# Rettelser

II

**The Faculties of Engineering, Science and Medicine**
**Software**
Selma Lagerlöfs Vej 300
Telephone +45 9940 9940
Fax +45 9940 9798
http://www.cs.aau.dk/

**Title:**
Hopla Helpdesk

**Theme:**
Programming

**Project period:**
SW3, fall semester 2010

**Project group:**
S305A

**Participants:**

Alex Bondo Andersen

Kim Jakobsen

Magnus Stubmann

Kristian Kolding Foged-Ladefoged

Lasse Rørbæk Nielsen

Rasmus Veiergang Prentow

**Advisor:**
Nadeem

**Page count:** 55

**Appendices count:** 0

**Finished:** 17/12–2010

**Synopsis:**

# Contents

# Part I

# Analysis

# 1 Task

*This chapter presents the subject, our goal, and how we planed to model the system. It will present the purpose of our project, our system definition which defines how our project should end up, and a context where the system is seen from a perspective of the end user.*

---

## 1.1 Purpose

When maintaining an office environment different problems are bound to arise. Computers will break down, printers will need installation and light bulbs will need replacement. Larger organizations will most likely have people hired to do this job exclusively. We develop this system with the purpose of easing the process of solving problems, as well as distributing knowledge as to how to solve trivial problems that will reoccur, and keeping track of already known problems. Thereby relieving the maintenance staff of these trivial tasks, effectively increasing the productivity of the employees.

## 1.2 Introduction

[1] We live in a world where we distribute the problems at hand to the people who are best at solving them. Sometimes this is a easy process, and other times, it can be an immense waste of time trying to find or contact someone who we can ask. Often, we end up solving problems ourselves, which someone else could have solved in, let us say one fifth of the time we used, at a lower cost. Not only did we waste our time, but we could have spent the time working with other things which we are particularly good at, keeping our work efficiency at a maximum.

Hopla Helpdesk is about solving this particular problem. Hopla Helpdesk purpose is – in short – to distribute specific problems to the group of people who are best at solving them, while solving other subproblems such as not overburdening one individual with all the problems etc.

[1] FiXme Dødelige: Læs vænligst dette igennem, og slet denne fixme, hvis du synes dette stykke tekst er fint.

Hopla Helpdesk does not only serve as a one-way information stream about existing problem, but also delivers a system where the people who solve the problems can report back to the client with solutions or questions regarding the problem. All this while other clients who might have a similar or identical problem, can subscribe and thereby follow the problem from the beginning to the end.

The result of using Hopla Helpdesk this way, is a database full of problems, with – hopefully – attached solutions. Hopla Helpdesk uses these informations in a preventive way, which displays possible solutions to a client, and in some cases, these solved trivial problems with attached solutions might help the client instantly, keeping the workload of the people solving problems down, as well as the clients spent time to a minimum.
Productivity goes up, wasted time goes down.

Comprehensive knowledge of the OOA&D method is assumed.

## 1.3 System Definition

To better be able to define our system we will use the FACTOR method[7, p. 39]. This model contains different parts of the requirements, and it will help us derive the system definition.

**F**unctionality
Our program will contain features aimed at easing the problem-solving process for service employees. Therefor the system will have the following features:

- Ranking of problems
  Unsolved Problems are displayed on a ranked list, depending on their importance.
- Keep track
  The system should be able to keep track of all the information regarding a problem, this being
  - When it approximately will be solved
  - Who is assigned to solve the problem
  - What the deadline is, if the staff approved it, and if it is exceeded
  - What tags and categories are related to the problem
  - Comments related to the problem
- Knowledge saved
  The system will focus on the ease of use for all users, saving and suggesting solutions to problems before a similar problem is submitted. Furthermore the system should be able to run on all systems without any prior software installation.[2]
- Statistics
  Allow the supervisor to monitor the work process of the employees. [3]

[2] FiXme Dødelige: hvorfor staar der noget om platform under Knowledge saved?
[3] FiXme Dødelige: Indskriv flere features

**A**pplicationdomain
This system will be applicable to office environments which deals with solving of problems.

**C**onditions
The problem submitting users are not required to have any expert knowledge to use the helpdesk. The service personal will have to learn to use the staff interface to solve problems. The administrators will have to learn to use the functionality in the administrator interface.

**T**echnology
In the development of our program we are going to use the programming language C# together with the Model-View-Controller framework 2 that exists within the ASP.NET framework[4]. We will set up a development server and use AnhkSVN and Microsoft SQL server along with visual studio 2010.

The end result will be a web interface running on a webserver, with underlying SQL database.

**O**bjects
Problems, solution, clients, staffmembers, admins, departments, categories and tags.

**R**esponsibility
Our system is responsible for keeping track of all technically related problems within an organization. It is also responsible for distributing tasks amongst employees and supplying statistics on their progress to their supervisors. Finally it is also responsible for enabling users and technical staff to communicate about a problem and the following solution.
[5]

Using these FACTOR criteria, we arrive at the following system definition:

- The system should be web-based so that we can create an application capable of running without prior installation.
- The system should contain prioritized tags, enabling us to determine the importance of problems
- The system should keep track of the problems Estemated time of compleation, comments and what employee is responsible for the problem. This will enable us to create detailed statistic about problem solving efficiency and individual employees.[6]
- The system should be able to save and suggest prior problems and their solutions to users.

[4] FiXme Dødelige: er mvc 2 et framework i asp.net frameworket?
[5] FiXme Dødelige: skal vi aendre users, employees og supervisors = clients, staff og admin ?
[6] FiXme Dødelige: skriv hvad statistics skal gore

- The system should take into account, the workload of the employees to be able to estimate how long a problem will take to finish.
- The system should only use dynamic data, in order to run in different environments.
- The system should contain a structure that can handle problems, categories, tags, employees, supervisors, ordinary users and department.

## 1.4 Context

Our system will target a work environment or an educational environment. The analysis and design process will be based upon a university, but the system should be easily implemented at any other work environment.

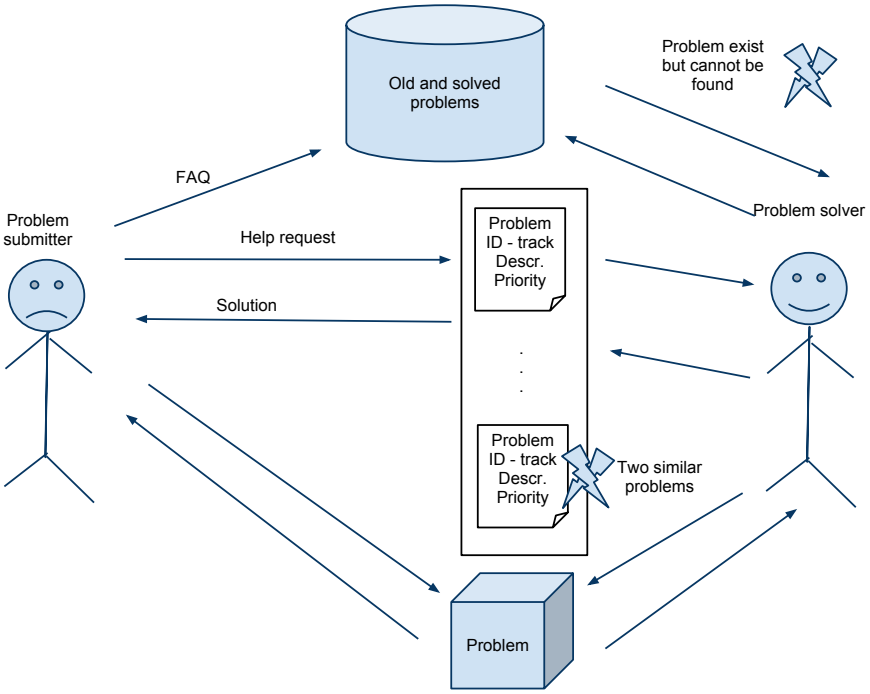


Figure 1.1: *Rich picture of our system*

To fully understand the context we draw a rich picture which can be seen on figure 1.1. The central aspects on the rich picture is that both the problem submitter and the problem solver can act on the problem, and will communicate through the system. From the rich picture we see a few conflicts in the environment:

- The first is that problem exist but cannot be found.
- The second is that there could exist two similar problem.
- The central objects in the system are the problems and solutions.

### 1.4.1 Problem Domain

A central phenomena in our system will be to add a problem to the system. This will occur when a client finds a problem in the organization and wants to submit it, in order to get it solved.

Another important phenomena in our system is when a problem is solved. This is initiated by a staffmember and will result in a notification to the clients who are subscribing to the problem.

Chapter 2 gives a detailed analysis of the problem domain and will elaborate on the phenomena.

### 1.4.2 Application Domain

The people who will be acting on the system is the problem solver and submitter. Most likely the submitter will be a student in a university environment and the solver will be the technical staff. But the submitter could be a staff as well. Beside the two main actors there is an admin who can administrate the staff, clients, and the system itself.

A detailed analysis of the application domain is presented in chapter 3, where the actors and there use cases are further described.

---

*The purpose of this project is shown in this chapter along with the system definition and the context of which the system should be viewed.*

# 2 Problem Domain

*In order to determine the nature of our solution we have to analyze the context in which it is going to be used. The classes and events described in this chapter are reflections of how the system is in the physical world. This chapter also includes a description of the behavior of each class, in order to understand the different classes better.*

---

## 2.1 Classes and Events

In the process of finding classes and events we made up several classes and events some of which are not used in the actual program. To come up with the classes and events we talk about different possible scenarios and based on that we developed our class diagram. This section describes all the classes and events in the final iteration of the process and all the redundant classes and events are omitted. On figure 2.2 the relations between classes and events are illustrated.

### 2.1.1 Classes

The following is a list of the classes from the class diagram 2.1.

**Problem** Problem is the basic building brick in our system. It is used to hold information about the specific problems which it represents.

**Deadline** The deadline can be approved or not approved and is a part of the problem.

**Comment** A comment belongs to a problem.

**Solution** Contains state and information about a given solution

**Person** This contains attributes about the users in the system, name, address, phone number ect. We will look into the details later in the report.

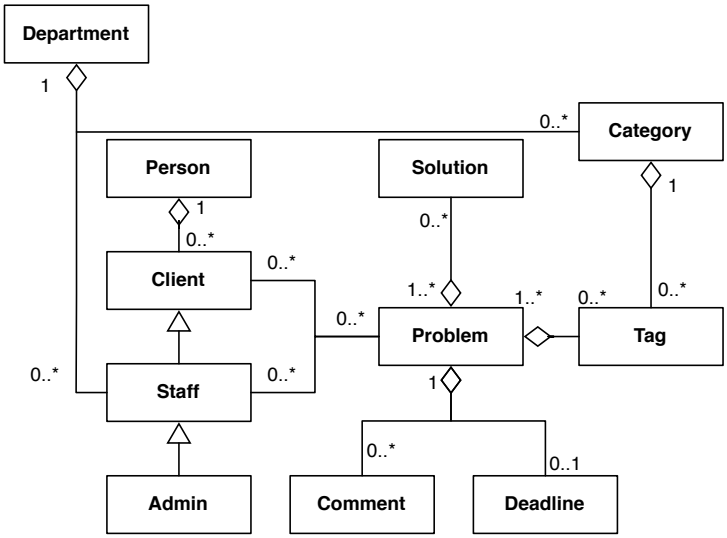


Figure 2.1: Class diagram

**Client** A client can be subscribed to a problem and be associated with the comments that the actor client posts.

**Staff** The staff class inherits from the client class. staff can be assigned to problems.

**Admin** The admin is not associated with any other classes and is only used to administrate the users.

**Department** Contains information about staff and categories.

**Category** Categories contain Tags

**Tag** Tags is used to determine the problems category and thereby department.
[1]

## 2.2 Structure

The class diagram is based on the classes and events, which was described earlier in chapter 2.1. The class **person** aggregates a role. The role class itself is removed since only **client** inherited from it. **staff** inherits from **client** and **admin** inherits from **client**. Alternatively the role pattern [7, p. 80] could have been used. This prescribes that **admin**, **client**, and **staff** all would have been a subclass of *role*. We considered that it made more sense that they inherited

[1] FiXme Dødelige: slåevents sammen, tilfœg de nye events, flere classes se klass diagram, fjern urelevante, kald den general, kontroller at checkmarks er sat rigtigt (department)

| | ***Classes*** | | | | |
|---|---|---|---|---|---|
| ***Events*** | Problem | Solution | Staff | Department | Person |
| Problem added | ✓ | | | ✓ | |
| Problem solved | ✓ | ✓ | | ✓ | |
| Problem updated | ✓ | | ✓ | ✓ | |
| Problem assigned | ✓ | | ✓ | ✓ | |
| Problem unassigned | ✓ | | ✓ | ✓ | |
| Problem deleted | ✓ | | | ✓ | |
| User replied | ✓ | | ✓ | ✓ | |
| Staff replied | ✓ | | | | |
| Department created | | | | ✓ | |
| Department closed | | | | ✓ | |
| Role assigned | | | | ✓ | ✓ |
| Role unassigned | | | | ✓ | ✓ |
| Person created | | | | | ✓ |
| Solution found | ✓ | ✓ | ✓ | | ✓ |
| Solution assigned | ✓ | ✓ | ✓ | | ✓ |
| Problem categorized | ✓ | | | ✓ | |

Figure 2.2: *Problem-domain analysis event table*

from each other, since all the privileges **staff** has the **admin** has as well. Same applies for **staff** and **client**.

For the class person different relations appear depending on his role.

- **clients** can subscribe to **problems**.
- **staff** can be assigned to **problems** and **staff** belongs to a department.
- **admin** does not have any relations. But is necessary for administration of users.

A problem consist of none or many *comments*, none or many *solutions* and, one or many *tags*. A **tag** belongs to a category and a **category** belongs to a **department**. The class diagram is illustrated at figure 2.1

## 2.3 Behavioral Pattern

In this section of the report, we will describe the behavior of the two major classes **Problem** and **Person**. These two are the only classes which can have more states.

### 2.3.1 Problem

Figure 2.3 shows the behavior of a problem. Note that you can solve the problem by attaching one or more solutions. A problem can have an unlimited number of solutions. You can at all times delete the problem, thus the arrow points from the edge of the box.

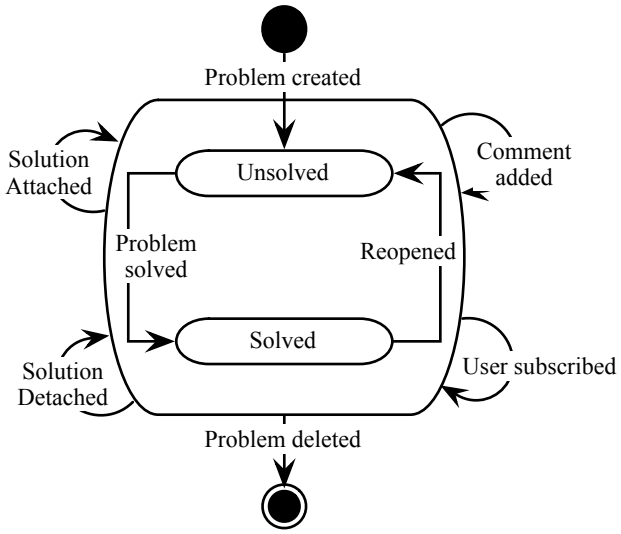


Figure 2.3: *The statechart of the problem class*

### 2.3.2 Person

As shown in figure 2.4, a person is assigned a role in the system as soon as he is created. He can only have one role at a time. The roles inherit from each other. Meaning that a admin can do the same as the client, but clients can do the same as admin.

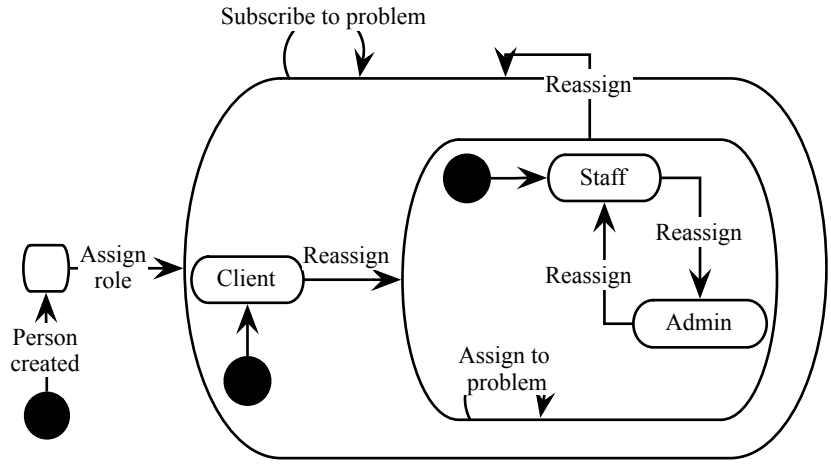


Figure 2.4: *The class person's statechart*

---

*The classes and events which will be used to model the reality are described in this chapter. These descriptions includes a definition of each class and event, the structure in which the classes resides, and the behavior of each class.*

# 3 Application Domain

*In this chapter the application domain will be analyzed and our choices regarding the application domain explained. The purpose of this chapter is to determine the system's usage requirements.*

## 3.1 Usage

We have identified three actors and picked out the six most relevant use cases. The three actors are: client, staff, and admin. The use cases we picked are: Submit problem, My problems, Worklist, Solve problem, Administrate, and Statistics.

| | ***Actor*** | | |
|---|---|---|---|
| ***Use case*** | Client | Staff | Admin |
| Submit problem | ✓ | ✓ | ✓ |
| My problems | ✓ | ✓ | ✓ |
| Worklist | | ✓ | ✓ |
| Solve problem | | ✓ | ✓ |
| Administrate | | | ✓ |
| Statistics | | | ✓ |

Figure 3.1: *Actor & use case table*

Figure 3.1 shows the relationship between use cases and the actors of our system. All three roles are able to "Submit problems" and see "My problems". The staff and admin have a "Worklist" and can "Solve problem". The admin can "Administrate" the system and access the "Statistics"

The use cases in figure 3.1 are described in subsection 3.1.2.

### 3.1.1 Actors

The system has two primary actors: client and staff. Client is the lowest privileges human actor, his primary use case is to submit new problems. The client

is also able to track his problems and communicate with the staff.

The staff members are more privileged and can solve problems. All staff members can act as clients if they themselves has a problem which should be addressed to another department. E.g. the lightbulp in the IT-administrators office is broken and the maintenance officer should fix it.

These two actors are described in details in figure 3.2 and 3.3. Beside staff and client the system has another actor called admin. The admin is a more privileged staff or a manager of the staffmembers.

The admin can manage tags, categories, departments and view statistics of each staffmember, the actor admin is decribed in details in figure 3.4.

| ***Client*** |
| --- |
| **Goal:** A person who has a problem, and his goal is to get his problem(s) solved.<br>**Characteristics:** The clients are employees or students with different knowledge and experience with similar systems.<br>**Examples:**<br>Client A prefers face-to-face communication with the working staff whenever he has got a problem.<br>Client B prefers web/mail communication as a substitution of face-to-face communication so he does not have to leave his working space in order to get help. |

Figure 3.2: *Description of the actor client.*

| ***Staff*** |
| --- |
| **Goal:** The staff solves the clients problems and use the system as a taskmanager.<br>**Characteristics:** The staff are employees and has various levels of technical knowledge.<br>**Examples:**<br>Staff A prefers to speak to his manager and the client face-to-face.<br>Staff B enjoys getting his daily tasks from a computer system, |

Figure 3.3: *Description of the actor staff.*

### 3.1.2 Use Case

The use cases from figure 3.1 are described below.

| ***Admin*** |
| --- |
| **Goal:** The admin is system and staff manager.<br>**Characteristics:** Responsible for the Hopla Helpdesk or staff manager.<br>**Examples:**<br>Admin A is a software engineer and are responserable for maintaining the system by management departments, category, and tags.<br>Admin B is the boss of the department and evaluate staffmembers based on the statistics generated by the helpdesk. |

Figure 3.4: *Description of the actor admin.*

**Submit problem** The use case submit problem is only used by the actor client. Except for cases when staff or admin acts as client. A use case diagram is shown in figure 3.5.

- **Use Case:** Submit problem is initialized when a client has a problem and wishes to submit that problem to the system in order to get help from the staff. First he has to select a category and choose one or more tags, he can select tags from more then one category. When the client is done selecting tags the system compares the selected tags with other problems. If similar problems is found the client is presented with these. If one of these matches his particular problem, he can subscribe to the problem if it is unsolved or read the solution(s) if the problem is closed, thus hoping that this will lead to solving the clients problem. If no similar problem was found the client creates a problem with a title, description and the previously selected tags. Hereafter the problem gets assigned to a staff.
- **Objects:** Problem, tag, category, client, staff.
- **Functions:** Search existing problems, compare problems, create problem, subscribe to problem. [1]

**My problems** The use case my problems is used by clients, staffs, and admins. It show a list all problems submitted by the user. From there details can be viewed for each problem.

**Statistics** The use case statistics is accessible by the admin. It shows statistics about how much time each staff use to solve problems.

**Solve problem** The use case solve problem is the staffs primary usage of the system. When the staff get his worklist he can select a problem and solve the problem. A statechart diagram is shown in figure 3.6.

- **Use Case:** The use case is initialized when the staff wants to check his worklist. The staff is then presented with a list of unsolved problems assigned to him. The staff can then click on one of the problems to read

[1] FiXme Dødelige: fix saa det stemmer med functions listen, naar den er blevet opdateret

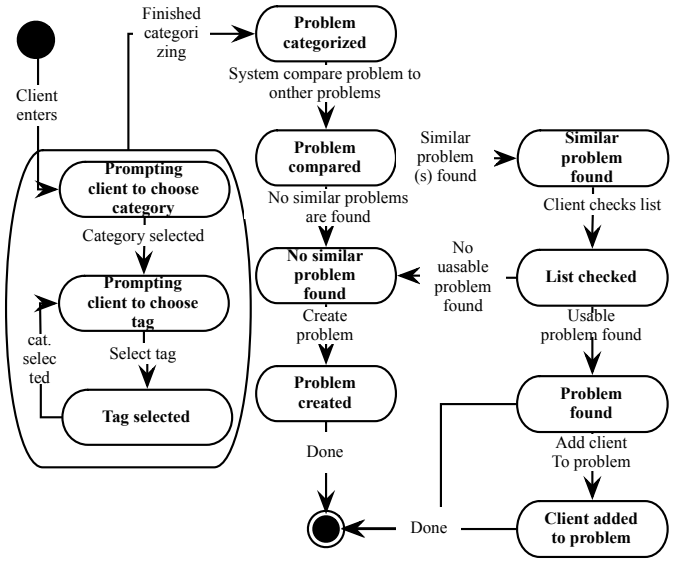


Figure 3.5: *A state chart diagram of the use case submit problem.*

the problem, see status of it, add comments to it, search the database for similar problem, reassign it, or write a new solution.

- **Objects:** Problem, solution, comment, client, and staff. fixmestaff must at least have the same objects as client

- **Functions:** Add comment, reassign, change status, search database, create solution, attach solution and get staff worklist.[2] [3]

**Administrate** The use case administrate is used by the admin to administrate persons, tags, categories, departments and view statistics about the specific staffmembers. A statechart diagram is depicted on figure 3.7.

- **Use Case:** The use case starts as the admin enters the site. The admin can manage people by change role, department, email and delete the person from the system. The admin is also able to create new departments and add categories thereto. To each category new tags can be added. Tags have a priority which can be changed at anytime. Tags cannot be removed when they have been used, but they can be hidden so they cannot be used to categories new problems.

- **Objects:** Staff, Client, category, tag, department. [4]

[2] FiXme Dødelige: make it right, when the functions are updated.
[3] FiXme Dødelige: staff must at least have the same functions as client
[4] FiXme Dødelige: admin must at least have the same objects as client

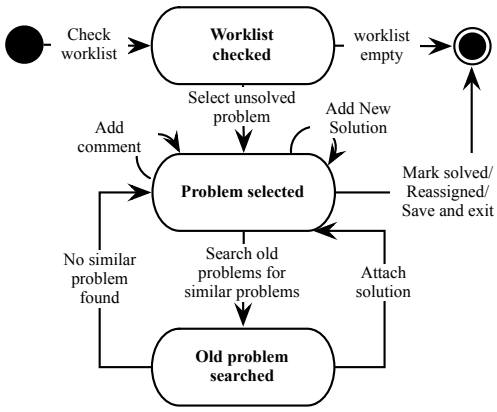


Figure 3.6: *A state chart diagram of the use case solve problem.*

- **Functions:** Delete person, Add person, create department, set permissions, delete department, set priority, add category, delete category, set tag visibility, set category visibility, create tag, delete tag. [5] [6]

## 3.2 Function

The purpose of this section is to "determine the system's information processing capabilities" [7, p. 137]. Ultimately resulting in "a complete list of functions with specification of complex functions." [7, p. 137] All functions with a medium or complex complexity are explained below.

**Search for problems** This function searches the model for problems with specific tags, then presents them in an ordered list assorted after problems with most similar tags. It is both *read* and *calculate* because it reads in the model and compute which problems are most similar. This function is *complex*.

**Get Average Time Consumption of a Tag** This function calculates the average time consumption of a tag. It has to divide the number of problem solved and the total amount of time all problems with this tag required to be solved.

**Manage Tag Time of Problem** This function takes the time a staff has used to solve a problem and adds to the related tags time consumption and increments the solved problems property.

[5] FiXme Dødelige: admin must at least have the same functions as client
[6] FiXme Dødelige: admin must at least have the same functions as client

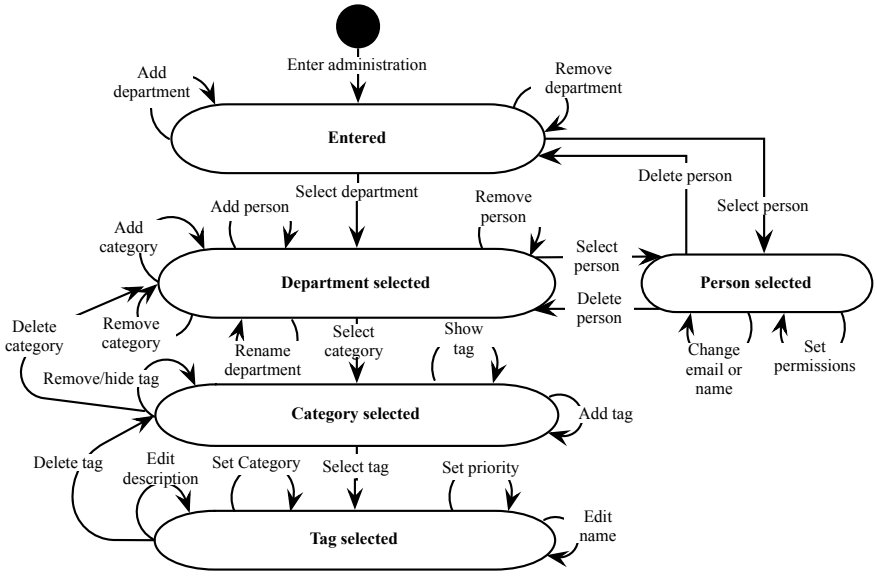


Figure 3.7: *A statechart diagram for the use case administrate.*

**Get Estimated Time Consumption of Problem** This function calculates the amount of time a specific problems acquires to be solved. This is based upon the related tags. The tags has a property describing the average time consumption.

**Get Expected Time of Completion of Problem** This function takes all the problems of the assigned staff member which are expected to be solved prior this problem and adding up the amount each is estimated time each problem will consume.

**Get Sorted Worklist** This function returns the work list of a staff member in sorted order after priority.

**Reset person password** If a person forgets his password, then he can get a new password send to his mail. The password is randomly generated. This function updates the model to match the computed password. This function is *medium*.

**Balance workload** This function compares the workload of staff and distributes their problems equally among other staffmembers in the same department. This function calculate the workload of staffs and then moves problems in the most optimal way. This function is Complex.

**Get statistics** The statistics function calculate how long each staff use to solve a problem in average. The function can also find the total average for departments. This function is medium.

| Name | Complexity | Operations |
|---|---|---|
| Get problem tags | Simple | Read |
| Search for problems | Complex | Read/Calculate |
| Create problem | simple | Update |
| Reassign staff to problem | Simple | Read/Update/Signal |
| Subscribe / unsubscribe to problem | Simple | Update |
| Attach / detach solution to problem | Simple | Read/Update |
| Manage Tag Times | Medium | Update/Calculate |
| Get Problem Est. Time Consumption | Medium | Read/Calculate |
| Manage Tag Times | Medium | Update/Calculate |
| Get Problem Est. Time Consumption | Medium | Read/Calculate |
| Get Problem Expected Time of Completion | Medium | Read/Calculate |
| Get Tag average time consumption | Simple | Read/Calculate |
| Get sorted worklist | Medium | Read/Calculate |
| Approve deadline | Simple | Update |
| Create department | Simple | Update |
| Create category | Simple | Update |
| Create tag | Simple | Update |
| Hide / show category | Simple | Update |
| Hide / show tag | Simple | Update |
| Delete person | Simple | Update |
| Reset person password | Medium | Update/Signal |
| Balance workload | Complex | Calculate |
| Get statistics | Medium | Read/Calculate |
| Distribute Problem | Medium | Read/Calculate/Update |

Figure 3.8: *Function list*

## 3.3 User Interfaces

Our system's user interface is divided into three sub-interfaces – one for each human actor. These interfaces are described in the following subsections.

### 3.3.1 Client Interface

The clientinterface is illustrated in figure **??**. After login, the client is presented with the *main* window.

#### Main

The clients main window allows the clientto choose what to do in the helpdesk system. The "Main" window have three buttons:

- The "Commit Problem" button which sends the user to the "Categorize new problem" window.
- The "My problems" button which sends the user to "Search" window.
- The "Search for problems" button which sends the user to "Search" window.

#### Search

The search windows is used to search for problems containing specific tags or problems posted by the client self. The "Search" window contains the following elements:

- The "Categories and tags" frame, see paragraph below.
- A settings panel.
- A search button that triggers the search, see paragraph below.
- A list containing the problems matching the settings and tags.

A problem can be double clicked to view details about the problem in the "Client problem view" windows.

**Settings** The "Settings" frame contains configurations options for the search. The frame contains the following elements:

- My problems
- Unsolved problems
- Solved problems
- Number of search results

When a search is preformed these options are checked and they will effect the result of the search.

**Categories and tags** This frame is used in a few other windows as well. The frame contains:

- A list of categories and the tags attached to it.
- A checkbox for each tag.

### Client problem view

This window contains information about the selected problem. The window contains the following:

- A info field which contains relevant information about the problem e.g. the description of the problem.
- A subscription check-box, clients subscribed to the problem will receive notifications when then problem changes.
- A text field with all the previous entered comments if any.
- A field to write new comments in.
- The button "Submit comment" which submits the comment.
- A field with non or more solutions written by the staff members.

### Categorize New Problem

To add a new problem the client need to select the tags which best describes the problem. The window contains the following elements:

- The frame "Categories and tags".
- The button "Search for this kind of problems" which sends the client to the window "Search for similar problems"

### Search for similar problems

A search is preformed with the selected tags from the previous window, a list with similar problems are displayed. This window includes the following:

- The window "Search" with all it elements.
- A button called "No problem suffice" which loads the window "Create new problem".

The found problems can be double clicked and the window "Client problem view".

**Create new problem**

The "Create new problem" window's purpose is to describe, categorize, suggest deadline and submit the problem. The window contains the following elements:

- A text field where information about the problem can be entered.
- The frame "Categories and tags" enables the clientto change and add tags so the problem is best describe.
- A calender where a deadline can be suggested to the staffmember.
- "Create" submits the problem and send the user to the "Client problem view".

### 3.3.2 Staff Interface

The staff interface is illustrated in figure 3.9. After login, the staff is presented with the *main* window.

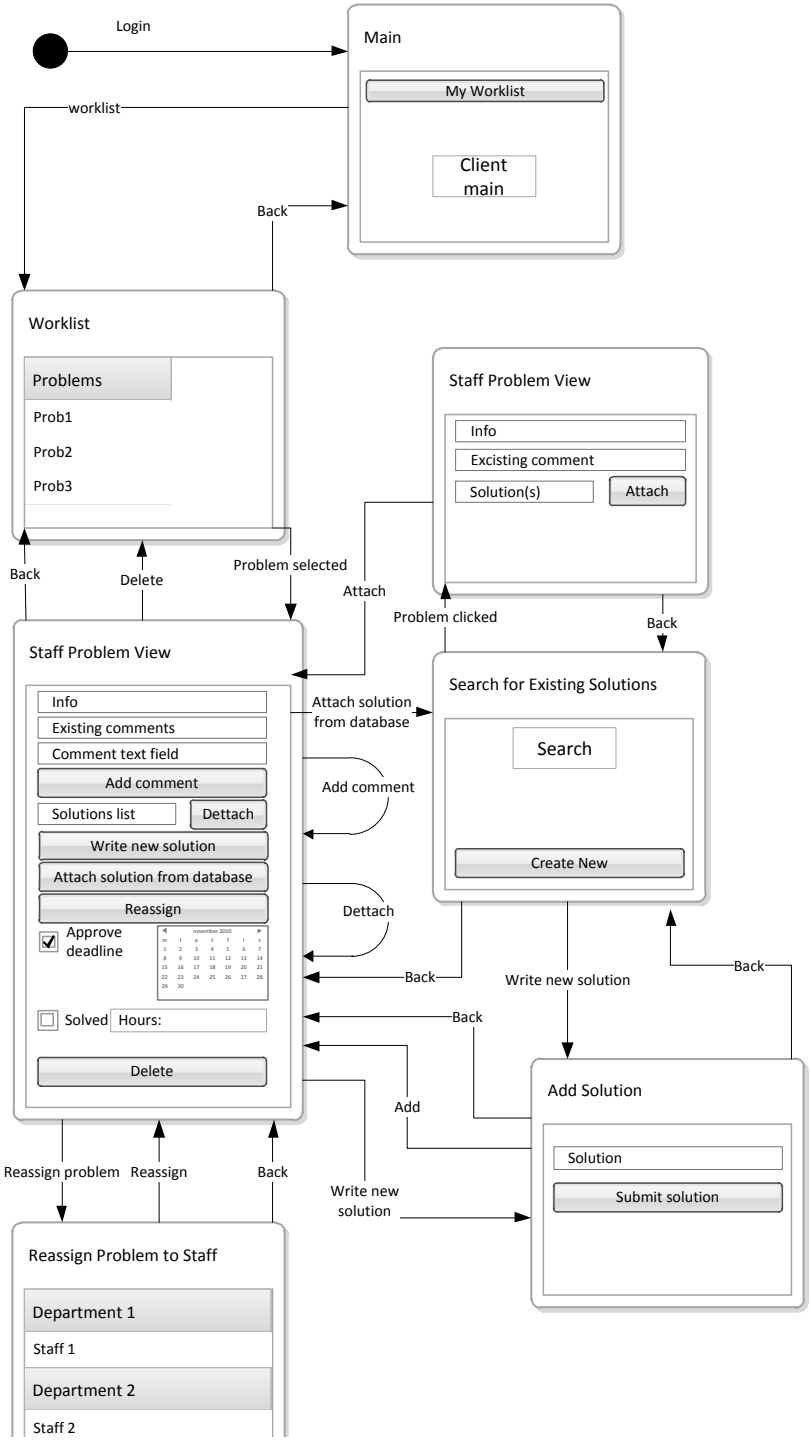


Figure 3.9: *Staff Interface*

### Main

The staff main window give the staffmember access to all the functionality from the staffactor and from the clientactor. The staffhave one button:

- "My Worklist" which directs the staffmember to the "Worklist" window.

plus the menu button from the clients main menu.

### Worklist

The "Worklist" is a list of all the problem assigned to a specific staffmember. The window has the following elements:

- A list with all the problems assigned to the staffmember who is signed in.

The list show the following properties: Name, Deadline, Priority, and ETC. When a problem is clicked the window "Staff problem view" opens.

### Staff problem view

This window shows all the information related to a specific problem. The "Staff problem view" features the following:

- The window contains a text field which contains information about the problem
- A text field with all the existing comments related to the problem.
- A text field where new comments can be entered.
- The button "Create comment" to send the entered comment.
- A text field containing none or more solutions.
- The button "Attach solution from database" which send the staff to "Search for existing solutions".
- The "Reassign" button which opens the "Reassign problem to staff" window.
- A checkbox to mark the problem as solved.
- A calender to set deadlines.
- The checkbox "Approve deadline" approves a suggested deadline if one is suggested.
- The button "Write new solution" which sends the staff to "Add solution"

#### Search for existing solutions

This window enables the staff to search for existing solutions among existing problems. The "Search for existing solutions" window contains the following elements:

- The "Search" window from the client interface 3.3.1 is reused, and it enables the staff to search for problems.
- The button "Create New" which sends the staffto the "Add solution" window.

If a problem is double click the staff is send to "Staff problem view".

#### Staff Problem View

The "Staff Problem View" show properties of the selected problem. This window contains the following:

- A text field with relevant information about the problem.
- A text field with all the existing comments related to the problem.
- A text field containing none or more solutions.
- The button "Attach" inserts the selected solution into the solutions list from "Staff Problem View", and the staff is send to the "Staff Problem View" window.

#### Add Solution

The "Add Solution" window allows the staff to enter a new solution. The window contains the following elements:

- A text field where the new solution can be entered.
- The "Add" button which sends the entered solution to the Solution list in the "Staff Problem View" window, the staffis send to the "staff Problem View" window.

### 3.3.3 Admin Interface

The admin interface is illustrated in figure **??**. After login, the admin is presented with the *main* window. All the window have a back function which enables the admin to return to the previous window.

#### Main

The "Main" adminwindow gives access to administration options and the functionality from the staffand clientmain windows. The window contains:

- The button "Manage Deparments" directs the admin to the "Department Administration" window.
- The button "Manage People" directs the admin to the "Person Administration" window.

- The button “statistics” directs the admin to the “statistics” window.

plus the buttons from the staff and client “main” window.

#### Person administration

When the “person Administrate” window is opened, the adminis able to browse through all staffmembers. The window contain the following element:

- A list with all the persons in the system.

#### Department administration

This window shows a list of staff and categories for the selected department. The “Department administration” contains the following:

- A dropdown menu where department can be selected.
- A list of staff members who are a part of the selected department.
- A list of categories which is related to the department.

When the staff is clicked the admin is send to the “person view”. When a category is clicked the admin is send to the “Category view”.

#### Person view

The “person view” shows information about a selected person. The window contains the following elements:

- A info field where relevant information is displayed about the person
- A dropdown where roles can be set.
- A dropdown where departments can be set.
- The button “delete” which deletes the person and sends the user back to the previous window.

#### Category view

The “Category View” window shows information about the selected category and allows it to be modified. The window contains the following:

- An info field where relevant information is displayed.
- A list with all the tags belonging to the category.
- The “Create new tag” which send the admin to “Tag View” window.
- The “hide” button which hide category and its tags.

Each tag have an “edit” button which allows a tag to be modified, the button sends the adminto the “Tag view” window.

**Tag view**

This window is used to create new tags and edit already existing tags. It contains the following:

- A info field which can be edited.
- An "Hide" button which hide the tag.
- An "save" button which save the changes made.

When the hidden button is pressed then the admin is directed back to the "category view" window.

---

*This chapter shows the Application Domain Analysis for our system. First the usage of our system is described, then the functions which are used to manipulate data in Problem Domain or signal actors in the Application Domain is presented and defined, lastly the User Interfaces are described.*

# Part II

# Implementation

# 4 Development

*Top*

## 4.1 Development Tools

In the creation of our web application we use some development tools which will be described in the following section.

### 4.1.1 IDE

The primary development tool has been Microsoft Visual Studio Ultimate [6], which include a large variety of inbuilt tools for unit testing, SQL and data modeling. None group members had previous experience with development using Visual Studio Ultimate. The alternative were MonoDevelop [8], which is a open-source cross platform .NET IDE, but since none had experience with this tool either it were not preferred. As most group members work at a daily basis on Windows Based pc's Visual Studio is the favorite choice of IDE.

### 4.1.2 Collaboration

For collaboration we used Subversion(SVN) and for the code sharing we used AnkhSVN [1] together with SVN. AnkhSVN is a source control provider for Microsoft Visual Studio. Alternatively we could have used Team Foundation Server [2], but this requires installation and configuration of a Team Foundation Server. We chose AnkhSVN since we already had a running SVN server.

### 4.1.3 Database

As our main data storage we use a Microsoft SQL Server. We choose this data storage vendor since it is compatible with ADO.NET Entity Data Model Designer and Visual Studio Ultimate comes with a inbuilt Microsoft SQL Server manager. Which allows for editing the SQL server from our workstations and not only from the server itself. We considering using postgreSQL, but using postgreSQL with C# and Visual Studio required a plugin in order to use the ADO.NET Entity Data Model Designer. As a database vendor neither choice

would not give us any advantages on the data layer, since the required functionality is supported by both systems.

## 4.2 Development Method

---

*Tail*

# 5 Program Presentation

*This chapter outlines and present the process of using our system from the three points of perspective, based on the three user roles. This is done to show that our systems usage is consistent with the use cases we presented in the Application Domain Analysis, chapter 3.*

---

As described there are three roles a user of our system can have:

- Client
- Staff
- Admin

As with all users of the system, the first thing which the user is greeted with, is the welcome page, followed by the login screen. After that, the path is split up according to the role the user has. All the pages of our system shares the same master file, particularly the menu which holds the functionality of the logged in user. The shared master gives a top of every page, which is seen on figure 5.1. The points in the menu changes depending on which privileges the current user has.

Below the most common usages of our system is described. These are based on the use cases from our analysis in section 3.1.2. We will start by presenting the clients usage.


Figure 5.1: *The menu which the master file is responsible for making. The Commit Problem, My Problems, and Search for Problems are menu points which are allow for clients. The My Worklist menu point is a menu point for the staff users. The Mange Department, Mange People, and Statistics menu points are allowed for admins only*

## 5.1 Client Usage

The client can generally do three things; commit a problem, see status of his/her problems, or search the database for problems. The first two are based on there corresponding use cases in section 3.1.2. The search function was added as a functionality for client because we assume that it would primarily be persons with a new problem who might want to search for others problems.

The following subsections describes the three usage of the system which the client has access to.

### 5.1.1 Commit a Problem


Figure 5.2: *The categorization step in committing a problem*

The process of committing a problem starts with a click on the Commit Problem menu point, which can be seen in figure 5.1. When Commit Problem is clicked the user is asked to categorize the problem in the window showed in figure 5.2. As the figure shows the tags, the check boxes, are ordered under headlines. These headlines are categories.

When the problem is categorized the system searches for problems matching the specified tags, the search function is described in 7.2. The problems found in the search are listed in a table with columns showing number of matching tags, deadline, estimated time of completion, and title and description. The last column also shows whether the given problem is solved or not.

The client can choose a problem(or even more problems) to subscribe to in order to receive notifications when the problem is updated, e.g. a solution is attached to it. The client can also choose to write a new problem if none of the existing problems suffice to match his/her problem. The screen showed when creating a new problem is seen in figure 5.3

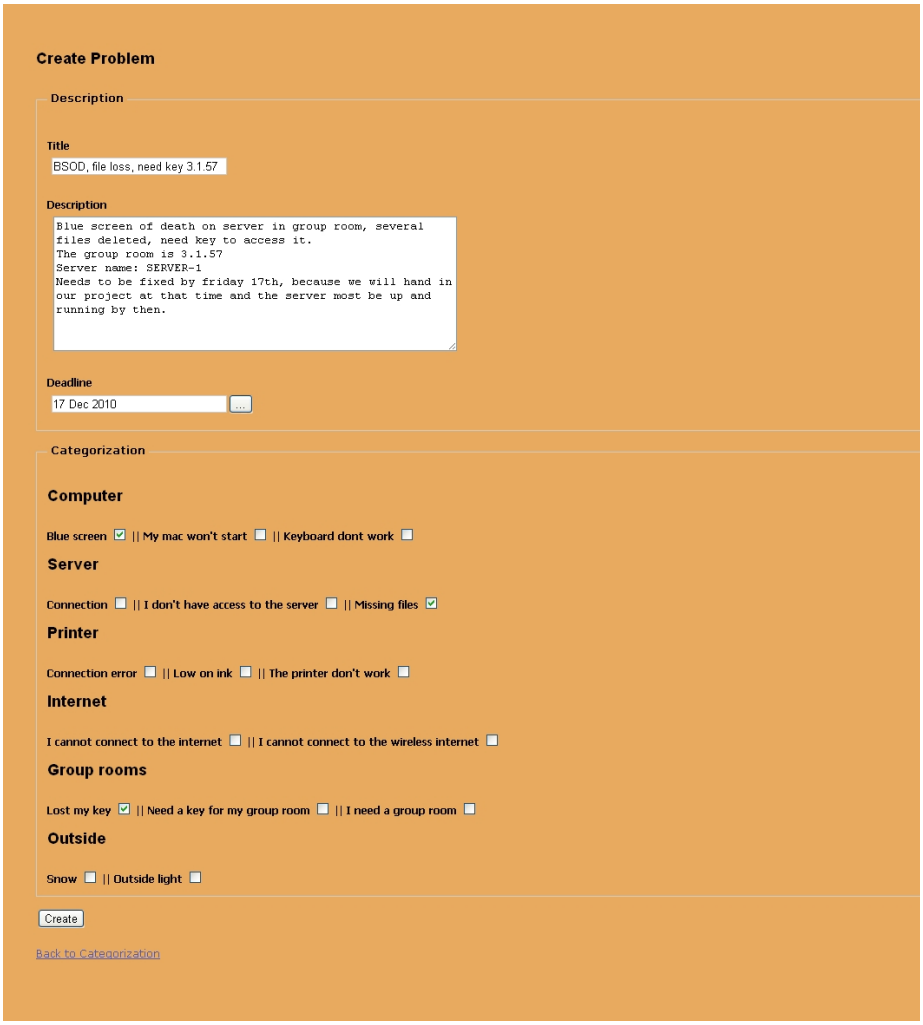


Figure 5.3: *The create step of committing a new problem*

### 5.1.2 See Own Problems

### 5.1.3 Search for Problems

## 5.2 Staff Usage

## 5.3 Admin Usage

---

*This chapter outlines and present the process of using our system from the three points of perspective, based on the three user roles.*

# 6 Enviroment

*hvad og hvorfor*

## 6.1 MVC Framework

The Model-View-Controller (MVC) was originally designed by Trygve M. H. Reenskaug in 1979, who was at that time working for Xerox PARC. The goal was to create an envoirontment in which the users (developer, customer ect.) could preserve the original perception of the datastructure, while being able to view and edit portions of it.

### 6.1.1 Components

The (MVC) framework consists of three different types of components; Views, controllers and models. Each created with a different purpose, see figure 6.1

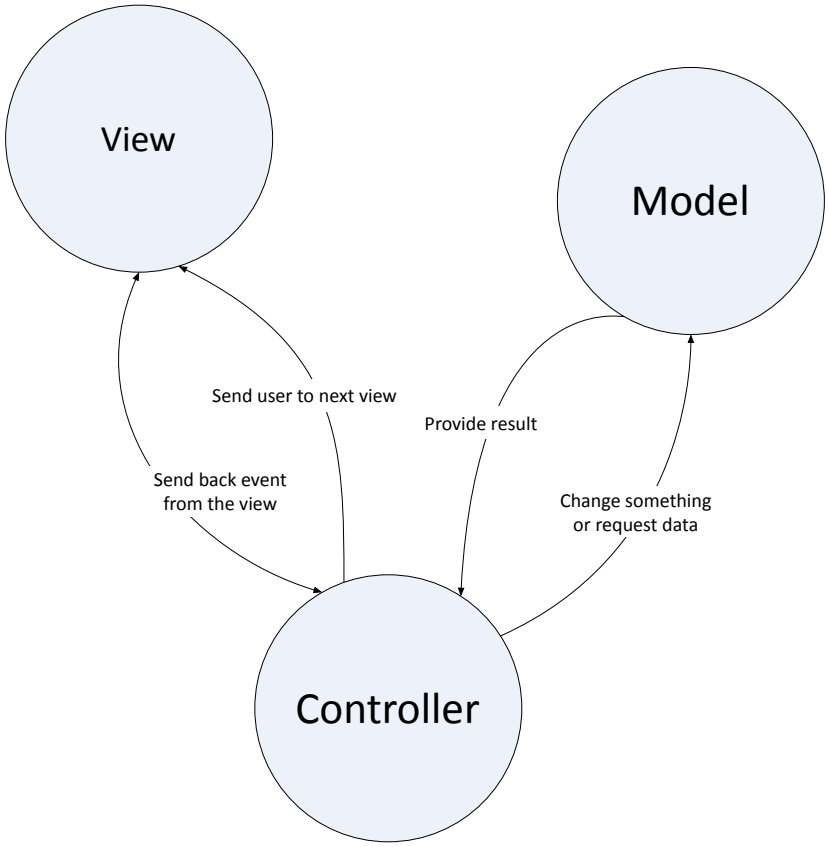


Figure 6.1: *These are the three main components in the MVC*

- Models

The models are the components that handle the data domain of the application, , from which it reads and writes data. This way developers can handle the data, without worrying about the actual data connection.

- Views

The views display the data and UI. They are typically created with data provided from the model. These views are full-featured (X)HTML, with the addtion of C# or Basic code, to fetch content from the model.

- Controllers

The Controllers reacts to user requests the users provides through the UI. Based on this they deside which view should be rendered. If the data the view should display does not exist in the model (input, ect.) the controller can also provide this data to the view to display.

### Master-page

Master-pages is not a part of MVC as a design-idea, it is a component implemented by Microsoft. The master-page is a page that is always loaded when

displaying a view. Like the view, it consists of (X)HTML and C#/Basic, however, it also has content-containers. These containers can be created anywhere on the webpage, and are filled with content by views.

### 6.1.2 Structure

An interesting thing about MVC is that there are no actual HTML files, all content is generated generically as the users interacts with the diferent components within the webpage. When entering an URL you actually call mothods in controllers, you can even add parameters to the calls. The controllers then redirects you to the correct view based on the input and the method called. The view then executes, which results in a plain webpages that can be displayed in a browser.

Because of the seperation into three independent modules, developers are able to create different parts of the application with different kind of approches, thus enabling them to better manage complexity, create applications with a high maintainability, and create UI that does not have to show the actual datastructure, which can often be confusing for an ordinary user of the webpage.

#### Data validation

Client side validation Server Side validation

#### Testing

When creating web applications with MVC you have the possibility of creating unit tests to thoroughly test your code. These tests can be automatically generated from within Visual Studio, and run as a seperate function, in order to allow the developer to test for all possible inputs. For a more detailed description of Unit-testing in MVC, look in the section **??**

[**?** ] [9] [**?** ]

## 6.2 Database Structure

We distinguish two parts of the database, the login part, and the model part. The model part is created from our Model [1] using the ADO.NET Entity Framework(EF) which is described in subsection 6.2.1. We choose to use ADO.NET EF because we do not have to worry about converting our tuples to objects and to make sure that changed properties are mapped to the database correctly. To administrate users we use the inbuilt ASP.NET membership provider which provides a login system with support for role authorization. We use this provider since it saves time instead of building our own login system. Security is not a big concern in this project and therefore we do not want to spend time creating a secure login system. The major disadvantage is that including the membership providers database scheme with the model is not supported and will not work proper. Therefore we had to add a person entity and change the register functionality to also save the registered person in our person table. This gives some redundant data, but only the username and email. When we needed to

[1] FiXme Dødelige: Link til MODELLEN sættes ind her.

access data from the membership tables we had to use SQL statements and not the object oriented approached we used for the rest. This is limited since our main functionality depends on the model.

### 6.2.1 ADO.NET Entity Framework

The ADO.NET is designed by Microsoft with the purpose of making a disconnected database architecture to use with the .NET framework [3]. This architecture is an improvement from the older database architectures where a connection were made and held open in the entire runtime. With a disconnected database the connection is only open when data is needed [**?** ].

The ADO.NET EF is an object relational mapping framework [5] build upon ADO.NET. Together with the ADO.NET Entity Data Model Designer and a SQL Server ADO.NET EF gives a strong tool for mapping a database and using the data in a object oriented matter, without dealing with the transformation from entity to class and from tuple to object. The Designer is built into Visual Studio Ultimate.

Using the ADO.NET and the Entity Data Model Designer can be done in two ways. The model can be created from an already existing database or by creating the model and generate a database from this [4]. We use the last approach and creates the database from the model. In this way we do not have to worry about setting the right foreign keys and relational tables. Instead we get a fully functional model linked to a database.

Our model as it is in the ADO.NET Entity Model Designer is shown on figure **??**

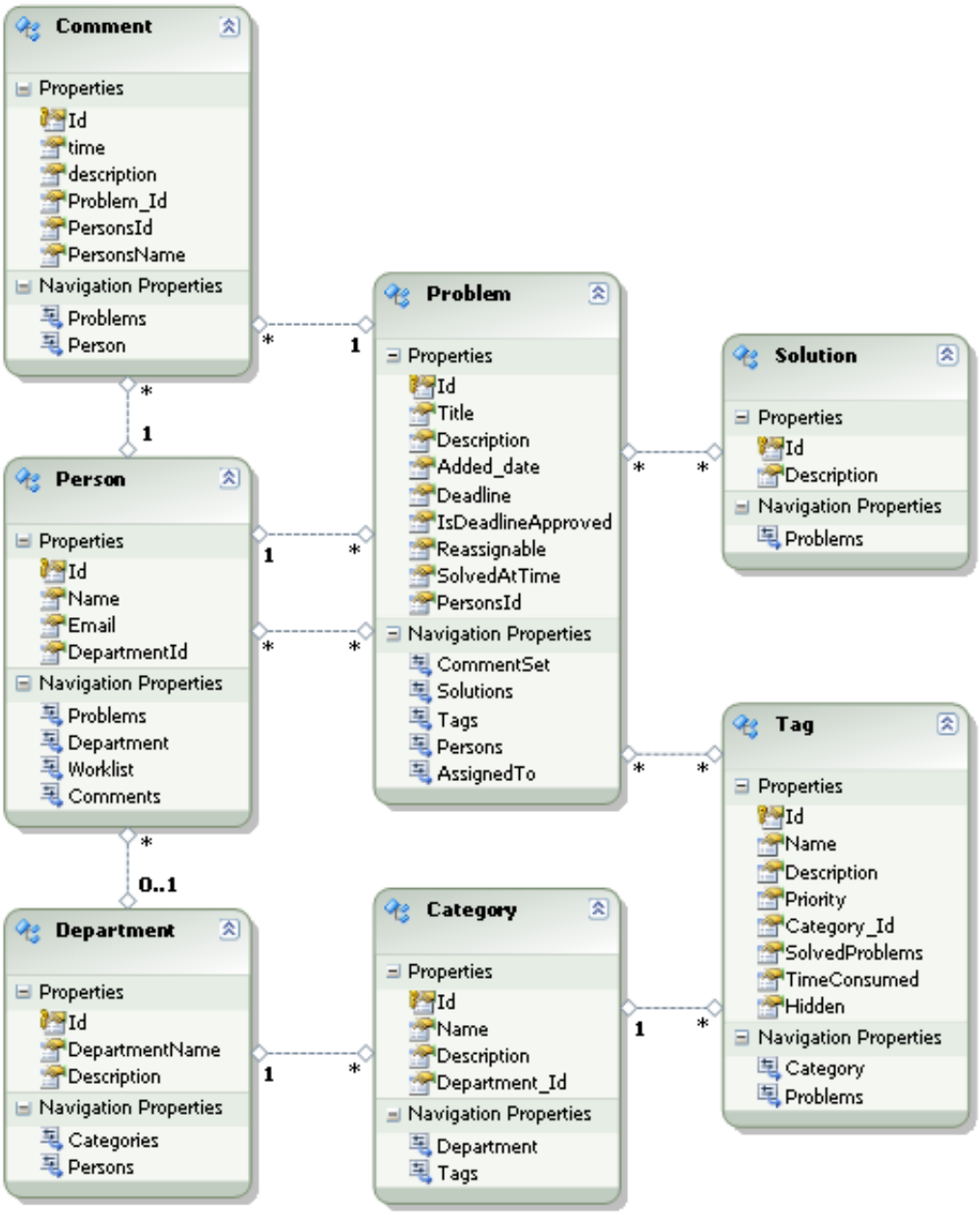


Figure 6.2: *Our model as it is seen in the ADO.NET Entity Data Model Designer.*

---

*Hvad*

# 7 Key Points

*This chapter describes the key points in our program. The key points which are included are problem searching and workload balancing. These are chosen because they both are central functions to the entire system. We want to search for problems to avoid clients committing similar problems and to search the database for a specific problem according to the system definition in section 1.3. We have also chosen that our system should balance the workload between staff members in order to provide more efficient solving as we state in our system definition.*

---

## 7.1 Problem Prioritization

When assigning multiple problems to a staff member, they will appear as a list. Due to the human nature, we might pick a specific order of solving the problems in, which does not necessarily take priority, deadlines and such into account. Therefore, the order of which the problems appear in in the staff members work list, is important.

We have defined two elements which define a problems importance, and therefore its placement in the list of problems. These two elements are:

- Whether or not a deadline is approved
- The priority

We acknowledge that problems which has a deadline should be at least reviewed by the assigned staff member earlier than problems without a deadline.

The list is ordered by priority of the problem, however problems with approved deadlines will always appear on top of the list regardless of their or other problems priority. This splits the list into two parts. Above, priority-sorted problems with approved deadlines, and below priority-sorted problems with or without not approved deadlines.

If a problem with an approved deadline is overdue, then the priority will go up to the maximum value 10, which in turn will make the problem appear on the top of the list.

An example can be seen in figure [1]

[1] FiXme Dødelige: Indsæt figur fra "Program presentation" section der viser det.

## 7.2 Problem Search

The system needs to be able to search for problems. This search is based on tags. It should find an amount of problems which match the specified tags and order them by number of tags matching. The amount of problems this function will find is depending on a specified number know as "Minimum number of problems", which determines when the function should stop searching for more problems.

The input for this function is:

- Selected tags
- Problems to search among
- All tags
- Minimum number of problems to find

The *Search* function is called with the parameters above. It calls an *InternalSearch* function which is private, with the same parameters and a compare delegate which determines how the problems should be sorted. The *InternalSearch* function is described in subsection 7.2.1 and 7.2.2. Subsection 7.2.3 describes the *SearchSolvedFirst* function, which takes into account whether or not a problem is solved, when ordering the list of problems to return. It does so by calling the *InternalSearch* function with another compare delegate.

### 7.2.1 Search for Problems by Tags

The most important part of the *InternalSearch* function is the while loop shown in code snippet 7.1. Generally, this loop finds problems which matches the tags specified in the input to the function and orders them with the problems with the most amount of matching tags in the beginning of the result list. The while loop beginning in line 4 will continue to run as long as there has not been found enough problems to suffice the minimum number of problems input and there still is at least one tag to search for. If there still is not enough problems another part of the search function will take care of this. This part is described in subsection 7.2.2.

The function will increase the number of tags to remove from the tag list which was input to the function every time an iteration ends in the outer while loop; lines 4-32. In the first iteration no tags are removed, this means that the function will find every problem which has every tag which is being search for and put these in the beginning of the result list. Further more the function sort the problems each step by the least amount of tags. The reason for this is that the less tags a problem has which are not searched for, the more likely it is that the given problem matches the search. For example if a search is run for the tags "Computer" and "Harddisk", the problems only containing these tags will be listed first and if a problem contains the tags "Computer", "Harddisk", "Database", and "Connection" it will be listed further down on the result list because it has unrelated tags attached to it. See a more detailed example of a run of the search function in appendix **??**[2].

[2] FiXme Dødelige: Tilføj dette eksempel eller fjern denne linje

```
1  .
2  .
3  .
4  while (result.Count < listMinSize && noOfTagsToRemove < tags.Count)
5  {
6      tempResult = new List<Problem>();
7      tagsToRemove = new List<int>();
8      for (int i = 0; i < noOfTagsToRemove; i++)
9      {
10         tagsToRemove.Add(i);
11     }
12     try
13     {
14         List<Tag> currentSearch = tags.RemoveCurrent(tagsToRemove);
15         while (true)
16         {
17             temp = allProblems.ToList();
18             foreach (Tag tag in currentSearch)
19             {
20                 temp = temp.Where(x => x.Tags.Contains(allTags.
                       FirstOrDefault(y => y.Id == tag.Id))).ToList();
21             }
22             tempResult.AddRangeNoDuplicates(temp.ToList());
23             currentSearch = tags.RemoveNext(ref tagsToRemove);
24         }
25     }
26     catch (NotSupportedException)
27     {
28         noOfTagsToRemove++;
29         tempResult.Sort(compare);
30         result.AddRangeNoDuplicates(tempResult.ToList());
31     }
32 }
33 .
34 .
35 .
```

Code snippet 7.1: *The while loop which finds and sorts problems matching the input tags*

The inner while loop spanning the lines 15-24 iterates over the tags to remove, this does not have any effect when no tags are to be removed. However if the function gets the tags "Computer" and "Harddisk" as input, we first want to find every problem with both tags, then find every problem with the "Computer" tag, and finally find every tag with the "Harddisk" tag. The order of the last two is not important because they are sorted in a single list, which is then inserted into the result list. The *RemoveNext* function called in line 23 is responsible removing the tags which are not to be search for in the in the next iteration of the inner while loop in lines 15-24. It will throw a **NotSupportedException** when it has removed every combination of tags, which will break the inner while loop and add the problems found in the current search to the result list, which will be returned to the call site later. The function *AddRangeNoDuplicates*, is used instead of the in-build *AddRange* function, because otherwise one problem could be added several times, which is not wanted. One problem should only appear one time in the list returned from this function, because it would simply not make sense in relation to the minimum number of problems, since a single problem would be counted several times towards find-

ing enough problems. Further more the client seaching for problems should not have the same problem appear on his/her list more than once. Therefore at some point in our code we would have to filter out the duplicates, we chose to do it here, because it is the earliest step in finding problems in our database. If we were to filter the duplicates out another place, the search function could potentially return a list containing a single problem several times, which then – when filtered – only yields a single problem and thereby rendering the minimum number of problems nearly useless.

The for-each loop in the lines 18-21 finds all the problems match the current search. The current search is the tags being input to the function without the tags to be removed. The for-each loop removes every problem not containing a specific tag in each iteration, until every tag in the current search is covered.

The initialization of the tags to remove is done in the for loop in the lines 8-11. It sets the first $x$ tags to be removed where $x$ is the current number tags to remove. This means that if e.g. three tags should be removed it will initially be the first, second and third tag, which are removed.

### 7.2.2 No Tags to Remove

If there has not been found enough problems to suffice the minimum number of problems during the search in tags the function will start to look for problems with no tags at all, then problems with one tag etc. This part of the function will start by finding every problem with no tags and add them to the result list, then every problem with a single tag is found and added. Here the problems are also added using the *AddRangeNoDuplicates* applying the same reasoning as above. This part of the function will continue to run until enough problems are found or it is about to search for more tags then there is in the "All tags" input. This means that it can actually return less problems than minimum number of problems if it cannot find any more, but at this point it has searched every problem, this means that it will actually return every problem in the "Problems to search among" sorted.

### 7.2.3 Order by Solved

In some cases we want to order the problems by whether or not the problems are solved. E.g. we want to show the solved problems first to clients who are categorizing a problem, which might already exist. The reason for this is that the client should be presented with problems with a solution first, in hope that the client can use a solution and does not need to subscribe to a problem or add a new one.

This search function makes use of the *InternalSearch* function but with a different compare input to the *Sort* function. This compare function sorts first by whether or not a problem is solved, then by the least number of tags.

## 7.3 Balance Workload

Whenever a staff member is removed from a department or has marked a problem as solved, it becomes necessary to balance the workload of each staff member

in the department, since we do not want the staff members to be overloaded with problems.

To balance the workload, each staff members workload must be calculated. The workload of a staff member is defined by the amount of time estimated that each problem on his workload takes to be solved. The workload is calculated by the *GetWorkload* method.

The time a problem takes to be solved is estimated by the average time consumption of the tags connected to the problem. This is calculated by the *CalculateTimeConsumption* method.

The *BalanceWorkload* method works by finding the staff member in the department with the minimum workload and the staff member with the maximum workload. Then it moves the highest priority problems from the maximum staff member to the minimum. It keeps reassigning problems until the minimum staff member has a higher or equal workload than the maximum staff member. If the minimum staff member has a higher workload, then the algorithm checks if the balance can be balanced even further by calculating if the last moved problem should be moved back or not, and moves it accordingly. See lines 25 to 42 in code snippet 7.2.

All this is iterated once per staff member minus one in the department. E.g. if there are two staff members it is ran once, if there is three it yields two etc. The pseudo code is displayed in code snippet 7.2.

The primary concern of the algorithm is to distribute the problems so each staff member has as balanced workload as possible.

We also wanted the algorithm to take the individual priority of the problems into account. This algorithm makes sure that the high priority problems will be distributed. If the priority was unrelevant the algorithm would take the problem with lowest time consume, since this would give the most balanced workload. An example of the algorithm is shown on figure 7.1.

---

*The key points presented in this chapter are problem searching and workload balancing. The problem search describes how our system is able to search and sort problems. The balance workload method is used to distribute problems among staff members.*

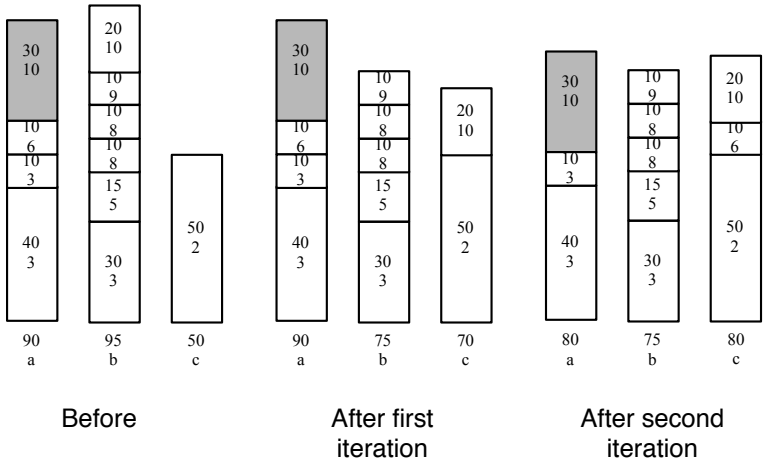


Figure 7.1: *A diagram of the balance workload method. Each collumn represents a staff members workload. Each box is a problem. The upper number is the estimated time consumed of that problem. The lower is the priority. The problem colored dark grey is not reassignable. There are tree staff members a, b, and c. The problems that will be moved is colored light grey.*

```
bool couldStillMove = true;
do
{
    // Finde the reassignable problem with the highest priority
        which has not been moved yet.
    var problemToBeMoved = max.Worklist.FirstOrDefault(y =>
        y.Reassignable == true &&
        y.HasBeen == false &&
        y.SolvedAtTime == null);

    // If none can be moved leave the while loop
    if (problemToBeMoved == null)
    {
        couldStillMove = false;
    }
    else
    {
        // Mark as has been moved
        problemToBeMoved.HasBeen = true;

        // Reassign the highest priority problem to staff member
            called min.
        problemToBeMoved.AssignedTo = min;

        if (min.Workload >= max.Workload)
        {
            // Initialize variables for checking whether or not to
                move the last problem back
            double beforeMoveBack = 0.0;
            double afterMoveBack = 0.0;

            // Calculate difference before moving
            beforeMoveBack = Math.Abs(max.Workload - min.Workload);

            // Move it back
            problemToBeMoved.AssignedTo = max;

            // Calculate difference after moving
            afterMoveBack = Math.Abs(max.Workload - min.Workload);

            // Compare
            if (beforeMoveBack < afterMoveBack)
            {
                problemToBeMoved.AssignedTo = min;
            }
            couldStillMove = false;
        }
        else if (min.Workload == max.Workload)
        {
            // Don't move back if they are equal
            couldStillMove = false;
        }
    }
} while (couldStillMove);
```

Code snippet 7.2: *A code snippet of the balance workload method. The presented code is within a for loop running for each staff member minus one. "min" and "max" are the person objects of which the algorithm are currently moving problems between.*

# Part III

# Appendix

# Bibliography


[1] Collabnet. Ankhsvn. WWW, 2010. URL http://ankhsvn.open.collab.net/. Last viewed: 6/12.

[2] Microsoft Corporation. Team foundation. WWW, 2005. URL http://msdn.microsoft.com/en-us/library/ms181232(VS.80).aspx. Last viewed: 6/12.

[3] Microsoft Corporation. Design goals for ado.net. WWW, 2010. URL http://msdn.microsoft.com/en-us/library/7b13c12s(v=VS.71).aspx. Last viewed: 6/12.

[4] Microsoft Corporation. Ado.net entity data model designer. WWW, 2010. URL http://msdn.microsoft.com/en-us/library/cc716685.aspx. Last viewed: 6/12.

[5] Microsoft Corporation. The ado.net entity framework overview. WWW, 2010. URL http://msdn.microsoft.com/en-us/library/aa697427(VS.80).aspx. Last viewed: 6/12.

[6] Microsoft Corporation. Visual studio ultimate. WWW, 2010. URL http://www.microsoft.com/visualstudio/en-us/products/2010-editions/ultimate. Last viewed: 6/12.

[7] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Oriented Analysis & Design*. The Johns Hopkins University Press, 1 edition, 2000. ISBN: 87-7751-150-6.

[8] Mono-project. Mono, 2010. URL http://www.mono-project.com/Main_Page. Last viewed: 7/12.

[9] Trygve M. H. Reenskaug. Mvc xerox parc 1978-79. www. URL http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. Last viewed: December 2010.