HOPLA
HELP! DESK

I

AALBORG UNIVERSITY

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 Stubman Reichenauer

Kristian Kolding Foged-Ladefoged

Lasse Rørbæk Nielsen

Rasmus Veiergang Prentow

Advisor:
Nadeem Iftikhar

Page count: 112

Appendices count: 1

Finished: 17/12–2010

Synopsis:

Larger organizations often have one or more maintenance departments. These departments have to respond to requests from people who encounter problems. The request can come in different forms; for example email, phone calls, or personal contact. If the problem is big and takes a lot of time to solve the person who requested help might like to follow the process, he would then have to make a phone called or email the service employee. If a problem occurs often and has a trivial solution, help requests are made every time a person encounters the problem, thus the service employee has to explain the same solution several times, wasting precious time – time which could have been spent solving other problems. A supervisor of a maintenance department might also like to keep track of the time employees spend solving problems on average, so that he/she can rearrangements inside the department to get a better average solving time.

In this report we try to address these problems by developing Hopla Helpdesk. In the report we have described our development process and the resulting application.

Preface

This report is written to document the 3rd semester project by group s305a – Software Engineering students from Department of Computer Science at Aalborg University. The project is commenced at September 2, 2010 and finished December 17, 2010. The overall theme of the project is programming. We have chosen to utilize the knowledge from the PE courses(project related courses) System Analysis and Design (SAD) and Object Oriented Programming (OOP), by designing our application using the Object Oriented Analysis & Design method – from here on called OOAD – and programming Hopla Helpdesk in the object oriented paradigm.

In order to get a full understanding of this report it is necessary to understand the OOAD method before hand. Traditionally OOAD has a terms involving the "system", e.g. system component. We are however making an application and are therefore exchanging these terms to an equivalent with the word "application" instead of "system", in order to keep consistency when talking about Hopla Helpdesk.

The report is split into five parts, Analysis, Design, Implementation, Testing, and Epilogue. Each part consist of several chapters. In the top and bottom of every major chapter there is a small piece of text written in italic. This marks the head and tail of each chapter. The head outlines the content and the reason for the chapter to be in the report, whilst the tail summarizes the chapter. In the end of the report there is an appendix which is recognized as the sixth part of the report.

Citations are written in square brackets i.e. [xx]. The number within the bracket is a reference to the bibliography which can be seen on page 101. References to chapters or sections inside the report or in the appendix of this report are referred to either by the number of the given chapter or section, or by the page number where the given chapter or section is found.

The CD contains the complete source code of Hopla Helpdesk, a complete database dump with sample data, the test project, and the report as a PDF file.

When references to the code are made we use special notation for this which can be seen as follows: **properties**, **variables**, **classes**, and *methods*. In the report there is inserted code snippets. In these code snippets, "..." means that some code has been omitted.

We would like to give our thanks to Kristian Torp for help during setup of the database for our application and to Rene Hansen for helping us choose a framework to build our application upon. Finally we would like to thank our supervisor Nadeem Iftikhar from Department of Computer Science at Aalborg University for continuous feedback and advice during creation of this report and our application.

Contents

List of Figures

1 Introduction

We live in a world where we usually distribute the problems at hand to the people who are best at solving them. Sometimes this is an easy process, other times it can be an immense waste of time trying to find the correct person to hand the problem to. Often, we end up solving problems ourselves because it is simply too difficult to find the right person for the job. Solving problems ourselves is often inefficient compared to how well an expert in the specific field solves the same problem. This approach will effectively waste manpower in a company, as it is not the most effective way to solve problems. Organizing problems can also be difficult for the one who is dedicated to solving them. If no systematic approach to organize the problems exists, time might be wasted organizing problems or solving problems which are not important at all.

The people who are experiencing problems and the people who are solving them presumably have one common goal, which is to solve all problems as fast as possible. This is why large organizations are likely to have dedicated people to solve problems that may arise for the regular employees of the organization.

What we want to do with this project is to make it easier for the problem solvers and the regular employees to reach this goal. To reach this goal the regular employees who wants to get a problem solved could categorize the problem, thereby easing the distribution of the problem. This would probably not be more time consuming because the employee would have to describe the problem anyway in order to get it solved by someone else.

Saving solved problems could result in a faster solveing times for problems that reoccur later. If the regular employees have access to the solved problems they might even get the solution to their problem without having to bother any of the problem solvers. The regular employees could be allowed to see problems which are currently being solved. This would allow the problem solvers more peace to work as they will not be disturbed unnecessarily by request which they are already aware of. The productivity would be increased, and the amount of wasted time would be decreased.

Part I

Analysis

2 Task

This chapter presents the purpose of our project, our application definition which defines the requirements to the project, and a context where the application is seen from a perspective of the end user. The application definition in particular is very essential to the project because it defines our application.

2.1 Purpose

We develop this application with the purpose of easing the process of solving problems, as well as distributing knowledge of how to solve reoccurring problems. Thereby relieving the maintenance staff of these trivial tasks, effectively increasing the productivity of all employees in an organization, as they spend less time occupying themselves with trivial problems, and more time actually solving important ones.

2.2 Application Definition

In order to define our application we use the FACTOR criterion. [17, p. 39] These criteria contains different parts of the requirements, and it will help us derive the application definition. The FACTOR criterion for our application is as follows:

Functionality
Our application contain features aimed at easing the problem-solving process for service employees, therefore the application needs to have the following features:

- Problem distribution
  The application distributes problems to the relevant service department and balance the workload of service employees so all have a equal workload.
- Problem categorization
  The problems can be categorized and thereby making it easy to find relevant problems.

- Ranking of problems
  Unsolved Problems are displayed on a ranked list for service employees, depending on their importance.
- Keep track
  The application is 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
  - When the deadline is, and if it is exceeded
  - Which tags and categories are related to the problem
  - Communication between the service employee and the person who committed the problem
- Knowledge saved
  The application is able to recognize common problems and recommend similar problems to the user, and by that saving the service employee the work of solving common problems several times.
- Statistics
  Allow the supervisor to monitor the work process of the employees.

Application domain
This application is applicable to office environments which deals with solving of problems. The persons who will be using our application to commit problems are the "regular" personnel or employees while the service employees or service personnel are the ones who will solved the problems.

Conditions
The persons who commit the problems are not required to have any expert level knowledge to use the application. The service personnel have to learn to use the application to solve problems. The administrators have to learn to use the functionality in the administrator interface of the application.

Technology
In the development of our program we use C# along with the Model-View-Controller framework 2 that exists within ASP.NET framework. We use AnkhSVN and Visual Studio 2010 to collaborate on development. To store data we will use a DBMS(database management system). The choice of DBMS is made later, namely in subsection **??**.

The result is a web interface running on a web server, with an underlying SQL database.

Objects

- Problems
- Solutions
- Departments

- Categories
- Tags
- Clients are the persons who commit the problems
- Staff members are the service personnel
- Admins are the administrator and bosses

Responsibility
Our application is responsible for keeping track of all problems within the physical environment of an organization. It is also responsible for distributing tasks amongst employees and supplying statistics about the progress to their supervisors. Finally it is also responsible for enabling both regular personnel and service personnel – staff members, to communicate with each other about a problem and the following solution.

Using the FACTOR criterion the following application definition is derived:

- The application is be web-based so that we can create an application capable of running without prior installation.
- The application contains prioritized tags, enabling us to determine the importance of problems
- The application keeps track of the estimated completion time of the problem along with comments and which employee is responsible for solving the problem. This enables us to create statistics about problem solving efficiency and individual employees.
- The application is able to save and suggest prior problems and their solutions to users.
- The application takes into account, the workload of the employees to estimate how long a problem will take to solve.
- The application contains nothing but dynamic data within the problem domain, so that the application can run in different environments.
- The application contains a structure that can handle problems, departments, categories, tags, regular personnel, supervisors and staff members.

2.3 Context

Our application targets a work environment or an educational environment. The analysis and design process is based upon a university, but the application is easily implemented at any other work environment.

To fully understand the context we draw a rich picture which can be seen on figure 2.1. The central aspects of the rich picture is that both the problem submitter and the problem solver can act on the problem, and will communicate through the application. From the rich picture we see a few conflicts in the environment:

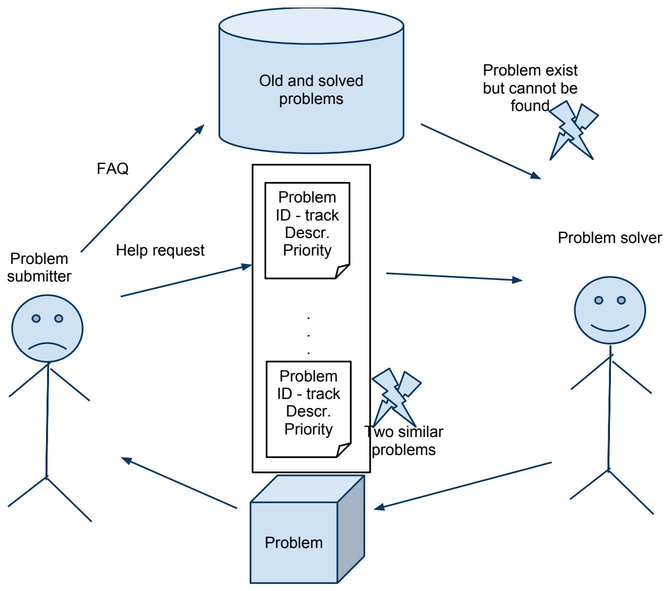

Old and solved problems
Problem exist but cannot be found
FAQ
Problem ID - track Descr. Priority
Help request
Problem submitter
Problem solver
Problem ID - track Descr. Priority
Two similar problems
Problem

Figure 2.1: *Rich picture of our application*

- 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 application are the problems and solutions.

2.3.1 Problem Domain

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

Another important phenomena in our application is when a problem is solved. This is initiated by a staff member and will result in a notification to the clients who are subscribing to alterations of the problem, for example solutions or comments being added.

Chapter 3 gives a detailed analysis of the problem domain and will elaborate on the phenomena.

2.3.2 Application Domain

The people who will be acting on the application are 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 member. But the submitter could be a staff member as well. Beside the two main actors there is an admin who can administrate the staff, clients, and the application itself.

A detailed analysis of the application domain is presented in chapter 4, where the actors and their use cases are further described.

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

3 Problem Domain

In order to determine the nature of our solution the context of the application is analyzed. The classes and events described in this chapter are reflections of how the problem domain is modeled. This chapter also includes a description of the behavior of each class, in order to understand the different classes better.

3.1 Classes and Events

In the process of finding classes and events several classes and events are found, some of which are not used in the actual program, since it might be more efficient to exclude some classes. This is elaborated further in the Design part. To come up with the classes and events several different possible scenarios where discussed and from those scenarios we base our classes. This section describes all the classes and events in the final iteration of the process. On figure 3.1 an early version of the relations between classes and events are illustrated.

3.1.1 Classes

The following is a list of the classes we found.

Problem Problem is the basic building block in our application. It is used to contain 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

Client client submits problems and subscribe to problems in order to be associated with it, and thereby receive updates about this problem.

Events	***Classes*** 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 3.1: *Problem-domain analysis event table*

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.

3.2 Structure

The class diagram is based on the classes and events, which is described earlier in section 3.1. 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 **staff**. Alternatively the role pattern could have been used. This prescribes that **client**, **staff**, and **admin** all would have been a subclass of ***role*** [17, p. 80]. We consider that it makes more sense that they inherited from each other, because they have a lot of shared properties and privileges.

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.

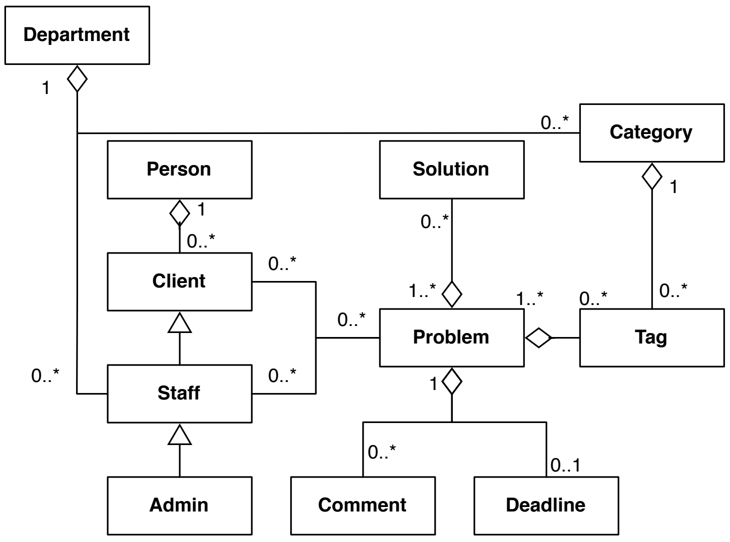


Figure 3.2: *Class diagram*

- **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 3.2

3.3 Behavioral Pattern

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

3.3.1 Problem

Figure 3.3 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.

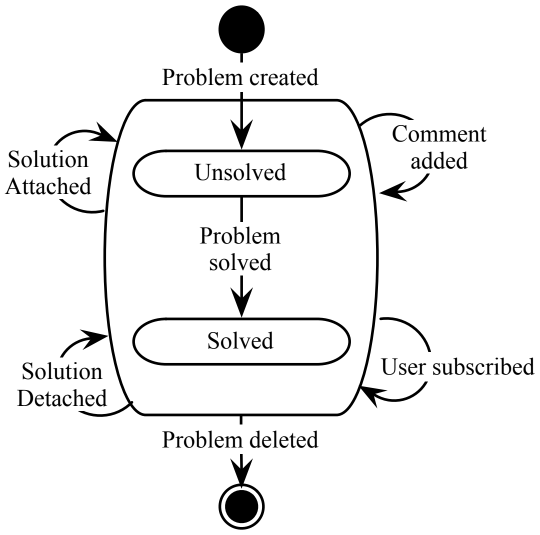


Figure 3.3: *The statechart of the problem class*

3.3.2 Person

As shown in figure 3.4, a person is assigned a role in the application 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 not do the same as admin.

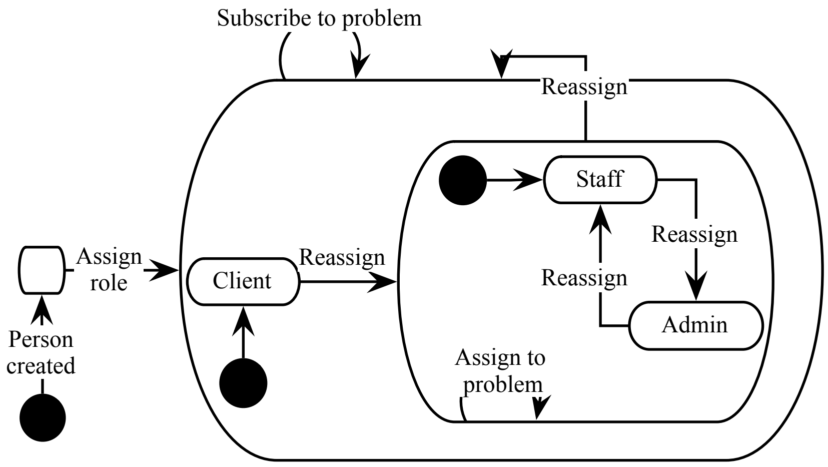


Figure 3.4: *The class person's statechart*

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.

4 Application Domain

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

4.1 Usage

Three actors are identified and the six most relevant use cases are selected. The three actors which are identified are: Client, staff, and admin. The selected use cases 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 4.1: *Actor & use case table*

Figure 4.1 shows the relationship between use cases and the actors of our application. 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 application and access the statistics

The use cases in figure 4.1 are described in subsection 4.1.2.

4.1.1 Actors

The application has two primary actors; client and staff. Client is the lowest privileges actor, his/hers 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 have a problem which should be addressed to another department. E.g. the tortoise in the IT-administrators office needs to be fed and the maintenance officer should feed it.

Client

Goal: A person who has a problem, and his goal is to get his problem(s) solved.
Characteristics: The clients are employees or students with different knowledge and experience with similar applications.
Examples: Client A prefers face-to-face communication with the working staff whenever he has got a problem.
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 4.2: *Description of the actor client*

Staff

Goal: The goal of the staff members are to solve the problems which the clients commit to the application.
Characteristics: The staff are employees and has various levels of technical knowledge.
Examples: Staff A prefers to speak to his manager and the client face-to-face.
Staff B enjoys getting his daily tasks from a computer application,

Figure 4.3: *Description of the actor staff*

Admin

Goal: The admins' goal are to make the application run smoothly by managing the tags and the persons in the application.
Characteristics: Responsible for the Hopla Helpdesk or staff manager.
Examples:
Admin A is a software engineer and are responsible for maintaining the application by management departments, category, and tags.
Admin B is the boss of the department and evaluate staff members based on the statistics generated by the helpdesk.

Figure 4.4: *Description of the actor admin*

These two actors are described in details in figure 4.2 and 4.3. Beside staff and client the application 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 staff member, the actor admin is described in details in figure 4.4.

4.1.2 Use Cases

The use cases from figure 4.1 are described in this subsection.

Submit problem

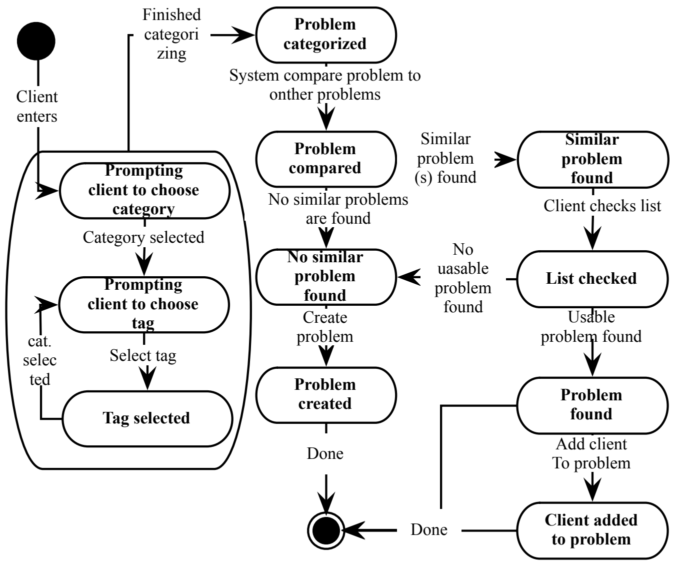


Figure 4.5: *A state chart diagram of the use case submit problem*

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 4.5.

- **Use Case:** Submit problem is initialized when a client has a problem and wishes to submit that problem to the application 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 application compares the selected tags with other problems. If similar problems are 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 member of the staff.
- **Objects:** Problem, tag, comment, category, deadline, client, and staff.

- **Functions:** Get problem tags, Search for problems, Create problem, Subscribe / unsubscribe to problem, and Get Est. Time Consumption of Problem.

My problems

The use case my problems is used by clients, staff members, 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 member uses to solve problems.

Solve problem

The use case solve problem is the staffs primary usage of the application. When the staff get his worklist he/she can select a problem and solve the problem. A statechart diagram is shown in figure 4.6.

- **Use Case:** The use case is initialized when the staff member wants to check his/her worklist. The staff member is then presented with a list of unsolved problems assigned to him/her. The staff member can then click on one of the problems to read 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, tag, comment, category, deadline, client, staff, department, and solution
- **Functions:** Get problem tags, Search for problems, Create problem, Subscribe / unsubscribe to problem, Get est. time consumption of problem, Manage tag times, Get expected time of completion of problem, Get tag average time consumption, Get sorted worklist and Approve deadline.

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 4.7.

- **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 application. 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:** Problem, tag, comment, category, deadline, client, staff, department, and solution

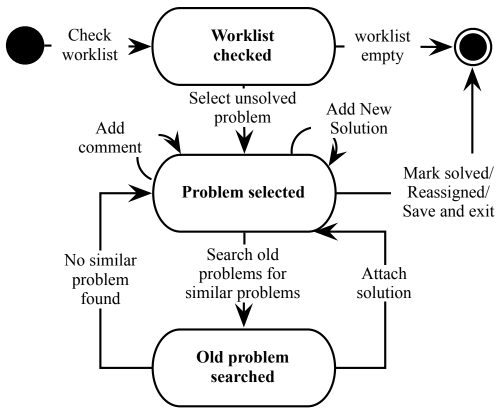


Figure 4.6: *A state chart diagram of the use case solve problem*

- **Functions:** Get problem tags, Search for problems, Create problem, Subscribe / unsubscribe to problem, Get est. time consumption of problem, Manage tag times, Get expected time of completion of problem, Get tag average time consumption, Get sorted worklist, Approve deadline, Create department, Create category, Create tag, Hide / show category, Hide / show tag, Delete person, Reset person password, Balance workload, Get statistics, and Distribute problem.

4.2 Function

The purpose of this section is to "determine the system's information processing capabilities" [17, p. 137]. Ultimately resulting in" a complete list of functions with specification of complex functions." [17, 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 sorted 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.*

Manage tag time This function takes the time a staff members has used to solve a problem and adds it to the related tags time consumption and increments the solved problems property of the inflicted tags.

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.

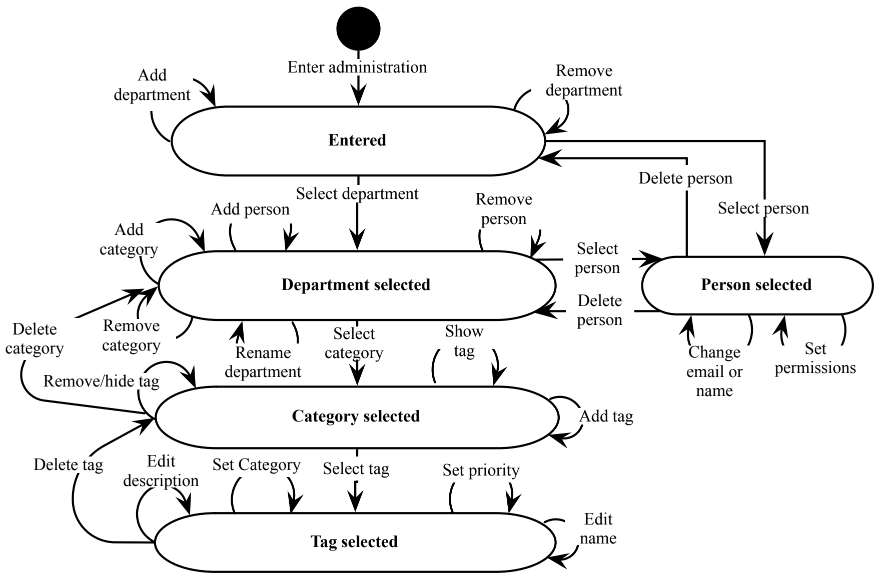


Figure 4.7: *A statechart diagram for the use case administrate*

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 est. time consumption of problem	Medium	Read/Calculate
Get expected time of completion of problem	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	Calculate/Update

Figure 4.8: *Function list*

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 persons 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 members and distributes their problems equally among the staff members 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 calculates how long each staff member uses to solve a problem in average. The function can also find the total average for departments. This function is medium.

4.3 User Interfaces

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

4.3.1 Client Interface

The client interface is illustrated in figure 4.9. After login, the client is presented with the *main* window.

Main

The clients main window allows the client to choose what to do in the helpdesk application. 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:

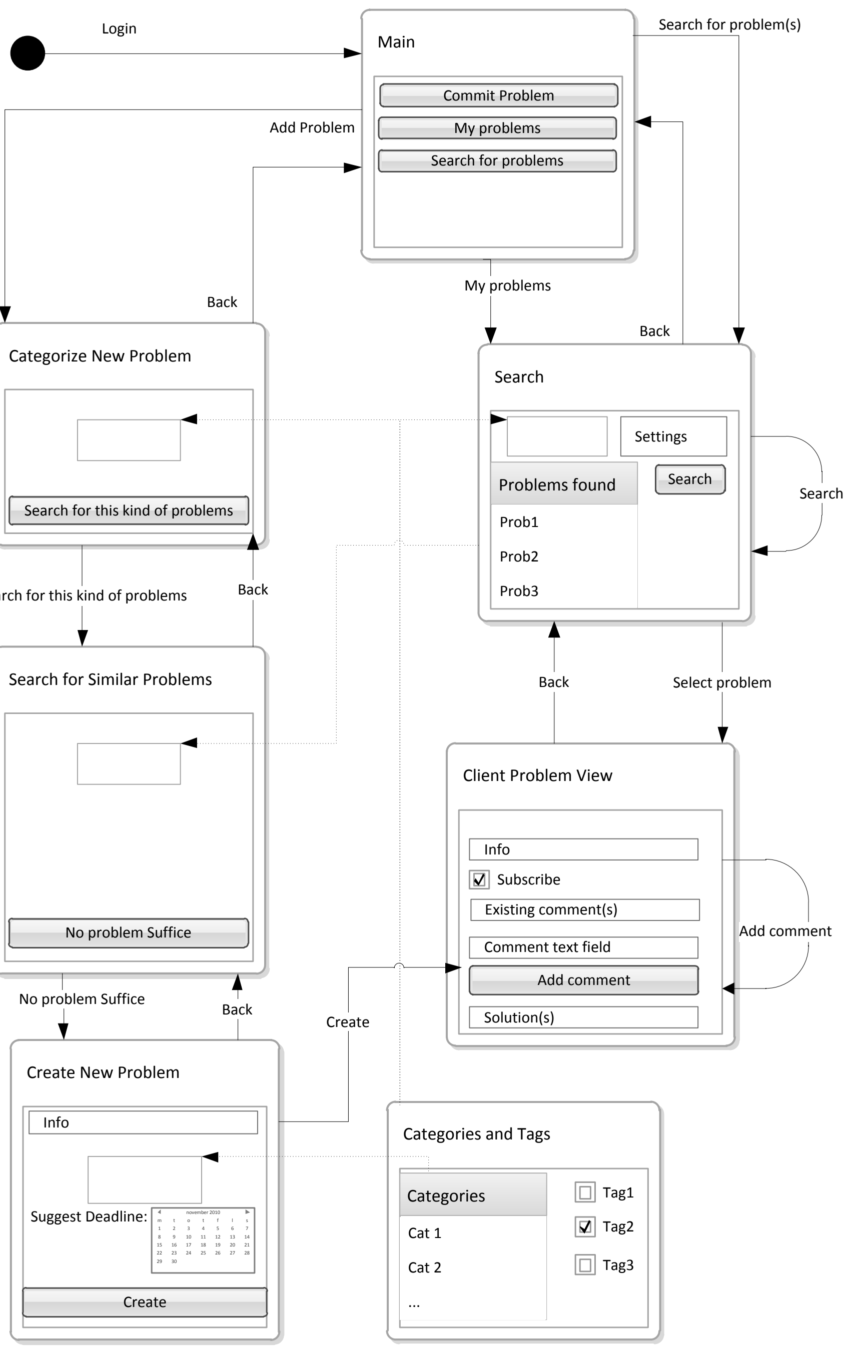


Figure 4.9: *Navigation diagram of the client interface*

- 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 clicked to view details about the problem in the “Client problem view” window.

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 performed these options are checked and they will affect 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 needs 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 performed 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 the the search property.
- A button called “No problem suffice” which loads the window “Create new problem”.

The found problems can be clicked and the user is redirected to 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 client to change and add tags so the problem is best describe.
- A calendar where a deadline can be suggested to the staff member.
- “Create” submits the problem and send the user to the “Client problem view”.

4.3.2 Staff Interface

The staff interface is illustrated in figure 4.10. After login, the staff is presented with the *main* window.

Main

The staff main window give the staff member access to all the functionality from the staff interface and from the client interface. The staff have one button:

- “My Worklist” which directs the staff member to the “Worklist” window.

Plus the menu buttons from the clients main menu.

Worklist

The “Worklist” is a list of all the problem assigned to a specific staff member. The window has the following elements:

- A list with all the problems assigned to the staff member 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.

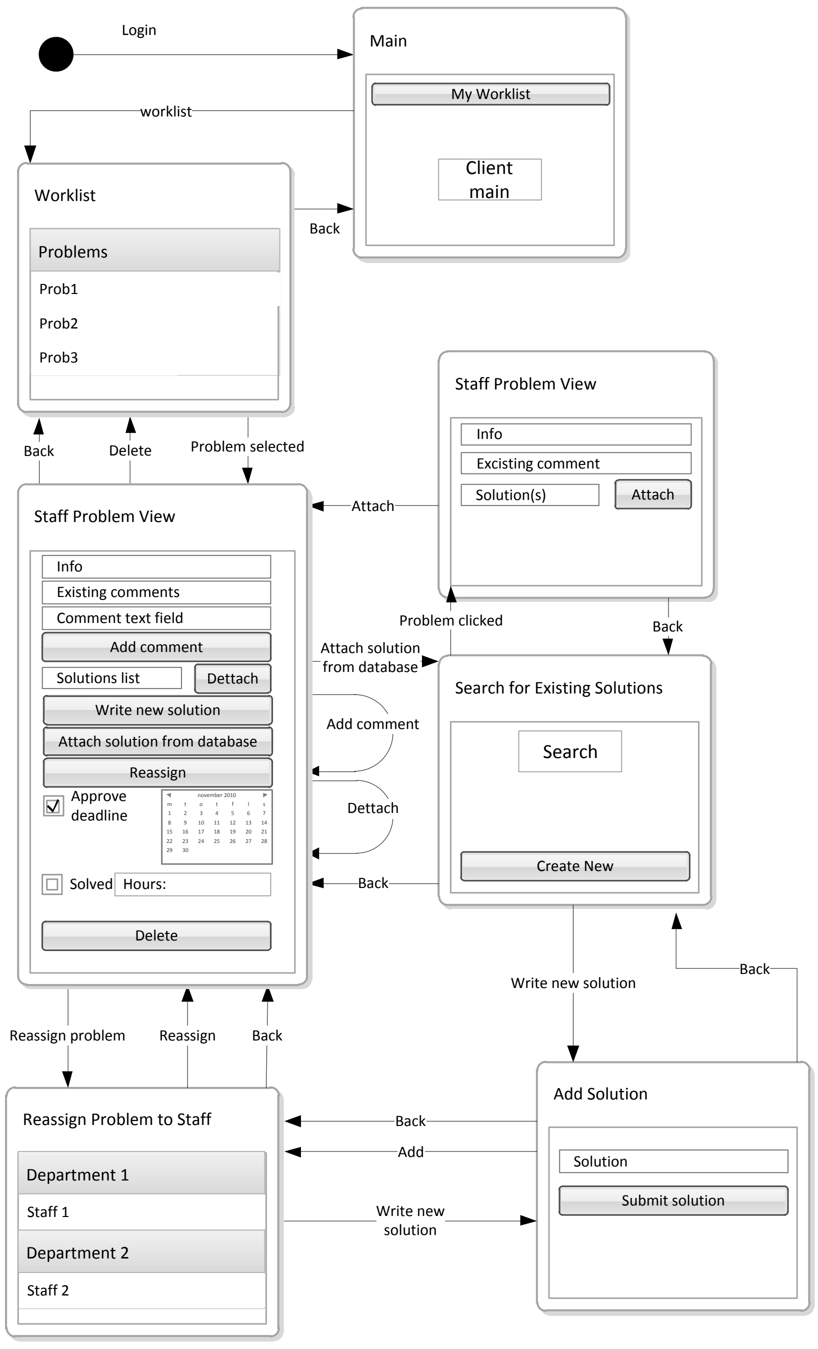


Figure 4.10: *The navigation diagram of the staff interface*

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 calendar 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 4.3.1 is reused, and it enables the staff to search for problems.
- The button "Create New" which sends the staff to 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 staff is send to the “staff Problem View” window.

4.3.3 Admin Interface

The admin interface is illustrated in figure 4.11. 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” admin window gives access to administration options and the functionality from the staff and client main windows. The window contains:

- The button “Manage Departments” 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.

The window includes the buttons from the staff and client “main” window.

Person Administration

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

- A list with all the persons in the application.

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”.

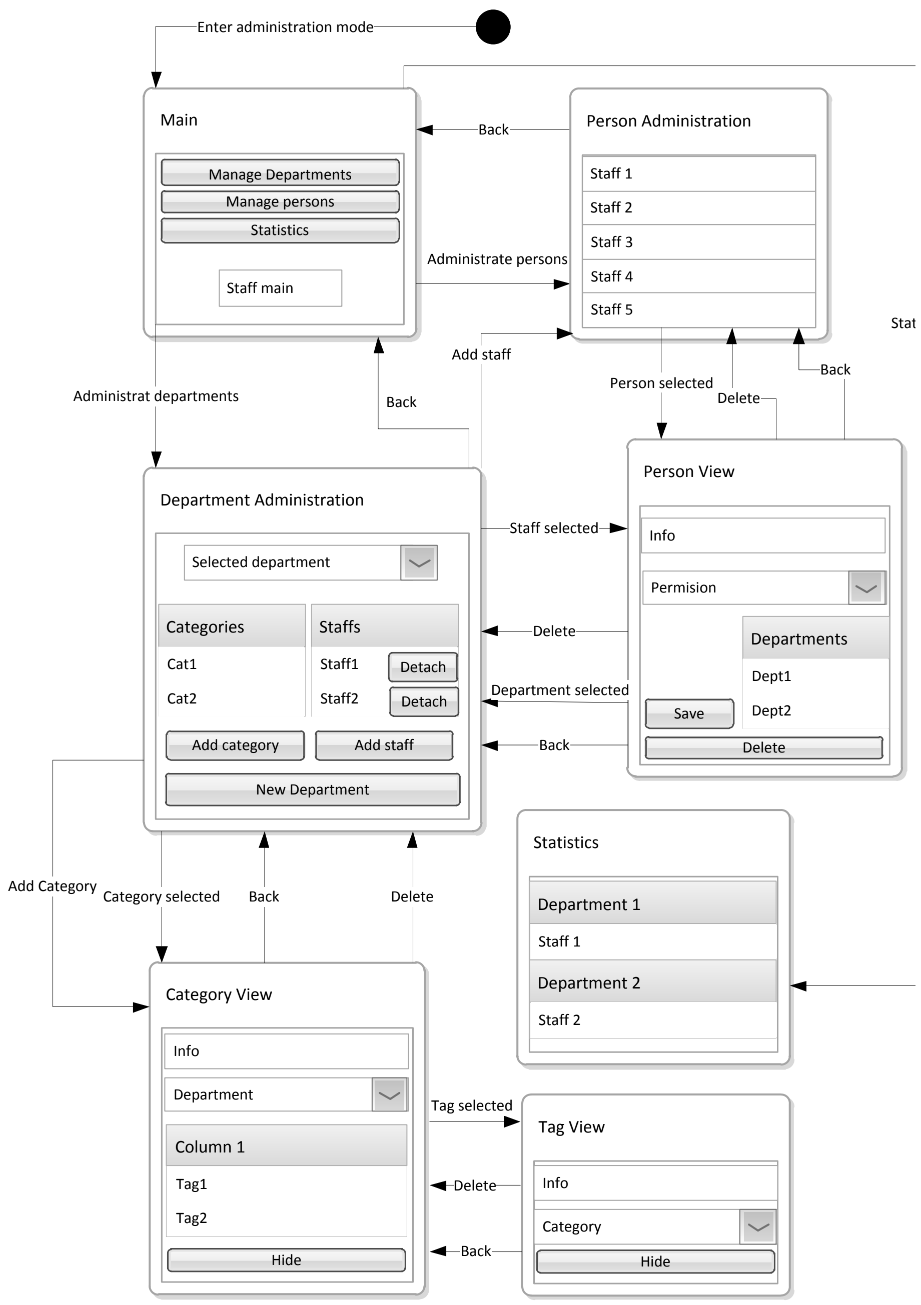


Figure 4.11: *The navigation diagram of the admin interface*

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 admin to 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 application. First the usage of our application is described, then the functions which are used to manipulate data in the Problem Domain or signal actors in the Application Domain is presented and defined, lastly the User Interfaces are described.

Part II

Design

5 Task

This chapter presents the purpose of our application in a short and precise text and gives the quality goals to which the project can be evaluate against during design, implementation, and after deployment.

5.1 Purpose

Our application should ease the process of solving and organize problems related to the physical environment of a given organization. This should be done by

- easing the problem solving process in the applied environment
- balancing the problems between the staff members based on their workload
- having a method for the staff and client to communicate
- enabling the users of the application to search through existing problems and monitor their own
- making the application accessible everywhere through the Internet and organize the already solved problems in a way which makes them easy to browse through

5.2 Corrections to the Analysis

After a review of the analysis, we come up with some corrections. Subsection 5.2.1, 5.2.2, and 5.2.3 contains the corrections, modifications, and supplement to the analysis.

5.2.1 Classes and events

The following classes are removed because they where redundant:

Deadline The class **deadline** which is now an attribute.

Client, Staff and Admin The role system is changed because we use the MVC framework, which is described in section 6.2.4. A person can have more roles. The new role structure is seen on figure 5.1.

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

Figure 5.1: *Actor & use case table*

5.2.2 Class diagram

The class diagram from figure 3.2 is modified so the following is changed:

- **Problem and tag relation**
  A tag can be connected to multiple different problems and a problem can have many tags connected to it.
- **Problem and person relations**
  A person can have from zero to three simultaneous roles. When a person have zero roles, the person is unable to do anything at all, except login. This is to give the admins the ability to bann a user, if such need should arise.

5.2.3 Interfaces

The navigation diagrams from section 4.3 are reflecting of the final design except for a few changes:

Client interface In the client navigation diagram only the Main window is changed. It is changed to a navigation bar and thereby making it easy to navigate between the different windows. When a new problem is created the user is directed to the Search window, there the user have the options to see his/hers problems or search for problems.

Staff interface Similar to the Client interface the Main window is changed to a navigation bar. In the Staff Problem View there was added a save button, to do what the name implies.

Admin interface The Main window is changed to a navigation bar. The Department administration is divided into two windows, this is done to avoid updating the two tables every time a new department is selected, see subsection 10.3.2 for a more throughly explanation.

The final navigation and design is described and shown in chapter 10.

5.3 Quality Goals

The design of Hopla Helpdesk is specified from the following quality goals: Usable, secure, efficient, consistence, reliable, maintainable, testable, comprehensible, reusable, portable, and interoperable. The definition of these quality goals – or simply criteria – is shown in figure 5.2. [17, p. 178]

Criterion	***Definition***
Usable	The end user can easily use the application
Secure	Precautions against unauthorized access
Efficient	How well the resources available are being used
Consistence	How correct the data in the model is
Reliable	The degree of the applications accessibility
Maintainable	The cost of locating and fixing application errors
Testable	The cost to ensure that the application performs intentionally
Flexible	How easily the application can be setup to fit the structure of an institution.
Comprehensible	How easy it is to understand the application
Reusable	The potential for using parts of this application in another application
Portable	How much effort needed to change the platform of the application
Interoperable	How well the application cooperates with other applications

Figure 5.2: *Definition of criteria*

We have chosen to use a four step priority scale: Very important, important, less important, and irrelevant. Each criteria defined in figure 5.2 is prioritized in figure 5.3. Furthermore the figure shows a column labeled “Easily Fulfilled” which specifies whether a given criteria will be fulfilled without much effort.

How we have prioritized the criteria is based on our application definition, which is found in section 2.2. The reasoning for the priority of each criteria in figure 5.3 is shown bellow.

	Very Important	Important	Less Important	Irrelevant	Easily Fulfilled
Usable			✓		
Secure			✓		
Efficient				✓	
Consistence		✓			
Reliable			✓		
Maintainable				✓	
Testable		✓			
Flexible	✓				
Comprehensible		✓			
Reusable				✓	
Portable				✓	✓
Interoperable			✓		

Figure 5.3: *The criteria with a priority*

Usable It is important that our application is user friendly because a helpdesk needs to be very usable to ease the process of getting help and the end user will not use the helpdesk if it is unusable. it is however not very important since this is a study project and we are more concerned with the functionality of the application than the usability, hence; less important

Secure For the Hopla Helpdesk security is less important. We want to distinguish between clients, staff members and admins – the clients are e.g. not allowed to solve problems or choose who should be assigned to what problem. We do not have any sensitive information, other than e-mail addresses and passwords. We do not take any direct measures to prevent data interception or any other serious security flaw. We do, however encrypt the passwords which are stored in the database. We do however not have any sensitive information, so we take no measures to prevent data interception or any other serious security flaw.

Efficient We do not focus on the efficiency of our application, but only that it works which lead us to irrelevant for this criterion.

Consistence It is important that end users can see which problems are solved and which are not. Furthermore the model should not contain duplicates, since it could compromise the integrity of the statistics which the application generates. This lead us to prioritize consistence as important.

Reliable The reliability of the Hopla Helpdesk application is not of great interest to us. We do not actively do anything to increase the reliability of the application, neither do we intensionally decrease it. We want to pay our attention to other criteria instead, therefore this criterion is prioritized less important.

Maintainable Since we do not intend to maintain the application after it is finished, it is prioritized as irrelevant.

Testable We want our application to work and to make sure it does, we will run tests. Therefore we want our application to be testable, hence testable is important.

Flexible It is very central to our application that it is flexible, because we want it to be generic – that it can be adapted to any organization without much or any cost. To insure this we added features to manage departments, manage persons, manage categories, and manage tags at runtime. We have chosen this to be a very important criteria.

Comprehensible Since this is study project it is important that our application is comprehensible in order to explain how it works.

Reusable Since we do not care much for the application after it is deployed, we do not care whether or not it is reusable, hence irrelevant.

Portable Our applications portability can be divided into two, the client side and the server side. We do not care about the portability of the server side, because we will rather focus on the portability on the client side and the functionality of the application. Therefore it is considered irrelevant. The end users will access our application through a browser, so we assume that it can be easily fulfilled since there are many browsers for different platforms. [13][12]

Interoperable If it is possible we want to be able to use an existing database for authentication to our application. This is however the only other application which we plan on cooperating with, therefore it is prioritized less important.

The purpose of our application is defined in this chapter followed by the quality goals which our application should fulfill when it is deployed.

6 Technical Platform

This chapter describes which choices are made with respect to equipment, system software, and design language. Further, the alternatives which we have considered in the decision process are also shown. Our decisions are primarily based on the fact that our application definition in section 2.2 should hold true, and the quality goals which are prioritized in section 5.3.

6.1 Equipment

The equipment needed to power our software can be any computer with a reasonable amount of processing power, and RAM storage together with at least one NIC(Network Interface Controller) to connect to the network.

We will be using a Dell Optiplex 960 with a Core2 Duo CPU, E8400 @ 3.00GHz, with 4 GB RAM installed. We choose this setup because we have the opportunity to borrow it from our IT department at AAU.

6.2 Software

Our application relies on two things in order to run:

- Database
- Web server

These two are described in following subsections.

6.2.1 DBMS

Using an external DBMS allows us to access the data from multiple machines, making the application easier to distribute, run, and develop.

The differences which we have considered between these two DBMS's are listed in figure 6.1. [14]

Our application use a DBMS supporting the query language SQL and the relational database model. [24]
We choose Microsoft SQL Server version 10.50.1600.1 in favor of PostgreSQL,

because PostgreSQL is less integratable with Visual Studio 2010, which we use as our IDE.

6.2.2 Operating System

We are using Microsoft Windows server 2008 R2 Enterprise because we choose to run the Microsoft SQL Server 2008 R2 operating system.

6.2.3 Web Server

We chose to create a web application in order to minimize the amount of software the users of our application have to install before using it, in order to run this application it needs a web server. As Microsoft Windows Server 2008 R2 already has Internet Information services (IIS), which contains a web server, we find it natural to use this, however, we never deploy the application, and only test it on the development web server built into Visual Studio.

6.2.4 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 environment in which the users (developer, customer etc.) could preserve the original perception of the data structure, while being able to view and edit portions of it. [22]

Components

The MVC framework consists of three different types of components; Views, controllers and models. Each created with a different purpose, see figure 6.2

Models The models are the components that handle the data domain of the application, it is typically mapped to a database 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 addition of C# or Visual 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 decide which view should be rendered. If the data which the view should display does not exist in the model (input, etc.) the controller can also provide this data to the view.

Master-page Master-pages is not a part of MVC as a design pattern. 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#/ Visual Basic, however, it also has content-containers. These containers can be created anywhere on the web page, and are filled with content by views.

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 different components within the web page. When entering an URL the user actually call methods in controllers, the user can even add parameters to the calls. The controllers then redirects the user to the correct view based on the input and the method called. The view then executes, which results in a plain web page that can be displayed in a browser.

Because of the separation into three independent modules, developers are able to create different parts of the application with different kind of approaches, thus enabling them to better manage complexity, create applications with a high maintainability, and create UI that does not have to show the actual data structure.

Testing When creating web applications with MVC you have the possibility of creating unit tests to thoroughly test your code. Test scaffoldings can be automatically generated from within Visual Studio, and run as a separate 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 13.2. [**?** **?**]

6.2.5 ADO.NET Entity Framework

The ADO.NET Entity Framework (EF) is designed by Microsoft with the purpose of making a disconnected data architecture[1] to utilize with the .NET framework [7]. 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 [16].

The ADO.NET EF is an object relational mapping framework [9] 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 an 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 [8]. 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.

6.3 Design Language

As we have chosen to develop a web application, we are limited to the programming and markup languages in that domain, most commonly (X)HTML. Because we have all followed a course in object oriented programming (OOP) we would like to harness our recent knowledge of this type of language. We were taught Visual C# and we will use this because we are all familiar with

[1]A disconnected data architecture is simply that the connection is closed between request instead of being kept open until the session is closed.

it. Fortunately the MVC 2 Framework from Microsoft allows us to combine the two, and use HTML and C# alongside each other.

6.3.1 Coding Standard

All functions, properties, and classes will follow big camelcase naming convention, as well as public variables. All private variables names will as seen in code snippet 6.1 be prefixed by an underscore and use small camelcase. Input variables are like private variables except for the absence of the underscore.

```
class SampleClass
{
    // private variables should be written as smallCamelCase
        // with an underscore as prefix
    private int _privateVariable;

    // public variables should be written as BigCamelCase
    // without an underscore as prefix
    public int PublicVariable;

    // properties should be written as BigCamelCase
    public int DummyProperty
    {
        get { return _privateVariable; }
        set { _privateVariable = value; }
    }

    // functions should be written as BigCamelCase
        // input variables should be written as smallCamelCase
    public int DummyFunction(int inputVariable)
    {
        _privateVariable = inputVariable;

        return _privateVariable;
    }
}
```

Code snippet 6.1: *SampleClass.cs*

This chapter describes the choices made with respect to equipment, system software, and design language. The choices which we have decided among are all presented as well.

Feature	**PostgreSQL**	**Microsoft SQL Server**
Accessible	BSD Open source	Requires license, but we do have this through the university.
Cooperative with our tools	Hard to incorporate into Visual Studio 2010.	Easy to incorporate into Visual studio 2010.
Experience in our work group	We have had a course based on PostgreSQL.	None, but resembles other SQL DBMS's.

Figure 6.1: *DBMSs compared*

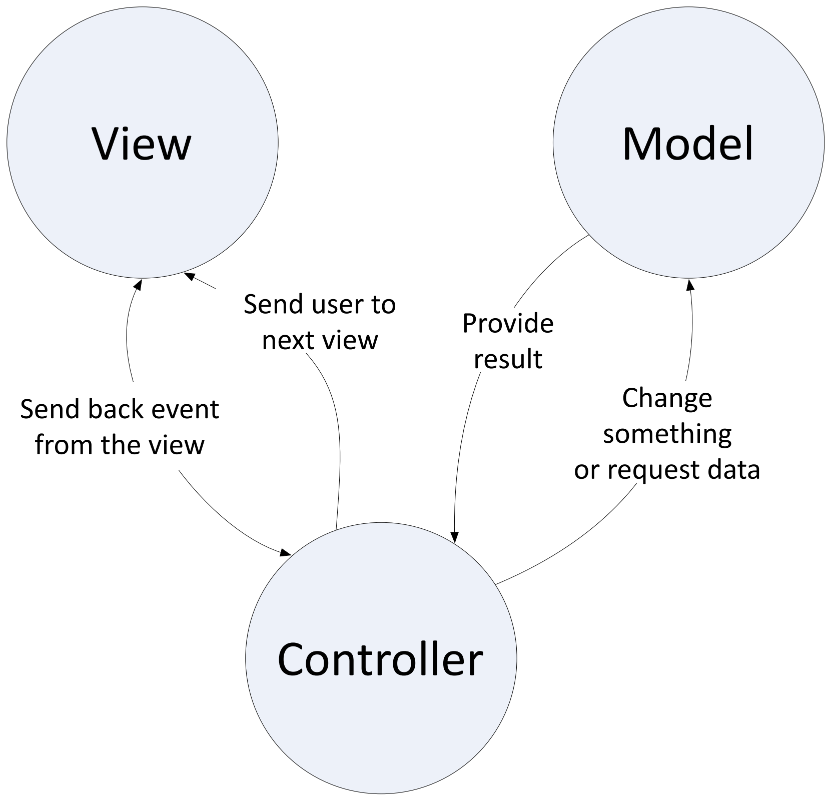

View
Model
Send user to next view
Provide result
Send back event from the view
Change something or request data
Controller

Figure 6.2: *These are the three main components in the MVC*

7 Architecture

This chapter shows the layered architecture and the client-server architecture of our application. Both of these pattern architectures are described along with our own use of these patterns in our application.

7.1 Component Architecture

On the basis of section 2.2 and our evaluation of criteria in section 5.3 we found that our main priority is flexibility. In order to obtain high flexibility we will use ASP.NET MVC 2 framework described in section 6.2.4, to make components easily swappable. The the application will access the database by using the ADO.NET data provider described in section 6.2.5. This gives us a component design as seen on figure 7.1. The figure illustrates the components dependency among each other.

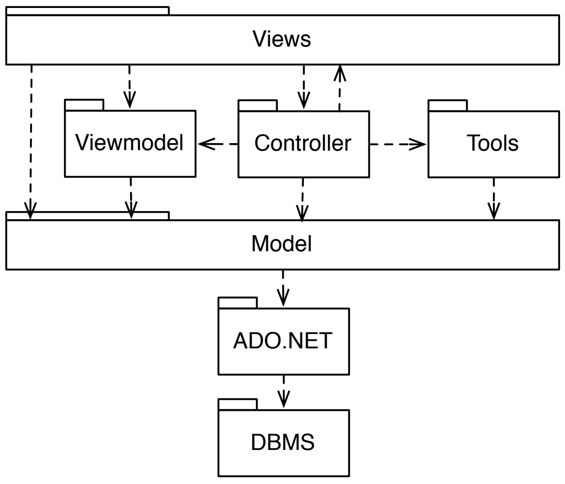


Figure 7.1: *The application component. The arrows represent dependency*

Model The responsibility of the model is to represent the objects and their data which is stored in- and fetched from the database. This component is further described in 8.1.1.

Tools The tool component consist of various help functionality. For example the problem distributer and the search functionality is placed in this component. The tools is used by the controller and it relies on the model.

View Model view models are dependent on the model since it is a container of elements from the model.

View Views dependent on the model, controller and view model. It is dependent on the controller because the views responsibility is to make buttons and links which have to match the correct controller. It is dependent on the model and view model because this is the data it needs to present. The dependency on the model could be removed and then all view data has to be parsed though a view model, but we allow this because it is not always necessary to put the model into a view model.

Controller The controllers all have independent responsibilities, but the general responsibility of the controller components is to process the data in the model and return this to a specific view, possibly through a view model. The controller depends on view, model, view model and tool. Changing any of these components will most likely result in a change of the controller.

ADO.NET This component is not build by us, this is an already made component which is used as a database data provider.

7.1.1 Client-Server

Because we are designing a help desk, we are dealing with users who are not present in a specific location. Therefore we also designed the application using a client-server architecture. On the client side there is a user interface and on the server side the functionality and model are located as seen on figure 7.2. By using Local Presentation [17][p. 200] we enable clients to access the application from anywhere, and still keep the functionality and model on the server and thus keeping the application flexible.

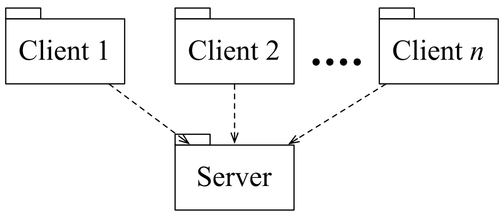


Figure 7.2: *The client server design pattern*

The architecture of our Hopla Helpdesk application is described in this chapter along with standard patterns which we use.

8 Components

This chapter documents the internal structure of the components which comprise our application and a definition of each class within each component. In short this chapter present the entire application on a class level. This chapter is important because it documents the structure of all the components which Hopla Helpdesk consists of.

8.1 Structure

Hopla Helpdesk consists of seven components: View, View model, Controller, Tool, Model, ADO.net, and the DBMS. The structure of the most relevant components are described in this chapter. These components are: Model, Controller, and Tools.

8.1.1 Model

The class diagram showed in figure 3.2 is altered to apply the MVC design pattern. The new class diagram can be seen on figure 8.1. Each class will be described in detail in the following.

There are additional navigation properties on some of the classes such as the **Solutions** attribute on **Problem**, which lists the attached solutions. These are not outlined below as they are generated by the ADO.NET entity frame. ADO.NET is described in section 6.2.5.

Problem

Purpose: To represent problems and their relations to persons, comments, tags, and solutions.

Attributes: Id, Title, Description, Added_date, Deadline, IsDeadlineApproved, Reassignable, and SolvedAtTime

Operations: *GetPriority* – the priority of the problem, *MangeTagTimes* – saves the solve time to the tags attached to the problem, *CalculateETA* – calculates the estimated time of completion, and *CalculateEstimatedTimeConsumption* – calculates the estimated time consumption.

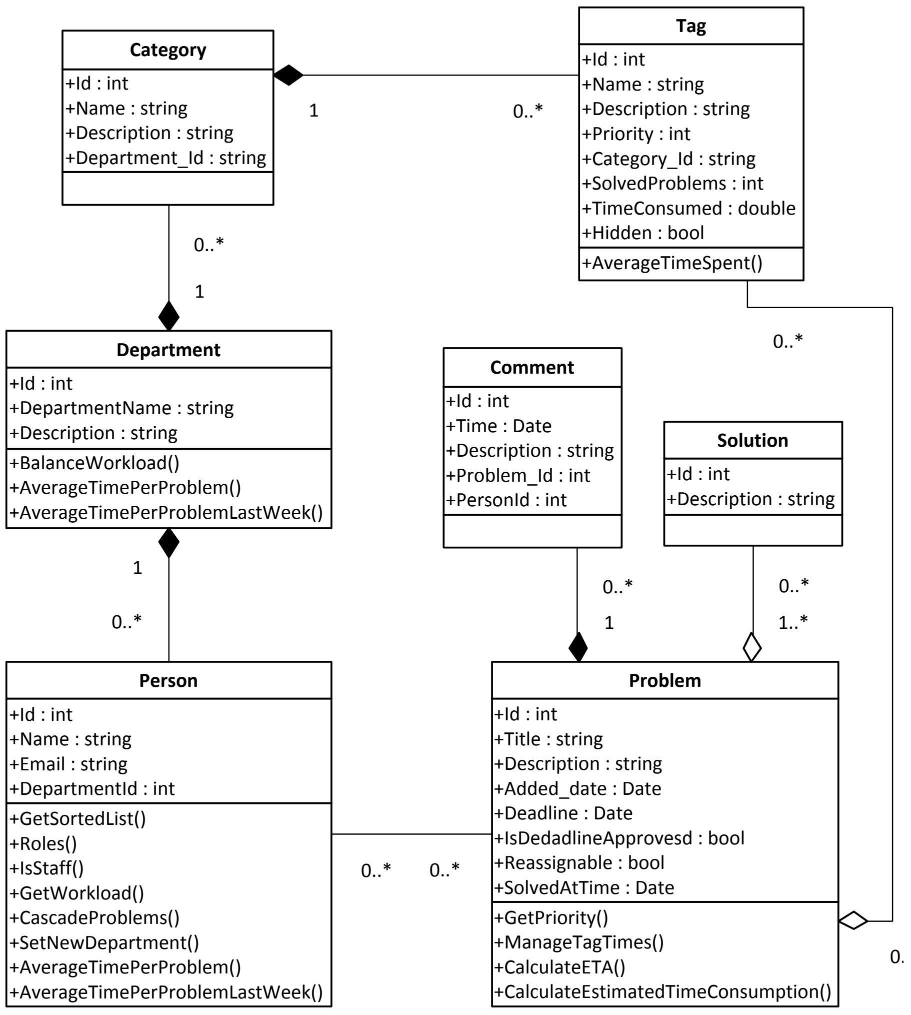


Figure 8.1: *The class diagram*

Person

Purpose: To represent the persons using our application and their relations, as well as perform operations associated with a person.

Attributes: Id, Name, Email, and DepartmentId

Operations: *GetSortedList* – gets the worklist for a staff member in sorted order, *Roles*, *IsStaff*, *GetWorkload*, *CascadeProblems* – reassign the problems to other staff members of a given department, *SetNewDepartment*, *AverageTimePerProblem* – gives the average time it takes for problems to be solved, and *AverageTimePerProblemLastWeek* – gives the average time it takes for problems to be solved during the last seven days.

Department

Purpose: To represent departments and their relations, as well as handle some statistics and balance the workload across the persons associated with the department.

Attributes: Id, DepartmentName, and Description

Operations: *BalanceWorklaod* – balances the problems between staff members of the department, *AverageTimePerProblem* – gives the average time it takes for problems to be solved, and *AverageTimePerProblemLastWeek* – gives the average time it takes for problems to be solved during the last seven days.

Solution

Purpose: To represent solutions and their relations.

Attributes: Id and Description

Operations: none

Category

Purpose: To represent categories and their relations to departments and tags.

Attributes: Id, Name, Description, and Department_Id

Operations: none

Tag

Purpose: To represent tags and their relations to categories and problems.

Attributes: Id, Name, Description, Priority, Category_Id, SolvedProblems, TimeConsumed, and Hidden

Operations: *AverageTimeSpent* – the average time to solve problems with the given tag.

Comment

Purpose: To represent comments and their relations to persons and problems.

Attributes: Id, Time, Description, Problem_Id, and PersonId

Operations: none

8.1.2 Controller

The controller classes provides functionality to the application. They are responsible for sending data from the Model to the View component and updating the Model. The controllers does not have any noticeable attributes, hence the attributes entries are not applicable here.

Account

Purpose: Register new users, manage the user login and authorize user roles.

Operations: *ChangePassword* – change the password of a **Person**, *Register* – register a new **Person**, *LogOff*, and *LogOn*.

Person

Purpose: Manage the objects of the class **Person**, which represents the users of the application.

Operations: *Create*, *Details*, *Edit*, *Delete*, *ChangeDepartment* – changes the department of a **Person** who is a staff, and *ChooseDepartment* – selects a department for a **Person** who is a staff, but does not have a department.

Department

Purpose: Create and manage departments.

Operations: *Create*, *Details*, *Edit*, and *Delete*

Category

Purpose: Create new categories and manage categories as well as aggregate categories and tags.

Operations: *Create*, *Details*, *Edit*, *Delete*, *HideUnhide* – toggles the **Hidden** property of every **Tag** belonging to the **Category**, and *TagHideUnhide* – toggles the **Hidden** property of a single **Tag** belonging to the **Category**.

Tag

Purpose: Create, delete and manage tags.

Operations: *Create*, *Details*, *Edit*, and *Delete*

Problem

Purpose: Create and manage problems.

Operations: *Create*, *Details*, *CategorizeNewProblem* – allows a client to categorize a new problem, *SimilarProblems* – finds problems using the *Search* method which is described later in figure 8.3, *Subscribe*, and *Unsubscribe*

Home

Purpose: The Primary purpose of this controller is to redirect to user to the login screen or to the views which the user is allowed to use. The secondary purpose is to create client, staff and admin roles if the helpdesk does not contain these roles. If there is no users with the admin role the home class will create a root user.

Operations: *Index.*

Reassign

Purpose: Assign problems to staff members and reassign problems to staff members on request.

Operations: *Assign* – assigns a problem to a staff member.

Statistics

Purpose: Calculate statistics for departments and staff members.

Operations: *Index* – sends statistical data to a view.

8.1.3 Tools

The tools classes is a collection of static methods that is used to implement advanced functionality, however, they do not display any content. The Tools component does not have any noticeable attributes, hence the attributes entries are not applicable here.

ProblemDistributer

Purpose: Distribute problem based on problem tags and staff members workload.

Attributes: none

Operations: *DistributeProblems* – seen in 8.2.

ProblemSearch

Purpose: Allows users who is logged on as client or staff to search through problems, the search is based on tags and status.

Operations: *Search* – seen in figure 8.3.

SQL

Purpose: A collection of functions for SQL query and insertion for the aspnet membership database

Operations: Check if user have a role, add role to user, check if user is staff, check if user exists, remove role from user, get all roles for a user, get all roles, assign client to user, add a new role, remove user from aspnet membership, reset password, get email for user.

AverageAllTags

Operation	DistributeProblems	
Category	_Active xPassive	xUpdate _Read xCompute _Signal
Purpose	Distributing problems between staff members in each department, this is done to make sure that a single staff member do not have all the problems.	
Input data	Problem id, staff id	
Conditions	There must be problems attached to the department calling distribute functions.	
Effect	Problems will be distributed equally between staff member until their workload balanced.	
Algorithm	The algorithm has to find the staff member with the minimum and the maximum workload, and then distribute their problems between these two staff members, until their workload is balanced. The algorithm has to do this x times where x is the number of people in the department minus one.	
Datastructures	Lists	
Placement		
Involved objects	Staff id, department id, problem id	
Triggering events	Problem added to a department	

Figure 8.2: *The operation specification of DistributeProblems method from the **ProblemDistributer** class*

Operation	Search	
Category	xActive _Passive	_Update xRead xCompute _Signal
Purpose	Make is easier for users to look through problems.	
Input data	Tag, status.	
Conditions	The user must be logged in as either client or staff.	
Effect	A page with search results will be shown.	
Algorithm	Searches through the model for all input tags, whereafter it searches again for all input tags minus one, then minus two, etc. until sufficient number of problems has been found.	
Datastructures	Lists	
Placement		
Involved objects	Tags, status, client or staff	
Triggering events	This operation occurs whenever a client or staff commit a problem or search for a problem.	

Figure 8.3: *The operation specification of Search method from the **ProblemSearch** class*

Purpose: Calculates the average time spent for a problem which has no tags.

Operations: Calculate average time.

MaxPriority

Purpose: Return the maximum priority which exists in the model.

Operations: Find maximum priority.

StatTool

Purpose: Returns the average time consumed per problem, which will be used to calculate statistics.

Operations: Calculate average time consumed per problem.

The internal structure of the three main components – model, function, and interfaces – and their subcomponents is presented in this chapter. This chapter also defines each of classes in our application.

Part III

Implementation

9 Development

The tools which we have used to develop our application are specified in this chapter along with the development methods. This is important since it is vital for the understanding of the work process of the development phase.

9.1 Development Tools

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

9.1.1 IDE

We use Microsoft Visual Studio Ultimate [10] as our development tool, which includes a large variety of built in tools for unit testing, SQL connections and data modeling. We have experience with development using Visual Studio Ultimate from the Object-Oriented Programming course. Alternatively MonoDevelop [19] can be used – which is an open-source cross platform .NET IDE – however, since none of us have experience with this tool we prefer Microsoft Visual Studio.

9.1.2 Collaboration

For collaboration we use subversion(SVN) along with we use AnkhSVN [4] to subvert our code. AnkhSVN is a source control provider for Microsoft Visual Studio. Alternatively we could use Team Foundation Server [6], but this requires installation and configuration of a Team Foundation Server. We choose AnkhSVN since we already have a running SVN server.

9.2 Development Method

In this section a few different Software development methods that we considered using will be described. Subsection 9.2.5 explains the method we use in this project, which is a combination of the different methods described in this section.

9.2.1 Waterfall

The waterfall method [23], first published by Dr. Winston W. Royce in 1970 as a flawed method of development, has derived its name directly from the concept of the method. If following this method in its traditional form, development will go through several completely separated steps, from which it is never possible to go back as seen on figure 9.1. These steps will not necessarily be done by the same development teams which means that everything has to be documented thoroughly before starting the next step.

Royce originally meant for this method to be developed into a fully iterative model, however, most software development companies adapted to non-iterative form.

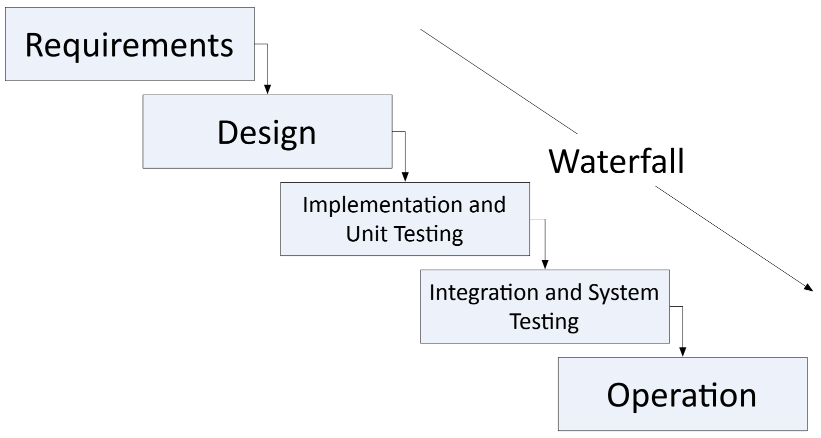

Requirements
Design
Waterfall
Implementation and Unit Testing
Integration and System Testing
Operation

Figure 9.1: *This illustrates a traditional waterfall in which the "water" can only proceed downwards until reaching the end (process completion)*

Unfortunately, this software development method has a lot of drawbacks, the main one being its inflexibility. Because the method is not designed to handle change in the specification during the development period, the software developed with this method could in the worst cases be useless by the time the development finishes. Therefore it is very common for software developers to incorporate change during the development, however, because the documentation for every little piece of work has already been made at this time, it is a colossal task to change even the smallest features. Because the changes must be made to all the documentation prior to the step where the change is made.

9.2.2 Agile Methods

Because of the traditional and wide spread development method being so inflexible and difficult to work with, a lot of other methods is suggested. These methods is commonly known as "Agile" [11] and [15]. In 2001 a large group formed by representatives from them most popular Agile methods came together to form "The Agile Alliance". This alliance created the "Agile Manifesto", which states four things that the alliance think Agile Development should be:

"

- **Individuals and Interaction** over processes and tools.
- **Working Software** over comprehensive documentation.
- **Customer Collaboration** over contract negotiation.
- **Responding to change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more. [11]

"

9.2.3 Scrum

Scrum is an Agile development method that relies heavily on self-organizing teams. Scrum is best explained by the different components it contains:

Customer / Product owner

The customer is the person or organization for whom the development team is currently working.

ScrumMaster

The ScrumMaster is the manager of the scrum process, and is there to ensure that all developers use Scrum to achieve the maximum level of performance.

Development Team

The Development team is a set of developers working on the project. These can vary in size depending on the size of the task they have to solve.

User Stories

This is a set of stories in which the customer explains in plain language what features the product should contain, and how he/she would like them to work.

Product Backlog

This is the complete list of features – or user stories – that the customer would like the product to have. Each feature is prioritized to determine the order of which these should be implemented.

Sprints

Sprints are a limited timespan, in which developers work on coding, testing and implementing the features contained in the sprint backlog.

Sprint Backlog

The Sprint backlog is the set of features that the specific development team is working on in the current sprint.

Burndown Charts

Burndown charts are the set of features that still lack implementing. Typically, a burndown chart is created for both the individual sprints and the entire project.

Putting the Components Together

When understanding all of these “components”, one can begin to understand how they are working together to create Scrum.

The main component in Scrum is a sprint; any sprint starts with a planning meeting in which the development team decides what features from the product backlog to work on, starting from the highest priority. This priority is decided by the product owner. When they have decided upon the features, the development team transfers them to the sprint backlog and start working.

Once a day during the sprint all team members meet with the ScrumMaster and the product owner and talk about what they have been working on the previous day, what they will be working on that day, and what might be holding them from proceeding. These meetings are held to synchronize the team members.

After completing a sprint the team demonstrates all the features added to the product. This is done to assure that the customer and the developers are on the same page. Of course this meeting might lead to alteration of the recently implemented features, however, it might also lead to more features being added to the product backlog.

After this meeting the team starts from the top with a new planning meeting and repeat this process until the customer is satisfied with the features implemented.

9.2.4 Extreme Programming (XP)

Extreme Programming differs from the other agile methods as it is more a set of recommendations than an actual method. All sources in this subsection is from the following cites [2, 3]. The recommendations of XP are as following:

Planning

There are three different overall plans in XP:

- **Release Planning:** The Overall plan for the project
- **Iteration Planning:** The plan for this iteration of development. This is very similar to the Scrum Sprints
- **Daily Planning:** The day to day planning between developers.

Small Releases

This is simply about getting working software to the customer as fast as possible, better make a small release fast, than a large release slowly.

Metaphor

Find a metaphor in the real world that resembles your project.

Simple Design

Design only what you will be doing in this iteration of the project. Do not spend time designing something you might not even need later on.

Testing

The Development of the project should actually be test-driven. That means that you write unit-tests before you write actual code. Also, XP recommends that you run all tests simultaneously, so that you can see if a bug-correction has fixed other problems.

Refactoring

Make the code simple to modify and understand, modify existing and working code so that it is easier to modify and understand - you never know when you have to add features using or modifying existing code.

Pair Programming

In Extreme Programming one developer should never work alone. Two developers always work alongside on a single computer. XP states that this will deliver code just as quickly as two developers with two computers, however, the code will be of higher quality.

Collective Ownership

Nobody owns any code – all code can be altered by anyone. This is possible because unit tests already exist for everything which allows anyone to refactor or optimize the code as long as the unit tests pass afterwards.

Continuous Integration

Always integrate the feature you have just written in the project right away. This enhances the possibility to make a lot of small releases, and eliminates any large scale integration project at the end of development.

No Overtime

In XP projects no one should work overtime on a regular basis. Of course it can be necessary if something in the planning slipped, however, if it becomes a regular thing something is wrong with the planning.

On-Site Customer

XP recommends that there is always a customer available, preferably on site. This will enhance the collaboration between customer and developers and hopefully developers will not have to change as much as of the features that they have implemented.

Coding Standards

This is simply needed to implement some of the other concepts of XP, however, coding standards should not be enforced by someone higher in the company hierarchy, and it should be implemented by the developers themselves.

9.2.5 Development During the Project

In our project we do not strictly follow any specific development method. Instead we implement elements from scrum and extreme programming. At the start of each day we hold a meeting to discuss what we will be doing during the day. Furthermore we have a lot of small sprints, where we focus on developing, integrating and testing new features. These sprints are very short, typically maximum a couple of days, because our development phase do not last for more than a couple of weeks.

Beside the principles borrowed from scrum, we also use a lot of elements from Extreme Programming:

- We plan our project in a similar way to XP.
- We are constantly been making small releases
- We write a release every time we have developed/implemented a new feature.
- We have collective ownership of all our code.
- We write tests for all our major components.
- We are continuously integrating the created features into our main application.
- We do not have regular overtime in this project.
- We try to maintain coding standards.

Using these elements it much easier for us to delegate our work and plan all the aspects of our project. Also, it has given us a better understanding of our code, as all collaborate to achieve our common goal, and often edits each other's code. The swift development of our application and the fact we almost never work late has helped us to maintain a high interest in the project.

The development tools and methods is outlined in the chapter.

10 Program Presentation

This chapter outlines and present the process of using our application based on the three user roles. This is done to show that our application usage is consistent with the use cases we presented in the Application Domain Analysis, chapter 4.

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

- Client
- Staff
- Admin

As with all users of the application, 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 application 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 10.1. The points in the menu changes depending on which privileges the current user has.

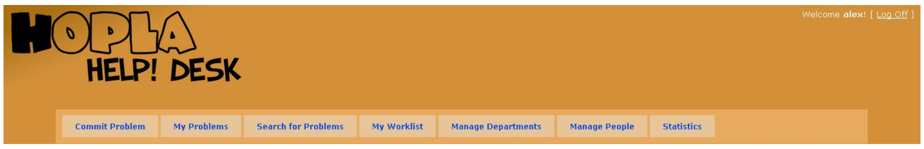


Figure 10.1: *The menu which the master file is responsible for making. The Commit Problem, My Problems, and Search for Problems are menu points which are allow for clients. The My Worklist menu point is a menu point for the staff users. The Mange Department, Mange People, and Statistics menu points are allowed for admins only*

In subsection 10.1 the most common usages of our application is described. These are based on the use cases from our analysis in section 4.1.2. We will start by presenting the clients usage.

10.1 Client Usage

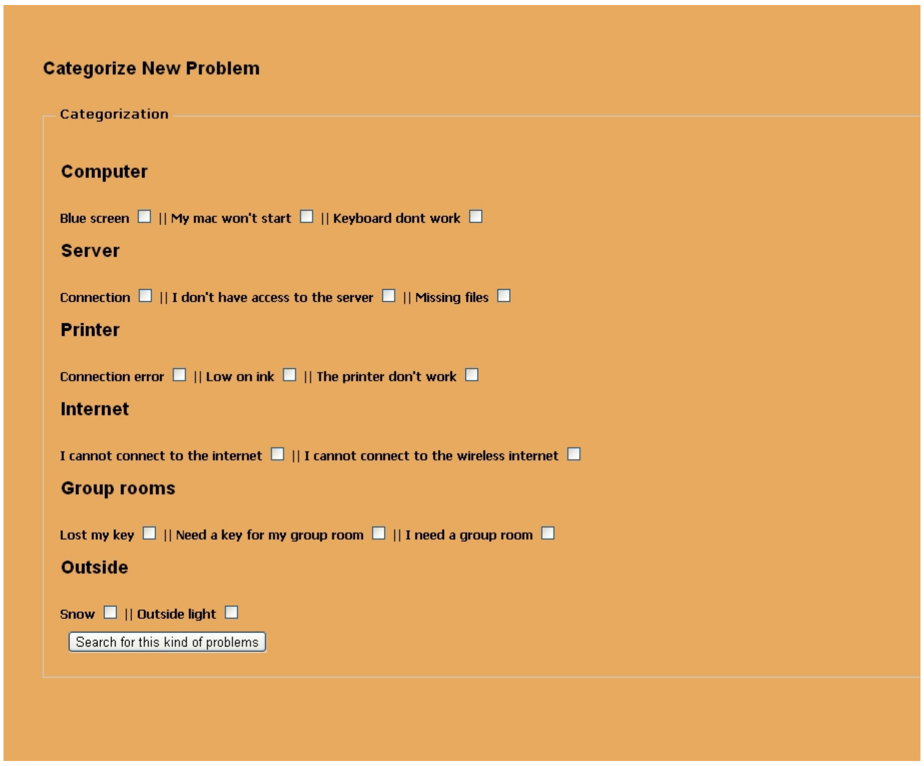
The client can generally do three things; commit a problem, see status of his/her problems, or search the database for problems. The first two are based on their corresponding use cases in section 4.1.2. The search function was added as a functionality for client because we assume that it would primarily be persons with a new problem who might want to search for others problems.

The following subsections describes the three usage of the application which the client has access to. The process of searching for problems and seeing status of the clients problems have switch places in the subsections below because in order for a client to check the status of his/her problems a search will be initiated.

10.1.1 Commit a Problem

The process of committing a problem consists of three steps: Categorizing the problem, considering existing problems, and if no problems suffice the to match the new problem; a creation step.

Categorizing the Problem



Categorize New Problem

Categorization

Computer

Blue screen ☐ || My mac won't start ☐ || Keyboard dont work ☐

Server

Connection ☐ || I don't have access to the server ☐ || Missing files ☐

Printer

Connection error ☐ || Low on ink ☐ || The printer don't work ☐

Internet

I cannot connect to the internet ☐ || I cannot connect to the wireless internet ☐

Group rooms

Lost my key ☐ || Need a key for my group room ☐ || I need a group room ☐

Outside

Snow ☐ || Outside light ☐

Figure 10.2: *The categorization step in committing a problem, note that the page has been cropped*

The process of committing a problem starts with a click on the Commit Problem menu point, which can be seen in figure 10.1. When Commit Problem is clicked the user is asked to categorize the problem in the window showed in figure 10.2. As the figure shows the tags, the check boxes, are ordered under headlines. These headlines are categories.

Consider Existing Problems

When the problem is categorized the application searches for problems matching the specified tags, the search function is described in 12.2. The problems found in the search are listed in a table with columns showing number of matching tags, deadline, estimated time of completion, title, and description. The last column also shows whether the given problem is solved or not.

Create New Problem

The client can choose problems and subscribe to them in order to receive notifications when the problems is updated, e.g. a solution is attached to them. 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 10.3. The categorization part of this page remembers what was entered in the categorization step, but allows the client to change it if he/she wants to reconsider the chosen tags. The tags which were marked in the categorization step are remembered and are also marked when this step is started.

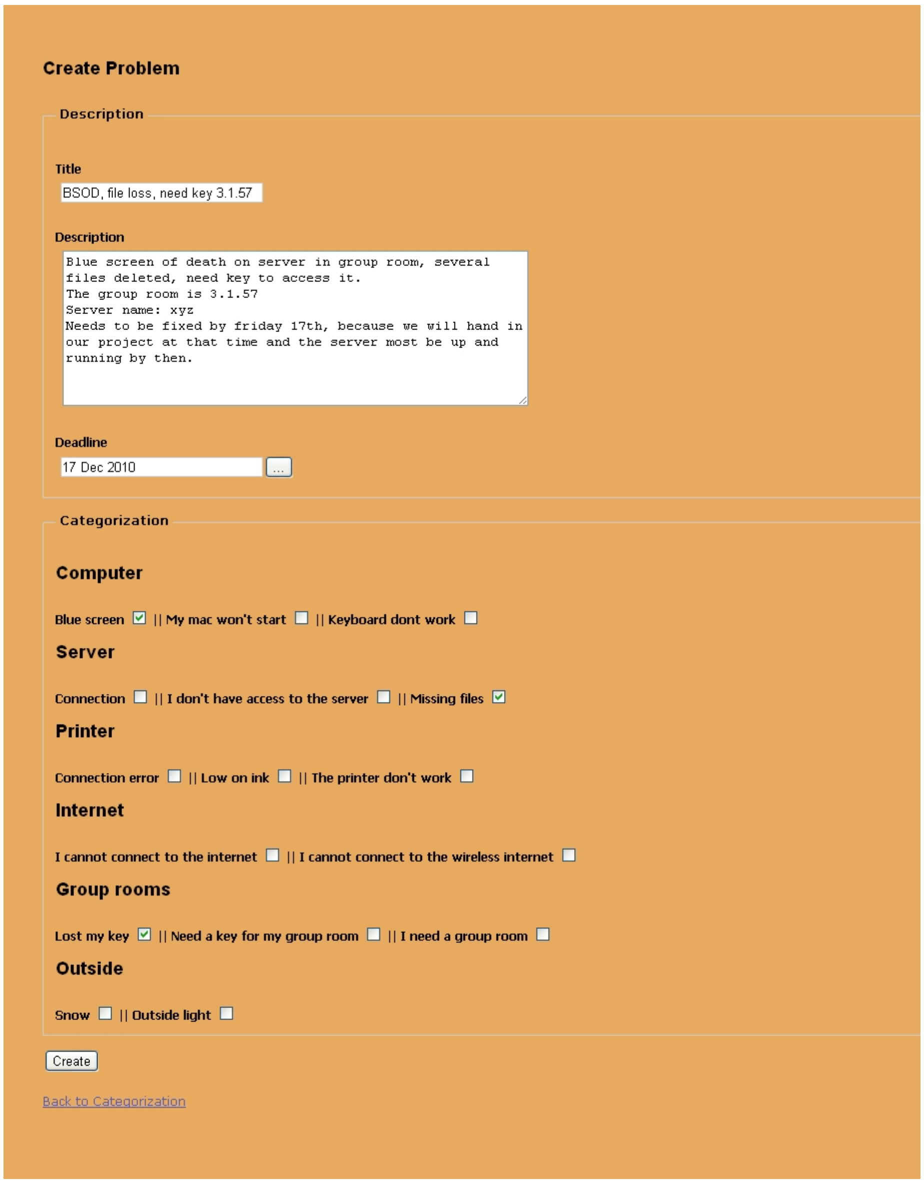
When the problem has been described and given a title it can be created. Optionally the client can give the problem a deadline for the staff member assigned to the problem to consider. When the problem is created the client is presented with a page showing whether or not the problem was successfully added to the application. To see the problem later, the client can either search for it, as described in subsection 10.1.2, or go to the My Problems menu point, which is described in subsection 10.1.3.

10.1.2 Search for Problems

To search for problems, the client must select the menu point called Search for Problems, which can be seen in figure 10.1. The page which is then shown can be divided into two fields; search field and result field. The search field is similar to the categorization step of the commit new problem usage, which can be seen in figure 10.2 except that a few more options, namely: Only my problems, only unsolved problems, only solved problems, and minimum number of problems to find. The three first are represented as check boxes and the last is a text field, which accepts a number above 0.

If the only my problems check box is checked, the result will only show problems which the client is subscribed to. The only solved problems and only unsolved problems check boxes does as their names indicate. If both check boxes are checked, no problems will be found of course.

The minimum number of problems is used as an input to the search function, which specifies how many problems should be found before the search function stops running. The reason for this is that the search is done stepwise, lowering



Create Problem

Description

Title

BSOD, file loss, need key 3.1.57

Description

```
Blue screen of death on server in group room, several
files deleted, need key to access it.
The group room is 3.1.57
Server name: xyz
Needs to be fixed by friday 17th, because we will hand in
our project at that time and the server most be up and
running by then.
```

Deadline

17 Dec 2010

Categorization

Computer

Blue screen ✓ || My mac won't start ☐ || Keyboard dont work ☐

Server

Connection ☐ || I don't have access to the server ☐ || Missing files ✓

Printer

Connection error ☐ || Low on ink ☐ || The printer don't work ☐

Internet

I cannot connect to the internet ☐ || I cannot connect to the wireless internet ☐

Group rooms

Lost my key ✓ || Need a key for my group room ☐ || I need a group room ☐

Outside

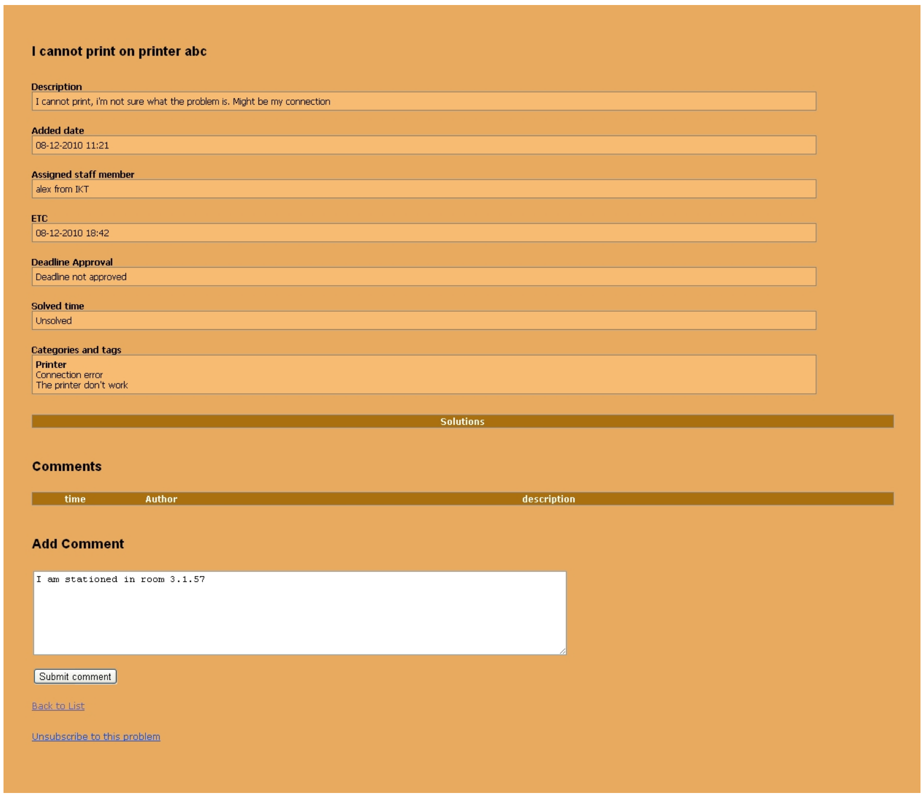
Snow ☐ || Outside light ☐

Figure 10.3: *The create step of committing a new problem, note that the page has been cropped*

the bound for similarity at each step. In order to make the search run faster it then needs a number which indicates if the size of the result is satisfactory. The search function is described more thoroughly in section 12.2.

The result field displays the problems found by the search function. This list is initially empty, because the search button must be clicked in run a search. The way the problems are displayed resembles the table in the Consider Existing Problems step in section 10.1.1 except that it does not have a column with the number of matching tags but instead has a column with the time it was solved – if the given problem is solved. The problems in the list can be clicked to enter the details of the given problem, this detail view is described in the following sub-subsection.

Problem Details



I cannot print on printer abc

Description: I cannot print, i'm not sure what the problem is. Might be my connection

Added date: 08-12-2010 11:21

Assigned staff member: alex from IKT

ETC: 08-12-2010 18:42

Deadline Approval: Deadline not approved

Solved time: Unsolved

Categories and tags: Printer / Connection error / The printer don't work

Solutions

Comments

time	Author	description

Add Comment

I am stationed in room 3.1.57

Figure 10.4: *The details of a problem*

The properties of a problem can be seen in the problem details view in figure 10.4. This view also allows the client to add comments to the problem and subscribe or unsubscribe to the problem. If a client is subscribing to a problem, he/she will receive an e-mail every time a solution is added to the problem or when a comment is added.

The details which can be seen in the problem details are: The title, description, added date, assigned to staff member, ETC, deadline approval, solved time, categories and tags, and if the deadline is approved; the actual deadline. If there is any solutions attached to the problem, these are shown in this view. The same applies to comments.

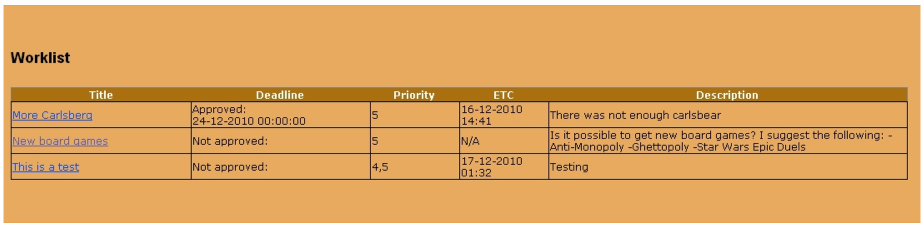
10.1.3 See Own Problems

For a client to see his/her problems, he/she has the My Problems menu point, which can be seen in figure 10.1. This actually initiates the search for problems process as described in subsection 10.1.2. The difference is the headline and the fact that the only my problems and only unsolved problems check boxes are checked. Also a search is initiated when the menu point is selected. It is still possible to specify your search further and change the specification of the search, i.e. it works the same way as the search usage described in subsection 10.1.2 from this point on.

10.2 Staff Usage

The staff members main usage of the is to solve problems. This is corresponding to the solve problem use case in section 4.1.2. This use case is how ever quite large, so it is divided into three steps; choosing a problem to solve, communicate with the subscriber(s), solve the problem. The communication with the subscriber(s) step might not be applied during the solve usage of the client. The three steps are described in subsections 10.2.1, 10.2.2, and 10.2.3.

10.2.1 Choosing a Problem to Solve

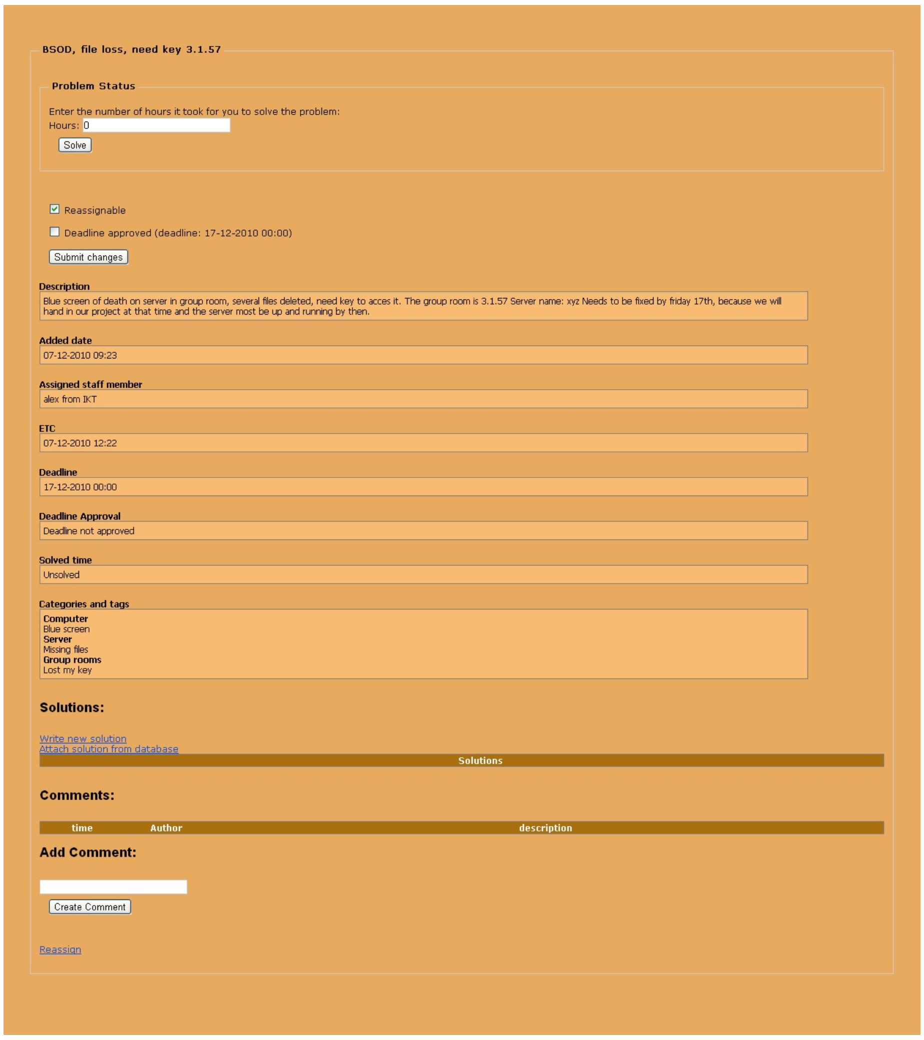

Worklist

Title	Deadline	Priority	ETC	Description
More Carlsberg	Approved: 24-12-2010 00:00:00	5	16-12-2010 14:41	There was not enough carlsbear
New board games	Not approved:	5	N/A	Is it possible to get new board games? I suggest the following: -Anti-Monopoly -Ghettopoly -Star Wars Epic Duels
This is a test	Not approved:	4,5	17-12-2010 01:32	Testing

Figure 10.5: *A staff members worklist*

When a staff member wants to solve a problem he/she clicks the menu point called My Worklist which can be seen in figure 10.1. This shows the worklist of current logged in staff user. An example of a worklist is shown in figure 10.5. The staff member can see the title, description, deadline, priority, and ETC. He/she can click on a problem in order to get a more detailed view of that specific problem. The staff members are responsible for choosing the right problems in the right order them selves, but they can use the priority as a guideline for which problems should be solved first. This is made easier by the application because the list of problems is ordered by the priority of the problems.

It is from problem details view that the staff member can communicate with the subscribers, accept the deadline, lock the problem to him/her self, attach


BSOD, file loss, need key 3.1.57

Problem Status

Enter the number of hours it took for you to solve the problem:
Hours: 0

☑ Reassignable

☐ Deadline approved (deadline: 17-12-2010 00:00)

Description
Blue screen of death on server in group room, several files deleted, need key to acces it. The group room is 3.1.57 Server name: xyz Needs to be fixed by friday 17th, because we will hand in our project at that time and the server most be up and running by then.

Added date
07-12-2010 09:23

Assigned staff member
alex from IKT

ETC
07-12-2010 12:22

Deadline
17-12-2010 00:00

Deadline Approval
Deadline not approved

Solved time
Unsolved

Categories and tags
Computer
Blue screen
Server
Missing files
Group rooms
Lost my key

Solutions:

Comments:

time	Author	description

Add Comment:

Figure 10.6: *A staff members problem details view*

solutions, and finally declare the problem solved. Even though this view share name with the problem details that a client can access, the staff members do have more options, as described above. The staff members problem details view can be seen in figure 10.6. To accept the deadline a staff member simply checks a check box and submit the change. When a staff member has chosen problem to start working on, he/she can make it non-reassignable by unchecking the reassignable check box. This will stop the balance workload function, which is described in section 12.3, from removing that given problem from the staff member while he/she is working on it. If the staff member does not do this, the problem might be reassigned to another staff member and thereby reducing the effectiveness of the recourses because to staff members will be work on the same problem.

The staff members can reassign problems which have been assigned to them. When doing so, he/she can either reassign directly to another staff member or to a department. In either case the problem will be marked non-reassignable, in order to stop the balance workload function from moving it to another staff member or back to the staff member who had it in the first place.

10.2.2 Communicate With Subscribers

To communicate with the subscribers, the staff member can write comments to a problem, which the subscribers can see and again respond to. This communication is important because the staff member might need more details about a problem in order to solve it.

10.2.3 Solve the Problem

With the staff members problem details is it possible to attach an existing solution or write a new one. Once a problem has been given a solution the subscribers are notified and they can check if the solution actually solved their problem. They can then write a comment back telling the assigned staff member whether or not the solution solved the problem for them. If the subscribers are content with the solution(s) the staff member can mark the problem as solved and enter the time he spend on the problem, so that future problems similar to the given can be given an estimation.

If a staff member wants to use an already existing solution he will have to search the database for it. Here is used a search view very similar to the one which the clients have, which is described in subsection 10.1.2. The difference is that the staff does not have the opportunity to choose any of the following options: Only my problems, only solved problems, only unsolved problems, and minimum number of problems. The reason for this is to make it simpler and faster to find the solution. When the staff member finds a problem with the right solution, he/she can attach this to the problem currently being solved.

To write a new solution the staff member clicks the corresponding link in the staffs problem details view. He/she is then presented with a text area and a create button. When the button is clicked, a new solution is created with the given text is attached to the problem being solved. The problem is not marked as solved as a solution is attached to it, because the solution might not actually have solved the problem or the staff member might want to attach more than one solution before he/she wants to mark it solved.

10.3 Admin Usage

The use cases administrate and statistics, which are described in section 4.1.2, are the two main usages of the application which the admin will use. The administrate use case is divided into two usages; manage people and manage departments. These three usages are described in subsections 10.3.1, 10.3.2, and 10.3.3.

10.3.1 Manage People

For an admin to manage the people in the application – or rather the people's users – he/she can click the menu point called Manage People which can be seen in figure 10.1. The view which the admin is presented with is a table of all the users in the application. The table includes: Actions on the user, id number, e-mail, department name, workload, and the roles of the user. The department name and workload is only available for staff users, because they are the only ones with a department and a workload. Other users simply gets "N/A" in these cells of the table. The actions on the users are: Edit, delete, reset password. This is done by clicking the corresponding link next to the user which the action is to be invoked on.

The edit link takes the admin to the edit view of the corresponding user. From here he/she can see the id and user name of the user, and can edit the e-mail, department, and the roles of the user. The department can only be edited if the user is a staff member. If a non-staff user is given the staff role, the admin is asked to give the user a department, because we do not want to staff members without a department.

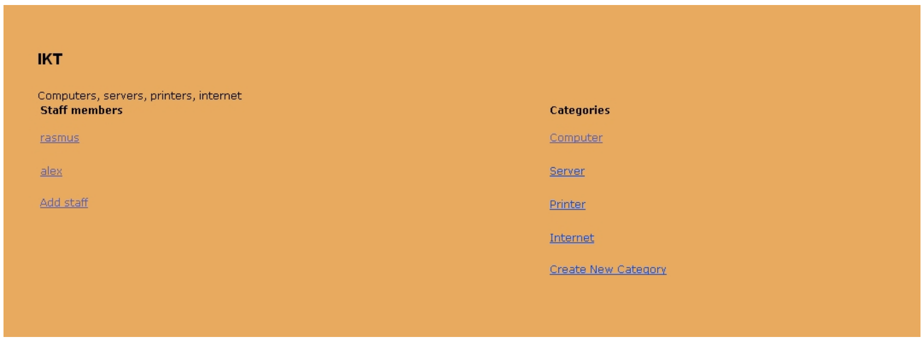
The deletion of a user will result in permanent removal of the user from the application. This can only happen if the user is not subscribed to any problems and is not assigned to any problems.

The reset password action initiates a random password generator to set as the new password for the given user. After that an e-mail is sent to the user containing the newly generated password.

10.3.2 Manage Departments

The menu point Manage Departments which is seen in figure 10.1 allows the admins to administrate the departments, categories, and tags in the application. The view shown when this menu point is clicked, the admin is presented with a list of every department in the application along with a link to create a new department.

When creating a new department, the admin creating it must provide the name of the department and a description. A newly created department is initially empty, i.e. it contains no categories and no staff members are part of it. To add categories and staff members an admin can select the department in the list of the Manage Department view.



IKT

Computers, servers, printers, internet

Staff members

rasmus

alex

Add staff

Categories

Computer

Server

Printer

Internet

Create New Category

Figure 10.7: *The department view, note that the screen shot has been cropped*

When a department is selected, the admin is presented with a view similar to figure 10.7. The view contains two lists; one with the staff members associated with the department and one with the categories associated with the department. Links to add a new category and to attach staff member to the department are also provided in this view. These two parts of the department view are described in the following sub-subsections.

Manage Staff Members

When a staff member is selected, the admin is redirected to the person editor of that staff member, which is described in subsection 10.3.1. To add a staff member to the selected department, the admin can click the Add staff link in the department view. This renders a view with a table of every person in the application who has the staff role. The table resembles the table described in subsection 10.3.1 except that the id for the staff members are not shown.

Manage Categories

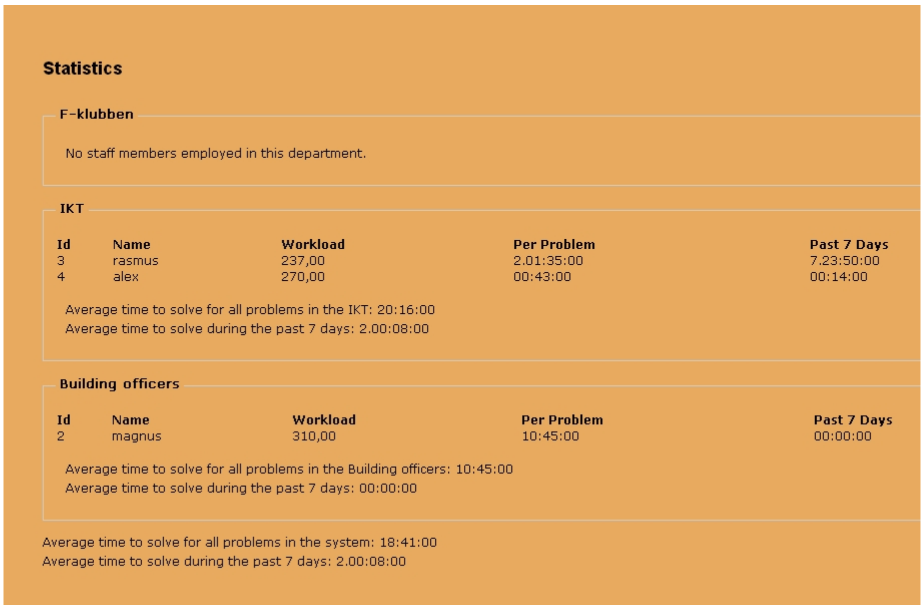
To create a category, an admin clicks on the corresponding link and provides a name and description for the category. Like a newly created department, the new category is empty, thus containing no tags. To change the new category – or another one – an admin can click on it in the department view for the department which the category belongs to. The view which is then presented shows the id, name, description, department, and whether or not the category is hidden. Furthermore the category can be deleted, hidden, and edited from this view. Hiding a category will hide all the tags associated to that category. If the category is hidden it can be unhidden(revealed), which will unhide all the tags associated to the category. To delete a category, every tag must be removed beforehand. It is possible to create new tags from the category view, these tags will initially be attach to the category from which they were created.

During creation of a tag, the admin must provide a name for tag, a description, and a priority. The priority of a tag is a number which indicates how important a problem with the tag is relatively to problems with other tags. There is no inbuilt scale, if the organization using out application wants their tags to vary in priority from 0 to 10 they can do that, the only constraint is that the priority is a 16 bit integer, resulting in a range from 2^{15} to $2^{15} - 1$.

In the category view, the tags can also be individually be hidden, edited, viewed in details, and deleted. Hiding a tag means that it will not be shown in e.g. the problem search view of the clients. When editing a tag, the admin can set the same properties of the tag as when a tag is created. The detailed view of a tag shows the properties of the given tag, namely: Id, name, description, priority, category id, number of solved problems, time consumed, average time spend, and whether or not it is hidden. The time consumed indicates the time, in minutes, for how long every problem with this tag has taken to solve. It is used along with number of problems solved to calculate the average time spend to solve problems with that tag. This is again used to calculate the estimated time of completion of unsolved problems.

To delete a tag, no problem can be associated with it. The deletion of tags is only supposed to be used if a tag is created incorrectly, e.g. another tag already exists or it is created in the wrong department.

10.3.3 Statistics



Statistics

F-klubben

No staff members employed in this department.

IKT

Id	Name	Workload	Per Problem	Past 7 Days
3	rasmus	237,00	2.01:35:00	7.23:50:00
4	alex	270,00	00:43:00	00:14:00

Average time to solve for all problems in the IKT: 20:16:00
Average time to solve during the past 7 days: 2.00:08:00

Building officers

Id	Name	Workload	Per Problem	Past 7 Days
2	magnus	310,00	10:45:00	00:00:00

Average time to solve for all problems in the Building officers: 10:45:00
Average time to solve during the past 7 days: 00:00:00

Average time to solve for all problems in the system: 18:41:00
Average time to solve during the past 7 days: 2.00:08:00

Figure 10.8: *The statistics view, note that the screen shot has been cropped*

An admin can get statistics about how long problems in average remain unsolved. To do this he/she must select the Statics menu point as seen in figure 10.1. He/she is then presented with a view which shows the average time for a problem to remain unsolved for each particular staff member, each department, and for the application over all. It also shows the average time for problems to be solved for the last week for all of the three groups above. This is seen in figure 10.8.

These statistics can be used in any way the admin wants, e.g. to see if a particular department needs more staff members if that given department is generally slower than the others. It is only admins who are allowed to see this page because clients might use the statistics to categorize their problem, so that it will end up in the department which is generally fastest to solve problems, but this will actually make the problem be solved slower if the staff member has to figure out where to reassign it. The staff members can neither see this page in order to avoid internal competition for fastest solve time, because it could lead to a lack in quality of the solutions.

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

11 Database

In this chapter the database and the creation of the model is presented. This is important since it is a fundamental part of the project.

We distinguish between two parts of the database, the login part, and the model part. The model part is created from our Model 8.1.1 using the ADO.NET Entity Framework(EF) which is described in subsection 6.2.5. We choose to use ADO.NET EF because then 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 built in 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. The major disadvantage is that including the membership providers database scheme with the model is not supported and does not work properly, 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 need to access data from the membership tables we have 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.

11.1 Model Designer

To administrate the model, we use the Visual Studio inbuilt tool, ADO.NET Entity Model Designer. This tool can create a database from the model, but also generate the model from the database. We generate our database from the model. If any changes is made it is easy to update both the model and the database. Our model as it is seen in the ADO.NET Entity Model Designer is shown on figure 11.1.

The lines between the entities represents relations. If it is a many to many relation, the needed relationship table is created behind the scenes. This means the created database actually contains several tables more than entities. To use the relations the navigation properties is used.

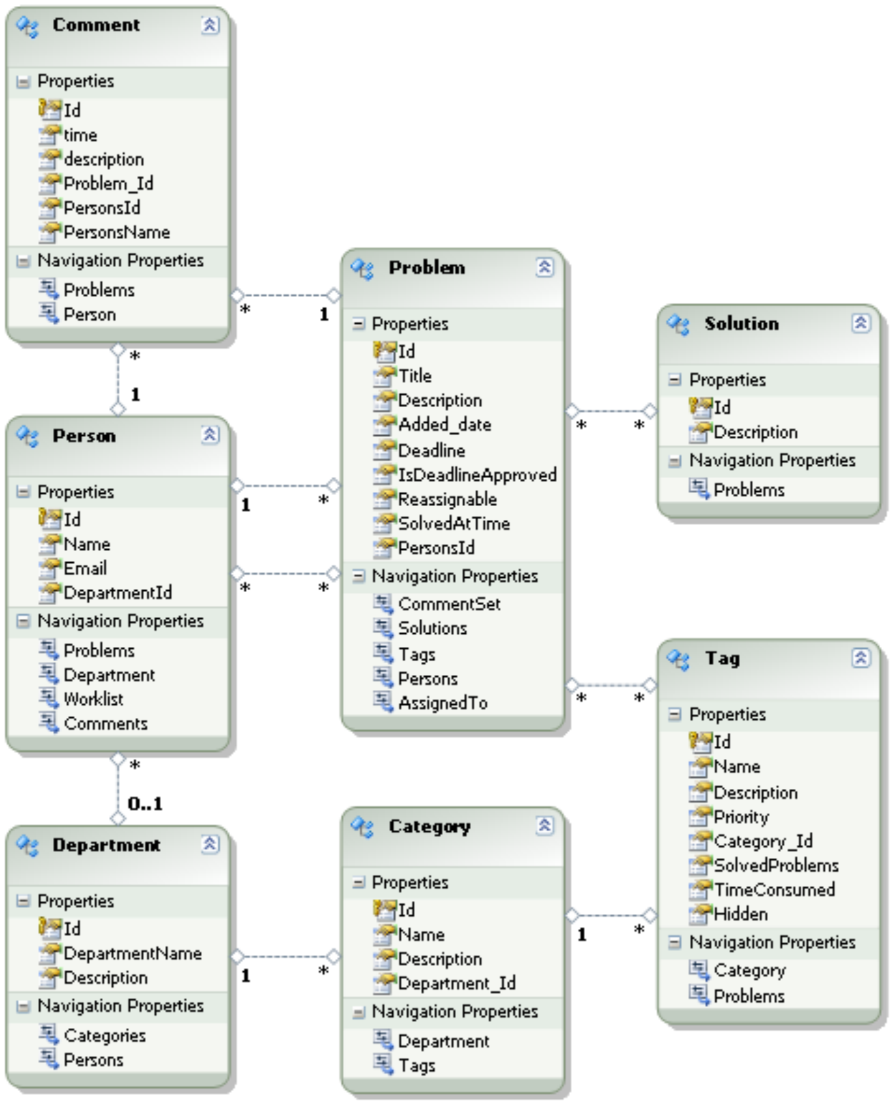


Figure 11.1: *Our model as it is seen in the ADO.NET Entity Data Model Designer*

This chapter presents the database, the model and their creation.

12 Key Points

This chapter describes the key points of our application. The key points which are chosen to be explained in this chapter are central functionalities of the Hopla Helpdesk. The most complex key points are described thoroughly, while the simpler key points are described more briefly.

12.1 Problem Prioritization

The priority of a problem is used in the staffs worklist, where all problems are sorted by priority, however all problems with deadlines are shown first on the list. The division of the worklist is made to create a better overview. If a problem with an approved deadline is overdue, the priority will go up to the maximum priority, which in turn will make the problem appear on the top of the list.

The worklist should be sorted because some problems are more important then others and should be solved prior to lesser important problems.

When a staff member is assigned to a problem, he should read the new problem and approve the problems deadline if it is reasonable.

An example of a worklist can be seen in figure 10.5.

12.2 Problem Search

Our application needs to be able to search for problems. The method for this specific purpose is simply called *Search* and is 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 12.2.1 and 12.2.2. Subsection 12.2.3 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.

12.2.1 Search for Problems by Tags

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

Code snippet 12.1: *The while loop which finds and sorts problems matching the input tags*

The most important part of the *InternalSearch* function is the while loop shown in code snippet 12.1. Generally, this loop finds problems which match 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 2 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 12.2.2.

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 2-30. 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. Furthermore the function sorts the problems each step by the least amount of tags. Because the less tags a problem have 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.

The inner while loop spanning the lines 13-22 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 21 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 13-22. 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 finding enough problems. Furthermore 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 when filtered only yields a single problem and thereby rendering the minimum number of problems nearly useless.

The for-each loop in the lines 16-19 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 6-9. 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.

12.2.2 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, and 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 examine every problem, this means that it will actually return every problem in the "Problems to search among" sorted.

12.2.3 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* method makes use of the *InternalSearch* method 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.

12.3 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 that 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 that each problem on his worklist estimated 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 attached to the problem. This is calculated by the *CalculateTimeConsumption* method which is described in section 12.5.

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 problem with the lowest estimated time consumption from the maximum staff member to the minimum. It keeps reassigning problems until the minimum staff member has a greater or equal workload compared to the staff member with the maximum workload. If the minimum staff member has a higher workload, then the algorithm checks if the workload balance between the two staff members 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 45 in code snippet 12.2.

```
...
bool couldStillMove = true;
do
{
    // Finde the reassignable problem with the highest priority
        which has not been moved yet.
    var problemToBeMoved = maxWorklist.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 12.2: *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 is currently moving problems between. **maxWorklist** is a sorted worklist, which is sorted before the code snippet is called. It is sorted in non-deacreasing order according to the estimated time consumption.*

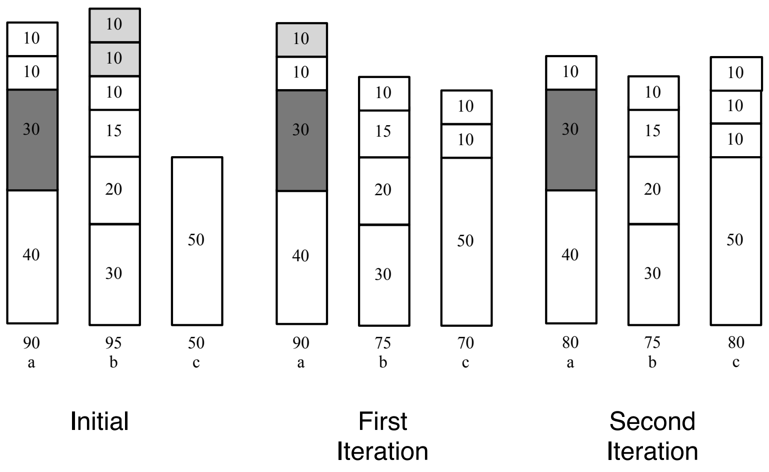


Figure 12.1: *A diagram of the balance workload method. Each collumn represents a staff members workload. Each box is a problem. The height represents the estimated time consumption. The problem colored dark grey is not reassignable. There are tree staff members a, b, and c. The problems that will be moved is colored light grey.*

All this is iterated once per staff member minus one in the department. E.g. if there are two staff members it is run once, if there is three it runs twice etc. The source code is displayed in code snippet 12.2.

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

An example of the algorithm is shown on figure 12.1.

12.4 Tag Time Management

```
public void ManageTagTimes(double StaffTimeSpentInput)
{
    int MinutesUsed = (int)(StaffTimeSpentInput*60);

    foreach (var tag in Tags)
  {
        if (tag.SolvedProblems == null)
        tag.SolvedProblems = 0;

        tag.SolvedProblems++;

        if (tag.TimeConsumed == null)
            tag.TimeConsumed = 0;

        tag.TimeConsumed = tag.TimeConsumed + MinutesUsed;
    }
}
```

Code snippet 12.3: *The ManageTagTimes method*

When a problem is solved the staff member who solved it sets an estimate of how long he/she used to solve the problem, this estimate is saved in the tags of the problem and later used to calculate statistics of problems with these tags. The *ManageTagTimes* method updates the properties **SolvedProblems** and **TimeConsumed** of the class **Tag**. By incrementing the **SolvedProblems** and **TimeConsumed** properties of each tag respectively by one and the time used for the problem to be solved. The method is shown in code snippet 12.3.

12.5 Estimated Time Consumption

```
 1 private TimeSpan CalculateEstimatedTimeConsumption()
 2 {
 3     int NumberOfTags = 0;
 4     decimal? ProblemTime = 0;
 5     int Minutes = 0, Hours = 0, Days = 0;
 6     decimal? average = 0;
 7
 8     foreach (Tag tag in Tags)
 9     {
10         if (tag.AverageTimeSpent != null)
11         {
12             ProblemTime = ProblemTime + tag.AverageTimeSpent;
13         }
14         NumberOfTags++;
15     }
16
17     if (NumberOfTags == 0)
18     {
19         average = Tools.AverageAllTags.averageAll;
20     }
21     else
22     {
23         average = ProblemTime / NumberOfTags;
24     }
25
26     Hours = (int)average % 60;
27     Minutes = (int)average - (Hours*60);
28     Days = Hours % 24;
29     Hours = Hours - (Days * 24);
30
31     return new TimeSpan(Days, Hours, Minutes, 0);
32 }
```

Code snippet 12.4: *The ManageTagTimes method. In lines one to seven a property is shown, which wraps the method in lines nine to forty-three.*

The *CalculateEstimatedTimeConsumption* function – which is a method in the **Problem** class – estimates how much time is needed to solve the specific problem, under the assumption that there only exists this single problem on the assigned staff members queue. The idea is to sum up all the **AverageTimeSpent** property as well as all the count the number of tags, and lastly calculate the average time for all the tags. See code snippet 12.4. The *CalculateEstimatedTimeConsumption* function uses functionality from the **AverageAllTags** class which can be seen in code snippet 12.5.

The **AverageAllTags** class is used if no tags are attached to the problem which the *CalculateEstimatedTimeConsumption* method is on. The property **averageAll** the average time among all tags. It is belonging to the **AverageAllTags** class.

```
public static class AverageAllTags
{
    static hoplaEntities db = new hoplaEntities();
    public static decimal averageAll
    {
        get
        {
            int NumberOfTags = 0;
            decimal? ProblemTime = 0;

            foreach (Tag tag in db.TagSet.ToList())
            {
                if (tag.AverageTimeSpent != null)
                {
                    ProblemTime = ProblemTime + tag.
                        AverageTimeSpent;
                }
                NumberOfTags++;

            }
            return (decimal)(ProblemTime / NumberOfTags);
        }
    }
}
```

Code snippet 12.5: *The **AverageAllTags** class, which is stored in the Tools folder.*

12.6 Expected Time Of Completion

```
private DateTime CalculateETA()
{
    DateTime DateTime = DateTime.Now;
    foreach (Problem problem in AssignedTo.SortedWorklist)
    {
        DateTime = DateTime.Add(EstimatedÂ¨TimeConsumption);
        if (problem.Id == Id)
            break;
    }
    return DateTime;
}
```

Code snippet 12.6: *The CalculateETA method*

The *CalculateETA* method – which is a method in **Problem** class – estimates time when a specific problem will be solved, based on all the problems in the staff members worklist. The method is shown in code snippet 12.6

12.7 Get Sorted List

The *GetSortedList* method – shown in code snippet 12.7 – purpose is to sort the staff members worklist as defined in section 12.1.

```
private List<Problem> GetSortedList()
{
    List<Problem> problemList = Worklist.ToList().Where(
                        x =>
                        x.SolvedAtTime == null &&
                        x.IsDeadlineApproved == true).ToList();

    List<Problem> problemWithoutDeadline = Worklist.ToList().Where(
                        x =>
                        x.SolvedAtTime == null && (
                        x.IsDeadlineApproved == false ||
                        x.IsDeadlineApproved == null)).ToList();


    problemList.Sort(Problem.GetComparer());

    problemWithoutDeadline.Sort(Problem.GetComparer());

    problemList.AddRange(problemWithoutDeadline);

    return problemList;
}
```

Code snippet 12.7: *The GetSortedList method*

12.8 Reset Person Password

The purpose of the *PassMail* method – seen in code snippet 12.8 which is part of the **PersonController** controller – is to reset a user's password to a new random value, and e-mail it to the user. Although not a key feature, the *PassMail* method implements the functionality which sends a user an e-mail, which we consider a key feature since notifying a user is important. The *PassMail* method utilizes the *ResetPassword* method – seen in code snippet 12.9 – which actually generates the new passwords, and updates the database.

```
[Authorize(Roles = HoplaHelpdesk.Models.Constants.AdminRoleName)]
public ActionResult PassMail(int id)
{
    //Getting name from person user
    var person = db.PersonSet.FirstOrDefault(x => x.Id == id);
    String user = person.Name.ToString();
    try
    {
        string msg = HttpUtility.HtmlEncode("Person.PassMail, User
            = " + user);

        MailMessage mail = new MailMessage();
        SmtpClient SmtpServer = new SmtpClient("smtp.gmail.com");
        mail.From = new MailAddress("helps305a@gmail.com");
        mail.To.Add(SQLf.GetEmail(user));
        mail.Subject = "Hopla Helpdesk: Your password has been
            changed!";
        mail.Body = "Username: " + user + "\nPassword: " + SQLf.
            ResetPassword(user) + "\n\nDo not reply to this email.\
            n--\nKind Regards\nHopla Helpdesk - Staff Team";
        SmtpServer.Port = 587;
        SmtpServer.Credentials = new System.Net.NetworkCredential("
            helps305a", "trekant01");
        SmtpServer.EnableSsl = true;
        SmtpServer.Send(mail);

        //Adding view if mail was success
        ViewData["Success"] = "The users password was reset to a
            random value and emailed to the user.";
        ViewData["View"] = "Index";
        return View("Success");
    }
    catch
    {
        //An error view will be added if an error occurred
        ViewData["Error"] = "The password cannot be resetted, " +
            user + " don't have a mail";
        ViewData["View"] = "Index";
        return View("Error");
    }
}
```

Code snippet 12.8: *The PassMail method*

```
public static String ResetPassword(String user)
{
        //Define array for new password
    String[] passarray =
    {
        "a","b","c","d","e","f","g","h","j", "i" ,"k",
        "l","m","n","o","p","q","r","s","t","u",
        "w","x","y","z","A","B","C","D","E","F",
        "G","H","I","J","K","L","M","N","O","P","Q",
        "R","S","T","U","V","W","X","Y","Z",
        "0","1","2","3","4","5","6","7","8","9"
    };

    String setPass = "";
    //Preparing two randoms, one for password length and one for
        choosing a char from passarray.
    Random RandomNumber = new Random();
    Random RandomPass = new Random();
    int x = RandomNumber.Next(10,25);
    //Putting the passowrd together
    for (int i = 0; i < x; i++)
    {
        int y = RandomPass.Next(passarray.Length);
        setPass += passarray[y].ToString();
    }
    //Finding the user
    MembershipUser u = Membership.GetUser(user);
    //Generating a temporary password for the user
    String np = u.ResetPassword();
    //Changing the password for the user by using the the temporary
         password as old password
    u.ChangePassword(np, setPass);
    return setPass;
}
```

Code snippet 12.9: *The ResetPassword(String user) method*

12.9 Get Statistics

```
public ActionResult Index()
{
    var departments = db.DepartmentSet.ToList();
    List<Problem> problems = new List<Problem>();
    List<Problem> problemsPastWeek = new List<Problem>();
    var now = DateTime.Now;
    var since = now.Subtract(new TimeSpan(7, 0, 0, 0));
    foreach(var dep in departments) {
        foreach (var person in dep.Persons) {
            problemsPastWeek.AddRange(person.Worklist.Where(x => x.
                SolvedAtTime > since));
            problems.AddRange(person.Worklist.Where(x => x.
                SolvedAtTime != null));
        }
    }
    var viewModel = new StatisticViewModel()
    {
        AverageLastWeek = StatTool.AveragePerProblem(
            problemsPastWeek),
        AverageAllTime = StatTool.AveragePerProblem(problems),
        Departments = departments
     };
    return View(viewModel);
}
```

Code snippet 12.10: *The StatisticsController controller*

```
public static TimeSpan AveragePerProblem(IEnumerable<Problem>
    problems)
{
    if (problems == null || problems.Count() == 0)
        return new TimeSpan();
        int totalTime = 0;
        int problemsCount = 0;
        foreach(var problem in problems){
            problemsCount++;
            totalTime = totalTime + (int)((TimeSpan)(problem.
                SolvedAtTime - problem.Added_date)).TotalMinutes;
        }
    return new TimeSpan(0,totalTime / problemsCount,0);
    }
}
```

Code snippet 12.11: *The AveragePerProblem method*

The **StatisticsController** controller – seen in code snippet 12.10 – calculates the statistics, by utilizing the *AveragePerProblem* method – seen in code snippet 12.11. A screenshot showing how the statistics are used can be seen in figure 10.8.

12.10 Distribute Problems

Whenever a new problem is added it needs to be distributed to the staff member in the correct department with the lowest workload. The correct department is found by looking through all the tags of the problem and choosing the department that the most tags belong to.

There are two special cases which requires some attention. The case where two or more departments are equally represented in a problem and the case where a problem has no tags. In the first case a department is simply chosen by the order in which the department is in the unsorted list of possible departments. In the second case the staff member with the lowest workload will get the problem, no matter what department he/she belongs to. If a problem is assigned to an inappropriate department we assume that any staff member would have the acquired knowledge to reassign the problem to the appropriate department.

```
public static IPerson GetStaff(Problem Problem, List<Person>
    PersonSet)
{
    var persons = new List<IPerson>();
    foreach (var item in PersonSet)
    {
        persons.Add(item);
    }
    return GetStaff(Problem, persons, GetDepartment(Problem.Tags));
}

public static IPerson GetStaff(Problem Problem, List<IPerson>
    PersonSet)
{
    return GetStaff(Problem, PersonSet, GetDepartment(Problem.Tags)
        );
}

public static IPerson GetStaff(Problem Problem, List<Person>
    PersonSet, Department department)
{
    var persons = new List<IPerson>();
    foreach (var item in PersonSet)
    {
        persons.Add(item);
    }
    return GetStaff(Problem, persons, department);
}

public static IPerson GetStaff(Problem Problem, Department
    department)
{
    return GetStaff(Problem, department.Persons.ToList(),
        department);
}
```

Code snippet 12.12: *The overloads for the GetStaff() method*

The method that does the actual work is the *GetStaff* method which has several overloads. The overloads can be seen in code snippet 12.12. The actual method can be seen in code snippet 12.13. All overloads takes a **Problem** as parameter. The other takes either a department or a list of **IPerson**, which is a **Person** interface made to make it more testable. Some overloads use both parameters.

```
public static IPerson GetStaff(Problem Problem, List<IPerson>
    PersonSet, Department department)
{
    IEnumerable<IPerson> persons = null;
    if (department != null)
    {
         persons = PersonSet.Where(x => x.Department == department
             && x.IsStaff() == true);
         if (persons == null || persons.Count() == 0)
         {
             persons = PersonSet.Where(x => x.IsStaff() == true);
         }
    } else {
         persons = PersonSet.Where(x => x.IsStaff() == true);
    }
    Double min = Double.MaxValue;
    if (persons.Count() == 0 || persons == null)
    {
        throw new ArgumentNullException("The Person List were empty
            ");
    }
    IPerson staff = persons.First();
    foreach (var person in persons)
    {
            var workload = person.GetWorkload();
            if (workload < min)
            {
                min = workload;
                staff = person;
            }
    }
    if (persons == null)
    {
        staff = persons.First();
    }

    return staff;
}
```

Code snippet 12.13: *the GetStaff() method*

The key points presented in this chapter are all central functions to our application.

Part IV

Testing

13 Test Methods

The different test methods which we have used to test our application are described in this chapter. It is important to understand the different methods in order to understand how our tests works and the reasoning for the different test cases.

13.1 Black and White Box Testing

There is a definite difference between the two concepts black- and white box testing; White box testing requires knowledge of the code being tested whereas black box testing requires no knowledge of the code being executed what so ever. White box tests are generally conducted by the developers because they have the needed knowledge, whereas black box testing is often conducted by others because they have no idea how the program works and are therefore better able to determine whether or not a program provides the expected output. We mainly use white box testing because we do have knowledge of the code which we are conducting tests of, white box testing is therefore preferred since we can make more accurate test cases through analysis of the code.

Through an analysis of the code we can see how much of a component is actually tested. The size of a tested component compared to the size of the entire component is a measure for how much of the code in the component which is covered by the tests. This is called "code coverage" and can be measured in several ways. These, along with the ones we are using, are described below in subsection 13.1.1.

13.1.1 Code Coverage

Code coverage is a measure of how much of a particular component is covered by the test cases written. [5] There are different units of measure for code coverage, e.g. statement coverage, decision coverage, condition coverage, loop coverage and path coverage.

We will mainly focus on path coverage with loop coverage taken into account. It involves finding every code path the specific component can take, e.g. every if statement should be tested where the condition is both true and false. It is also needed to test the edge conditions for the loops within the code of the component being tested. These edge conditions are zero runs, one run, and more than one run.

This is done by making a flow chart of the component being tested to see which code paths there are. The flow chart can then be used as a graph and then every path from the start of the component to the end can be found using depth first search which will run until every edge is in a path leading from the start to the end vertex. Every path is recorded so that a test case covering every path can be created. [21]

13.2 Unit Testing

Throughout the development we have used the Team Test Feature within Visual Studios, which is an integrated unit-testing framework. [18] The idea behind unit testing is to check an individual method by executing it with appropriate input, and afterwards check that its output corresponds to the expected.

Generally a unit test is classified as a white box test because it should cover each code path in order to make sure that the test actually suffice to test the given unit. [25, p. 39], thus it requires knowledge of the code to create such a test. To make sure that each code path is covered, the method for code coverage described in subsection 13.1.1 is used.

All major functions have been tested with Team Test. This means that we have not made any unit testing on controllers. Throughout the development, we have omitted to write complex methods and functions in the controllers, and instead written them in the model or our tool component. We have therefore deemed it unnecessary to unit test each controller. Instead we have tested them manually by running the program and seen that it behaves as expected, in effect running black box tests of our controllers.

We have run tests for the search and the problem distribution methods, these are described in chapter 15 and 14 respectively, along with other parts of the tool component and the model.

To better explain a unit test we divided it in three parts: an arrange phase, an act phase, and an assert phase. The arrange phase sets up the input and requirements for the test. The act is the run of the method and the assert is the actual testing, e.g. where the result from the method is compared to an expected value.

In chapter 14 a white box unit test of the balance workload method is presented and in section 14.1 a unit test with dependency injection is explained.

13.3 Regression Testing

Once the unit tests have been run and passed, the program is not necessarily done. If a feature is altered (optimized, refactored etc.) later, tests should be run again to ensure that the alteration did not break any functionality. [23] This type of testing is known as regression testing.

Usually only test cases regarding the component being changed are run, since running every test case will usually take a large amount of time. However, we do not have that many test cases, so we run every test case whenever a major component was changed. By doing so we ensure that our program will never regress.

The black and white box principles of testing are explained in this chapter along with a description of unit testing and regression testing.

14 Balance Workload Test

The test cases made for the BalanceWorkload method of the ***Department*** *class are described in this chapter. This chapter is important since it provides an example of a unit test. Code snippets are shown along with the description to make the test cases easier to understand.*

To test our *BalanceWorkload* method we have chosen to use a unit test. This is not the only unit test we have made on this component. This is more an introduction of unit testing than an actual test case. Meaning that it does not test the method properly to ensure full code coverage. It tests if the method behaves as expected with the given input.

The *BalanceWorkload* method balances the workload between all staff members in the department. The method is fully described in section 12.3. To arrange the test a department object is initialized and the required properties are set. This means the **Persons** property is set to a list of **Persons** and each person is assigned a number of problem with tags. The arrangement of a test case can be seen in code snippet 14.1.

Now the expected result needs to be calculated. Mike has three problems and this gives him a total workload of $20 + 10 + 10 = 40$. The workload is calculated by **TimeConsumed** divided by **SolvedProblems** of each tag (See lines one to four). This gives the estimated time consumption of the problem. Add up all problems and the workload is calculated.

John has 1 problem which gives him a total workload of 10. This implies that Mike's workload is overbalanced. The algorithm is expected to reassign the problems to balance the workloads. Since there are four problems three with an estimated time consumption of 10 and one with 20. The most balanced possibility with the given problems is when one has a workload of 30 (the sum of $10 + 20$) and the other 20 (the sum of $10 + 10$).

This can be expressed with a boolean expression which can be tested with the assert method *IsTrue*. This assert can be seen on code snippet 14.2.

It is necessary to test various scenarios, e.g. test cases where problems are solved, not reassignable, problems with extreme estimated time consumption, staff members with no problems, an empty department, various estimate time consumptions, and different priorities. To make a complete white box test an analysis has to be made to ensure that every code path is covered.

```
var tag1 =  new Tag(){ TimeConsumed = 20, SolvedProblems = 1 ,
    Priority = 1  };
var tag2 =  new Tag(){ TimeConsumed = 10, SolvedProblems = 1 ,
    Priority = 2  };
var tag3 =  new Tag(){ TimeConsumed = 10, SolvedProblems = 1 ,
    Priority = 3  };
var tag4 =  new Tag(){ TimeConsumed = 10, SolvedProblems = 1 ,
    Priority = 4  };

var prob1 = new Problem() { Tags = new EntityCollection<Tag> { tag1
     }, Reassignable = true };
var prob2 = new Problem() { Tags = new EntityCollection<Tag> { tag2
     }, Reassignable = true };
var prob3 = new Problem() { Tags = new EntityCollection<Tag> { tag3
     }, Reassignable = true };
var prob4 = new Problem() { Tags = new EntityCollection<Tag> { tag4
     }, Reassignable = true };

var mike = new Person() { Name="mike", Worklist = new
    EntityCollection<Problem>() { prob1, prob2, prob3 } }; //
    Workload = 40
var john = new Person() { Name= "John", Worklist = new
    EntityCollection<Problem>() { prob4 } };               // = 10

Department target = new Department()
{
    Persons = new EntityCollection<Person>()
    {
        mike, john
    }
};
```

Code snippet 14.1: *The arrange phase of the unit test of balance workload*

```
target.BalanceWorkload();
Assert.IsTrue(
        (john.Worklist.Contains(john.Workload == 30 && mike.
            Workload == 20) ||
        (mike.Workload == 30 && john.Workload == 20)
    ));
...
```

Code snippet 14.2: *An example unit test which tests a specific instance of the balanceWorkload method.*

14.1 Dependency Injection

For some methods dependency injection must be made in order to execute a proper test. In the example with *BalanceWorkload*, dependency injection can be made on the workload property of **Person**. This is not made because the workload method has been tested at the time *BalanceWorkload* is tested.

To properly make a dependency injection the method in action has to be programmed to an interface. This makes it easy to swap the implemented class with another class. We do this to test the method *GetStaff* in the **ProblemDistributer** class. The method is described in section 12.10.

GetStaff is designed to be more testable and therefore it acts on the **IPerson** interface. The method depends on the **Workload** property of **Person** and at the time of the unit test for this method the **workload** is not fully implemented and neither is the **IsStaff** property. Therefore we made a **TestPerson** class which implemented the **IPerson** interface. The **TestStaff** implementation of the *IsStaff* method returns true unless the **Name** is "john". The *workload* property just counts the number of items in the **Person** class's **Worklist**.

In this way it is possible to make a unit test work.

14.2 Regression Testing

Regression tests are used every time a significant change is made to balance workload method. This is to ensure that the new change does not ruin anything that already worked. The regression tests are simply all the unit tests of this method, which can be run every time a change is made to the method if necessary.

This chapter shows how the test cases of the BalanceWorkload method is made. Examples from the test cases source code are given in this chapter to give a better understanding of the test cases.

15 Problem Search Test

*In this chapter the test cases regarding the **ProblemSearch** class and its methods: Search and SearchSolvedFirst are described. It is generally the white box code coverage unit tests which are described in this chapter, because those are the most important, since we are focused on the functionality of our application and therefore want as much of our code tested as possible.*

15.1 Unit Test Cases for the Search Function

To test our problem search function, we have made an analysis of the code paths of the function. The flow chart generated from this analysis is seen in figure 15.2. The decisions in the figure are labeled with a letter. This is done because we want to refer to these decisions – which are the conditions for loops to run another iteration – later. Notice that the loop labeled *C* is portrayed as a do-while even though code snippet 12.1 shows that this loop is a while loop. The reason for this is that the loop is a `while(true)`, which breaks when an exception of the type **NotSupportedException** is thrown. This means that the loop will always run at least once, like a do-while loop. This is more thoroughly described in subsection 12.2.1.

To make test cases a set of tags and problems are needed to test upon. These are seen in figure 15.1.

Problem	Tag				Solved
	0	1	2	3	
0	✓	✓			
1	✓	✓			✓
2					
3	✓				
4		✓			✓
5				✓	

Figure 15.1: *The problems and tags which the test cases are based upon*

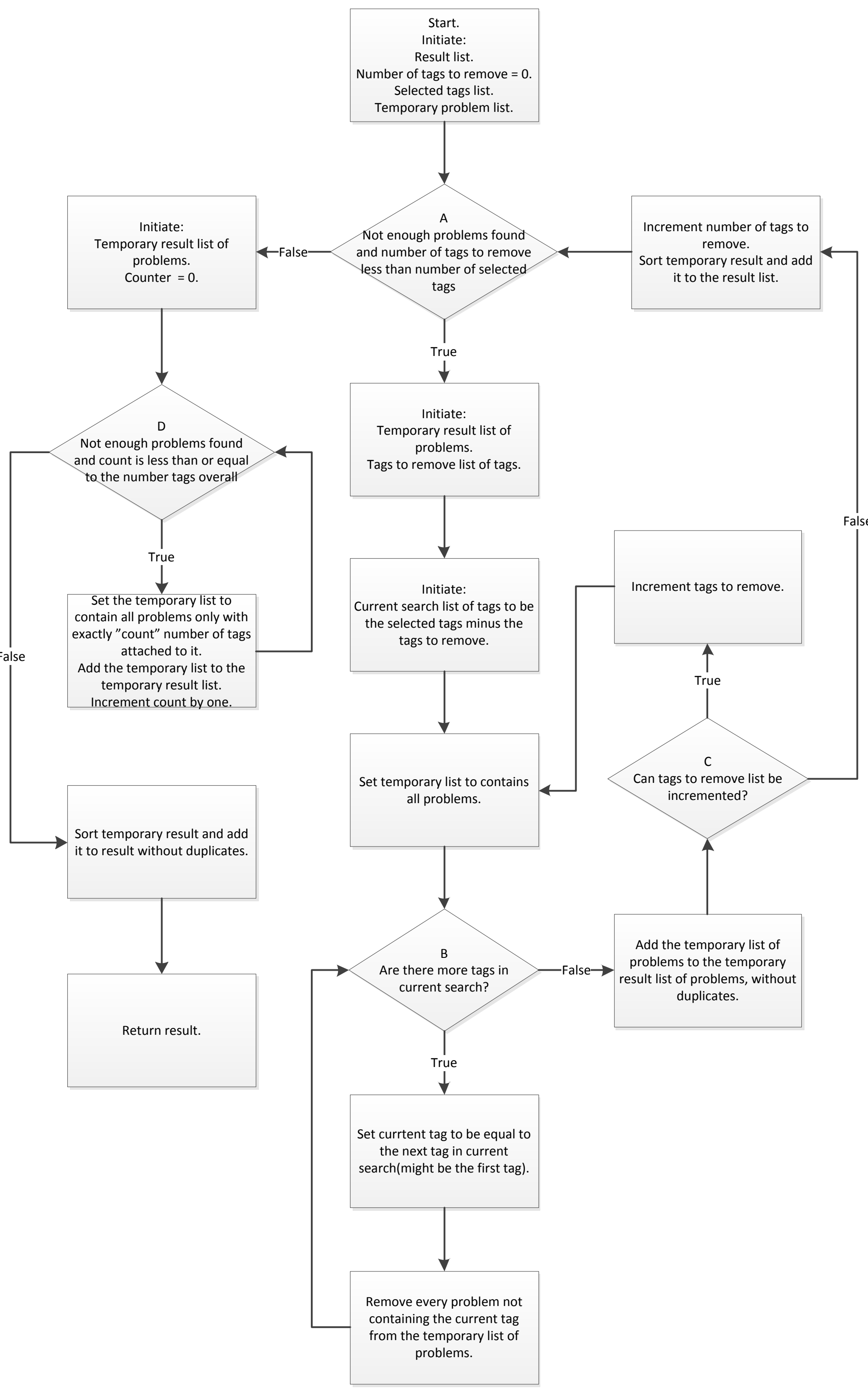

Start.
Initiate:
Result list.
Number of tags to remove = 0.
Selected tags list.
Temporary problem list.
A
Not enough problems found and number of tags to remove less than number of selected tags
False
True
Initiate:
Temporary result list of problems.
Counter = 0.
Increment number of tags to remove.
Sort temporary result and add it to the result list.
Initiate:
Temporary result list of problems.
Tags to remove list of tags.
D
Not enough problems found and count is less than or equal to the number tags overall
True
False
Set the temporary list to contain all problems only with exactly "count" number of tags attached to it.
Add the temporary list to the temporary result list.
Increment count by one.
Initiate:
Current search list of tags to be the selected tags minus the tags to remove.
Increment tags to remove.
False
True
C
Can tags to remove list be incremented?
Set temporary list to contains all problems.
Sort temporary result and add it to result without duplicates.
Return result.
B
Are there more tags in current search?
False
True
Add the temporary list of problems to the temporary result list of problems, without duplicates.
Set currtent tag to be equal to the next tag in current search(might be the first tag).
Remove every problem not containing the current tag from the temporary list of problems.

Figure 15.2: *Flow chart of the search function. The letters in the decisions indicates which loop they are associated with, e.g. the decision labeled A is associated with the loop A*

15.1.1 Test Case One

```
#region Test 1: Search for no tag, minimum number of problems = 0
[TestMethod()]
public void A0_D0_SearchForNoTagNoProblems()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 0;

    expected = new List<Problem>();
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag,
        problems, minNoProb);
    #endregion

    #region Assertions
    Assert.AreEqual(expected.Count, actual.Count);
    #endregion
}
#endregion
```

Code snippet 15.1: *The test case for no run of any loops*

Since the search method only contain loops and no if statements, the focus for the test cases regarding the search is on the loops edge conditions. The first test case will test the option where none of the loops are run. If A is not run, B and C will never be run because they are inside of the A loop. A will not be run if either the size of the result is at least equal to the minimum number of problems to find or if the number of tags to remove is at least equal to the number of selected tags. To not run the A loop, the number of selected tags is set to 0, i.e. no tags are selected. The D loop will not be run if enough problems are found or if the **count** variable is higher than the number of all tags. Because of this the minimum number of problems to find is set to zero. This test case is seen in code snippet 15.1. The **ProblemSearch** is the class which is being tested in the code snippet, particularly the *Search* method in this test case. The variable **catTag** is of the type **CategoryTagSelectionViewModel** and contains categories, which in turn contains the selected tags. No tags are however selected in this test case. The **problems** variable contains the six problems defined in figure 15.1.

15.1.2 Test Case Two

The next test case will test the same as the above test case, except for the fact that the D loop will run exactly once instead of zero times. This time the minimum number of problems to find is set to one, while keeping all other input the same as the previous test. This will result in the A loop not to run and the D loop to run exactly once, because it would then find problem number two, which is the only problem with no tags attached.. As seen on figure 15.1. The source code for this test case is shown in appendix A.

15.1.3 Test Case Three

This test case should not run loop A, but run loop D more than once. This is accomplished by increasing the minimum number of problems to find to 6 while keeping the number of tags to be zero. This test case is shown in appendix A.

15.1.4 Test Cases Four to Six

Now that we have tested all code paths not containing any runs of the A loop, the next test cases will consider the cases where A is run once. To run loop A once, there most be a positive number of tags selected. If the A loop is run then both the B and C will be run at least once, for the following reasons:

- B will be run because the first time the current search is initialized, it will contain all selected tags, which must be at least one, therefore the B decision in figure 15.2 will render true at least once and run the B loop.
- As described earlier the C loop acts as a do-while loop for reasons explained earlier in this section, thereby allowing the C loop to be run at least once.

For test case four the A, B, and C loop will be run one time and the D loop zero times. Test case five will allow D to be run exactly one time, and test case six will let D run several times. All the test cases can be seen in appendix A. To vary the number of times which D is run, the minimum number of problems to find is simply raised while the number tags selected remains one in order for the loops A, B, and C to run only a single iteration each.

15.1.5 Test Cases Seven to Nine

The test cases should run loop A multiple times and change the number of runs the loops B, C, and D runs. As shown above the number of times loops A and D runs can be changed independently. The number of iterations that the number B and C loops will take is however very dependent on the number of times A will run. For A to run more than once, the number of selected tags must be more than one and the number of problems found in the first run of loop A must not exceed the minimum number of problems to find. At the first run of A the B loop will run x times, where x is the number of selected tags. Because the current search during the first iteration of A will contain every selected tag, which is more than one, thereby letting B run more than one time. During the second run of the A loop, the C loop will run x times, where x is the number of selected tags. The reason for this is that the search will be "softened" and only search for problems with all but a specific tag in each iteration, which will render searches in which a different tag is missing each time.

The three test cases which will cover the last code paths are:

- The A, B, and C loops are run several time and the D loop is run zero times.

- The A, B, and C loops are run several time and the D loop is run exactly once.
- The A, B, C, and D loops are run several time.

To ensure that the A, B, and C loops are run more than once, the number of selected tags is set to two and the minimum number of problems to find is set appropriately. The test cases are shown in appendix A.

15.2 Regression Test Cases

The same test cases as described above are used as regression test cases, to ensure that no new change will break any of the already working code . There could be made a new analysis of the code each time a change is made, but this will take a lot of time, and will not necessarily generate better test cases.

This chapter describes the test cases used to test the Search method belonging to the ***ProblemSearch*** *class. Beside the code snippet in this chapter, the appendix A holds the source code for the described unit tests.*

Part V

Epilogue

16 Improvements

This chapter outlines and discusses possible future enhancements, which we for various reasons did not implement in our application.

16.1 BalanceWorkload

The original *BalanceWorkload* method – which is described throughly in section 12.3 – distributes the problems by looking at the staff member which has the highest workload as well as the one with the lowest, and then shifts the reassignable problems until the workload between the two are as equal as possible. This approach does not take the priority nor the deadline into account, which is why we suggest the following new method, which is shown in code snippet – 16.1.

The main idea behind the *FutureImplementationBalanceWorkload* method is that the algorithm only has to balance the workload by some extent as a problem would always be added to the person with the lowest workload every time a problem has been marked as solved or a new problem is submitted, the algorithm will then run again to distribute the workload properly.

This simple approach of distributing problems allows us to choose which problems to distribute first. All we have to do then is sort a temporary list of problems to distribute them in the manner we want. In the *FutureImplementationBalanceWorkload* method, we have chosen to have the problems with an approved deadline at the top of the temporary list, sorted by highest priority at top. Below that all problems without a deadline.

In short, the *FutureImplementationBalanceWorkload* method does the above explained by first unassigning all the reassignable problems from all members of the specific department, followed by sorting them in the fashion explained above, and at last assigns the problems from the top of the temporary list, until the temporary list is empty.

Pros are that problems which are more important will be solved faster.

Cons are that the workload will be a slightly bit more unbalanced between the individual staff members in the department.

We did not implement this feature because we decided early in the beginning only to focus on workload and not priority or deadlines. We discovered too late that this was not a hard issue to solve, however we decided not to implement it as we felt that we did not have the time. Note that this method is still in the application together with a simple testing method as which we did a very limited amount on testing on it to ensure that it works. We would like to state explicitly that the *FutureImplementationBalanceWorkload* method is not used by the application although it is in the source code of it.

```
public void FutureImplementationBalanceWorkload()
{
    Person dummyPerson = new Person();
    List<Person> staffMembers = Persons.ToList();
    List<Problem> problemList = new List<Problem>();

    foreach (var member in staffMembers)
    {
        foreach (var problem in member.Worklist)
        {
            problemList.Add(problem);
        }
    }

    problemList = problemList.Where(x => x.Reassignable == true &&
        x.SolvedAtTime == null).ToList();

    foreach (var problem in problemList)
    {
        problem.AssignedTo = dummyPerson;
    }

    while (problemList.Count > 0)
    {
        // find the person with the lowest workload
        Person min = Persons.FirstOrDefault(y => y.GetWorkload() ==
             Persons.Min(x => x.GetWorkload()));

        // assign the most important problem to the person
        problemList[0].AssignedTo = min;
        problemList.RemoveAt(0);
    }
}
```

Code snippet 16.1: *A possible future implementation of the BalanceWorkload algorithm*

16.2 Staff Calendar

When calculating the expected time of completion our application does not take the actual work day of a staff member into account. Instead it assumes that each staff member is always working, which is obviously incorrect. To fix this problem a calendar for each staff member can be introduced. The calendar should besides his working hours, days off and holidays, contain the deadlines of the staff members' assigned problems. It should be visible to the staff. This will allow the staff to easily see if any deadline is approaching. It will also enable the application to more efficiently distribute problems, since it will take into account when a staff member is off duty.

Pros are that the expected time of completion of solving a problem will be more accurate and the application will more efficiently distribute problems.

Cons are that it requires more administration of the application, as schedules change and employees sometimes are sick or on holiday.

We did not implement this feature because it is not one of the main features of the project.

16.3 Flexible User Roles

The roles we grant users in our application are hard-coded into our application and not the permissions graded by the role. If the Hopla Helpdesk should be implemented at different environments, the role system would need to be more flexible. It is not given that all workplaces make use of the three predefined roles, therefore a more flexible role system would be needed in order to make the application more efficient and make the application more flexible as a whole. It should also be possible to create custom user-roles with custom names. This can be improved by introducing a double layered role system. This work by making a role for each controller or even the methods within the controller, then adding a high level role set which can access the low level roles. This allow for agile changing the privileges for an entire group of users. E.g. if the user of the application wants to allow clients to see statistics this can easily be changed with this improvement. An example of this alternative role system can be seen on figure 16.1.

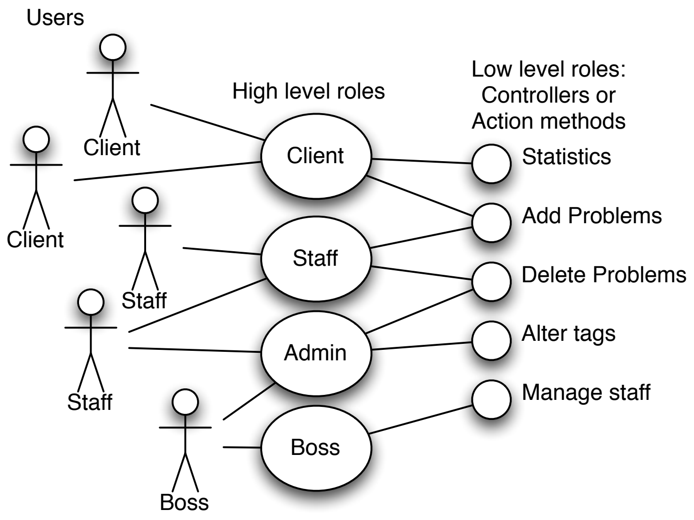


Figure 16.1: *Example of how the roles can be structured with an improved role system*

Pros are that the application allows more specific configuration.

Cons are the time it takes to implement it.

We did not implement this feature because we realized too late that the functionality was missing, thus we did not have time to implement it.

16.4 Text Based Search

If a client has a problem which cannot be described sufficiently by the tags provided by the application/admin, the search for a similar problem could be improved by adding text based search.
This feature can be extended by using it as statistics; If a word is often searched for, a tag with that word might be needed. The admin could have a view allowing him to see often searched keywords and allowing him to create tags with these keywords. This feature would ease the admin's process of creating the categories and tags, and improve the categorization process of adding a new problem and finding similar problems.

Pros are that clients searches would be more successfully as well as relevant tags will be added which otherwise would be omitted.

Cons The implementation time.

We did not implement this feature because it is not one of the main features of the project.

16.5 Dynamic Worklist

Currently, the worklist of staff members appears static in the webbrowser, until the page is refreshed. With interactive web technologies such as AJAX it is possible to remove or add a specific problem from the worklist almost instantly when it is assigned/reassigned. This improvement removes the need of refreshing the page whenever the staff member would like to see if he/her has been assigned or reassigned problems, as well as make it more intuitive that the member should use the "reassignable" feature to "lock" the problem which he/she is currently working on. Comprehensive use of the "reassignable" feature also reduces the amount of organizational problems which might occur when two staff members both work on the same problem, because one of them got the problem reassigned by the workload balancer, while the other was not aware that the problem had been reassigned to another member.

Pros are reduced organizational problems due to misunderstandings.

Cons takes time to implement and increased bandwidth usage.

We did not implement this feature because it is not one of the main features of the project

---

This chapter outlines and discusses possible future enhancements.

17 Evaluation

We used the SAD course to make a thorough analysis and design. We did however design our application prior to choosing the framework on which we would built it – namely the ASP.NET MVC 2 framework. This meant that a lot of the time were initially spent on the design, were completely wasted as the original design could not be implemented properly using the framework. Therefore the implementation was commenced late in the process which resulted in lack of the required time to make a full application test and correct all bugs. This could be avoided by achieving knowledge of the framework prior to the design phase. In general we used a lot of new technologies which we initially had a very sparse knowledge of, however, due to the boundaries of the semester we did not have any other choice. If we were to start over, we will be able to save a great deal of time and start programming sooner, as we now have experience with the OOAD method and how to use it to develop an application. Furthermore we would give the technical platforms more thought as this to some extent slowed down the process as technical issues were encountered.

18 Conclusion

This chapter concludes whether or not our application lives up to our application definition, which is seen in section 2.2.

The front end of our application – which is described in chapter 10 – is web based, and can run in a browser, thereby enables the users of our application, without any installation, to access our application – assuming they have a browser.

Our application contains tags which have priories. These priorities are used to calculate the importance of a problem. The priority of a given problem is found by calculating the average priority of all tags attached to the problem.

The problems each has an estimated completion time, which is – as the name suggests – an estimation on when the problem is solved. Another estimation called estimated time consumption is used to calculate the estimated time of completion. The estimated time consumption is based on the time which the tags that are attached to the given problem takes to solve on average.

The estimated time consumption of a problem is the sum of all problems placed above the problem on the staff worklist.The way the estimated time of completion is calculated is described in further details in section 12.6 and the estimated time consumption is described in section 12.5.

The Hopla Helpdesk saves every problem – it is actually impossible to delete problems through our front end. This allows the application to suggest old problems when a client is about to commit a new problem. The problems suggested is based on the the tags which the client has used to categorized the problem. These problems are found using our *Search* method, which is described in section 12.2. The *Search* method finds problems which contains the same tags as the ones which are used to categorize the new problem. It orders them according the most matching tags.

When a problem is created and solved it is given a time stamp. These time stamps are used to calculate the time it takes for a problem to get solved. The time to solve a problem is the time from when a problem is committed to when it is solved while the solve time is the time the staff member who solves it has spent working on it. The time to solve is used to generate statistics, which the admins can access to see if the resources could be distributes better, e.g. by transferring a staff member from one department to another. How the statistics are generated is described in section 12.9. The solve time – which is different from “The time to solve a problem” – is entered by a staff when he/she declares a problem solved. The solve time is saved on the tags attached to the problem being solved and is used to calculate the estimated time consumption of unsolved problems with these tags.

The tags, categories and departments are all dynamic, i.e. they can be changed, the same applies to the persons, problems, and solution. This means that our model of the problem domain is dynamic.

We thereby conclude that our application lives up to our application definition, but there is room for improvements as chapter 16 suggests.

19 Perspective

The Hopla Helpdesk application is made to ease the cooperation between service employees and the other members at an organization. This chapter shows other ways in which the Hopla Helpdesk could be used if it was modified, and other approaches which could have been taken to make a help desk application.

By using the same approach as we have used to make our help desk we can make a hospital system containing patients and cures. The patients in the hospital applications would be saved as problems in our help desk. The departments, categories, and tags in our application could be hospital wards, symptom categories, and symptoms respectively. When a patient is hospitalized, all his/her symptoms are entered in the system and it will find the ward which he/she should be admitted to. The staff members would be the doctors and surgeons at the hospital. The way the problems in our application are distributed should possibly be changed, in order to avoid having patients change doctor several times. The main idea with this is that the solutions/cures should be saved and be accessible to anyone, to help future patients with symptoms matching a cured patients. This does however violate the confidentiality of the patients [1], because anyone can see their journal. This could be overcome in some way by saving the patients without name and any other identification, and having his/her personal journal alongside the Hopla Helpdesk-hospital application.

Instead of managing problems, a system managing tasks could be made from the same principles. The tags might be directly associated with an employee. An example of the usage could be a group of programmers each responsible for one or more components of the piece of software they are developing. If one of the developers gets an idea for new a component which he/she is not working one, this idea could be committed and the system would then send the idea to the developer(s) currently working on the given component. This is probably not effect in a small group, but might come in handy when some developers do not know who is working on other components.

It might also be used as a bugzilla; an application where all bugs are collected and possible distributed to developers to fix them. The bugs in such an application would then be the counterpart to the problems in our application. The bugs would be committed in the same way as the problems in the Hopla Helpdesk application, where they are categorized, in order to distribute them correctly. The staff members would be the developers of the program which the bugzilla is associated with or perhaps a small part of the developers who are specialized in bug fixing.

The way we have approached the help desk is based on the assumption that every problem – or at least a huge part of them – can be categorized using tags which the admins have made. Another approach could be to let the users add the tags which they think cover their problems. This would make a more flexible application for the clients, but might be more difficult for the distribution method to find the right department for the problems because the clients might use slightly different tags which means the same, e.g. one client writes "Connection Error" as a tag another might write "Connection Failure" to categorize the same or a similar problem.

An even more flexible way might be to avoid using tags and only use a textual description of the problem. To compare problems this text has to be analyzed. Text analysis can be very complex, because different people write differently. The clients are much freer because they can describe their problem in plain text. This will hopefully lead to better described problems, but it might be harder to distribute the problems using this approach.

Another approach to make a help desk is to avoid the staff role altogether and allow everyone to commit and solve problems. We have assumed that the organizations which are to use our application have one or more departments which take care of solving the problems which are committed. The problems should not be distributed to anyone, but rather be collected into a "Free Problems" list. Everyone can access this list and pick a problem to solve. The solved problems and their solutions can be accessed by everyone as well to cross reference new problems to already solved problems if such should arise.

Our application can be modified to fit other needs such as those of a hospital, or to track tasks and/or bugs rather than problems. The Hopla Helpdesk is not the only way to make a help desk, but just the result of our approach to it, which we have documented in this report.

Bibliography

[1] American Medical Association. Patient confidentiality. Webpage, 2010. URL http://www.ama-assn.org/ama/pub/physician-resources/legal-topics/patient-physician-relationship-topics/patient-confidentiality.shtml. Last viewed: 15/12.

[2] Pragati Software By Pradyumn Sharma, CEO. An introduction to extreme programming. Webpage, 2004. URL http://my.advisor.com/doc/13571. Last viewed: 10/12-10.

[3] Chromatic. An introduction to extreme programming. Webpage, April 2001. URL http://linuxdevcenter.com/pub/a/linux/2001/05/04/xp_intro.html. Last viewed: 10/12-10.

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

[5] Steve Cornett. Code coverage analysis. Webpage, 2010. URL http://www.bullseye.com/coverage.html. Last viewed: 13/12.

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

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

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

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

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

[11] Ward Cunningham. Manifesto for agile software development. Webpage, 2001. URL http://agilemanifesto.org/. Last viewed: 14/12.

[12] Firefox. Install or update firefox: different platforms. Website, 12 2010. URL http://mozilladanmark.dk/produkter/firefox/.

[13] Google. Install or update google chrome: System requirements. Webpage, 2010. URL http://www.google.com/support/chrome/bin/answer.py?answer=95411. Last viewed: 17/10.

[14] Leo Hsu and Regina Obe. Cross compare of sql server, mysql, and postgresql. Webpage, May 2008. URL http://www.postgresonline.com/journal/index.php?/archives/51-Cross-Compare-of-SQL-Server,-MySQL,-and-PostgreSQL.html. Last viewed: 19/11.

[15] John and Mayo-Smith. Two ways to build a pyramid. Web, oktober 2001. URL http://www.informationweek.com/news/development/tools/showArticle.jhtml?articleID=6507351.

[16] AUSIF MAHMOOD. *DISTRIBUTED DATA APPLICATION USING VB.NET & ASP.NET*, chapter Disconnected Database Access. Number 3. University of Bridgeport. URL http://www1bpt.bridgeport.edu/sed/projects/cs597/Fall_2001/dkiran/chapter3.html. Last viewed: 7/12.

[17] 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.

[18] Mark Michaelis. A unit testing walkthrough with visual studio team test. Webpage, March 2005. URL http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx.

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

[20] Department of Computer Science at Aalborg University. Helpdesk software tool (sw3). Webpage, 2010. URL https://intranet.cs.aau.dk/fileadmin/user_upload/Education/E10/Helpdesk_Software_Tool_SW3_E10_.pdf. Last viewed: 16/12.

[21] Roger S. Pressman. White box test. Webpage, 1992. URL http://www.cs.aau.dk/~normark/prog1-01/html/noter/test-dokumentation-slide-whiteteknik-section.html. Last viewed: 10/12.

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

[23] Dr. Winston W. Royce. Managing the development of large software systems: concepts and techniques. *ICSE '87 Proceedings of the 9th international conference on Software Engineerin*, pages 328 – 338, 1987. URL http://delivery.acm.org/10.1145/50000/41801/p328-royce.pdf?key1=41801&key2=7652332921&coll=DL&dl=ACM&CFID=2004164&CFTOKEN=66826314. ISBN: 0-89791-216-0.

[24] Abraham Silberschatz, Henry F. Korth, and S Sudershan. *Database System Concepts*. McGraw-Hill, internation edition 2011 edition, 2011. ISBN: 978-007-128959-7.

[25] Laurie Williams. Testing overview and black-box testing techniques. Webpage, 2006. URL http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf. Last viewed: 13/12.

Part VI

Appendix

A

Test Cases for Problem Search

```
#region Search code coverage
#region Test 1: Search for no tag, minimum number of problems = 0
[TestMethod()]
public void A0_D0_SearchForNoTagNoProblems()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 0;

    expected = new List<Problem>();
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.AreEqual(expected.Count, actual.Count);
    #endregion
}
#endregion

#region Test 2: Search for no tag, minimum number of problems = 1
[TestMethod()]
public void A0_D1_SearchForNoTagOneProblem()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 1;

    expected = new List<Problem>
    {
        problems[2]
    };
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 3: Search for no tag, minimum number of problems = 6
[TestMethod()]
public void A0_Dx_SearchForNoTagSixProblems()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 6;

    expected = new List<Problem>
    {
        problems[2],
        problems[3],
        problems[4],
        problems[5],
        problems[0],
        problems[1]
    };
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 4: Search for tag 3, minimum number of problems = 1
[TestMethod()]
public void A1_B1_C1_D0_SearchForTag3OneProblem()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 1;

    tags[3].IsSelected = true;
    expected = new List<Problem>
    {
        problems[5]
    };
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.IsTrue(actual.Count >= minNoProb);
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 5: Search for tag 3, minimum number of problems = 2
[TestMethod()]
public void A1_B1_C1_D1_SearchForTag3TwoProblem()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 2;

    tags[3].IsSelected = true;
    expected = new List<Problem>
    {
        problems[5],
        problems[2]
    };
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.IsTrue(actual.Count >= minNoProb);
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 6: Search for tag 3, minimum number of problems = 4
[TestMethod()]
public void A1_B1_C1_Dx_SearchForTag3TwoProblem()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 4;

    tags[3].IsSelected = true;
    expected = new List<Problem>
    {
        problems[5],
        problems[2],
        problems[3],
        problems[4]
    };
    #endregion

    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.IsTrue(actual.Count >= minNoProb);
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 7: Search for tag 0 and 1, minimum number of problems
    = 4
[TestMethod()]
public void Ax_Bx_Cx_D0_SearchForTag0And1FourProblems()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 4;

    tags[0].IsSelected = true;
    tags[1].IsSelected = true;
    expected = new List<Problem>
    {
        problems[0],
        problems[1],
        problems[3],
        problems[4]
    };
    #endregion


    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.IsTrue(actual.Count >= minNoProb);
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 8: Search for tag 0 and 1, minimum number of problems
    = 5
[TestMethod()]
public void Ax_Bx_Cx_D1_SearchForTag0And1FiveProblems()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 5;

    tags[0].IsSelected = true;
    tags[1].IsSelected = true;
    expected = new List<Problem>
    {
        problems[0],
        problems[1],
        problems[3],
        problems[4],
        problems[2]
    };
    #endregion


    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.IsTrue(actual.Count >= minNoProb);
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion

#region Test 9: Search for tag 0 and 1, minimum number of problems
    = 6
[TestMethod()]
public void Ax_Bx_Cx_Dx_SearchForTag0And1SixProblems()
{
    #region Arrange
    List<Problem> expected = null;
    List<Problem> actual = null;
    int minNoProb = 6;

    tags[0].IsSelected = true;
    tags[1].IsSelected = true;
    expected = new List<Problem>
    {
        problems[0],
        problems[1],
        problems[3],
        problems[4],
        problems[2],
        problems[5]
    };
    #endregion


    #region Act
    actual = ProblemSearch.Search(catTag, problems, minNoProb);
    #endregion

    #region Assertions
    Assert.IsTrue(actual.Count >= minNoProb);
    Assert.AreEqual(expected.Count, actual.Count);
    for (int i = 0; i < actual.Count; i++)
    {
        Assert.AreEqual(expected[i].Id, actual[i].Id);
    }
    #endregion
}
#endregion
#endregion
```

Code snippet A.1: *The nine test cases for the problem search function*

B Project Proposal

The following text is the project proposal which our project is based on:

“

Helpdesk Software Tool (SW3)

The topic of the semester project is to implement a web-based helpdesk tool that gives students access to all relevant current and historic information associated to their problems/tasks. The tool significantly improves communication efficiency by supporting direct collaboration with clients (students) and service providers (members of technical staff). It must support basic problem management functions such as the following:

Trouble ticketing - each problem should be given a job number for tracking purposes

Priority management - it should be possible to prioritize jobs so that effective use of the technical staff is made in solving those problems

Records of solutions - the solution of the solved problem should be saved, so that identical problems should be cross-referenced and the (identical) solutions are not saved multiple times

Searchable - it should be possible to search the records for similar problems, at least a keyword based search should be incorporated

Statistical information - it include service goals for problem processing times, most frequent problems, creation of FAQs etc.

It may also support software specific functions such as following:

Estimation - to estimate the solution date and time

Performance - to easily navigate among the existing solutions

Etc.

”

[20]