft® to allow developers to work with webbased applications more efficiently. In order to understand ASP.NET it is important to first understand HTML and the idea of server side applications.

4.4.1. HTML

When surfing the internet with a browser, the most used file format is HyperText Markup Language (HTML), which is, as the name suggests, a format providing formatting of information such as text and images. A browser interprets the HTML, and displays the

³A test for the constructor was also created by the system, but it has been omitted since it does nothing.

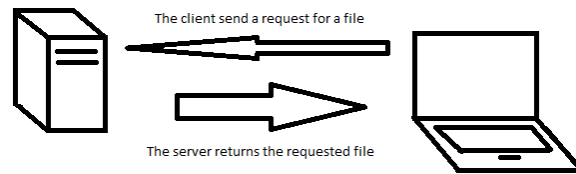


Figure 4.1.: Requesting an HTML document from a server.

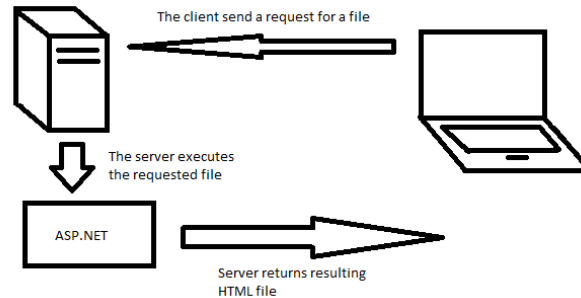


Figure 4.2.: Running a server side application.

information accordingly. HTML makes use of tags, such as the body tag, "<body>". All these tags must be closed with a matching close tag, which for the body looks like "</body>". The number of tags supported by HTML depends on the version (currently 4.01), and is rather extensive⁴.

4.4.2. Server Side Applications

The difference between simply viewing an HTML page (Figure 4.1) and using a server side technology such as ASP.NET (Figure 4.2) is that the server side application provides a better platform for interaction with the user, as content can be created dynamically from data that has been inputted as well as data stored in a database.

When creating a server side application all inputs, such as textboxes and buttons, must be placed inside a form tag. When pressing a button within the form, a new request is sent to the url specified in the form tag. Form data is sent back with the request using either the POST or GET method. They both have strong points and weak points:

POST sends the data back as part of the request. This means that sensitive data is not being transmitted openly for everyone to see, but there is no way to link to the resulting page. This can be a problem if the information is a search result the user wants to share with someone.

GET sends the data as part of the url. This should only be used if the data is not sensitive, but can be very helpful when the user links to the page. Since all the data is transmitted in the url, there is less data to transmit, so the GET method is slightly faster.

ASP.NET also requires the server to be a Windows server with Internet Information System (IIS) installed. Code for ASP.NET does not need to be precompiled like a normal application though, as this is handled by ASP.NET, whenever a request is sent to the server from a client. ASP.NET returns pages that are not HTML, but rather XHTML, a

⁴The official specification for the HTML 4.01 standard can be found at [4].

new kind of markup language, very similar to HTML. It uses many of the same tags, but it does not provide the same forgiveness as HTML. Normally, a browser reads the HTML, and displays the data according to the format specified in the file. If the HTML contains an error, the browser will try to correct it before displaying it. However, different browsers correct these errors in different ways. There is no such error correction in XHTML.

An ASP.NET page uses the .aspx file extension and is built up by three files:

- ***.aspx** - The file containing the XHTML along with the specific ASP.NET controls.
- ***.aspx.cs** - The file containing the code for the page, including the required `Page_Load` function which is called whenever the concerned page is loaded⁵.
- ***.aspx.designer.cs** - The file containing the code, connecting the .aspx file with the .aspx.cs file. The file is autogenerated by Microsoft[®] Visual Studio and should not be modified.

The *.aspx file uses XHTML syntax along with some ASP.NET specific tags, such as "<ASP:Label>". A Hello World example of an aspx page can be found in code example 4.8 on page 34. The other files for the page can be found in code examples 4.9 on page 34 and 4.10 on page 35. The aspx page defines a form containing a label, which is a text container, with no text defined. However, the `Page_Load` function sets the text of the label to "Hello World".

Since XHTML does not have a definition for the aspx label tag, ASP.NET converts it into a span tag, "", that has similar properties. In addition to the span tag, most HTML tags are usable in ASP.NET through the `WebControls` namespace.⁶

⁵The `Page_Load` function is also called when a page is reloaded after a button has been pressed.

⁶A list of all available controls can be seen at <http://tiny.cc/cswc>.

Code Example 4.6 The TestMyMath class before implementation

```
1 using Math;
2 using Microsoft.VisualStudio.TestTools.UnitTesting;
3 namespace TestMath
4 {
5
6
7     /// <summary>
8     ///This is a test class for MyMathTest and is intended
9     ///to contain all MyMathTest Unit Tests
10    ///</summary>
11    [TestClass()]
12    public class MyMathTest
13    {
14
15
16        private TestContext testContextInstance;
17
18        /// <summary>
19        ///Gets or sets the test context which provides
20        ///information about and functionality for the current test run.
21        ///</summary>
22        public TestContext TestContext
23        {
24            get
25            {
26                return testContextInstance;
27            }
28            set
29            {
30                testContextInstance = value;
31            }
32        }
33
34
35        /// <summary>
36        ///A test for myMultiply
37        ///</summary>
38        [TestMethod()]
39        public void myMultiplyTest()
40        {
41            MyMath target = new MyMath(); // TODO: Initialize to an appropriate value
42            int a = 0; // TODO: Initialize to an appropriate value
43            int b = 0; // TODO: Initialize to an appropriate value
44            int expected = 0; // TODO: Initialize to an appropriate value
45            int actual;
46            actual = target.myMultiply(a, b);
47            Assert.AreEqual(expected, actual);
48            Assert.Inconclusive("Verify the correctness of this test method.");
49        }
50
51        /// <summary>
52        ///A test for myAdd
53        ///</summary>
54        [TestMethod()]
55        public void myAddTest()
56        {
57            MyMath target = new MyMath(); // TODO: Initialize to an appropriate value
58            int a = 0; // TODO: Initialize to an appropriate value
59            int b = 0; // TODO: Initialize to an appropriate value
60            int expected = 0; // TODO: Initialize to an appropriate value
61            int actual;
62            actual = target.myAdd(a, b);
63            Assert.AreEqual(expected, actual);
64            Assert.Inconclusive("Verify the correctness of this test method.");
65        }
66    }
67 }
```

Code Example 4.7 The TestMyMath class after implementation

```

1 using Math;
2 using Microsoft.VisualStudio.TestTools.UnitTesting;
3 namespace TestMath
4 {
5
6
7     /// <summary>
8     ///This is a test class for MyMathTest and is intended
9     ///to contain all MyMathTest Unit Tests
10    ///</summary>
11    [TestClass()]
12    public class MyMathTest
13    {
14
15
16        private TestContext testContextInstance;
17
18        /// <summary>
19        ///Gets or sets the test context which provides
20        ///information about and functionality for the current test run.
21        ///</summary>
22        public TestContext TestContext
23        {
24            get
25            {
26                return testContextInstance;
27            }
28            set
29            {
30                testContextInstance = value;
31            }
32        }
33
34
35        /// <summary>
36        ///A test for myMultiply
37        ///</summary>
38        [TestMethod()]
39        public void myMultiplyTest()
40        {
41            MyMath target = new MyMath();
42            int a = 2;
43            int b = 2;
44            int expected = 4;
45            int actual;
46            actual = target.myMultiply(a, b);
47            Assert.AreEqual(expected, actual);
48        }
49
50        /// <summary>
51        ///A test for myAdd
52        ///</summary>
53        [TestMethod()]
54        public void myAddTest()
55        {
56            MyMath target = new MyMath();
57            int a = 2;
58            int b = 2;
59            int expected = 4;
60            int actual;
61            actual = target.myAdd(a, b);
62            Assert.AreEqual(expected, actual);
63        }
64    }
65 }

```

Code Example 4.8 default.aspx

```
1 <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="
  TestProject_Default" %>
2
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
  xhtml1-transitional.dtd">
4
5 <html xmlns="http://www.w3.org/1999/xhtml" >
6 <head runat="server">
7   <title>Hello World</title>
8 </head>
9 <body>
10   <form id="form1" runat="server">
11     <div>
12       <asp:Label runat="server" ID="HelloLabel"></asp:Label>
13     </div>
14   </form>
15 </body>
16 </html>
```

Code Example 4.9 default.aspx.cs

```
1 using System;
2 using System.Collections;
3 using System.Configuration;
4 using System.Data;
5 using System.Linq;
6 using System.Web;
7 using System.Web.Security;
8 using System.Web.UI;
9 using System.Web.UI.HtmlControls;
10 using System.Web.UI.WebControls;
11 using System.Web.UI.WebControls.WebParts;
12 using System.Xml.Linq;
13
14 namespace WebApplication1
15 {
16     public partial class _Default : System.Web.UI.Page
17     {
18         protected void Page_Load(object sender, EventArgs e)
19         {
20             HelloLabel.Text = "Hello World";
21         }
22     }
23 }
```

Code Example 4.10 default.aspx.designer.cs

```
1 //-----
2 // <auto-generated>
3 //     This code was generated by a tool.
4 //     Runtime Version:2.0.50727.4927
5 //
6 //     Changes to this file may cause incorrect behavior and will be lost if
7 //     the code is regenerated.
8 // </auto-generated>
9 //-----
10
11 namespace WebApplication1 {
12
13
14     public partial class _Default {
15
16         /// <summary>
17         /// form1 control.
18         /// </summary>
19         /// <remarks>
20         /// Auto-generated field.
21         /// To modify move field declaration from designer file to code-behind file.
22         /// </remarks>
23         protected global::System.Web.UI.HtmlControls.HtmlForm form1;
24
25         /// <summary>
26         /// HelloLabel control.
27         /// </summary>
28         /// <remarks>
29         /// Auto-generated field.
30         /// To modify move field declaration from designer file to code-behind file.
31         /// </remarks>
32         protected global::System.Web.UI.WebControls.Label HelloLabel;
33     }
34 }
```

Databases

This chapter will describe the basics of relational databases. This will be followed by a section on Structured Query Language (SQL) which was used in this project. The source used in this chapter is the textbook [5] used in the Software Architecture and Databases course this semester, unless other references have been made.

5.1. History

Database systems arose as issues with normal filestoring became apparent. Traditionally, programs and scripts were created to change values stored in files or to find information needed for everyday tasks. This created a problem, as when new needs appeared, it was necessary to rewrite existing scripts or write a new script from scratch. This was not ideal, as workhours and money were poured into further development of already "completed" systems.

Other problems with filestoring were also becoming more and more visible. It was highly likely that, as more and more users were added to a system, file inconsistency would become a problem. Meaning, the way filesystem storing worked in general, had a natural, generic flaw. Users requesting file data at the same time, and afterwards changing values in the data, would almost always result in problems, as those values would be changed according to their common starting value. As such a subtraction from three different sources of 30 each, would not amount to more than a total subtraction of 30, as each change would simply set the value to "Startingvalue - 30" instead of "Startingvalue - 30 - 30 - 30".

Databases, in general, work differently, and as such fixes a lot of these problems. For instance; it is only necessary to write a new *command* to get the data needed, instead of a complete program or script. This means less time wasted on development. Furthermore, actions in a database are atomic, that is, they happen completely or not at all. Therefore the attribute values will not be incorrect or corrupted, if the system crashes during the execution of a command - it will either complete the operation, or no change will occur at all. Also, changes to the values are done instantly according to the value at execution time, which means that the subtraction problem mentioned before stops happening.

5.2. Relational Database Theory

A relational database is a collection of tables, called relations, with unique names. Each table consists of rows and columns, like in a spreadsheet. The column header in a table, is called an attribute. The attribute has a set of permitted values, called the **domain** of the attribute, which will be expanded upon in section 5.2.1 on the facing page.

Filling in rows of a database relation creates what is called tuples, which means:

$$t \in r$$

where t = tuple and r = relation. It is worth noticing that tuples within a relation is not necessarily sorted. Because of this, the whole relation will be searched through, when looking for a specific tuple.

5.2.1. Domain and Domain Value

A domain is a set of permitted values for an attribute, denoted as D . A table, *customer* - (c), with four attributes: lastname - (l), firstname - (f), address - (a) and phonenumber - (p), is therefore a subset of the domain of the four individual attributes:

$$D_c = D_l \times D_f \times D_a \times D_p$$

Using the mentioned example, a tuple could look like this:

Hansen, Hans, Aalborgvej 92, 98115540.

The domains of the attributes are not necessarily the same. In this example the last- and firstname attributes would have variable character domain, which is mostly used for strings, while the phonenumber would have an integer domain. Trying to enter values not allowed in the attribute domain will result in an error. It is, therefore, important to choose an appropriate attribute domain.

In the case of the phone number, you can further use the domain to ensure that only danish phone numbers are inserted in the database, by requiring a number between 10000000 and 99999999, which will represent a Danish phone number.

5.2.2. Keys

After the database has been created, one needs to be able to distinguish the tuples within a given relation. This happens through the use of *keys*. To identify a unique tuple, the *primary key* property is used. A primary key is an attribute, which must contain different values for each tuple. Usually, an auto-incrementing integer is used as value for this key, as this ensures uniqueness.

A *foreign key* in a relation is a primary key of another relation. This means that a relation schema, r_1 , could among its attributes contain the primary key of another relation, r_2 . That attribute is then a foreign key in r_1 , referring to r_2 . This is useful as one can then make relationships between relations. A visual example can be seen in figure 5.1 on the next page. The example shows two relations - one containing team names and one containing team members. They both have primary keys in the form of the auto-incrementing id attribute, but the TeamMember relation also have the TeamId attribute, which is a foreign key referring to the id attribute of the Team relation. This ensures that no team member can be inserted into the database without being assigned a team too.

5.2.3. Data-Manipulation Language and Data-Definition Language

Using a modern database requires one to interact with the database with the help of query languages, which can be divided into Data Manipulation Language (DML) and Data Definition Language (DDL). Both have very straightforward uses according to their names. DDL allows one to define what data should be in the database and where the

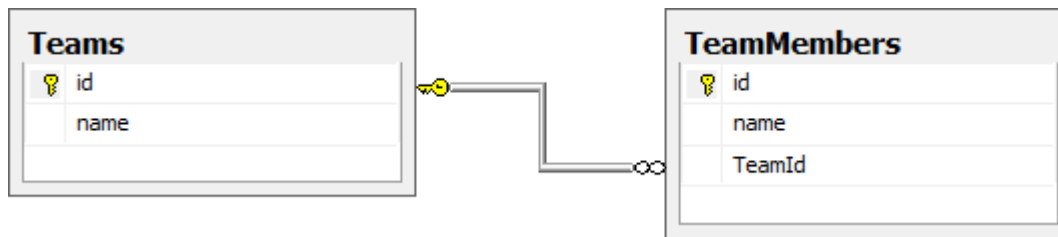


Figure 5.1.: Primary and foreign keys in a small database.

data should be stored. DML allows one to manipulate and use that data in many different ways. This spans from simple queries that select data from specific locations in the database, to more complicated queries that select data across several different tables, combines it, sorts it and creates whole new tables with different properties. This solves the problems with conventional file storage mentioned in section 5.1 on page 36 as new arrangements of data is easy to obtain, and it makes it easier to retrieve any kind of data from the database.

5.2.4. Designing Relational Databases

Database design, like many other design jobs, starts with an analysis of the requirements for the database. This focuses mainly on the data needs of the users, and how they expect to interact with that data. After a careful analysis of the users' needs, one chooses a data model that suits those needs. Analyzing and designing the database can be done with many tools. This report will focus on the Entity-Relationship Model for designing it. Analyzing the database should be done following the theory in chapter 2 on page 10.

5.2.4.1. Entity-Relationship Model

An Entity-Relationship model¹ (E-R model) is a effective tool for designing databases. It can give a complete overview of the database and all the attributes in it, which includes how they relate to one another.

The figure 5.2 on the facing page shows a small part of an E-R model for an entire banking database. One can clearly distinguish the tables (the squares), the attributes (the ellipses) and the relations between the tables (the diamonds). This way, it is easy to see what kind of data one would have in the database, but also what kind of relations one would need to construct to make it work. For instance, it would be a bad idea not to have a clear relation between the customer and the account database, but it could be hard to spot such an error without the visual help of such a model.

This report will not go into further details about the E-R model, as this project does not require the use of the more advanced features of this tool.

5.3. Structured Query Language

In this project, a combined DDL and DML language called Structured Query Language (SQL) has been used. Multiple versions of this language exists, this project has used the Microsoft® SQL language, MSSQL. This section will cover the basics of SQL, and then, more advanced features, which has been used in the project, will follow.

¹An entity is in this case what is also known as a tuple.

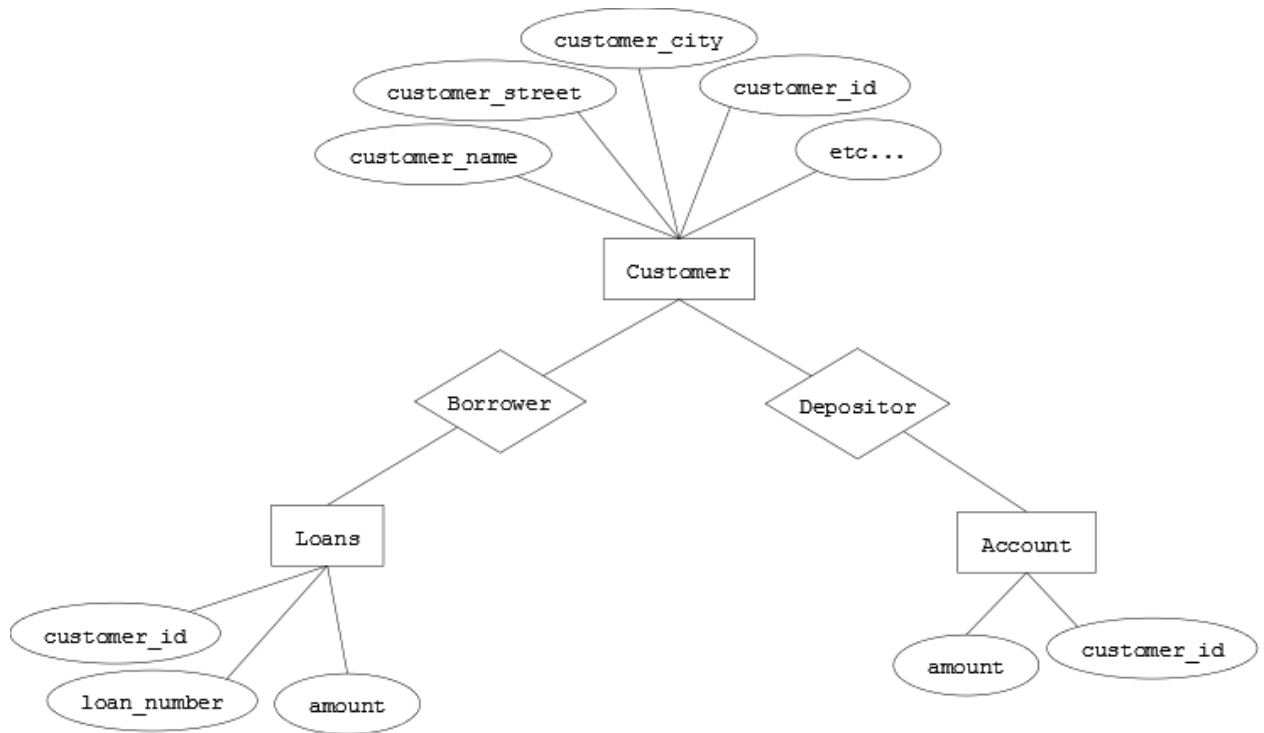


Figure 5.2.: An E-R model for a small part of a banking system.

SQL is a database query language, which, among other things, is used to manipulate data in a database. It is widely used, and supports most major databases. It allows for integrity constraints, which is very useful in distributed systems, as you cannot expect the user to always properly type in values. These features are readily accessible in the SQL DDL, which furthermore allows for defining exactly where on the physical hard drive the data is to be stored. Along with the SQL DML it gives complete control over data in the user's database, from creating tables to the deletion of said tables. The database administrator will be able to control exactly what data is accessible to whom, and he can do exactly what he needs to with that data.

SQL can also be embedded in several major programming languages, such as C++, Java or, as used in this project, C# / ASP.NET.

5.3.1. SQL Servers

There are many different database servers available. As mentioned above this project uses the MSSQL server, more precisely the 2005 version. An "express"² version of this server is provided along with Microsoft® Visual Studio, but since the MSDNAA³ provided access to the more powerful professional version, this was used in the project. Most SQL servers provide an administrator interface making it easier to perform many of the tasks associated with setting up and maintaining a database. They use SQL commands, and as such have no abilities above what a commandline access to the server provides, but it does make it far easier for inexperienced users to administrate it. Commands such as CREATE TABLE, ALTER TABLE and DROP TABLE are used by the administrator interface without the user having to write the commands himself. Since this section will

²Many of the development tools from Microsoft® come in express versions, that are not time limited like many trials, but rather limited functionality.

³Microsoft® Developer Network Academic Alliance

only explain the commands used by the developed application, they will not be explained here. However, interested readers can find this information in [5, chapter 4].

5.3.2. SQL Commands

This section will be a short description of the SQL-commands used in the project. Every command will be shown with a short example and an explanation of the various parameters and what they do.

5.3.2.1. Insert Into

To insert data into a table, the INSERT INTO [5, p.78] command is used, and can be seen in Code Example 5.1.

Code Example 5.1 The structure of the SQL insert into statement.

```
1 INSERT INTO r
2 VALUES (V1, V2, V3)
```

The r represents the name of the table the data should be inserted to. The values V_1 through V_n represent the values to be inserted in the 1st through n^{th} column. According to the database constraints, one or more of these values may or may not be null values or simply left empty. Using more constraints makes the database less prone to failure from entering wrong values. An alternative way to use the insert command specifies exactly which columns to put each value in. The structure can be seen in code example 5.2.

Code Example 5.2 A more detailed example of the structure of the SQL Insert Into statement.

```
1 INSERT INTO r
2 (A3, A1, A2)
3 VALUES (V3, V1, V2)
```

This gives the user a lot more control over exactly what is entered in the database, and also makes it easy to leave columns empty if needed, and even enter data in a different order.

5.3.2.2. Delete

After entering data into the database, one might need to delete some of it again. The SQL-command for this is DELETE [5, p.78-79]. The structure of this SQL statement can be seen in code example 5.3.

Code Example 5.3 The structure of the SQL delete statement.

```
1 DELETE FROM r
2 WHERE P
```

Again, r is the table from which to be deleted. The WHERE⁴ keyword is used to set some predicate P of the values to be deleted. For instance, it could be (**amount** < 200)

⁴The WHERE keyword can be used by almost all SQL-commands to limit the tuples affected by the command.

which would delete all tuples with the amount attribute less than 200. If the WHERE keyword and the predicate is removed, the command is performed on all tuples of r . It is important to realize that this does not delete the table, but only deletes the tuples from the table.

5.3.2.3. Select

The SELECT statement requests data from the table, and returns a new table with the requested data. The SELECT statement can be seen in code example 5.4. The attributes that should be selected from the table, are named attributes A_1 to A_n . If multiple tables are used in the FROM statement, these attributes must be specified with TableName.AttributeName. The SELECT statement can also be used with a "*" symbol. In this case, all attributes from the specified table(s) are returned. Again, P , is a predicate used the limit the tuples selected.

Code Example 5.4 Select statement.

```
1 SELECT A1, A2, ..., An
2 FROM r
3 WHERE P
```

5.3.2.4. Update

It is also possible to update a tuple in the database. It is done by using the UPDATE statement. The UPDATE statement can be seen in code example 5.5.

Code Example 5.5 Update statement.

```
1 UPDATE r SET
2 A1 = V1,
3 A2 = V2,
4 ...,
5 An = Vn
6 WHERE P
```

Again, r , represents the table, in which the attributes should be updated. The attribute A_1 is updated to V_1 , A_2 to V_2 and so on. This is done to those tuples, where the predicate P , if any, applies.

5.3.2.5. Advanced SQL functions

The above mentioned functions are the basic SQL statements, which can be used to complete most database tasks. However, one might need more advanced functions for performing specific tasks, such as grouping and sorting results. Not all of these advanced functions will be mentioned, only those used in the program.

Order By When selecting some tuples, it might be necessary to sort them in a specific order. That can be achieved by applying the ORDER BY statement after a select statement, as shown in code example 5.6 on the following page. The attribute A_1 is used as the ordering attribute, and it is in the example ordered in ascending order. It could also be ordered in descending order, in which case the keyword would be *DESC* instead of *ASC*.

Code Example 5.6 Select statement with an order by statement.

```
1 SELECT A1, A2, ... An
2 FROM r
3 ORDER BY A1 ASC
```

Code Example 5.7 Group by statement.

```
1 SELECT A1,
2 AVG(A2) AS V
3 FROM r
4 WHERE P
5 GROUP BY A1
```

Group By The GROUP BY statement[5, p.90] is used to group tuples with a common attribute value together. This can, for instance, be used together with the AVG statement⁵ that computes the average in a column, a rating, for instance. But when used together with GROUP BY, it could be used to request an average rating for each article in the relation, if they were grouped by articleId. An example of the GROUP BY statement can be seen in code example 5.7, where an attribute A₁ is selected along with an average of the attribute A₂. Instead of returning each tuple for itself, an average of A₂ from all tuples with the same value in A₁, will be returned.

Code Example 5.8 SQL join statement.

```
1 SELECT r.A1,
2 s.A2
3 FROM r
4 JOIN s
5 ON r.A1 = s.A2
```

Join It is also possible to join the data from two, or more, tables by using a JOIN statement[5, p.110-112]. An example of a JOIN statement can be seen in code example 5.8, which will return attribute A₁ from table *r* and A₂ from table *s*, where A₂ has the same value as A₁. There exists several JOIN statements:

- JOIN and INNER JOIN: Returns rows, where there are matches in both tables.
- LEFT JOIN: Returns every row from the left table (in code example 5.8: *r*), and any matching rows from the right table.
- RIGHT JOIN: Returns every row from the right table (in code example 5.8: *s*), and any matching rows from the left table.
- FULL JOIN: Returns every matching row from both tables.

Scope Identity When adding a new tuple or record to a database table, it might be necessary to know, what unique id, the tuple has been assigned, as it is set by the database itself. This can be requested by the SCOPE_IDENTITY() SQL function. The SCOPE_IDENTITY function is called after an INSERT statement, and will return the most recently added id. It can be seen in code example 5.9 on the next page, a new tuple is added to the database, and afterwards the id is requested by the name *T*.

⁵Will be explained later.

Code Example 5.9 The SQL scope identity.

```

1 INSERT INTO r
2 (A1)
3 VALUES (V1)
4 SELECT SCOPE_IDENTITY()
5 AS T

```

Math Statements In SQL, there exists several math functions. For example, it is possible to request the total sum (*SUM*), the average sum (*AVG*), largest (*MAX*) or smallest (*MIN*) value of the selected attribute, and the amount of tuples (*COUNT*). By exchanging the *AVG* keyword in the code example 5.10, with the keywords noted in upper case in parentheses, the other functions will be available.

Code Example 5.10 The built-in SQL average functions.

```

1 SELECT
2 AVG(A1)
3 FROM r

```

5.3.2.6. Full Text Search

MSSQL Server provides a service called Full Text Search, to make searches much faster. It does this by creating an index of the database. In order to use the feature, the administrator must first set up a catalogue, and then index the database. The index must be built manually everytime it needs to be updated. In large databases this is a very large operation, and should therefore not be performed everytime the database is changed. In smaller systems this is however not a problem.

An example of how to setup and build the index can be seen in code example 5.11.

Code Example 5.11 The commands needed for Full Text Search to work.[6]

```

1 --1 Create the catalog (unless you already have)
2
3 EXEC sp_fulltext_catalog 'FTCatalog','create'
4
5 --2 Add a full text index to a table
6
7 EXEC sp_fulltext_table 'Departments', 'create', 'FTCatalog', 'pk_departments'
8 EXEC sp_fulltext_table 'Employees', 'create', 'FTCatalog', 'pk_employees'
9
10 --3 Add a column to the full text index
11
12 EXEC sp_fulltext_column 'Departments', 'ProductName', 'add'
13 EXEC sp_fulltext_column 'Employees', 'Description', 'add'
14
15 --4 Activate the index
16
17 EXEC sp_fulltext_table 'Departments','activate'
18 EXEC sp_fulltext_table 'Employees','activate'
19
20 --5 Start full population
21
22 EXEC sp_fulltext_catalog 'FTCatalog', 'start_full'

```

The EXEC keyword is used for executing commands on the server. The last command in the example is used whenever the index needs to be rebuild. As mentioned before this

can be done anytime the administrator believes this to be appropriate, either at the same time each day, or whenever a change is made to the database as it can be added to the end of any other SQL statement.

5.3.3. Connecting to the database

The SQL statements can only be used, when a connection to the database has been created. The way to do this in C# can be seen in code example 5.12, which will be described here in details. In the first line, a new instance of the `SqlConnection` class⁶ (a part of the .NET class library) is created. As a parameter, it is given a connection string, containing the information needed to connect to the server. In this example, four parameters are used in the string; the servername, the userid, the password and the database.⁷ The second line is used to call the *open* function from the `SqlConnection` class. This opens a connection using the connection string given in line one. In line four, the SQL query is set to a `SELECT` statement that will return everything in the table *r* in the database stated in the connection string. In line five, a new `SqlCommand` is created. As parameters, it uses the connection opened in line two, and the SQL query made in line four.

In line six, the `ExecuteReader` function is called, which executes the query on the server. This function returns a `SqlResult` object which is assigned as *R*. The `Read` function will read the next line in the result, by moving the pointer to the line. It can then be used in a `WHILE` statement in lines eight to eleven. In this example it adds the ID of the current tuple to a list of ids. When the pointer is moved **beyond** the last row, the `read` function returns false, and the loop is stopped. In line twelve, the database connection is closed again, as there is no need for it any longer. It is very important to close the connection, because open connections will slow the server and the application greatly. The entire code should be placed in a try-catch statement, as any problems with either the query or the connection will result in an exception.

Code Example 5.12 SQL connection statement.

```
1 var SC = new SqlConnection("server=SERVERNAME;User ID=111;password=222;database=333");
2 SC.Open();
3
4 var query = "SELECT * FROM r";
5 var SCommand = new SqlCommand(query, SC);
6 var R = SCommand.ExecuteReader();
7
8 while (R.Read())
9 {
10     this.ListOfIds.add((int)R["Id"]);
11 }
12 SC.Close();
```

⁶The chosen `SqlConnection` class contains a connection pool handler. This ensures that connections can be reused, as long as the connection string is the same.

⁷A complete list of the possible keywords can be found at <http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.connectionstring.aspx>.

Agile development and Scrum

This chapter will describe the reasoning for using agile software development, and describe an agile development method, Scrum. The key features of the Scrum method will also be described. The chapter is based on the first 6 chapters of Agile Software Development with Scrum [7].

6.1. Agile development in software engineering

As a relatively new field of engineering, software development is still in a phase where everyone is trying to find the best possible way to deliver functional products on time. When software development first got popular, companies approached it much like they did any other product development; some straightforward work assignments designed to create reliable products. This had after all worked out well in the past in all other major product lines, so it was natural to assume the same would be the case with software development.

However, things did not turn out quite the way everyone hoped for. It became more and more visible that software production could not be tossed onto a production line like other products, as there were, and still are, too many factors working against this approach.

The biggest problem is the fact that a software development process is not a perfectly defined one. You cannot have your developers sit down and do a perfectly defined job each day, and expect to end up with a product of the same quality everytime. That approach works fine on a production line where the exact same product is created everytime, but that is not the goal of software development.

The difference is that software development is a complex and everchanging entity, and as such needs an empirical approach[7, p. 24]. Since development problems are not readily defined, one has to find and define these problems during development. The development process, therefore, needs to accommodate for this. Agile development tries to do just that by not planning everything in advance, as one would normally do. Instead the focus is on being able to change the plan along with the development process, according to what suits one's needs at the moment.

A common development method which also incorporates this idea, is the iterative development approach, one which is often seen in agile development. It states that the system should be developed in chunks. Chunks, that are optimized and put together as the development progresses, ultimately making the system include more features as development progresses. The idea is that a workable system should be ready very quickly after development starts, with only basic functionality, and then be expanded upon. This also makes it a lot easier to work together with potential buyers, as they

can see the system shaping up, as opposed to it taking more than half the development phase before a working prototype is ready.

In general, agile development tries to make software engineering more easily understandable to outsiders, and more forgiving and manageable to developers. Both of which help to make better systems. This can be done in several different ways, though most of them resembles each other quite a bit. This project will be focusing on only one of them, called Scrum.

6.2. Scrum

Scrum is a, fairly, new approach to the development process of software, compared to other development methods. It acknowledges the fact that one need to develop software empirically, and tries to incorporate this into the common organization structure of most modern companies. One of the desirable points of Scrum however, is the fact that it sets its participants "free". By removing a lot of the management's influence on the daily workroutines, it lets developers work according to their own assumptions of what is best. It does so by applying several "practices", which primarily focuses on the interactions between developers as the project progresses. The most common practices will be listed in this section, but for a full review of suggested practices, please refer to [7].

This walkthrough will focus on Scrum as a process in a company, and not in a study group. In section 9.2 on page 77, it will be explained how Scrum was modified, so it could be used in this project.

6.2.1. Scrum Master

The Scrum Master is the "leader" of the team and the one who connects the development team to the upper management. The idea is that all communication to the team, from the management or anyone else, should be facilitated through the Scrum Master. This is done to ensure that the team is not disturbed in its work, and to make sure that the Scrum Master always knows what is going on.

To that end, it is also the Scrum Master's responsibility to make sure that the team does the work expected of them. He needs to make sure the communication in the team is flowing, and that everyone does their best to get the task done in time. One way he does this, is by conducting Daily Scrum Meetings¹ each morning, and fix the problems brought up at those meetings. In this way, the team members themselves do not have to use their working hours fixing minor problems, but can focus on doing the tasks they are supposed to.

Another major responsibility of the Scrum Master is to work together with the management and the Product Owner on the Product Backlog (See section 6.2.2 on the next page). Afterwards, it is also his job to make sure that the development team picks out a reasonable amount of the Product Backlog as working assignments for the next Sprint (See section 6.2.3 on the facing page).

All of this makes the Scrum Master a very important part of Scrum development. Without a Master, Scrum would not work out the way it is supposed to. It is therefore also very important that the Scrum Master dares to make the decisions expected of him, since some of his jobs are not normally something anyone would do in a workplace - mainly, he should not do as management says, but rather what the team would like.

¹Which will be expanded upon in section 6.2.4 on the facing page.

6.2.2. Product Backlog

The Product Backlog is made solely by the Product Owner, as he is the one paying for the product, and therefore has the final word in what the product specifications should be. Basically, the Product Backlog is a list of features that the Product Owner wants in the final product. These can vary wildly, from letting users type their name as a property of a file description, to letting administrators decide who gets access to the database. More or less, any conceivable feature or function of the final program can, and should, be described in the Product Backlog.

After the Product Owner has made the Backlog, and prioritized the features as he wishes, it is then shown to and discussed with the management and the Scrum Master. They will then reorder the Backlog according to assignment workload, connections between features and so on. When all agree that the Backlog is as it should be, the Scrum Master takes it to the development team who then picks an appropriate part of it as assignments for the next Sprint. This way, the team always knows what to do at any time during development, and they have a clear, reachable goal for each Sprint.

6.2.3. The Sprint

A Sprint is normally a period of 30 days, although it can be more or less depending on the teams preferences. The idea of a Sprint is to commit to a certain amount of assignments and then make sure to have them done by the end of the Sprint - neglecting any other pressing matters in that period of time. Other assignments will have to be put into the Product Backlog during the Sprint and can then be done in the next Sprint.

Of course, a Sprint is allowed to deviate from the course set by the Backlog, but that is solely a decision made by the Product Owner and the management. The development team itself can only comment on what would be best given the current situation of the development.

6.2.4. Daily Scrum Meeting

During a Sprint, the team members work uninterrupted by outside sources, as these would have to go through the Scrum Master first. To make sure that the team is working as intended, the Scrum Master will therefore conduct a Daily Scrum Meeting each morning, where the team will discuss current development, and the Scrum Master can bring information from the management to the team. During this meeting, anyone can be present and especially the upper management is more than welcome to attend. This is to make sure that anyone, with interest in the project of the group, knows what is going on at all times.

It is, however, strictly forbidden for any non-team members to say anything, or even raise their hand, at these meetings. They are to remain quiet during the whole session - this includes any management personnel present. This is to make sure that it remains a meeting for the team, and to make sure that it does not take more than approximately 15 minutes. Team members take turns telling:

- What they have been doing since the last Daily Scrum Meeting - to bring the rest of the team up to par.
- What they will be doing until the next Daily Scrum Meeting.
- What might make it harder, or impossible, for them to do this. This includes any work related problems, such as technical problems, complaints from management, noise in the workspace and anything else that disturbs the member. But more

importantly, it also includes any personal problems a member might have. In case of severe personal problems, he does not have to explain himself during the meeting, but he should tell the Scrum Master afterwards.

Each point is to be as short as possible to make the Daily Scrum quick and concise. This makes it a valuable tool to quickly ascertain the situation in the group.

Part II

Analysis and Design

Analysis of the Proposed F-Klub Website

This chapter will feature the analysis of the system to be developed in the project. The theory from chapter 2 on page 10 will be used as a basis for the analysis. The chapter will end with a short summation of what the project group learned during the analysis, followed by the problem statement for the project.

7.1. System Definition

As the project proposal already contained a requirement list, the project group agreed that this list would be used as foundation for the development phase. Therefore, a customer meeting with the F-Klub was deemed unnecessary. It is instead assumed that the requirement list in the project proposal is the result of a fictional meeting. The list can be seen in appendix B on page 108.

The project group reserves the right to edit the list if needed.

7.1.1. FACTOR and System Definition

The FACTOR criterion, which was explained in section 2.1.2 on page 11, and a system definition based on this is needed as basis for further analysis tasks. The developed FACTOR list contains:

- **Functionality** - *Ordering and payment of articles from the shop. It should also be possible to browse an article catalogue and rate or comment on the articles. Additionally, the system should be able to support several different usertypes as well as managing an inventory of articles.*
- **Application Domain** - *The F-Klub and its members.*
- **Conditions** - *The system will be used by students at the University, and it will be administrated and controlled by a few chosen ones amongst the users - namely the F-Klub board. As so, the administrators of the system do not necessarily have any education regarding the running of a company.*
- **Technology** - *The system will be developed on the ASP.NET-platform and will be run on standard browsers. It is not expected that the users of the system will have high-end hardware, as it is not necessary, because only an internet connection and an internet browser is needed, to use the system. A Windows server will be required,*

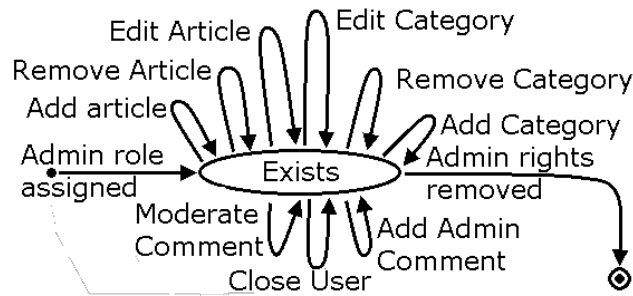


Figure 7.1.: State chart for the admin class.

because the system will be developed with Microsoft® software, and therefore also make sense that the system will be runned on a Windows platform.

- **Objects** - *The problem domain will consist of the users of the system (customers, guests and administrators), articles in the system and the shipping company.*
- **Responsibility** - *A webshop with support for different usertypes.*

Final System Definition

A computerized system, which will be used by the F-Klub and its members, facilitating ordering and payment of articles from the shop, and managing an inventory of articles. It should also be possible to browse an article catalogue and rate or comment on the articles. The system will be used by both administrators and normal users. Also, guests should be able to browse the catalogue, but not buy articles. The system will be administrated and controlled by a few select people amongst the users - namely the F-Klub board. As so, the administrators of the system do not necessarily have any education regarding the running of a company. The system will be developed on the ASP.NET-platform and will run on standard browsers. It is not expected that the users of the system will have high-end hardware, and as such it will not be required. The system will be developed for a Windows server.

7.2. Problem Domain

This section will apply the theory described in section 2.2 on page 11, and analyze the problem domain of the proposed F-Klub system.

7.2.1. Classes

Every class found in the problem domain during the analysis, will be noted here and followed by a behavior model for each individual class.

- **Admin**

Definition: The administrators of the webshop, who can edit information about articles, for instance price and article description. They can also access the inventory to monitor the stock.

Behavior: The state chart for the admin class can be seen in figure 7.1.

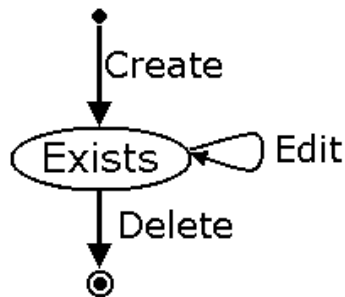


Figure 7.2.: Generic state chart, which is true for several classes.

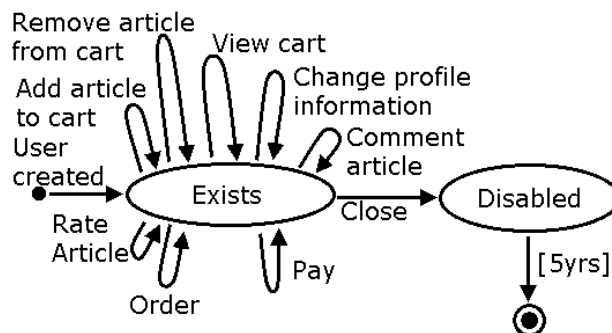


Figure 7.3.: State chart for the customer class

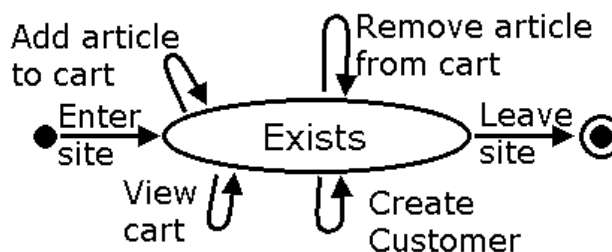


Figure 7.4.: State chart for the anonymous class.

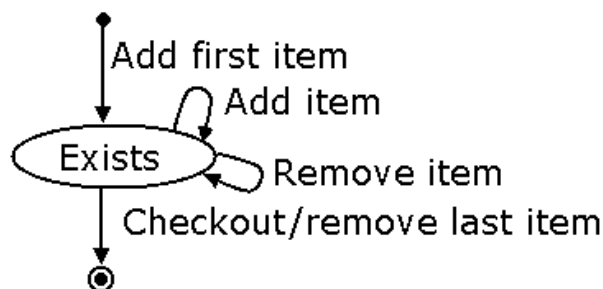


Figure 7.5.: State chart for the inCart class.

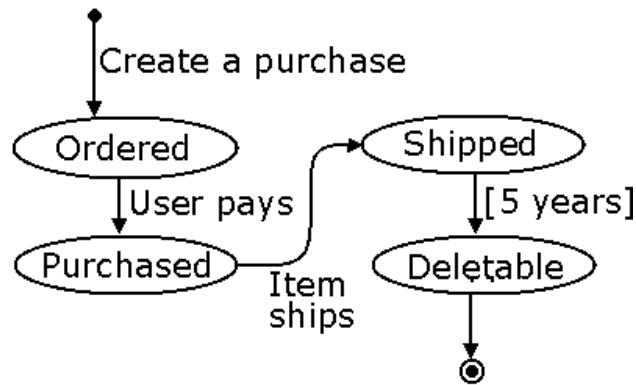


Figure 7.6.: State chart for the purchase class.

- **Anonymous**

Definition: Users of the webshop who have not yet created an account. They can put articles in the cart and browse the catalogue of articles. Anonymous users cannot complete purchases and their cart will be deleted after some time of the user being idle.

Behavior: The state chart for the anonymous class can be seen in figure 7.4 on the facing page.

- **Article**

Definition: The articles that are available for purchase in the shop. This will contain a brief description of each article, as well as any special attributes they might have.

Behavior: A generic statechart, which fits the article class, can be seen in figure 7.2 on the preceding page.

- **InCart**

Definition: A complete overview of all wanted purchases by any kind of user. Every user has access to his own articles in the cart, which they can view at any time. When a user creates a purchase, the articles of that user are moved from the cart to the purchase object.

Behavior: The state chart of the inCart class can be seen in figure 7.5 on the facing page.

- **Category**

Definition: A collection of articles, which have similar attributes, and are therefore grouped together for better overview.

Behavior: A generic statechart, which fits the category class, can be seen in figure 7.2 on the preceding page.

- **Customer**

Definition: Created users, who are able to order, view previously completed orders, and complete transactions. They are also able to comment on all articles, and rate articles that they have previously purchased.

Behavior: The state chart of the customer class can be seen in figure 7.3 on the facing page.

- **Comment**

Definition: A written comment given by registered users. A comment can also be an answer to another comment, given by an administrator.

Behavior: A generic statechart, which fits the comment class, can be seen in figure 7.2 on page 52.

- **Inventory**

Definition: A collection of all items in stock, with information on how many articles are left, and when new articles should be ordered from the supplier.

Behavior: A generic statechart, which fits the inventory class, can be seen in figure 7.2 on page 52.

- **Purchase**

Definition: An overview of each purchase made within the system. This includes articles, amount, price, shipping address, etcetera. This is used by both the administrators, to ship out articles, and by the users, to keep track of previous purchases.

Behavior: The state chart of the purchase class can be seen in figure 7.6 on the previous page.

- **Rating**

Definition: A rating of the individual product given by a user who have previously purchased it. The users who browse the webpage will not see the individual ratings, but only an average computed by the system.

Behavior: A generic statechart, which fits the rating class, can be seen in figure 7.2 on page 52.

7.2.2. Structure

This section shows and describes the program structure based on section 2.2.2 on page 12, including clusters and class connections.

The structure of the analyzed system can be seen in figure 7.7 on the facing page, and contains the classes mentioned in section 7.2.1 on page 51. The Article class aggregates the Comment and the Rating class. These three classes can also be grouped as a cluster, as they are closely related. The Comment and Rating classes are used as user feedback for the articles, and are therefore always connected to an Article.

The Inventory and Category class both aggregates the Article class. Each Article belongs to a Category, which unifies several articles, whereas the inventory contains an overview of the stock. An administrator (an instance of the Admin class) is able to edit the Inventory and Category class, and therefore, these classes must be associated.

Both the InCart and the Purchase class are aggregations of the Article class, because the purchases and the cart contains all articles in the cart/purchase. The Customer class is an aggregation of the Purchase class, as each Customer can view his former purchases.

The Cart class aggregates the Anonymous and Customer class, as each user, registered (customer) or anonymous, have access to his own articles in the common cart. The Customer class is also an aggregation of the Purchase class, as all customers should be able to access previous purchases.

7.2.3. Event List

Hereby a summation of all the events that can, and will, occur during the use of the webshop. Each event is explained in detail and linked to certain functions that directly depend on it. A detailed description of what an event is can be found in section 2.2.1 on page 11.

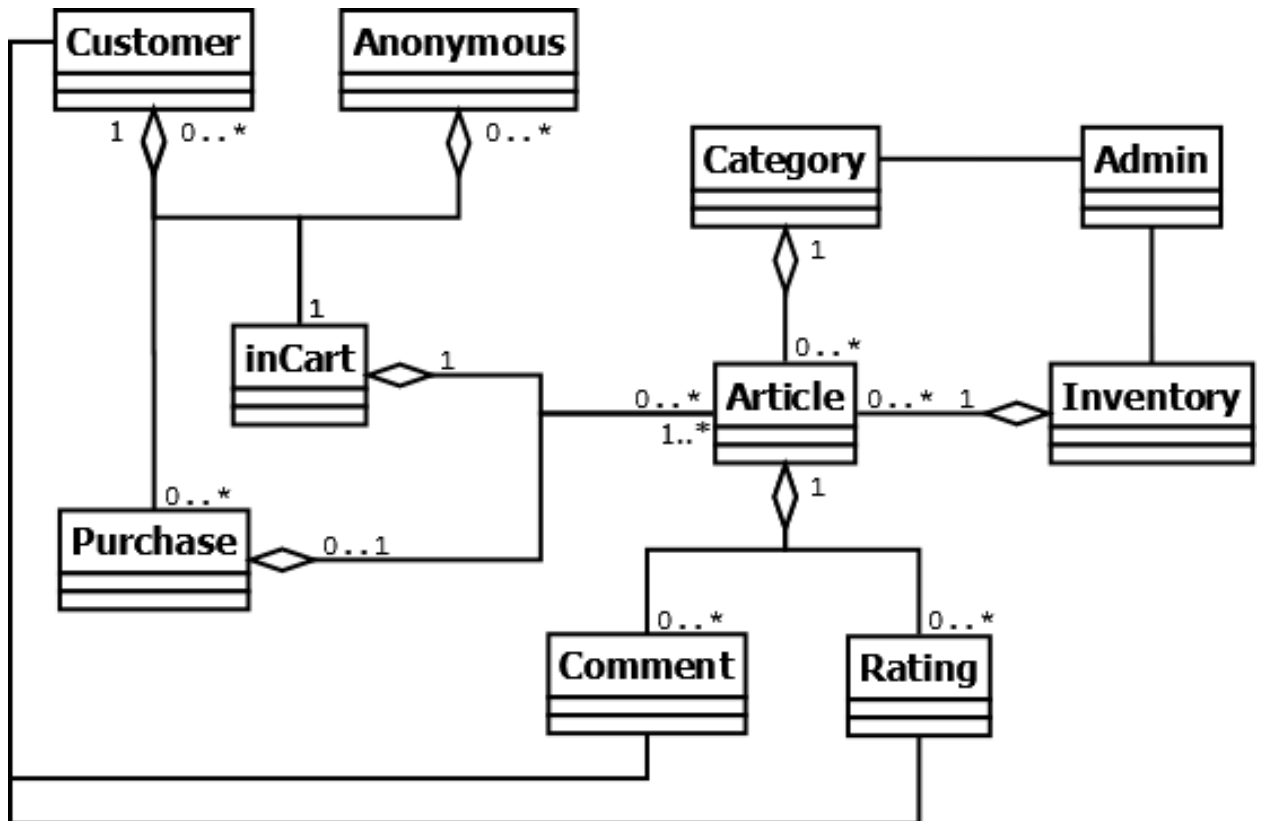


Figure 7.7.: The class diagram as it was designed in the analysis.

- **Order** - Occurs whenever a registered customer checks out his cart, and thereby agrees on buying the products in the cart. The event initiates the function *Payment* which in turn will initialize the event *Paying*.
- **Paying** - Occurs when a customer finalizes an order and submits payment information. The customer will then wait for the shop to send the items and first then be charged for the purchase. Furthermore, the event will initiate the function *Confirm order*.
- **Customer feedback** - Occurs when the customer has received the purchased product(s), and wishes to give feedback on how satisfying the product was. The customer can then look through his completed purchases and rate them according to their liking. This initiates the function *Rate article*.
- **Customer comment** - Occurs when a customer submits a comment on an article. The event initiates the function *Comment on article*.
- **Administrator replies** - Occurs when an administrator replies to a comment. This will initiate the function *Add admin answer to comment*.
- **Put article in cart** - Occurs when a customer or an anonymous user puts an article in their cart. The event initiates the function *Add article to cart*.
- **Remove article from cart** - Occurs when a customer or an anonymous user removes an article from the cart, if added. The event initiates the function *Remove article from cart*.

Event/Class-table	Admin	Anonymous	Article	InCart	Category	Customer	Comment	Inventory	Purchase	Rating
Order			x	x		x			x	
Paying						x		x	x	
Customer feedback			x			x				x
Customer comment			x			x	x			
Administrator replies	x		x				x			
Put article in cart		x		x		x				
Remove article from cart		x		x		x				
Browse inventory		x	x		x	x				
Add article to inventory	x		x		x			x		
Change article information	x		x		x			x		
Delete article from inventory	x		x		x			x		
New customer		+		+		+				
Change customer profile information						x				
Close customer profile						+				

Figure 7.8.: The events and their associated classes.

- **Browse inventory** - Occurs when a user of any kind wishes to see the total product inventory. This will initiate the function *search article database* with a predefined parameter set to result in every possible article in the inventory.
- **Add article to inventory** - Occurs when an administrator adds a new item to the database. This will initiate the function *Add article to inventory*.
- **Change article information** - Occurs when an administrator wishes to change the information on a given product. This can be any attribute, from price to size. This will initiate the function *Change article information*.
- **Delete article from Inventory** - Occurs when an administrator wishes to remove an article from the list of available articles given to customers. This will initiate the function *Remove article from inventory*.
- **New customer** - Occurs when an anonymous user decides to create a user in the webshop, and this will enable the customer to purchase any wanted articles. This will initiate the function *Create user* and afterwards the *Change Customer profile information* function.
- **Change customer profile information** - Occurs when a customer wants to change his personal information, or when the user is first created. This will initiate the function *Change Customer profile information*.
- **Close customer profile** - Occurs when a customer no longer wishes to have a user in the database, he will send a request via the system and an administrator will initiate the function *Close customer profile* which will put the profile on standby, and may be deleted after 5 years [8].

The events are associated with various classes. These associations have been found by analyzing the proposed system, and they can be seen in figure 7.8.

7.3. Application Domain

This section will feature the analysis of the application domain. The contents will follow the application domain part of the analysis theory in section 2.3 on page 15.

7.3.1. Actors

The following section contains a list of all the actors in the system. For a description of what actors are, please refer to section 2.3.1 on page 16.

7.3.1.1. Administrator

- **Goal:** A person who administrates the system, which for example means adding and updating articles or replying to and moderating comments.
- **Characteristics:** There *can* be more than one administrator of the system, but this is not necessarily the case. He will have access to a special part of the webshop, which is not visible to anyone else.
- **Examples:** Administrator A is told by the owner of the webshop, that a new brand of articles is being added to the featured articles. He will then add these to the system, including prices, description and inventory amount.

Administrator B is notified that a comment is harassing to other user. He will then find the comment and delete it or edit it as necessary.

7.3.1.2. Customer

- **Goal:** The customer is able to browse the catalogue and put things in his shopping cart. The user can also checkout, and thereby order the desired articles.
- **Characteristics:** The customer will have access to more features on the webshop, than an anonymous user. This includes a profile page and a purchase overview.
- **Examples:** Customer A browses through the articles and finds some articles, he wants. However, he realises he has not got enough money at the moment, so he logs out, and leaves the webshop. A few days later he returns to the webshop, logs in, and finds out that his cart has been saved, and is still ready for him to order when he has enough money.

Customer B is browsing through the webshop, and notices an article, with a question from another user about the article. Customer B has bought that article a while ago, and is therefore able to answer the question, by posting a message with an answer.

7.3.1.3. Anonymous

- **Goal:** The anonymous user is able to browse through the webshop, and put articles in the cart. He can then choose to create a user on the webshop, to be able to purchase the content in the cart, if not the cart is lost after a short period of idling.
- **Characteristics:** Any person who visits the webshop is handled as an anonymous, until they either create a user, log in as a customer or an administrator, or leave the webshop.

- **Examples:** A visitor of the webshop stumbles upon an item he wants and puts it in the cart. By pressing checkout, he is informed that he needs to register on the webshop, to be able to purchase his articles. If he does so, his usertype will be changed from anonymous to customer, allowing the purchase to happen.

7.3.2. Use Cases

All possible actions, a system actor can carry out, is called *Use Cases* and now that the actors have been found, it is possible to find use cases, the actors can end up performing. The advantages with creating and using these use cases is, that the buyer of the system can write their own use cases, which gives the development team the ability to develop exactly the features, the owner of the system wants.[9]

In this project, the use cases have already been more or less specified by the F-Klub, as the requirements were described in the assignment. All the use cases, and their associated actors, can be seen in figure 7.9 on the facing page. The use cases in the section are made according to the use case theory described in section 2.3.1.1 on page 16.

- *New Customer*
 - **Use Case:** An *anonymous* user chooses to register himself on the FHOPPEN webshop. He fills out information about himself: name, address, postal code and email address, and specifies his desired password. When he has submitted this information, he is then able to log in to the system as a *Customer*.
 - **Objects:** Anonymous, Customer
 - **Functions:** Create customer.
- *Search Article*
 - **Use Case:** An *anonymous* user or a *customer* wants to search for an article. He types in the search word, and a new page appears, with the search result. It is then possible to click on the search results to view the given article.
 - **Objects:** Article
 - **Functions:** Search article database
- *View Article*
 - **Use Case:** While browsing through some articles, an *anonymous* user or a *customer* wants to know additional information about a given article.
 - **Objects:** Article, Category
 - **Functions:** View article, Browse category
- *Put Article In Cart*
 - **Use Case:** When an *anonymous* user or a customer has found an article he wants, he puts it in the cart through the interface.
 - **Objects:** InCart, Article, Customer
 - **Functions:** Add article to cart

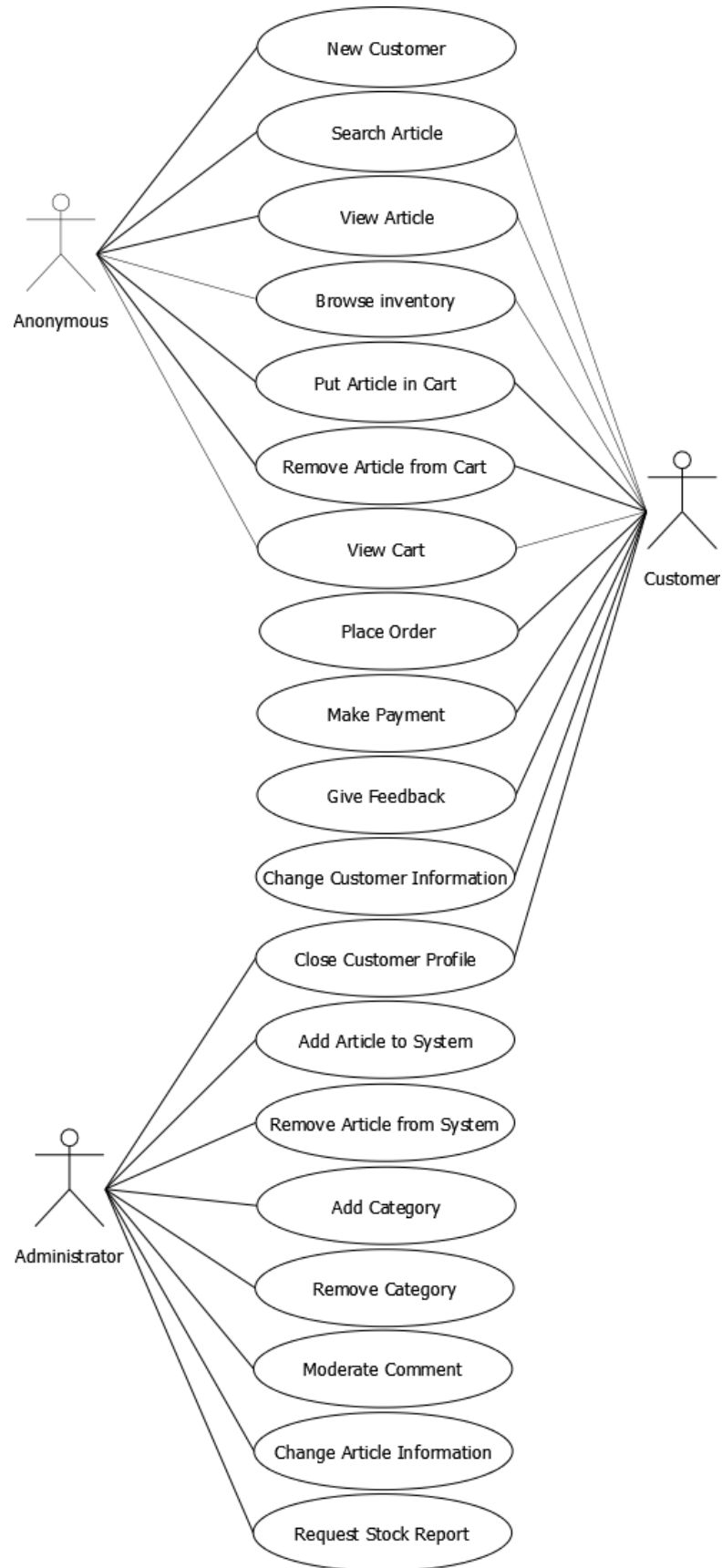


Figure 7.9.: The use case diagram of this project

- *Remove Article From Cart*
 - **Use Case:** A *customer* or an *anonymous* user wants to remove an article already added to the cart.
 - **Objects:** InCart, Article, Customer
 - **Functions:** Remove article from cart
- *View Cart*
 - **Use Case:** A *customer* or an *anonymous* user wants to look at some detailed information about the contents of his cart. He can do this in the cart overview.
 - **Objects:** InCart, Customer
 - **Functions:** View cart
- *Place order*
 - **Use Case:** A user has added at least one article to his cart and now wishes to purchase the content of the cart.
 - **Objects:** InCart, Article, Customer
 - **Functions:** Checkout
- *Make Payment*
 - **Use Case:** A *customer* checks out his cart and is prompted for credit information, he submits it and receives a receipt for the purchase. The shopowner will wait until such a time that he has sent the purchased item, and then charge the user for the agreed amount.
 - **Objects:** Purchase, InCart, Customer, Admin
 - **Functions:** Payment, Confirm order, Update purchase information
- *Give Feedback*
 - **Use Case:** A *customer* has bought an article, and he wants to rate it. He could also have a question regarding an article, he considers to buy, so he wants to post a comment with his question.
 - **Objects:** Article, Comment, Rating, Customer, Purchase
 - **Functions:** Comment on article, Rate article
- *Change Customer Information*
 - **Use Case:** A *customer's* address has changed and therefore, he wants to change the address in his profile information.
 - **Objects:** Customer
 - **Functions:** Change customer profile information
- *Close Customer Profile*
 - **Use Case:** If a *customer* does not want to be a part of the webshop anymore, he has the option of closing his profile, which will tell the *administrator* to delete it. However, the profile may not be deleted for the next five years[8]. The *administrator* is also able to close a certain profile if it is considered offensive.
 - **Objects:** Customer, Admin
 - **Functions:** Disable customer profile, Delete customer

- *Add Article to System*
 - **Use Case:** An *administrator* wants to add an article to the inventory of the shop and thereby make it available for purchase.
 - **Objects:** Article, Admin, Category, Inventory
 - **Functions:** Add article to inventory
- *Remove Article From System*
 - **Use Case:** An article is not available anymore, and it needs to be removed from the system. The *administrator* should remove the article from the database.
 - **Objects:** Article, Admin, Inventory, Category, Purchase¹
 - **Functions:** Removes article from inventory
- *Add Category*
 - **Use Case:** A new type of article is to be added to the system and thus a new category needs to be created.
 - **Objects:** Category, Inventory, Admin
 - **Functions:** Add category to inventory
- *Remove Category*
 - **Use Case:** A certain type of articles are to be deleted from the webshop, as they are no longer available for some arbitrary reason.
 - **Objects:** Category, Inventory, Admin, Purchase²
 - **Functions:** Remove category from inventory
- *Moderate Comment*
 - **Use Case:** Even though the comment field on every article page is meant for asking questions about the given article, some people tend to use the comment field to harass other people. However, the *administrator* is able to take a look at the comments and delete those, that do not have any relevance to the article. Also it is possible for the *administrator* to add an answer to an existing comment if needed.
 - **Objects:** Comment, Admin, Article
 - **Functions:** Moderate comment, Add admin answer to comment
- *Change Article Information*
 - **Use Case:** An attribute of an article has, for some arbitrary reason, changed. Be it a new price or description. Therefore an *administrator* wishes to correct the article information.
 - **Objects:** Article, Category, Admin
 - **Functions:** Change article information

¹The purchase object is included since the article should not be able to be completely removed if the article has been sold within the last five years.

²Again the system needs to make sure that no customer bought an item from the category for the last five years.

- *Request Stock Report*
 - **Use Case:** At some point, an article will run out of stock. Before this happens, the *administrator* should receive a warning about the article.
 - **Objects:** Article, Inventory, Admin
 - **Functions:** Low stock warning

7.3.3. Function List

All the functions mentioned in the use cases need to be developed for the system to function efficiently according to the needs of the F-Klub. They will be listed here in the form of a list containing the necessary information:

- **Name - Complexity - Function type** - Description

The complexity is an evaluation of how hard it will be to implement the feature in the system, according to the kind of code needed. The function type is one of the types mentioned in section 2.3.2.1 on page 17.

The functions are listed in the order they appear in the use cases:

- **Create customer - Complex - Update** - Creates a customer, an upgrade from the anonymous user. The customer is then able to comment and rate articles, he has bought.
- **Search article database - Complex - Read** - Search for any article in the inventory database. Gives a list of possible matching articles.
- **View Article - Simple - Read/Compute** - Allows a user to view an article and its information.
- **Browse category - Simple - Read** - A user of the system browses through the articles in the database which are sorted in categories.
- **Add article to cart - Simple - Update** - Puts the selected article in the cart.
- **Remove article from cart - Simple - Update** - Removes the selected article from the cart.
- **View cart - Simple - Read** - Takes the user to the cart screen where he can see the content of the cart.
- **Checkout - Complex - Update** - Takes the user to the checkout screen where last minute changes to the shopping cart can be made, before confirming the order. This also moves the articles from the cart to a purchase object.
- **Payment - Complex - Compute/Update** - Lets the user pay for the articles of the confirmed order.
- **Confirm Order - Moderate - Signal** - Sends out the confirmation email that tells the customer that the purchase has been submitted to the system, and informs the administrators that a shipment needs to be send out.
- **Update Purchase Information - Moderate - Update** - Used to update the state of a purchase after the order has been submitted to the system. This update could be things such as "Shipment sent" or "Order delayed".
- **Comment on article - Simple - Update** - Adds a user comment to an article message board.

- **Rate article - Simple - Update** - Sets the user's rating for a specific article.
- **Change Customer profile information - Simple - Update** - Lets the customer change his profile, for instance, change his address.
- **Disable customer profile - Simple - Update** - Disables the possibility to login through the profile.
- **Delete customer - Simple - Update** - An administrator can delete a customer, and thereby remove the customer information. This can only be done, after the profile has been disabled for a period of 5 years.
- **Add article to inventory - Moderate - Update** - Adds an article to the inventory list and thereby makes it available for purchase.
- **Remove article from inventory - Simple - Update** - Removes an article from the inventory list, and thereby makes it unavailable for purchase.
- **Add category to inventory - Simple - Update** - An administrator adds a new category to the system to accommodate new needs.
- **Remove category from inventory - Simple - Update** - An administrator deletes a category from the system.
- **Moderate comment - Moderate - Update** - Gives the administrator the ability to edit or remove comments on article message boards.
- **Add admin answer to comment - Moderate - Update** - Adds an administrator answer to a comment on an article message board.
- **Change article information - Simple - Update** - Change the name, price or description of an article.
- **Low Stock Warning - Complex - Signal** - The administrator is warned, if an article stock is running low.

7.4. Analysis Summation

The analysis helped the project group figure out which areas to focus on during the development of the project. The requirement list already set forth by the F-Klub will be the basis of the product requirements. It would not be possible, however, to develop every single feature, without all or some of them lacking in functionality or efficiency. Instead it was decided that the focus should be to make the administrative part functionable, and make a decent interface for the administrator commands. That way, a stable program would be available at the end of the project, and it could be completed in full by adjusting it to the future administrator's preferences.

The analysis showed that the program would demand a lot more functions than the team anticipated. As such, some of the predetermined ideas about the system need to be redefined, for instance the different user types, which will be a lot more complex than presumed. Other than that, most of the demands set forth by the F-Klub, described the necessary components well. With the analysis done, it is now possible to set up a general problem statement for this project.

7.4.1. Problem Statement

"How is a system that lets a group of IT-inexperienced individuals sell their products over the internet designed, so it is easy to manage? How does one keep track of the articles of the webshop, while making those data easily accessible for both the customers and administrators? Lastly, how can it be made sure that the customer's wishes are taken care of, within the program?"

System Design

This chapter will feature the system design based on the system analysis. First, the general task will be described, and any corrections needed compared to the analysis will be explained. A review of the design of the technical platform and the program architecture will follow, and the chapter will end with a design of the user interface.

8.1. The Task

The task in this project is to make a webshop for the F-Klub, which has been named FHOPPEN. The webshop is going to sell different merchandise, for instance; pens, mugs and shirts. The system should also be able to be implemented on a larger scale.

8.1.1. Corrections

Changes to the classdiagram has been necessary, as the Inventory class has been removed. The functionality it provided was better implemented in the Article class instead. This was done, to make a more functional and easier-to-understand program. Furthermore, the Purchase class needed to be split up. It is now two classes called Purchase and OrderedArticle, to provide the functionality originally assigned to the Purchase class. A detailed explanation of the new architecture will be found in section 8.4 on the next page.

The state chart diagrams has not changed during the design phase, and have therefore not changed. They can be seen in the list at section 7.2.1 on page 51.

8.2. Criteria

As described in section 3.1 on page 20, the demands to the system needs to be found, before it is possible to define the architectural structure of the system.

The system being developed in this project, should be prioritized as:

- **Usable:** It is *less important* whether the system is usable, as it is not the focus of this semester, if a user can interact with the system.
- **Secure:** It is also *less important*, since the payment will be handled by a third party company which will be responsible for the security. Another security aspect is the storage of passwords, but it is not a major concern.
- **Efficient:** The efficiency of the system is *irrelevant*, since a server is at the disposal of the project group, along with free Microsoft® software. This means that it is the

responsibility of the project group, to find the right software to develop the system. This will also be discussed in section 8.3.

- **Correct:** The priority of the correctness of the system is *very important*, as the system should fulfill the requirements set in the system analysis.
- **Reliable:** It is *important*, that the system is reliable as the system should perform the required tasks, when asked.
- **Maintainable:** The priority of the maintainability of the system is *irrelevant*, as it is not a focus of this semester.
- **Testable:** It is *important*, that the system is able to perform the tasks it is supposed to do. Therefore, testing is an important part of the development, which makes the testability of the code a necessity.
- **Flexible:** As the aim is to deliver a complete system from the start, changing the features of the system is not a focus area, therefore it is *less important*. However, as the system is developed object orientedly, it is still relatively easy to implement new classes if necessary.
- **Comprehensible:** Since the final system is not to be administrated by IT-professionals, it is *important* that the general usage of the system is easily understandable.
- **Reuseable:** It is *less important*, because it is being developed object orientedly, resulting in automatic reuseability. That is, it **can** be reused, but it is not a major focus area.
- **Portable:** Portability is *irrelevant*, as it is not a demand set forth by the F-Klub.
- **Interoperable:** It is *less important* if the system is interoperable, since it is presumed that no other systems are to be used along with it.

8.3. Technical Platform

In this project, the platforms ASP.NET and Microsoft® SQL Database have been used. As ASP.NET is a Microsoft® developed programming language, the option requiring the least configuration was to select a Microsoft® server, and thus, Microsoft® Server 2008 was chosen.

First, the project group had decided to develop the system using PHP, an object oriented web programming language, but the only reason for this was the unawareness of the close relation between ASP.NET and the programming language C#. As C# is the programming language used in the OOP course this semester, this was a straightforward choice. As the system being developed is a webshop, HTML and Cascading Style Sheets (CSS)¹ has also been used in the forming of the pages of the webshop.

8.4. Architecture

The architecture of the system has been changed since the analysis of the system. In this section, the reasons behind those changes will be explained. Furthermore, a complete overview of the architecture will be described and explained.

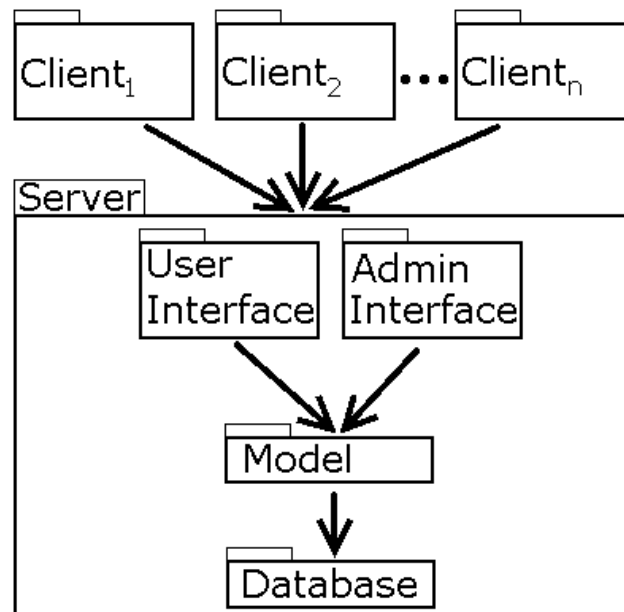


Figure 8.1.: The basic architecture of the Server-Client setup of the FHOPPEN webshop.

8.4.1. Component Architecture

In figure 8.1, the designed architecture for the FHOPPEN can be seen. Notice that it has been decided to avoid the functions layer, which will be explained in section 8.5.2 on the following page.

The webshop is based on a server-client architecture, in which the server handles all user requests, and returns an HTML-page to the user, which the user's browser interprets and converts to a viewable web page.

8.5. Components

A more detailed illustration of the webshop architecture can be seen in figure 8.2 on the next page. All interface drawing tasks is done by the Layout class, which calls various functions from the model layer. Each class in the model layer has a connection to the database, which setup ensures a quick response from the database.

8.5.1. Interface Component

The interface part of the page is responsible for making the website readable for a browser, which then makes it visible to the user. The interface layer consists of two components, the user interface and the administrator interface.

User Interface The user interface is what the user sees when he enters the page and during the navigation around the page. It makes functions that are available for the current usertype available, but other than that, it has no responsibility.

¹A tool to ensure consistent layout across all pages of the website.

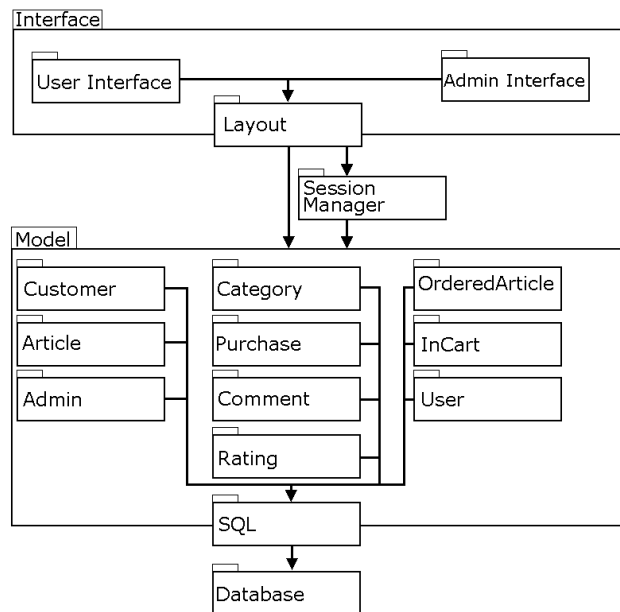


Figure 8.2.: A detailed figure of the component architecture of the FHOPPEN webshop.

Administrator Interface The administrator interface contains special administrator functions, that are located in the model layer. These functions must be separated from the ordinary interface, so no administrator is tempted to edit shop contents to his own advantage. The operations of the administrator interface are:

- *AddArticle*: Adds an article to the system.
- *RemoveArticle*: Removes an article from the system.
- *EditArticle*: Edits an existing article in the system.
- *AddCategory*: Adds a category to the system.
- *RemoveCategory*: Removes a category from the system.
- *EditCategory*: Edits an existing category in the system.
- *AnswerComment*: Replies to a comment from a user.
- *ModerateComment*: Deletes or moderates an inappropriate comment.
- *CloseUser*: Makes a user account inaccessible.

State Charts When a user is granted administrator privileges, he is able to add, edit and remove articles and comments, as well as moderate comments given by customers, and create administrator answers. He will also be able to shut down customer accounts. The only way an administrator object can cease to exist is, if the administrator rights are removed from the user. The state chart for the administrator class can be seen in figure 7.1 on page 51.

8.5.2. Function Component

A function component contains the most complex functions, whereas the simpler functions are assigned to classes in the model component. A complete list of all functions is used during the design of the function component, however, read and update functions of each class attribute are defined as operations, and will not be mentioned in the function list. The list is similar to the list in figure 7.3.3 on page 62.

- **Payment:** Belongs to the Purchase class in the model component.
- **Confirm Order:** Belongs to the Purchase class in the model component.
- **View Article:** Read function belongs to the Article class. The compute part of the function - the calculation of the rating - is done in the interface class for the article view.
- **Search article database:** Belongs to the Article class in the model component.
- **Low Stock Warning:** Belongs to the Article class in the model component.

As it can be seen, the analysis defines all functions to be simple so that they could be stored in the model component. The function component is therefore unnecessary, as it would only contain forwarding functions between the interface component and the model component.

8.5.3. Model Component

The model component represents the problem domain in its actual state. All data, and simple operations, exists in this layer. The actual architecture of the model component must be considered thoroughly, as it is one of the most basic components of the system architecture, which means other components depend on it. To ensure and clarify, which tasks each class must fulfill, the classes, their attributes and functions, must be defined and explained thoroughly.

8.5.3.1. Structure and class description

The structure of the model component can be seen in figure 8.3 on the following page. Each class contains its attributes and functions along with their types.

Article

The Article class contains informations about every article in the system.

List of attributes:

- *Name:* Name of the article.
- *Description:* A detailed description of the article.
- *Price:* The price of the article.
- *List of Ratings:* A list of all ratings given to the article.
- *List of Comments:* A list of all comments given to the article.
- *Unique Identifier(ArticleId):* Used to distinguish one article from another.

State Chart The Article class is relatively simple, as it, along with several other classes, can be created which makes it "exist", and then it can be deleted so it ceases to "exist". While existing, the only operation available to it is an edit function which does not change the state of the object in any way. The generic state chart (figure 7.2 on page 52) therefore holds true for this class.

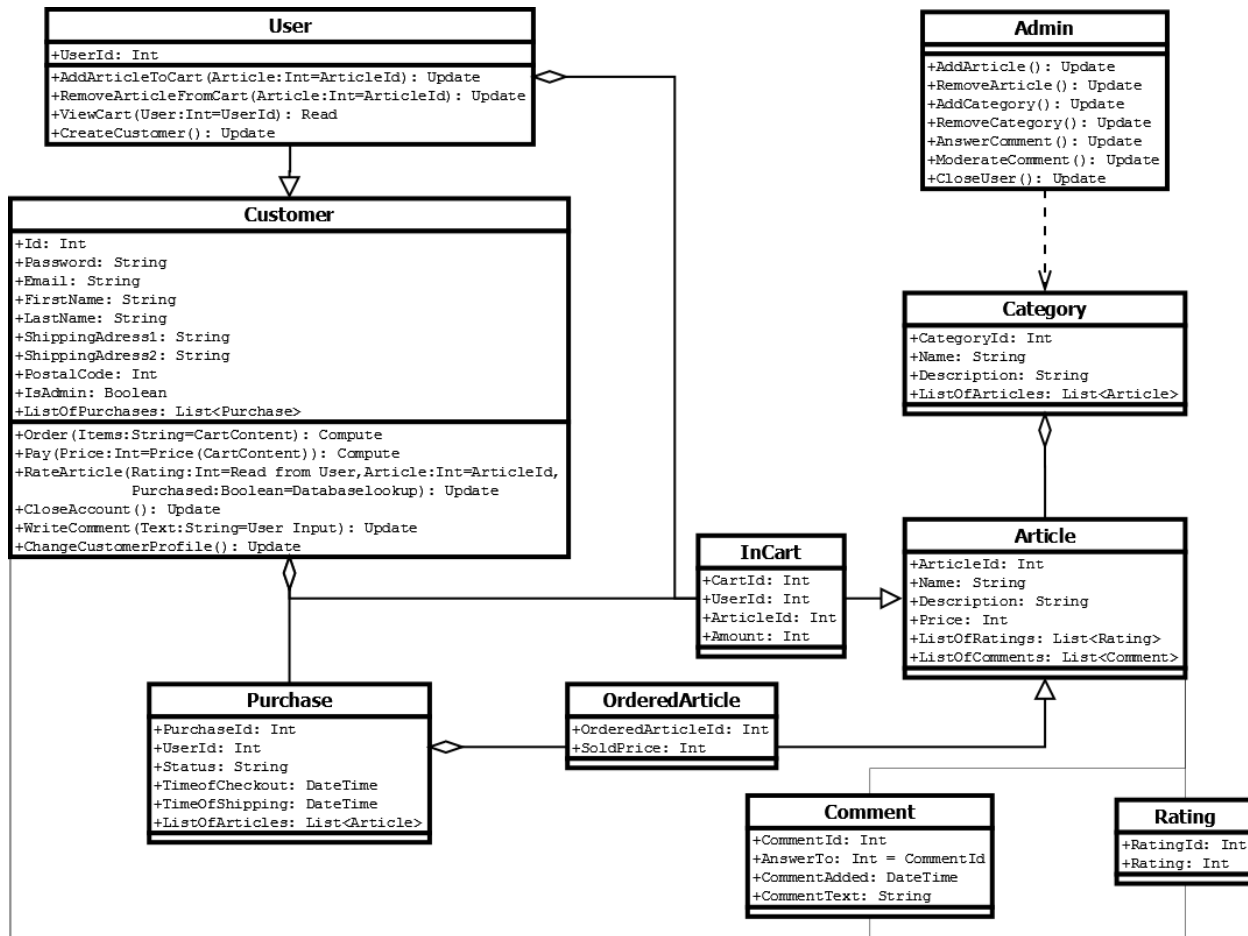


Figure 8.3.: The architecture of the model component.

Category

A category is a collection of articles with similar attributes. For example, milk and juice could be grouped into the category "Beverages".

List of attributes:

- *Name*: The name of the category.
- *Description*: A more detailed text, which describes the category.
- *List of Articles*: A list of all Articles in the category.
- *Unique Identifier(CategoryId)*: Used to distinguish one category from others.

State Chart An instance of the category class can be created, edited and deleted. Therefore it also uses the generic state chart in figure 7.2 on page 52.

Comment

A Comment is the means for a customer to ask questions about a certain article. It is possible to comment without having bought an article, but it is necessary to be logged in, to avoid spambots.

List of attributes:

- *UserId*: The user, who added the comment.
- *CommentAdded*: The time, the comment was posted.
- *CommentText*: The comment text.
- *Unique Identifier(CommentId)*: Used to distinguish one comment from others.
- *AnswerTo*: If an administrator answers to a user's comment, this attribute is the id of the comment in question.

State Chart An instance of the comment class can be created, edited and deleted. Therefore it also uses the generic state chart in figure 7.2 on page 52.

Customer

A customer is a registered user, who has access to all customer functionalities in the webshop.

List of attributes:

- *Password*: An encrypted string of the user's password.
- *Email*: The user's email.
- *FirstName*: The user's first name.
- *LastName*: The user's last name.
- *ShippingAddress1*: The first line of the user's address, whereto the order should be shipped.
- *ShippingAddress2*: The second line of the user's address, whereto the order should be shipped.
- *PostalCode*: The user's postal code.
- *List of Purchases*: A list of all purchases the customer has made.
- *ViewCart*: View details about the articles added to the cart.

- *Unique Identifier(CustomerId)*: Used to distinguish one user from others.

Operations:

- *Order*: Order the current content of the cart.
- *Pay*: Pay for an order, and request shipping.
- *RateArticle*: Set the rating for an article.
- *CloseAccount*: Close the account, so it becomes inaccessible.
- *WriteComment*: Comment on an article, for instance ask questions or elaborate a rating.
- *ChangeCustomerProfile*: Change the stored profile information.

State Chart When a customer is created, he is able to add and remove articles to or from the cart, view his cart, change his profile information, rate or comment on articles and order, and pay for articles. If the customer wants to close his account, or an administrator finds it necessary to do so, it will only be disabled, as customer information must be saved for at least five years [8]. When five years has passed, the customer will finally be deleted. This can also be seen in figure 7.3 on page 52.

InCart

The InCart class contains information on the articles, a user is currently considering to buy. If the user is registered, the cart content is saved between sessions.

List of attributes:

- *List of Articles*: All Articles, the user has added to the cart.
- *Amount*: The number of articles, the user wants to purchase.
- *Unique Identifier(CartId)*: Used to distinguish one cart from others.

State Chart An instance of the inCart class can be created, edited and deleted. Therefore it also uses the generic state chart in figure 7.2 on page 52.

OrderedArticle

The OrderedArticle class is an extension of the Article class, and is used when a customer has made a purchase.

List of attributes:

- *SoldPrice*: The price of the article, at the moment it was sold.
- *Unique Identifier(Id)*: Used to distinguish one ordered article from others.

State Chart An instance of the OrderedArticle class can be created, edited and deleted. Therefore it also uses the generic state chart in figure 7.2 on page 52.

Purchase

When a user orders a collection of articles, he creates a purchase object.

List of attributes:

- *Status*: A text message about the current status of the order (for instance, if an article is out of stock, and if the order therefore is on hold).
- *TimeOfCheckout*: The time and day, the user purchased the order - if it is not yet purchased, the attribute is set to a default value.
- *TimeOfShipping*: The time and day, the order was shipped - if it is not yet shipped, the attribute is set to a default value.
- *List of Articles*: A list of all Articles in the Purchase.
- *Unique Identifier(PurchaseId)*: Used to distinguish one purchase from others.

State Chart An instance of the Purchase class can be created, edited and deleted. Therefore it also uses the generic state chart in figure 7.2 on page 52.

Rating

A rating is an integer between one and five, both included, which represents the user's rating of an article.

List of attributes:

- *Rating*: The rating, the user has given the particular article.
- *Unique Identifier(RatingId)*: Used to distinguish one rating from others.

State Chart An instance of the Rating class can be created, edited and deleted. Therefore it also uses the generic state chart in figure 7.2 on page 52.

User

A user is an unregistered user at the site. A user is able to browse the page and add or remove articles to a cart, but nothing else.

List of attributes:

- *Unique Identifier(Id)*: Used to distinguish one user from others.
- *inCart object*: The cart that belongs to the user.

Operations:

- *AddArticleToCart*: Put an article into the cart.
- *RemoveArticleFromCart*: Remove an article from the cart.
- *ViewCart*: View details about the articles added to the cart.
- *CreateCustomer*: Become a member of the shop, and make it possible to order articles and give feedback.

State Chart When a user enters the page, he will be able to add articles to the cart and remove them again, along with viewing the cart, and register to become a customer. The user object will cease to exist, when the user leaves the page or when he signs up as a customer. A state chart of the User class can be seen in figure 7.4 on page 52.

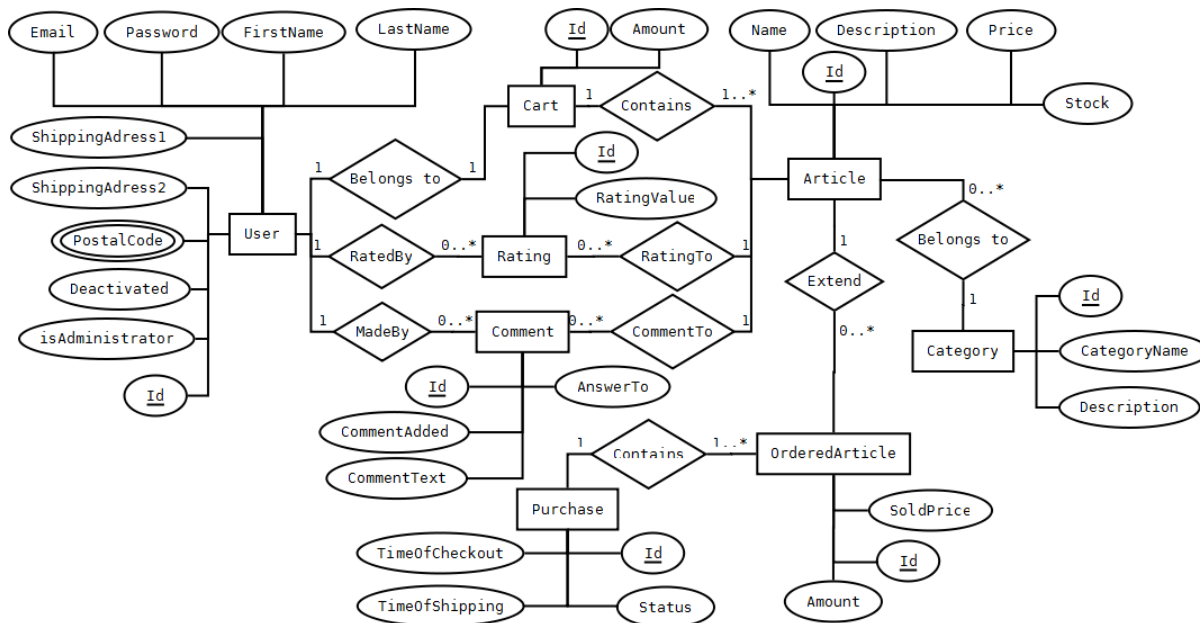


Figure 8.4.: Entity relationship model for the proposed F-Klub webshop.

8.5.4. Database Component

The database should contain all classes and attributes in the entity relationship diagram (E-R model) seen in figure 8.4. It contains eight classes, which represents a table each. In addition to those, a table of postal codes and their correlating towns will be added, to avoid redundant data in the database in the customer table. The reason herefor is that the postal code attribute contains multiple attributes, which values depends on each other (also shown in the E-R model as an oval with double edges).

8.5.5. Component Connection

To ensure high cohesion and low coupling between the interface component and model component, all connections are designed to be operation calls. This also applies, when the model component needs to import data from the database component. An illustration of the component connection can be seen in figure 8.5 on the next page.

8.6. User Interface

The user interface has not been a focus area in the lectures of this semester, and thus, it has not been prioritized by the developers. This does not mean that the user interface of the system has not been developed, but rather that it has not been developed according to any theory, but only what the developers thought would work. The design is supposed to resemble other webshops.

The page proposition is constructed to look similar to a frame. The left column is a menu, containing links to every category. The middle part contains the main content of the various pages. For example, when viewing an article, this part will contain all article information. An example can be seen in figure A.2 on page 104.

The pages should be linked together as seen in figure A.1 on page 103. The main page links to almost every page in the system, and vice versa. The administrator interface should actually be accessible from every page (as it is contained in the menu), but only

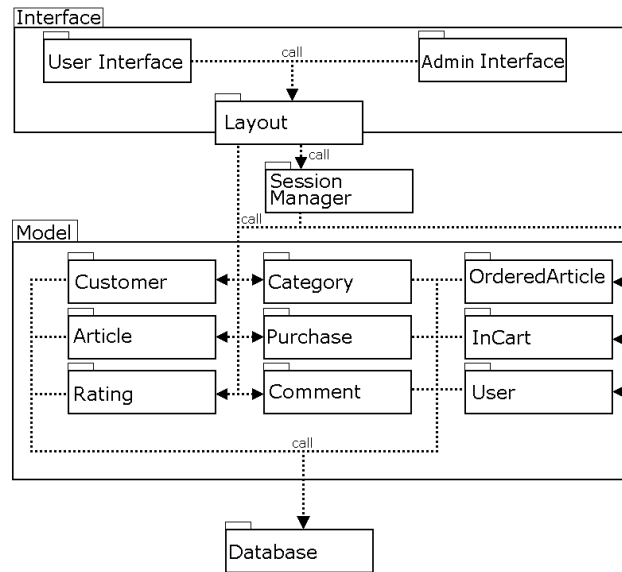


Figure 8.5.: Class and Component Diagram with inter-component connections.

the frontpage should be accessible from the administrator interface in order to separate it from the webshop.

The front page should contain some article suggestions for the user. The selection of these is done, partly by taking the newest articles, and partly by taking the most popular articles. This would look as shown in figure A.3 on page 105.

In the appendix A on page 102, other interface examples can be seen. These will, however, not be described, but they have been included for reference.

Part III

Product

Development

This chapter describes the work process of the project group during the development of FHOPPEN. It also explains the implementation of the agile development method Scrum which was described in chapter 6 on page 45.

9.1. Development Phase

For the development phase, the tables in the grouproom were rearranged into small workstations. These workstations were used to make small programming groups, consisting of two people per workstation, as seen in figure 9.1 on the next page. Each development team member would then work individually with an assignment, but if one got stuck, one could ask the other person at the workstation. If this person was not able to solve the problem either, it was allowed to ask the other members of the group.

Unscheduled breaks were used during the workday. These breaks were only used for getting out of the grouproom, get some fresh air and rest the brain. This would often result in a different view on a piece of code, and solve a problem, the person may have had before the break. These breaks were unplanned, since it was impossible to know when a groupmember needed a break. This was caused by great variety in the difficulty and frustration level of the tasks.

9.2. Agile Development in this Project

In this project, it was decided to try to incorporate the basics of the agile software development method Scrum. However, it would not be possible to use Scrum as described in chapter 6 on page 45, since this is a study project, and management would not be present in a relevant form. Furthermore, the development phase would not be long enough to warrant any kind of real iterative programming. However, several concepts were used, and not only in the programming phase, but also during the analysis and design phase, as well as the writing the report.

9.2.1. Analysis and Design

The most iterative part of the development was the analysis and design part. This was because both analysis and design was done over several weeks, each week adding new analysis methods and theories. Thereby, the System Analysis document slowly changed shape and was edited into the document in the beginning of this report. However, this was the only agile concept incorporated during the analysis phase, since most work was

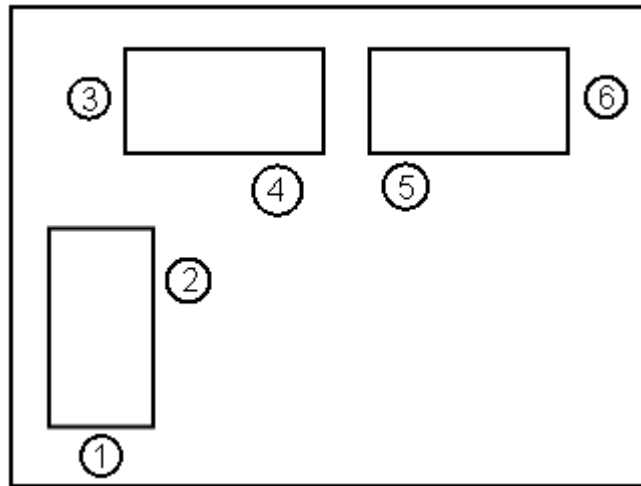


Figure 9.1.: This is how the workstations were positioned in the grouproom during the development phase.

done according to theories and workmethods learned during the Software Analysis and Design course of this semester.

9.2.2. Development

During the development phase, it was decided to use as many aspects of Scrum as possible. This section will cover most basic Scrum aspects along with argumentation for choosing or not choosing them for this project. Experiences gathered during the use of these methods will be found later in section 11.4 on page 98.

- **Scrum Master** - An actual Scrum Master was not used during the project for several reasons. Mainly, it was deemed unnecessary, since a Scrum master's primary assignment is to facilitate communications between the team and any outside sources. Since this was not a concern during this project, his influence would be limited. Instead a team member was assigned as group coordinator, whose job was mainly to keep an overview of the project and follow up on assignments. The Scrum master's other jobs, like managing Daily Scrums, were instead shared between the members.
- **Product Backlog** - The Product Backlog for this project was done slightly different than in most projects, however it was still fully implemented. All the assignments were written down on a piece of paper, and then put up on a bulletin board. Each team member participated in organizing the assignments according to not only importance, but also the difficulty of the assignment. The bulletin board then functioned as the Product Backlog.
- **Sprints** - Sprints were not used in this project, solely for the reason that the general program developing period was only a week long. It would not have benefitted to anything.
- **Daily Scrum** - The Daily Scrum was a major part of the development process, and was even used for writing the report. The team got together each morning and talked about the day before, and discussed how to proceed. It was not timed, as it was deemed necessary to learn how to use the Daily Scrum efficiently first. The

Daily Scrum only included the team members, and as such, the only input from outside the group came from the scheduled supervisor meetings.

FHOPPEN

10.1. Product Walkthrough

The functionality of the final system is similar to many other webshops. It allows visitors to browse articles, and allows registered customers to purchase articles. It is also possible for administrators to add and remove articles and categories. When first entering the site, any user has the option to browse the categories and see details and ratings of articles and comments left by other users. All users, logged in or not, have the option to put items in the cart, but in order to checkout they must log in.

Before one can log in as a customer, one must first be registered in the system. This is done by filling out a form on the website with personal information, namely the customer's name, email, and shipping address. Once registered, the user must use his email and password to log in to the system. This grants access to more features, such as checking out, commenting on articles, and rating bought articles.

The administrator is like a normal customer, but he has also access to the administrator interface which provides functions for adding, removing or updating articles and categories. This interface also provides the administrator with a list of the orders that needs to be shipped.

10.2. User Interface

The user interface contains a menu part and an area for the main content. The original design idea was to have two menu blocks, one in each side of the page. The left part should contain links to all categories. The right part should contain the cart and the login box, or, if a user was logged in, the user information box with links to previous orders and the user profile. The logout function should be there as well. Finally, the administrator control panel should be available for users, who had been given administrator rights.

However, the category box has, in the actual webshop, has been moved to the right side of the page to allow more space for the main content. The page has been optimized for a 1024 * 768-resolution, since research done by [10], shows that this is, currently, the most used resolution.

The interface can be seen in figure 10.1 on the facing page. The figure contains the front page of the webshop and is what the user will first see when he visits the webshop. As described, the main content of the shop is at the left and center part of the page, while the menu is to the right. On the front page, the main content area shows a few selected articles the user might be interested in, namely the most popular two articles and the two newest.

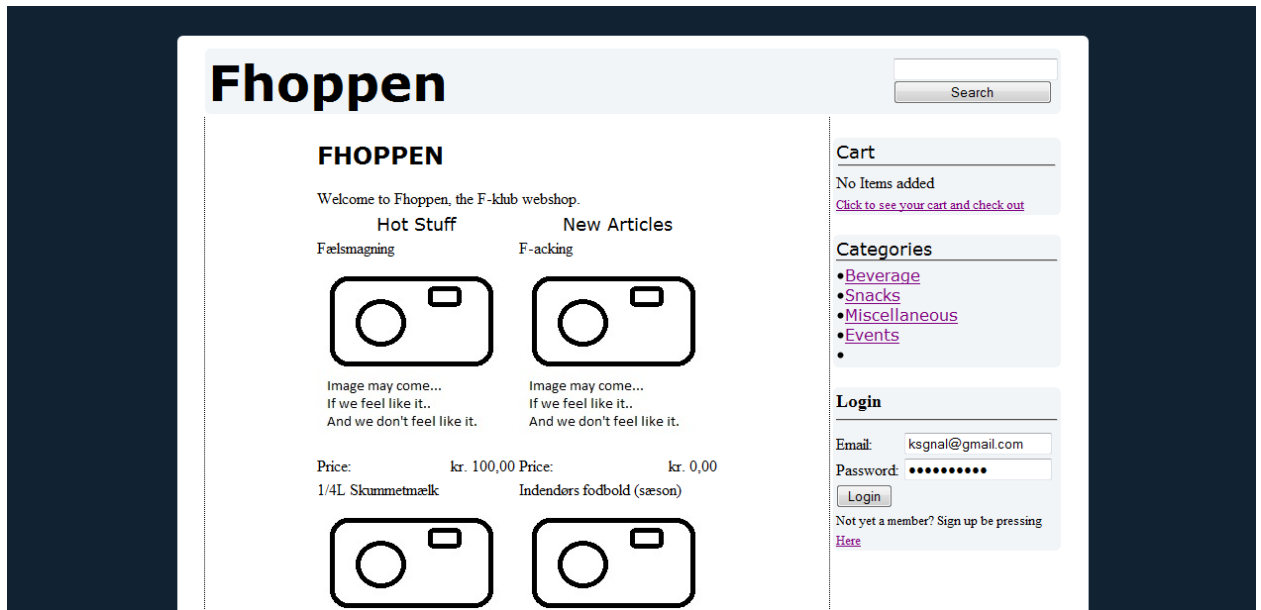


Figure 10.1.: The FHOPPEN frontpage.

When a user selects an article, possibly from the front page, he will be redirected to the page seen in figure 10.2 on the next page. It contains; a total rating of the article, an article description, an image of the article and an option to add the article to the cart. The right part of the page is the same as on the front page. If the user is logged in, the right part will change, as can be seen in figure 10.3 on the following page.

10.3. Classes

The most important classes will be shown in this section, and all methods of these classes will be described. However, the entire code will not be shown as it is very large. Particularly interested readers can look in the source code comments or read the product documentation, located on the CD-ROM attached to the report. There are 12 classes in the final product. The classes; Article, Comment, Layout, WebMsgBox and SessionManager are going to be the focus of this section.

Many of the classes in the application are a part of the designed model layer, and as such, are very similar. Most of their functions use SQL statements, and since the classes mirrors tuples in the database, they all have at least two constructors - one for retrieving information for the object, and one for inserting new tuples in the database. Some classes also have an empty constructor for creating an empty object. This constructor does not, unlike the others, have an SQL connection variable as it makes no connection.

The two database constructors work very similar in all classes. Therefore we will only describe the constructors of one of the classes, namely the Article class.

Inserting constructor This constructor is used whenever the system needs to create an object and input its data in a table in the database.

In code example 10.1 on page 83 the inserting constructor for the article class can be seen. It starts with a definition of the constructor and its parameters and in lines 3-7 the private instance of the variables of the class are being set the given parameters. The following lines are standard in a method that uses SQL-queries, explained earlier in



Figure 10.2.: The FHOPPEN article page.

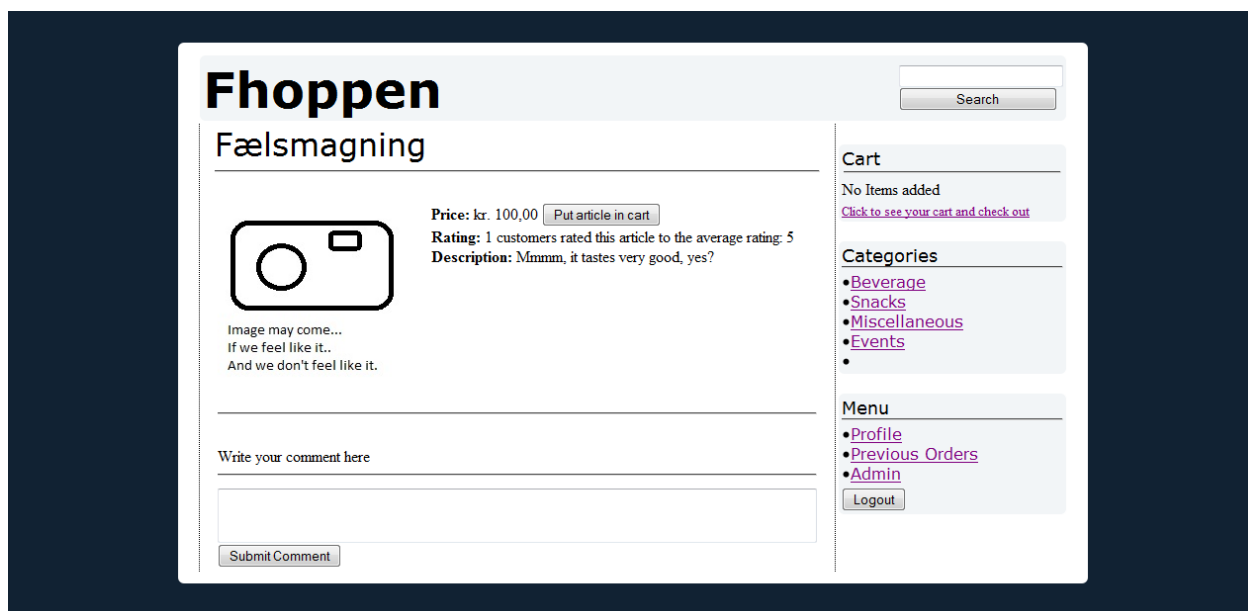


Figure 10.3.: The FHOPPEN article page when logged in.

Code Example 10.1 The inserting constructor of the Article Class.

```

1 public Article(string NewName, string NewDescription, int NewPrice, int NewCategoryId, int
   NewStockAmount)
2     {
3         this.Name = NewName;
4         this.Description = NewDescription;
5         this.Price = NewPrice;
6         this.CategoryId = NewCategoryId;
7         this.StockAmount = NewStockAmount;
8
9         var AddArticleConnection = new SqlConnection("server=INQUIZITZSERVER;User ID=sa;
   password=Big2;database=fhoppen");
10
11         try
12         {
13             AddArticleConnection.Open();
14             var query = "INSERT INTO Articles(Name, Price, Description, CategoryId, Stock)
   VALUES('" + this.Name + "', " + this.Price + ", '" + this.Description + "', "
   + this.CategoryId + ", " + this.StockAmount + ") SELECT SCOPE_IDENTITY() AS Id
   EXEC sp_fulltext_catalog 'FTCatalog', 'start_full' ";
15             var SCommand = new SqlCommand(query, AddArticleConnection);
16             var R = SCommand.ExecuteReader();
17
18             while (R.Read())
19             {
20                 this.ArticleId = Convert.ToInt32(R["Id"]);
21             }
22
23             AddArticleConnection.Close();
24         }
25         catch (Exception EX)
26         {
27             WebMsgBox.show(EX.Message);
28         }
29     }

```

section 5.3.3 on page 44. Line 14 contains the definition of the query. This query takes the values of the instance variables and inserts it to the database and then it gets the id of the newly inserted article from the database and assigns it to the articleId of the object.

Getting the id of the new object is not of major relevance in the article class, but it *could* be of some use. In other classes though, Purchase for instance, it is very relevant since the id is needed in the constructor to relate the object to the list of articles to be purchased. An example of this can also be seen in the product documentation.

Retrieving constructor This constructor is used whenever the system needs to retrieve data from the database and use it to create an object.

In code example 10.2 on the next page the retrieving constructor for the article class can be seen. First the parameters are given, while in line 3 the connection definition for the SQL can be seen. A new query is defined and executed on lines 7-10 selecting everything from the Articles table where the ArticleId parameter equals the ArticleId that was given as a parameter.

While the loop condition on line 11 is true, the current instance of the variables of the object are being set to equal those extracted from the database (line 13-18). The try block ends in line 19, and the connection closes in line 21. After execution the method has successfully build a new object with the data from the database, or else the catch block will show an exception in a web messagebox.

Code Example 10.2 The retrieving constructor of the Article Class.

```
1 public Article(int ArticleId)
2 {
3     var SC = new SqlConnection("server=INQUIZITZERVER;User ID=sa;password=Big2;database=
4         fhoppen");
5     try
6     {
7         SC.Open();
8         var getExistingArticle = "SELECT * FROM Articles WHERE Articles.Id = " +
9             ArticleId;
10        var SCommand = new SqlCommand(getExistingArticle, SC);
11        var R = SCommand.ExecuteReader();
12
13        while (R.Read())
14        {
15            this.ArticleId = ArticleId;
16            this.Name = R["Name"].ToString();
17            this.Description = R["Description"].ToString();
18            this.Price = (int)R["Price"];
19            this.CategoryId = (int)R["CategoryId"];
20            this.StockAmount = (int)R["Stock"];
21        }
22        SC.Close();
23    }
24    catch (Exception EX)
25    {
26        WebMsgBox.show(EX.Message);
27    }
28 }
```

10.3.1. Article

This class represents the articles to be sold in the webshop. The article class contains methods that have to do with the articles as well as methods facilitating comments and ratings, both major parts of article functionality. To see the implementation in detail, please refer to section 10.2 on page 80. The class methods that has to do with the article itself:

- *Constructors* - This class has three different constructors. The first constructor requires an article name, a description, a price, a category id and an amount. The constructor will create a new object from scratch using these parameters and uploads it to the database. The second constructor only requires the article id as a parameter and is used to retrieve article information from the database, if it exists, and use this information to create a new object. The third constructor is used to create a new article without assigning values to any of the properties.
- *GetPriceInKr* - This method returns the price as a string formatted in Danish Krones. Price is stored in the object as an integer in Danish øre.
- *UpdateArticleInfo* - This method gives the user the ability to change article information in the database. An administrator might use it, if there is a mistake in article description, or if the price needs to be changed.
- *UpdateRatings* - This method retrieves all the ratings that has been given to the article from the database and puts this information in the list of rating objects.
- *AverageRating* - This method reads the list of rating objects and calculates the average rating for the article.
- *UpdateComments* - This method reads all the comments attached to the article from the database and puts this information in the list of comment objects.

- *RemoveArticle* - This method removes the article from the database.
- *IsSafeToDeleteArticle* - This method checks if it is safe to delete the article. It checks if some users have ordered this article and if that is the case, then the method will return false. However, if nobody has purchased the article, it will return true, making it safe to delete.

10.3.2. Comment

Objects of the type *Comment* has to be attached to an article, as well as the customer or administrator who made it. Each object also hold a string, representing the text typed in by the customer or administrator, as well as the time and date of its creation. In the case of the comment being an answer, it will also contain an *answerTo* variable containing the id of the comment it answers. Methods used in the class are:

- *Constructors* - This class has three different constructors. The first constructor is used when a user or an administrator, wants to add a comment to an article and it requires a user id, a comment text and the id of the article as parameters. Note that the date of a comment is added automatically when a comment is inserted in the comment table in the database, and it is therefore not required as a parameter. The second constructor is used whenever an administrator wants to answer another customer's comment, and it requires an additional parameter, namely the *commentId* of the comment it answers. The third constructor is used whenever comments need to be retrieved from the database, for example when drawing comments on an article page.
- *UploadComment* - This method uploads a comment to the database. The method first checks if the comment is an answer to another comment. If this is not the case, it runs an SQL-query that does not utilize the *answerTo* variable. However, if the comment is an answer to a comment the method uses a different query that also updates the *answerTo* attribute in the database.

10.3.3. Layout

The *Layout* class is used to connect the model layer classes to the graphical interfaces. It does this through a number of draw functions, aswell as a number of functions that build *HtmlControls*¹. It uses many of web controls of the .NET framework class library to build the entire page in the memory as objects before adding it to the actual page.

The main layout of the page is build up of a number of tables². Some cells are used to create the impression that the main content page is placed above the background like a poster on a wall. Other cells are used to contain the content of the page. These cells are split up in two types; A single content cell that contains the main content of the page, like the article and category views, and the ones that stays the same on all pages, like the header. The *Layout* class keeps a pointer to the single content cell as the property "*ContentCell*".

The following is a list of all the methods in the class, followed by a detailed discription of some of the methods.

- *Constructor* - Creates a new *Layout* object and sets the page object. The page object is an object that, among other things, contains pointers to the server and session objects³ of the current webpage.

¹A sub class of the *Control* class making it possible to write HTML code in C#.

²HTML tables, not database tables.

³These are objects that provide access to methods and properties of the current server and session.

- *CreateHeader* - Adds the header information, but not the title⁴, needed by all the pages in the webshop.
- *CreateBody* - Adds the table and the contents that make up the main layout of the page.
- *DrawArticle* - Draws the given Article as well as the comments and rating associated with it, in the ContentCell.
- *DrawCategory* - Draws the given Category in a grid inside the ContentCell.
- *DrawCreateCustomer* - Draws an interface for creating a new Customer in the ContentCell.
- *DrawCategories* - Draws a menu containing all categories in the given Control.
- *DrawFrontPage* - Draws the frontpage in the ContentCell.
- *DrawUpdateCustomer* - Draws the interface to allow a customer to change his information.
- *DrawViewOrder* - Draws a given purchase in the given Control.
- *DrawShowCart* - Draws the current customer's cart in the ContentCell, if the customer is logged in.
- *DrawCheckout* - Draws a checkout in the ContentCell.
- *DrawSearchCategory* - Draws the Search Category in the ContentCell.
- *DrawCustomerOrders* - Draws a customer's previous purchases in the ContentCell.
- *BuildCategoryDropDownList* - Returns a DropDownList containing all the categories with their id, as their value.
- *BuildArticleDropDownList* - Returns a DropDownList containing all the articles with their id, as their value.
- *LoginButton_Click* - The Click-event that occurs when the login button is clicked.
- *SearchButton_Click* - The Click-event that occurs when the search button is clicked.
- *LogoutButton_Click* - The Click-event that occurs when the logout button is clicked.
- *SubmitCommentsButton_Click* - The Click-event that occurs when the "submit comment" button is clicked.
- *PutInCartButton_Click* - The Click-event that occurs when the "put in cart" button is clicked.
- *CheckoutButton_Click* - The Click-event that occurs when the checkout button is clicked.
- *ButtonCreateCustomer_Click* - The Click-event that occurs when "create new customer" button is clicked.
- *RateButton1_Click* - The Click-event that occurs when the "rate 1" button is clicked.
- *RateButton2_Click* - The Click-event that occurs when the "rate 2" button is clicked.
- *RateButton3_Click* - The Click-event that occurs when the "rate 3" button is clicked.
- *RateButton4_Click* - The Click-event that occurs when the "rate 4" button is clicked.
- *RateButton5_Click* - The Click-event that occurs when the "rate 5" button is clicked.
- *RemoveItemButton_Click* - The Click-event that occurs when the "remove item from cart" button is clicked.

⁴The title should be set within the .aspx file itself

- *PaymentButton_Click* - The Click-event that occurs when the payment button is clicked.
- *ButtonUpdateCustomer_Click* - The Click-event that occurs when the "update customer info" button is clicked.

Most of the methods can be divided into the following groups: Draw methods, build methods and click-events. The following will be a walkthrough of the basic concept of each group, along with code examples to provide a better understanding of the more complex parts of the methods.

10.3.3.1. Draw methods

The draw methods, which also includes the *CreateHeader* and *CreateBody*, work by using the *HtmlControl* classes provided by the .NET class library. They use these classes to build HTML controls that can be inserted into the actual HTML document, and returned to the user. This can be seen in code example 10.3, where a label is created and added to a div tag, <div>.

Code Example 10.3 The use of *HtmlGenericControl* and *Label*.

```
1 var HeadContainer = new HtmlGenericControl("div");
2 var HeadLine = new HtmlGenericControl("hr");
3 var HeadLabel = new Label();
4
5 HeadLine.Style.Add("Margin", "2px");
6 HeadLabel.Text = "Cart";
7 HeadLabel.Font.Name = "Verdana";
8 HeadLabel.Font.Size = FontUnit.Large;
9
10 HeadContainer.Controls.Add(HeadLabel);
11 HeadContainer.Controls.Add(HeadLine);
```

The example shows some different ways of using *HtmlControls*. One can either use the *HtmlGenericControl*, and specify what kind of HTML tag it should represent, or use one of the expanded *HtmlControl* classes representing a single HTML tag.⁵ Common for all *HtmlControls* are, among other things, the lists *Style* and *Controls*. The *Controls* list, is a list of the tags inside the tag represented by this control, and is used to add one control as a child of another. The *style* list is the mirror of the style attribute of a HTML tag, and uses Cascading Style Sheets⁶ (CSS) syntax to change the appearance of the tag.

The resulting HTML code can be found in code example 10.4.

Code Example 10.4 The result of the method from code example 10.3.

```
1 <div>
2   <hr style="Margin:2px;"></hr>
3   <span style="font-name:Verdana;font-size:large;">Cart</span>
4 </div>
```

10.3.3.2. Build Methods

The build methods of the *Layout* class builds *HtmlControls* and returns them. The given example is the *BuildCategoryDropDownList* method, which gets a list of the names and

⁵A complete list of the tags that can be represented by an *HtmlControl* without using the *HtmlGenericControl* can be found at <http://tiny.cc/cswebcontrols>

⁶More info about CSS, can be found at <http://www.w3.org/TR/CSS21/>

ids of all categories from the database, and inserts them into a DropDownList control before returning it. The method can be seen in code example 10.5. The GetCategories method, used in the example, returns a list of all categories in the database.

Code Example 10.5 The BuildCategoryDropDownList method.

```
1 public DropDownList BuildCategoryDropDownList ()
2 {
3     DropDownList ReturnDropDownList = new DropDownList ();
4     foreach (Category C in GetCategories ())
5     {
6         ReturnDropDownList.Items.Add (new ListItem (C.Name, C.Id.ToString ()));
7     }
8     return ReturnDropDownList;
9 }
```

10.3.3.3. Click-Events

The click-events in the Layout class are the functions called when a button on the web-page is clicked. They are a part of the event handler system of .NET which will not be described in detail in this report. Instead, code example 10.6 shows how to subscribe to a new event, meaning the client will be notified whenever the event occurs. Most of this code can be created automatically using Microsoft® Visual Studio.

Code Example 10.6 Subscribing to a clickevent.

```
1 [...]
2 SearchButton.Click += new EventHandler (SearchButton_Click);
3 [...]
4
5 protected void SearchButton_Click (object sender, EventArgs e)
6 {
7     page.Response.Redirect ("/Search.aspx?SearchString=" + SearchTextBox.Text);
8 }
```

The example is the click event of the searchbutton which redirects the user to the search page.

10.3.4. WebMsgBox

This static class is far smaller than the other classes, and is used only for providing feedback to the user. ASP.NET has no build-in class for presenting the user with a message box. Therefore this class was created to handle this problem. It has no advanced functions, as it used the JavaScript Alert function. It is only capable of displaying a text to the user. In code example 10.7 on the facing page the code for the Show method, the only method of the class, can be seen. It adds a JavaScript alert box to the current HttpContext. It is an adaption of the Visual Basic implementation of a message box found on [11].

10.3.5. SessionManager

The SessionManager handles all login and logout methods, and is also the way to check, whether the page viewer is a registered user or not, as some functionalities are restricted to registered users. The SessionManager also contains a function to get the current user's id.

Code Example 10.7 The show method of the WebMsgBox class.

```

1 [...]
2 SearchButton.Click += new EventHandler(SearchButton_Click);
3 [...]
4
5 protected void SearchButton_Click(object sender, EventArgs e)
6 {
7     page.Response.Redirect("/Search.aspx?SearchString=" + SearchTextBox.Text);
8 }

```

By using sessions, as the SessionManager does, it is possible to make an anonymous user register, and then let him checkout with the articles, he added before registration. This functionality could not be implementable without the use of sessions.

In code example 10.8 on the next page, the login method is shown, and all SessionManager methods are described in a list below:

- *Login* - This method tries to log an existing user in and validates password and username. When the user is logged in, he has 20 minutes, with no activity, before the session terminates.
- *Logout* - This method logs out an existing user.
- *IsLoggedIn* - This method checks, if the user is already logged in.
- *GetActiveCustomer* - This method returns the id of the logged in user of the current session.
- *Isadministrator* - This method checks, if the user is an administrator.

Since this method, like many others, uses standard SQL connection code, the SQL parts will not be described here. More information about the standard SQL connection code can be read in the explanation of the code shown in code example 5.12 on page 44.

On line 3, a password entered by the user in the textbox is encrypted for safety reasons and saved in the UserPassword variable. The query on line 8 selects the users password and id by searching for tuples, where the email attribute contains the email entered in the textbox. On line 15, the password from the database is taken out and put in the variable Passwordtest. As database entries of passwords are already encrypted, encryption is not needed. On line 17, an if-statement compares the password, the user entered in the textbox, and the password recovered from the database. If those two passwords are the same, the UserId is stored in the session array at the key UserId on line 19.

The password could also be tested in the SQL query instead. The reason for not doing this is that the user could be advised that the entered password is wrong, if the email is entered correctly. This feature, however, is not yet implemented.

10.4. Encryption

Every time a customer logs in, this is done by a combination of the email and password. These values are saved in the database to check, if the user enters his credentials correctly. If anyone should get access to the database, it should not be possible to read the password directly. Therefore, it has been chosen by the development team to use an md5 encryption to secure the password in the database.

Code Example 10.8 The Login function of the SessionManager class.

```
1 public void Login(string UserEmail, string UserPassword)
2     {
3         UserPassword = Encryption.md5(UserPassword);
4         var SC = new SqlConnection("server=INQUIZITZERVER;User ID=sa;password=Big2;database=
5             fhoppen");
6         try
7         {
8             SC.Open();
9             var SearchLogin = "SELECT Password,Id FROM Users WHERE Email='" + UserEmail + "'"
10                ;
11             var SCommand = new SqlCommand(SearchLogin, SC);
12             var R = SCommand.ExecuteReader();
13             while (R.Read())
14             {
15                 string Passwordtest = R["Password"].ToString();
16                 if (Passwordtest == UserPassword)
17                 {
18                     Session["UserId"] = (int)R["Id"];
19                     WebMsgBox.show("The user with the ID " + (int)R["Id"] + " has logged in")
20                         ;
21                 }
22             }
23             SC.Close();
24         }
25         catch (Exception EX)
26         {
27             WebMsgBox.show("we ran out of coffe, so the sql showed this message: " + EX.
28                 Message);
29         }
30     }
```

An MD5-encryption is a so-called one-way encryption which means that once encrypted, it cannot be decrypted again⁷. The password is therefore saved in the database in its encrypted form, and no one but the original user really knows what the password is. The code used for this project for the MD5-encryption can be seen in figure 10.9 on the next page.⁸

As the MD5-encryption works on binary data, the first operation is to convert the password sting into a byte array. Then, by using the foreach loop, the encrypted bytes are converted to hexadecimal, which is then concatenated and returned.

10.5. Database

The database consists of nine different tables: Articles, Cart, Categories, Comments, Customers, OrderedArticles, PostalCodes, Purchases and Ratings. The tables are the same as described in section 8.5.4 on page 74.

The database tables and their relationships can be seen in figure 10.4 on page 92.

Note that the actual database architecture differs from the model component classes. This is because the model component classes contain a collection of objects, while the database model tables contain the id of the parent object as a foreign key.

All table identification attributes are called id, and a foreign key is denominated by the name of the table, it refers to, followed by "Id", for example ArticleId. The Comment

⁷It can be broken, but it is impossible to find out what the actual password was before it was encrypted.

⁸As encryption is not a part of this semester, the code has been copied from <http://tiny.cc/md5>.

Code Example 10.9 The encryption class, which is responsible for password encryption in the system.

```

1 namespace fhoppen
2 {
3     /// <summary>
4     /// Static class for encrypting the password
5     /// </summary>
6     static public class Encryption
7     {
8         /// <summary>
9         /// Encrypts the given string using a oneway MD5 encryption
10        /// </summary>
11        /// <param name="sPassword">the string to encrypt</param>
12        /// <returns>the encrypted password</returns>
13        public static string md5(string sPassword)
14        {
15            System.Security.Cryptography.MD5CryptoServiceProvider x = new System.Security.
16                Cryptography.MD5CryptoServiceProvider();
17            byte[] bs = System.Text.Encoding.UTF8.GetBytes(sPassword);
18            bs = x.ComputeHash(bs);
19            System.Text.StringBuilder s = new System.Text.StringBuilder();
20            foreach (byte b in bs)
21            {
22                s.Append(b.ToString("x2").ToLower());
23            }
24            return s.ToString();
25        }
26    }

```

table differs from this, as the AnswerTo attribute is a reference to the id attribute of its own table. If a comment has an AnswerTo attribute set, it is an admin answer to a user comment. The AnswerTo attribute then refers to the id of user comment.

10.5.1. Applied SQL

The SQL theory in section 5.3 on page 38 has been applied to the program. So far only the basics of creating a table and altering the contents of it has been covered. This is primarily used when setting up the database, and not so much during the general usage of it. Using an SQL-database generally boils down to selecting data from a specified position which satisfies some predicate. An example can be seen in code example 10.10 on the next page, where the articleId and the amount of the article, a specific user wants to buy, is selected from the cart table.

However, only viewing the database entries is not always sufficient. For example, it could be useful to insert a new tuple into the database, which could be done as seen in code example 10.11 on page 93, where a new article and the amount of the item, the user wants to purchase, is added to a users cart.

It might also be necessary to update an existing tuple. This can be done as seen in code example 10.12 on page 93, where the amount of articles, the user wants to purchase is updated.

Finally, the ability to delete a tuple from the table has been implemented as seen in code example 10.13 on page 93, where an article is removed from the users cart.

Only one SQL statement uses other SQL functions than *select*, *insert into*, *update* and *delete*. On the front page, the two highest rated articles and the two newest articles are shown. The SQL statement that selects the two highest rated articles can be seen in code example 10.14 on page 93.

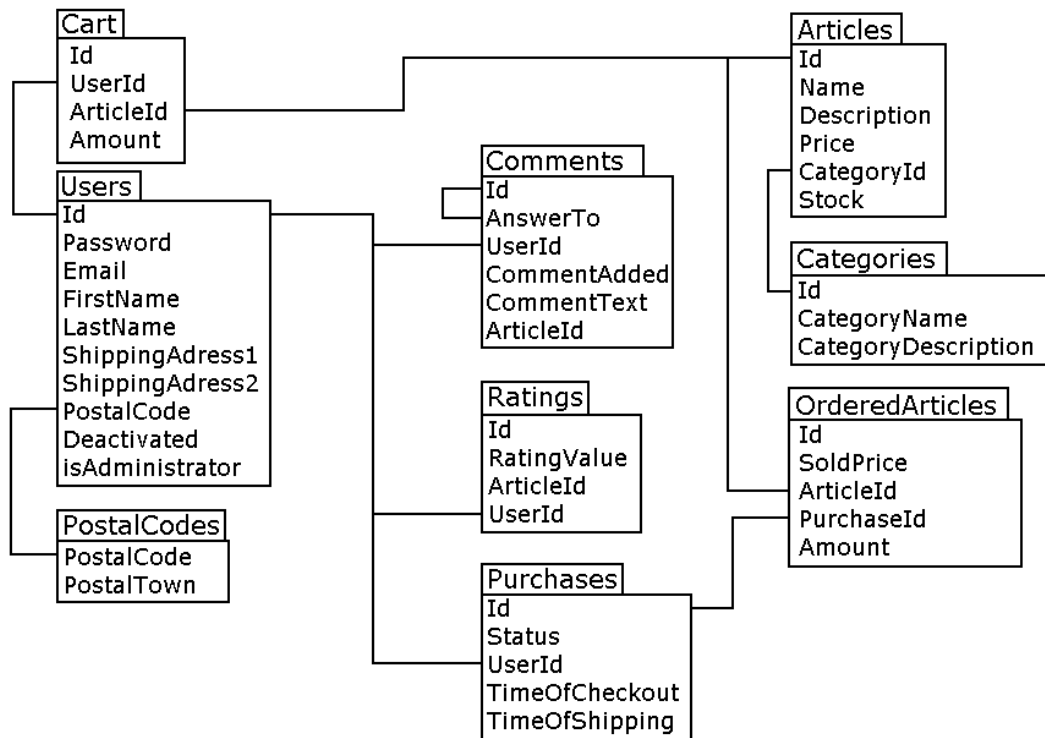


Figure 10.4.: The database tables and their relations.

Code Example 10.10 A select statement with a restricting statement.

```
1 SELECT ArticleId, Amount
2 FROM Cart
3 WHERE UserId = " + UserId + "
4 ORDER BY ArticleId ASC
```

10.6. Architecture

The agreed client-server architecture (which is shown in figure 8.1 on page 67 and described in section 8.4.1 on page 67) has been followed as much as possible, as the page is located on a server, and the user is able to view the page only by having a web browser, as it was demanded. The actual server-client connections are handled by ASP.NET, and as such only the layers inside the server component have actually been developed.

The component architecture of the server component is as shown in figure 10.5 on the facing page. The system contains a general user interface, which is connected to all classes by the layout class. All classes have direct access to the database.

The project group tried to follow the initial architecture design as much as possible, however, some of the classes that were present in the initial design were removed during the programming phase.

The Admin class was removed since methods that were needed for administration could not have been placed in the class. Instead, all of the administrator methods were placed in the classes where they belong, for instance, the article class contains the RemoveArticle method which is only accessible by the administrators. The administrator methods does not contain the usertype check which is needed to decide if the user is able to use the method or not, instead, this check was implemented by the introduction of the administrator interface which is only visible for administrators.

Code Example 10.11 An insert into statement.

```

1 INSERT INTO Cart
2 (ArticleId, UserId, Amount)
3 VALUES (" + Article + ", " + this.UserId + ", " + Amount + ")

```

Code Example 10.12 An SQL update statement.

```

1 Update Cart
2 SET Amount = " + this.ArticleAmount[newArticleLocation] + "
3 WHERE articleId = " + this.ListOfArticles[newArticleLocation].ArticleId + "
4 AND userId = " + UserId

```

Code Example 10.13 An SQL delete statement.

```

1 DELETE FROM Cart
2 WHERE ArticleId = ' " + Article + "'
3 AND UserId = ' " + this.UserId + "'

```

Code Example 10.14 The group by and join statements.

```

1 SELECT TOP " + showArticleAmount + " Articles.Id,
2 AVG(ratingValue) AS rv
3 FROM Articles
4 LEFT JOIN Ratings
5 ON Articles.Id = Ratings.ArticleId
6 GROUP BY Articles.Id
7 ORDER BY rv DESC

```

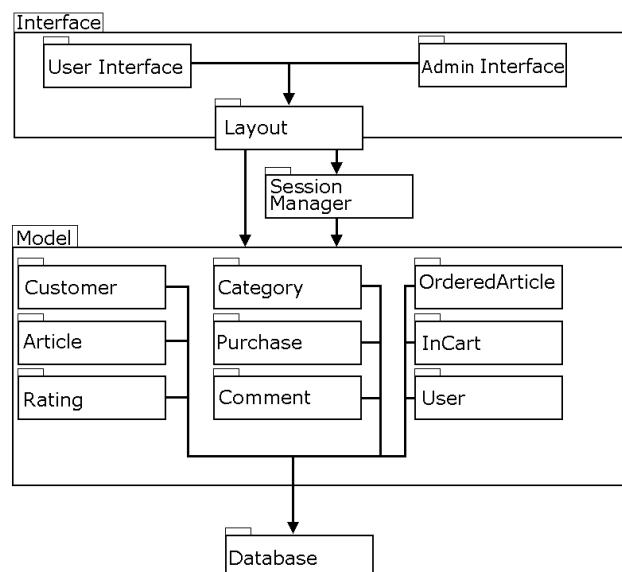


Figure 10.5.: The architecture of the system.

Part IV

Conclusion

Perspective

This chapter will explain, in which parts of the system, both the design and final implementation, the project group finds that changes are needed. First, the modifications needed in the final product will be discussed by describing the errors, and then suggesting a better solution. Also, the implemented architecture of the system will be discussed, and suggestions to improvements will be given. Afterwards, the general work period will be described, also with suggestions to improvements for later projects.

11.1. The FHOPPEN webshop

After the final build was made, the system was tested, and some needed modifications were discovered:

Login

If a user tries to log in with a wrong username or password, the system returns him to the frontpage with no feedback. Instead, an error message "Invalid username and/or password" should be presented to the user.

Checkout

After accepting an order, a customer can see the articles in the order and pay for it. Here, he is asked to check if his shipping address is correct, and if not, change it. However, the only place he can do this, is from his profile page. From there, he is not able to return to the accepted order and pay for it anymore. This is because the contents of the cart is moved to a purchase, when the customer accepts his order. He will, however be able to see the order in his previous orders, but the status will be null and unchangeable.

This can be prevented by implementing the profile information into the "Pay for Order" page, so the customer does not have to leave this page, and thereby loose access to the order. This, however, does not fix the problem if the customer leaves the "Pay for Order"-page for another reason.

Another way to prevent this is to empty the user's cart after the customer has paid for it. This could be done by recoding the Purchase constructor, which takes in two parameters, so it will create the Purchase object being added to the database as a placed order, *after* the customer has paid for the order. Also, the cart object will not be deleted, if the customer decides to add more articles to the cart after checkout, but before paying.

Update Profile

If a user wants to update his profile, and edits some information, but not the password, the password would still be updated, and therefore set to an empty string. This should, of course, not happen. Instead, the function should just skip updating the password, if the field is empty.

Update Indexing

When editing the article table, either by editing or deleting a tuple, the index of the article table, which the search function uses, is not updated. This means that changed articles will still show up in their original form instead of the updated. Also, deleted articles will still show in the search result, at least until a new article is created (which is where the index is updated).

Moderate Comment

The moderate comment function which it was decided to use in the design phase, has not been implemented. The reason herefor is that it was deemed at least temporarily unnecessary, as the administrator has the possibility to delete comments, if needed.

11.1.1. Other Improvements

The above mentioned modifications are errors in the implementation of the design. Apart from that, the system has basis for extension, too. This mostly pertains to the payment implementation, but new customer features could also improve the system.

For instance, when adding an article to one's cart, one has to click on the "Add article to cart" button. A better implementation would be to put a number field next to the button, so one could declare, how many items to add to the cart.

It has been the plan that a third party module, Fri Betaling[12], should handle the payment part of the system, and the security of it. However, it was decided to skip this, because the function would require a financial investment, which the project group decided, was unnecessary. Instead a payment simulation feature was added, which updates the order status to "paid", and returns a "payment OK" message.

Another problem with the current system is the lack of communication between the shop and the user. There are a number of places this would be a great benefit. For example, a welcome email can be sent to a new user, or an email containing information when a purchase is completed.

It should also be noted that the system in its current state only handles the relation between the customer and the shop. A complete system would require an extensive overhaul of the inventory system, mainly so that data could be sent from the system to relevant packaging firms, wholesalers and banks.

A part of this project has been implementing unit testing. Due to time limits only a single class is actually tested, in order to show the principle.

11.2. Architecture

It was discovered that some classes in the model component were unnecessary - an example of that would be the Admin class, which was removed from the final product.

Some classes, such as the SQL class, were present in the original design schema which can be seen on figure 8.4.1 on page 67, but were not implemented in the final version of the product, since they were deemed unnecessary due to lack of experience with more advanced systems. This is one of the design flaws of the system, since the presence of an SQL class would solve the connection pool problem, mentioned in section 8.4.1 on page 67.

The system architecture would also be more comprehensible if the SQL class was present, as all SQL connections would be handled by one class, instead of being present in every method that uses SQL connections. This would, first of all, make it easier for future programmers to understand the system, but also make it easier to make changes

to the database connection, as it would now be time-consuming to port the system to another database.

Another design flaw in the system is that no convention for class constructors was decided. However, it was discovered that the constructors of the classes all fall into one of these three types:

- **Empty constructor** - an empty constructor, which creates an empty object.
- **Constructor with one parameter** - A constructor, which creates an object with requested data from the database.
- **Constructor with several parameters** - A constructor, which creates an object by the data passed along with the function, and adds it to the database.

11.3. Knowledge Gained

The members of the project group have gained knowledge in the following areas:

In order to create a webshop, the group learned how to use databases, including SQL, data storage and relations. The group has also learned that creating a webshop from scratch is not as simple, as it seems. Through the OOAD-course, the group learned how to:

- Define a system by using the FACTOR-criterion, and analyzing the problem domain and application domain.
- Analyze and design the system by using an analysis document and design document.
- Adjust criteria to the system, by returning to the problem and application domain when the group discovered a flaw in the system.
- Design the system by using methods from OOAD.

11.4. Work Process

During the project, several different methods have been used to make the group members feel responsible for the development of the project, some with greater success than others. It was decided against to use any kind of "punishment" for unfulfilled expectations.

Task Assignment

Each team member were assigned a task and a deadline, and was then expected to have the task done by the deadline. This was done to give the members the freedom to work as they wanted. It has some drawbacks, since it does not encourage people to do more work than agreed upon, which could have been achieved without deadlines. Combine this with the fact that some tasks were not finished at the deadline, and one can see, how the project timetable collapse.

Also, the assigned tasks were bigger than originally agreed upon. This made the agile development process hard to achieve. Furthermore, the iterative work flow of the report did not come into full effect until the last part of the project.

Starting Time and Meetings

The group decided to make the project work phase an eight hour shift. As some group members were against working after 4. pm, the meeting time was scheduled at 8. am. Some members, however, had a problem living up to this, and the meeting time was rescheduled to 8.30 am instead. This did not resolve the problem, and as it was decided to conduct a meeting as soon as all group members were present, the project progress suffered.

To resolve the problem, a consequence could be considered. Previous experience shows that economic punishment will not work. Instead, it should be considered to ignore the fact of missing members. Also, it could be considered to move the meeting to a later time. By doing so, it would also be possible for group members to start with their current task when they show up.

Agile Development

In the project, the Scrum agile development process was tried. It is, however, clear that many of the agile concepts, apart from the iterative work flow, are not intended, nor required or effective, for a study project.

For instance, the sprints were not designed to be used in a project as small as a study project, mainly because the development time is too small. A normal sprint would be around a month long and consist of several major updates to the system. In this project, the first sprint was only a week long, but still amounted to a complete working system. The second sprint resembled a "minisprint", consisting only of minor fixes to the system and an implementation of the administrator interface. This was not the most iterative process, but it still amounted to a working system within the time allotted.

The only concepts, the group would use again are the daily Scrum meeting and the iterative work flow.

Problem Solving

The group originally intended to use the hot chair concept to relief tension in the group. The concept lets group members comment on irritating moments in a civilized way. This was not used as intended, as a hot chair meeting should be held at least every fortnight, and not just once in a while as was the case.

Generally, it has been made clear to the project group that internal problems should be dealt with quickly, instead of letting them pile up. This can be done in several different ways. Mostly it should be agreed upon in the group that every member needs to accept criticism in a good manner, instead of acting defensively when his opinions is challenged.

Conclusion

During this project the group has developed a working prototype of a webshop system. It has been developed according to the demands set forth by the F-Klub and has been made with functionality as a major focus area. The design of the program was done in part according to those demands, but also according to an extensive analysis of the problem at hand, including the future usage of the system.

The development process was not problem free. The objective approach to the design phase was utilized relatively easy, but the actual objective development caused problems. This was partly due to the previous development experience of the group, but also because the system was bigger than anything the group had tried before, and therefore harder to get an overview of. This meant that several key-components of the system had to be moved, rewritten or scrapped and re-designed during the initial part of the development phase.

To create a webshop that is easy to manage, all functions must be easily accessible and all interface elements must be easily understandable. This allows for IT-inexperienced individuals to interact with the system. Also, the system should be compact and not filled with functions not needed for day-to-day work so as to not create unnecessary confusion.

The articles in the developed system is maintained through an administrator interface, that allows utilizing the articles part of the database, without having any knowledge of how to operate a database. The customers can see the available articles through the system, including any data about the article that might be useful for the customer. All of this makes the administration of articles easy for the administrator and browsing and purchasing articles easy for the customer.

The commenting and rating of the articles lets the customers give their opinion about the wares featured by the shop. The administrators can view these comments and answer to them, to create the possibility of conversation between the customers and administrators. Also, the ratings let the shopowners know exactly which items are, not only being sold, but liked by the customers. This can make it easier to choose new articles for the shop.

BIBLIOGRAPHY

- [1] Mathiassen, L., Munk-Madsen, A., Nielsen, P. A., and Stage, J. (2000) Object Oriented Analysis & Design, Marko, . 4, 10, 11, 16, 17, 18, 20, 24
- [2] Definition of "shop" <http://wordnetweb.princeton.edu/perl/webwn?s=shop> (2006). 8
- [3] MSDN Try-catch description [http://msdn.microsoft.com/en-us/library/0yd65esw\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/0yd65esw(VS.80).aspx). 26
- [4] World wide web consortium <http://www.w3.org/TR/html401/>. 30
- [5] Silberschatz, A., Korth, H. F., and Susarshan, S. (2006) Database System Concepts, McGraw-Hill, . 36, 40, 42
- [6] Hayman, R. Full text search in sql <http://www.geekzilla.co.uk/View0E316ED1-7A99-42FD-B80D-272C15C98027.htm> (2006). 43
- [7] Schwaber, K. and Beedle, M. (2008) Agile Software Development with SCRUM, Pearson Education, . 45, 46
- [8] Bendtsen, B. Bogføringsloven - kapitel 5 - paragraf 10 stk. 1. <https://www.retsinformation.dk/Forms/R0710.aspx?id=27298> Juni 2006. 56, 60, 72
- [9] Salomon, I.-L. <https://blog.itu.dk/EBT1-E2009/files/2009/09/ilssystemudvikling.ppt> December 2008. 58
- [10] W3Counter Global web stats <http://www.w3counter.com/globalstats.php> November 2009. 80
- [11] Asp.net message box <http://www.freevbcode.com/ShowCode.Asp?ID=5918> (2009). 88
- [12] Fribetaling <http://www.fribetaling.dk/> (2009). 96

Part V

Appendix

A

User Interface Suggestions

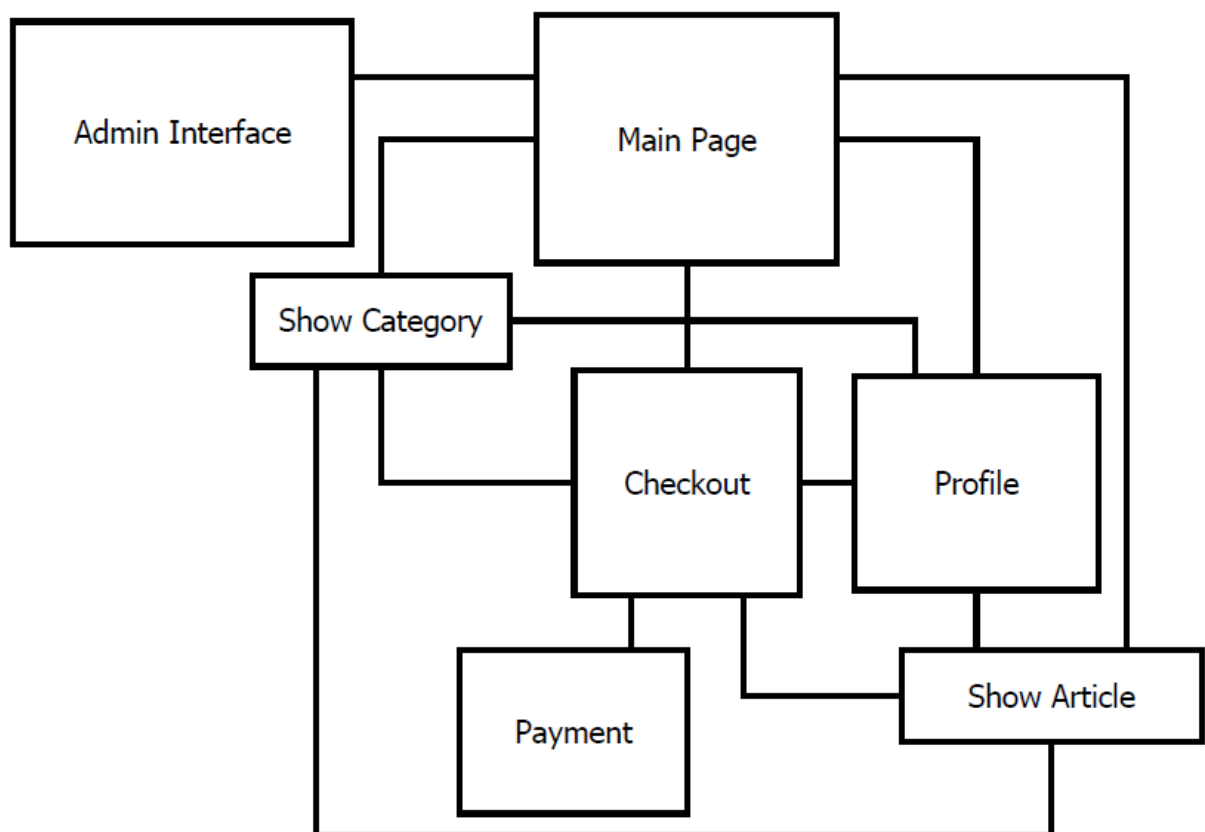


Figure A.1.: The connection of the pages in the system.

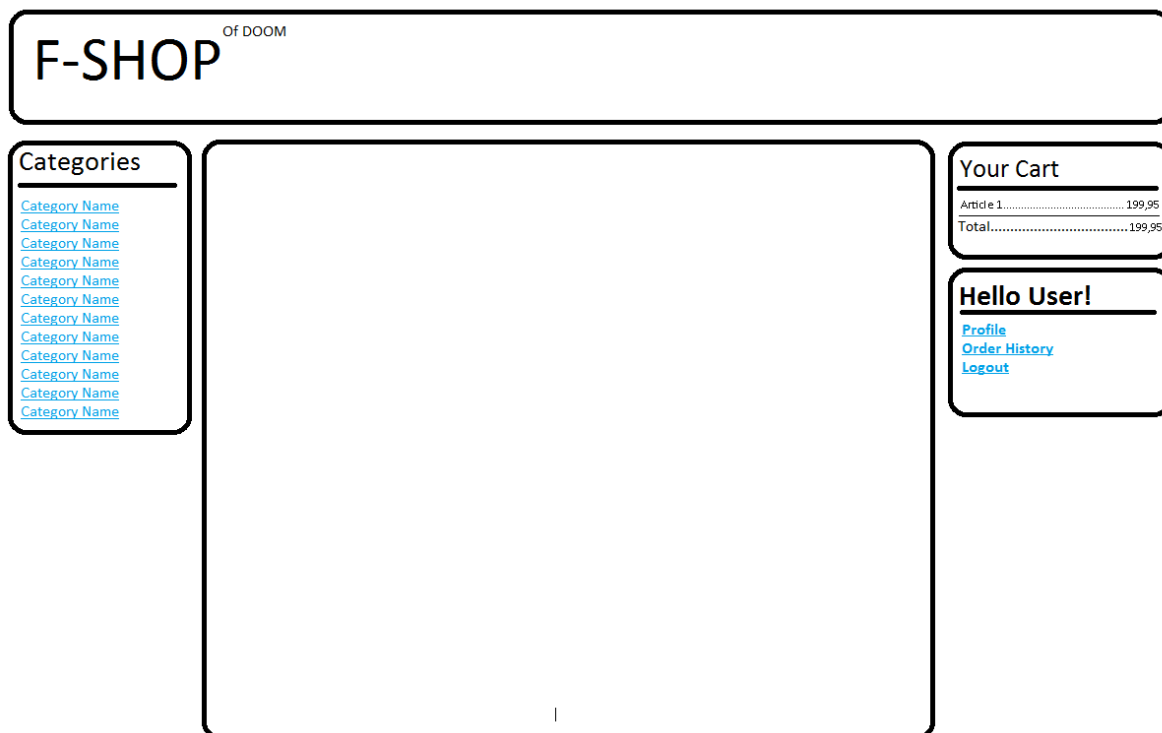


Figure A.2.: The view of the "standard" elements in the webpage - in the top, a banner. To the left, a category menu, and to the right, the rest of the menu. In the blank area in the middle, the main content is put.

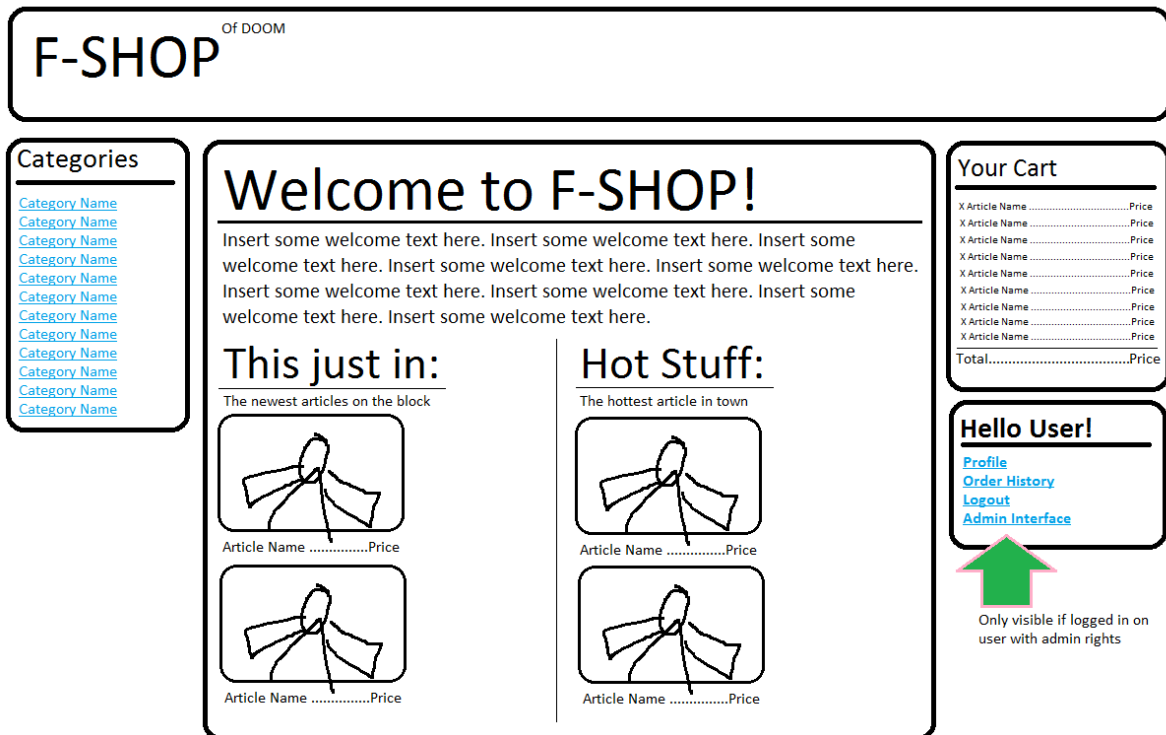


Figure A.3.: The look of the front page, when the user is logged in.

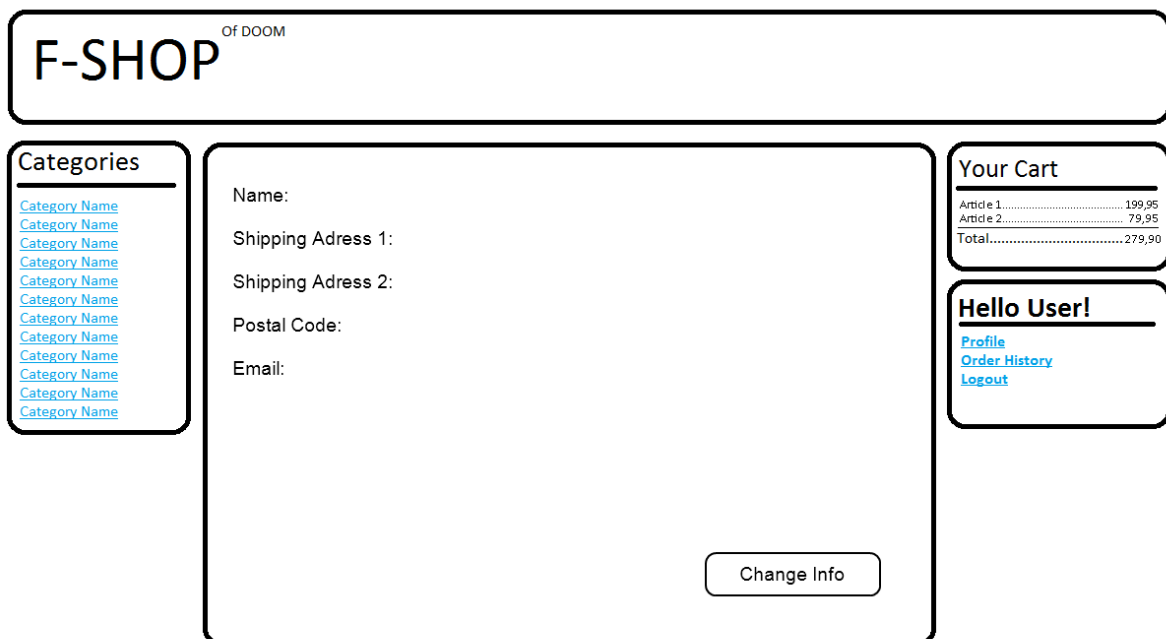


Figure A.4.: The profile information page. The user can see and edit his profile here.

OF DOOM

F-SHOP

Categories


[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)

Welcome to F-SHOP!


Insert some welcome text here. Insert some welcome text here. Insert some welcome text here. Insert some welcome text here. Insert some welcome text here. Insert some welcome text here. Insert some welcome text here. Insert some welcome text here.

This just in:

The newest articles on the block




Article NamePrice



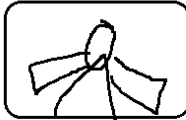
Article NamePrice

Hot Stuff:

The hottest article in town



Article NamePrice



Article NamePrice

Your Cart

X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
X Article NamePrice
Total.....Price

Login

Username:
Password:

[Forgotten password?](#) | [Register](#)

Figure A.5.: The look of the front page, when the user is logged out.

OF DOOM

F-SHOP

Categories

[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)
[Category Name](#)

Your Cart

Your Cart

Hello User!

[Profile](#)
[Order History](#)
[Logout](#)

Article

Amount

Price

Article 1..... 1 199,95

Article 2..... 1 79,95

Total 279,90

CHECKOUT

Figure A.6.: The look of the payment page of the site.

106

FHOPPEN

B

SW3 Project Proposal

SW3 Project

Autumn 2009

1. F-Shop Overview

The topic of the semester project is to develop and test a web-based shopping system **F-Shop** for the F-club (the F-Club is planning to extend its services and offers). The basic requirement is to build a system that supports the daily operation of F-Club; it is expected that the system could be extended to support other shops in general.

The F-club will sell the following articles: a variety of food and drinks, office material (e.g. pens, writing pads or folders), the Aalborg University's souvenirs (e.g. shirts, caps or buttons), and a selection of books, mainly the books recommended by the lecturers.

The system needs to support three different types of users: anonymous users, customers, and administrators.

- Anonymous users can browse through available articles and put the desired ones into a shopping cart. For a final order, however, a login is required. Any anonymous user can sign up and create an account.
- The customer role is granted to a user after login, and additional functionality becomes available. Apart from completing the order, customers can create their personal profiles, see purchase history, create wish lists, and comment and rate articles.
- An administrator is a special system user which has a privileged access. They can add/remove/edit articles of the shop. Also they can remove inappropriate comments and ban users from the system.

2. Requirements

2.1 Core requirements

The following are core requirements and must be implemented.

1. **Catalog browsing.** The homepage of the F-Shop should provide a catalog of available articles. The user can select products by category, popularity (most purchased), ratings, and etc. The products (in category) can be sorted by name, price, date, availability, etc. Every article in the catalog can be clicked and a more detailed page is provided by the system.
2. **Login.** The users can create an account where all their purchases will be tracked. As previously stated, only logged in users can complete the purchase. However, it is not required to login in order to browse the available articles. The login system must be secure and accessible for all user roles. After login, the user is granted either the customer or the administrator role.
3. **Shopping cart.** Any available article can be put into the shopping cart. Anonymous users may fill the shopping cart as well as logged in users do. If an anonymous user puts something into her shopping cart and logs in afterwards, the articles should be added to the shopping cart the user created in previous sessions. When users change the quantity of any article in the shopping cart, the system should recalculate the price.
4. **Payment.** For this application we assume that users have to transfer the money to a specified account from their credit card or bank account. Encryption is required when users send credit card information to ensure that the sensitive information is secure.
5. **Personal user profile.** After signing up in the system, the user can create her personal account by entering personal information (name, surname, birth date, shipping address, etc). A wish list feature has to be provided as well. The profile can be edited any time.
6. **Search.** The system needs to provide advanced search feature for the users. Articles can be searched in any fields (or selected fields) and wildcard characters must be supported.
7. **Comments.** The system needs to provide the possibility for the users to express their opinion or ask questions about the articles. That is, under each available article the user can leave a comment.
8. **Rating.** The system should provide the means for the users to rate the products. The user can rate only the products she has purchased.

2.2 Optional requirements

Students are free to choose from the following requirements to implement. Students are also welcome to come up with their own proposals on the functional requirements of the F-Shop.

1. **Recommending subsystem.** Since the system is tracking user purchases and ratings, sort of recommending subsystem can be implemented. The students are free to try (or invent) various recommending algorithms here.
2. **Inventory management subsystem.** A simple implementation is to send an e-mail notification to administrators when only five items of any product is left. More complicated implementation is to send an email 5 days before the stock is going to run out.
3. **Award-point subsystem.** The customers can earn some points from how much they pay. The points can be accumulated and can be used for payment by some conversion rule.
4. **New article request subsystem.** The customers can request new articles that are not in stock. When the articles are in stock, the customers will be notified by emails.
5. **Flee market subsystem:** The customers post advertisements for sale or exchange their own stuff in a market board.

2.3 Basic interface requirements

User interface has to be a web-based and accessible from a browser (browser compatibility is not required).

Different interfaces are required for anonymous visitors, customers and administrators.

Every article needs to be shown in a small preview window in browsing or search result pages. When users click on a particular article, the article's detail page shows all information available about the item, e.g., a name, a description, a price, a picture, and comments and ratings, if any.

The homepage of the F-Shop shows the most popular articles based on user purchases and ratings.

Please note that usability is NOT a primary concern of this project. In other words, students are not expected to create fancy GUIs and web designs. Standard HTML page elements like plain text fields and radio buttons are sufficient to accomplish the user interface requirements