# Aalborg University
**Department of Computer Science**

**Title:**
StudyLife - User, Friend, Group, Master Layout and Search Components

**Topic:**
Application Development

**Project Period:**
February 1st - May 28th, 2010

**Project Group:** s601a
Jesper Kjeldgaard
Jens-Kristian Nielsen
Simon Stubben

**Supervisor:**
Laurynas Siksnys

**Copies:** 5

**Pages in report:** 76

**Appendices:** 2

**CD-ROM:** 1

**Total pages:** 103

## Abstract:

The topic of this semester is *Application Development.* The report documents the development of a social web application, StudyLife, made in conjunction with multiple project-groups, and covers all the parts of the development process. It describes how the master layout and the specific components; user, friend, group and search, are designed and implemented on the basis of requirements derived from an analysis.

The main goal was to create a web application for social networking and collaboration at a University, as well as gaining experience and knowledge regarding development of a large software application from a set of requirements. The process of the project has been focused, such that both the development process is thoroughly explained and a reflection of this is carried out.

Tests were performed, both on the specific components in the integration process and on the whole StudyLife system. The tests revealed errors that were corrected. After this the components were correctly integrated and the main functionalities worked as intended. The requirements for this project were fulfilled as well, and the goals of the semester was met.

# Preface

ï»¿This report is the result of Software Engineering group s601a's 6th semester project at the Department of Computer Science, Aalborg University. The report documents the project, proceeding in the period from 1st of February until 28th of May 2010.

The topic of this semester project was *Application Development* and the goal was to gain knowledge of application development through collaborating with other groups in a multi-project.

Cites and references to sources will be denoted by square brackets. Abbreviations will be represented in their extended form the first time they appear. When "we/our" is mentioned, it refers to the authors of the report. When referred to he in the report "he/she" is meant. When referred to "the system", that system is StudyLife, the product of this multi-project. Implementation specific terms are written in *italic*. Classes, methods and class properties are written in `monospace`.

The readers of this report are expected to have a basic understanding of web development and/or have completed the 5th semester of Software Engineering or equivalent. The code examples in the report are not expected to work out of context and the appendices are located at the end of the report. The source code for the software project, the report in PDF, images are included on the attached CD-ROM.

### Report Structure

**Chapter 1**
Introduces the report and the StudyLife system. An overview of the process is gone through as well.

**Chapter 2**
Goes through relevant elements of the contract developed in the first month.

**Chapter 3**
Describes the development method and a significant tool used in development process.

**Chapter 4**
Includes the relevant theory for the development of StudyLife.

**Chapter 5**
Describes the analysis of the system, and eventually specifies the requirements.

**Chapter 6**
The design of the system is documented in Chapter 6.

**Chapter 7**
Describes how StudyLife was implemented.

**Chapter 8**
Describes the different test methods performed to evaluate the system.

**Chapter 9**
Goes through the development process elaborating the pros and cons of the chosen development method.

**Chapter 10**
Finally Chapter 10 concludes the report and suggests further developments for StudyLife.

Jens-Kristian Nielsen

Jesper Kjeldgaard

Simon Stubben

# Contents

# Chapter 1
# Introduction

Over the past decade the web has become an increasingly social place to be. With communities like MySpace, Twitter and Facebook, a large part of the earth's population are connected with other persons; they communicate and share information over the Internet.

Solely in Denmark nearly half of the population uses Facebook [fac10]. It is widespread and widely used. Newspapers write articles about Facebook groups with some specific goal, companies create policies to exclude social web applications for not taking up all the time of the employee, and finally you have just accepted an invitation for an event the coming weekend.

Social web applications are popular and they are here to stay. Can they however, be included in a university setting for collaboration and networking?

Throughout this project, there has been worked with the creation of a social web application called StudyLife. The problem domain of the project has been collaboration between people affiliated with a university. The development of StudyLife has been done by looking at relevant theory, analyzing application requirements, designing based on the requirements, and finally implementing the chosen designs.

After having implemented, various tests have been performed to ensure reliability and correctness of the functionality, and the whole process of developing an application of this size have been reflected upon.

In the end, all parts of the project have been summed up and concluded.

## 1.1 Project Method

This project was carried out as a multi-project, which means that multiple project groups collaborated on developing a single software application. The application was developed by four different project groups with a total of 13 students. The individual project groups wrote reports based on the software application with focus on the components they implemented.

During the first month of the project no work was delegated to the individual groups. The goal of this period was to produce a document containing common analysis and design choices. This document will be referred to as *the contract* and it can be found in Appendix A. The contract documents, among other things, an architecture, which allows division of the application into components that naturally could be delegated to the groups by the end of the first month.

For the rest of the project period, the individual groups developed their delegated components and had a representative to meet at weekly meetings to discuss project status and possible dependencies in-between project-groups and components. At every other status meeting the components were integrated. Furthermore, RedMine, a web-based project management support tool, was used for documentation and collaboration, and Git was used for version control.

Three weeks before hand-in, an integration test was performed to ensure that the system worked as intended.

## Chapter 2

# Joint Analysis and Design

As stated in Section 1.1, all groups collaborated on the analysis and design of StudyLife during the first month, which resulted in a contract. This contract included aspects such as the system architecture, class and interface diagrams, functional and non-functional requirements. In this chapter, specific parts of the contract, that are crucial for this project, are interpreted and elaborated. The entire contract is included as Appendix A.

## 2.1 System Definition

In the first week all the groups agreed what the system was to become and how it should be defined. The following is the system definition for StudyLife.

> StudyLife is a social web community for students and supervisors, inspired by social networks like Facebook. The purpose for developing StudyLife is to create a social web community that improves the collaboration of users in between.
>
> Access to the system should be available globally through a web browser, i.e. registered users must be able to login to their accounts by directly accessing the website and then signing in with their user name and password anytime and anywhere.
>
> The system should ease communication and collaboration through information-sharing between users by having different ways of interaction, and give opportunity for relations between users which reflect real-life relations. Furthermore, the system should aid in organization and coordination by offering users a calendar with the possibility to create and share events.

## 2.2 Target Group and Actors

The system definition defines the target group of StudyLife as being people associated with universities. It is therefore expected of the user to have basic knowledge of using a computer.

Since the system was thought as a social web community, it was decided that there were to be as little hierarchy among users as possible. Due to this decision three actors were chosen for the system, namely visitors, registered users and administrators. The following describes each actor:

- A visitor is an unregistered person of the system. Visitors only have access to the login, registration and about pages. For a visitor to gain access to the system he needs to register a new user and log in using this new user.
- A registered user has a unique identification and a password. Using these credentials the user can log into the system and gain access to all of its functionality.
- The administrator is a developer of the system who is able to modify the code and database. This actor is not implemented in the system.

Based on these actors, functionality for registering and authorizing users was needed.

## 2.3 System Components

Based on the system definition, a set of functionalities, to be included, was derived through a brainstorm by all groups. To be able to split the work among the groups without imposing too much dependency in-between the groups, the system was split into separate components. The following lists and briefly describes the components of StudyLife.

- **User:** Visitors should be able to register as users and should then be able to log in. Furthermore, users has a profile page which can be edited.
- **Friend:** Users should be able to form friend relationships with other users.
- **Group:** Users can create, join and invite other users to groups. Depending on if the group is public or private.
- **Wall:** Each user and group has a wall that contains messages, comments, etc.
- **Notifications:** Users get relevant notifications, like when other users request friendship etc.
- **Files:** Users can upload files and organize them in folders. Groups also has a file folder that members can upload files into.
- **Calendar:** Each user and group has a calendar and can create events.
- **Chat:** Users can chat with each other in chat-rooms.
- **Search:** Users should be able to search in the system to find various information.
- **Master Layout:** A general layout of the system. This is not a component, but a significant task equivalent to the components.
- **Data access layer:** A common interface to the database. Like the Master Layout, this is not a component.

For a more detailed description of the other product functions not in this project see Appendix A.1.6.

StudyLife does not have an actual *core* component, but the heart of the system lies the User and Group components and data access layer as they are used by all of the other components. Figure 2.1 shows the UML component diagram for StudyLife.

All database access was done through the data access layer, using the functionality defined in the `IRepository` interface. Functionality regarding users, groups, requests and notifications was contained in separate components and accessed through individual handlers.

## 2.4 Quality Goals

In general, the quality attributes of a software application are measures of software quality, and are divided into two subsets [Mat09]:

- Static quality attributes refer to the actual code and its documentation, e.g. availability of documentation, testability and modifiability.
- Dynamic quality attributes relate to the application's behaviour when its being used, e.g. reliability, correctness and usability.

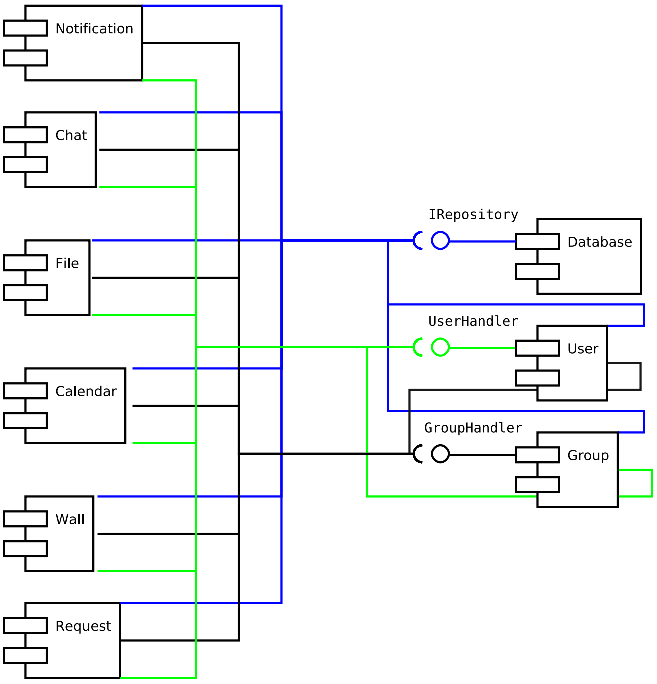


**Figure 2.1:** Component diagram for StudyLife

| **Importance** | **Medium** | **High** |
|---|---|---|
| Reliability | X | |
| Security | X | |
| Usability | X | |
| Availability | | X |
| Correctness | | X |
| Testability | | X |
| Modifiability | X | |

**Table 2.1:** Selection of quality goals defined in the first month

Table 2.1 contains a selection of quality goals for StudyLife important for this project, the rest can be seen in Appendix A.1.4.

As can be seen on Table 2.1 it was decided to focus on developing a reliable system, which is secure in the sense of handling personal data. The system should be somewhat usable as well, since the users are not necessarily “expert computer users”, but have, in general, average computer skills.

Since the StudyLife system was meant as being online and accessible on the Internet, availability was highly prioritized, as well as having a correct and functional system that was easy to test. Due to the system being split into components it was of medium importance to have a modifiable system.

## 2.5 System Architecture

During the first three weeks of the multi-project, four architecture solutions were proposed. The following briefly covers these four proposals, which are illustrated and more thoroughly described in Appendix A.2.2, and finally describes the chosen architecture.

### 2.5.1 Common Datastore Architecture

The common datastore architecture was thought of as one central datastore functioning as the “core” of the application. Components have their own representations of data and each ask the datastore for the data, hence all data is fetched and saved through the core. This architecture can be seen in Figure A.2.

An advantage of using this architecture would be the independence achieved by the total separation of all components. A disadvantage would be that including several components on one web page could prove difficult. Another disadvantage is that each component would need their own classes to represent shared data, hence creating a lot of redundancy in the system.

### 2.5.2 Layered Architecture

In the layered architecture the system consists of a number of layers: the database, data classes, services for exposing the data, and at the top, the components, which fetches data from the layers below. This architecture can be seen in Figure A.3.

An advantage of this architecture is that the logic is centralized which introduces control over the data. On the other hand, a disadvantage is that changes made to data classes might require changes in each of the components.

### 2.5.3 MVC

The Model-View-Controller (MVC) architectural pattern uses *models* for representing data, *views* for presentation, and *controllers* for managing business logic and input. This architecture can be seen in Figure A.4.

An advantage of using MVC is that the separation of *controllers*, *views* and *models* makes it possible to have multiple representations of the same data.

### 2.5.4 Web services

A centralized application that exposes interfaces to data via web services. This architecture can be seen in Figure A.5.

An advantage of this architecture is that it would be possible to reach multiple platforms with e.g. an application for handheld devices, a desktop application etc. A disadvantage is that the system would be sparse in functionality as it cannot be reused in-between the various applications.

### 2.5.5 Chosen Architecture

The architecture chosen for StudyLife was a combination of the first three proposals; Common Datastore Architecture, Layered Architecture and MVC. The advantage of these architectures was that all could be split in components, so the groups that were part of the multi-project could have specific components to develop. The combination is illustrated in Figure 2.2.

As can be seen on the figure the system was separated in four major layers, each described in the following starting from the top.

The Components layer included all the shared and nonshared components, where all were split into Model-View-Controller as can be seen on Figure 2.3. The shared components made interfaces available such that the other component were able to make use of these and their shared functionality.

The Data Access Layer (DAL) made it possible to read and write to the database, so it was not possible for the components to mess up the database.

The database layer included all the data for the StudyLife system.

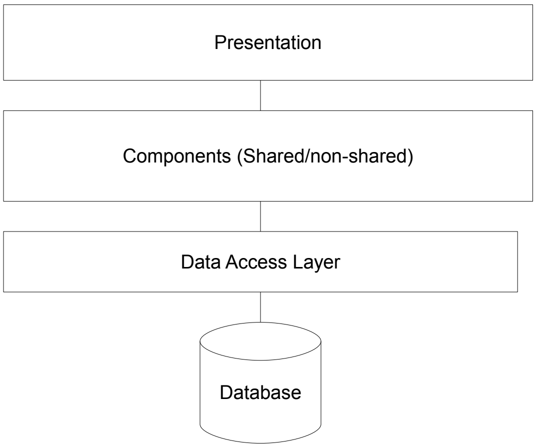


**Figure 2.2:** Architecture of StudyLife

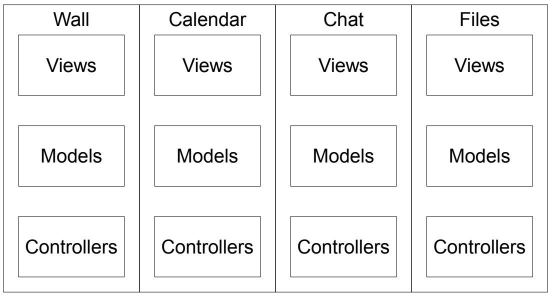


**Figure 2.3:** Shared/Non-shared components of StudyLife

The presentation layer incorporated all the *views*, from the various components in the MVC layer, in a master layout that was made available as the StudyLife user interface for the end-user.

## 2.6 Technical Platform

The technical platform chosen for this project was ASP.NET 3.5 SP1 and C# 3.0. Alternatives to the chosen platform was discussed in the multi-group, this discussion can be found in the contract. The choice of technical platform was based on the following points:

- C# is a powerful object-oriented language, enabling encapsulation of data and logic.
- Access to the large .NET library which contains solutions to common programming problems.
- Nearly all project participants had used ASP.NET and C# in previous projects, hereby minimizing time spent on learning a new language.

Within this technical platform, the newly released ASP.NET MVC 2 framework was chosen. This decision was based on the following points [Esp09]:

- Supports the choosen architecture.
- The MVC pattern benefits from high extendability, reuseability and maintainability.
- Strict control over markup and styling, as opposed to ASP.NET WebForms where a lot of code is generated.
- Easy to unit test logic.
- *Areas* functionality enables better modularity and separation of components.

To interface with the database, Fluent NHibernate, an Object Relational Mapping API, was chosen. The reason for choosing Fluent NHibernate was that it enabled handling data in an object-oriented manner, rather than writing SQL queries.

As Microsoft technologies were chosen for the technical platform it was decided to use the Microsoft IIS and SQL Server for the server setup. This choice was mainly to avoid any cross-platform issues.

## 2.7 Delegation of Tasks

As the system was split into components it was possible to divide the work among the project-groups. The following lists the responsibilities of each project-group.

- **Group s601a (This project):** User, Friend, Group, Search and Master Layout.
- **Group s602a:** Request, Chat, Notification and Newsfeed.
- **Group s603a:** Wall, Comment and File.
- **Group s604a:** Database and Calendar.

Brief user stories were made for all functionalities in the system, enabling all groups to agree on what the components should provide by means of functionality for the user. For more detail see Appendix A.1.5.

The responsibilities of this project were to create functionality for users, friends, groups, search and the master layout of the site. For each of these responsibilities, the user stories of the contract were used to define what needed to be implemented in order to fulfill the responsibilities of this project group. In the analysis chapter, each of these responsibilities are examined and a set of requirements is derived.

## 2.8 Summary

During the first month of the multi-project a common understanding of the system was agreed on in collaboration between the groups. This agreement was formalized in the contract. The contract contains overall choices for requirement analysis, design, collaboration tools and meetings.

With the joint process covered, the next step was to start developing the components in the individual groups. Prior to starting development, a development method needed to be chosen, this is covered in Chapter 3.

# Chapter 3
# Development Method

There exist many different software development methods that each try to solve the complex problems that may arise during the development of a software product.

Due to the project being a multi-project, concerning several groups, with iterative test and integration phases, an iterative development method was preferred. It was also required that the development method should be easy to implement. Lastly, the development method should comply with the size of the team, being three persons.

The waterfall method, Extreme Programming and Scrum were discussed as potential development methods for this project. Scrum was chosen, because it fitted the development method requirements, but also because it was a development method that none of the group members had experience with, and this project would be an ideal opportunity to acquire some. The following section will describe Scrum in detail and the implementation of it.

## 3.1 Scrum in General

Scrum is a popular agile development method, which emphasizes self-directed and self-organizing teams, daily team measurement, and avoidance of an already established process as in e.g. the waterfall model [Lar08]. Some of the reasons for Scrum's popularity is its simple practices, such as daily builds, no additional work to iterations and decisions to be taken in one hour, and its emphasizes on a set of project management values. Furthermore, practices from other agile methods can easily be combined or used in complementary to the Scrum practices [Lar08]. The following presents and describes the project management values of Scrum [SB02]:

- Commitment
  The Scrum team commits to a defined goal in an iteration. This enables all participants to have a common goal to strive for throughout the iteration.
- Focus
  Focus on the defined goals of the iteration. Focusing only on defined goals helps ensure that the project does not steer in a wrong direction.
- Openness
  The Product Backlog makes the work visible and the daily Scrum meetings (called Scrums) make the overall and individual status visible. Work trends and velocity are made visible with a backlog graph or burndown chart. Feedback from communication, backlog and graphs helps keep the project on the right track by enabling (and urging) all project participants to stay updated on the status of the project.
- Respect
  The different team members are respected for their different strengths and weaknesses, and respect for the whole team is emphasized over the manager. Respecting diversity and the individual team members helps create a more thriving work environment.
- Courage

> The team has the courage to take responsibility for self-direction and self-management. Not having a strict top-down management makes it more interesting as a developer to get involved in all aspects of the development process.

The process of Scrum is illustrated on Figure 3.1. The process consists of the following phases: Planning, staging, development and release [Lar08]. In the beginning of the Scrum process, i.e. in the first planning stage, a product backlog is created. The product backlog is a collection of user stories that are all conceivable requirements. Thereafter, a Sprint backlog is created, which is the backlog used in the current Sprint, including tasks, responsibilities, remaining work hours etc. An iteration in Scrum is called a Sprint, and is usually 2-4 weeks long. Furthermore, expectations are set, by analyzing and evaluating the backlog for setting priorities, and it is ensured that all tasks can be achieved. The staging phase identifies tasks and prioritizes enough for the next iteration.

The implementation is carried out in the development phase, where Scrums are held each day. At the end of the Sprint, the release phase deploys the working increment of the software and a Sprint review meeting is held, where the work accomplished during the Sprint is presented. Moreover, a Sprint retrospective is done. In this, all team members discuss what worked and what did not.

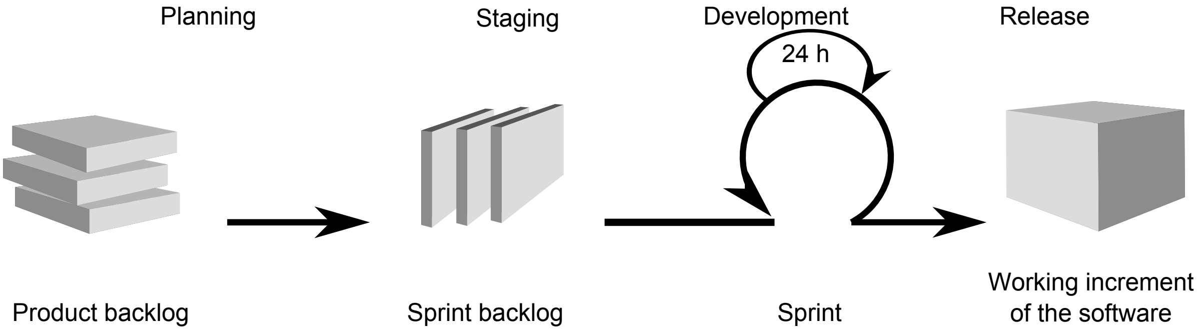


**Figure 3.1:** The Scrum process

Even though Scrum focuses on the team as a whole, some roles have been defined in order to delegate management tasks.

### 3.1.1 Roles

The people involved in the Scrum process are separated into four different roles [Lar08].

- Product owner
  Responsible for creating and prioritizing the product backlog, choosing goals for the next Sprint from the product backlog and reviews the system at the end of each Sprint.
- Scrum Team
  The Scrum team (developers) works on the Sprint backlog. All members of the Scrum team are equal in rank.
- Scrum master
  Manages the Scrum team, by removing blocking issues and ensuring Scrum values are held high, but is also a member of the Scrum team.
- Chickens
  Everyone else may observe, but not interfere. These persons are referred to as chickens.

## 3.2 Scrum in This Project

Scrum was rather simple to implement, and the iterations was chosen to be two weeks which fitted the project very well. Before the the first Sprint was started a product backlog was created, which did not have a concrete product owner, but was influenced by the work done in the first month of the project. The user stories in the product backlog were prioritized by their value, based on whether other project-groups depended on them.

Daily Scrums were held internally for ensuring communication, potential delegation of tasks and to track progress. Each week a new person was appointed the Scrum Master role. The role of the Scrum Master in this project was solely to ensure practices were performed correctly and continuously by all team members. Simple tools were used, namely the blackboard, for putting up tasks and drawing a burndown chart. During development, the possibility of instant knowledge sharing and communication had to be available, therefore the team decided that no one was allowed to write code at home.

### 3.2.1 Sprints

The project period was divided into four Sprints of two weeks each. The tasks on which the other groups depended, were put into the first two Sprints. These tasks were user and group handling. The Sprint planning was done prior to starting Sprint 1.

| **Sprint 1** | **Sprint 2** |
|---|---|
| 1. Log in<br>2. Registration (User)<br>3. Log out | 1. Create group<br>2. View group profile<br>3. Friend |
| **Sprint 3** | **Sprint 4** |
| 1. View user profile<br>2. Edit user profile<br>3. Edit group | 1. Search<br>2. Online status |

**Table 3.1:** Sprints

The Master Layout was developed throughout Sprints 1 to 3.

### 3.2.2 Practices

As mentioned earlier, Scrum has a set of simple practices that can be picked depending on the nature of the project, the team and other circumstances. For this project the following practices were used [Lar08]:

- Sprints.
  As written above, each Sprint had a duration of 14 days. These short iterations made the process more manageable, because developers were able to focus on the tasks at hand. This practice was included, because it is central to the Scrum process.
- Common room.
  This principle was well suited for this project since it was a university project, where group rooms were available. Exchange of knowledge was significantly easier, when all development was performed with all developers working in the same room. This practice was chosen because all development could ideally be done in the group room.
- Burndown chart for tracking progress.
  A burn-down chart is a graph representing work done versus time. The burndown chart made it possible for the team to get a quick overview of the progress of the project. The chart is especially

good if it is always visible in the room. This practice was chosen because a it is an easy way to keep track of progress.

- Test first.
  This principle is borrowed from XP. The test first principle enforces writing unit tests, because the developer may not proceed with implementing a feature until unit tests are written for it. This practice improves correctness of the implemented code.

- Standing Scrums.
  The purpose of Scrums is to update the team on the current state of the project, identify blocking issues and delegate work. This practice was chosen because it was an efficient way to keep the team up to date, as well as delegate tasks for the day.

- Decisions in one hour.
  One hour decisions is an easy way to avoid quandaries, that can otherwise block the process, and cost valuable time. This practice was chosen because it is an efficient way to keep the workflow going.

With the practices chosen, the development method was ready to be implemented in the project.

## 3.3 Distributed Version Control System: Git

Version control is used for sharing and keeping track of changes to project source code. This section elaborates on Git, a distributed version control system.

In the multi-project part it was agreed to use a new type of version control system (VCS). In previous projects, Subversion had been the VCS of choice for most groups due to it being well known and groups have had good experiences with it. The VCS used in this project however was Git, a fairly new way of handling source code in a software project.

Whereas Subversion is a centralized system revolving around one central repository, Git is a distributed system where each clone contains a complete copy of the repository. This allows developers to access the entire file history from their own repository [Git10].

One of the reasons for using Git in this project was its focus on branching. A Git repository can consist of many branches that can be merged and derived from each other. Branching is a very useful way to share source code in larger projects as components can be contained in separate branches and then later on be merged together to make up the entire system [Git10]. A branch could for example contain a specific feature or component. Figure 3.2 is a small piece of the branching history of the StudyLife development.

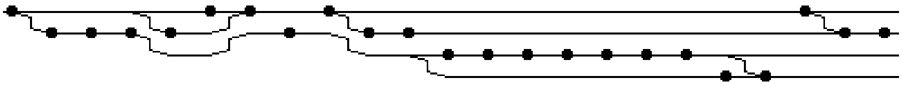


**Figure 3.2:** Git branching history graph

Each dot in Figure 3.2 represents a commit to a branch. Each horizontal line is a separate branch and horizontal lines represent a merge or branching to/from a branch.

## 3.4 Summary

This section covered the Scrum development method and the VCS Git. Furthermore, the Sprint planning was documented clarify what was to be done at each stage of the development.

The next chapter covers preliminary theory used in this project.

Chapter 4

# Preliminary Theory

During the development of StudyLife various technologies were used to facilitate the functionality and architecture defined in the contract. Before analyzing the components developed in this project, the essential technologies are briefly covered. These technologies include the Model-View-Controller architectural pattern, nHibernate Object-Relational Mapping for .NET, the web security vulnerabilities the system should avoid, AJAX and software testing. ï»¿

## 4.1 Model-View-Controller

Model-View-Controller (MVC) is an architectural pattern that originated in SmallTalk-80 as a pattern for separating the user interface (UI) from the data model and business logic. It consists of several methods that later have been transformed into separate design patterns, mainly the Observer, Composite and Strategy design patterns [GHJV94].

The following covers how MVC works in ASP.NET MVC, as it was used in this project.

### 4.1.1 ASP.NET MVC

ASP.NET MVC separates an application into models for maintaining data, views for displaying data, and controllers that handle events that affect views or models. Figure 4.1 depicts this separation. The arrows on Figure 4.1 show the relation between models, views and controllers. Controllers instantiate and manipulate models and views, hence arrows go to both model and view. Views fetch data from models and do not make direct contact with controllers, therefore only one arrow. Models contacts neither views nor controllers.

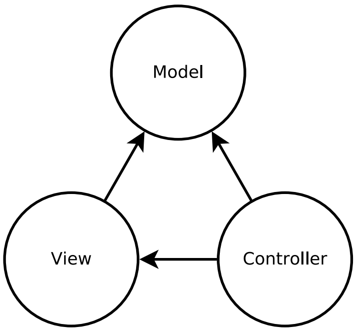


**Figure 4.1:** MVC sequence diagram [Mic09]

Models are used to represent data in the system. ASP.NET MVC does not specify how this data is stored, but some kind of data store like a database is often required. The separation of models makes it easier to use the same controllers and views for different models.

Views are the user interface that presents models to the user. The separation of views, from controllers and models, enables for multiple views of the same data. For instance, a list of users can be displayed in a large table with many details, or as a number indicating which users are online. In a web application a view can be an entire web page, a table with data or even an RSS feed. Which view is used, is determined by the controller.

ASP.NET MVC has two kinds of views: regular and partial views. Regular views are used for most pages. Partial views are used for content that is to be included on multiple pages and can be inserted into regular views. An example of a partial view could for example be used for a list of users that is to be displayed on multiple views. Creating a partial view for this list makes it easier to maintain and include on other views.

Views can also be strongly-typed allowing type errors to be caught at compile time. A view can be typed to the models which it receives from the controller. Listing 4.1 shows how to set the type of a strongly-typed view.

```
<%@ Page Title="" Language="C#" Inherits="System.Web.Mvc.ViewPage<FooBar>" %>
```

**Listing 4.1:** Defining a strongly-typed view

In Listing 4.1 the type of the view is set to `FooBar`. In 4.2 a `FooBar` model is passed from the controller to the view.

```
public ActionResult FooBarAction() {
  // perform some logic
  return View(new FooBar());
}
```

**Listing 4.2:** Passing a model from controller to view.

Controllers are the logic that determines which models and views to use for the response. A controller contains a number of actions that each contains logic for a request. For instance, a controller for user functionality could have actions for login, logout and registration. Which action is executed is determined though URL routing. Table 4.1 displays how the URL is parsed to determine which controller and action should be executed.

| | |
|---|---|
| **Structure** | Domain/Area/Controller/Action/Id |
| **Example** | www.studylife.com/Groups/Group/Profile/12 |

**Table 4.1:** ASP.NET MVC URL routing

The example URL shown in Table 4.1 would invoke `ProfileAction()` method in the `GroupController` class, which is located in the `Groups` *Area*. Areas are folders in the ASP.NET MVC project which can contain a set of controllers, views and models. This feature makes it possible to split a system into components each having its own MVC structure.

When a user requests a page, a lot is happening behind the scenes before he gets the requested page displayed in his browser. Figure 4.2 is a simplified illustration of what happens in the ASP.NET MVC framework when a page is requested.

When the server receives a HTTP GET request from the user's browser, the URL router instantiates the desired controller and action based on the requested URL. The action then performs its logic and possibly creates a model, which fetches some data from a database, and instantiates a view. The view then renders its content and possibly fetches data from the created model. When the view is rendered, it is sent to the user's browser as an XHTML page. ï»¿

## 4.2 Object-Relational Mapping

Object-Relational Mapping, in short ORM, is used to map objects in an object-oriented programming language into a relational database. By mapping classes into the database, the developer does not need to work directly on the database at all, only write the mapping files that defines the tables, data and relationships. This abstraction eases the development of many types of applications by removing the need for doing direct work on the database. Such direct work include creating and updating tables, as

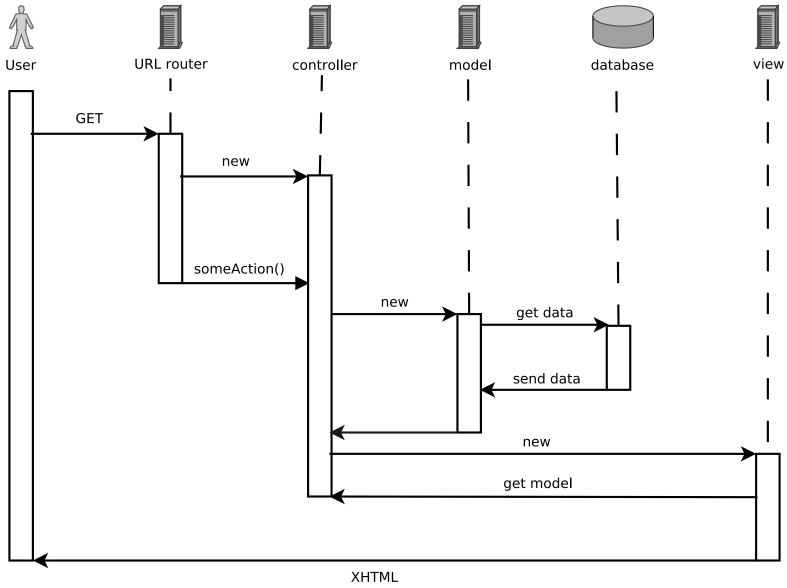


**Figure 4.2:** ASP.NET MVC sequence diagram

well as writing SQL queries. Defining mappings are most commonly done in separate files, decoupling it from the actual code and thus making it easier to change ORM version or framework [wA09]. Figure 4.3 illustrates the mapping between some class and a relational database.

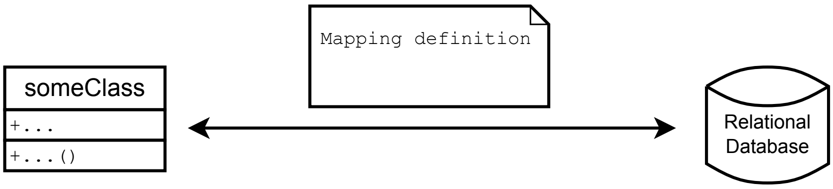


**Figure 4.3:** Object-relational Mapping

A disadvantage of using an ORM is that it imposes a performance overhead on the system. This overhead is however relatively small as opposed to the benefits of using an ORM. To tackle this problem, most ORM frameworks provide functionality for writing custom SQL queries and executing Stored Procedures [Tea10].

There exists a wide variety of ORM software. The following covers the technologies used in this project, namely NHibernate and its variance Fluent NHibernate.

### 4.2.1 NHibernate

NHibernate is an ORM framework which purpose is to map .NET classes to relational databases. NHibernate is based on Hibernate, an ORM library for Java developed by RedHat.

NHibernate uses XML files to define mappings for classes, e.g. how properties of classes should be stored i.e. data types, relationships between classes i.e. keys. For each class that should be mapped to the data store, a separate mapping file is required. The logic that operate on the mapping files is accessed through a dynamic-link library (dll) file.

NHibernate automatically generates the queries to be performed on the database to perform the desired action. For passing variables to the queries, parameterized queries are used. This method is safer and faster than dynamically building the query string. It is safer because it handles any kind of manual string escaping like those used in SQL injection attacks. It is faster as the database can cache queries and avoid parsing and compiling as the parameterized queries are recognized no matter their input.

### 4.2.2 Fluent NHibernate

Fluent NHibernate provides an alternative to regular NHibernate XML mapping files, by writing mapping files in native C#. This enables mappings to utilize the strongly typed features of C#, hereby improving type safety [osc09]. It eases the writing of mapping files as well.

Fluent NHibernate includes a feature called *Auto Mapping* which automatically maps class properties based on a set of conventions. The feature utilizes the principal of the Convention over Configuration pattern (meaning that only unconventional aspects needs to be specified), and hereby improves readability and conciseness of mappings [Gre09].

#### Example Mapping

The following covers an example mapping of a C# class into a relational database, using Fluent NHibernate. The class (see Figure 4.4) represents a user with and id, email, name and a list of friends.

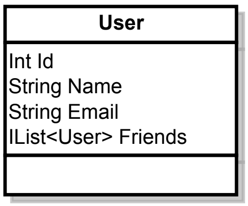


**Figure 4.4:** User Class

The Fluent NHibernate mapping class, Listing 4.3, used to map the User class is written in C#.

```
...
public class UserMap : ClassMap<User> {
  public UserMap() {
    Id(x => x.Id);
    Map(x => x.Name);
    Map(x => x.Email);
    HasManyToMany(x => x.Friends)
      .Table("Friends").ChildKeyColumn("friendId");
  }
}
...
```

**Listing 4.3:** Fluent NHibernate User mapping file

All functionality regarding fluent mapping is located in the namespace `FluentNHibernate.Mapping`. The mapping class (`UserMap`) is implementing the generic class `ClassMap<T>`, with the type set to `User`. This is to define which class it is mapping, and make the link between mapping and application model.

The constructor of `UserMap` uses several methods to define how properties of the `User` class should be mapped. The `Id`, `Map` and `HasManyToMany` methods are used to map class properties into table columns and relationships [osc09]. Parameters to these methods are passed as lambda expressions, where `x` denotes an instance of the class being mapped, hence `x` in Listing 4.4 denotes an instance of the `User` class. The line `x => x.Id` means: the `Id` property of the passed (`User`) object `x` is passed as a parameter to the method being called [Mic10b]. The `Id` method is used to denote that a property is an identifier of the class. The identifier is required and the value has to be unique. The `Map` method is used to map trivial properties of a class, e.g. name, email etc. The `HasManyToMany` method is used to create relationships, and in the example given Listing 4.3 to create a friend relationship between multiple users. There exists several other methods for creating mappings, see [osc09].

`UserMap` uses several Fluent NHibernate conventions to take advantage of the auto mapping functionality. For instance, `Id` is mapped into an auto incrementing integer in the table.

When the application is built and run, Fluent NHibernate synchronizes tables in a database, corresponding to the defined mappings. Figure 4.5 shows how the `User` class is stored in a relational database.

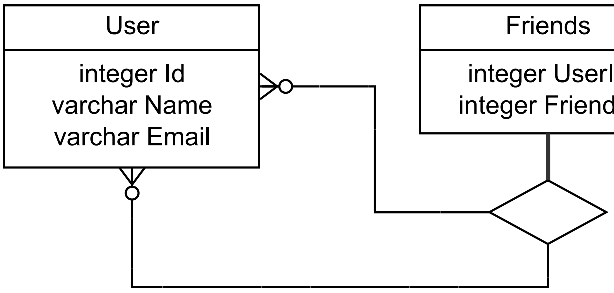


**Figure 4.5:** User Class mapped into a relational database using Fluent NHibernate

## 4.3 Web Security Vulnerabilities

With the Internet's growing popularity, millions of people share information and make purchases on the Web every day. This has also made it a primary target for the unlawful. StudyLife is a web application that it is available via network access, and it will contain personal data, such as email-addresses, passwords, private communication, acquaintances etc. It is important that these data are protected from unauthorized access by applying security facilities.

This section contains a description of the most common vulnerabilities in web applications, and how these can be avoided.

The Open Web Application Security Project has made a detailed list of top ten web application security risks for 2010 [OWA10]. Since this is not a project only concerning security in web applications, only three of the ten will be touched, namely:

- 1: SQL Injection
- 2: Cross-Site Scripting (XSS)
- 5: Cross-Site Request Forgery (CSRF)

The section also covers how passwords can be protected using hashing and hash salts.

### 4.3.1 SQL Injection

For an application to communicate with a database, it needs to perform queries. These queries might contain user input such as an email address or a blog post. This makes it possible for an attacker to write input that, when inserted, changes the outcome of the query. This security vulnerability is called an injection flaw, and since SQL is a very popular language for web data stores, SQL injections are one of the most well known vulnerabilities [MSK09].

An SQL injection is where user input, from e.g. a login form, is inserted directly into SQL query strings and changes its outcome. If the user input is not checked in any way the attacker can post SQL syntax that changes the outcome of the query string, i.e. wrongfully authenticating the attacker or deleting data in the database.

Table 4.2 gives some examples of how an SQL injection looks like.

| **Bypassing Authentication** | **Input field values** |
|---|---|
| To authenticate without any credentials: | Username: `'  OR '1' = '1`<br>Password: `'  OR " = '` |
| **Causing Destruction** | **Input field value** |
| To drop the users table in the database: | Username: `';DROP TABLE users;- -` |

**Table 4.2:** Examples of SQL Injection. Source: [MSK09]

As can be seen in Table 4.2, these injections can have very harmful consequences. They can however be avoided by strictly filtering the query strings for string literal escape characters (' or "). Or by replacing the SQL statements with stored procedures (or parameterized queries) and/or specify users, roles, and permissions [MSK09].

#### Injections in Fluent NHibernate

Generally it is rather difficult to make SQL injections in Fluent NHibernate, since it uses parameterized queries for all generated SQL statements on databases that support these. If however the developer uses custom methods that accepts user input, hereby bypassing the ORM's fucntionality, the system might be open to injection attacks [OWA10]. It is therefore recommended to use the ORM's functionality.

### 4.3.2 Cross-Site Scripting

Cross-site scripting, referred to as XSS, is a technique that, like most other vulnerabilities, arises from insufficient input/output validation. However, XSS is not specifically targeted at the application, but rather the users of the application [MSK09].

An example could be an attacker, posting a message containing executable content to a guestbook. When this message is read by another user, the browser will interpret it as code and execute it, and potentially give the attacker access to the user's system.

Nearly all XSS exploits are made on applications where it has failed to safely manage HTML input. Table 4.3 shows some examples of how XSS is used.

| **XSS Attack Type** | **Example Payload** |
|---|---|
| Script injection into a variable: | `page.asp?var=<script>alert('you got hacked')</script>` |
| Injecting JavaScript into the src attribute of an HTML image tag: | `<img src="javascript:alert('you got hacked')">` |

**Table 4.3:** Common XSS Payloads. Source: [MSK09]

To avoid these kinds of attacks one can filter input parameters for special characters. That is, no input in the web application should accept the following characters if possible: < > " [MSK09]. Another method could be to scan input for specific keywords like, "<script>", "javascript" etc.

In case the web application uses cookies, one can use the ASP.NET `HttpOnly` function, that prevents cookies from being accessed by scripts. This feature is used by default in ASP.NET 2.0. In addition, one can encode any data entered by website users to HTML by using the ASP.NET functionality `HTML.Encode` [Mic10c].

### 4.3.3 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) has been known for a long time, but it is just recently that it became recognized as a serious vulnerability [MSK09].

Web applications often give the users a persistent cookie, so they do not have to authenticate at each request. However, if an attacker is able to provoke a request from the user's web browser, the attacker can exploit the persistent cookie and execute actions on behalf of the victim. An attack may have malicious results for the victim, e.g. change of password, transfer of funds.

Avoidance of CSRF can be done by somehow binding the incoming request to the authenticated session. That is, the web application could generate a random value, store it in the session and as a cookie in the users browser. By matching the two values on browser requests, the application is able to ask for re-authentication if the values does not match. Another method is to require re-authentication each time users perform vital changes [MSK09].

In ASP.NET MVC, CSRF can be prevented by using the `Html.AntiForgeryToken()` helper. This helper does the same as explained above; binding a random key in the session, as a cookie and a hidden field in the form [Wal09]. The keys must then match when the form is submitted.

### 4.3.4 Password Hashing

A common security weakness in web-applications is passwords that are stored as plain text. This is a security weakness because, if an attacker somehow could obtain a database table of user names and passwords, for instance through operating system or server software exploits, he would be able to log in as any of those users and perform harmful actions.

The weakness can be avoided by storing a hash value of the password instead of just plain text. A hash value is a number with a fixed length, which unequivocally identifies strings of arbitrary length. There exist many hash functions, one such is SHA1. The password is hashed at user account creation or password alteration, and whenever the a user tries to log in, the entered password is hashed. It is then compared with the hash value in the user name/password hash database table. If an attacker were to obtain the content of this table, he would have to guess or brute force every conceivable password and hash them until he found a match.

#### Salted hash

An attacker may still be able to do a *Dictionary attack*, which is done by hashing every entry in a dictionary (a list of words) and comparing it to the hashed passwords in a database table. This is a relatively effective attack, since many users use common words or names as their passwords, if they are allowed to do so. This weakness can be avoided by concatenating a *salt*, which is a arbitrary value, to the password before hashing. The attacker now has to brute force every conceivable salt, for every entry in the dictionary [Wil04]. When a user logs in, the salt concatenated to the entered password, which is then hashed, and compared to the value in the database corresponding to his username.

## 4.4 AJAX

AJAX stands for Asynchronous JavaScript And XML and is not a technology in itself but rather a combination of existing technologies. In short, AJAX is a technique for updating selective parts of a page using JavaScript. It is asynchronous because it removes the need of having to reload the entire page when updating its contents. As JavaScript is executed in the browser (using the `XMLHttpRequest` API), it can perform the updating mechanism autonomous of the user. The data being sent by an AJAX request and received from the server is often in the form of XML (Extensible Markup Language) or JSON (JavaScript Object Notation) [Eer06]. Figure 4.6 shows the sequence of a typical AJAX request.

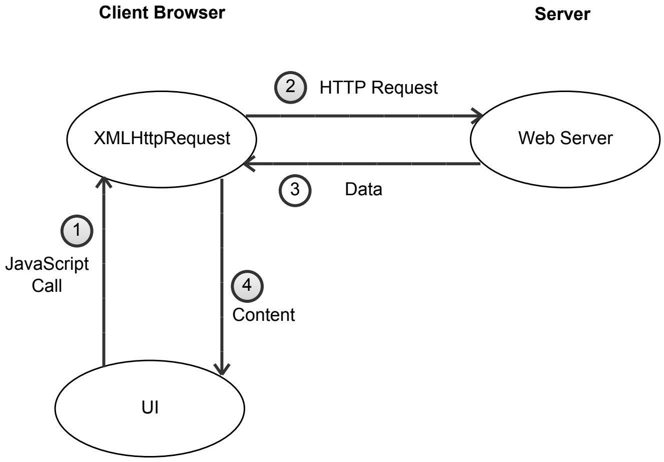


**Figure 4.6:** Typical AJAX request using `XMLHttpRequest`

(1) An AJAX request starts with a JavaScript call to the `XMLHttpRequest` API. (2) `XMLHttpRequest` then sends a HTTP request to the server possibly containing data. (3) The server responds to the request by returning data. (4) The data is then rendered if needed and passed to the UI.

There are two main methods of updating the content on a web page using AJAX: client- and server-side rendering. With client-side rendering, the AJAX request fetches raw data (JSON or XML) from the server and manipulates the DOM (Document Object Model) accordingly. This could for example be updating a table by adding or removing rows such that it displays changes in the data. With server-side rendering the AJAX request fetches content (XHTML/CSS) from the server that is ready to be inserted directly into the page.

Client-side rendering minimizes bandwith usage and server load, but requires JavaScript logic for rendering the content. Server-side rendering requires less JavaScript logic sent at the client, but increases bandwith usage and server load.

## 4.5 Software Testing

A large problem in software development is to ensure that a system does exactly what it is intended to do. This problem can be expressed by the quality attributes, correctness and reliability. As the codebase for the system grows even larger, the problem becomes increasingly complex, and the probability of reliable software often decreases [Plc07]. Before describing how tests of a software application can be done, one need to know about these two basic concepts related to testing, namely the reliability and correctness.

Both reliability and correctness are dynamic quality attributes, which means they are measured when the application is running and are therefore not the simplest attributes to test. It is possible to make use of automatic testing on the basis of the application's requirements.

Reliability of an application is defined to be the probability of its successful execution on a randomly selected element from all given inputs [Mat09]. Compared to correctness, which is defines an application to be correct if it behaves as intended on all given inputs, the reliability is a continuous metric whereas correctness is a binary metric; either correct or incorrect [Mat09]. Correctness is generally not the objective of testing, as it may be difficult to test on all possible inputs. Nevertheless, requirements that specifies the intention of the application and how it is to be used, increases the chances of a more correct and reliable application [Mat09].

To increase both reliability and correctness software testing on the basis of requirements is a way of ensuring that a system works as intended.

Besides testing, several other methods for ensuring a system works as intended are available. Some of the other methods are model-checking and verification. Model-checking is only applicable to model-driven development and verification is very costly and is intended for real-time systems rather than web application.

The following will cover unit testing, integration testing and system testing, as these are the typical approaches for testing web-applications.

Unit-, integration-, and system testing are performed at various stages of the development phase. Figure 4.7 depicts where the different forms of testing can occur, during the development process.

In an ideal situation all tests pass and it is not necessary to alter the code. This is however an unlikely situation and test that fail will require alterations in the code in order for them to pass.

The following describes unit-, integration-, and system testing for web applications.

### 4.5.1 Unit Testing

Unit testing is testing code at a functional level and not the system as a whole. In an OOP language unit tests are primarily written to test separate pieces of code like classes and methods. Testing at this level is also referred to as *white-box testing*. Unit testing can also be written for testing external sources

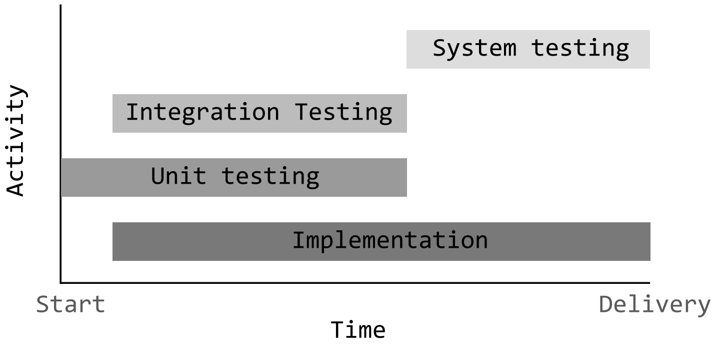


**Figure 4.7:** Testing activities over time

like interfaces and API's. Testing at this level is *black-box testing*, as the developer might not have any knowledge of the internal implementation of the functionality being tested [Mat09].

Unit tests mostly consist of statements that test if a specific value returned by the functionality being tested is the same as what would be expected. There exist many different unit testing frameworks that make it more convenient to write and run unit tests. For ASP.NET C# the most commonly used frameworks are NUnit and Visual Studio Unit Testing Framework. Listing 4.4 shows a simple unit test using the Visual Studio Unit Testing Framework.

```
[TestMethod]
public void ExampleTest() {
  String actualName = "johndoe";
  User expectedUser = new User(actualName);
  Assert.AreEqual(expectedUser.Name,actualName,False);
}
```

**Listing 4.4:** Unit test example

The unit test in Listing 4.4 tests if the constructor for the `User` class sets the `Name` property correctly. The `Assert` class contains several methods for verifying if a unit test should fail [Mic10d]. In this example, the `AreEqual` method is used to verify that the actual name, is the same as the expected name by comparing the two `String` objects. The last parameter denotes whether the comparison should be case-sensitive.

Unit tests can be written at any time during the development, but the value of having unit tests for a system is higher if they are written prior to implementing the functionality being tested. When having written the unit test beforehand, the developer gets valuable feedback during the development and can easily see if the functionality is broken or works as intended. The method of always unit testing functionality before implementing it is also called Test-Driven Development (TDD) [Mat09].

The amount of code in a system that is tested is referred to as the *code coverage* of a system and is measured in percent. Code coverage is sometimes a business decision as the correctness of a system can have a huge impact on a business. There exist several tools for measuring the code coverage.

### 4.5.2 Integration Testing

Integration testing is the part of the development where individual components are gathered (integrated) into one system and tested as a whole.

Integration testing can be done using either a nonincremental integration; big bang, or by using incremental integration; top-down, bottom-up or smoke testing [Pre05]. The following describes these methods.

#### Big Bang Testing

In this approach all components are combined at once, and the application is tested as a whole. In theory one does not waste any test effort, but in practice this is not the case, since errors will most probably occur and may be very hard to correct due to the expanse and complexity of the application. Besides, major errors may be discovered very late in the process, which may postpone the release [Pre05].

#### Top-down Testing

Top-down integration testing starts by testing top level functionality and then moving down to reach more specific functionality. The top-down strategy discovers general errors in an application early in the process, but component specific issues might not be detected until late in the process [Mat09]. In a social network web application top level functionality would be the handling of users and displaying the layout. Bottom level functionality would be specific actions like removing a friendship between two users or setting a user's name property.

#### Bottom-up Testing

Bottom-up integration testing is very similar to top-down, but starts by testing the lowest level of functionality and then moves upwards. The bottom-up strategy discovers component specific issues first and general issues last [Pre05].

#### Smoke Testing

Smoke testing is a way of ensuring that code changes work as intended and does not break the build, before being submitted to the codebase of the system. Smoke testing is one of the most cost effective methods for catching and fixing errors in software [Mic05]. One of the reasons for this is that a lot of errors are kept from being submitted to the codebase and possibly breaking the build for all other developers on the team.

### 4.5.3 System testing

System testing occurs after the application is finished. It is a method of evaluating if the requirements are fulfilled. There are many different types of system testing, but only usability-, ad hoc- and manual scripted testing will be covered.

#### Usability Testing

Usability testing is used to evaluate a product by letting target users use the system and then monitor their usage and feedback. The target user tests for ease of use, functionality and performance [Mat09]. Usability testing is a great way of testing if a system meets its intended purpose and if it has any value for the user.

#### Ad hoc Testing

Ad hoc testing is a form of testing that, tests the system for defects. This however is done without any systematic approach, and each test is only tested once, unless an error occurs. Hence, it is up to the tester to improvise and find as many bugs as possible [Mat09].

#### Manual Scripted Testing

The manual scripted testing method tests software by having test cases or test specifications that are created and reviewed by the project group. The method is in many variations, and the test specifications can be made at a functional level or by having a scenario for the tester to follow. The fact that test scenarios are used, decreases the need for test supervision and supports consistent execution of tests [Tes].

## 4.6 Summary

Throughout this chapter relevant theory for the development of StudyLife has been described. In Section 4.1 MVC was describe. Hereafter, ORM was explained in Section 4.2. In Section 4.3 some of the most prominent web security vulnerabilities and solutions for them were described. Then a description of AJAX was done, and finally some of the different test methods that can be used to test a large system were described in Section 4.5.

After having gone through some relevant theory, the documentation of the StudyLife analysis is next.

# Chapter 5
# Analysis

The contract included some requirements for the system, but each component still needed a more thorough analysis.

The analysis is divided in sections covering the parts of the system that this project was concerned with. This includes, the master layout, web security measures, Group-, User-, Friend and Search components (as specified in Section 2.7).

## 5.1 Master Layout

The master layout was the UI that formed the general design and layout template for the whole system. This section contains the analysis that was done regarding the requirements of the master layout.

The analysis that formed the basis of the master layout, was carried out in the first month. This analysis included a definition of the target users, which were people affiliated with a University. The target users were characterized in the form of three personas. As these personas all had an above-average level of understanding when it came to using computer software, extensive guidance and documentation in StudyLife was not found to be required. Therefore, the main focus were not put on the usability of the site, but rather the functionality. However, the system was designed to be fairly simple to use and not require a big effort to understand, but still give appropriate feedback or messages in case of errors. This complied with the quality goals set in the contract (covered in Section 2.4). Furthermore, since the users were expected to be affiliated with a university, the language of StudyLife was decided to be in English to accommodate international students and supervisors.

Cf. the user stories in Appendix A.1.5, the visitors on the site should be able to create a user profile and interact with the other users in the system through different functionalities. These functionalities should be made available and therefore visible in the design of the system.

A UI is defined as an interface that is used by a user to interact with a given system. The quality of a UI affects usability and depends on who the users are and where the system is to be used. A good UI is adjusted to the users' tasks and their understanding of the system [MMMNS01].

To enhance the user experience of StudyLife self-updating elements could be included on the page. This could for instance be an instant notification to the user if he receives a request. Such instant feedback should be included to give the user a better experience of using a social site [Arl06].

To sum up, StudyLife should be fairly easy to use. The master layout should make the functionality of the system accessible. Furthermore, StudyLife should provide instant feedback on relevant user activity.

## 5.2 Web Security Measures

In StudyLife security is weighed medium, as can be seen on Table 2.1, since the system contains personal data, such as email, passwords and private communication. Hence, the system needs to be secure, as in able to reject unauthorized access and protect the system from destructive attacks.

This section will describe authentication and authorization as solutions to some of the security issues pointed out in Section 4.3.

### 5.2.1 Authentication

StudyLife should be available on the Internet, but users might have personal information they do not want to share with outside users. For this reason, and to make sure that users were who they declared to be, StudyLife required authentication.

Authentication is defined as "the process of obtaining identification credentials such as name and password from a user and validating those credentials against some authority" [Mic10a]. This means that users of StudyLife initially needs to register using self-declared credentials for letting the authority, StudyLife, validate their credentials upon log in. The weakness of this system is that passwords can be stolen, revealed or forgotten. However, there exist solutions to these issues, e.g. digitally signed certificates, retrieval of passwords by email etc.

In StudyLife authentication would benefit the system by, besides the before mentioned, letting the system know who the current user is. This was an important feature, since all functionality given to the users in StudyLife were user specific. For example, each user should have personal user information, a personal calendar, and may be part of a specific group.

The authentication should give options for users to register, login and logout of the system. The register functionality should obtain some general user information, identification and a password, which eventually should be used when the user logs in to the system. Password recovery was not a requirement. Additionally, to increase security, passwords should not by stored as clear-text, but rather as salted hash-values.

### 5.2.2 Authorization

Authorization is defined as "a filtering mechanism that allows access to the client or server environment only by those individuals with appropriate authorization codes" [Pre05]. Thus, when the user has been authorized, he should be permitted to make use of all the functionality in StudyLife. If not authorized, the user should not be allowed into the system.

Authorization should give the possibility of having private and public functionality in StudyLife. For example, a certain group may be private, where only users who are member of this group can see its content. A user who is not a member of the private group, may therefore request for membership and become a member. Additionally, authorization for groups should make it possible for, e.g. only letting group members edit a group.

Since a user has a personal wall, it should be possible to determine if another user is authorized to write on this wall. For example by using functionality from the User component to find out if the two users are friends. Moreover, users may have personal data they don't want other users to see, therefore a user should be able to hide specific information for everyone but the user's friends.

Authorization should have the possibility of finding out if a user is logged in or not, to keep unauthorized users out of the system, tell if a user is a member of a specific group, and find out if two users has a friend relationship.

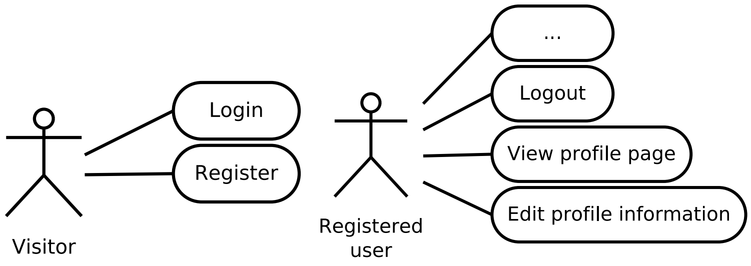
## 5.3 User Component

This section contains an analysis that consolidate the design and implementation of the user component, by means of deriving a set of requirements.

The User component regarded the modeling of real life persons, so they could be part of the StudyLife social network as users. The contract states that the target users should be university people, who might want to share work, or use the system for either social networking or for educational purposes. Therefore, a user should have functionality that accommodated both work and social matters. This functionality however, could only be made available if the user was registered in the system, and was logged in.

Newly arrived users, called visitors, should be able to register a new user profile in StudyLife, as can be seen in Figure 5.1a, with information that suited a social networking system in a university community. This could for example be a name, so the user could be identified, study, so users attending the same course could find each other, and a description, to attract other users' attention and increase the social aspect. After a successful registration, the user should be logged in to StudyLife.

To give the users value of the system, the user should have access to different kinds of functionality cf. Figure 5.1b. Besides the user specific functionality, change of the personal information and logout of the system, a registered user should have the ability to make use of all the functionality given by the different components of StudyLife. In correlation to the user specific functionality the system should have functionality to handle specific users, e.g. saves, reads and deletes user data.



**(a)** Use case diagram for a visitor

**(b)** Use case diagram for a user

**Figure 5.1**

### 5.3.1 Online Status

One of the prime functionalities of a social network is communication between users. In StudyLife this was done through chatting and wall posting. Hence, it would be convenient for a user to see when other users are online, enabling him to assess when to expect an answer to e.g. a wall post or friend request.

Furthermore, a user should only be able to chat with users that are online, as a chat is a real-time conversation. This means that the chat component would require information about a chat room's occupant's current online status.

Thus, it is required for the social network to be able to find out about users' online status and make it visible.

## 5.4 Friends Component

For creating a social community users must be able to connect with other users. This connection is referred to as a friend relationship in this project. Hence, friends in StudyLife denotes relationships between users.

Friends are important for the social aspect, the collaboration and whole idea of StudyLife. By letting users form friend relationships, means that the system would be community-driven.

As a result, Figure 5.2 depicts the requirements of friends. Users should be able to request friendship to other users and likewise receive requests. Secondly, a user should be able to get an overview of his friends. Furthermore, in case a relationship brakes, users should have the ability to remove a friendship.

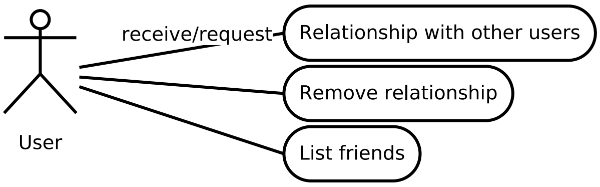


**Figure 5.2:** Use case diagram for a user and his friend relationships

## 5.5 Group Component

As written in the contract a group can be created by any user in StudyLife. The group can either be public or private, meaning that anyone in the StudyLife community can request memberships for public groups, whereas in private groups, users need invitations join.

Groups consist of a set of users, and to be able to fulfill the requirements of a university project group, e.g. sharing news, files and a calendar, the pages with these functionalities should be present on the group page.

As for real world groups, groups in StudyLife should be able to expand, contract or vanish, hence it should be possible to add and remove users in the group as well as remove the whole group.

As mentioned above, the groups can have different purposes. The groups need to have a name and a description that tells what they are meant for. Moreover, for groups that get requests from other users wishing to join a specific group, a user responsible for replying should be available. Such a user will be referred to as the group contact.

From a user's point of view a group should have all of the above mentioned functionality. This functionality is illustrated in Figure 5.3.

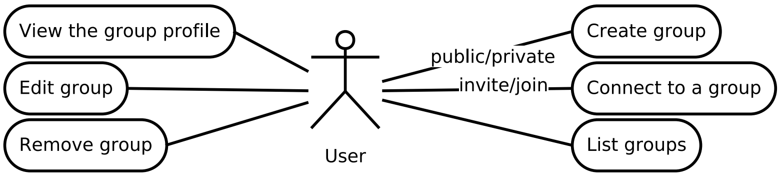


**Figure 5.3:** Use case diagram for groups

## 5.6 Search Component

As websites become increasingly large and contain more pages and information, the time required for users to locate specific information rise. Several approaches can be taken to reduce the time users spend looking for information on a website. One approach could be to organize pages in an intuitive hierarchy and provide a page that displays an overview of this hierarchy. Such a page hierarchy is often referred to as a sitemap, i.e. a map of the site. Figure 5.4 shows an example of a sitemap for a small website.

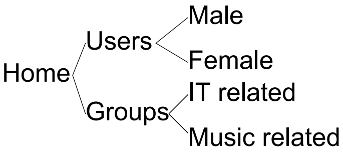


**Figure 5.4:** Example of a sitemap for a small website.

While a sitemap gives a good overview of the site, it does not provide enough information as to what the pages actually contain but only where the user might find the information he is looking for. Furthermore, as a website becomes larger, so does the sitemap, making it more complex and harder for the user to locate a specific page.

An alternative approach, to help users locate information on a website, is to include search functionality. Doing so enables users to quickly type in what they are looking for and instantly get feedback from the site in the form of search results. The fact that most users know how to use a search field, further supports the reasons for including search functionality [Nie97].

On sites like StudyLife it is hard to organize the content, because it is generated by users and therefore can practically be anything. Having search functionality takes care of this problem as it is a more dynamic way of locating information.

When a user wants to find specific information on a website, one of the things that he looks for is a search field. By including a search field on all pages and in a consistent location makes it easy for users to search for specific information [Nie97].

When a user performs a search he is taken to a result page where the results are displayed. For all searches, date in the following components should be searched; users, groups, events, files and chats.

To sum up, StudyLife should include search functionality (in a consistent location) that enables users to search the site. The searchable data is; users, groups, events and files. The search results should be categorized to make it easier to locate relevant results.

## 5.7 Requirements Specification

In this chapter, the components assigned to this project was analysed to derive a set of requirements. These requirements contain both the requirements from the contract and requirements derived this project. The following lists the functional and non-functional requirements for this project. Requirements from the contract are denoted by a *.

### 5.7.1 Functional Requirements

- Master Layout
  - Fairly simple and consistent user interface.
  - Make the functionality of StudyLife accessible. *
  - Provide instant feedback on relevant user activity.
  - English as primary language.
- User Component (including authentication and authorization)
  - A user should have a personal profile that is visible to other users. *
  - A user should be able to edit his personal profile. *
  - A user profile should have sub-pages displaying wall, calendar, files and chat components.
  - A user should be able to delete his account.
  - A user should be able to see which other users are online.
  - User online status should be available to the chat component.
  - A visitor should be able to register an account to become a user. *
  - A user should be able to authenticate himself by logging in. *
  - A user, who is logged in, should be able to log out again. *
  - Passwords should be stored as salted hash-values rather than clear-text.
- Friend Component
  - A user should be able to form friend relationships to other users. *
  - A user should be able to list his friends
  - A user should be able to break friend relationships to other users.
- Group Component

  - A user should be able to create groups. *
  - A group should be able to be either visible to all users (public) or only visible to its members (private). *
  - A group should have a profile page displaying general information on the group. *
  - Members of a group should be able to edit the group profile.
  - Members should be able to invite other users to join the group. *
  - Users should be able to a request membership for public groups. *
  - A group profile should have sub-pages displaying wall, calendar, files and chat components.
  - A group should have a set of users associated with it (members). *

- Search
  - Users should be able to search the system for relevant information. *
  - Search results should be categorized into relevant categories, e.g. users, groups, etc.
  - The search functionality should be available on all pages in a consistent location.

## 5.8 Summary

Requirement analysis for the individual components delegated to the project group, was covered in this chapter. The chapter elaborated on requirements regarding the master layout, security concerns, the User, Friend, Group and Search components. Finally the requirements were collected in a requirement specification.

The next chapter covers design of the components delegated to this project-group.

# Chapter 6
# Design

The contract contained an overall design of StudyLife, but more comprehensive design for the individual components was needed in order to implement them.

This section contains documentation of all the components that were delegated to this project group. This includes the master layout, authentication approaches, the Users component, the Friends component, the Group component and the Search Component.

To avoid ambiguousness, technology specific terms like model, view and controller are written in *italics*. Classes, methods and class properties are written in `monospace`.

## 6.1 General Design

Before going into detail of designing the specific components, an overview of the system is useful. Hence, in this section an explanation of what a component generally contains, and a sitemap of the components of this project. In addition, the Entity model and Entity handler are described.

A component could generally be split into five parts: the *views*, *controllers*, *handlers*, *mappings* and *models*. The purpose of handlers is to encapsulate code that regards component specific content, such as models, so that other multi-project groups may disregard these component specific concerns. The following sections contains a description of the parts of the components.

All the *views* for the different components, delegated to this group, can be seen in the sitemap, illustrated in Figure 6.1. When the user initially visits the StudyLife, he is met with the login form. From here users can authenticate through login or registration. Authentication is symbolised by the lock icon. The *views* below the lock are the ones that only can be accessed when the user is authenticated. The individual pages are described in the different component sections.

### 6.1.1 The Entity Model

The `Entity` *model* class should contain a single property: `Id`. Both the `User` and `Group` classes should inherit from the `Entity` class. As both users and groups have similar components like wall and calendar, making them inherit from the `Entity` class would mean that such components only need a relation to the `Entity`. This would make the system more extendable, since the central models, `Group` and `User`, could remain unaltered when adding new *models*. For instance, both users and groups would have a calendar. Instead of having a relation to calendar from `User` and `Group`, or making extentions to the calendar for both classes, the calendar could have an `Entity` property, which could then be either a `User` or a `Group`.

The relation between `User`, `Group` and `Entity` can be seen in Figure 6.2.

Mapping the `Entity` class could be done using the `Id` method described in Section 4.2.2.

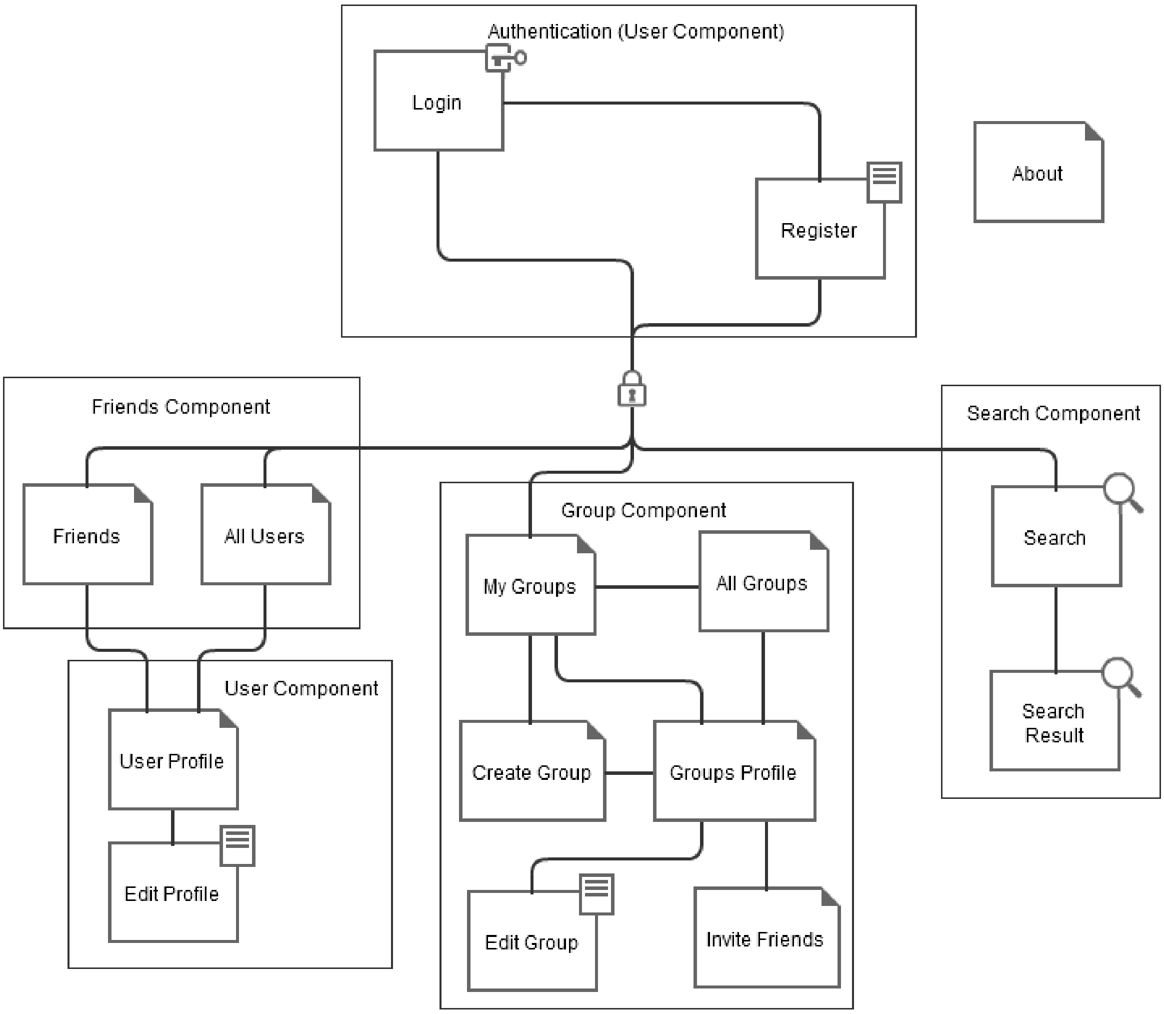


**Figure 6.1:** General sitemap of the the components in this project.

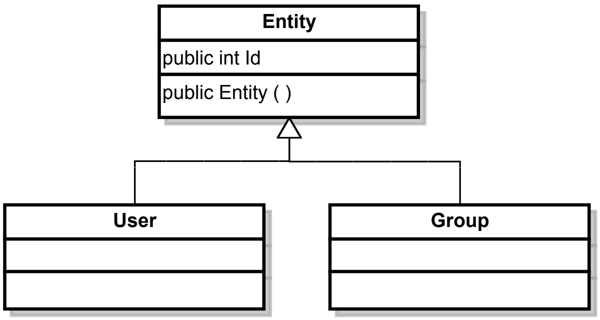


**Figure 6.2:** The `Entity` class.

### 6.1.2 The Entity Handler

The `Entity` handler class should contain a single method: `getEntity`. This method should return either a `User` or `Group` object depending on whether the input corresponds to a `User` or `Group`.

## 6.2 Master Layout

The requirements for the master layout states that it should: have a simple and consistent UI, make accessible the functionality of StudyLife, provide instant feedback on relevant user activity and have English as primary language. The following covers how these requirements could be satisfied.

The design of the master layout was inspired by the design of Facebook. Using elements from the design of the biggest social network available, gave StudyLife a chance of being easier to use. The reason for believing StudyLife would be easier was the fact that nearly half of the Danish population in 2010 had a profile on Facebook [fac10].

To further improve the usability of StudyLife, three design principles [RPS06] were applied. In addition, some UI patterns were used to keep the design simple and organized. The following describes how the master layout of StudyLife was designed in the order and subject of colour, UI patterns and structure.

A common color palette was chosen to achieve a consistent layout throughout the system, i.e. all components looked alike in terms of colors. Figure 6.3 shows the color palette chosen for StudyLife.

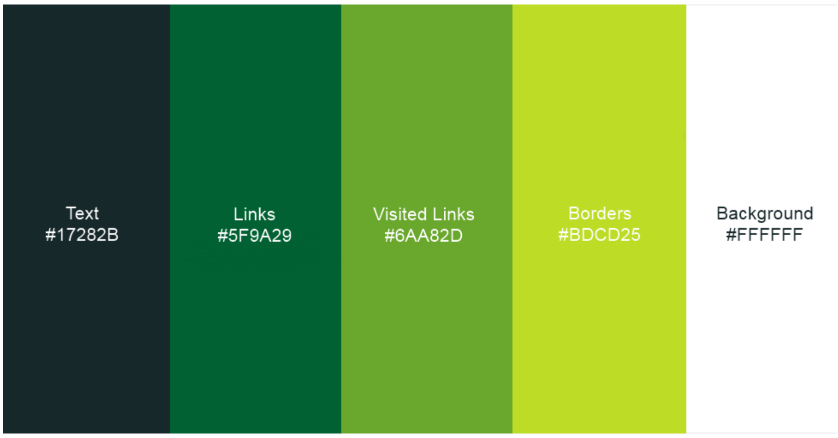


**Figure 6.3:** Color Palette.

The palette was found using [COL10], and each square was used for different elements on in the design.

### 6.2.1 User Interface Design Patterns

Several of UI patterns exists, but only some of them were found usable, in the light of the requirements specification made in Section 5.7. These patterns were:

- Top-level navigation
  The top-level navigation pattern provides a top-level menu, often coupled with a logo, that makes it possible to directly navigate to any of the system's major functionality [Pre05].

- Fill-in-the-blanks
  The fill-in-the-blanks pattern concerns forms and input, and allows alphanumeric data to be entered in a text box [Pre05].

- Simple search
  Provides the ability to search a web site or data source for an element, by entering an alphanumeric string in a search field [Pre05].

The top-level pattern is a very known pattern, and most sites on the Internet uses it. For this reason, the users of StudyLife would be able to recognize elements in system and the top-level pattern was therefore decided to be used.

When a user needs to input different sorts of data, the fill-in-the-blanks pattern is essential.

The simple search pattern is very handy and is used in many places on the Internet. The search functionality should be taken care of in the Search component. The pattern is included here to underline that a search field should be visible at all times.

### 6.2.2 Contents of the Master Layout

After having gone through the different sorts of UI patterns, a mock-up was created, as can be seen on Figure 6.4. The StudyLife master layout was divided in three different parts. The following describes each part.

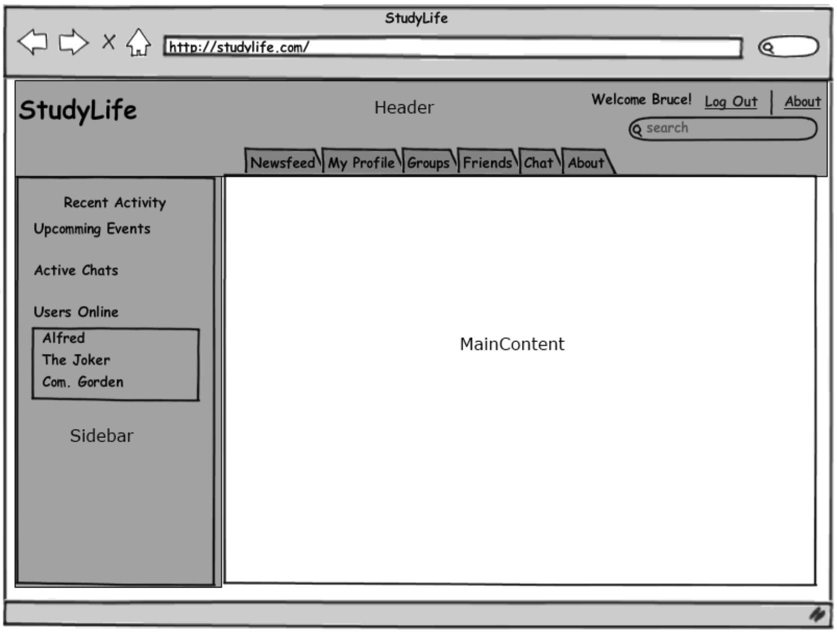


**Figure 6.4:** Master layout mock-up.

#### Header

The header, at the top of the page, should contain the title, menu, search field and login/logout links.

Below the title, there should be a tabbed-menu. The tabbed-menu should consist of all the common links for a user, e.g. profile page, groups, file sharing, etc. In case a tab is hovered by the mouse, the tab is highlighted and in case the tab link is clicked it will show the specific page and have a different color than the other tabs. Next to the menu a search field and a link to login/logout is placed.

#### Sidebar

The left part of the layout should contain short lists of upcoming events from the user's calendar, active chats the user participated in, awaiting requests and a list showing the users that are online. This content is displayed on all pages once the users is logged in. To satisfy the requirement that the master layout should provide instant feedback on activity in StudyLife relevant to the user, the sidebar should be made to update continuously. That is, the sidebar should update without the user having to reload the page. To implement such an update feature, AJAX (see Section 4.4) could be used.

The content from other components to be shown in the sidebar would be provided as static *partial views*. This means that the sidebar could not be rendered client-side as this would need an agreed on representation of the data to be displayed. For this to be possible the content returned on an AJAX request should be raw data, that would then be rendered client-side using JavaScript. For this reason the sidebar should be rendered server-side and inserted into the sidebar container.

#### Main Content

The grey area on Figure 6.4 is content that is to be the same on all pages in the system where a user is logged in. The main content area can be changed as needed for the given page. This is where the bulk of StudyLife functionality is to be displayed.

### 6.2.3 Structuring Static and Dynamic Content

To be able to have both static and dynamic content in the master layout, some sort of method of reusing content in multiple *views* would reduce redundancy in the system. Such method exists in ASP.NET and is called Master Pages. A master page defines a layout by inserting placeholders into a layout made of XHTML markup containing static content. *Views* then inherit from this Masterpage and fill out the placeholders with their dynamic content. Master pages hereby allows for static content like menus, footer and header to be reused in multiple *views*, thus reducing redundancy in the code.

A simple alternative to master pages could be to create a common stylesheet to get a consistent look and structure. Each view would then add static content like navigation by inserting a user control. This solution would introduce redundancy in the code.

## 6.3 Web Security Measures

Designing security components for a system, is often a very important process, as the system may contain sensitive data that can be abused. Since security of StudyLife is not of highest priority, cf. Section 2.4, it has still been thought of and enhanced as much as time allowed.

This section contains descriptions of how authentication and authorization could be designed and how they ended up being designed.

### 6.3.1 Authentication Approaches

The purpose of authentication is to identify a user of the system, as well as make it possible for new users to register themselves.

Alternative methods have been considered when designing authentication. The methods discussed in this project were: Authentication made from scratch, OpenID and ASP.NET Authentication. The following describes the methods and points out their pros and cons.

#### Authentication from scratch

By developing authentication from scratch would be very educational and challenging. It could be done using HTML forms and cookies. In addition, a lot of security issues revolving around authentication of users needs to be handled.

However, since this project focused on reliability and correctness (cf. Section 2.4) this method was thought as being too overwhelming considering the limited time available.

### OpenID

OpenID is an open, decentralized authentication standard which can be used for access control, permitting users to log on to various services with the same digital identity [Ope10]. It is a rather new authentication service, but large companies such as Google and Yahoo already support it.

An advantage of using OpenID is that once implemented, users can skip the registration process and login with their OpenID. There would however still need to be a registration phase on first login that ties a StudyLife user to the provided OpenID.

A disadvantage of using OpenID would be that designing a user model in accordance with the contract would add unneccesary complexity.

### ASP.NET Authentication

As of version 2.0, ASP.NET includes authentication functionality. ASP.NET implements authentication through a provider, which is either the Windows Authentication Provider or the Forms Authentication Provider [Mic10a].

The Windows Authentication Provider makes it possible to use the Windows OS authentication to secure the ASP.NET application [Mic10a].

The Forms Authentication Provider allows creation of an application-specific user component that can be authenticated either by using ones own code or by using the ASP.NET membership provider. The ASP.NET membership provider includes functionality to collect users credentials and information, and authenticate them [Mic10a].

As ASP.NET MVC was chosen as development framework, one of the advantages of using this solution is the fact that it is part of the framework, which increases consistency of the code and eases implementation.

By focusing on ease of development, and hereby rapid deployment, the ASP.NET Authentication solution was chosen.

## 6.3.2 Authorization

Authorization was included in StudyLife to increase security, by keeping unauthorized users out of the system, utilize the privacy settings in the components, and assure that registered users' private information is not accessible from outside the system.

In the design of authorization two solutions were discussed; make it from scratch as with authentication or make use of the authorization in ASP.NET.

The first solution, make authorization from scratch, could be done by checking if the user is logged in on every page, such that no matter what page is visited, an authentication check is done.

On the other hand, as ASP.NET was already chosen as framework and to be used for authentication, it was decided to use it for authorization as well. ASP.NET has a builtin authorization module, which makes it easy and reliable to add authorization to specific pages in StudyLife. Listing 6.1 shows how authorization can be added to an *action*.

```
    [Authorize]
    public ActionResult Index()
    {
        return View();
    }
```

**Listing 6.1:** Example of a *controller* action with authorization property

In Listing 6.1 the `[Authorize]` attribute only allows authenticated users to access the index *view*.

The following describes how authorization will work regarding a visitor trying to access the system, a user trying to access another user's profile and group privacy.

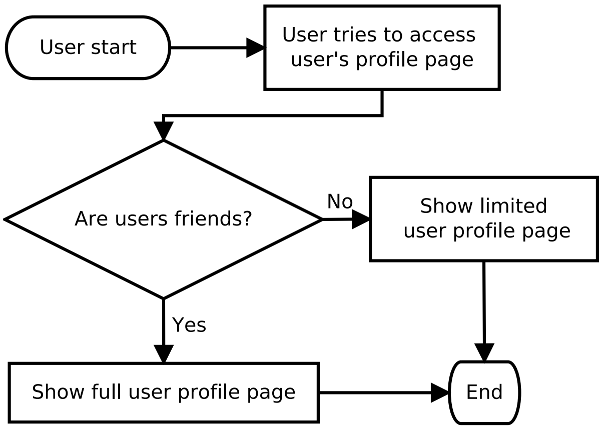

User start
User tries to access user's profile page
Are users friends?
No
Show limited user profile page
Yes
Show full user profile page
End

**Figure 6.5:** Flow chart of a user accessing another user's profile.

If a user has a friend relationship with another user, he may see personal information as illustrated on Figure 6.5. In case the two users are not friends, a limited profile page will be shown, that is a profile page consisting of only name and a link for requesting friendship.

The privacy setting in the group component is carried out by checking if the group is private and if it is then check if the user is a member of the group. The flow chart on Figure 6.6 illustrates the flow of a user clicking a private group.

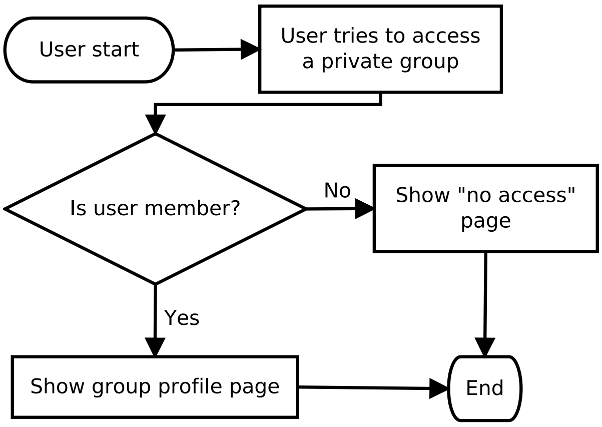


**Figure 6.6:** Flow chart of a user accessing a private group

## 6.4 User Component

The design of the User component, is made on basis of the requirements defined in Section 5.7. Figure 6.7 displays the contents of the User component.

As StudyLife was developed using ASP.NET MVC, as written in Section 4.1, and therefore used the MVC pattern, the following describes how the *handlers*, *models*, *mappings*, *controllers* and *views* were designed. Furthermore, this section describes the design of the online status of users.

### 6.4.1 Models

The User component contains two *models*; the `User` class models users and the `OnlineStatus` class *models* the time for users' last activity. Figure 6.8 shows the properties of the `User` class that is to be used for modelling users in the system, derived in Section 5.3 in the analysis.

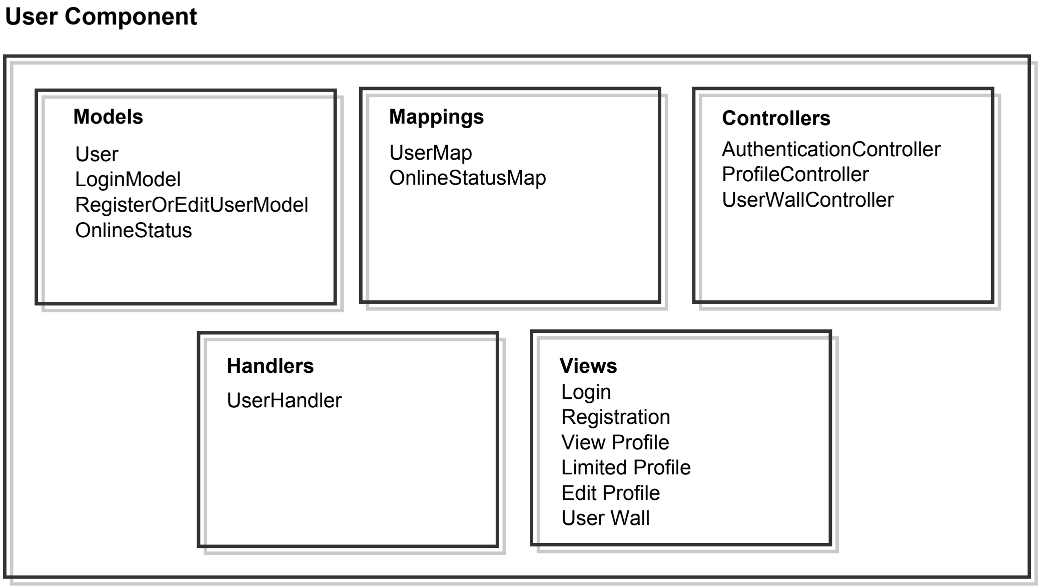


**Figure 6.7:** The User component.

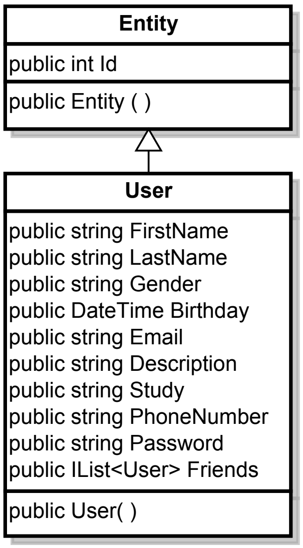


**Figure 6.8:** The `User` class.

The `Id` of `User` is derived from the `Entity` class. The properties in the `User` class are mostly self-describing, except for `Email`, which was meant as the user's unique username, and `Friends` that was a list containing all the friend relationships of a specific user.

The `OnlineStatus` class is used in the online functionality that enables displaying users currently online in the system. Figure 6.9 shows the properties of the `OnlineStatus` class.

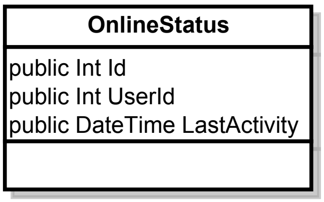


**Figure 6.9:** The `OnlineStatus` class.

The `OnlineStatus` class contains an id, a user id and a timestamp denoting the time of the users' last activity in the system.

### Mappings

The User component contained two mappings; `UserMap` for mapping the `User` class and `OnlineStatusMap` for mapping the `OnlineStatus` class.

`UserMap` should map the properties of the `User` class defined in Figure 6.8. As the user ID is mapped in the `Entity` super class, all but the `Friend` property can be mapped using the `Map` method.

`OnlineStatusMap` should use the `Id` method to give entries a unique, auto-incrementing identifier. The `Map` method should be used to map the the `UserId` and `lastActivity` properties.

## 6.4.2 UserHandler

The purpose of the UserHandler was to provide an interface for retrieving and saving data regarding the user *model* and other interfacing to the User component. Figure 6.10 shows and UML diagram of the contents of the `UserHandler`.

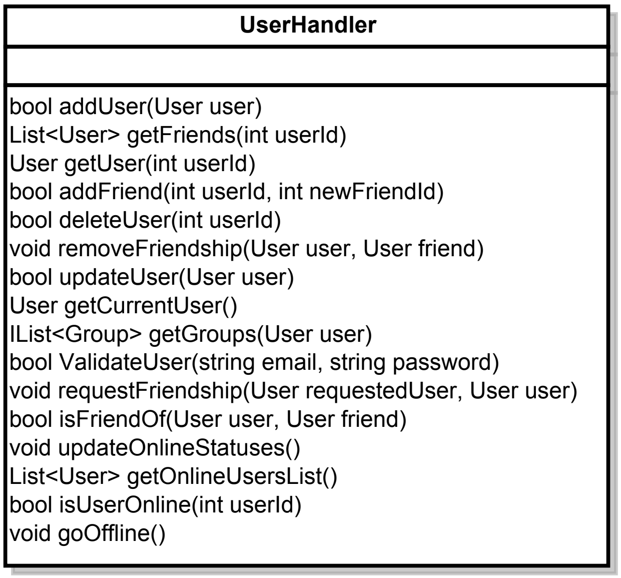

UserHandler
bool addUser(User user)
List<User> getFriends(int userId)
User getUser(int userId)
bool addFriend(int userId, int newFriendId)
bool deleteUser(int userId)
void removeFriendship(User user, User friend)
bool updateUser(User user)
User getCurrentUser()
IList<Group> getGroups(User user)
bool ValidateUser(string email, string password)
void requestFriendship(User requestedUser, User user)
bool isFriendOf(User user, User friend)
void updateOnlineStatuses()
List<User> getOnlineUsersList()
bool isUserOnline(int userId)
void goOffline()

**Figure 6.10:** `UserHandler` UML.

Description of the induvidual methods in the UserHandler follows:

**`addUser`** Stores a new user. Return true if the email provided in the User object provider not in use and false if it is.

**`getFriends`** Retrieves frfiends by user id.

**`getUser`** Retrieves a specific user by entity id or email.

**`addFriend`** Creates a new friendship relation between two users. Returns false if the friendship already exists.

**`deleteUser`** Deleltes a specific user. Returns false if the user does not exist.

**`removeFriendship`** Removes a friendship relation between two users.

**`updateUser`** Updates a user. Return true if the email provided in the User object provider not in use and false if it is used by another user.

**`getCurrentUser`** Retrieves the user that is currently logged in.

**`getGroups`** Retrieves the groups for a specific user.

**`validateUser`** Determines whether the given email and password belongs to any registered user.

**`requestFriendship`** Creates a friendship requrest.

**`isFriendOf`** Checks if two users are friends, and returns true in case the users have a friend relationship.

**`getSHA1`** Is used to hash user passwords using the SHA1 algorithm.

**`getSalt`** Returns the input, but with an abitrary value concatinated.

**`updateOnlineStatus`** Is used to hash user passwords using the SHA1 algorithm.

**`getOnlineUseres`** Is used to Retrieve all the users that are currently online.

**`isUserOnline`** Return true of the the specified user is online.

**`goOffline`** Is used when user explicitly logs off.

### 6.4.3 Controllers

The User components contains three *controllers*; `AuthenticationController`, `ProfileController` and `UserWallController`.

The `AuthenticationController` should include functionality for registering new users, login and logout. As described in Section 6.3, authentication should be implemented using ASP.NET Authentication.

The `ProfileController` should include *actions* for editing and displaying user profiles.

The `UserWallController` should simply return the wall *view* containing the user's wall.

### 6.4.4 Views

As can be seen on Figure 6.7, the User component contains six different *views*.

#### Login View

When a user visits StudyLife, he is met with a login form, containing an email and a password field. Figure 6.11 shows a mock-up of the login page.

If the user is already registered, he can log in using his email and password to authenticate himself and access the system.

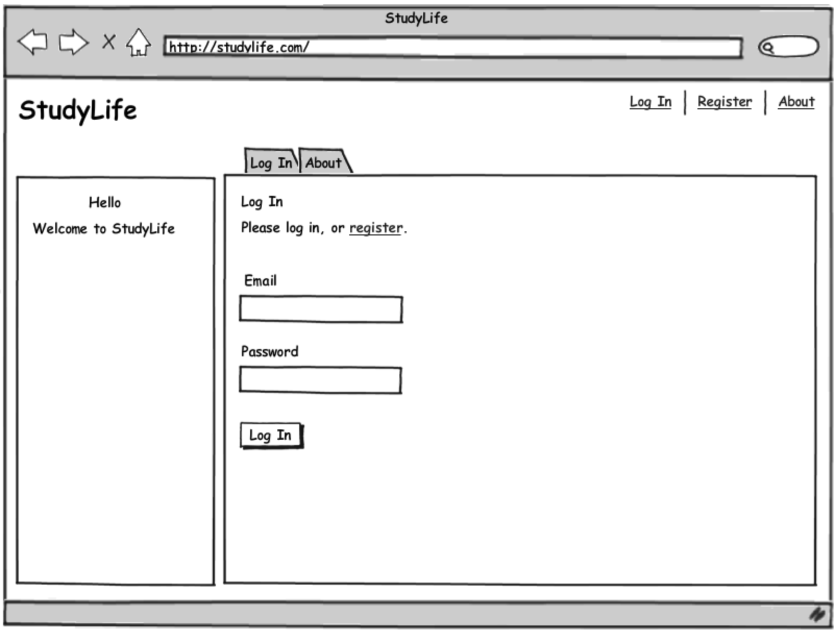
No matter what part of the system the user is currently using, he always has the option of logging out, by pressing a link labeled "log out" in the upper right corner of the page.

#### Registration View

If a visitor visits StudyLife he can create one by clicking the register link from the login form, and fill out the registration form.

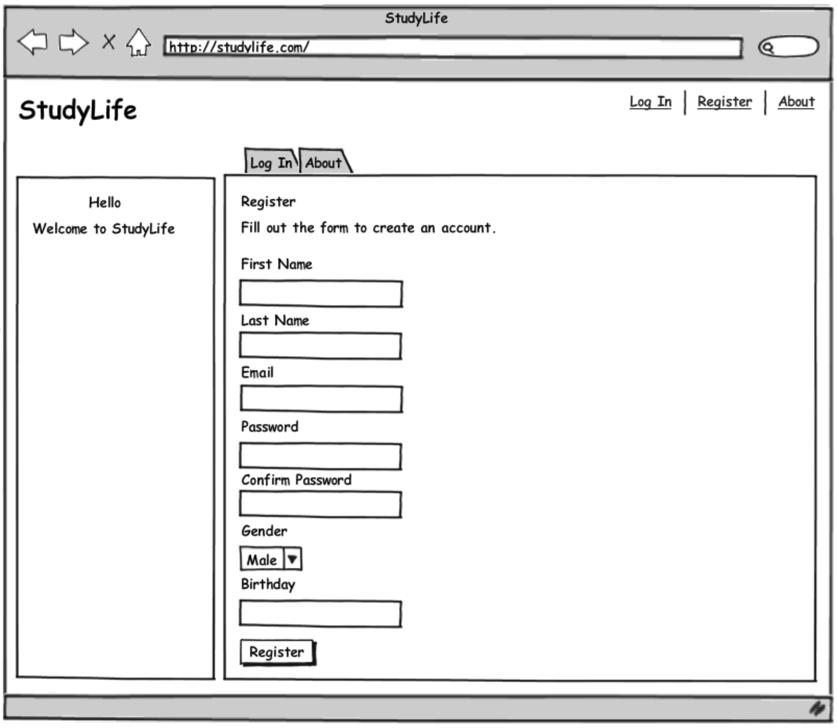
Figure 6.12 shows a mock-up of the registration form.

When the user fills out the registration form, he provides the basic information needed for the user to be able to start using the system.



Please log in, or register.

**Figure 6.11:** Login page mock-up



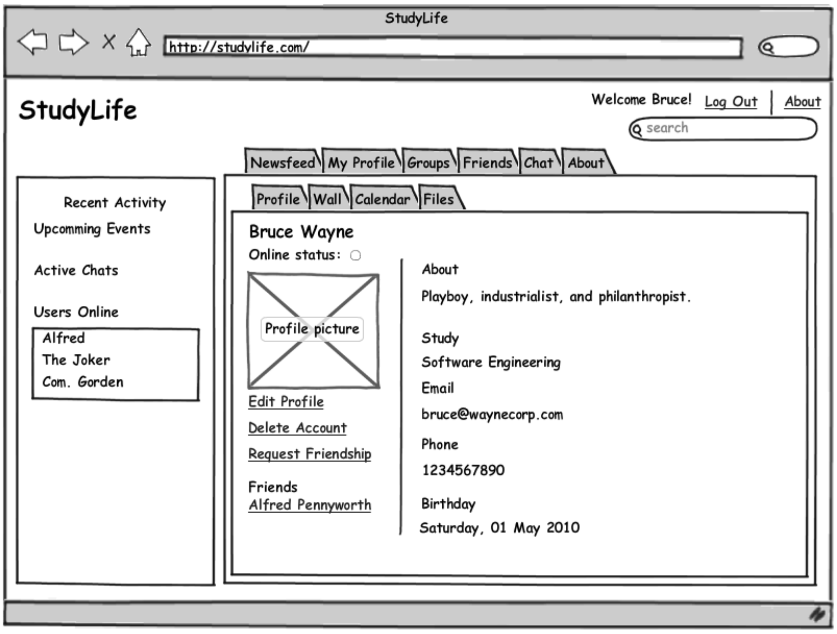
Fill out the form to create an account.

**Figure 6.12:** Registration page mock-up

### View, Edit and Limited Profile views

A user's profile is displayed using either the *view* or limited profile *views*. Which *view* is presented depends on whether the user has friend relationship with the user which profile he is viewing. If they are friends, the view profile *view* is displayed, containg all the information on the user. If they are not friends, the limited profile *view* is displayed, containg only a profile image, the users name and a link to request a friendship.

The view profile *view* is illustrated in Figure 6.13 as a mock-up. The edit and limited *views* should follow the same design as Figure 6.13.



Bruce Wayne
Online status: ○

About
Playboy, industrialist, and philanthropist.

Study
Software Engineering

Email
bruce@waynecorp.com

Phone
1234567890

Birthday
Saturday, 01 May 2010

**Figure 6.13:** User profile page mock-up.

The edit profile view should contain forms equivalent to the user model, making it possible for a user to update his personal information.

### User Wall View

This *view* should include a *partial view* from the Wall component, showing the users Wall.

## 6.4.5 Online Status

As described in Section 5.3.1, users should be able to see which other users are online in the system at any given time. The users currently being online could be listed in the sidebar, making it visible no matter which page the user is visiting. Besides the sidebar panel, online status should also be indicated by a colored bullet point on the user profile page and in listings of users. A red point means that the user is offline, and a green point means that the user is online. This is illustrated in Figure 6.13. The terms online and offline has to be properly defined for implementation:

- A user is online when he logs into the system and has been active in the last three minutes.
- A user is offline when he logs out, or when he has been inactive for three minutes.

This could be solved by having a list of online users as a table in the database. The functionality that updates the online list could be added to the configuration of the system, enabling it to be invoked at every request (to any component). As stated in Section 6.2.2, the sidebar would be continuously updated with AJAX, hence if the user is on the site for more than three minutes without performing any activity he will not be marked as online, as the AJAX request updates the online list on every request.

To sum up, online functionality should be visible on the users profile and in the sidebar. A list of online users should be stored in the database with a user `Id` and the time of his last activity. Users being inactive for more than three minutes should be considered as offline. The online list is updated at every request to the web-server.

## 6.5 Friend Component

The relationship between users, called friends, was designed to comply with the requirements found in Section 5.4. Figure 6.14 shows how the friends component was split into *views* and *controllers*. As a friend was another user, the user *model* was used as *model* for friends as well. The `User` class then had a `List<User>`, i.e. a list of users, called `Friends`.

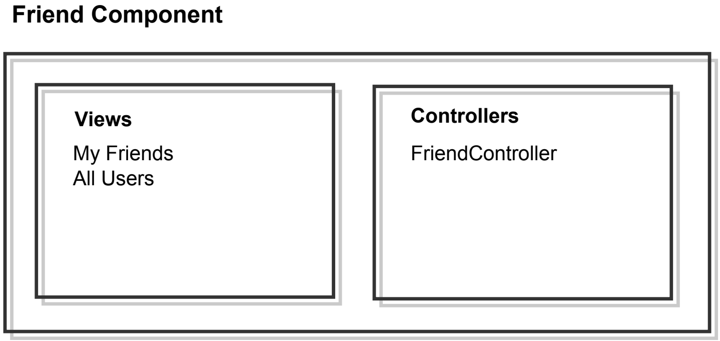


**Figure 6.14:** The friend component

The following sections describes the component on the basis of Figure 6.14.

### 6.5.1 Views

The first *view* for the Friend component called “My Friends” lists the `Friends` property of a supplied `User`. From here it should be possible to click the friends’ names and be directed to their respective profile pages. The table should also include a link to remove friendships. Another *view* called “All Users” should list all the current users in the StudyLife system.

### 6.5.2 Controllers

The *controller* for friends should include the methods for showing, requesting, adding and removing friendships.

## 6.6 Group Component

The Group component handled all functionality regarding groups in StudyLife. The contract contained an overall design of the Groups component, but a more thorough design was needed in order to implement it.

Figure 6.15 illustrates how the component was split in five parts. The following contains a description of each part.

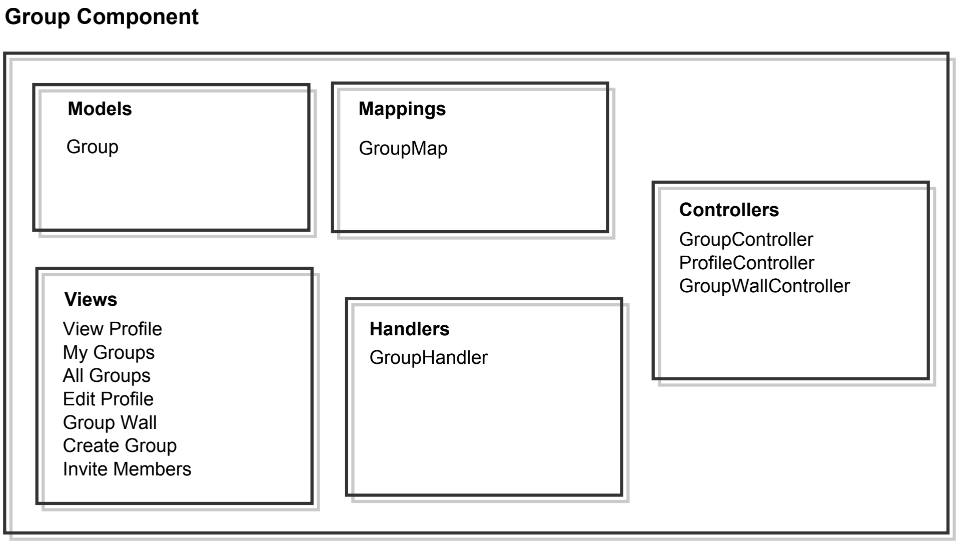


**Figure 6.15:** The group component

### 6.6.1 Models

The Group *model* is a class that is used to model groups in the system. It inherits from the Entity class for the sames purposes as for the `User` class. Figure 6.16 is an UML showing the Group *model*.

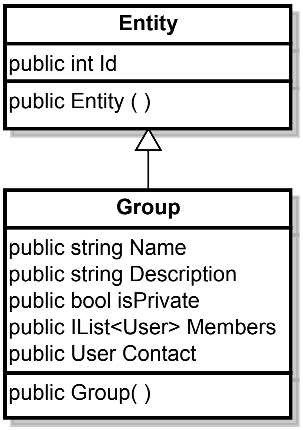


**Figure 6.16:** The Group model

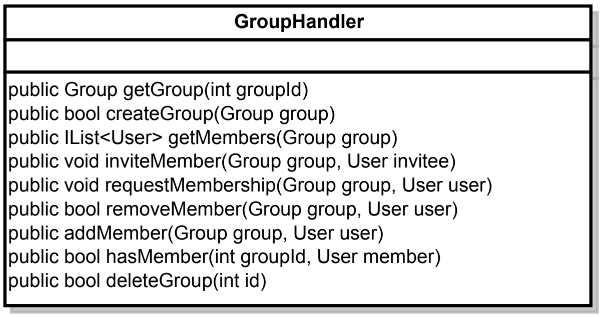
As can be seen on the figure, the Group *model* has a set properties. `Name` and `Description` are self-explanatory. `IsPrivate` is a boolean that denotes whether the group is a private or public. `Members` is a list of Users, that are members of the group. `Contact` is the User whom will get all join requests for the group. The contact can be changed to any of the Group's members on the edit group page. If a contact leaves a group then the group is deleted, with a warning being shown prior to the deletion.

### 6.6.2 Mappings

The mapping class created for Groups should map the properties of the Group *model* to the database. As written in Section 4.2.2 the `HasManyToMany` can be used for the `Members` property and the `Map` can be used for the rest.

### 6.6.3 Handlers

The GroupHandler class is very similar to the UserHandler class. It is a class containing a number of methods that provide an interface for fetching, saving, altering and deleting data regarding groups. The GroupHandler was useful because it reduced some redundancy. Figure 6.17 shows and UML diagram of the contents of the GroupHandler.



```
GroupHandler

public Group getGroup(int groupId)
public bool createGroup(Group group)
public IList<User> getMembers(Group group)
public void inviteMember(Group group, User invitee)
public void requestMembership(Group group, User user)
public bool removeMember(Group group, User user)
public addMember(Group group, User user)
public bool hasMember(int groupId, User member)
public bool deleteGroup(int id)
```

**Figure 6.17:** The GroupHandler class

The following describes each method of the GroupHandler.

**getGroup** retrieves a group from the database.

**createGroup** is used to add new groups to the database.

**getMembers** retrieves the members of a group.

**inviteMember** is used to invite a new member to a group.

**requestMembership** is used to request a membership of a group.

**removeMember** removes a user from a group.

**addMember** adds a new member to a group.

**hasMember** is used to check if a specific user is member of a specific group.

**deleteGroup** removes a group from the database.

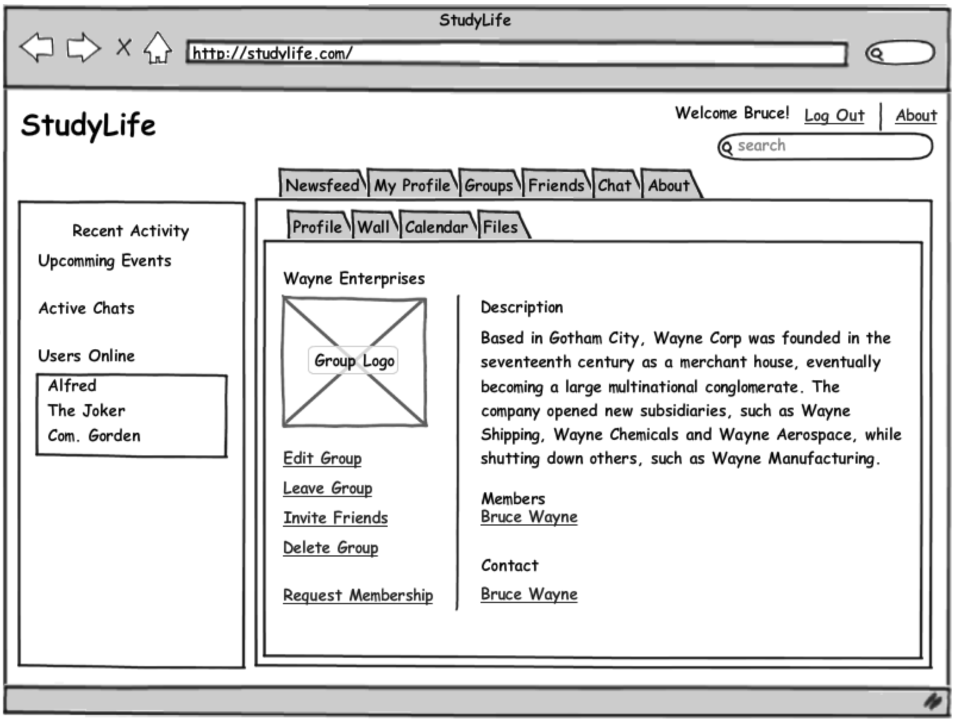
As the UserHandler, the GroupHandler is responsible for notifying the ComponentAnnouncer about group creating and deletion.

### 6.6.4 Views

#### Group Profile Page

The purpose of the group profile page was to give an overview of general information about a group, such as name, description, members etc. At the profile page, the user may have a number of options dependent on if he is a member of the group and if the group is private. If he is a member, he may edit the group, leave the group or invite new members. In case the user is not already a member of the public group, he is presented with a request membership option, which, if pressed, links to an action that sends a request to the contact person of the group. In case the group is private and the user is not member, he is redirected to an "no access" page.

A mock-up of the group profile page can be seen in Figure 6.18.



Wayne Enterprises

Description

Based in Gotham City, Wayne Corp was founded in the seventeenth century as a merchant house, eventually becoming a large multinational conglomerate. The company opened new subsidiaries, such as Wayne Shipping, Wayne Chemicals and Wayne Aerospace, while shutting down others, such as Wayne Manufacturing.

Members
Bruce Wayne

Contact
Bruce Wayne

**Figure 6.18:** Group profile mock-up

### My/All Groups Page

When a registered user clicks the Groups tab in the master layout, he is directed to the “My Groups” page containing a table with names, descriptions and leave-option for the groups he is member of. The titles of each of the groups are links to the Group profile pages.

At the “My Groups” page there is a link to the “All Groups” page which list of all the groups in the system, and there is a link to the “Create Group” page.

### Create Group Page

This page contains a form where the user can fill out information for creating a new group by entering a name, a description and an option for the group to be private or not. The only way to become member of a private group is to be invited; one cannot request a membership. When a user creates a new public group, he automatically becomes a contact for the group, which means all membership requests will be sent to him.

### Group Wall

The Group wall page will display the wall specific for the current group. The wall was developed by another group part of the multi-project, and should be included as a partial view.

### Edit Group Page

The edit Group page allows any group member to alter information about a group, such as name or contact. This should be done by having an edit field for all `Group` properties.

#### Invite members Page

The invite members page contains a list of all the user's friends that are not already a member of the particular group. This includes an option for inviting each person on the list to the group.

### 6.6.5 Controllers

There are three *controllers* for the Group component. These are `GroupWallController`, which should do exactly the same as the `UserWallController` i.e. return the *view* of a group's wall, `ProfileController` and `GroupController`. The `ProfileController` should include an action for showing the Group profile page, and the `GroupController` should contain actions for all the major group functionalities, e.g. creating, editing and deleting groups.

## 6.7 Search Component

The requirements for the Search component of StudyLife states that searching should be available on all pages in a consistent location and search results should be categorised. This section describes how these requirements could be fulfilled. After the alternatives are covered, a search method is chosen and the contents of the Search component is described.

### 6.7.1 Including Search On All Pages

To ensure that the search field is included on all pages in a consistent location it is included in the master layout. As stated in Section 6.2 the master layout includes a header that is displayed on all pages. The search field is inserted into this header making it visible on all pages and in a consistent location.

### 6.7.2 Search Methods

A search is performed server-side, invoked by a query from the user, and the result is sent back to the user. Figure 6.19 depicts a search in StudyLife.

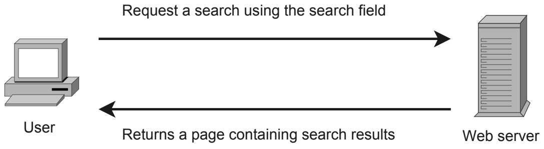

Request a search using the search field
User
Returns a page containing search results
Web server

**Figure 6.19:** Server-side search.

The user performs a search by entering a query into the search field at the top of the page. When the user submits the search query, the server perform the search and returns the results displayed on a web-page.

Having chosen to perform searches server-side, opens up a set of possibilities on how to perform the actual search in users, groups, events and files. The search query is a string without any keywords or boolean logic. The following describes three solutions for performing the search: Lucene, database search and SQL Server Full-Text Search.

#### Lucene

Lucene is an information retrieval library that was originally written in Java, but is also ported to other languages including .NET. The Lucene library provides a simple API for indexing and searching full-text [Apa10]. The indexing part of Lucene creates index tables in the database for all searchable data to make searches faster and more efficient. The searching part searches these indexes for a given query [HGM09].

A package called Fluent NHibernate. Search has recently (8th of March 2010) been released and it provides an interface for Lucene.NET (NHibernate.Search). The project is however at an early beta stage and would require extensive modifications in the Data Access Layer.

To use NHibernate.Search would require defining index mappings for content that should be searchable, i.e. a seperate mapping for each model denoting searchable properties. Listing 6.2 shows a simple search using NHibernate.Search.

```
var posts = NHibernate.Search.Search.CreateFullTextSession(session)
        .CreateFullTextQuery<Post>("Title:foobar")
        .List<Post>();
```

**Listing 6.2:** NHibernate.Search (Lucene) search example

### Database Search

Searching can also be done directly on the database using the Fluent NHibernate ORM. Fluent NHibernate will generate SQL queries that contain the `LIKE` condition. The following show an example search in Fluent NHibernate.

```
var posts = repository.FetchListByCriteria<Post>(x => x.Title.Contains("foobar")).
```

**Listing 6.3:** Fluent NHibernate search example

The actual search query of Listing 6.3 is made up by a LINQ (Language Integrated Query) expression passed to the `FetchListByCriteria` method, which is provided by the Data Access Layer. This enables building search queries in a more intuitive language than SQL using the relevant models properties. Listing 6.4 shows a SQL query corresponding to 6.3.

```
SELECT * FROM post WHERE title LIKE ’%foobar%’
```

**Listing 6.4:** SQL search example

The % signs in Listing 6.4 denotes wild cards, meaning *foobar* just needs to be somewhere in the field being searched.

### SQL Server Full-Text Search

Microsoft SQL Server 2008 provides functionality for full-text search on many different data types. SQL Server Full-Text Search uses the same approach as Lucene by building indexes for data and then searching them instead of the actual data. The search is performed by using a several specific clauses. Listing 6.5 show an example of using SQL Server Full-Text Search.

```
SELECT * FROM post WHERE CONTAINS(title, ’foobar’)
```

**Listing 6.5:** SQL Server Full-Text Search example.

## 6.7.3 Chosen Search Method

While Lucene is great for improving search in full-text, most of the data in StudyLife will consist of a few sentences, hence the performance gain using Lucene will be fairly small. Furthermore, Fluent NHibernate.Search is at a very early beta stage (currently at version 0.3) and poorly documented.

Fluent NHibernate does not support SQL Server Full-Text Search to perform search queries on the database. Hence, using SQL Server Full-Text Search would require either a modification of Fluent NHibernate or to circumvent it connecting directly to the database to perform SQL queries. Modifying

Fluent NHibernate is out of the scope of this project, and accessing the database directly (circumventing the database component) would violate the architecture agreed on in the contract.

Performing searches on the database using Fluent NHibernate was the chosen search method for StudyLife. This method allows for easily writing the expressions in C# using the LINQ expressions.

### 6.7.4 Search Component Contents

With the search method chosen, the contents of the search component was defined. Figure 6.20 shows the content of the Search component.

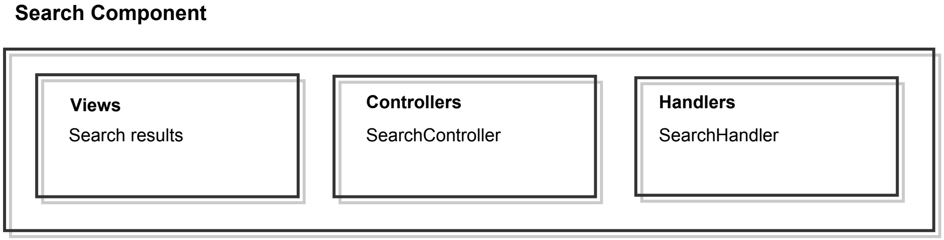


**Figure 6.20:** Search component contents.

To implement the search component using the database search method, a *view* for the search results, a *controller* for handling the search request and a *handler* for performing the actual search, is needed. Figure 6.21 is a diagram showing how a search is performed in the search component.

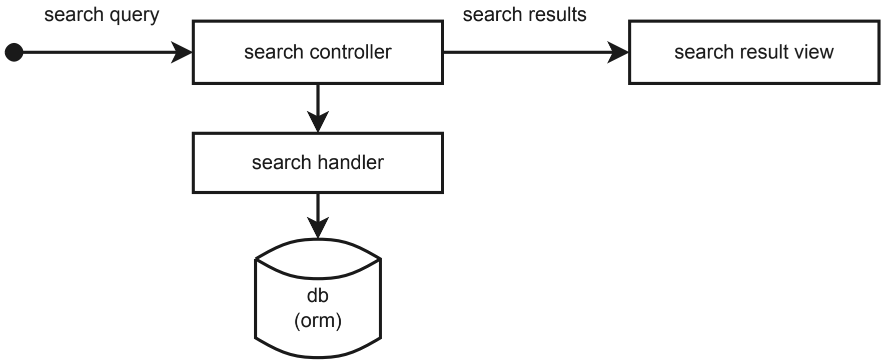


**Figure 6.21:** Search diagram

The search *controller* handles the request, calls the search *handler* and returns the search result *view*. The search result *view* will contain the search results categorised in separate tables. The search *handler* performs the search on the database by means of statements similar to Listing 6.3. Figure 6.22 shows an UML diagram of the `SearchHandler` class.

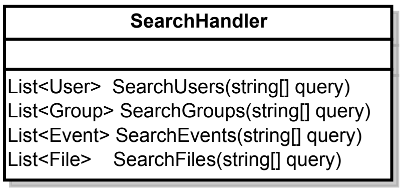

SearchHandler
List<User>  SearchUsers(string[] query)
List<Group> SearchGroups(string[] query)
List<Event> SearchEvents(string[] query)
List<File>    SearchFiles(string[] query)

**Figure 6.22:** UML of the `SearchHandler` class

## 6.8 Summary

Throughout the design chapter, various solutions for satisfying the requirement specification have been covered. Section 6.1.1 introduced the entity model which was to be the superclass for the user and group models. In Section 6.2 it was decided to use various UI patterns, common color palette and dynamic and static content to form the master layout of StudyLife. In Section 6.3 it was decided to use ASP.NET built-in functionality for authentication and authorization of users. Furthermore, Section 6.3 defined user and profile access rights for visitors and users. Sections 6.4 through 6.6 defined the contents in terms of *models*, *mappings*, *views*, *controllers* and *handlers* for the user, friend and group components. In Section 6.7 database search was chosen as search method for the system.

# Chapter 7

# Implementation

After designing the master layout, User, Group and Search components, implementation is the next step to realize the application. The implementation was carried out in ASP.NET MVC, as decided in Section 2.6. Some components have been implemented before others due to other multi-project groups depending on them.

Deciding the order of implementation was done during Sprint planning as described in Section 3.2.1. Figure 7.1 shows this plan, which was split in four Sprints. In the first Sprint the User model and authentication were implemented, since these were components to which the other groups had dependencies. In Sprint two and three the group, friend and the rest of the user component was implemented, and finally in the last Sprint the search and online status were implemented.

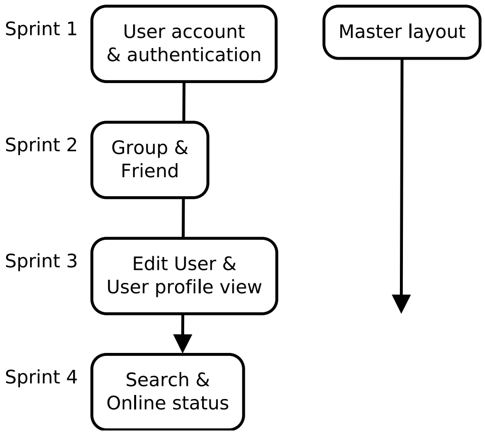


**Figure 7.1:** Implementation plan

In general each component was implemented by first writing the models and the mappings, if any. Then the *controllers*, *handlers* and *views* were created and implemented in terms of content and logic.

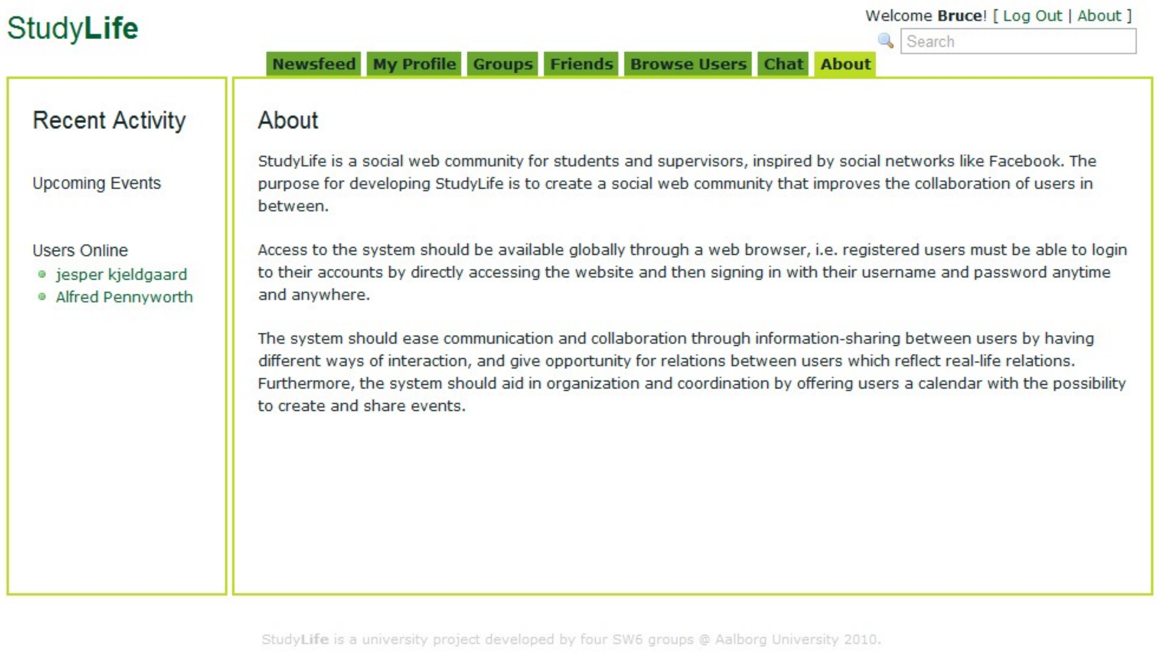
The following explains how the components were implemented to comply with the design decisions.

## 7.1 Master Layout

The master layout implementation consisted of a style sheet and a master page. The style sheet defined the layout, typography and coloring of text and borders. Including this style sheet on all pages enabled for a consistent visual theme of StudyLife. Figure 7.2 shows a screen shot of the about page in StudyLife.

The master page contained three placeholders that pages could insert content in.

- **TitleContent:** This placeholder was placed in-between `<title>` tags to enable pages to change the title of the page accordingly.
- **CssAndJavascriptIncludes:** This placeholder was placed in-between the `<head>` tags to allow pages to add JavaScript, CSS and meta information to the head region of the page.



About

StudyLife is a social web community for students and supervisors, inspired by social networks like Facebook. The purpose for developing StudyLife is to create a social web community that improves the collaboration of users in between.

Access to the system should be available globally through a web browser, i.e. registered users must be able to login to their accounts by directly accessing the website and then signing in with their username and password anytime and anywhere.

The system should ease communication and collaboration through information-sharing between users by having different ways of interaction, and give opportunity for relations between users which reflect real-life relations. Furthermore, the system should aid in organization and coordination by offering users a calendar with the possibility to create and share events.

StudyLife is a university project developed by four SW6 groups @ Aalborg University 2010.

**Figure 7.2:** The About page.

- **MainContent:** This placeholder was placed in-between the `<body>` tags and are used for the content specific of each page.

The master page contained content that was displayed on all pages of StudyLife. The following content was included in the master page:

- A header including the StudyLife title, the navigation menu, search field and login/logout links.
- A sidebar displaying recent activity.
- A footer with credits.

The following describes the search field and the sidebar.

### 7.1.1 Sidebar

The sidebar was positioned to the left of the main content and contained recent activity in StudyLife relevant to the user. The sidebar consisted of four *partial views*, requests, upcoming events, active chats, online users, which were included using the `RenderPartial` helper.

The sidebar content was refreshed every five seconds using server-side rendering AJAX as described in Section 4.4. Listing 7.1 shows the code that performs the update of the sidebar.

```
<script type="text/javascript">
function ajaxRequest() {
  $.ajax({
    url: '/sidebar/index',
    cache: false,
    success: function (data) {
      $('#ajaxSidebarPartials').html(data);
    }
  });
}
  ajaxRequest();
  window.setInterval("ajaxRequest()", 5000);
```

```
</script>
<div id="ajaxSidebarPartials">
...
// here the partial views will be inserted.
...
</div>
```

**Listing 7.1:** AJAX that updates the sidebar

The AJAX request was performed using the JQuery JavaScript framework. An advantage of using JQuery is that it prevents some cross-browser compatibility problems regarding JavaScripts. The URL "/sidebar/index" directed to the `SidebarController` which returned the sidebar *view*. The `cache` option forces the browser to not use a cached version of the content. The content of the `ajaxSidebarPartials` div tag was replaced with the data (updated sidebar) received from the AJAX request.

The `setInterval` function called the `ajaxRequest` function every 5000 milliseconds (5 seconds).

### 7.1.2 Search Input Field

The search input field at the top of the page showed, per default, "Search" in gray. When the user clicked the input field it cleared, and was ready for a search query to be entered. If the field was left empty, the gray "Search" text reappeared. This functionality was added to remove the need for a separate label denoting what the input field was for. Figure 7.3 shows the different states of the search input field.

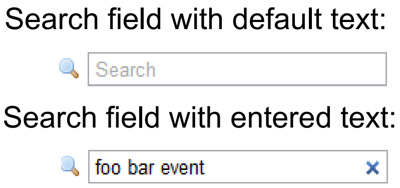

Search field with default text:
Search
Search field with entered text:
foo bar event

**Figure 7.3:** Search input field states

During the refactoring the search field on top of the page was optimized to use a new HTML 5 tag. Prior to refactoring, the search field required 20 lines of JavaScript to get the desired effect. Using the new `placeholder` tag from HTML 5 no JavaScript was needed, hereby reducing the amount of code on all pages in StudyLife. Listing 7.2 shows how the `placeholder` tag is used on input fields. Another HTML 5 feature used was the type of the input field being set to search. This enabled a clear button (see Figure 7.3) on the input field when text was inserted.

```
<input type="search" name="search" id="search" placeholder="Search" />
```

**Listing 7.2:** HTML 5 placeholder tag example

## 7.2 User Component

The implementation of the User component was made on the basis of Section 6.4. The following describes how the User *model*, UserHandler and user profile, authentication, authorization, and online status were implemented.

### 7.2.1 Models

This section describes the implementation of the *models* in the User component.

### User Model

The User *model* was a class that was used to represent a user in the system, which inherits the entity class. To meet the requirements, the user *model* had a number of properties, namely `FirstName`, `LastName`, `Gender`, `Birthday`, `Email`, `Password`, `Description`, `Study`, `PhoneNumber` and `Friends`. Listing 7.3 is an extract of the user *model*, that shows some of these attributes.

The majority of the properties in the user class had special ASP.NET MVC attributes attached, which were used in relation to forms; for instance to perform form validation. An example of this could be the `Required` attribute which fails validation if the field for a given a property is not filled. As can be seen on Listing 7.3, the `Email` and `Password` properties had custom validation attributes, being `DataType.EmailAdress` and `ValidatePasswordLength` respectively. The `Email` attribute ensured that the email string is of the correct format, i.e. example@email.com. The `ValidatePasswordLength` attribute ensures that the password string is at least 6 characters long.

```
public class User : Entity
{
  [Required(ErrorMessage = "*")]
  [DisplayName("First Name")]
  public virtual string FirstName { get; set; }
  ...
  [Required(ErrorMessage = "*")]
  [DisplayName("Email")]
  [DataType(DataType.EmailAddress)]
  [Email(ErrorMessage = "Invalid Email Address")]
  public virtual string Email { get; set; }
  ...
  [Required(ErrorMessage = "*")]
  [ValidatePasswordLength]
  [DataType(DataType.Password)]
  [DisplayName("Password")]
  public virtual string Password { get; set; }
  ...
  public User() { }
}
```

**Listing 7.3:** An extract of the user model class

### RegisterOrEdit Model

The `RegisterOrEdit` *model* inherited the User *model* with a `ConfirmPassword` property. It also had a custom validator for requiring that the password and the confirm password were identical.

### Login Model

A *model* was created for the purpose of storing the data sent between the `LogIn` action and the *view*. It contained `Email`, `Password` and `RememberMe` properties. All three had the `Required` validator.

### OnlineStatus Model

The `OnlineStatus` *model* was used to store the online status of a `User`. It contained an `Id` for identifying a specific `OnlineStatus`, a `UserId` for binding the `OnlineStatus` to a specific `User` and a `LastActivity` for showing the time and date of a user's last activity.

## 7.2.2 Mappings

This section describes how the specific User mappings were implemented.

### UserMapping

The mapping for the User *model* inhereted from the generic class `SubclassMap` used for mapping subclasses in NHibernate. All properties except `Friends` were mapped using the map method. `FirstName`, `Lastname`, `Gender`, `Email` and `Password` where also not nullable. Furthermore, `Email` was unique. `Friends` were mapped as shown in Listing 7.4.

```
HasManyToMany(x => x.Friends).Table("Friends").ChildKeyColumn("Friend_id");
```

**Listing 7.4:** Friends mapping.

### OnlineStatusMapping

For `OnlineStatus` the `Id` property was mapped using the `Id` method as it made it an auto-incrementing uniqie identifier. `UserId` and `LastActivity` was mapped using the `Map` mathod. All mapped properties were not nullable.

## 7.2.3 Controllers

This section describes the *controllers* of the User component. As written in Section 6.3.2 ASP.NET MVC provides a simple way for checking authorization of a user. One adds the `Authorize` attribute to any action in which authorization is required.

### AuthenticationController

All the functionality for authentication was collected in a separate authentication *controller*.

When a user visited the login page, the `LogIn` action was called in the authentication *controller*. This action was overloaded, so that there was a version for when the page is requested via HTTP GET, and one for when it was requested with a HTTP POST. When `LogIn` was requested an empty login form was returned. When the user entered login data into the form and submits, the data was send as a HTTP POST to the login action. This action contained logic for validating the entered data. In the case that the data was valid, an *authentication cookie* was set and the user was redirected to the newsfeed page. Before the redirection, the password was hashed.

The `LogOut` action handled logging out of StudyLife. A link labeled "Log out" in the upper right corner directed to the `LogOut` action in the authentication *controller*. In this action the authentication cookie was deleted by using the ASP.NET `FormsAuthentication.SignOut()`, and the user was redirected to the login form.

### ProfileController

The `ProfileController` contained an `Index`, an `Edit` and a `Delete` action. The code for the user profile page was contained within the `Index` action of the profile *controller* and a corresponding *view*. The action used the `Authorize` attribute, as described above and shown in Listing 7.5, which ensured that it may only be accessed by logged in users.

If the `Index` action in Listing 7.5 was called with a user `Id`, it would return profile data to the *view* for the user with the given `Id`; see Figure 7.5. In case no user `Id` was supplied, it returned data for the currently logged in user.

```
[Authorize]
public ActionResult Index(int? id)
{
  if (id != null)
  {
    User user = UserHandler.getUser((int)id);
    if (user != null)
    {
      return View(user);
    }
  }
  else
  {
    return View(UserHandler.getCurrentUser());
  }
  return RedirectToRoute("NoAccess");
}
```

**Listing 7.5:** The user profile index action

The `Edit` action was overloaded. One method returned the model of the logged in user to the *view*. The other was called when the form in the *view* is posted. In this method, the altered data for the user was stored if the form validation passed.

The delete action called the delete method from the UserHandler method, and created a notification using the Notification component.

### UserWallController

The `UserWallController` returned a *view* that contained the Wall *partial view*. The user was either a specific user, fetched by means of `userHandler.getUser(id)`, or the current user. The returned *view* was used by the Wall component to display the user's wall.

## 7.2.4 UserHandler

The purpose of the `UserHandler` was to provide an interface for retrieving and saving data regarding the `User` *model*. All communication to the database was done through the Data Access Layer provided by another project-group. All the methods specified in Figure 7.4 were implemented.

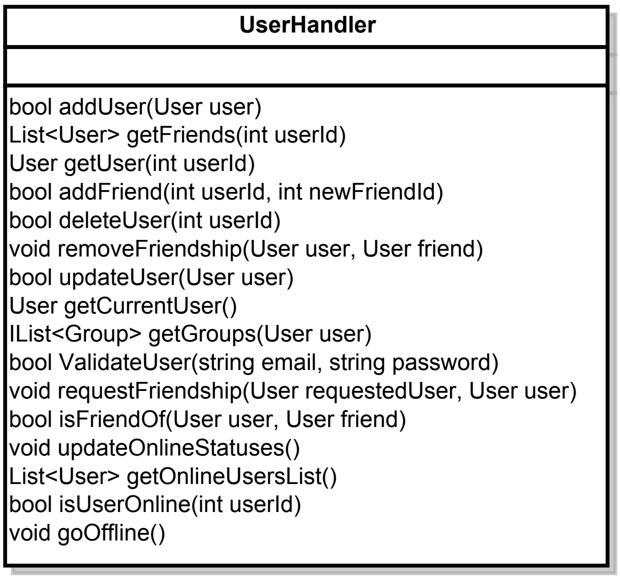

UserHandler
bool addUser(User user)
List<User> getFriends(int userId)
User getUser(int userId)
bool addFriend(int userId, int newFriendId)
bool deleteUser(int userId)
void removeFriendship(User user, User friend)
bool updateUser(User user)
User getCurrentUser()
IList<Group> getGroups(User user)
bool ValidateUser(string email, string password)
void requestFriendship(User requestedUser, User user)
bool isFriendOf(User user, User friend)
void updateOnlineStatuses()
List<User> getOnlineUsersList()
bool isUserOnline(int userId)
void goOffline()

**Figure 7.4:** `UserHandler` UML.

The following describes selected methods.

### getCurrentUser

`getCurrentUser` uses the following line of code, to retrieve the user `Id` from the authentication cookie:

```
int userId = int.Parse(HttpContext.Current.User.Identity.Name);
```

The User *model* was then retrieved using `getUser` described below.

### getUser

To get a specific user by means of a User `Id` or `Email`, the `getUser` method was used. As can be seen on Listing 7.6 the method is overloaded, and fetches either specifically by the `Id` or by matching the `Email` property with an email in the database.

```
public User getUser(int userId)
{
  return repository.FetchById<User>(userId);
}

public User getUser(string email)
{
  return repository.FetchByCriteria<User>(u => u.Email == email);
}
```

**Listing 7.6:** The getUser method.

### addUser

When a new user is registered, the newly created `User` *model* was stored in the database. The `addUser` method handled this functionality by checking if the user's email was already registered, to avoid duplicates, and saving the User *model* in the database. The method was responsible for notifying the `ComponentAnnouncer` about the user creation.

The `ComponentAnnouncer` was a class, delegated to one of the other groups in the multi-project, implementing the observer pattern, that notified other components about entity creation and deletion. The purpose of this was, for instance, to create a new Wall when a new user registers to StudyLife.

### updateOnlineStatus

The online status for the logged in user was updated whenever he made a request. This was done the with the `updateOnlineStatus` method, which was defined in the UserHandler. This method was called whenever the `Application_EndRequest` was invoked. The method updated the `LastActivity` property to the current time for the `OnlineStatus` *model*, belonging to the currently logged in user. In case the user did not already have an `OnlineStatus`, the method created one.

### ValidateUser

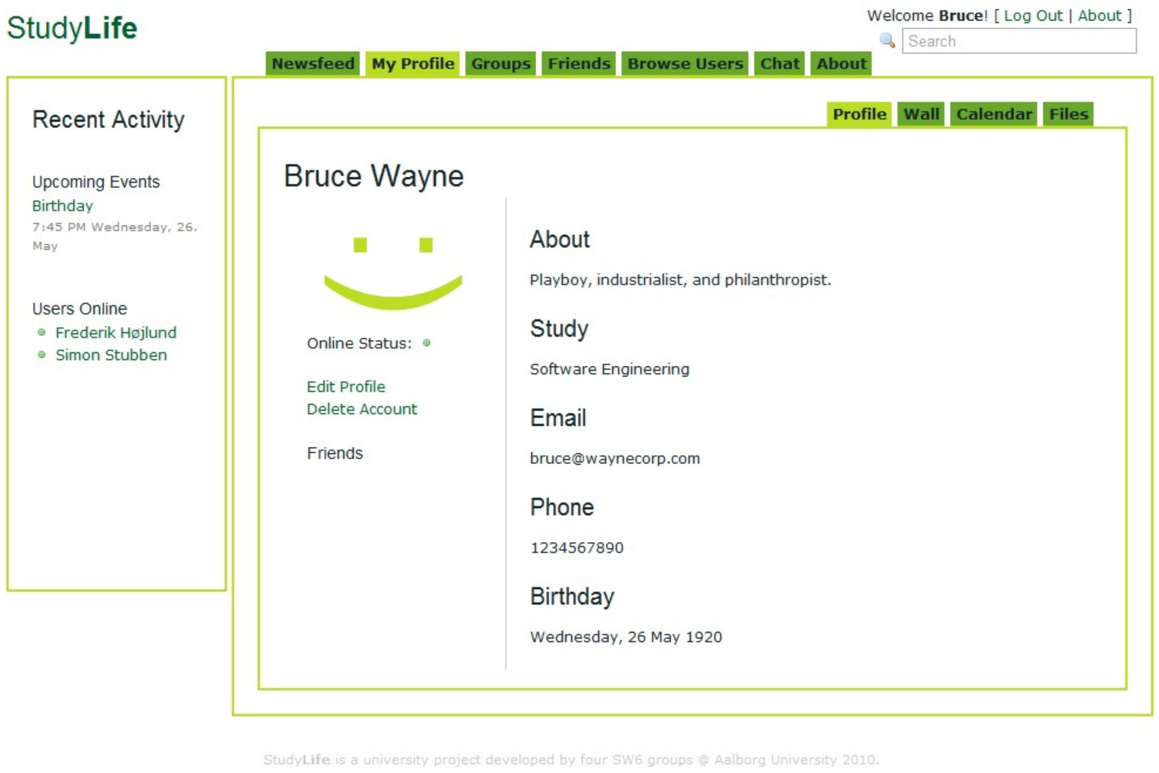
`ValidateUser` was used when a user logged into the system. The method checked if the email provided by the user at login was present in the database and if the password entered matched the fetched email's corresponding password. In case the email existed and password matched, the user was logged in and `ValidateUser` returns true.

## 7.2.5 Views

The following describes the how the `views` in the User component were implemented.

### View Profile

This *view* displays a users profile and as can be seen on Figure 7.5. It contains user information and a small menu with options regarding the user, for instance, edit profile, delete profile or request friendship if it is another user. Furthermore it contains links to the wall, calendar and file-folder components, shown as tabs.



Recent Activity

Upcoming Events
Birthday
7:45 PM Wednesday, 26. May

Users Online
- Frederik Højlund
- Simon Stubben

Bruce Wayne

About

Playboy, industrialist, and philanthropist.

Study

Software Engineering

Email

bruce@waynecorp.com

Phone

1234567890

Birthday

Wednesday, 26 May 1920

StudyLife is a university project developed by four SW6 groups @ Aalborg University 2010.

**Figure 7.5:** The user profile page.

### Limited Profile

This is a simplified version of the “View Profile” *view*, used to render user profiles in cases where the viewer is does not have a friend relationship to the user in question. The *view* displays a profile image, name and a “Request Friendship” link for the user in question.

### Edit Profile

Edit profile contains a form for altering previously entered user profile data. The entered data is validated by means of the attributes added to the `User` *model*, see Listing 7.3.

### LogIn

This *view* contains a form with a input field for email and password, a checkbox enabling “remember me” functionality, and a submit button.

If the login fails, by not passing the validation checks, an error message is diaplayed in red text.

### Register

This *view* contains a form with a input fields for; first name, last name, email password, confirmed password, study, birthday and phone number.

If the registration fails, by not passing the validation checks, an error message is displayed in red text.

To make it easier for the user to type in his birthday in the right format, a JQuery calendar pops up when the birthday field is clicked.

#### User Wall

This *view* contains a *partial view* provided by the project-group responsible for the Wall component. The *partial view* takes a user Id as parameter and displays that users' Wall.

#### OnlineStatusList

Online status for all users was represented in the sidebar, as can be seen in Figure 7.5. A partial *view* was responsible for providing the list of the currently online users for the sidebar. It represented the list of users provided by the method `getOnlineUsersList`, which returned all the the users with an `OnlineStatus` with a `LastActivity` no older than three minutes, except for the current user himself.

#### OnlineIndicator

Another *partial view* was responsible for showing online status on profile pages and user lists. This was the `OnlineIndicator`, which was provided with a user `id`, that would show a red point if that user was offline and a green point if the user was online. This can be seen in Figure 7.5. The `OnlineIndicator` used the `isUserOnline` method.

## 7.3 Friend Component

As written in the design for the Friend Component, Section 6.5, the functionality for handling friend relationships was contained within the Friend *controller* as well as a few *views.* The following describes how the implementation of the Friend component was carried out.

### 7.3.1 FriendController

When the user clicked the "Friends" tab in the menu, the `Index` action of the `FriendController` was invoked, which passed the friends of the currently logged in user to the "My Friends" *view.*

The user also had the option to click the "Browse Users" tab in the menu, which invoked the `All` action. This action passed a list containing all users, except for the logged in user himself.

The friends *controller* also contained actions for handling requests and removal of friend relationship.

Requesting a friend relationship was implemented by means of the Request component; if a user accepted a request send by another user, the two users would become friends. The relation was handled by the ORM, which made sure that both users had a friend relationship to each other. Figure 7.1 illustrates user A being friend with user B. As can be seen on the figure, user B has a relation to user A as well. This was done to avoid having a one-way friend relationship, i.e. user A is in a friend relationship with user B, but not vice versa, and to notify users properly by means of the Notification component.

Removal of a friendship relation was implemented by removing entries in Figure 7.1 for both users.

| **User_Id** | **Friend_Id** |
|---|---|
| 1 | 2 |
| 2 | 1 |

**Table 7.1:** Example of friend relationship

### 7.3.2 Handler

The Friend component did not have its own handler, however it used a series of methods in the `UserHandler`. The reason why these methods were not put in a seperate Friend *handler* was that the friend functionality was initially a part of the User component, and to avoid conflicts regarding dependencies of methods already implemented in the `UserHandler` to groups part of the multi-project. These methods were `getFriends`, `addFriends`, `removeFriendship`, `requestFriendship` and `isFriendsOf`.

The following section describe the `addFriend` method.

#### addFriend

The `addFriend` method took the users' `Ids` as parameter, retrieved the `User` *models* for those `Ids` and added the models on each other's `Friends` property. This was only done if they did not already appeared on the lists, as can be seen in Figure 7.7. The method returned false, if the friends relationship already exists, or if the two `User Ids` provided are the same.

```
public bool addFriend(int userId, int newFriendId)
{
  User user = this.getUser(userId);
  User friend = this.getUser(newFriendId);

  if (!user.Friends.Contains(friend) && !friend.Friends.Contains(user) && userId !=
      newFriendId)
  {
    user.Friends.Add(friend);
    repository.Save<User>(user);

    friend.Friends.Add(user);
    repository.Save<User>(friend);
    return true;
  }
  else
  {
    return false;
  }
}
```

**Listing 7.7:** The addFriend method.

### 7.3.3 Views

#### My Friends

The "My Friends" *view* presented the list of friends in a *partial view*, that was used in all cases where a list of users had to presented. This *partial view*, which is used in the screen shot in Figure 7.6, provided a consistent way of viewing users throughout the system. It provides the `Name` and `Description` property for each user, as well as options for requesting or breaking friendships, dependent on the current status of the relationship between the users.

#### All Users

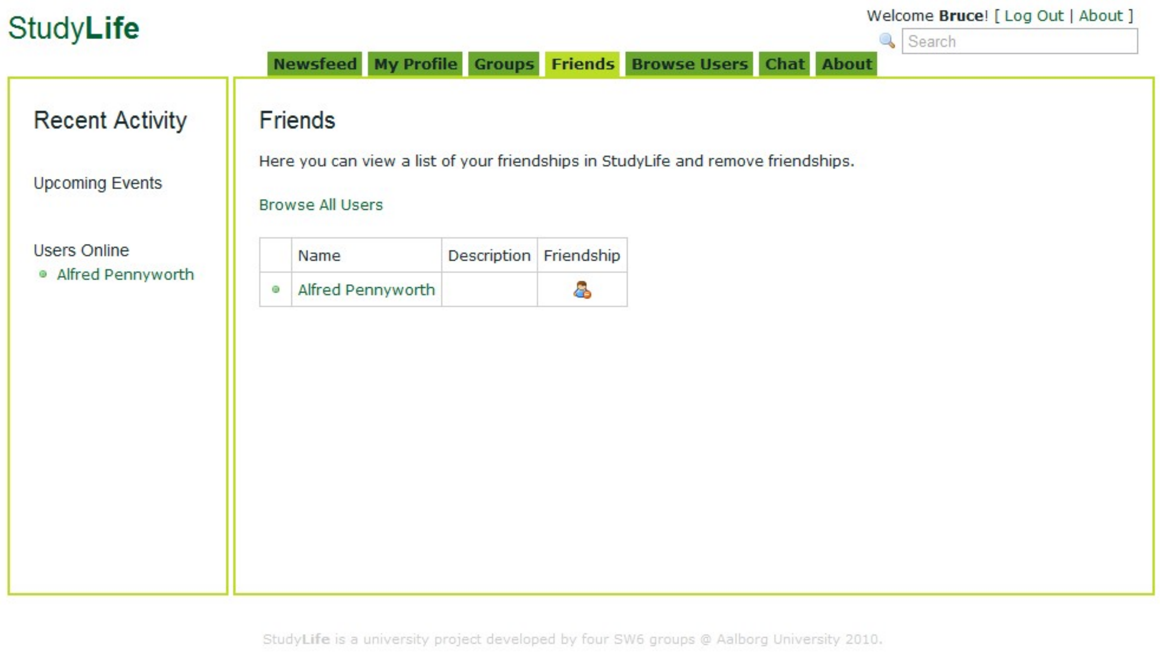
The "All Users" *view* was very similar to the "My Friends" *view*. It uses the same partial view to render all the users provided by the `All` action.

## 7.4 Group Component

This section covers the implementation of the Group component. As with the design of the Group component, all the seperate parts are covered.

### 7.4.1 Group Model

The Group *model* is a class that is used to represent groups, for instance study groups, in StudyLife.



Recent Activity

Upcoming Events

Users Online

● Alfred Pennyworth

Friends

Here you can view a list of your friendships in StudyLife and remove friendships.

Browse All Users

| | Name | Description | Friendship |
|---|---|---|---|
| ● | Alfred Pennyworth | | 👤 |

StudyLife is a university project developed by four SW6 groups @ Aalborg University 2010.

**Figure 7.6:** The My Friends page.

As the `User` model class, the properties of the `Group` class uses the `Required` attribute, that are used to ensure that fields are filled during form validation whenever a view is strongly typed with `Group` *model*. Aside from `Members` all the properties of `Group`, given in the design Section 6.6, have been implemented with the `Required` attribute.

### 7.4.2 Group Mapping

The mapping of the `Group` *model* to the database, was done by using the Fluent NHibernate `Map` method on `Name`, `Description` and `IsPrivate`. `IsPrivate` was additionally not allowed to be nullable. The `Contact` was handled by the `References` method, since it referenced a User. Finally, the `HasManyToMany` method maps User members of the group to the database.

### 7.4.3 GroupHandler

The `GroupHandler` provided an interface for fetching, saving, altering and deleting data regarding the Group *model*. The methods described in the design were all implemented and made use of the functionality provided by the `Data Access Layer`. Some of these methods are described briefly.

`getGroup` and `getMembers` returned a Group by providing a group `Id` and a group's members by providing a group, respectively. The `createGroup` and `Deletegroup` both made use of the `ComponentAnnouncer` to announce that a group was created or deleted in StudyLife. `removeMember` and `addMember` methods both made use of the Group property `Member`, which was a list of Users that added or removed from.

The follow the rest of the methods of the `GroupHandler`.

#### inviteMember & requestMembership

To invite a user or for a user to request a membership of a group, `inviteMember` and `requestMembership` are used. Both methods make use of the `RequestHandler` provided by the Request component. Listing

7.8 shows how a user request for a group is implemented.

```
public void requestMembership(Group group, User user)
{
  User receiver = group.Contact;
    RequestHandler.CreateRequest<GroupRequestHandler>(receiver,
  user.FirstName + " " + user.LastName + " requests to join " + group.Name,
  group.Id);
}
```

**Listing 7.8:** Method for requesting a group membership

The receiver of this request will always be the Group `contact`, and is notified with the string message, as can be seen in the listing.

The `inviteMember` is done very similar, except for having the Group `contact` as sender and invited user as receiver.

#### hasMember

The `hasMember` method checks if some user is member of a specific group. Since the `Member` property of Group is a list, the C# method `contains` is used to return true, if the user is member, and false in case he is not.

### 7.4.4 Views

#### Group Profile Page

The group profile page is made to be similar to the user profile page for consistency. The *view* presents all the data about a group, the properties, as well as all the options the user has regarding the specific group, provided by the `GroupHandler`. If the user is a member, he is presented with the options "Edit group", "Leave group", "Invite friends" and "Delete group".

#### My/All Groups pages

The "My Group" page is the initial page that is shown when a User presses the "Groups" link in the main menu. The page contains a list of all the groups that the current user is member of and a link that directs to the "All Groups" page. The list is created by means of the `userHandler.getGroups`, which fetches a user's groups, and is presented with a *partial view*, `GroupList`, which lists basic information about groups: `Name`, `Description` and `RequestGroup` or `LeaveGroup` functionality.

The "All Groups" view lists all the groups by fetching them from the database and presenting them with the `GroupList` partial view.

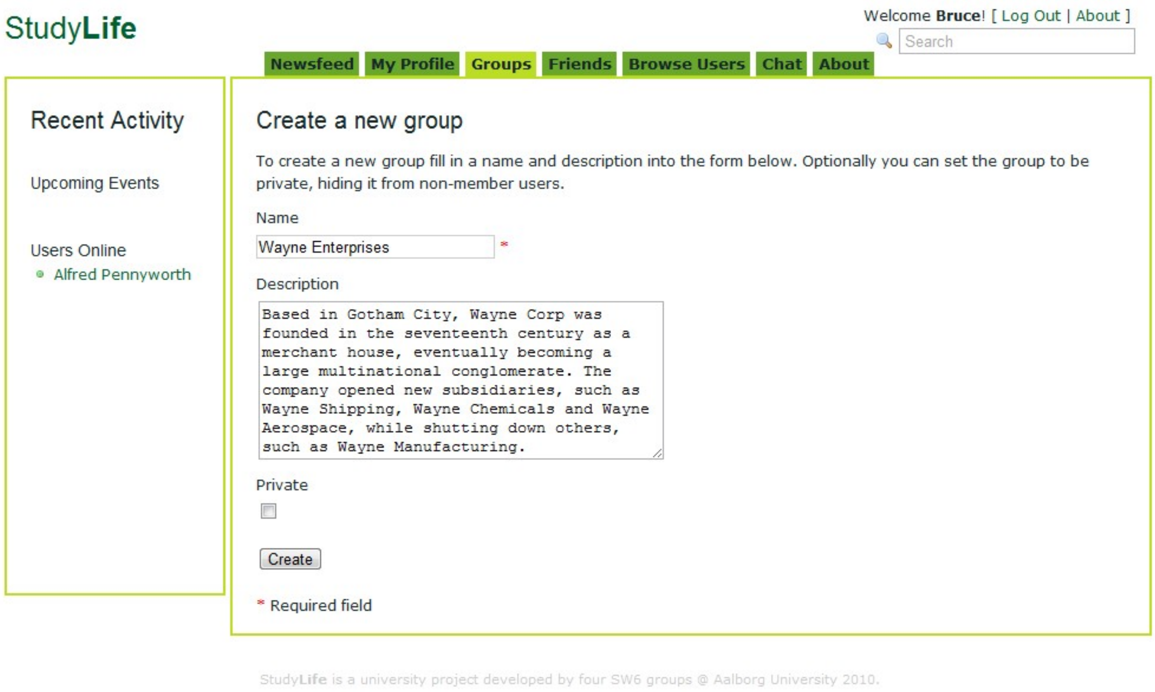
### 7.4.5 Create Group Page

The create group page contains a form that can be used to create a new group. The page can be seen in figure 7.7.

As can be seen on the figure, the properties `Name`, `Description` and `IsPrivate` are displayed and able to be filled, where `Name` is the only required property. When submitted, by pressing "Create" a new group is created.

#### Edit Group Profile Page

The "Edit group" page is another form page, very similar to the create group page in appearance, except for the contact field, which allow the user to choose a new contact from a drop-down list consisting of existing members. When submitted, the newly submitted data is updated for the given group.



Recent Activity

Upcoming Events

Users Online

- Alfred Pennyworth

Create a new group

To create a new group fill in a name and description into the form below. Optionally you can set the group to be private, hiding it from non-member users.

Name

Wayne Enterprises *

Description

Based in Gotham City, Wayne Corp was founded in the seventeenth century as a merchant house, eventually becoming a large multinational conglomerate. The company opened new subsidiaries, such as Wayne Shipping, Wayne Chemicals and Wayne Aerospace, while shutting down others, such as Wayne Manufacturing.

Private

□

* Required field

StudyLife is a university project developed by four SW6 groups @ Aalborg University 2010.

**Figure 7.7:** The create group page.

### Invite Friends

The "Invite friends" page contains a list of User friends that are not already members of the group. For each friend in the list, a button that sends a group invitation to the friend when pressed is visible.

## 7.4.6 Controllers

Three Group *controllers* were presented in the design. The following describes how each were implemented.

### Profile Controller & Group Wall Controller

The `ProfileController` only contains one action, namely `Index`. This action is very similar to the `Index` action of the user profile *controller* described in Section 7.2.3, but uses the `GroupHandler` and renders the Group profile *view* if the group is public, or if it is private and the current user is a member. In case the User is not a member of the private group a "You are not a member of the private group." message is shown. As with the User profile, the Group profile make use of the `Authorize` attribute.

The `GroupWallController` fetches the group provided by `Id` in the parameter, and returns a *view* consisting of the group, which the Wall component uses.

### Group Controller

As the design of Group describes, the `GroupController` contains all the major functionality of the Group component.

The `Index` action in the `GroupController` return the groups of the current user to the "My Groups" view. The `All` action returns all public groups to the "All Groups" view.

The create and edit *views* have corresponding `Create` and `Edit` actions, that take a group as parameter and saves it when submitted. The user that creates a group becomes the initial `Contact` for that group.

The ability to leave a group is taken care of by the `LeaveGroup` action, and in case the User is `Contact` the `LeaveGroupConfirm` action is used. This action deletes the group if the `Contact` confirms to leave the group.

The `DeleteGroup` and `RequestGroup` deletes a group and requests for a group membership, respectively, and the `InviteMember` is used by the "Invite Friends" view, for inviting new members to the group.

## 7.5 Search Component

In the design of Search component, Section 6.7, database search was chosen as the search method, and a solution consisting of a search result *view*, a search *controller* and *handler* was designed (see Figure 6.21). The following describes how this design was implemented.

### 7.5.1 Search Controller

As there is only one way to perform a search in StudyLife, the `SearchController` only contains one *action*. This method, named `All`, consists of two steps:

1. Check that a search query is provided and parse it for any illegal characters which are deemed not-searchable.

2. Call the search *handler* with the provided search query and pass the search results to the view.

Step 1 is done by matching the search query with a regular expression. Listing 7.9 shows the method used for validating the search query.

```
public bool isValid(string query)
{
  Match match = Regex.Match(query, @"^[a-zA-Z0-9\s\@\.]+$");
  return match.Success;
}
```

**Listing 7.9:** Method used to validate search queries

The `isValid` method uses the `Match` class to check if the given query satisfies the provided regular expression. The regular expression allows regular and capital letters, numbers, periods, @ and white-spaces. The periods and @ are allowed to enable searching for email addresses. If the query contains any other characters than those allowed, `isValid` returns `false` and the user is redirected to an error page.

The query is actually checked twice when sent to the server. ASP.NET contains a module (ASP.NET Request Validation) that validates all requests. In the configuration file (web.config) of StudyLife it is defined that all requests is to be checked for potentially dangerous values like HTML, JavaScript etc. If a request value is matched as potentially dangerous an exception is thrown and the request is discarded.

The reason for including the `isValid` method is that ASP.NET Request Validation allows some special characters which should not be searchable.

The query is then split at every whitespace into an array of separate strings. This is done to get less strict results for any given search. This can be seen in Figure 7.8, where "bruce wayne" is the search query. If the queries were not split and contained more than one word, they would not give any results. The array containing the query is passed as parameter to the search *handler*.

Step 2 of the `SearchController` is to pass the search result, returned from the `SearchHandler`, to the search result *view*. This is achieved by storing the returned `List` objects in the `SearchController`'s `ViewData`. This makes the search results, e.g. lists of objects matching the search query, available in the *view* such that it can render the results.

### 7.5.2 Search Handler

The `SearchHandler` is where the actual search is performed. It contains a method for each type of content that is to be searched in. The methods are very similar, but each returns a list of objects of the type which is searched. Each method takes an array of strings, the search query, as parameter. Listing 7.10 shows the `SearchGroups` method.

```
public List<Group> SearchGroups(string[] query)
{
  List<Group> resultList = new List<Group>();
  foreach (String q in query)
  {
    List<Group> groupNameResult = repository.FetchListByCriteria<Group>(x => x.Name.
        Contains(q) && x.IsPrivate == false).ToList();

    foreach (Group g in groupNameResult)
    {
      if (!resultList.Contains(g)){
        resultList.Add(g);
      }
    }

    List<Group> groupDescriptionResult = repository.FetchListByCriteria<Group>(x => x.
        Description.Contains(q) && x.IsPrivate == false).ToList();

    foreach (Group g in groupDescriptionResult)
    {
      if (!resultList.Contains(g)){
        resultList.Add(g);
      }
    }
  }

  return resultList;
}
```

**Listing 7.10:** The `SearchGroups` method

For each string in the query array the `Name` and `Description` properties of the public groups are searched. This is done using the method described in Section 6.7.2. Private groups are not searchable, to keep data visible only to members. The groups containing the search query are added to the `resultList` if they are not already added. After each string in the query array has been searched for, the `resultList`, containing the groups that matched the query, is returned.

As the system becomes larger, searches will take longer, return more results and increase the load on the web-server and database. This problem could be solved by using either Lucene or SQL Server Full Text Search, as these solutions creates indices that are faster to search because they contain less data compared to this Search solution.

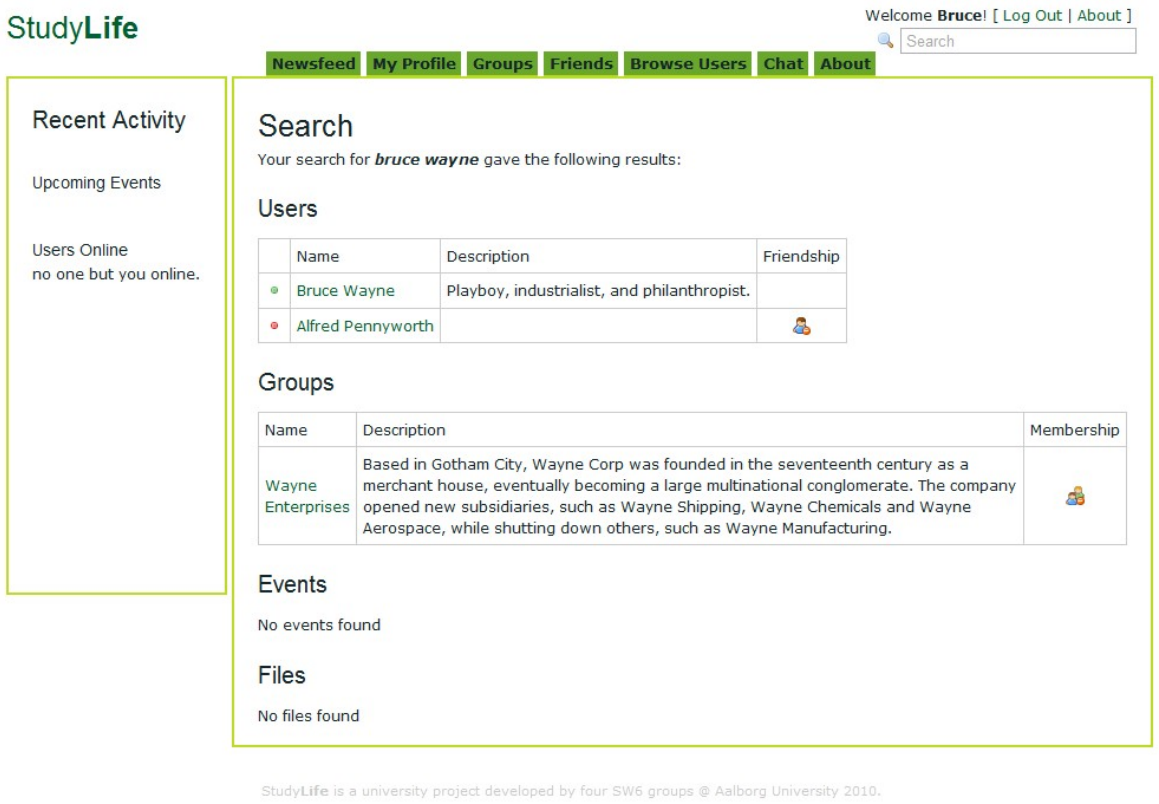
Another solution could be to create separate searches for searching users, groups, events and files. The user could then choose what he wishes to search in. Such functionality could easily be implemented as the `SearchHandler` already contains a separate methods for each type of searchable content.

### 7.5.3 Search Result View

The search result *view* displays the results returned by the `SearchHandler`. The search results are categorized into separate tables each with a separate title. I.e. there is a separate table for found users, groups, events and files. Figure 7.8 shows a screenshot of the search results *view*.

The search results are rendered by means of the `UserList` and `GroupList` *partial views*, and links to the Events found. Files are not linked to as the project-group responsible for this component have not provided functionality for linking directly to a file.

The implementation of search was done according to the design chosen in Section 6.7.3. It works as planned and returns results that makes sense for the given search query.



Recent Activity

Upcoming Events

Users Online
no one but you online.

Search

Your search for ***bruce wayne*** gave the following results:

Users

| | Name | Description | Friendship |
|---|---|---|---|
| ● | Bruce Wayne | Playboy, industrialist, and philanthropist. | |
| ● | Alfred Pennyworth | | |

Groups

| Name | Description | Membership |
|---|---|---|
| Wayne Enterprises | Based in Gotham City, Wayne Corp was founded in the seventeenth century as a merchant house, eventually becoming a large multinational conglomerate. The company opened new subsidiaries, such as Wayne Shipping, Wayne Chemicals and Wayne Aerospace, while shutting down others, such as Wayne Manufacturing. | |

Events

No events found

Files

No files found

StudyLife is a university project developed by four SW6 groups @ Aalborg University 2010.

**Figure 7.8:** Search results page.

As the system becomes larger, the search result page will contain more and more search results and take longer to load and become too large for the user to quickly find what he is looking for. A solution for this could be to split the search results into separate pages if the search results pass a defined amount. Another solution could be to paginate the search results and only show up to 10 rows in each table at a time.

## 7.6 Summary

In this chapter the implementation of the design was covered. The chapter covered implementation of the master layout, the users component, the friends component, the groups component and the search component.

The next chapter covers how the system was tested.

# Chapter 8
# Testing

Testing software is important for ensuring that a software application works as intended. Since StudyLife prioritized correctness and reliability high and medium respectively (cf. the quality goals in Section 2.4), its functionality needed to be reliable and correct when given different kinds of input. Hence, testing was carried out to ensure the components of this project were correct and successfully executed on any given input.

As written in Section 4.5 testing can be done in various ways, but since the project was under a deadline, limited unit, integration and system testing were performed. In addition to these methods, manual testing was done continuously. Manual testing is the method of manually testing the software for faults. Thus, a tester is required to perform actions, as an end-user, on the system. As this was done continuously, each developer played the role as tester after he had written some code. By doing manual testing it is ensured that the features of the software application works as intended.

Nevertheless, as the manual testing did not follow a test plan, all partitions of input (valid and invalid) might not have been tested. For this reason automated unit tests and system testing were performed. Due to the project being a multi-project with several components to be integrated, integration testing was carried out as well.

The following sections includes descriptions of how the tests were performed and what was achieved by performing them.

## 8.1 Unit Testing

The unit testing of the components of this project did not end up as expected, since the group initially had decided to do test-driven development (TDD) and thereby have tests for each method before the code and functionality was written.

Due to lack of testing experience among group members, lack of time and a focus on working code, TDD was dropped and unit testing never had the role it was expected to have. However, unit testing was done on all the components of this project, but only on the *controllers*. The *controllers* were chosen rather than the handlers, since the specific actions made use of almost every method in the specific handlers. Furthermore, by testing the *controllers* you are also able to use the *views*, so these are tested simultaneously.

The unit tests were implemented similar to the example given in Section 4.5.1 and performed by looking at expected and actual data that was compared. In case the expected and actual were not equal the test would fail. The test data were set up using mocks that represented the database and inserted random users that belonged to a random group for each test. This was done to guarantee the expected outcome of the method to be tested. The session was also mocked, in order to test logging in and out.

The following sections describe the test methods for each of the components, namely User, Friend, Group and Search.

### 8.1.1 User Component Unit Tests

Table 8.1 contains tests for a user to login, register, view his profile, edit his profile and delete his user account. The table contains a result column as well, which indicates a passed (✓) or failed (✗) test. As can be seen on Table 8.1 all tests pass and this confirms that the *controller* was implemented correctly.

| Test description | Result |
|---|---|
| Is "Login" *view* displayed. | ✓ |
| Does "Login" *view* show error message on invalid input. | ✓ |
| Is "Register" *view* displayed. | ✓ |
| Does "Register" *view* show error message on invalid input. | ✓ |
| Does `Register` *action* redirect to correct. *view* on success | ✓ |
| Is the "Profile" shown if no `User` id is supplied. | ✓ |
| Is the "Profile" view shown if an user id is supplied. | ✓ |
| Is "Edit profile" *view* displayed. | ✓ |
| Is "Edit profile" data saved. | ✓ |
| Is a `User` deleted. | ✓ |

**Table 8.1:** Results of the User component unit tests

### 8.1.2 Friend Component Unit Tests

The Friend component contains functionality for making friend relationship between users. The tests performed checked for a *view* listing all users, a *view* listing all friends, functionality for removing and requesting relationship to another user. Table 8.2 shows how all the tests passes and therefore confirms the `FriendController` to be correctly implemented.

| Test description | Result |
|---|---|
| Is the "all users" *view* displayed. | ✓ |
| Is the "My Friends" *view* displayed. | ✓ |
| Is a friend relationship created. | ✓ |
| Can a `User` request a friend relationship. | ✓ |

**Table 8.2:** Results of the Friend component unit tests

### 8.1.3 Group Component Unit Tests

The Group component contains functionality for users to create, view and modify groups. Table 8.3 contains all the tests performed on the group *controller*, which includes creation, editing, leaving, inviting, requesting and deleting a group. Table 8.3 shows that all tests pass, hence the group *controller* was correctly implemented.

| Test description | Result |
|---|---|
| GroupIndexTest | ✓ |
| Is the "Create group" *view* displayed. | ✓ |
| Is a `Group` created. | ✓ |
| Is the "edit group" *view* displayed. | ✓ |

| | |
|---|---|
| Is an edited `Group` saved. | ✓ |
| Can a member leave a `Group`. | ✓ |
| Is a confirm *view* displayed if last member leaves a `Group`. | ✓ |
| Can a `User` request membership to a public `Group`. | ✓ |
| Is a `Group` deleted. | ✓ |
| Can a `User` be invited to join a `Group`. | ✓ |
| Is "Invite Members" *view* displayed. | ✓ |

**Table 8.3:** Results of the Group component unit tests

### 8.1.4 Search Component Unit Tests

The Search component was tested by giving a random string query and check for the *view* not returning `null` and check for returning the correct *view* result. As can be seen on Table 8.4 this was done with and without a query. Both tests passed, meaning that the search *controller* was implemented correctly.

| Test description | Result |
|---|---|
| Is the search results *view* displayed when an legal query is supplied. | ✓ |
| Is the error *view* displayed when an illegal query is supplied. | ✓ |

**Table 8.4:** Results of the Search component unit tests

## 8.2 Integration Testing

When a software application consists of multiple components, it is necessary to see if these work as intended when put together. When components are integrated, data might be lost due to bad interfacing or functionality might not produce the desired output. Thus, integration testing was performed. Some of the different methods of integration testing are described in Section 4.5.2.

The multi-group in the project decided that smoke testing should be performed, such that the system was integrated incrementally and every group made sure that their own components were able to build and perform as intended. A common code repository and automated builds further allowed for this kind of integration.

It was also planned to do integrations every second week, at the multi-group meetings, but this was only carried out at the last meeting due to dependencies and lack of time.

At the end of the project, three weeks before deadline, the system was integrated and a test specification for the system testing was agreed on. As the system was integrated incrementally in the process, only minor integration flaws were found. The following section describes the system testing performed.

## 8.3 System Testing

After the integration test was carried out a system test was done. As written in 4.5.3 there exists a series of different tests to fully exercise and test an application. Manual scripted testing and ad hoc testing were chosen by the multi-group as the system test methods for StudyLife.

The next sections describe how the system testing methods were performed.

### 8.3.1 Manual Scripted Testing

The multi-group of the project agreed that each group had to create test specifications for each of their components, which were made on the basis of the user stories. These test specifications had to contain some specific steps, and were after creation handed out to another group that was part of the multi-project. The reason for having a test specification, was to formalize the test method and have somewhat similar tests done to all components.

The test specification made for the components of this project resulted in four test cases. The test result of the test specification can be seen in Appendix B. Each step has been checked with an “X”, meaning the step has been accomplished. The tests resulted in catching a number of errors:

- Wrong notifications when a user joined a group
- Edit group was not provided with a `Contact` user due to validation
- No link to users on notifications

These mistakes were debugged and corrected.

### 8.3.2 Ad hoc Testing

The last test method to be performed was ad hoc testing. This was done informally by letting all the developers run StudyLife simultaneously on the server. Then any actions they could think of was performed and all errors caught was gathered. It was then up to the separate groups to correct the errors in their components.

Many small bugs and “nice-to-have” features were found and corrected or implemented.

## 8.4 Summary

This chapter has gone through the different test methods used to test. Each section contained descriptions of how, unit testing, smoke testing, manual scripted testing and ad hoc testing, were performed and what was achieved by using them. Errors caught during testing, were corrected.

After considering the software, the next chapter takes a look at the project process.

# Chapter 9
# Process Reflection

The subject of this project has focused on application development and the development process.

This chapter includes a reflection of this project's process. First, a reflection of the multi-project is carried out. Then the development method of this project is reflected upon, followed by an in-depth Strengths, Weaknesses, Opportunities, and Threats (SWOT) analysis of the software engineering tools and practices used. Finally, an evaluation of the SWOT analysis and a summary of the chapter is done in the summary section.

## 9.1 Multi-project Reflection

Working together in a large group, consisting of four smaller groups, proved to be quite a different experience than projects on earlier semesters. For example by changing the contract took up more time than it would if it was just a small group. Therefore, many decisions that might have been trivial in a single-group project, took longer to decide due to the many stakeholders of the project. Hence, the multi-group decided that a single person from each group met once a week to discuss potential risks, changes and status. Doing this gave each of the multi-project groups insight in what was going on in the other groups and if development went as planned.

Tools were used to ease and increase communication. A mail list made available through Google groups, which every developer in the multi-group was part of, was used to send mails that regarded major changes, critical bugs or important messages. For example, each weekly meeting had an agenda that was placed in the Google group. In case a group updated the agenda by adding a new topic, it was possible to send an email to every developer in the multi-group and have them discuss the topic before attending the weekly meeting.

Other tools used were the version control system Git and the project management web application Redmine. Using Git as a version control system made it easy for the groups to share code in between and collect it a central place. The branching in Git made it possible for the different groups to have their own branches. This made it possible to smoke test changes and dependencies in their own branch before pushing to the multi-group branch. Redmine was used as a wiki and Git browser, where the wiki included documentation for each group's specific code, e.g how to use a specific component's interface, and the Git browser gave an overview of who made commits to the server and the different branches. The wiki of Redmine was frequently used, since it contained all the groups' specific component documentation, and was highly beneficial.

It was a very valuable experience to work on such a relatively large project, and it will be a great asset in the future. During the development, problems and situations emerged which would have been hard to prevent. Besides, using an agile development method allowed for adding tasks that were discovered along the way to running and future Sprints.

## 9.2 Project Reflection

During the course of the project, Scrum turned out to be a useful asset. It was easy to implement and use because of its simplicity, and made planning of tasks and delegation of these almost seamless, compared to other methods used by the developers in earlier projects. This made it possible spend more time on actual development instead of the management of the project.

Another reason Scrum was easy to use and implement was because of its adaptability. Only the practices that made sense in the context of the project were chosen.

The following section does an in-depth analysis of the tools and practices used in this project.

### 9.2.1 Software Engineering Tools and Techniques

The purpose of this section is to find good and bad Software Engineering tools and techniques in correlation with this project. To do this, it was chosen to perform a SWOT analysis. The goal of the SWOT analysis is to find strengths, weaknesses, opportunities and threats in the development process. The SWOT analysis for this project is elaborated in the subsequent sections.

#### Strengths

The first six sections described below are all related to the Scrum development method. The last two are general agile statements.

**Blackboard & Burndown chart**

The black board consisted of the Sprint backlog and Burndown chart. It proved to be beneficial to the development process, since everyone in the group were able to follow the progress of each Sprint. Furthermore, when a group member finished a task he could physically move the task on the blackboard to the “Done” row, and then pick a new one to work on.

The Burndown chart was used to track the sprint progress and to make sure that the desired velocity was maintained to meet the deadline. As can be seen on Figure 9.1 is a picture of the blackboard taken in the end of a Sprint.

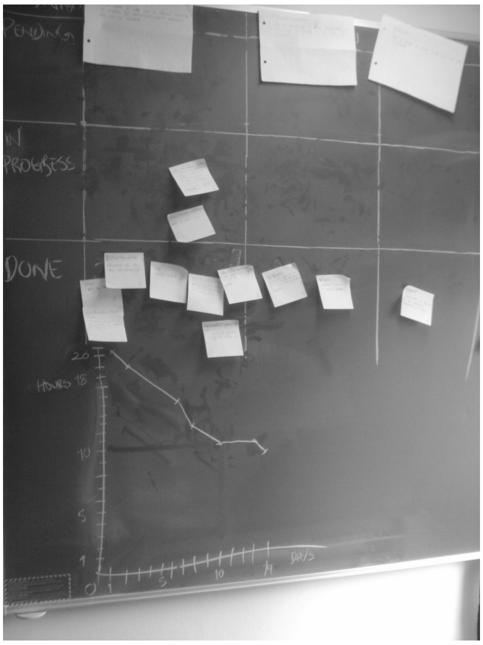


**Figure 9.1:** Picture of the blackboard, with Burndown chart in the bottom and sprint backlog above

Having a Burndown chart also affected short-term planning. For instance, if the Burndown chart was less than halfway down in tasks halfway through the Sprint in time, it meant that all tasks might not be completed before the Sprint ended.

#### Standing meetings

Each morning short standing meetings were held where project status was discussed. These meetings proved to be an efficient way to share information, status and delegate new tasks to the group.

#### Sprints & Product Backlog

The product backlog, which was created in the start of the project period, was advantageous as it split the project into several Sprints each containing a set of tasks. This made the process more manageable as the implementation was done in separated sprints, each with its own deadline. Moreover, due to many of the other components depending on the User and Group component, releasing these early in the process was well received.

#### Planning game

The planning stage done in the beginning of each sprint gave a good idea and estimation of what needed to be done in the next two weeks. Having this plan up on the blackboard in the form of post-it notes also gave a good indication of the progress of the current sprint.

#### Decision in one hour

Making decisions in one hour enabled to quickly continue the development and not waste too much time discussing lesser topics. If a decision was hard to agree on a group member was assigned to check for solutions and guidance on the problem and afterwards discuss the subject again.

#### Common room

A common agreement from the beginning was that code would only be written in the group room. This practice greatly increased the knowledge sharing amongst the group members. It also saved time during development, since topics could be talked about immediately and face-to-face, instead of having to write it down for discussion later.

#### Working software over comprehensive documentation.

Very little documentation was needed in the project group, since the option of talking to the other group members always existed while coding. In the multi-project group, however, communication was more limited. For this reason documentation was needed. This was handled by having the individual groups writing how-to's regarding only the functionality that was shared between the groups using Redmine.

#### Responding to change over following a plan.

By adhering to this statement, a more agile development process was achieved instead of blindly following a plan. In the spirit of the statement new tasks were allowed to be added to a running sprint, and thereby immediately have the ability to start working on critical tasks.

### Weaknesses

The following describes weaknesses of using Git as VCS.

#### Configuration management

The decision of using Git was an important decision, as there were 13 persons developing a single system. However, Git had a steep learning curve as it was a new tool for the all the developers.

Furthermore, it required a new rather different approach to version control than what the developers were used to. Using Git also had some annoyances as the Visual Studio project specific files were hard to manage when branching and merging.

### Opportunities

These are some opportunities that may have helped during the proces of the project.

#### Refactoring

No serious refactoring was done during the development process, but it could potentially prove to be useful. Refactoring could potentially have increased readability, simplicity and maintainability.

#### Test first

Using test first was definitely an opportunity as it may have caught several errors. This was not fulfilled because the multi-project groups depended on the User component. This meant tests were initially sacrificed for finishing the User functionality as early as possible. Writing the tests first may be more beneficial than writing tests after writing the code, as they provide valuable feedback during the development process rather than after. The project group decided to write unit test later in the process instead.

### Threats

This section contains elements that may have threatened the project process.

#### Wrong estimation

The threat of poor estimation was possible, since the group only had experience in estimating tasks from former university projects. If tasks are, for instance, underestimated, the deadline might be exceeded.

#### Waiting on dependencies

Dependencies of code which had been delegated to other groups would cause a block in the project process, if it was not completed by the time that it was needed.

## 9.3 Summary

In general Scrum made planning and delegation of tasks almost seamless compared to earlier projects. Of all the practices, the project group agreed the following Scrum practices to be the most useful:

- The black board, consisting of the Sprint backlog and Burndown chart.
- The product backlog.
- Scrums

The weaknesses of the project process were the version control system Git, which the developers did not know very well.

In future projects, the project group agreed that Scrum was working very well, as it is adabtable and allowed for using various sorts of practices.

In case the “Test first” practice is to be used in a future project, experience in testing and discipline would be useful.

# Chapter 10

# Conclusion

This project started with a proposal for creating a social web application similar to Facebook. During the first month this proposal was made more concrete by creating a system definition, specifying product functions and deciding an architecture. The defined system was then split into components and divided amongst the groups. This report has documented the process of analysing, designing, implementing and testing the master layout and User, Group, Friend and Search components. The following briefly outlines the topics covered in this report.

Chapter 2 described the joint analysis and design made in the first month by the four groups. Here a general definition of the system, its target users and overall functionality was introduced. In addition, the quality goals of the application, the chosen architecture, being a mix of several, and that ASP.NET MVC and Fluent NHibernate was used as technical platform, was presented.

In Chapter 3 Scrum was introduced and an explanation of how it was to be implemented in the project was performed. Furthermore, a series of practices suitable for the project was chosen and a description of the software engineering tool Git was done.

Chapter 4 went through relevant theory for developing the web application. First, MVC and ORM was explained. Then various web security vulnerabilities and solutions to these were presented, and since AJAX was used in the implementation of specific elements, a section covered how AJAX works. Moreover, the chapter included explanations of how software testing can be done.

The analysis of the delegated components was carried out in Chapter 5, where the master layout, web security measures and each component - User, Friend, Group and Search - was gone through. Based on the analysis, specific requirements were found and presented in the requirements specification.

In Chapter 6 the designs of the specific components were documented on the basis of the requirements specification. As in the analysis, the different parts was gone through separately. For the master layout use of master pages and a structure was chosen. The User, Friend and Group functionality were specified, and the Search method was decided to be done by means of non-indexed searching.

In Chapter 7 the implementation of the components delegated to the project-group was covered. For each component, the implementation of model, mappings, controllers, handlers and views were covered.

In Chapter 8 the different test methods, used to test the system, were described. This included unit testing, smoke testing and the specific system test methods manual scripted testing and ad hoc testing.

Finally, Chapter 9 presented a reflection of the whole project process. That is, how the multi-project and this project was carried out. Moreover, a SWOT analysis of the software engineering tools and practices was documented.

The requirement specification was satisfied, as all functionality regarding this project had a working implementation by the time of the system integration. As all the individual components was integrated the final product emerged. The experience of browsing through the finished system was rewarding as opposed to the system only containing the components of this project.

Throughout the project a lot of valuable experience has been gathered in terms of developing a larger application in cooperation with multiple groups. The group has learned that using an agile development method is beneficial when developing such an application. It was especially found that communication is a very important factor when several groups develop different components for a single system.

## Further Development

This section shortly summarizes some functionality that could be improved or added if more time was available for development.

### Advanced search

To acommodate a larger system with much more searchable data, the posibility of performing advanced searches could help users find what they are looking for. A special syntax could be implemented where characters could help make the search query more precise. Such characters could be "-" to remove results containing a given word. An example of a query using this syntax: "studygroup -test", would find results containing the word "studygroup" but without those including the word "test".

### Search Comments, Walls and Chats

Being able to search comments, walls and chats would make StudyLife fully searchable. As all communication in StudyLife is done through these components, users could quickly find relevant discussions and posts.

### OpenID Integration

Integrating OpenID authentication could remove the need for registering users and store login information in the database. The advantages of adding OpenID authentication was covered in Section 6.3.1.

### Profile Images

Functionality to link images from the File component to user and group profiles would make the profile pages more informative. This could make user profiles more personal and group profiles more "catchy".

### Retrieval of Lost Passwords

As of now there is no functionality for retrieving lost user passwords. Hence, a lost password will render an account useless. Functionality enabling users to reset their password would solve this problem, but also create new security issues that needed to be handled. A solution could be for users to have a personalised question they need to answer correctly before their password can be reset and a new one is emailed to them.

# Bibliography

[Apa10] Apache. Apache lucene.
http://lucene.apache.org, 2010.

[Arl06] Sagar G. Arlekar. The role of ajax in enhancing the user experience on the web.
http://www.roseindia.net/ajax/ajax-user-interface.shtml, 2006.

[COL10] COLOURlovers. Color trends + palettes :: Colourlovers.
http://www.colourlovers.com/, 2010.

[Eer06] Matthew Eernisse. *Build Your Own AJAX Web Applications.* Sitepoint, 2006.

[Esp09] Dino Esposito. Comparing web forms and asp.net mvc.
http://msdn.microsoft.com/en-us/magazine/dd942833.aspx, 2009.

[fac10] facebakers.com. Facebook statistics denmark.
http://www.facebakers.com/countries-with-facebook/DK/, 2010.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Git10] Git. Git scm wiki.
https://git.wiki.kernel.org, 2010.

[Gre09] James Gregory. Fluent nhibernate: Auto mapping introduction.
http://jagregory.com/writings/fluent-nhibernate-auto-mapping-introduction/, 2009.

[HGM09] Erik Hatcher, Otis Gospodnetic, and Michael McCandless. *Lucene in action.* Manning, 2009.

[Lar08] Craig Larman. *Agile & Iterative Development.* Addison-Wesley, 2008.

[Mat09] Aditya P. Mathur. *Foundations of Software Testing.* Pearson Education, 2009.

[Mic05] Microsoft. Guidelines for smoke testing.
http://msdn.microsoft.com/en-us/library/ms182613(VS.80).aspx, 2005.

[Mic09] Microsoft. Asp.net mvc overview.
http://www.asp.net/learn/mvc/tutorial-01-cs.aspx, 2009.

[Mic10a] Microsoft. Asp.net authentication.
http://msdn.microsoft.com/en-us/library/eeyk640h.aspx, 2010.

[Mic10b] Microsoft. Lambda expressions (c# programming guide).
http://msdn.microsoft.com/en-us/library/bb397687.aspx, 2010.

[Mic10c] Microsoft. Understanding html helpers (asp.net mvc framework unleashed).
http://www.asp.net/learn/mvc/tutorial-06-vb.aspx, 2010.

[Mic10d] Microsoft. Unit test framework.
http://msdn.microsoft.com/en-us/library/aa874515.aspx, 2010.

[MMMNS01] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objekt Orienteret Analyse and Design.* Marko, 2001.

[MSK09] Stuart McClure, Joel Scambray, and George Kurtz. *Hacking Exposed: Network Security Secrets and Solutions.* McGraw-Hill, 2009.

[Nie97] Jakob Nielsen. Search and you may find.
`http://www.useit.com/alertbox/9707b.html`, 1997.

[Ope10] OpenID. Openid foundation website.
`http://openid.net/`, 2010.

[osc09] open-source community. Fluent nhibernate wiki.
`http://wiki.fluentnhibernate.org/`, 2009.

[OWA10] OWASP. The open web application security project (owasp).
`http://www.owasp.org/`, 2010.

[Plc07] Ovum Plc. The need for software testing.
`http://www.marketresearch.com/product/display.asp?productid=1604029`, 2007.

[Pre05] Roger S. Pressman. *Software Engineering.* McGraw-Hill, 2005.

[RPS06] Y. Rogers, J. Preece, and H. Sharp. *Interaction Design, beyond human-computer interaction.* Wiley, 2006.

[SB02] K. Schwaber and M. Beedle. *Agile Software Development with Scrum.* Prentice-Hall, 2002.

[Tea10] ORMBattle.NET Team. Orm comparison and benchmarks on ormbattle.net.
`http://ormbattle.net/`, 2010.

[Tes] TestingGeek. Manual scripted testing.
`http://www.testinggeek.com/index.php/testing-types/execution-method/111-manual-scripted-testing`.

[wA09] Scott w. Ambler. Mapping objects to relational databases: O/r mapping in detail.
`http://www.agiledata.org/essays/mappingObjects.html`, 2009.

[Wal09] Stephen Walter. Understanding html helpers (asp.net mvc framework unleashed).
`http://stephenwalther.com/blog/archive/2009/03/03/chapter-6-understanding-html-helpers.aspx`, 2009.

[Wil04] Christoph Wille. Storing passwords - done right!
`http://www.aspheute.com/english/20040105.asp`, 2004.

# Appendix A
# StudyLife Development Contract

This contract is produced by a collaboration group of four 6th semester software engineering project groups. Its purpose is to cover system definition, analysis, requirements and choices for tools and technologies.

## A.1 Overall Analysis

The following analysis is based on the work done in cooperation between the groups in the first month of the project period. Before the analysis started, the groups chose to develop a web-based social software application, similar to the well-known social sites MySpace and Facebook.

### A.1.1 Overview of a similar system (role model: Facebook)

It is a good idea to look at existing systems that have similar functionality to what you are developing. We have chosen to look at a system called Facebook. Based on its high popularity it is safe to assume that the system is useful.

The requirements we have for our system are that it should be able to help people at a university work together, but also help people socialize with each other. Looking at Facebook, which is quite an elaborate system, there are a few core features that catch your eye. Facebook has the ability to establish friendships, meaning that a user on the system can request another user as a friend, thereby creating a link between the two users. They then have easy access to each other over the website. There are a variety of ways for users to interact with each other: - Walls - Chats - Groups - Messages.

A great feature of Facebook is the ability to use a walls or groups to communicate with other users. Obvious users to communicate with are ones that are listed as friends. Once two users are friends they are also able to use chats or messages to interact with each other. Some of these features are excellent for our social/educational networking site. We want to have friends and groups so users can communicate with the relevant crowd of people; either friends or group members (people that may attend the same courses).

### A.1.2 FACTOR

The FACTOR can be used to support system definition development.

- Functionality: A system that allows the users to collaborate, communicate and network.
- Application domain: Connect users, communication between users, organizing, and collaboration between users.
- Conditions: Development is split up in groups, and each group takes care of one or more subsystems.

| Importance | Low | Medium | High |
|---|---|---|---|
| Reliabillity | | x | |
| Security | | x | |
| Maintainability | x | | |
| Performance | x | | |
| Usabillity | | x | |
| Availabillity | | | x |
| Correctness | | | x |
| Testability | | | x |
| Modifiability | x | | |
| Reuseability | | x | |
| Portability | x | | |

**Table A.1:** Quality goals

- Technology: The system will run on a server; client access happens through a browser. The system will be developed to fit a specific architecture.

- Objects: Users, groups, calendar, calendar events, walls, wall posts, comments, files, requests and notifications.

- Responsibility: Social networking, collaboration and scheduling in relation to study life.

### A.1.3 System definition

StudyLife is a social web community for students and supervisors, inspired by social networks like Facebook. The purpose for developing StudyLife is to create a social web community that improves the collaboration of users in between.

Access to the system should be available globally through a web browser, i.e. registered users must be able to login to their accounts by directly accessing the website and then signing in with their user name and password anytime and anywhere.

The system should ease communication and collaboration through information-sharing between users by having different ways of interaction, and give opportunity for relations between users which reflect real-life relations. Furthermore, the system should aid in organization and coordination by offering users a calendar with the possibility to create and share events.

### A.1.4 Quality Goals

**Reliability**
Is somewhat important allowing minor for errors to occur.

**Security**
This is of medium importance because the system deals with personal data.

**Maintainability**
This is not important because this project focuses on the development phase, not continuous maintenance of the product.

**Performance**
The system is not expected to have a high user load, but should generally be responsive to user inputs.

**Usability**
This is of medium importance because the focus of the project is not on usability. The system will support users with computer skills above average; the users have used a social network service before.

**Availability**
It should be accessible to users at all times. Uptime is important due to the collaboration and planning features of the system.

**Correctness**
It is important that the site handles input data correctly and that site functionality acts as intended.

**Testability**
Is important because we need to ensure that all modules are correctly satisfying the specification.

**Modifiability**
This is of medium importance because it will help adapt to changes during the process.

**Reusability**
It should be easy to reuse existing functionality all over the site.

**Portability**
The system will be implemented on one server and is intended to be run desktop browsers.

### A.1.5 User Stories

This is the list of user stories attached to the system. These describe what functionalities a user has when using the system.

#### Wall

In short, a wall is a place for friends of a user to post messages.
As a user of the system I would like to be able to ...

... post content on the wall.

... view content posted on the wall.

... view representations of references posted on the wall.

... be able to comment on wall posts.

... delete my posts posted on any wall.

... delete my comments posted on any wall.

As a owner of the wall I would like to be able to ...

... delete posts posted on my wall.

... delete comments posted on my wall.

#### Chat

A chat room may contain multiple users which can all chat at a same time.

As a user of the system I would like to be able to ...

... create a new chat room.

… invite new users to a chat room.

… receive join requests from other users asking me to join there chat room.

… re-join chat rooms that i have left, including empty chat rooms.

… send text messages to all the other users in the chat room.

… set a topic for the chat room.

### Events

An event is a date in a calendar which has a description and a name.
As a user of the system I would like to be able to …

… create an event

… view an event that I have created or is invited to

… edit an event that I have created

… cancel an event that I have created

… share an event that I have created with other users

… list all upcoming events I am attending

… list all previous events attended

### Files

A file may be any type of file which can be uploaded to the system.
As a user of the system I would like to be able to …

… create a folder.

… upload the file to a folder.

… download a file from a folder.

… view the files in a folder.

… delete a file from own folder.

… change display settings in a folder.

### Newsfeed

The newsfeed is a log, which displays actions executed by a users friends.
As a user of the system I would like to be able to …

… see updates from other modules.

… flick through pages of updates.

### Request

A request is something sent from a user to another user, it can be a friend request, or a group invite request etc.
As a user of the system I would like to be able to ...

... see requests from other modules.

... answer requests.

### User Profile and Authentication

A user profile is a page containing information about a specific user. Authentication concerns with registration, logging in and logging out.
As a user of the system I would like to be able to ...

... log in with an existing user, or register a new user.

... log out of the system.

... change the user profile information and choose to add more information, which was not possible at profile creation.

... view my friends profile pages.

... view my own profile page.

### Groups

A group is a collection of users.
As a user of the system I would like to be able to ...

... create a group (public or private) for some kind of student activity like a study group, social club, etc..

... invite other users to join the group. These other users can then choose to either accept or reject the invitation.

... request to join a specific public group. The members of the group decide whether the user is accepted into the group.

## A.1.6 Product terminology

### Authentication

An external user needs to register in the system to become a user of the system. This will make the user part of the system and make it possible for him/her to login to the system by entering his/her credentials. When logged in the user will be able to make use of all the functionality the system provides.

### User

A user should have a profile page with information about him or her. Users should be able to connect with other users and form friend-networks. Similarly, users should be able to join and be a part of a number of groups.

A user should have a calendar, which should display events from his own calendar and events from the groups the user is a part of, and a wall for communication. A user should have a personal file folder. A user should have a newsfeed, which displays notifications and wall posts from the groups and users the user is connected to. Each user should be able to apply a filter to the notifications in order to personalize which sorts of notifications should be displayed in the newsfeed.

The system should include restrictions of users' data and functionality, e.g. privacy. A user may have personal data on his profile, and this should only be viewable by friends.

#### Group

A group is a collection of users. A group should be either public or private; public groups can be joined by everyone, where private groups require an invitation from one of the members of the group to join. If the user is a member of the group, he has access to all functions within the group, as do all other members.

A group should have a calendar, which should display events tied to that specific group, and a wall for interaction between users in groups. A group should have a file folder that is shared among the users of the group.

#### Wall

A wall is a collection of wall posts. Each user and each group has one wall. A wall post should be either text, combined with images or embedded video. Users that can see a wall should be able to post wall posts and comment on all posts made on the wall. The user can delete any posts and comments on his/her own wall and own posts and comments on his/her friends walls.

#### Notification

Notifications should be a log that registers actions that are triggered by users. Examples of notifications include; X becomes friend with Y, X joins group Y, X is attending event Y, X requested to be friend with you, X commented on your wall post Y, X commented on the wall post Y that you commented on, etc. Only the users that are somehow implicated in the action logged should be able to view the notification created for the action. A user can be implicated in an action in many ways, he/she could be a friend of the user who triggered the action or he could have triggered the action himself etc.

#### File Folder

The file folder should contain user uploaded files. The owner of the file folder may change the display the display settings of a folder (Thumbnails and Gallery).

#### Calendar

A calendar should be connected to either a group or a user. A calendar should consist of a number of events. Users connected to a calendar, either directly or through a group, should be able to create events in that calendar. A user's calendar should only be accessible to friends of that user. A groups calendar should only be accessible to members of that group. When a user is invited to an event, that event should be added to his or her calendar.

#### Events

An event is a date in a calendar with a name and a description. Other users or groups should be able to be invited to an event, thereby sharing the event. Users invited to an event can accept or reject the invitation. A user can only view an event if he or she is invited to that event or the event is public created either by a public group or a user. Only the creator of an event can cancel and edit the event, as well as invite other users to the event.

#### Chat

Chat facilities should be available to all online users. A user should be able to create a chat room and administrate users in the chat. Chat rooms cannot be joined unless the user is invited to join the chat room and once a user is invited to a chat room he cannot be uninvited. Any user of the system can invite any other user of the system to join a chat room he himself created.

#### Search

The search function should be always be reachable for logged in users, and searches through the users, groups and events.

### A.1.7 Non-functional requirements

#### Hardware Interfaces

All modern computers with an Internet connection may be used for accessing the StudyLife system.

#### Software Interfaces

The Operating Systems can be any version of Windows, Linux, Unix or Mac which supports a modern web browser, such as Internet Explorer 8, Google Chrome and Firefox 3.

#### Communication Interfaces

The communication interface is a Global Area Network, meaning that the site is accessible from the Internet.

### A.1.8 Actors

#### Visitor

A visitor is an unregistered person of the system, who does not have access to any of the system functionalities.

#### Registered user

A user is a registered person who has access to the system functionalities.

#### Administrator

An administrator is a person, e.g a developer of the system, that has access to the database, and is therefore able to remove, insert and edit data directly circumventing the application.

### A.1.9 Personas

The intended user profiles for this system are the people attached to a university, since they might in some way share work, or other activities at the institution. The capabilities of these users range from novice to experienced user. They all want to use the system for either social networking at the university, or educational purposes.

### Lars

Lars is 22 years old university student studying law; he understands and speaks both English and Danish. He actively participates in courses. Lars has many friends at the university, partly because he is a volunteer bartender in the university bar and a top player at the soccer team. Lars uses a computer on a daily basis, and is an intermediate computer user. He is looking forward to using StudyLife for keeping in touch with all his classmates, and also to keep track of any schoolwork he might have.

### Lisa

Lisa is 27 years old university student who studies biology. She is a very bright young woman who is excellent at speaking English. Lisa enjoys her studies and attends them with great enthusiasm, which is why she is a A+ student. Lisa attends many extracurricular activities, and also attends many extra classes. Her schedule is usually fully booked, which is why she does not have much time outside of school for sharing notes with her classmates. She eagerly wants to use StudyLife for sharing notes with the other students. Lisa's only problem is that she is not very skilled at working with computers. Her only day-to-day use consists of text editing and a little bit of information search on the Internet.

### Hiroshi

Hiroshi is a 35 year old foreign PhD student working at the university, and functioning as a supervisor for several groups each semester. Hiroshi is originally from Japan, but has also studied some years in Germany. Hiroshi speaks a little bit of English, but he is progressing fast in his nightly English courses. Hiroshi is studying international marketing, and as a result of this Hiroshi is a very skilled computer user, he uses many different graphical processing programs on a daily basis through his research work at the university. Hiroshi has not been at the university for long, and is looking to make new friends around the university.

## A.2 Overall Design

### A.2.1 Class diagram

Figure A.1 shows a draft diagram of the classes and their relations.

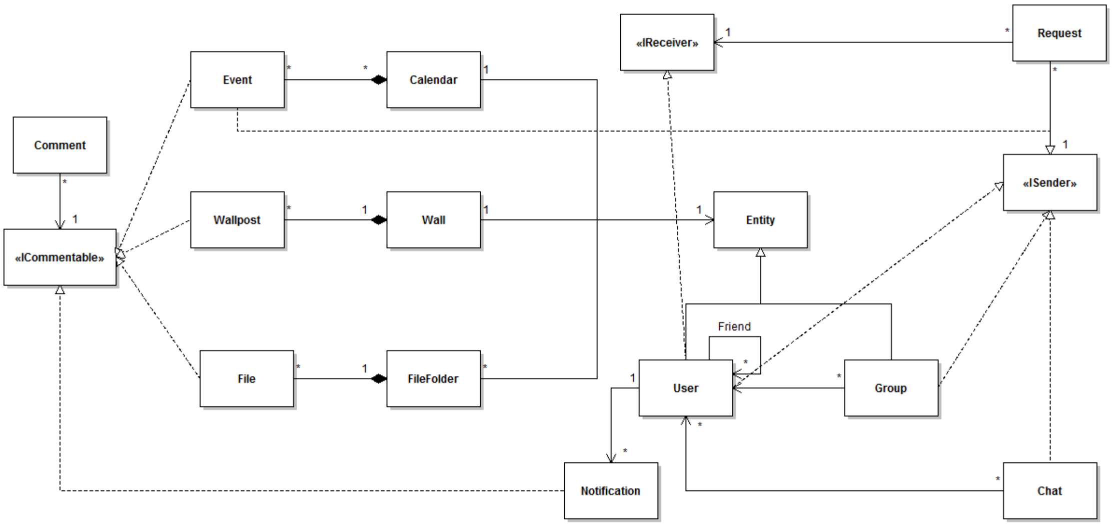


**Figure A.1:** Split in Components Architecture

### User Component

The user component will contain models and functionality regarding users in StudyLife. This functionality includes creating, viewing and edit users. All functionality that is to be used by other components is gathered in a user handler class for easy access. The user handler is defined in an interface to display to other components what functionality it will contain.

The user component will also be responsible for authentication. Authentication will contain functionality to authenticate and authorize users in StudyLife. The authentication part will contain functionality to register new users and functionality to authorize existing users by an email and password. The authorization part will provide functionality for applying access control to pages and functionality.

### Friend Component

The friend component will contain functionality to form and break friendship relations in StudyLife.

### Group Component

The group component will contain models and functionality regarding groups in StudyLife. This functionality includes creating, viewing, editing and joining groups. All functionality that is to be used by other components is gathered in a group handler class. The group handler is defined in an interface to display to other components what functionality it will contain.

### Calendar Component

The calendar component is responsible for handling all logic concerned with the previous described calendar and event product terms. The component must have an interface that lets other components get a view of a certain calendar. There must be two different views; the next 5 upcoming events for the user or group that calendar is connected to, and detailed month view of a calendar.

The component structure contains two classes, the event-class and calendar-class. Events provide functionality of creating and editing events and displaying events in a calendar.

### Search Component

The search component will include functionality to search the data of StudyLife. It is not to be used by other components, therefore no interface is defined.

### File Component

A file folder provides the functionality for a user to upload, organize and manage files inside the StudyLife system. The system supports a great variety of file types, if the files are images, these will be able to be viewed as thumbnails, or even in a gallery. Using this system, a user or a group will be able to share files with other users. It is not to be used by other components, therefore no interface is defined.

### Wall Component

A user and a group each have one wall. If a user is in some way affiliated with the group or wall, that user may post on that wall, or comment on other posts made. The wall system supports both text and images or videos in the same post.

### Comment Component

Provides the functionality for the user to comment on content in the system.

### Requests Component

The request component, provides an interface to the other components of the system, that allows them to post requests between users. A request is a question from one user to another, that requires an answer. The request component is also responsible for displaying pending requests to the users i.e. a list of questions that needs answering, and to redirect the answers to the requests to the components that created the requests.

### Chat Component

The chat component is responsible for providing instant messaging between users of the system. It does so by allowing the users to create chat room and invite other users to these chat rooms. A chat room is simply a list of messages that is updated in real time as the users post messages in the chat room.

### Notification & Newsfeed Component

The notification component is responsible of keeping the log and to expose an interface to the other components that enable them to access and store information in the log. The log is a list of user triggered events in the system. The notification component itself does not store any information in the log, it simply enables the other components to do so.

The newsfeed component is responsible for displaying relevant information in the log to the users of the system. Relevant information is information related to the user that is viewing the newsfeed. The newsfeed uses the interface provided by the notification component to retrieve the right log information, and displays it to the users.

### Master layout

Assembles the main layout of the web page by including the individual components.

### Database Access Layer

The database access layer is the communication layer between the physical database and the other components of the application. The responsibility of the database access layer is to map objects from the application with tables in the database and supply any component an easy and uniform interface to allow storing and retrieving objects from the database.

The database access layer structure consists of an interface which describes how the application components can communicate with the database, and an implementation of that interface.

## A.2.2 System Architectures

In the process of choosing an architecture we came up with 4 alternative ideas:

### Common Datastore Architecture

Every component share one common database which is responsible to hold all data of the system. Each component holds the views of the database that they need, and decide for themselves how they structure their respective modules. For instance, a user registration component would need a range of different data on a user, where as a calendar component may just need the user name and id, so each component would have their own representation of a user. What they would share would be a common interface to the data in the database.

The advantage of using this architecture, would be the independence of components, which are distributed among the groups, in how they choose to implement their specific component. Another ad-

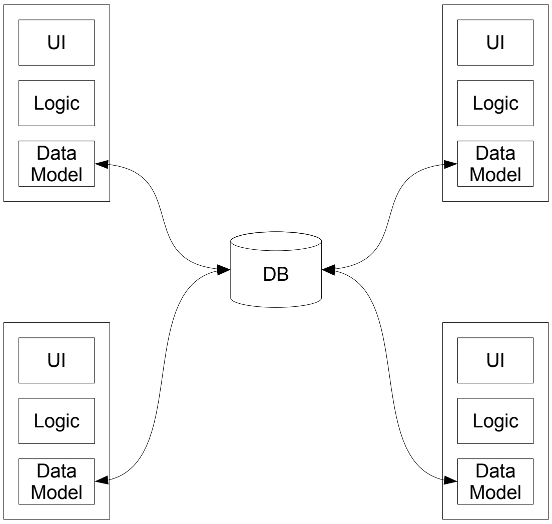


**Figure A.2:** Split in Components Architecture

vantage is, that the different components would not have to agree on any shared classes, merely what data that should be accessible for a given type.

The disadvantage is that all connections refers the database, so the different components need to create models that represents the other components to access the data, hence it would introduce a high degree of redundancy.

### Layered

The system consists of a number of layers: the database, models, logic, services for exposing the logic and at the top the components, which fetches data from the layers below.

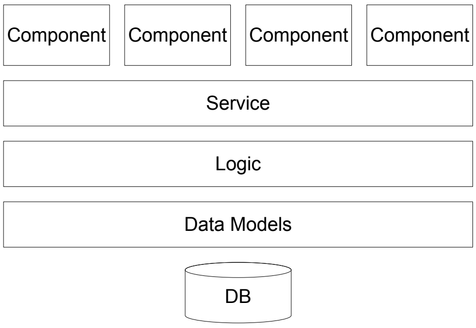


**Figure A.3:** Layered Architecture

The advantage of this architecture, is that the logic is centralized which introduces control over the data. Furthermore, the separation in layers is a separation of concerns, since each layer only communicates with the layer below and above.

The disadvantage is that there will be dependencies between the different components since some types will be used by multiple components. This in terms requires a lot more planning and contracting.

### Model-view-controller

The Model-view-controller architectural pattern. Models for representing the data in the database, views for presentation of the content to the user, and controllers for managing the logic in the system.

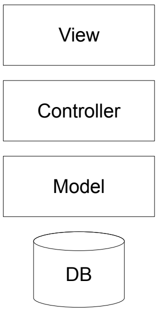


**Figure A.4:** MVC Architecture

The major advantage of the MVC architecture is that there is a great separation of concerns between what is presentation, what is model and what is logic, which makes it a lot easier to maintain and test. This also have the side-effect of promoting good design.

In cases of developing small web-applications, MVC may be overkill, since it requires a certain amount of setup and configuration.

### Web Service

A centralized application, that exposes interfaces via web services, which are available for whom that may need to use them. The architecture resembles the layered architecture. What divides the two is how the services are accessed. Using this architecture services are accessed through a web-service. By doing this it opens up for many different types of front-ends working with the core application such as mobile applications, web applications and ordinary desktop applications.

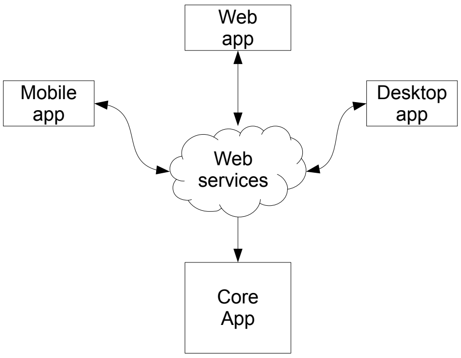


**Figure A.5:** Web Services Architecture

The advantage of this architecture is, that it enables us to allow so different types of applications to be developed for the core application. This is mostly an advantage since it could be really interesting.

The disadvantage is as with the layered architecture, that it introduce a lot of dependencies between the individual applications and the core application. Another disadvantage is the limitation currently presented when using web-services and transferring data objects in XML.

### Conclusion

The common advantages of these architectures are, that each of them gives the possibility of separating the application in several different components, where a large part of the component is completely independent of the other components.

We have chosen an architecture based on Common Datastore Architecture, MVC and the Layered Architecture, taking the best from both.

The architecture is designed in such way, that the functionality can easily be divided into independent components and extended with new ones. Splitting the system into separate components, made it easier to divide the implementation of the system among the groups as well as minimize the dependence in-between the groups.

All the individual components will make up a complete system with the use of the MVC pattern, allowing all the components to be responsible for their representation and allowing for flexibility and separation.

Dividing the system into separate layers, allowed for a unified way of interacting with the database by including a Data Access Layer, which handles all fetching and storage of data. The following presents the different layers.

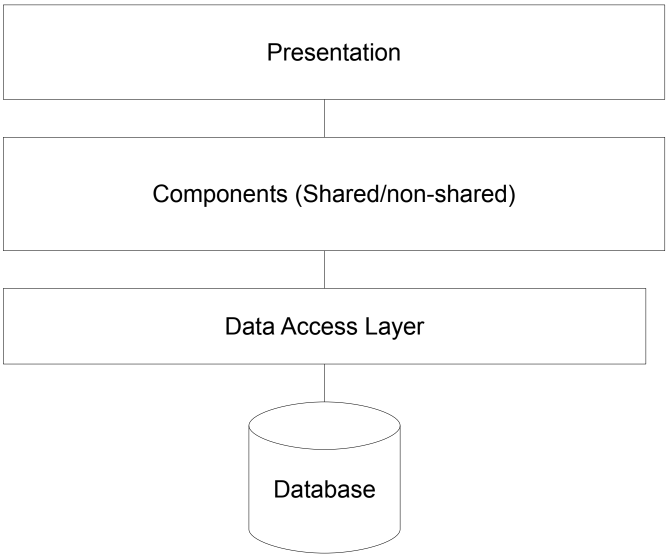


**Figure A.6:** Architecture of StudyLife

The Presentation layer is the glue that ties the system together and makes it available to the user by displaying it on web-pages. Using MVC enabled us to create separate views for each component. This made it possible to gather the system in the presentation layer simply by including the specific views suitable for the page being displayed. For example, a calendar can be represented as a large view containing an entire month, or as a small box showing the next five upcoming events.

The Data Access Layer (DAL) is responsible for a common way to save and fetch models to and from the database, in such a way that the components do not interact directly with the database.

The Component layer consists of the components, both the shared and the non-shared. The components are split into Model-View-Controller, see figure A.7. The interfaces are made available in this layer, such that it is possible for other components to implement; thus allowing non-shared components to access the shared functionality. There is no direct connection between non-shared components. All functionality that is to be used by all components is contained in the shared components.

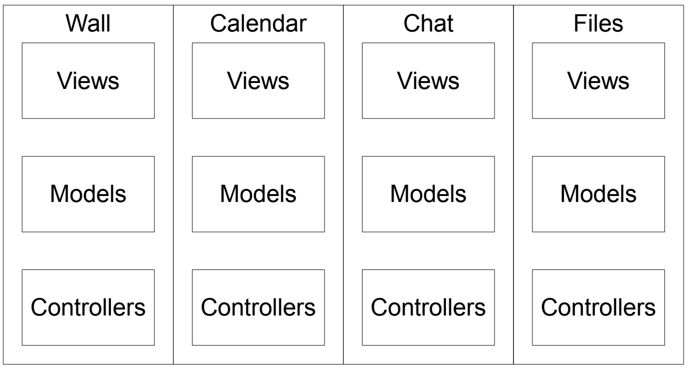


**Figure A.7:** Architecture of StudyLife's individual components

### A.2.3 Programming Languages and Frameworks

To select an existing development technology, some languages and its framework have been evaluated. The frameworks are: Zend, Django, Ruby on Rails, ASP.NET MVC.

All the evaluated languages are object oriented, the frameworks uses the MVC pattern, supports unit testing and object relational mapping for the database.

The following points out the pros and cons for the different frameworks.

**Zend (PHP)**

+ Platform independent
+ Large open source community
+ Huge documentation
+ Many supported IDEs
− No automation (generated code)

**Django (Python)**

+ ORM support
+ Many supported IDEs
+ Platform independent
− Not that known between group members

**Ruby On Rails**

+ Platform independent
+ Many supported IDEs
+ Easy setup
− No interface implementations

**ASP.NET C#/MVC 2**

+ Strongly typed language - some run - time errors avoided
+ ORM support
+ Well produced IDE

− Platform dependent

### Conclusion

Due to controversies, the choice of languages and framework was decided by voting. ASP.NET MVC 2 won by a single vote. The main argument among the ASP.NET MVC voters was that all the multi-group members had experience with C# and learning a new language would not have priority over the actual functionality of the system and collaboration between the project groups.

### Technical Requirements

Since ASP.NET MVC 2 was chosen, it was required to have a Windows server containing the web server and database. The database choice was MS SQL Server, as this made most sense in a Microsoft Windows environment.

### A.2.4 Division of tasks

All work must be divided between the participating groups. The components are distributed to the groups according to their cohesiveness, arranged such that, when possible related components will be assigned to the same group.

- Group s601a is responsible the User, Group and Search components and the master layout of the web page in ASP.NET.
- Group s602a is responsible for Notification, Request and Chat components and interfaces.
- Group s603a is responsible for the File, Wall and Comment components.
- Group s604a is responsible for the Calendar component and the database abstraction layer.

## A.3 Project Collaboration

### A.3.1 Version Control

To organize source code between groups, two version control systems was evaluated, namely Subversion (SVN) and Git. The following outlines the pros and cons of each system. Based on this list, we made a choice of which system to use.

**Subversion**

+ Well known
+ Branch and tag functionality

− Branches work as simple file copies which creates redundant storage

**Git**

+ Great for large projects with many developers
+ Decentralized
+ Made with focus on branching
+ Faster and requires less storage than subversion

– New Technology which require an effort to understand

**Conclusion**

Git was chosen for this multiproject because of it focusing on branches. The branching functionality in Git makes it easier to develop separate components in each sub-group and branch those into a collective stable branch. It will require an effort to learn how to use Git, but we view this multi-project as a opportunity to do so.

### A.3.2 Knowledge Sharing Tool

To organize the project development between groups, bug tracking and knowledge sharing systems are needed. We have chosen to look at the following systems: Bugzilla, Trac and Redmine. The pros and cons of each system are listed. Based on this list, we made a decision on which system to use.

**Bugzilla**

+ Well known
+ Easy to use
+ Cruise Control plug-in

– Pretty large

– No direct wiki support

– No Git support

**Trac**

+ Easy to use
+ Direct wiki support
+ Git plug-in
+ Cruise Control plug-in

**Redmine**

+ Easy to use
+ Direct wiki support
+ Direct Git support
+ Built-in calendar
+ Built-in Gantt charts
+ Very simple Cruise-Control like plug-in

#### Conclusion

Redmine was chosen for this multi-project, as it has all functionality needed. It is easy to set up, and it supports a wiki for knowledge sharing, which is its primary use in this project. Furthermore, we chose to use Redmine solely for knowledge sharing concerned with programming aspects, and Google Groups for meeting summaries and other non-programming specific aspects. Redmine also supports GIT.

### A.3.3 Common Build Server

The main repository will be situated on a server accessible to every developer. It is structured into branches. One main development branch and one for releases. When development on a new feature/sub system/component starts, a new branch should be created from the development branch (e.g. a wall- or chat-branch). When a component is at a state where it might be useful/interesting for other teams, the component branch should be merged into the development branch. When the development branch is stable it is merged into the stable branch. Each merge to development should include an expressive commit message.

When new content is committed to stable, the project is build to ensure that there are no errors.

### A.3.4 Project Management

#### Weekly Meetings

To control the multi project properly, weekly meetings should be held in order to discuss the overall progress. Each group will be represented by one group member, hence each meeting should have four attendees. One attendant is responsible for taking notes and for sending out a minutes to each group afterwards.

#### Procedure for Changes in Contract

The following describes the procedure of changing the development contract. Prior to the meeting, the group that are proposing a change to the contract must have clarified the specifics of this change, and how it will affect the other groups and their components. The group making the proposal must then explain the changes and the reasoning behind them. Based on this information, the meeting attendees will discuss the matter at hand, and make a decision on whether or not to accept the proposed contract change. In the case of an accepted proposal, it is imperative that the development contract is updated accordingly.

# Appendix B
# Test Specification - s601a

This test specification tests the components of group s601a, being user

## B.1 Test 1

### B.1.1 Test Manus Name

Test user adding a friend

### B.1.2 Test Manus Synopsis

Check that, when two users become friends, the right notifications are created.

### B.1.3 Test Manus Steps

**[X] Step 1**

Register a new user (will be referred to as user A) by clicking the register link on the front page. You are logged in and sent to your newsfeed. Click "My profile" and check the that the profile includes the earlier entered information. Log out.

Tester: "Works, I see my email, my birthday and my name"

**[X] Step 2**

Register a new user (will be referred to as user B) by clicking the register link on the front page. Go to Friends->Browse All Users and verify that you cannot access user A's profile.

Tester: "it works, I see a red text telling me that A is not my friend :("

**[X] Step 3**

Go to Friends->Browse All Users and request friendship with user A by clicking the request friendship image-link. Log out (user B).

Tester: "all good"

**[X] Step 4**

Log in as user A. Go to requests and verify that you have a request for friendship from user B. Click accept. Go to Friends and verify that you are now friends with user B. Click user B's name to verify you can see his profile. Log out (user A).

Tester: “no probs”

#### [X] Step 5

Log in as user B. Go to Friends and verify that you are now friends with user A. Click user A’s name to verify you can see his profile. Log out (user B).

Tester: “all good”

#### [X] Step 6

Log in as user A. Click the edit profile link and change your details. Verify that your changes are correct by viewing your profile. Log out.

Tester: “all good”

#### [X] Step 7

Log in as user B. Go to Notifications and verify that you have been notified that user A edited his profile.

Tester: “ok!”

### B.1.4 Conclusion

Did everything work as intended, or did you record any errors?

Tester: “everything worked as intended”

### B.1.5 Test executer and execution date

Jon 11-05-2010

## B.2 Test 2

### B.2.1 Test Manus Name

Test group add member

### B.2.2 Test Manus Synopsis

Check that, when a users joins a group, the right notifications are created.

### B.2.3 Test Manus Steps

#### [X] Step 1

Log on with user A. Make sure there is a user B which is friends with user A. Create a new public group. Log out

Tester: “all good”

**[X] Step 2**

Log in with user B, which is a friend of user A. Check that user B has received a notification that user A has created a public group. Request membership for the group. Log out.

Tester: “I cant click the group in the notification, but I got to it through the navigation menu. Everything else works”

**[X] Step 3**

Log in with user A Check that user A has received a request from user B to join the created group, and accept. Check that user B is now in the group. Log out Log in with user B and ensure that he is now in the group.

Tester: it works, but when I log in with user B I get the notification : “A has joined group <name of group>” shouldn’t it say B?

**[X] Step 4**

Log in with user A and go to the group profile. Edit the details of the group.

Tester: I cant edit the details, it says: “A group must have a contact.”, even though A is selected

**[X] Step 5**

Login as user B, and go to notifications. Check that user B receive a notification that the details were changed.

Tester: no notification, the details were not edited due the error in the previous step.

### B.2.4 Conclusion

Did everything work as intended, or did you record any errors

Tester: No i did not, and yes I did (see the comments above)

### B.2.5 Test executer and execution date

jon 11-05-2010

## B.3 Test 3

### B.3.1 Test Manus Name

Test User Profile page

### B.3.2 Test Manus Synopsis

Test that when a user is created, his profile page correctly displays working links to that users wall, calendar and filefolder.

### B.3.3 Test Manus Steps

**[X] Step 1**

Register a new user.

Tester: check

**[X] Step 2**

Go to your newly created user's profile.

Tester: check

**[X] Step 3**

Verify that there are working links to the users; calendar, wall and files.

Tester: check

### B.3.4 Conclusion

Did everything work as intended, or did you record any errors

Tester: yes it did, and no I did not.

### B.3.5 Test executer and execution date

jon+peter 11-05-2010

## B.4 Test 4

### B.4.1 Test Manus Name

Test Group Profile page

### B.4.2 Test Manus Synopsis

Test that when a group is created, its profile page correctly displays working links to that groups wall, calendar and filefolder.

### B.4.3 Test Manus Steps

**[X] Step 1**

Log in as a user. Go to groups and click "create new group". Give the group a name, description and do NOT set it to private.

**[X] Step 2**

Go to your newly created group profile.

**[X] Step 3**

Verify that there are working links to the users; calendar, wall and files.

Tester: no link to users

### B.4.4 Conclusion

Did everything work as intended, or did you record any errors

see above

### B.4.5 Test executer and execution date

jon+peter 11-05-2010

# Appendix C

# Bachelor Resumé (In Danish)

Temaet for dette semester var applikationsudvikling, hvilket lagde op at udvikle en applikation der løser realistiske problemer. Denne rapport dokumenterer derfor udviklingen af en social web applikation, kaldet StudyLife, der er udviklet for at gøre samarbejdet mellem studerende nemmere. Udviklingen af StudyLife er udført af en gruppe bestående af i alt 13 personer, fra fire forskellige projekt grupper. For at hver gruppe kunne bidrage mest muligt til applikationen, blev den delt op i en række komponenter, hvor denne gruppe blev tildelt: Master layout, User, Group, Friend og Search.

Rapporten beskriver udviklingen af disse, heriblandt analyse, design og implementation. Designet blev lavet på baggrund af kravsspecifikationen, og heri blev strukturen på master layout og en søge metode til Search valgt og User, Group, Friend specificeret. Implementationsdelen var baseret på designet og gik igennem implementationen af de førnævnte dele.

Igennem hele projektperioden er projektprocessen blevet vægtet højt og fokuseret en del på. Derfor er dette også dokumenteret i rapporten i form af udvælgelse af udviklingsmetode og en større refleksion af denne.

Efter implementering blev StudyLife testet både i form af generelle tests af på hele systemet, integration med de andre gruppers komponenter, og i form af tests på de specifikke komponenter nævnt ovenfor. Testene afslørede en række fejl, der blev rettet og et kapitel om tests blev tilføjet til rapporten.

Integrationen af de forskellige komponenter lykkedes, og de ønskede funktionaliteter virkede som tiltænkt. Kravene for projektet blev indfriet og i samme ombæring blev målene for dette semester nået.