# Rettelser

II

# Contents

# Part I

# Analysis

# 1 Task

## 1.1 Purpose

At any workplace or educational institution the employees or other attendees will encounter non work or non educational problems. e.g. a broken light bulb in an office is not related to the person working in the offices daily business. To solve the problem of the broken light bulb the person who encountered the problem require a maintenance officer, this is done by either calling him by phone, email or by personally attend his office. If the maintenance officer is busy he will not respond by phone or by attending his office. Mail will work better since it can be viewed when he has time.

For all the methods the officer does not get a full work list with all problems and it can be hard to keep record of the work and the problems.
Another problem with this approach is that the person who encounters a problem can solve the problem by himself, but just need a few informations. Then a system which collects all problems enable the users to see the solutions from a similar problem and use this to solve his problem.

From this we deduct that the overall purpose of the system is to ease the collaboration between the people who need help and the people who can help. The system should reduce the time spent by the people who submits problems and also the time spent by the people who should solve the problems. Thereby increasing the overall efficiency of the target environment.

## 1.2 System Definition

It is desired that our system should be as following:
A webbased system used to ease the collaboration between client, who has a problem, and technical staff, related to solving the problem, where the main emphasis is on getting the problems to the persons who are best at solving them, as efficient as possible.

Before a problem is actually submitted to the system, the client is asked to *categorizes* the problem by selecting tags from categories which define and describe the problem as specific as possible. Based on this categorization, the system will search for similar problems which solution might help the client, and therefore remove the need for submitting an identical or almost identical problem to the system.

If the client finds a problem which is identical to his/her own problem – but without a solution – the client is presented with the option to *subscribe* to the problem, and receive notifications whenever new comments are made or a solution gets attached to the problem.

If the client should not find a solution to his/her problem, he/she would be allowed to submit the problem, which would be assigned to an appropriate staff member, based on the clients selection of tags and categories, and the current workload of the staff members.

If a client can not determine the correct category for a specific problem, it will simply be submitted without the categorization, and the system will from there do one of the following three things, which a system administrator specifies:[1]

- distribute between all staff members, based on their workload
- distribute between all staff-personal associated with a specific department, based on their workload
- distribute to a single staff member, regardless of his workload

Every time a problem is assigned to any staff member, he or she will get a notification about it.

The system will generate priority sorted to-do lists for the staff. The priority of the problems will be based on the tags that they were given by the client who submitted the them. The client is able to suggest a deadline to the staff members, but this suggestion has no effect on the problem unless a staff member accepts the suggested deadline or sets one him/her self.

The system will make estimations based on time for solving problems of the same or somewhat equal categorization, the current workload of the assigned staff member, and the priority of the specific problem versus the other problems of which the specific staff member has to solve as well.

When problems are committed to the system and assigned to a staff member, the staff and client can add notes to the problem as a way of communication. Every time a change is made to a problem or a comment has been added, all associated clients and the assigned staff member get notified.

[1] FiXme Dødelige: Hvor kan admin gøre det henne? Jeg ar faktisk en ide: Hvad med at admin kan vælge 1 eller flere depts som skal tage sig af ukategoriserede problemer. Så kan det være alle departments eller en ghost department med en eller flere ansatte eller blot en almindelig department

### 1.2.1 FACTOR criterion

From the system definition we are here specifying the FACTOR criterion which we want our system to fulfill.

**F (Functionality)** Rankings, keep track, knowledge saved, Statistic time estimation, generate todo-lists.

**A (Application domain)** People at the university or at another organization

**C (Conditions)** Only people associated with university should access our system, a webbased system, to-do-lists must be agile

**T (Technology)** Web based, DBMS

**O (Objects)** Problems, solutions, staffs, admins, clients, departments

**R (Responsibility)** An administrative communication medium to ease communication between staff and client.

## 1.3 Context

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

To fully understand the context we draw a rich picture which can be seen on figure 1.1. The central aspects on the rich picture is that both the problem submitter and the problem solver can act on the problem, but will communicate through the system. From the rich picture we see a few conflicts in the environment. The first is that problem exist but cannot be found. The second is that there could exist two similar problem. The central objects are the problems and solutions.

### 1.3.1 Problem Domain

A central phenomena in our system will be to add a problem to the system. This will occur when a client finds a problem in the organization and wants to submit it, in order to get it solved. This phenomena will also include a staff member, namely the one who will initially be assigned to the problem.

Another important phenomena in our system is when a problem is solved. This is initiated by a staff member and will result in a notification to the clients whos are subscribing to the problem. A client who submit the problem can suggest it to be reopened if he/she is not satisfied with the solution.

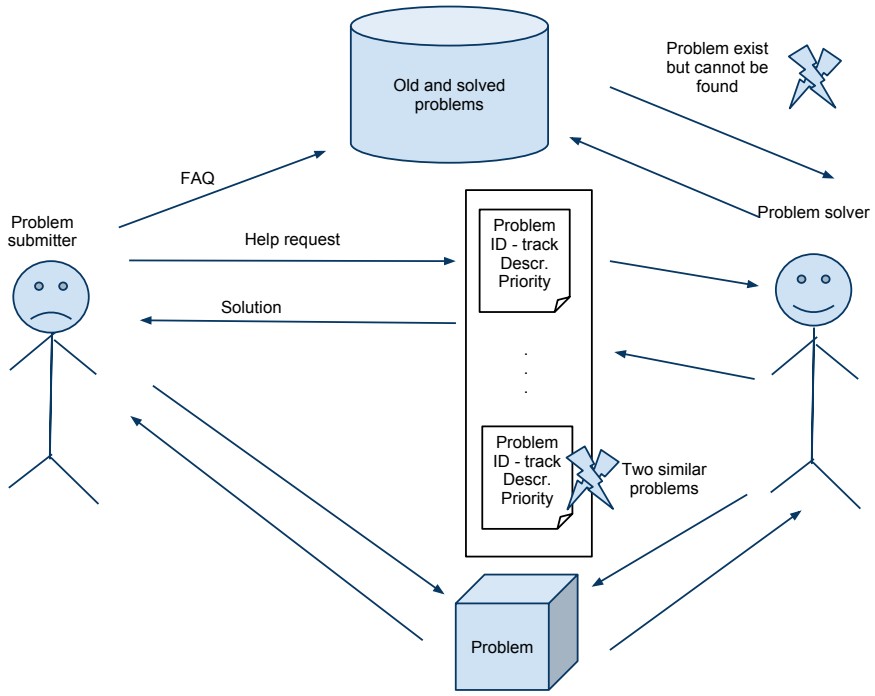


Figure 1.1: Rich picture of our system

### 1.3.2 Application Domain

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

# 2 Problem Domain Analysis

In order determine the nature of our solution we have to analyze the context in which it is going to be used. The classes and events described in this chapter are reflections of how the system is in the physical world.

## 2.1 Classes and Events

In the process of finding classes and events we made up several classes and events who are not used in the actual program. To come up with the classes and events we talk about different possible scenarios and based on that we developed our class diagram. This section describes all the classes and events in the final iteration of the process and all the redundant classes and events are omitted. On figure 2.1 the relations between classes and events are illustrated.

### 2.1.1 Classes

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

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

**Staff** The `Staff` class inherits from the `Person` class, e The Staff Class changes the state and properties of problems.

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

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

### 2.1.2 Events

**Problem added** This event occurs when a user has submitted a problem.

**Problem solved** Occurs when an event receives status of s̈olved:

**Problem updated** Occurs when the content of an event is altered.

**Problem assigned** This event occurs when the problem is assigned to a member of staff.

**Problem unassigned** This event occurs when the problem is unassigned from a member of staff.

**Problem deleted** When a problem is deleted.

**User replied** An event that occurs when the user replies to something the system has sent him.

**Staff replied** An event that occurs when the staff-member replies to something the system has sent him.

**Department created** An event that occurs when a new department is created.

**Department closed** An event that occurs when a department is closed.

**Role assigned** Occurs when a person has received a new role. We later found that this event was redundant with Staff employed, and decided not to use it.

**Role unassigned** Occurs when a person has lost a role. We later found that this event was redundant with Staff fired, and decided not to use it.

**Person created** This event occurs when a new

**Staff employed** This event occurs when a person is employed as a staff.

**Staff fired** Event that occurs when a staff-member is fired.

**Solution assigned** This event triggers when a solution is assigned to a problem.

**Problem categorized** This event occurs when a problem is tagged.
[1]

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

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

Figure 2.1: Problem-domain analysis event table

## 2.2 Structure

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

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

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

A problem consist of none or many *comments*, none or many *solutions* and, one or many *tags*. A **tag** belongs to a category and a **category** belongs to a **department**. The class diagram is illustrated at figure 2.2

## 2.3 Behavior

In this section of the report, we will describe the behavior of the five classes from our class diagram. The sequence of events is not determined. On the sequence diagram of a problem figure 2.3, the events “detach problem” can only happen after the event “attach problem” but it dose not show on the sequence diagram.

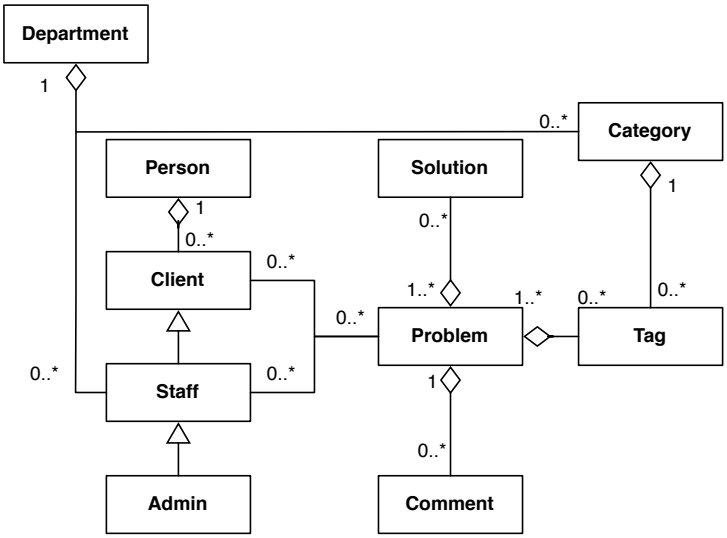


Figure 2.2: Class diagram

### 2.3.1 Problem

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

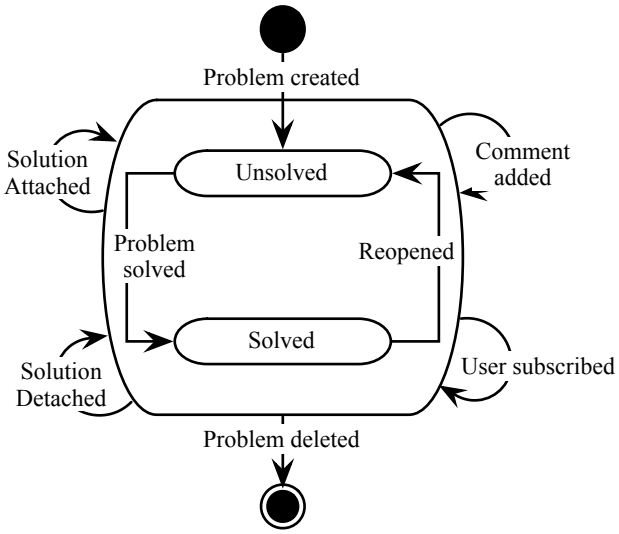


Figure 2.3: Statechart of a problem

### 2.3.2 Person

As shown in figure 2.4 a person is assigned a role in the system as soon as he is created. Thereafter multiple roles can be assigned to him.

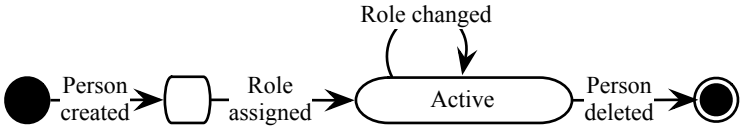


Figure 2.4: Statechart of a problem

### 2.3.3 Staff

Figure 2.5 shows the statechart of the staff class. As shown on the Staff is a role to be assigned to persons. After a person receives that role, he or she can start receiving problem assignments. When all problems are either unassigned or solved, the person will be idle, meaning that the person is not doing anything useful with it's time. The role Staff can be unassigned only when there are no more problems assigned to that person.

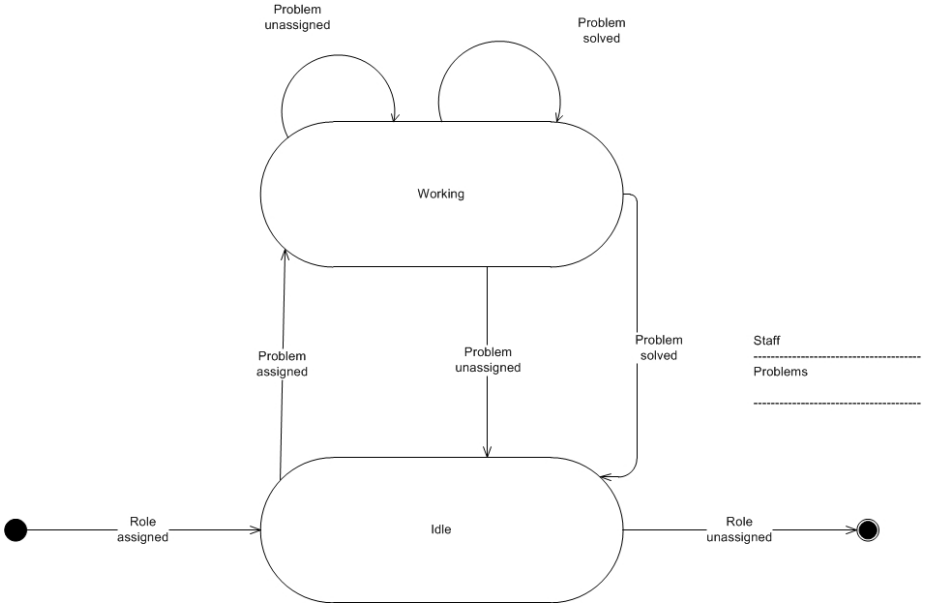


Figure 2.5: Statechart of a problem

### 2.3.4 User

The statechart for a user consists of working state in which the user can create problems and comment on them. This is shown by figure 2.6.

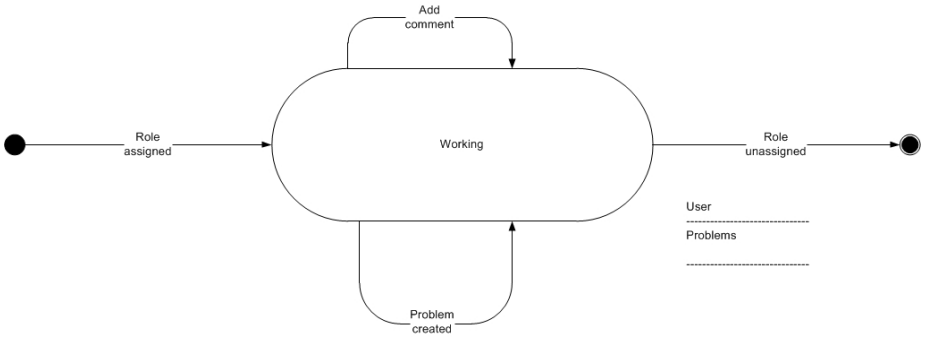


Figure 2.6: Statechart of a problem

### 2.3.5 Department

As shown in figure 2.7 can be open, after which staff can be both hired and fired. A department exists untill it is closed.

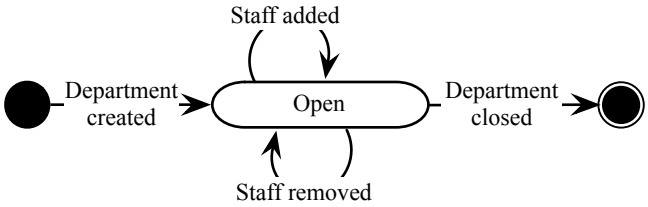


Figure 2.7: Statechart of a problem

# 3 Application Domain Analysis

*In this chapter the application domain will be analyzed and our choices regarding the application domain explained.*

## 3.1 Usage

There are two key elements in need of analysis in order to successfully model the usage of our system. Actors associated with the system, and events associated with the system.

| | ***Actor*** | | | |
|---|---|---|---|---|
| ***Use case*** | Client | Staff | Workload monitor | Admin |
| Submit problem | ✓ | ✓ | | ✓ |
| See all problems | ✓ | ✓ | | ✓ |
| Balance workload | | | ✓ | |
| Solve problem | | ✓ | | ✓ |
| Administrate | | | | ✓ |
| Get Statistics | | ✓ | | ✓ |

Figure 3.1: *Actor & use case table*

Figure 3.1 shows the relationship between use cases and the actors of our system. Note that staff members also are able to submit problems. This is due to the fact that both client and staff simply are *roles*. Persons associated with the role called staff can therefore also be associated with the role client, which are able to submit problems to the systems.

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

### 3.1.1 Actors

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

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

Figure 3.2: *Description of the actor client.*

The staff members are more privileged and can solve and delete problems. All staff members can act as clients if they themselves has a problem which should be addressed to another department. E.g. the lightbulp in the IT-administrators office is broken and the maintenance officer should fix it. These two actors are described in details in figure 3.2 and 3.3. Beside staff and client the system has the two actors: admin and the non-human actor, workload monitor. The admin is a more privileges staff or a manager of the staff.

The admin can change the tags and categories, create/remove departments and add/hire/remove staff to the departments. The workload monitor is an automated system which on a given time interval checks if some staff members is overload with tasks(problems). In that case it will reassign the problems.

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

Figure 3.3: *Description of the actor staff.*

### 3.1.2 Use Case

The use cases in figure 3.1 are described below.

**See all problems** The use case see all problem is used by both client and staff. It works as a searchable list.

**Balance loadwork** Balance loadwork is the operation of the workload monitor. It checks and compares all staff members workload and redistribute problems in order to equally balance the workload in each department.

**Get statistics** The use case get statistics is trivial and is a list of different views each showing a kind of statistics. The use case can be accessed by admin, client, and staff, but they cannot access all the same statistics.

**Submit problem** The use case submit problem is only used be the actor client. Except for cases when staff or admin acts as client. A use case diagram is shown in figure 3.4.

- **Use Case:** Submit problem is initialized when a client has a problem and wishes to submit that problem to the system in order to get help from the staff. He first has to select a category and from that choose one or more tags, he can change category and select even more tags. When the client is done selecting tags the system compares the selected tags with other problems. If similar problems is found the client is presented with these. If one of these matches his particular problem, he can subscribe to the problem(if active), reopen problem(if closed)[1], or use the information in the old problem to solve his problem independently. If no similar problem was found the client creates a problem with a description and the previously selected tags. Hereafter the problem gets assigned to a staff.
- **Objects:** Problem, solution, tag, category, client, staff. (Department)[2]
- **Functions:** Search existing problems, compare problems, create problem, attach user to problem.

**Solve problem** The use case solve problem is the staffs primary working usage of the system. This is where he goes to get his worklist and to solve problems. A diagram is shown in figure 3.5.

- **Use Case:** The use case is initialized when the staff wants to check his worklist. He is then presented with a list of unsolved problems assigned to him. He can then click on one of the problems to solve the problem, see status of it, add comments to it, search the database for similar problem, reassign it, or delete it.
- **Objects:** Problem, solution, tag, category, comment, client, staff and department.
- **Functions:** Add comment, reassign, change status, delete problem, search database, create solution, attach solution and get staff worklist.

[1] FiXme Dødelige: Udemærket her, vi skal bare huske at den i stedet laver et nyt problem
[2] FiXme Dødelige: Hvad betyder det at den er i parentes

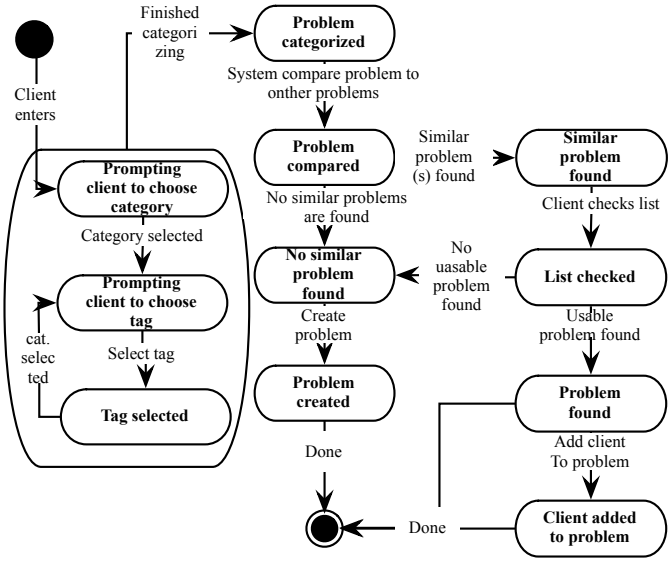


Figure 3.4: *A state chart diagram of the use case submit problem.*

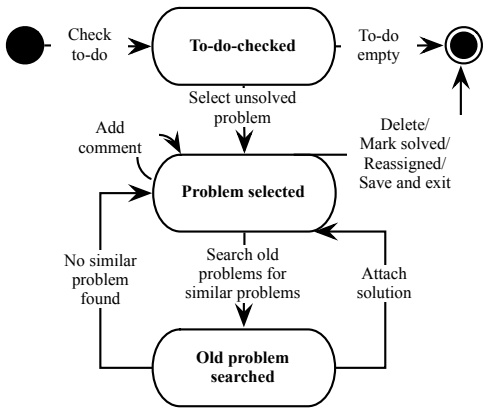


Figure 3.5: *A state chart diagram of the use case solve problem.*

**Administrate** The use case administrate is used by the admin to administrate persons, tags, categories and departments. A statechart diagram is depicted on figure 3.6.

- **Use Case:** The use case starts as the admin enters the site. He can add/remove persons and departments. The admin can select departments and from the selected department he can add or remove staffs and categories. From both a selected department and the main window he can select a person to edit. If he selects a department he can select a category to change and from that category select, add, and remove tags. From a selected tag the admin can set/change the priority of the specific tag. From any point he can go back or leave the administration.
- **Objects:** Staff, Client, category, tag, department.
- **Functions:** Delete person, Add person, create department, set permissions, delete department, set priority, add category, delete category, set tag visibility, set category visibility, create tag, delete tag.

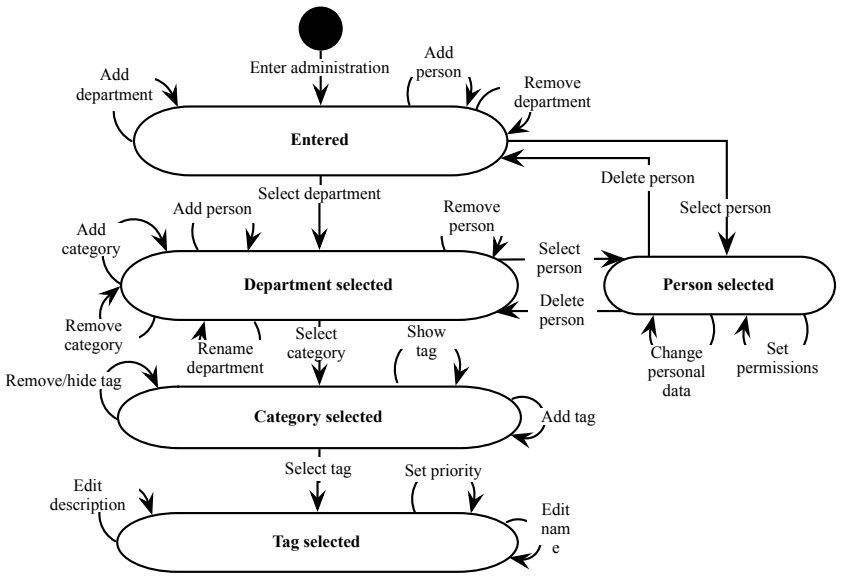


Figure 3.6: *A statechart diagram for the use case administrate.*

## 3.2 Function

The purpose of this section is to "determine the system's information processing capabilities" [2, p. 137]. Ultimately resulting in "a complete list of functions with specification of complex functions." [2, p. 137]

Comprehensive knowledge of the OOA&D method is assumed, and therefore we will not go in detail on function types etc. [3]

[3] FiXme Dødelige: overkill? – Nah det er okay

| | | |
|---|---|---|
| Add comment | Simple | Update/signal |
| Reassign Problem | Simple | Update/signal |
| Change problem status | Simple | Update/signal |
| Delete problem | Simple | Update |
| Search problems | Complex | Read/compute |
| Create solution | Simple | Update |
| Attach solution | Simple | Update/signal |
| Distribute problems | Complex | Read/update/signal |
| Create problem | Medium | Update/compute |
| Get staff todo list | Simple | Read |
| Create tag | Simple | Update |
| Create category | Simple | Update |
| Change visibility of category | Simple | Update |
| Change visibility of tag | Simple | Update |
| Delete tag | Simple | Update |
| Delete category | Simple | Update |
| Create department | Simple | Update |
| Delete department | Simple | Update |
| Add person | Simple | Update |
| Rename person | Simple | Update |
| Rename department | Simple | Update |
| Remove person | Simple | Update |
| Get statistics | Complex | Compute |
| Subscribe client | Simple | Update/signal |
| Get priority | Complex | Compute/read |
| Set priority | Simple | Update |
| Set permissions | Simple | Update |

Figure 3.7: Function list

**Add Comment** This function simply adds a comment to an existing problem, making the comment visible for other client and staff members. This function is marked as an *signal*-function because a notification is sent to the assigned staff member.

**Reassign Problem** The reassign problem function reassigns a specific problem from one assigned staff member to another. It is marked as a *signal*-function as the new assigned staff member received a notification of the newly reassigned problem.

**Change Problem Status** This function changes the status of a given problem. It is marked as a *singal*-function since the associated clients receives a notification.

**Delete Problem** To delete a problem, this function is used.

**Search Problems** This function searches the model for problems with specific tags. It is both *read* and *compute* because it will compute which problem(s) compares best to the tags which are being searched for. This function is *complex* because of the ordering of the found problems.

**Create Solution** This function simply creates solution to a problem and adds it to the model.

**Attach Solution** This function attaches an existing solution to a problem, and notifies the associated clients about it. This function is a *signal*-function as it notifies the associated clients.

**Distribute Problems** The distribute problem function is ran everytime a new problem is created and by a given interval to ensure that no staff is overcrowded with tasks. It distributes newly created problems by assigning them to an appropriate staff member, based on current workload. Marked as a *read*-function as it reads from the model. Marked as a *signal*-function as it notifies the associated clients, and the newly assigned staff member.

**Create Problem** This function creates a new problem and adds it to the model. Every time this function is ran, it computes which department is best suited for solving the problem, and thereafter assignes the problem to the staff member which has least problems to solve. [4]

**Get Staff Todo List** This function simply fetches the staff members todo list. Hence marked as a *read*-function.

**Create Tag** creates a tag, which is an element used for categorizing problems, which purpose is to make searching easier, also attaches the tag to a category.

**Create Category** To create a category, this function is used. It also attaches that category to a department.

**Delete Tag** If a tag is not attached to any problem it can be deleted, if not it can only be hided.

**Delete Category** If any tag is attached to the attempted deleted category it can not be deleted and can only be hidden.

**Change Visibility of Category** The Change Visibility of Category function changes the visibility of a category. Changes happen and a need of a category may vanish over time, but might be needed again in the future, thus making this function necessary.

**Change Visibility of Tag** This function hides tag if the admin no longer finds it useful. In cases a delete can not be done, because the *tag* is related to some problems.

[4] FiXme Dødelige: Hvorfor er "create problem" markeret som en compute-funktion?

**Create Department** All staff members are associated with a department in order to ease the process of distributing newly created problems to appropriate staff members.

**Delete Department** As the name dictates, this function simply deletes a department.

**Add Person** A person can either be a client, staff, or an admin. See figure 3.1 in section 3.1.1. This function creates a person with characteristics to define him/her in the system.

**Rename Person** This is a trivial function, which simply changes the name of a person.

**Remove Person** This function removes a person from the system.

**Get Statistics** This function returns statistics, e.g. average time for specific kinds of problems to be solved, hence "complex"-complexity. It is marked as a *compute*-function since it involves a fair amount of computation on the data in the model.

**Subscribe Client** To tie a client and a problem together through a subscription, this function is used. It is marked as a *signal*-function since the assigned staff member receives a notification.

**Set Priority** This function sets the priority of a tag.

**Get Priority** This functions computes the priority of a problem, based on the priority value of the tags attached to the problem and time since it were commited.

**Set Permissions** This is a trivial function, which sets the permissions for a given person.

## 3.3 Interfaces

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

### 3.3.1 Client Interface

The client interface is illustrated in figure 3.8. After login, the client is presented with the *main* screen, which gives three possibilities:

- Add problem
- My problems
- Search for problem(s)

- Statistics

These main possibilities are thoroughly described in the following sub-subsection, since they are the reflect the three use cases of the client in section 3.1; submit problem, see all problems, and get statistics. Due to the nature of our webbased system, the client can anytime terminate the session and logout/close.

#### Add Problem

As the name indicates, this button initiates the process of adding a specific problem to the system.

This is done by selecting *tags* which describes the problem. Tags are grouped under categories. Each tag can only exist under one category, however if the need arises, it is possible to create a duplicate tag under another category. This does not mean that the same tag exists under two categories but two different tags with the same name exists under two different categories.[5] We say that the problem is "categorized" when it has tags associated with it.

When the client is satisfied with the categorization of his/hers problem, he/she can click "Search", which will make the system search for problems with similar categorization, prioritizing those with a solution attached to it. From there, the idea is that the client might find a problem which is identical, or almost identical, which solution will also fix the client's problem. If such a problem is found, the client has the option to *"subcribe"* to the problem, making the client receive all the same notifications as the person who created the problem. This dramatically reduces redundancy in a case where multiple users has the same or similar problem. This also saves the clients the time to create and describe a new problem in the system. If no suitable problem is found, the client is allowed to describe his problem with words, as well as alter the tags which he/she selected earlier, in case he/she changed his/her mind. Ultimately, the problem is added to the database after fully described. This process will initialize the workload monitor, for distributing the problem to a staff member, who is likely to solve the problem.

#### My Problems

This button navigates to a window where the problems created by the client as well as the problems which the client has subscribed to is shown.

#### Search for Problem(s)

Clicking this button will bring the client to the problems search-screen, where he/she can search by specifying tags which possibly are attached to the problems that the client wishes to find.

#### Statistics

6

[5] FiXme Dødelige: Er det smart at have to tags der hedder det samme?

[6] FiXme Dødelige: Der skal laves et interface til statistics, ellers det skal i hvert fald beskrives.

### 3.3.2 Solve problem interface

[7] The solve problem interface is used by the staff to perform his daily duties. This is the interface he will be using to get his worklist, to communicate with the client and to solve the problems. Since the staff often acts as admin the button to go to the administration interface is presented at the main window of the solve admin interface. But only the staff with the correct permissions can enter. The main window contains the client main window, so staff has the same opportunities as client plus the additional options, which har described below.

#### Worklist Window

The worklist window shows a list of unsolved problems assigned to the staff. From here he can select a problem to work with.

#### Problem View

The problem view window contains all needed information about the selected problem. It contains the following buttons:

- **Add comment** takes the staff to the add comment window, where he can enter a comment and add to the selected problem.
- **Add solution** presents the staff for a searchable of problems. Every problem is a link that opens a simplified view of the problem with a button to attach solution. This button takes the staff back to problem view. From add solution the staff can press the button add new solution which takes him to a window where he can enter a solution description. And add the new solution
- **Change assignment** button takes the staff to a window where he is presented with a list of all staff in his departments. Every staff has a checkbox attached. The selected elements presents the staff assigned to the specific problem. Changing this and saving allows the staff to assign or reassign the staff.
- **Delete solution** If any solution is already attached each attached solution has a delete button to remove it from the problem.
- **Delete** button deletes the problem and returns the staff to his worklist.

From any point in the interface the staff is able to logout or close the browser.

### 3.3.3 Admin Interface

The Admin Interface is an extension of the Staff Interface, as the Admin Interface is entered through the Staff Interface. This makes sense because all persons given the permissions to enter the Admin Interface, will also be staff members. There are two major fields[8] of which the admin has influence on, departments and persons. These two are described in the following.

[7] FiXme Dødelige: Jeg synes bare det skal kaldes Staff Interface, how about it?
[8] FiXme Dødelige: Foretrækker noget andet her. Måske bare classes?

### Administrate Persons

When an admin enters the administrate persons window he/she is presented with a list of persons and a “New Person” button. From this window the admin can select an existing person and change the data of him or her

### Administrate Departments

---

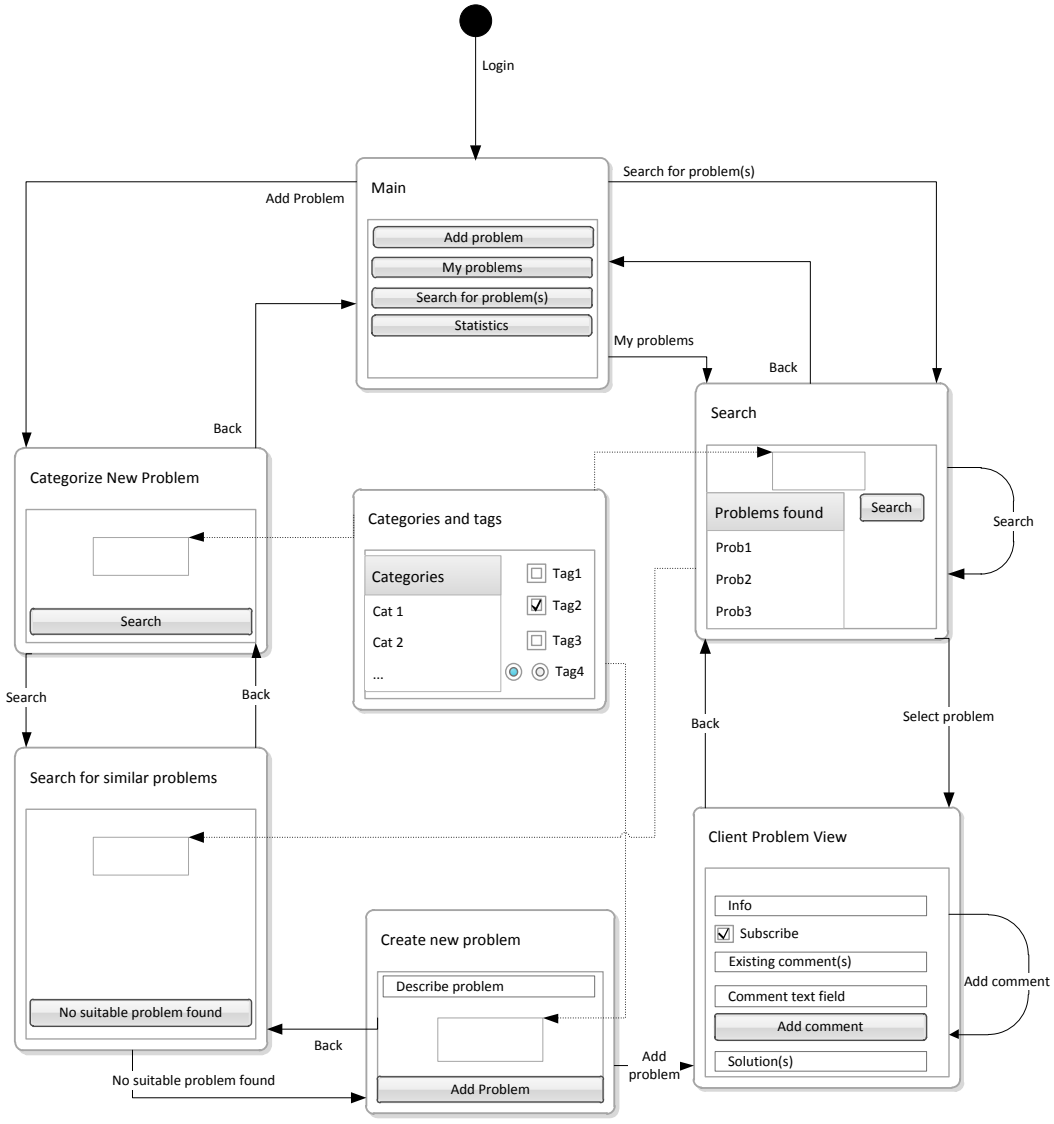


Figure 3.8: Client Interface

# Part II

# Design

# 4 Task

## 4.1 Purpose

Hopla Helpdesk should ease the problem solving process in the applied environment. It should balance the problems between the staff based on their workload, the ETA, the priority and the deadline of the problem. The system should support a method for the staff and client to communicate, enable the users of the system to search through existing problems and monitor their own problems, be accessible everywhere through the internet and organize the problems in categories and tags.

## 4.2 Criteria

The design of Hopla Helpdesk is specified from the following criteria: Usable, secure, efficient, consistence, reliable, maintainable, testable, comprehensible, reusable, portable, and interoperable. The definition of these criteria is shown in table 4.1. [2][p. 178]

### 4.2.1 Prioritizing Criteria

Since it is not possible to prioritize each criterion equally, we have chosen to use a four step priority scale: Very important, important, less important, and irrelevant. Each criterion defined in figure 4.1 is prioritized in figure 4.2. Further the figure 4.2 shows a column labeled "Easily Fulfilled" which specifies whether a given criterion will be fulfilled without much effort.

How we have prioritized the criteria is based on our system definition, which is found in chapter 1.2. The reasoning for the priority of each criteria in figure 4.2 is shown bellow.

**Usable** It is important that our Hopla Helpdesk is user friendly because it can be used in any organization, it is however not very important since we during this project primarily will tailor it to the university's system. Further more we are more concerned with the functionality of the system than the usability, hence; important

| ***Criterion*** | ***Definition*** |
|---|---|
| Usable | The end user can easily use the system |
| 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 systems accessibility |
| Maintainable | The cost of locating and fixing system errors |
| Testable | The cost to ensure performs intentionally |
| Flexible | The cost for the end user to modify the system after deployment |
| Comprehensible | How easy it is for the end user to understand the system |
| Reusable | The potential for using parts of this system in another system |
| Portable | How much effort needed to change the platform of the system |
| Interoperable | How well the system cooperates with other systems |

Figure 4.1: *Definition of criteria*

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

Figure 4.2: *The criteria with a priority*

**Secure** For the Hopla Helpdesk security is less important. We want to make sure whether it is a client or a staff member who is using the system – the clients are e.g. not allowed to solve problems or choose who should be assigned to what problem. 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 care for the efficiency of our system, 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. Further more the Hopla Helpdesk model should not contain duplicates, since it could compromise the integrity of the statistics which the system generates. This lead us to prioritize consistence as important.

**Reliable** The reliability of the Hopla Helpdesk system is not of great interest to us. We do not actively do anything to increase the reliability of the system, 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 not intend to maintain the system after it is finished, it is prioritized as irrelevant.

**Testable** We want our system to work and to make sure it does, we will run tests. Therefore we want our system to be testable.

**Flexible** It is very central to our system that it is flexible, because we want it be generic – that it can be adapted to any organization without much or any cost. We have chosen this to be a very important criteria.

**Comprehensible** Since the Hopla Helpdesk system is supposed to be generic, it should be easy for the user to understand it. However our focus is primarily on functionality, so we have prioritized it important.

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

**Portable** Our systems 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 system. Therefore it is considered irrelevant. The end users will access our system through a browser, so we assume that it can be easily fulfilled since there are many browsers for different platforms. [1] [1]

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

[1] FiXme Dødelige: Flere kilder

# 5 Design platform

## 5.1 Equipment and Software

The required software for our system consists of two major applications; a DBMS supporting the query language SQL, and a web server capable of running our web-scripting interface of choice. The data storage model will be the relational database model and the DBMS we will be using is PostgreSQL version 8.4.5 as our only DBMS. For the web server we will be using Microsoft Internet Information Services (IIS). The language of choice will be C# using the ASP.NET MVC(Model-View-Controller) framework.

The equipment needed to power these two applications can be any computer with a reasonable amount of processing power, and RAM storage. We will be using a Dell Optiplex 960 with a Core2 Duo CPU, E8400 @ 3.00GHz, with 4 GB RAM installed. Running Microsoft Windows server 2008 R2 Standard.

# 6 Architectural Design

*This chapter will present the criteria which we will use to determine what is important for our system and the structure of our components. The different criteria will be prioritized, so it is possible to determine where we should focus, when we start implementation. The Components section shows the layered architecture and the server-client architecture of our system.*

---

## 6.1 Components

With basis in our system definition in chapter 1.2 and our evaluation of criteria in section 4.2 we found that our main priority is flexibility. To obtain high flexibility we created a closed-strict layers component architecture, we did this to avoid complex dependencies between components and therefore making it easy to change an interface, a function component, or [1]. The architecture has three layers: User- and System-interfaces, Functions, Model as shown on figure 6.1. The figures components will be explained in section 6.1.1.

### 6.1.1 System Component

The systems three actors are displayed as interfaces. The clients and staffs functions are located in the Problem Handler component, while the admins additional functions are located in the Administrator component. The client only have limited access to the Problem Handler component while the staff and admin have full access to the Problem Handler's functions, in addition the admin uses the administrator component. To determine if a user is a client, staff or admin the component Authenticator is used. The Authenticator asks the Database Interface what the permission the user have. Based on what permissions the user have access are given to the different interfaces. The Problem Handler and the Administrator components sends info to the Database Interface who translate the info and sends it to the database. All the components are described below:

**Client Interface** The client interface get inputs from the client, through this interface the client submits problems, post comments and search for existing

[1] FiXme Dødelige: eller hvad?

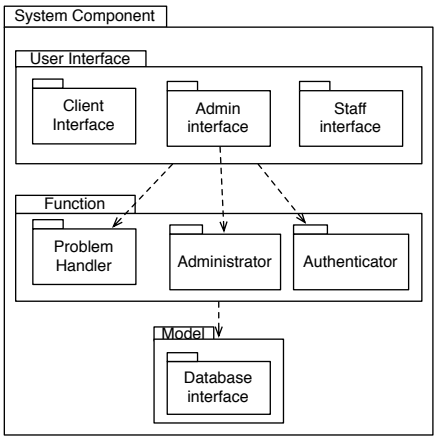


Figure 6.1: *Our system component*

problems. The functions associated with this interface are located in the Problem Handler component.

**Staff Interface** Through this interface the staff members can access all the client functions, submit solutions, post comments to active problems, view worklist and administrate problems. This Interface uses the Problem Handler component.

**Admin Interface** The interface has access to all the client functions, the staff functions, tags/category management, department management and administration of the client/staff. This is due to the fact that if a person is an admin, then he/she also is a staff and a client. The Admin Interface uses the Problems Handler and the Administrator component.

**Problem Handler** The problem handler component controls the handling of new problems, the handling of solutions, the searching function, recording and sending statistics, manages comments and sending notifications by using the notifier component. This component is also responsible for sending out signals to relevant actors. e.g. if a user posts a comment the relevant staff member(s) receives a notification.

**Administrator** Through the admin interface the admin component can be accessed. The functions affiliated with this component are control of departments, management of tags/categories and adding/removing of staffs and clients.

**Authenticator** To determine which rights a person have, they have to authenticate them selves as either client or staff/admin, this occurs through an authenticator component which communicates with the database interface.

**Database Interface** The database interface receives all database requests and translate them to the correct database language.

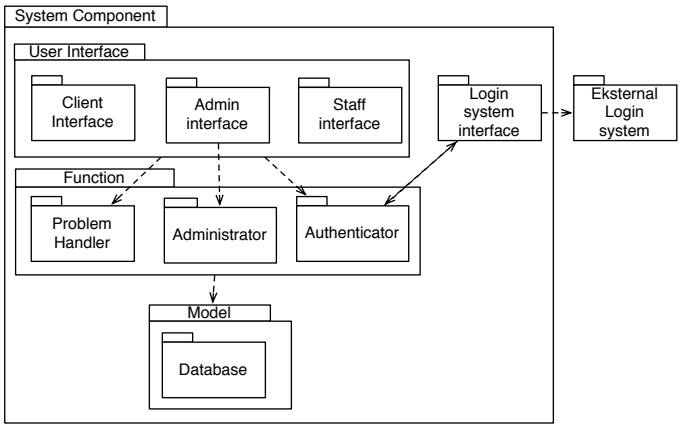


Figure 6.2:

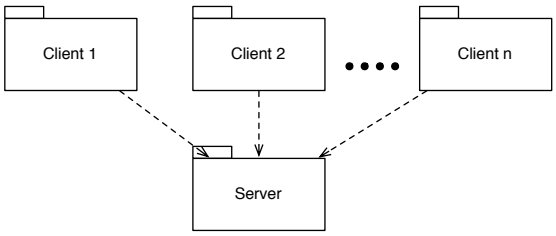


Figure 6.3:

### 6.1.2 Alternative System component

If Hopla Helpdesk is used by a company who already have an Active Directory, then the login system should be integrated with the AD. We designed a component who is able to communicate with an external database[2]. As seen on figure 6.2 a Login system interface component is added. We are not going to implement this component. [3]

**Login System Interface** The Login system interface communicates with the external database

### 6.1.3 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 system 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 6.3. By using Local Presentation [2][p. 200] we enable clients to access the system from anywhere, and still keep the functionality and model on the server and thus keeping the system flexible.

[2] FiXme Dødelige: Blev vi ikke enige om at vi ikke ville bruge external database
[3] FiXme Dødelige: kim: Jeg syntes det skal slettes, billedet og tekst

---

*The priority of the criteria for our system is presented in this chapter along with the reasoning for the priority for each criteria. The architecture of our Hopla Helpdesk system is described in this chapter.*

# 7 Component Design

## 7.1 Structure

Over all there are four components: User interface, system interface, function component, and model component. Note however that the system interface is only used in our Alternative System component in subsection 6.1.2. This section describes how these components internal structure is. A subsection for each component is presented below.

### 7.1.1 User Interface Component

This component consists of three sub-components: Client Interface, Staff Interface, and Admin Interface. Each of these interfaces is described in the sub-subsections bellow.

The sub-components share a login class, which is a window where a user enters his/her user name and password. Depending on the role of the user who is logging in, he/she is directed to the main window of the corresponding interface.

Every window in the sub-components will inherit from a class called "HelpdeskWindow". This class is described in paragraph **??**.[1]

This entire component depends on the function component to gain any functionality. The Authenticator makes sure that a user is directed to the correct Main Window after login, the Problem Handler provides problem search, modifiability, and addition for interfaces, the Admin Interface relies mainly on the Administrator sub-component for administrating person and departments.

#### Client Interface

The structure of the Client Interface is shown in figure 3.8. Each window represents a class and the navigation arrows indicates an association between the

[1] FiXme Dødelige: Dette er en reference til der hvor beskrivelsen af HelpdeskWindow kommer til at være. Indsæt gerne :)

classes. Each object of the different classes will hold a pointer to the object(window) that created it, so it is possible to go back from one window to another.

As mentioned, each window class inherits from the HelpdeskWindow class. Further the Client Problem View inherits from an abstract Problem View class. The reason for this is that there are different problem views – two in the Staff Interface – and making one class that the others inherits from will make the views similar and thereby making the system more comprehensible for the end user.

#### Staff Interface

The structure of the Staff Interface is visualized in figure 7.1. The windows in the figure represents classes of this component, the navigation arrows are translated to associations in UML. Like the Client Interface, the classes in this component

Notice that Staff Main Window aggregates the clients Main Window. Likewise does the Add Solution aggregate the Search Window from the Client subcomponent. If a staff member is an administrator he/she is presented with an administrate button in the main window, which will transfer the user to the Admin Main Window, which is described in the following sub-subsection.

#### Admin Interface

Figure 7.2 shows the navigation diagram of the Admin Interface. Every window inherits from the abstract HelpdeskWindow class.

Like the Staff Main Window aggregated the clients Main Window, the Admin Main Window aggregates the staff Main Window, thereby giving the Admin Interface all the possibilities that the two other interfaces has.

### 7.1.2 System Interface Component

[2] This component is responsible for connecting our system to another systems database, so that our system can use the user names, which already exists in the organization where out system is being implemented. There is no internal structure in this component since it only has a single class which holds the responsibility fr this entire component.

This component is however connected to the Authenticator sub-component in the function component. The connection between these components is that this component is a supplier of data for the Authenticator component. The Authenticator component asks this component to retrieve information about a user in the database and this component provides this information if it is available.

### 7.1.3 Function Component

The function components main purpose is to provide functionality for the users of the system.[3] This component id divided into three sub-components; Problem

[2] FiXme Dødelige: Det her kan nok gøres mere udførligt, men ved ikke om det er meningen
[3] FiXme Dødelige: Nødvendigt?

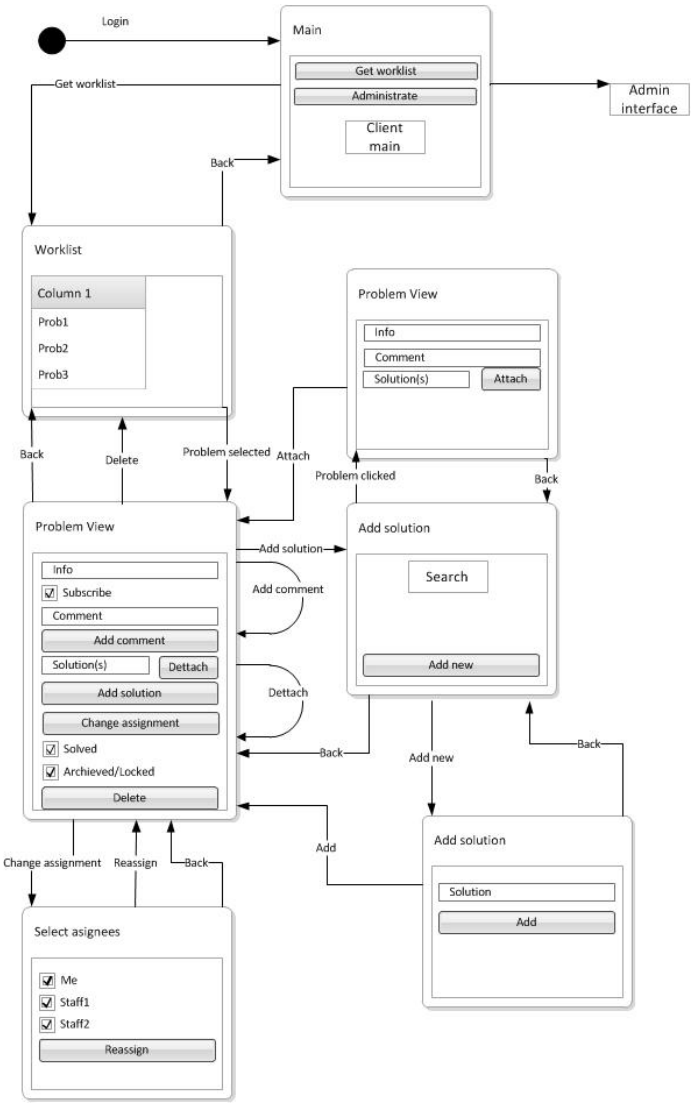


Figure 7.1: *Staff Interface*

Handler, Administrator, and Authenticator. These are described independently in the following sub-subsections.

#### Problem Handler

This sub-component is responsible for handling problems – as the name implies. This component consists of only a single class, which yields no internal structure. It is though connected to the Model component, where the systems data resides. The Problem Handler is able to operate on the model, particularly on the Problem class, but also on the Client class, Staff class, and Solution class.

The Problem Handler component can both create, modify, and delete problems as well as assigning problems to staff members. Note however that the Problem Handler does not assign problems by it self, but rather provide the functionality for the Staff Interface. The Solution class is handled in the sense that solutions can be attached and detached to/from problems through this component.

Both the Staff and Client classes are handle through the relations which are between these classes and the Problem class, namely Assignment, Subscription, and Comment. See subsection 7.1.4 for information on Assignment, Subscription, and Comment.[4]

#### Administrator

In short this sub-component handles everything in the model which the Problem Handler does not. This include adding, modifying, and deleting Person objects, Departments, Categories and Tags. As well as assigning Roles to the Persons.

Like the Problem Handler, this sub-component does not do anything by it self, it simply provides functionality for the Admin Interface.

#### Authenticator

The Authenticator makes sure that a user is authenticated and can only commit actions appropriate for his or her Role. This sub-component is either connected to an external database through the Login System Interface component or is connected to our systems database through our Model component.

### 7.1.4 Model Component

The classes of the Model component are described in section 2.1. However during revision of our class diagram, we decided to insert more classes. Figure 7.3 shows our revised Model component.

The new classes are: Assignment, Subscription, Comment, Tag, Category and Admin. The two first are simply mediators between Staff and Problem, and Client and Problem respectively. They are there to "remember" the events Problem Assigned and Client Subscribe.[5] Comment simply holds a comment for a given problem, and remembers the poster of the Comment in for of the aggregation from client. The Tag class objects can be tied to a problem in order to "Categorize" it. The Tags also belongs to a Category which again belongs to

[4] FiXme Dødelige: Skal tag også skrives på eller ligger det implicit?
[5] FiXme Dødelige: Den findes ikke endnu, men den skal vel laves, no?

a single Department. This way a Problem with some given Tags, can easily be sent to a Staff member of the Department which the problem is categorized to be in.[6]

## 7.2 Classes

In this chapter we will give a brief description and list of attributes for each class together, and a operating specification of each complex operation.

**Person:** This class purpose is to register new users in the helpdesk. When a person register, he/she need to provide the Person class with following attributes:

- name
- mail
- username
- password

**Login:** After a user is registered in the helpdesk, he/she can login using their username and password they choose when registering. The login class purpose it to make sure all users is registered before they can submit problems and add comments. The login class need the following attributes:

- username
- password

**HelpdeskWindow:** All windows in the helpdesk inherits from this class. The HelpdeskWindow class is an abstact class, which purpose is to provide a standard layout for all windows. This class use the following attributes:

- width
- height
- position
- er der mere?

**Problem:** When a user submit a new problem to the helpdesk, he will be using the Problem class. The Problem class responsibility is to make sure the user provides the required attributes:

- title
- text

[6] FiXme Dødelige: Lidt kryptisk måske..

**Client:** This class is the default role, that every user got after registering. The Client class purpose is to provide the user with basic rights, such as submitting a problem and adding a comment. This class doesn't require any attributes.

**Staff:** This is a employee role, you can only have this role if your a employee of the helpdesk staff. The Staff class give the employee rights to change problem status, assign problems, delete problems/comments, edit problems/comments, attach solution. This class doesn't require any attributes.

**Solution:** When a client submit a problem, it's the staff's job to find and attach a solution. The Solution class purpose is to let the staff create and attach a solution. This class requires the following attributes:

- file
- text

**Assignment:** This class purpose is to assign problems to staff members, and let staff members assign problems to other staff members. This class doesn't require

**Subscription:** Clients can subscribe to a problem, each time the problem is updated with either a comment, solution or status change, the client will be informed by mail.

**Comment:** This class give the clients the rights to comment on a problem. The class require the following attributes.

- Text

**Department:** When a problem is submitted, it is assigned to a department based on tags the client provided. The purpose of this class, is to make sure the problems is assigned to the right staff. The Department class require the following attributes:

- name

**Category:** A department contain one more more categories that partly describe the department. The Category class purpose is to make sure, problems is assigned to the right department. The class require the following attributes:

- Name
- Description

**Tag:** Before a client can submit a problem, he/she needs to add predefined tags to the problem. Each tag belongs to a category, more tags can have the same category, but one tag can't have more categories. This class purpose is to add tags to a category or a problem. This Tag class require the following attributes:

- Name
- Description

**Admin:** This is the highest role that a staff member can have. As admin got all rights, the class doesn't require any attributes.

---

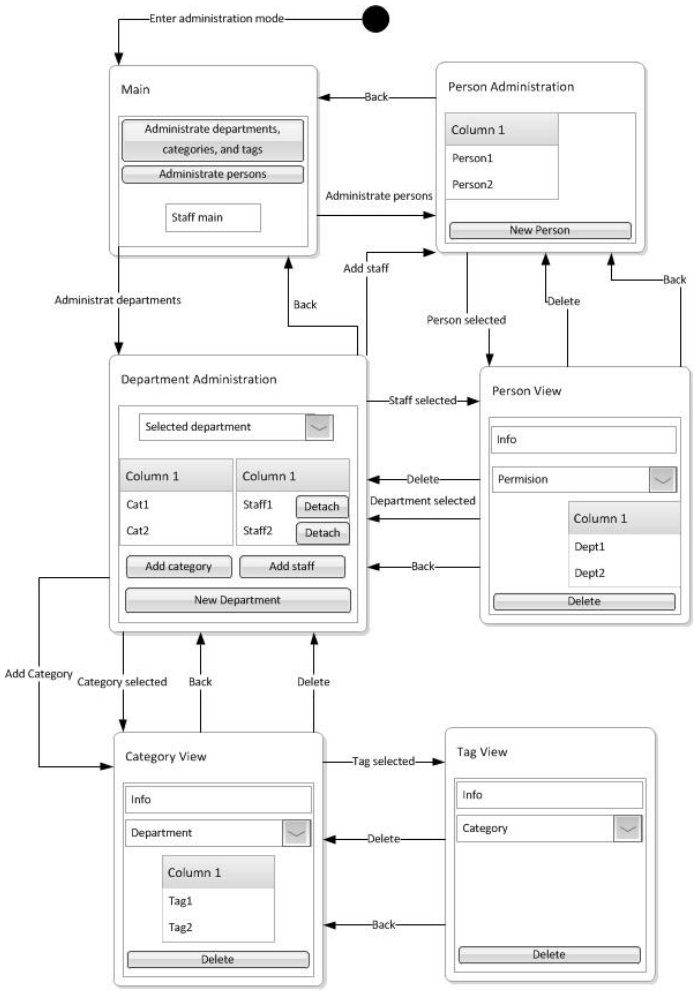


Figure 7.2: *Admin Interface*

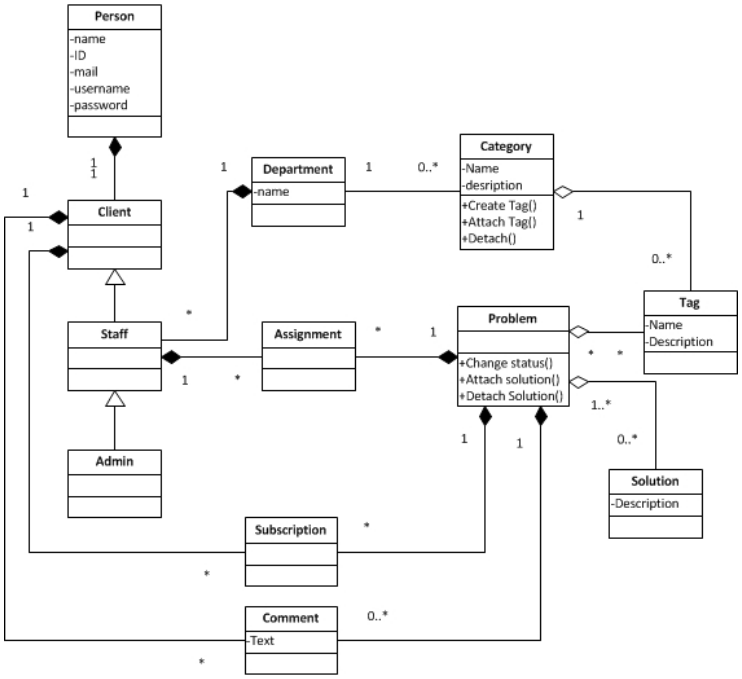


Figure 7.3: *The revised Model component*

# Part III

# Implementation

# 8 Database Scheme

*In this chapter the database scheme will be outlined and discussed.*

## 8.1 E-R Diagram

In order to get an overview of how to structure the database E-R diagrams gives a neat foundation. The E-R diagram can be seen on figure 8.1. The notation for the E-R diagram is based on the citation used in the book Database System Concepts [3, p. 305]. The E-R diagram is based upon the class diagram [1].

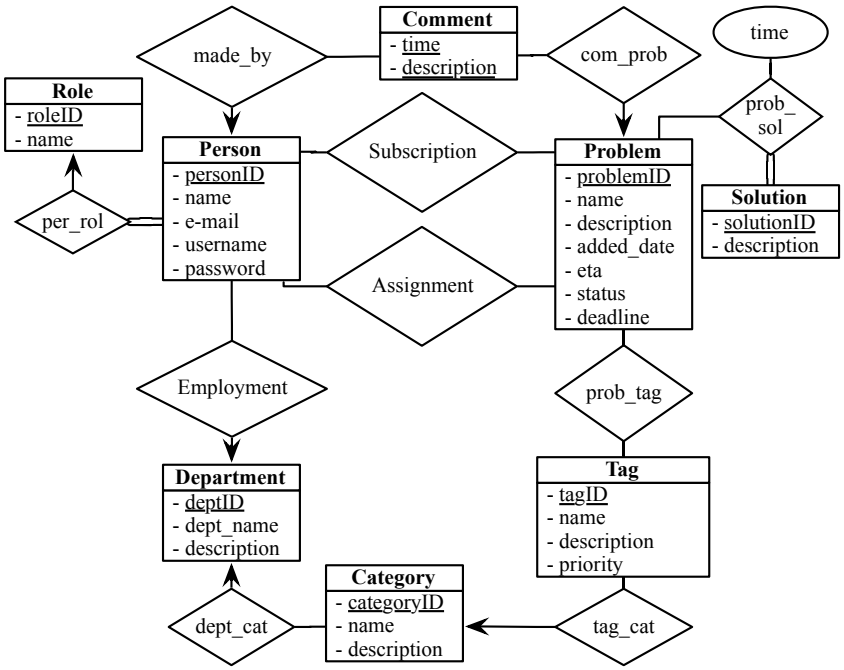


Figure 8.1: *The ER-diagram*

[1] FiXme Dødelige: Indsæt en reference til det første klasse diagram her!!

Every class is turned into a entity and every relation is turned into a relationship [3, p. 259 - 321]. The relationships contain the primary keys from the two entities it is a relation between. Only the relationship prob_sol has an attribute beside the primary keys. This is called time and represent the time for when the solution were attach to the problem, this is used to determine when the problem is solved. The last attached solution to a solved problem must indicate the final of the solving phase.

Any other relation does not need attributes beside the primary keys. It would only be necessary if the same entity could have more than one relation with the same entity e.g. if a system administrates a zoo, then the feeding of the tortoises could be done more than once by the same zoo keeper, and a time stamp would be necessary to avoid duplicates. In our system this problem does not occur at any point e.g. a client can not be subscribed twice two the same problem. This applies to all relations.

The three classes who inherits from each other, client, staff, and admin are combined into one entity named roles. This is due to that a person can only have one role, and therefore this representation is more efficient. This also gives a nice modifiability because adding a new role is simply just adding a new tuple to the entity.

## 8.2 Mapping ER-diagram to relational scheme

Our relations will be as follows:

*Problem (problemID, name, description, date, eta, deadline)*

*Person (personID, name, e-mail, username, password, dept_name)* – Note here that we have included `dept_name` in `Person` as we have a many-to-one relation to the `Department` scheme, and therefore the redundant table called `Employment` can be omitted.

*Solution (solutionID, description, date)*

*Department (deptID,dept_name, description)*

*Tag (tagID, name, description, priority, categoryID)* – Note here that we have included `categoryID` as we have a many-to-one relation to the `Category` scheme, and therefore the redundant table called `tag_cat` can be omitted.

*Category (name, description, dept_name)* – Note here that we have included `dept_name` in `Category` as we have a many-to-one relation to the `Department` scheme, and therefore the redundant table called `dept_cat` can be omitted.

*Role (roleID, name)*

*Comment (time, description, problemID, personID)* – Note that we have included `problemID` and `personID` as we have a many-to-one relationship with the `Problem` scheme, and therefore the redundant tables called `com_prob` and

`made_by` can be omitted.

*Assignment (personID, problemID)*

*Subscription (personID, problemID)*

*prob_tag (problemID, tagID)*

*prob_sol (problemID, solutionID, time)*

*per_rol (personID, roleID)*

# Part IV

# Testing

# Bibliography

[1] Google. Install or update google chrome: System requirements. WWW, 2010. URL `http://www.google.com/support/chrome/bin/answer.py?answer=95411`. Last viewed: 16/2.

[2] 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.

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