# Aalborg University

Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

**Title:**

Defect Tracker

**Project period:**

DAT1, Sept. 4th – Dec. 19th, 2002

**Project group:**

E1-114


**Members of the group:**

Anders Rune Jensen
Jasper Kjersgaard Juhl
Janne Larsen
Lau Bech Lauritzen
Ole Laursen
Michael Gade Nielsen

**Supervisor:**

Igor Timko



**Abstract:**

This report documents the development of an administration system, Defect Tracker, for reporting, tracking and managing software defects. It has been developed using the OOA&D method, and implemented with Java beans, server pages and servlets running on a Jakarta Tomcat server.

The study report reflects on the analysis, design and test phases, and look into the algorithmic problem of efficient string matching which is needed by a function in the system for highlighting search words.



**Number of copies:** 8

**Report – number of pages:** 120

**Appendix – number of pages:** 3

**Total amount of pages:** 123

# Preface

This report contains the development documents and study report for the DAT1 semester at Department of Computer Science, Aalborg University.

The development documents describes a functioning defect tracking system written in Java and include analysis and design documents developed according to the principles of the Object-Oriented Analysis and Design method by Mathiassen et al. [10], an implementation document describing the techniques used for implementing the system and a test document that describes our efforts of trying to ensure that the system works correctly.

References to the source code and running system are listed in Appendix A.

*Aalborg, December 2002,*


| | |
|---|---|
| *Lau Bech Lauritzen* | *Jasper Kjersgaard Juhl* |
| *Michael Gade Nielsen* | *Anders Rune Jensen* |
| *Ole Laursen* | *Janne Larsen* |

# Contents


**I Analysis** **9**

**1 Introduction** **11**

1.1 Error management without a defect report system . . . . . . . . . . 11
1.2 Suggested software solution . . . . . . . . . . . . . . . . . . . . . . 13
1.3 System definition . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**2 Problem domain** **17**

2.1 Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.2 Behaviour . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2.1 System . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2.2 Report . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2.3 Comment . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.2.4 User . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.2.5 Developer . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.2.6 Permission . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.3 Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

**3 Application domain** **23**

3.1 Actors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4 User interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.4.1 General patterns . . . . . . . . . . . . . . . . . . . . . . . 35
3.4.2 The user interface templates . . . . . . . . . . . . . . . . 36
3.5 System interfaces . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**II Design** **49**

**4 Presumptions** **51**

4.1 Criteria . . . . . 51
4.2 Technical platform . . . . . 53

**5 Architecture** **55**
5.1 Component architecture . . . . . 55
5.2 Database component . . . . . 57
5.3 Model component . . . . . 57
5.3.1 Common features of the classes . . . . . 57
5.3.2 System . . . . . 59
5.3.3 Report . . . . . 59
5.3.4 Comment . . . . . 60
5.3.5 User . . . . . 61
5.3.6 Permissions . . . . . 61
5.3.7 Database design . . . . . 61
5.3.8 Concurrency . . . . . 63
5.4 Function component . . . . . 63
5.4.1 Notify . . . . . 64
5.4.2 Search . . . . . 64
5.4.3 Statistics . . . . . 64
5.4.4 Highlight . . . . . 65
5.5 Browser client component . . . . . 65
5.5.1 The commonly applied controller/view pattern . . . . . 65
5.5.2 Overview of view and controller components . . . . . 66
5.5.3 Overview of client functions . . . . . 72
5.5.4 Error handling . . . . . 72
5.5.5 Security . . . . . 72

**III Implementation** **73**

**6 Implementation** **75**
6.1 Documentation . . . . . 75
6.2 Unimplemented features . . . . . 75
6.3 Database component . . . . . 76
6.4 Model component . . . . . 77
6.4.1 Obtaining database connectivity . . . . . 77
6.4.2 Ensuring persistence . . . . . 77
6.4.3 Removing persistent objects . . . . . 78
6.4.4 Verifying permissions . . . . . 79

6.5 Function component . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.6 Browser client component . . . . . . . . . . . . . . . . . . . . . . 81
6.6.1 Controllers . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.6.2 Views . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
6.6.3 Client-side . . . . . . . . . . . . . . . . . . . . . . . . . . 87

**IV Test** **93**

**7 Test** **95**
7.1 Unit tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.1.1 Testing *delete* . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.1.2 Test results . . . . . . . . . . . . . . . . . . . . . . . . . . 96
7.2 System tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
7.2.1 Test of reporting a defect . . . . . . . . . . . . . . . . . . . 97
7.2.2 Test results . . . . . . . . . . . . . . . . . . . . . . . . . . 98

**V Study report** **101**

**8 The development process** **103**
8.1 The analysis phase . . . . . . . . . . . . . . . . . . . . . . . . . . 103
8.2 The design phase . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
8.2.1 Designing the user interface . . . . . . . . . . . . . . . . . 104
8.2.2 Combining OOP with relational databases . . . . . . . . . 105
8.3 The test phase . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
8.3.1 Unit testing the user interface . . . . . . . . . . . . . . . . 107
8.3.2 Handling discovered defects . . . . . . . . . . . . . . . . . 107
8.4 Faster highlighting . . . . . . . . . . . . . . . . . . . . . . . . . . 108
8.4.1 The string-matching problem . . . . . . . . . . . . . . . . 108
8.4.2 The simple approach . . . . . . . . . . . . . . . . . . . . . 109
8.4.3 The Knuth-Morris-Pratt algorithm . . . . . . . . . . . . . 109
8.4.4 The Boyer-Moore algorithm . . . . . . . . . . . . . . . . . 112
8.4.5 Comparison of the algorithms . . . . . . . . . . . . . . . . 116
8.4.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . 118

**Bibliography** **119**

**A References to Defect Tracker** **121**
A.1 Running version . . . . . 121
A.1.1 Finding the open defects in our system . . . . . 121
A.2 System demonstration . . . . . 122
A.3 Source code of defect tracker . . . . . 122
A.3.1 Overview of our source code . . . . . 122
A.3.2 Source code of the model component . . . . . 122
A.3.3 Source code of the function component . . . . . 122
A.3.4 Source code of user interface components . . . . . 122
A.3.5 Source code for unit tests . . . . . 122
A.4 JavaDoc . . . . . 122
A.5 Test results . . . . . 123

# Part I

# Analysis

# Chapter 1

# Introduction

This chapter gives an introduction to the problems of managing defects and how these problems can be alleviated.

## 1.1 Error management without a defect report system

A typical way of dealing with defects in commercial applications is the end-user contacting the service center of the company, see Figure 1.1. If the cause of the problem is in fact a software defect, the service center then contacts the appropriate developer through the internal communication procedure, e.g. by email, and the developer adds the issue to his, or his group's, to-do list, perhaps merging it with other reports of the same problem.

Now the developer carries the responsibility of actually identifying and correcting the problem, or assigning the task to another developer. Finally, the error is corrected in the source code, perhaps after some discussion between various devel-

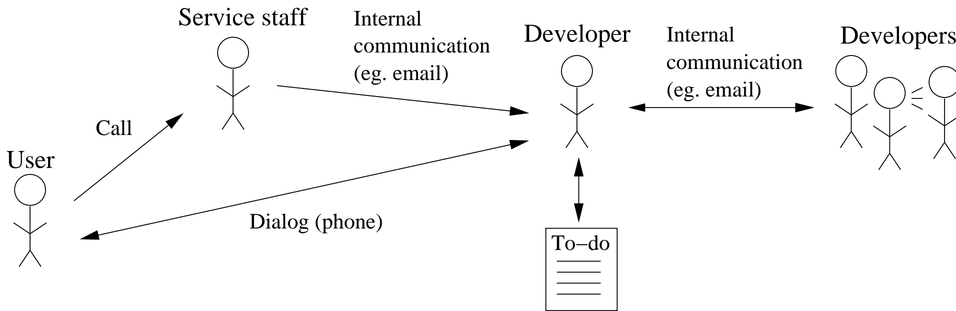



*Figure 1.1: How errors are managed in a company without a specialised defect report system.*

opers (and the user if the defect is difficult to reproduce), and the developer deletes the entry from his list.

Open source projects without a defect report system usually just have an email contact address; when a user finds a defect in the project, the person sends a message to the contact address with relevant information, such as steps to reproduce the problem, version of the software, etc. The appropriate developer then sends back an acknowledgement and adds the defect to the list of pending issues. When the error is corrected later, the developer removes the entry from the list. As with the commercial case, the actual defect diagnosis may require communication between developers and between developers and the user.

The most important problems with these ad-hoc approaches are:

- They do not scale well. One person can easily manage ten or twenty defects with these schemes, but handling a large amount of reports, for instance above 100, requires much trivial work when maintaining the data with an ordinary text editor.

  Similarly, having multiple developers use the same list of issues requires increased communication to ensure that defects are not reported twice, that corrected errors are not entered again, that reports are entered in a uniform fashion to ensure the structure of the list, etc.

- Extending and organising the reports in various ways, e.g. by priority or by a certain module hierarchy, is awkward and laborious since there is no support for manipulating reports as units and no support for manipulating multiple reports at a time.

- Information about a defect tends to get fragmented as individual developers communicate since including the relevant discussions requires manual insertion of e.g. text from email or notes from phone calls.

- There is no easy way of getting an overview of what needs to be done, and by whom. Other statistical processing of the defects, e.g. drawing a graph of the number of defects in the various phases of the project for use by the project manager, is also very difficult.

Essentially, the simple defect lists ensures that defects are not forgotten and not much more. For the initial phases of a project, this may be enough as problems are fixed quickly and everything is fresh in memory so each issue only requires a few short notes. A more elaborate process would just get in the way. However, for the long-term maintainance phases where the difficult to find and difficult to diagnose

errors are discovered there is a need for a system that supports the communication between the involved parties and the processing and storage of defects better.

## 1.2 Suggested software solution

An obvious solution is to let a dedicated software system keep track of the defects. Figure 1.2 shows the situation after such a system has been deployed. Defect reports, which are given pleasant interfaces and managed by the software system, are the central communication medium.

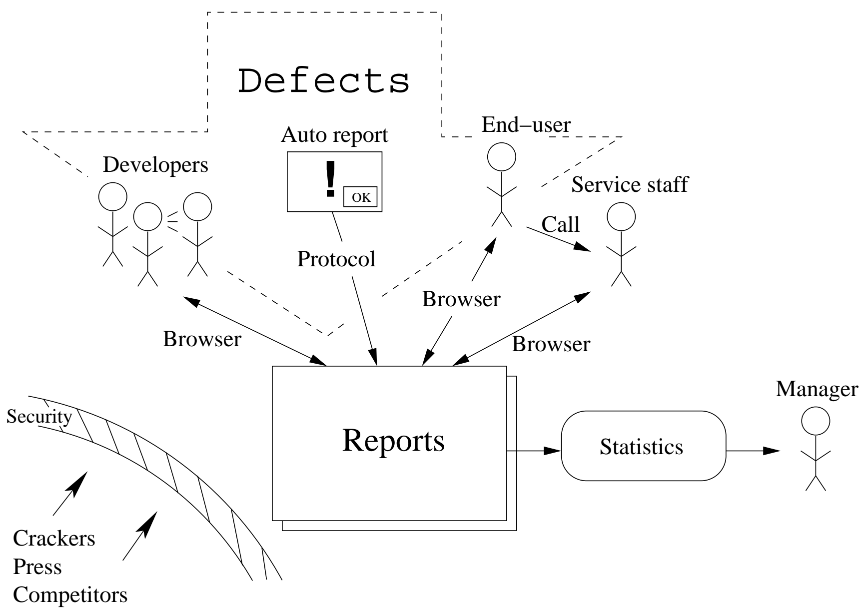



*Figure 1.2: A proposed better way of dealing with defects.*

The illustrated solution gives an end-user two choices:

- A defect can be reported directly to the system in the organization responsible for the software in which the defect occured.
- If however the end-user is more confident explaining the defect person-to-person, a service center can take care of adding the report to the system.

In some software applications the developers may have implemented a defect reporting feature that automatically – with the end-user's approval – reports the defect to the appropriate organization whenever a defect occurs. This feature is however limited to programming flaws that can be caught automatically, such as crashes.

The developers in the organization can also add defects to the defect report system, but their main task is in correcting and managing the reported defects. When a defect is corrected, the reporting user is somehow notified, perhaps with additional information about the next software release to contain the correction. Meanwhile, the developers can use the reporting system for discussing the defect and the user can see that the diagnosis is progressing, perhaps supplying more information if necessary.

As defect-free software releases carry less maintenance costs, the project manager is interested in minimizing the number of defects in future software releases. This can e.g. be achieved by carefully examining which areas have the highest concentration of defects and enhancing the developers' skills in those areas. The response time, from which a defect is reported to corrected, may also be of interest as to assess whether or not it is satisfactory. To assist the project manager in these tasks, the reporting system can provide a statistical overview of the current and previous defects.

It is vital for the organization storing the defects that these are not compromised by random non-authorized users. As a consequence, a number of security measures must be maintained.

## 1.3 System definition

Based on the previous section, the system can be specified as:

> A computerized system to ease the reporting and management of software defects. This is achieved by providing a customized user interface for the different type of users and organizing the defect reports in software systems. The user interface of the system must be simple, yet when reporting a defect the information required to be passed on must be adequate enough for the developers to be able to correct the defect.
>
> Management of the defects involves assigning priority, merging, and basic organization features such as moving and deleting. Furthermore the system incorporates a security model based on user assigned permissions, system responsibility, searching, and statistical overview of reported defects.
>
> The users of the system are expected to be familiar with computer systems in general, but no previous knowledge of defect reporting is expected. The user interface requires a web browser and an Internet

connection to a central web server. The system is intended as a communication medium between end-users and developers and as a repository for information about software defects.

# Chapter 2

# Problem domain

This chapter identifies classes, the relationships between them and their behaviour in the problem domain.

## 2.1 Structure

The problem domain can be modeled by a number of classes as shown in Figure 2.1. Note that a user is considered a user of our system, while an end-user is a person who reports defects.

Starting from the top left corner, the SYSTEM represents a system under development. Each system may contain subsystems; e.g. a compiler system may contain a preprocessor system, a parser system, a code generator system etc. One system is the root. This gives a tree structure where the leaves are systems without any subsystems. A SYSTEM can hold several defect REPORTs, and each report can contain several COMMENTs which are the primary contents.

The class USER represents all users of the Defect Tracker. They can report defects and comment on defects, although some users also have other capabilities. From the class USER, a subclass DEVELOPER is derived since a developer can be responsible for several reports.

The security measures taken are modeled with the abstract class PERMISSION, from which the different kind of concrete permissions are derived. Each user may carry any number of permissions. The READ permission is for searching and browsing a system (which may be the root) and its contents of subsystems, and the WRITE permission is for changing the contents. EDIT REPORT is a limited write permission for individual reports given to the report creator. SHOW STATISTICS grants permission to make the system process the defects to give an overview. The ADMINISTRATION permission allows a user to assign other users the same

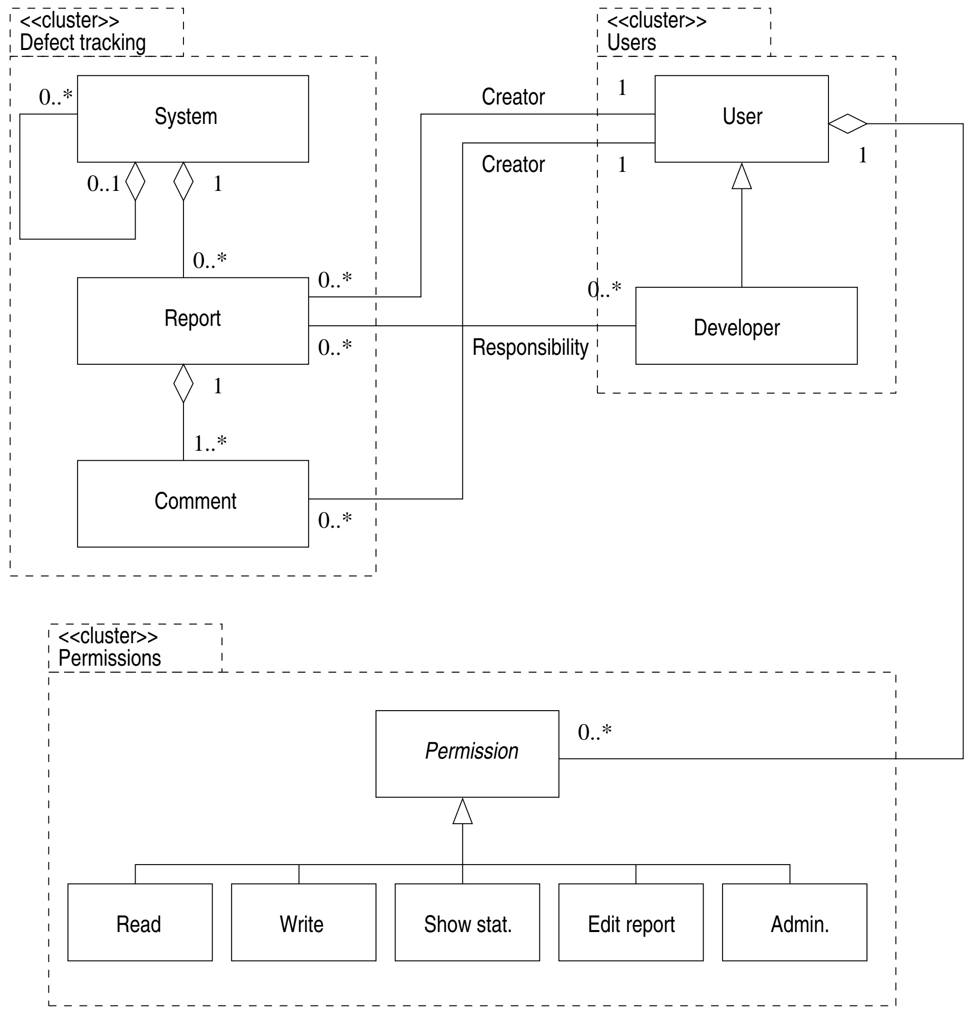



*Figure 2.1: Overview of classes and their relations in the problem domain. Associations from the permission classes to the defect tracking cluster is not shown as it would clutter the overall picture (*READ*,* WRITE *and* SHOW STATISTICS *are each associated with a system,* EDIT REPORT *is associated with a report).*

permissions as himself, and delete users in the system.

## 2.2 Behaviour

The behaviour of the relevant classes will be described by means of statechart diagrams and additional explanations.

### 2.2.1 System

A system is identified by a name and to each system belongs a description. Figure 2.2 illustrates possible states for the class SYSTEM. When a system is started, users will start finding defects and hence attach reports to the system. When these reports are no longer relevant they will be deleted. It is also possible to add and remove subsystems from the system. This is possible while the system is active; when there is no more interest in correcting defects in the system, the system will be put in a inactive state. At some point an inactive system is abandoned.

Attributes: *name* and *description*.

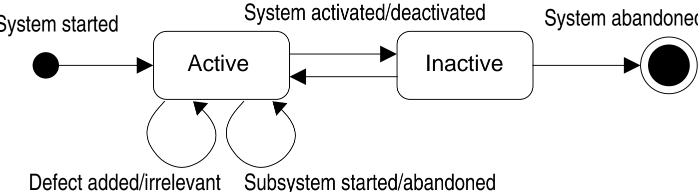



*Figure 2.2: Statechart diagram for the class* SYSTEM.

### 2.2.2 Report

A report is identified through a number. When it is submitted, its priority will start at “low”. The status of the report indicates whether or not the defect has been corrected, i.e. whether it is open or closed. Some reports may be classified and this will also be registered. When a report is registered, a number of things may occur, as illustrated in Figure 2.3.

A user might want to add additional comments to an already existing report. If a defect is assessed to be more or less severe it is possible to change the priority of the report. It is also possible to assign developer responsibilities to the report and later remove them. If two identical reports are submitted they can be merged; the most complete report will get the comments from the other report. It will in addition be registered how many times this defect has been reported. The subject

of the report can be changed should its content be re-assessed. A report can be closed if the defect is corrected and reopened if it is refound. When the defect is no longer relevant it is deleted.

Attributes: *number*, *priority*, *number of times reported* and *subject*.

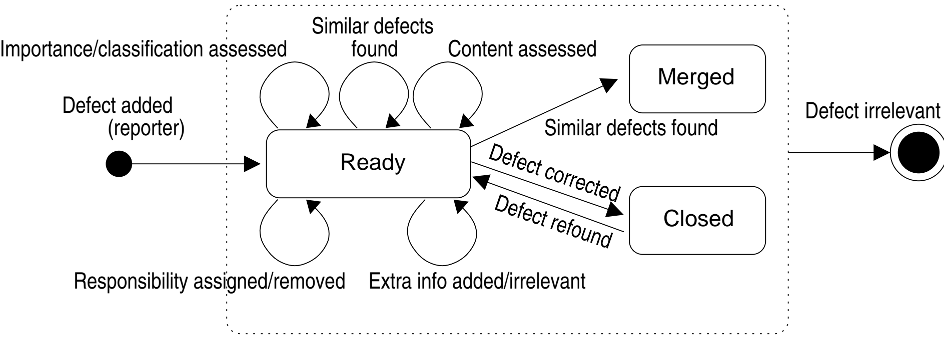



*Figure 2.3: Statechart diagram for the class* REPORT.

### 2.2.3 Comment

A comment contains a piece of information, and as illustrated in Figure 2.4, when a user or developer discovers additional information about a defect, he is able to comment on an existing report. When a defect is registered, one comment is always created as the defect description.

Attributes: *body*.

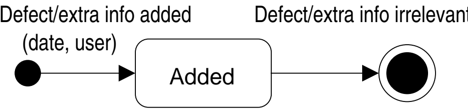



*Figure 2.4: Statechart diagram for the class* COMMENT.

### 2.2.4 User

A user is identified through his username. A registered user can register defects and add additional comments to reports. It is possible to change information about a user if, e.g., his name or address changes. A statechart diagram of the class USER is illustrated in Figure 2.5.

Attributes: *username*, *password*, *name*, *email address* and *whether to notify*.

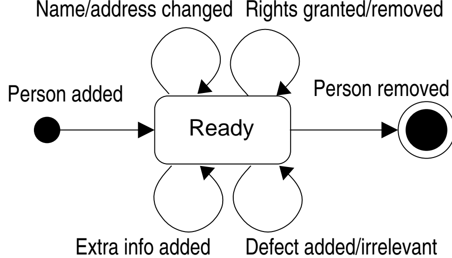



*Figure 2.5: Statechart diagram for the class* USER.

### 2.2.5 Developer

A developer inherits a username, password, name, email address and whether to be notified from the class USER. A developer can be made responsible for reports.

This is illustrated as a statechart diagram in Figure 2.6.

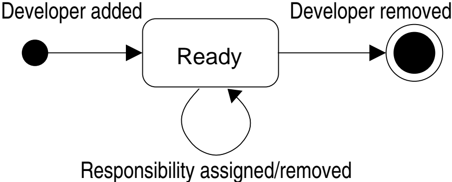



*Figure 2.6: Statechart diagram for the class* DEVELOPER. *The events that are derived from* USER *are not shown to prevent cluttering the picture.*

### 2.2.6 Permission

When rights are granted and removed, the various classes derived from PERMISSION are simply created and deleted. Figure 2.7 shows the situation with the permission base class.

*Figure 2.7: Statechart diagram for the class* PERMISSION.

## 2.3 Events

The events and classes are summarized in Table 2.1. Note that the user class in the problem domain is not like the user concept in the application domain and

consequently participates only in some events.

| *Events* | *System* | *Report* | *Comment* | *User* | *Developer* | *Permission* |
|---|---|---|---|---|---|---|
| System started/abandoned | + | | | | | |
| Subsystem started/abandoned | * | | | | | |
| System activated/deactivated | + | | | | | |
| Defect added/irrelevant | * | + | + | * | * | |
| Defect corrected/refound | * | + | | | | |
| Extra information added/irrelevant | | * | + | * | * | |
| Similar defects found | | * | | | | |
| Importance/classification assessed | | * | | | | |
| Person added/removed | | | | + | | |
| Name/address changed | | | | * | * | |
| Rights granted/revoked | | | | * | * | + |
| Developer added/removed | | | * | | + | |
| Responsibility assigned/removed | | * | | | + | |

*Table 2.1: Events and classes. A '+' indicates that the event occurs zero or one time for an instance of a class whereas an '*' indicates that an event may occur zero or several times.*

# Chapter 3

# Application domain

This chapter defines the requirements for the external behaviour of the system.

## 3.1 Actors

The actors in the application domain include developers, managers, end-users, automatic defect reporters, and administrators. Service center staff at a call center are considered end-users since they also report defects.

**Developers** The developers create and develop new systems, and correct and maintain the defects. They have a good understanding of the process of dealing with system organization and defects.

**Managers** Manager want to make the developers work more efficiently, e.g. by improving how they are organized or making them focus on their weak spots. Managers of software projects can be characterized as having at least a basic understanding of the defects and how software development works.

**Users** Users use the systems and report any defects that disturb their usage. They may not be familiar with the various defect concepts (e.g. severity) and how to make good reports, but it is supposed that they do have a basic computer knowledge since they have been using a piece of software where the defect has occurred.

**Automatic reporting systems** The purpose of the automatic reporting systems is to provide an easy way of reporting defects that can be detected automatically, e.g. crashes. They communicate with the defect reporting system using a separate interface.

**Administrators** Administrators manages the defect reporting system itself. For instance, an administrator may make it so that all users can search in the reported defects. As administrators they are familiar with system administration.

## 3.2 Use cases

The various use cases for interaction with the system are presented with general illustrations for overview and more detailed descriptions.

**Login**

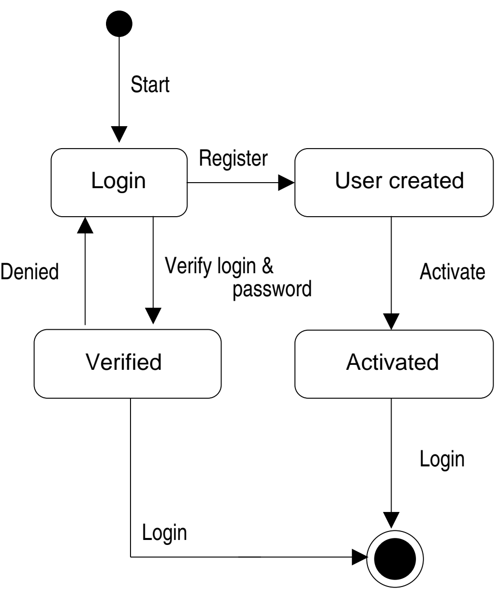



*Figure 3.1: Use case for logging into the system.*

As illustrated in Figure 3.1, when the system starts, the actor is required to login. If the actor has an account in the system, he enters and submits the appropriate username and password. The system grants him access if his username and password are valid. If not, access will be denied and the actor can try again. However, if the actor does not have an account he is able to create one. After creating and activating an account the actor is automatically logged in. Initially, it is very limited what a newly created user can do, so it is the administrators' responsibility to provide the user account with the appropriate permissions.

The *logout* function is available in all use cases once the actor is logged in, hence it is not explicitly mentioned in the following use cases.

Actors: *developer*, *manager*, *user* and *administrator*.

Objects: *user* and *developer*.

Functions: *login*, *activate user*, *send activation email* and *create user*.

### Manage report

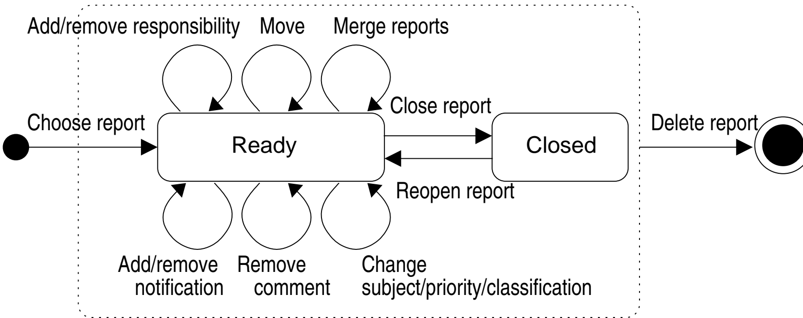



*Figure 3.2: Use case for managing reports.*

As illustrated in Figure 3.2, after choosing a report there are many options. A developer can become responsible for a defect, and any actor can make the system notify him when the report is changed. Moving the report is possible by specifying the destination system. When merging reports the actor must select a master report which the current report will be merged with. After a defect is corrected, it can be put it in a closed state, with the possibility of reopening it if needed. An irrelevant report can be deleted.

Actors: *developer* and *user*.

Objects: *system*, *report*, *permissions*, *developer*, *user* and *comment*.

Functions: *show report*, *show comment*, *merge reports*, *remove comment*, *change report attributes*, *add/remove responsibility*, *delete report*, *move report*, *close/re-open report*, *check permission* and *notify*.

### Manage system

As Figure 3.3 illustrates, after choosing a system the actor is able to edit attributes. This includes changing the name and the description of the system. The actor can delete and move a number of sub-systems and must confirm it and specify a new location, respectively. When creating a new system the actor must fill in the appropriate information. The actor can browse the system structure by choosing a new system or going one level up – as in a file system.

Actors: *developer*.

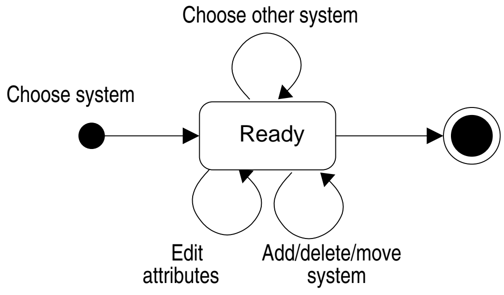



*Figure 3.3: Use case for managing systems.*

Objects: *system*, *permissions*, *report* and *comment*.

Functions: *show system structure*, *show system*, *change system attributes*, *create/delete system*, *move system* and *check permission*.

### Statistics

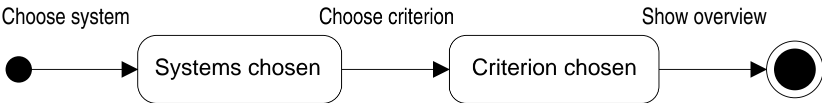



*Figure 3.4: Use case for generating statistics.*

As illustrated in Figure 3.4, the actor must choose a system. Once a system is selected all subsystems are also selected and the actor must enter the criterion that should be used to generate the statistics. When the criterion has be chosen the statistics are displayed.

Actors: *manager*.

Objects: *system*, *report*, *permissions* and *user*.

Functions: *show system structure*, *generate statistics* and *check permission*.

### Report defect

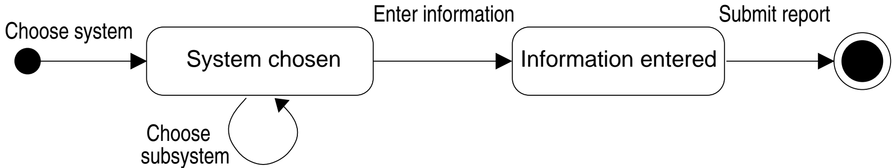



*Figure 3.5: Use case for reporting defects.*

As Figure 3.5 illustrates, in the process of reporting a defect an appropriate system is first chosen. When this is done, the information regarding the defect is filled in and submitted.

Actors: *developer*, *user* and *auto reporter*.

Objects: *system*, *report*, *comment*, *user* and *developer*.

Functions: *show system structure*, *create report* and *add auto report*.

### Edit user attributes

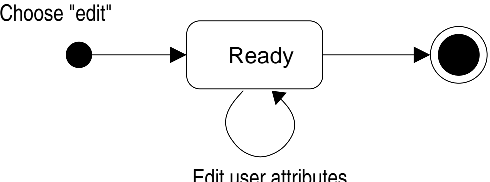



*Figure 3.6: Use case for editing user attributes.*

Editing of user attributes can be done by any actor who is logged into the system. As illustrated in Figure 3.6, the actor is able to modify some of his personal information, such as name, password and email address. When satisfied with the modifications the actor can leave this section.

Actors: *developer*, *user* and *administrator*.

Objects: *user*.

Functions: *show user* and *change user attributes*.

### Search

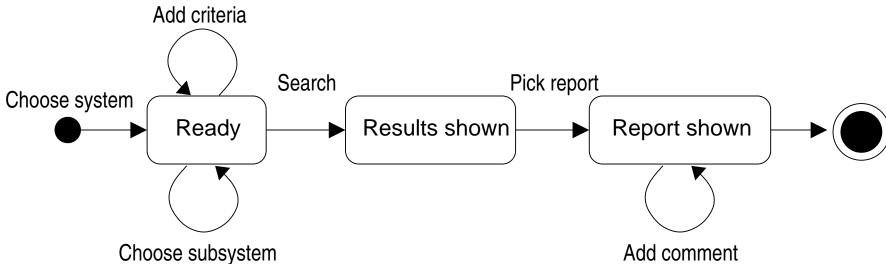



*Figure 3.7: Use case for searching.*

As illustrated in Figure 3.7, searching the defect database is done by choosing the system containing the desired defect and supplying a number of criteria consisting of status, priority, age, subject, comment, and creator. After submitting the

search, the results are shown. It is additionally possible to add a comment to the shown report.

Actors: *developer*, *manager* and *user*.

Objects: *system*, *report*, *permissions*, *comment*, *user* and *developer*.

Functions: *show system structure*, *show report*, *show comment*, *highlight*, *search*, *add comment* and *check permission*.

### Manage users

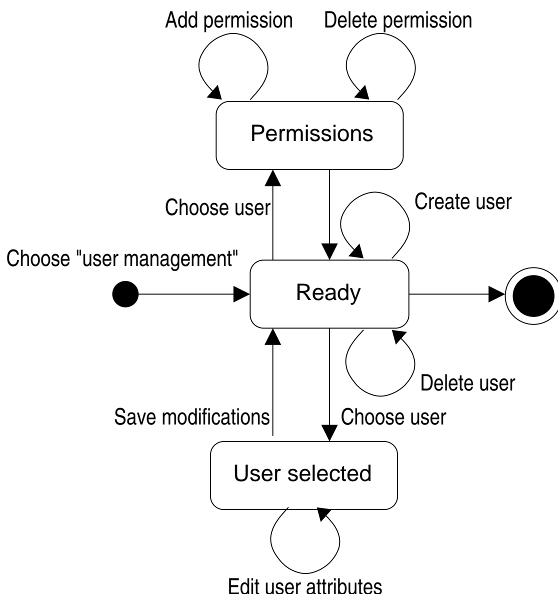



*Figure 3.8: Use case for managing users.*

As illustrated in Figure 3.8, entering the manage users section puts the system in a state making it possible for the actor to choose a user. In addition to the previous “Edit user attributes”, a permission modification option is available. This option is used to grant and delete permissions from users of the system. When the user is modified the actor is sent back to the previous state, ready for future user modifications.

Actors: *administrator*.

Objects: *user*, *developer* and *permissions*.

Functions: *show user*, *add/remove permission*, *create/delete user*, *change user attributes* and *check permission*.

**Summary**

The various use cases are summarized in Table 3.1.

| *Use case* | Developer | Manager | User | Auto reporter | Admin. |
|---|---|---|---|---|---|
| Login | + | + | + | | + |
| Report defect | + | | + | + | |
| Search | + | + | + | | |
| Statistics | | + | | | |
| Edit user attributes | + | | + | | + |
| Manage users | | | | | + |
| Manage report | + | | + | | |
| Manage system | + | | | | |

*Table 3.1: Summary of the actors' participations in use cases.*

## 3.3 Functions

This section describes the functions in the system along with their complexity and type, as listed in Table 3.2. The complexity of the functions are determined by estimating the effort needed for development.

- **Login**

  Since different users will use the system, the login function will enable the system to identify each user by providing him with a session. This function also handles verification of the submitted username and password, and determines if the user is granted or denied access to the system.

- **Logout**

  This function will end the current user's session.

- **Create/delete user**

  The create user function is used if a new user reports a defect or a new developer is created. All users can be deleted with the delete user function.

- **Create/delete report**

  When a user reports a new defect the create report function is used. The report and all its comments can be removed with the delete report function.

- **Create/delete system**

  The create function creates a new system as a subsystem. The delete system function removes a system and its subsystems.

- **Add/remove comment**

  These functions are used when a comment is added or removed from an existing report.

- **Add/remove permission**

  When a permission is added to a user or revoked, e.g. if the user is given permission to modify a specific system, these functions are used.

- **Add/remove responsibility**

  These functions are used when a developer becomes responsible for a report, or that responsibility is revoked.

- **Add auto report**

  If implemented in the application, this function is used whenever a defect occurs and ensures that the defect is automatically reported to the specified defect reporting system. The defect is communicated through the Internet to the defect reporting system's server using the HTTP protocol.

  The value of the automatically reported defect is entirely up to the implementers of the monitored application, as they are responsible for providing the appropriate mechanisms that gather together the details of a caught defect.

- **Change user attributes**

  This function is used when a user's attributes have to be modified, e.g. if the user's email address changes.

- **Change report attributes**

  If a report's attributes have to be modified this function is used, e.g. if the status or priority changes.

- **Change system attributes**

  This function is used to modify the name and description of a system.

- **Move system**

  Moves a system (and its subsystems) so that it becomes a subsystem of another system.

- **Move report**

  Moves a report to another specified location. The comments of the report are relocated as well.

- **Show system structure**

  This function is used to retrieve information about the system structure when browsing it.

- **Show report**

  This function retrieves information about a report, such as priority and subject.

- **Show user**

  This function retrieves information about a user, such as name and username.

- **Show system**

  This function retrieves information about a system, such as name and description.

- **Show comment**

  This function retrieves the body attribute of a comment.

- **Generate statistics**

  This function is used to compute a number of different statistics. When given a system this function can compute the number of defects in all subsystems, the number of defects per developer in all subsystems, the response time of all subsystems and the time line of the system. This function is estimated as very complex because of its many different computations and how they should be displayed.

- **Search**

  This function is used to search for reports and is estimated as very complex because of the amount of different criteria possibly resulting in search through multiple database tables.

- **Close/reopen report**

  This function changes the status of a report from closed to open or from open to closed.

- **Merge reports**

  This function takes one report and merge its comments with another report, the master report. It is estimated as complex because a proper way of merging the comments must be figured out and the appropriate permissions must be checked.

- **Notify**

  When a defect is updated this function is used to notify all the involved users. Only those users who have chosen to be notified will receive an email, hence the function must determine the recipients and format the message with appropriate hyperlinks back to the system. Furthermore the function must be able to communicate with an SMTP server. As a consequence of these requirements the function is estimated as very complex.

- **Highlight**

  When a search has been submitted this function takes care of highlighting keywords corresponding to the search query.

- **Send activation email**

  When a user has been created in the system, he is required to activate his account before gaining access to the system. This function properly formats and delivers an email to the user's specified email address.

- **Activate user**

  This function is used to activate the user, allowing him access to the system.

- **Check permission**

  Users must have permission to perform various tasks, e.g. merging two reports requires write and read permission to both reports. This function makes sure that the user indeed has permission to perform the selected task.

- **Add/remove notification**

  Adds or removes the notification of changes for a user for a given report.

## 3.4 User interfaces

Figure 3.9 provides a navigation diagram of the pages in the system. The following sections present some patterns used followed by illustrations of all of the pages (except very minor ones, such as error pop-ups) of the system.

| Functions | Complexity | Type |
|---|---|---|
| Login | Medium | Compute |
| Logout | Simple | Compute |
| Send activation email | Medium | Signal |
| Activate user | Simple | Update |
| Create/delete user | Simple | Update |
| Change user attributes | Simple | Update |
| Show user | Simple | Read |
| Create/delete report | Medium | Update |
| Add auto report | Medium | Update |
| Change report attributes | Simple | Update |
| Merge reports | Complex | Update |
| Close/reopen report | Simple | Update |
| Add/remove notification | Simple | Update |
| Move report | Simple | Update |
| Show report | Simple | Read |
| Create/delete system | Simple | Update |
| Change system attributes | Simple | Update |
| Move system | Medium | Update |
| Show system structure | Simple | Read |
| Show system | Simple | Read |
| Add/remove comment | Simple | Update |
| Show comment | Simple | Read |
| Add/remove permission | Simple | Update |
| Add/remove responsibility | Simple | Update |
| Generate statistics | Very complex | Compute |
| Search | Very complex | Compute |
| Notify | Very complex | Signal |
| Highlight | Medium | Compute |
| Check permission | Simple | Read |

*Table 3.2: Table of functions along with their complexity and type.*

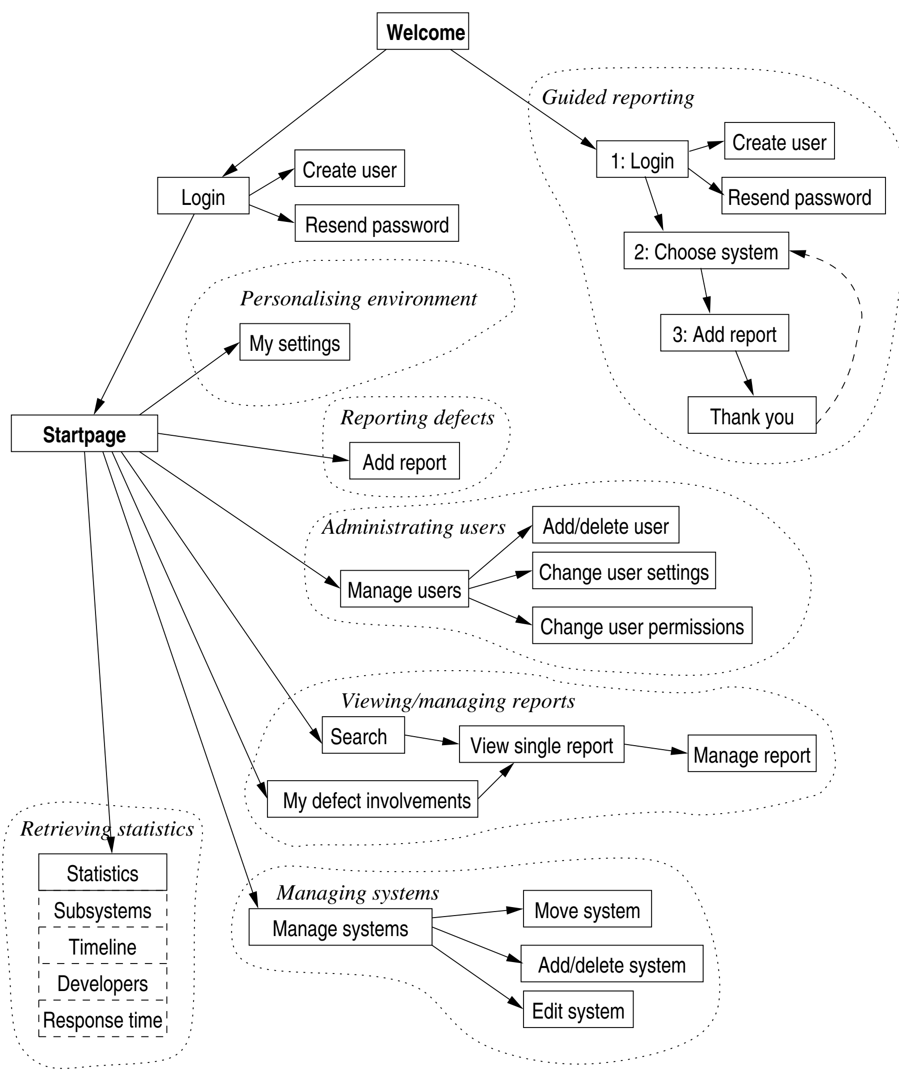



*Figure 3.9: Overview of the pages in the user interface. The arrows show the primary means of navigation.*

### 3.4.1 General patterns

In the process of constructing the user interface for the system, some patterns have been developed to give a consistent look and feel.

#### Navigational helpers

To make navigation of the system easy, a menu is included to the right on all pages that are not pop-ups. It contains links to My Involvements, Add Report, Search, My settings, Log Off, and optionally also Manage Systems, Manage Users, Statistics if the user has the necessary permissions.

The menu is assisted by a text at the top of each page that indicates where the page is located logically. It is labeled “You are here:” and consists of hyperlinks enabling the actor to step back through his previous actions in a specific task.

Note that these navigational helpers are not shown in the page illustrations in the next section.

#### Error handling

When an error occurs, the actor is informed in an orderly fashion. All input fields that are required to be filled in are checked. If any of the fields is empty, an error message is printed at the top of the page below the headline asking the actor to type in the missing information. The error text is shown with a bright red font. The fields which contains the error are also marked with this color.

#### Quickly adding reports

To increase the usability of the system for casual users, a special section is available from the front page with a guide that takes the actor stepwise through each part of adding a report. The guide consists of three steps:

1. Logging in, possibly creating the user if needed.

2. Choosing the right system for the report.

3. Entering information about the defect.

When this process is completed the actor is then prompted if he would like to commit another defect or log off the system. The step number is shown at the top of the page (e.g. “Step 2 of 3”). This guidance insures that the actor is aware of his position and finishes the intended action.

#### Highlighting of search words

To make the search functionality easier to comprehend, the search words are highlighted when a search has been performed. This is used both when displaying the search results and when viewing a report. Inspiration for this functionality comes from the Google search engine.

#### The system tree

To make manipulation of system hierarchies easy, the systems are shown in a tree with plus and minus signs for expanding and collapsing subtrees, much like the folder tree in the file manager in Microsoft Windows. The tree is used in selections of a system (for adding reports), in editing of permissions and in editing of the system hierarchy itself.

#### Buttons

All buttons follow the same pattern to help predictability. The button that accepts and continues the current action is placed to the far right whereas the back or cancel button is placed to the left. On most pages the actor has two choises; either to submit or to go back. To visualize this pattern “«” is displayed on back buttons and “»” is displayed on submit buttons.

The labels of buttons that evoke pop-up windows end with ellipsises (e.g. “Browse…”).

### 3.4.2 The user interface templates

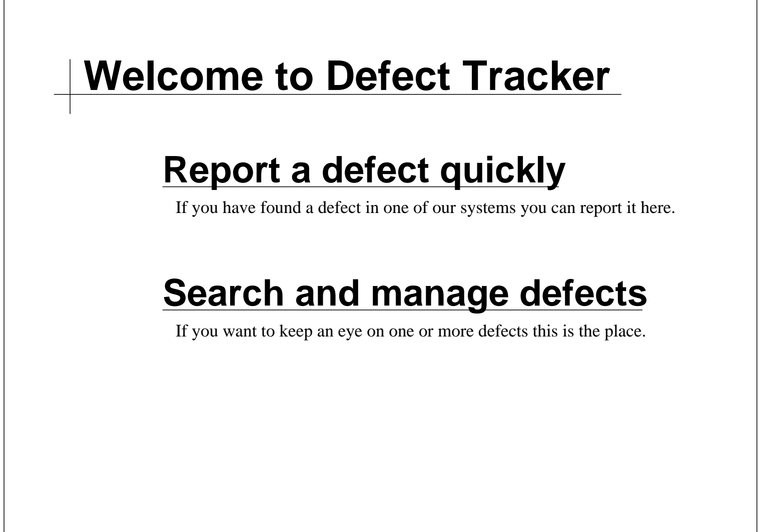



*Figure 3.10: The purpose of the welcome page is to help people end up the right place; casual users will usually choose the "Search and manage defects" option, or simply bookmark the start page that it leads to.*

*Figure 3.11: The login page is needed to authorize users for the session. Every actor needs to pass through it before it is possible to use the system.*

Create new user

Name: Freddy Nilson

E–mail: freddy@cs.auc.dk

Username: Freddy

Password: **********

Retype password: **********

<< Back Create >>

*Figure 3.12: The purpose of the create user page is to create the actor as a new user of the system. Submitting the information causes a confirmation message to be sent by email before the user is activated. It is also possible to access the page from within the system by an administrator; then user is activated right away.*

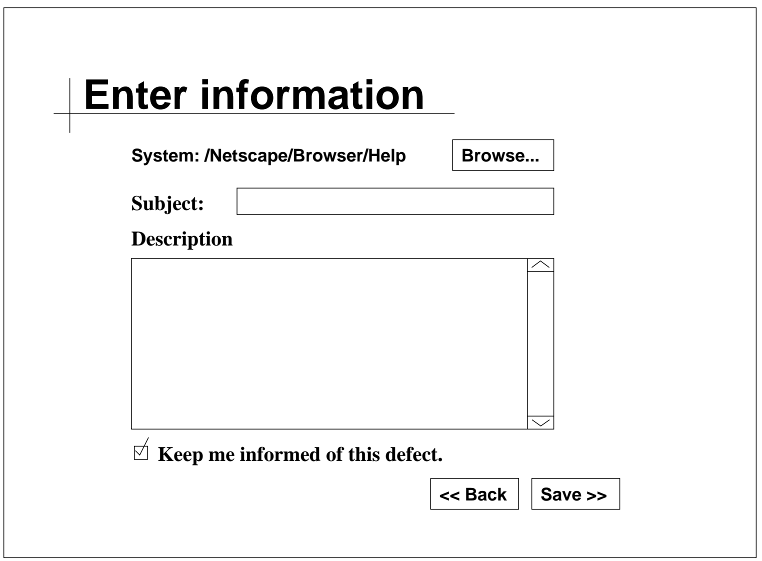



*Figure 3.13: The add report page prompts for the information necessary to report a defect.*

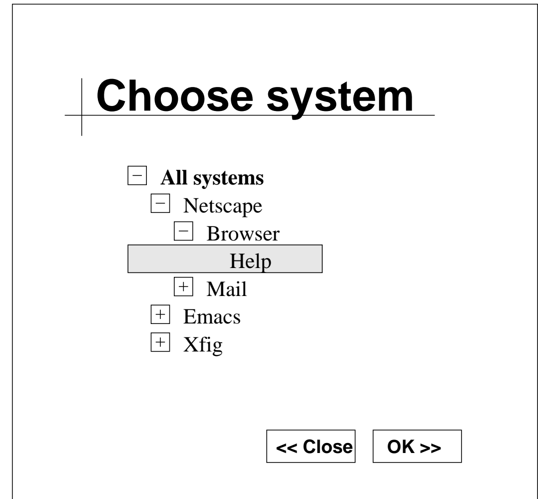



*Figure 3.14: The choose system screen (here shown in the pop-up version) makes it possible to choose a system. This is among others used when adding a report. The systems are displayed using the tree pattern.*

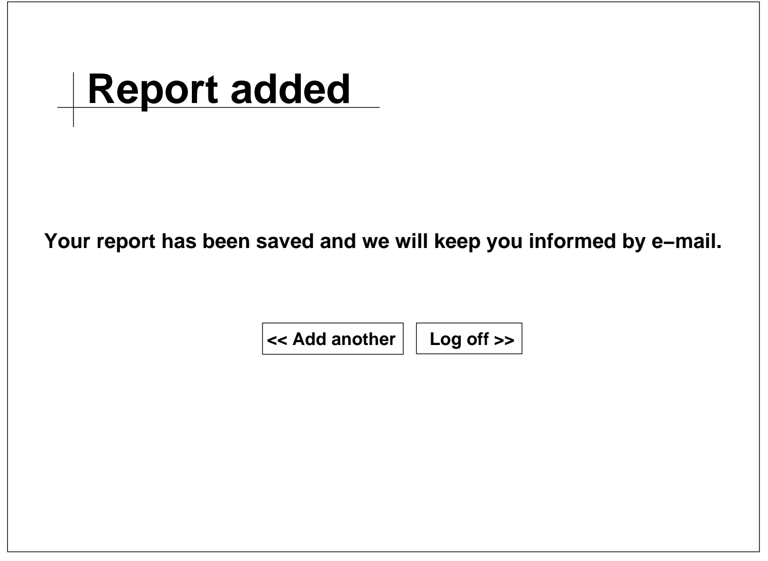



*Figure 3.15: When a defect is reported the actor is asked if he wants to submit another report or log off.*

*Figure 3.16: The purpose of startpage is to show and explain the different options that the actor's permissions allow.*

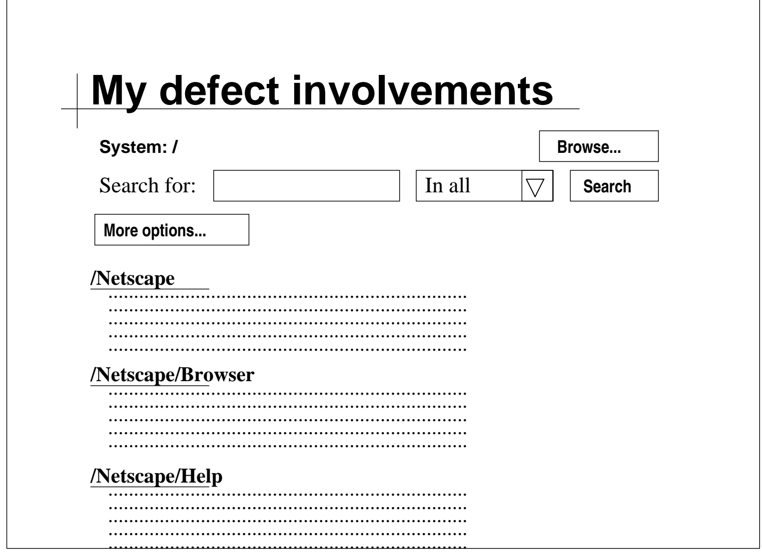



*Figure 3.17: My defect involvements is designed to give a good overview of the reports that the actor is affiliated with. Selecting one of the defects will forward the actor to the view report page.*

**Search for defect**

**System: /Netscape/Browser/Help** Browse...

Search for: [ ] In all ▽ Search

Subject
Contents

More options...

**Has recieved comments:**

From: YYYY–MM–DD

To: YYYY–MM–DD

**Person:** Frode

**Classification:**
- ◉ All
- ○ Unclassified
- ○ Classified

**Status:**
- ◉ All
- ○ Open
- ○ Closed

**Priority**

Low ☐ Medium ☐ High ☑ Critical ☑

**Results – 2 found**

| ID | Subject | Priority | Reporter |
|---|---|---|---|
| **/Netscape/Browser** | | | |
| 602 | Crash on exit | High | Feddy |
| **/Netscape/Browser/URL Handler** | | | |
| 1208 | Button not working | Medium | Freddy |

*Figure 3.18: The search page is used for the actor to search the defect database for a specific report or reports matching some criteria. The bottom picture shows the search results which are concatenated to the search page.*

**View report**

**System:** /Netscape/Browser/Help
**Number:** 1234
**Subject:**
**Reporter:** Freddy
**Developer:** keld, uffe
**Status:** Open
**Priority:** High
**Classified:** No

☑ Keep me informed of this defect.

2002–02–20 Frode

2002–02–16 Freddy

**Add new comment:**

**<< Back** **Add comment** **Manage this report >>**

*Figure 3.19: The purpose of view report is to show the information of a chosen report. A user can choose to be informed by email, if the information of the report changes. It is also possible for the user to add comments to the report, and a developer with the right permissions can go to the manage report page.*

# Manage report

**System:** /Netscape/browser/help [Move...]

**Report ID:** 1234

**Subject:** Spelling error

**Reporter:** Freddy

**Developer:** keld, uffe **Myself** ☑

**Status:** Open ▽

**Priority:** Low ▽

**Classified:** ☑

**Merge with:**

**Comments:**

| Summary | Reporter | Date | |
|---|---|---|---|
| | Freddy | 20–02–02 | |
| | Frode | 16–02–02 | Del |

Back | Delete this report | Reset | Save

*Figure 3.20: The manage report page is used by a developer to manage a report. He can assign himself to the current report, close, reopen or delete it. If the developer wants to merge the report with another, it is possible to do directly by entering a report id. The first comment is the body on the report and cannot be deleted.*

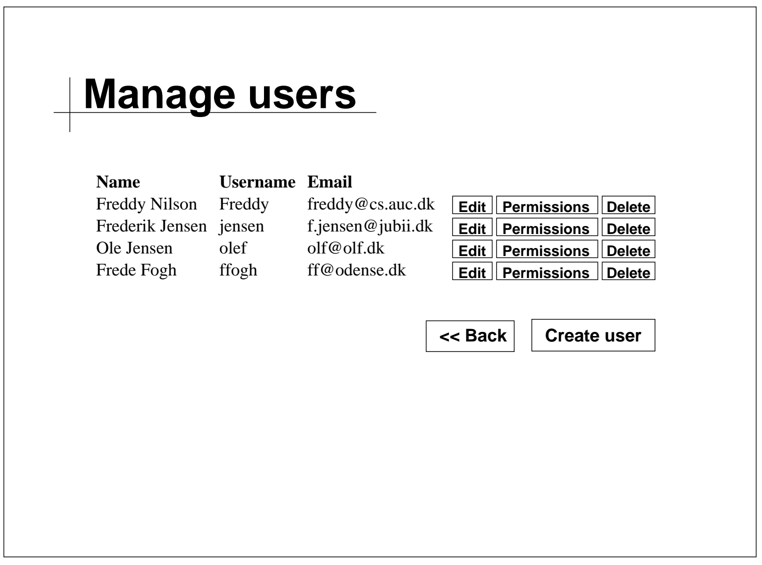



*Figure 3.21: From the manage users page it is possible to enter a page to edit a user's attributes (this page is not shown, but it looks similar to the create user page), and to enter a page to change permissions of a user. Users can also be deleted.*

**Modify permissions for freddy**

**Administrator** ☑

**Read permissions:** Edit...

/xfig Del

/Netscape/browser Del

**Write permissions:** Edit...

/Netscape Del

**Show statistic permissions:** Edit...

/Netscape Del

<< Back

*Figure 3.22: The modify permissions page provides options for an administrator to add and delete a users' permissions. Deleting can be directly, whereas editing opens the permission tree pop-up for the permission types that bind themselves to a system.*

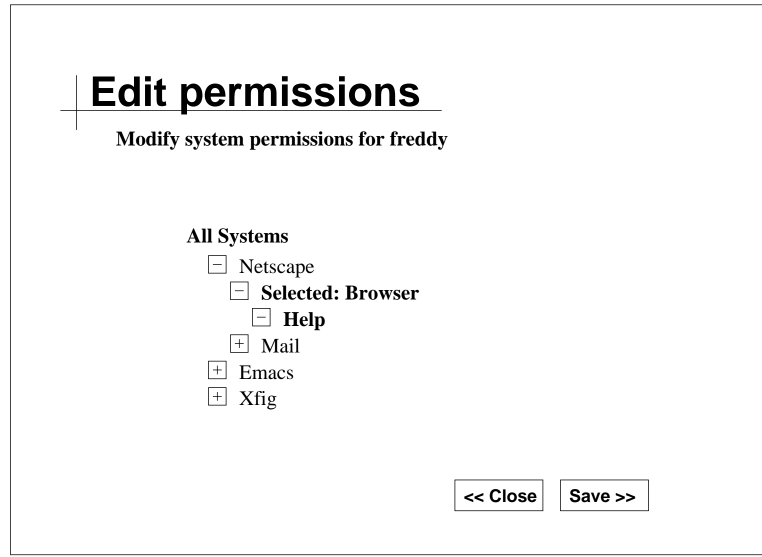



*Figure 3.23: The permission tree makes it possible to modify a given type of permissions for a user by selecting and deselecting systems in the tree.*

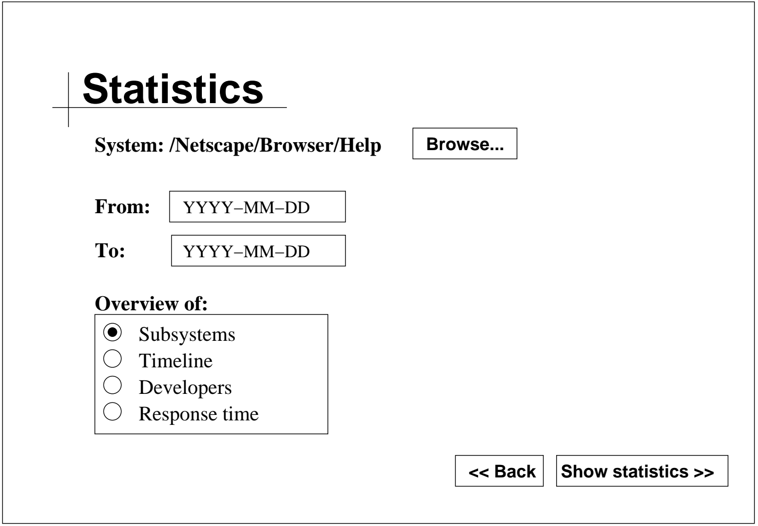



*Figure 3.24: These statistics lets a project manager generate an overview of a system using four different parameters: developers, response time, subsystem and time line.*

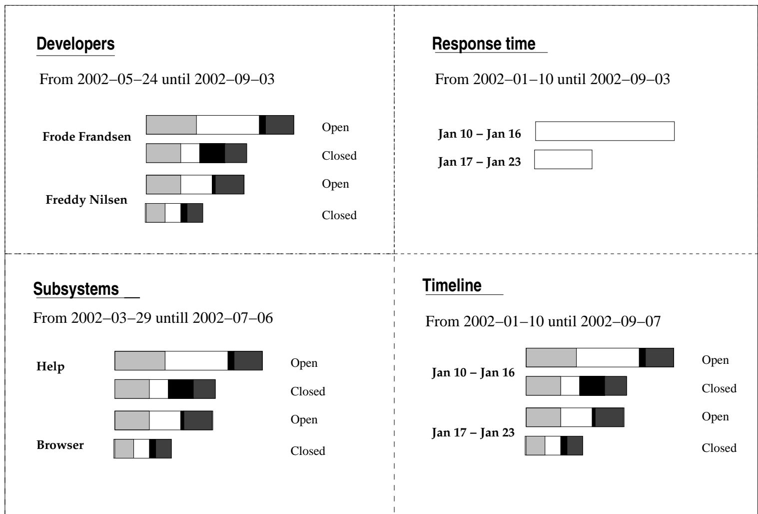



*Figure 3.25: How the result for the categories look like. The different kinds of gray illustrate the priority of the reports, e.g. dark gray indicates critical reports.*

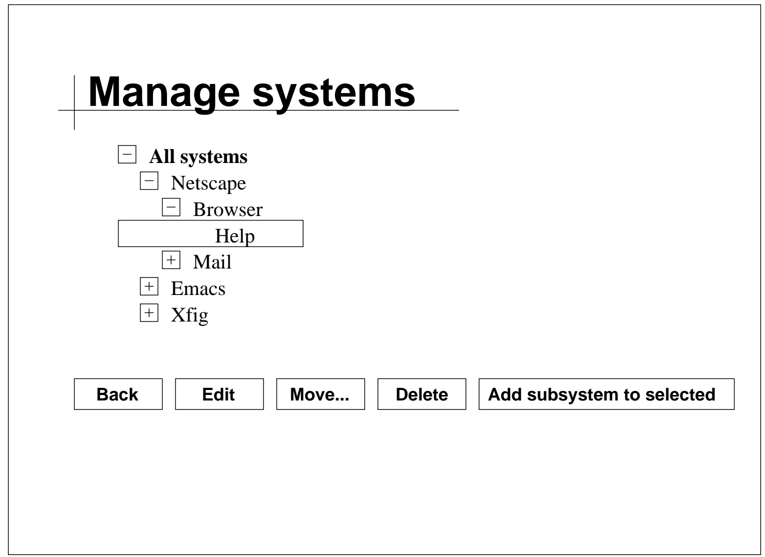



*Figure 3.26: Edit system structure allows modifying the system structure by selecting the appropriate system from the tree. A number of actions can be performed on the selected system: a subsystem can be added, the attributes the system can be edited, the system can be moved and it can be deleted.*

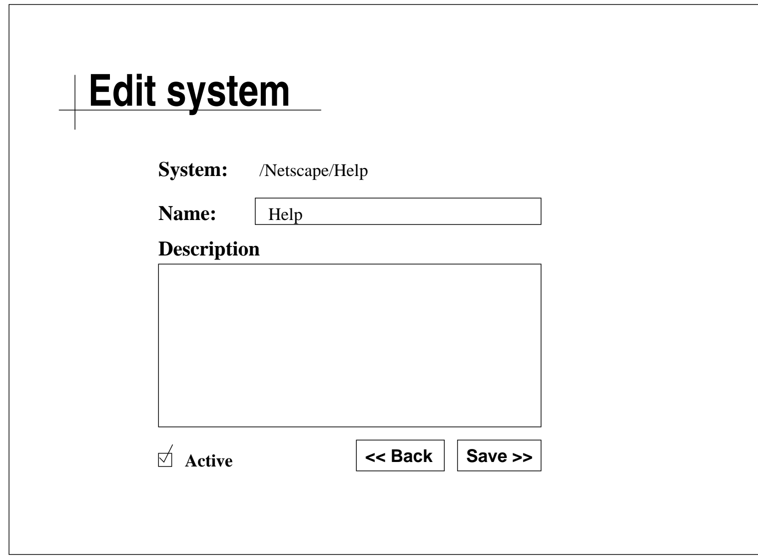



*Figure 3.27: Edit system allows the actor to change the information related to a specific system. Changing the systems state (active/inactive) is also possible. A page similar to this is also shown when a new subsystem is created.*

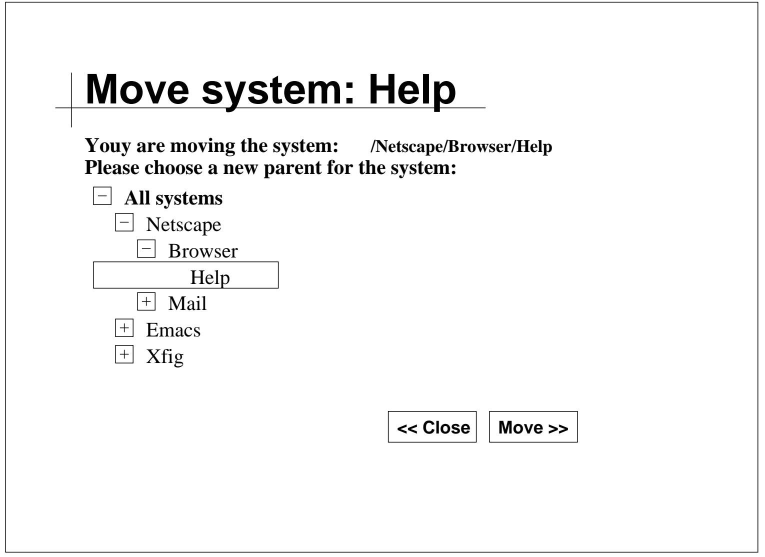



*Figure 3.28: Move system. In advance a system has been selected, consequently the task is to select the desired new parent. When this is done, clicking "Move" will apply the changes.*

## 3.5 System interfaces

There is only one system interface: the interface for automatic defect reporting applications. The interface only supports adding occurred defects; the system name, subject and description must be transfered. The HTTP protocol is used for transmission of information – all relevant information can be encoded in the URL:

```
http://www...?subject=Autoerror&body=Assertionerror:line40&system=...
```

# Part II

# Design

# Chapter 4

# Presumptions

This chapter describes some of the preliminary decisions that provide the basis for the design.

## 4.1 Criteria

Table 4.1 gives an overview of how important various design criteria are assessed for the system.

| Criterion | Very imp. | Imp. | Less imp. | Irrel. | Trivial |
|---|---|---|---|---|---|
| Usable | x | | | | |
| Secure | | x | | | |
| Efficient | | x | | | |
| Correct | | x | | | |
| Reliable | | | x | | |
| Maintainable | | x | | | |
| Comprehensible | | x | | | |
| Testable | | | x | | |
| Flexible | | | x | | |
| Reuseable | | | | x | |
| Portable (frontend) | x | | | | |
| Portable (backend) | | | x | | |
| Inter-operable | | | | x | |

*Table 4.1: Table of design criteria and their relative importance.*

**Usable** The system must be usable to be useful for the users, i.e. the user interface

must be easy to learn for new users, while still being sufficiently advanced to fulfill the needs of the developers with respect to defect reporting and tracking. This is the most important goal of the system.

**Secure** Since a software development company may want to keep some of the information that the system handles to itself, it is important that the system can keep track of which users are entitled to use the various parts of the system and prevent users from accessing the parts that they are not authorized to view or change. The permission class hierarchy from the analysis is part of this effort; another part is to ensure that these permissions are always checked before any operations are performed.

**Efficient** This criterion is important since developers may need to spend a considerable amount of their daily work in the system. Also the system may contain a high amount of defects and users, where efficient algorithms is very important.

**Correct** It is very important that the system fulfills the basic needs of the users to be usable, e.g. that it maintains security and organises the defect reports well. Minor features (e.g. moving systems and reports) are however less important, so overall correctness is just deemed important.

**Reliable** It is very important that the security related part of the system is reliable so that user rights are not confused. However, if the system should be down for some time, the developers can always revert to other communication media such as email. Thus the reliability is not critical and is rated less important compared to the other criteria.

**Maintainable** It is important that the cost of maintaining the system is not too high because the system is intended to be deployed for several years and be part of the developers daily life. It must not be too costly to correct annoying errors.

**Comprehensible** Since the system should be usable and maintainable it is important that it is comprehensible. On the other hand, it is not a very important criterion since end-users will usually just need to use a small part of the interface (the reporting interface), and developers will have plenty of time to learn the system since they are going to be using it a lot.

**Testable** Since the reliability is considered less important, it is also less important to reduce the cost of testing all of the system.

**Flexible** The cost of adjusting the system after it has been deployed is less important since the method of handling defects hardly changes over time.

**Reusable** This criterion is rated irrelevant for the system since the basic idea of the system is to specialize a very general communication medium, an online forum, into something that is better suited for defect reporting and tracking. The specialization is supposed to make the system easier to comprehend and easier to use. Thus it does not make sense to try to make it more general and reusable.

**Portable (frontend)** It is very important that the interface is portable since the various reporters and developers may use different platforms.

**Portable (backend)** It is however less important that the system backend can be moved from one technical platform to another. To ensure such portability it would be necessary to do a thorough research of database management systems and test the system on all these. The costs associated with this are deemed much to high compared to the benefit since it is anticipated that organisations will hardly ever need to move the backend.

**Inter-operable** Developers may want the system to inter-operate nicely with other software, e.g. importing from/exporting to spreadsheets or plain text todo lists, but this is outside the planned scope of the system. Consequently, it is considered irrelevant.

## 4.2 Technical platform

The technical equipment required by the system is a web server capable of Secure Socket Layer (SSL) transfer and executing Java Servlet 2.3 and Java Server Pages 1.2 (JSP), access to a MySQL database and a SMTP server. As a consequence, the system is designed for a Jakarta Tomcat 4.x web server which supports SSL and implements the JSP 1.2 and Java Servlet 2.3 standard.

The system is developed using the Java Development Kit (JDK 1.4.1) and certain components of Java 2 Enterprise Edition (J2EE). The clients of the system simply need a HTML 4.0 compliant browser capable of using SSL, receiving cookies and executing Java Script.

Unfortunately, not all browsers fully support Javascript and if they do, there are some differences between the implementations. This leads to a conflict between usability and portability in the frontend since we have to choose between supporting a large amount of browsers or be able to fully utilize Javascript. As a compromise,

we have choosen to support the most widely used browser, Internet Explorer 5.0 or above, and the very portable Mozilla.

# Chapter 5

# Architecture

The following chapter describes the structuring of the system into components, and their design.

## 5.1 Component architecture

All persistent data and functionality in our system is placed at one central place. Thus a client-server model is the most natural component architecture, see Figure 5.1.

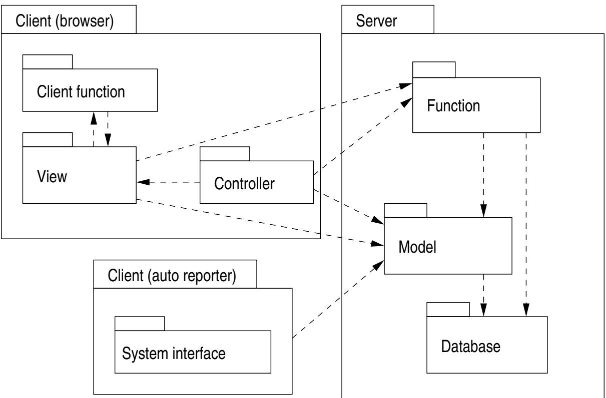



*Figure 5.1: The structure of the component architecture. The client-server pattern makes it possible to separate the code for handling the user interface from the code in the function, model and database components. Physically, most of the code of the interface components runs on the server machine, though (compare with Figure 5.2).*

The server part consists of a function, a model and a database component. The database component is used to encapsulate details about the database, i.e. how to

connect to it and some conversions of datatypes. The model component communicates with the database component for storage of data, and the function component communicates with both for efficiency reasons. By letting the function component communicate directly with the database component a theoretically unnecessary coupling is created. However, other approaches add severe performance loss, and since efficiency was not rated irrelevant, this is not acceptable. All server components are implemented as Java beans.

There are two types of clients: a user interface controlled by a browser and a system interface which can be used by an automatic defect reporting system to add reports directly to the system, through the model component to make sure the data is validated.

The browser client component contains all the functionality needed to provide a browser interface. This is split into three components: a view component that contains code for providing a user-visible view of the data, a controller component for manipulating the view and a client-side function component for catching mistakes early and handling client-side events. The client-side function component also checks that the user submits correctly formatted data. Having validation outside the server components breaks the encapsulation, but makes the system more pleasant to use, and usability is rated higher than maintainability.

In spite of the name, almost all of the code of the browser client is executed physically on the server, due to the architecture of the WWW, leaving only displaying of generated HTML pages and execution of Javascripts to the client machine, see Figure 5.2

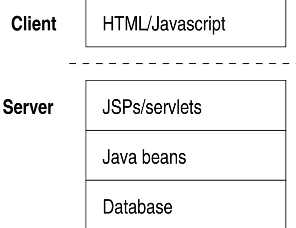



*Figure 5.2: A model of how the structure of the system is structured during runtime. On the server, the Java server pages and servlets run on top of the Java beans implementing the model and function components, which in turn run on top of the database. On a client machine, the generated HTML is displayed and Javascripts are executed.*

The system interface contains a single component providing a bridge between the software that needs to submit a report, and the model component. The component is implemented with a servlet so that it can be contacted easily with an HTTP request.

## 5.2 Database component

The database component is modeled as a single class, as illustrated in Figure 5.3. It contains a *get connection* method for establishing a connection to the database, and a *convert* method that converts Java built-in types to that of the database. Given this connection, the classes in the model component take care of formulating the SQL needed to retrieve and update information themselves.

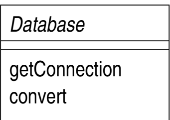



*Figure 5.3: The* DATABASE *class.*

The *get connection* method and the methods of the connection signals errors in the database requests by raising exceptions.

## 5.3 Model component

From the behavior state charts, the function list and the class diagram in the analysis document, the structure of the model component as shown in Figure 5.4 has been derived.

### 5.3.1 Common features of the classes

Some attributes and operations that are common to all of the classes are not displayed in the class diagram; instead they are discussed here. To realize persistence of the data in a consistent way a pattern is needed. We have designed a pattern to relate the Java objects to a relational database which all model classes except the permission hierarchy use, see Figure 5.5.

Each class has a unique integer database identifier as an attribute together with a constructor that takes an identifier and constructs an object with data from the database. Furthermore, a new object can be created using a constructor with no argument. This creates a new object without a unique identifier or data – when saved, the object gets a unique identifier from the database. These constructors are the create functions from the analysis.

The one-to-many aggregations and associations are formed by keeping the database identifier as an attribute in the many end (e.g. a report holds a system id).

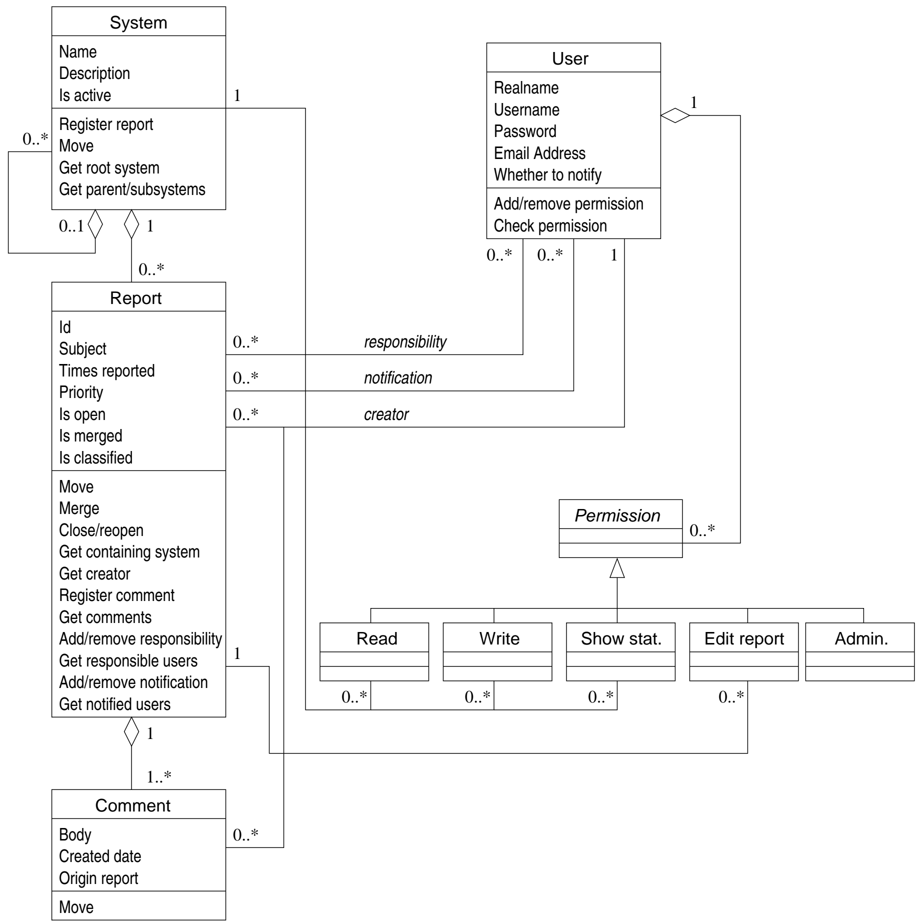



*Figure 5.4: Class structure in the model component. Some methods that are general for all classes are not shown (see Figure 5.5). Neither are the trivial get and set methods for all attributes.*

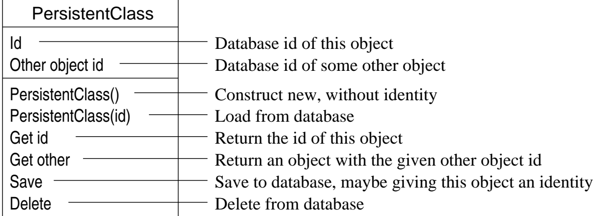



*Figure 5.5: The pattern for connecting the data in Java classes to a relational database.* PERSISTENTCLASS *is some class from the model component, and* OTHERCLASS *is some other model component class.*

Each class also has a *save* operation that saves the state of a given object to the database, and a *delete* operation for erasing the information from the database.

To ensure that the security is not compromised, permissions are checked in the model component by having all operations in REPORT that access the database take as an extra parameter the user that is performing the operation. The same is done for the operations in SYSTEM, COMMENT and USER that modifies the database; it is not necessary to check the read permissions since it is always possible to read a system, and comments are only made available through a report so that the read permission has already been checked. This way functions in the user interface cannot accidentally perform any unauthorized operations.

Any error that occur in the model component is handled by raising an exception that will continue up through the layers to the interface components.

### 5.3.2 System

The SYSTEM class represents a software system to which it is possible to add defect reports. A system may contain subsystems that are SYSTEMs themselves – each of these contains its own set of reports. By switching the *is active* attribute, it is possible to deactivate a system so that no further defects can be reported.

Supported extra operations are *register report* for registering reports in a particular system, *move* which moves the system to another system in the hierarchy, *get root system*, *get parent* and *get subsystems* for navigating the system hierarchy.

Moving the system requires write permission both to the system that is moved and to the destination system.

The read permission is required to read a system and the reports in it. To change the attributes of a system a write permission is required, either to the system itself or to one of its ancestors.

### 5.3.3 Report

A REPORT represents a single defect report with various information and a number of attached comments. The *id* attribute is a unique number that can be used to refer to the report when searching or when merging reports, this means that the id should be visible in the user interface. *Subject* is the header of the report. The number of times the defect has been reported is also stored – together with the *priority* which can be one of low, medium, high or critical.

Finally, a report keeps track of whether it is open or has been closed, using the *is open* attribute, whether it is merged with another report (in which case it is,

in effect, closed and its comments transfered to the other report) and whether it is classified.

The operations of REPORT are *register comment* for attaching comments to the report, *get comments* for retrieving all of the comments of the report, *get containing system* for retrieving the system that contains the report, *move* which moves the report to another system, *open* and *close* which opens or closes the report (a report will usually be closed when the reported defect is corrected).

Furthermore there are *add responsibility* and *remove responsibility* which both take a user as input for associating developers with reports, and *add notification* and *remove notification* for connecting users to the report so that they are notified when the report changes. The operations *get responsible users* and *get notified users* make it possible to retrieve the associations again. This way both of the many-to-many relations in the model component are encapsulated in REPORT.

REPORT also has the operation *merge* which merges a report with another master report. As input *merge* takes the master report to merge with. All comments from the report are moved to the master report. For each move, *notify* (from the function component, described later) is called. The number of times the master report has been reported is incremented by the number of times the duplicate has been reported, and the attribute *is merged* on the duplicate report is changed. Finally, a comment is added to the duplicate report with the id of the report that it has been merged with. This causes notify to be called on the duplicate report.

Reading report attributes and comments is always allowed if the user has read permission to the corresponding system. If the report is classified edit or write permission is required. It is possible to add reports to any SYSTEM that the user has read access to – the edit report permission to that report is then granted to the reporting user. Changing the attributes and comments requires either edit report permission on the report or write permission on the system corresponding to the report. Adding comments is always possible if the user has read permission to the report.

The permission needed to merge two reports is write (or edit report) permission on the duplicate report and read permission for the master report. The rights of the creators of the reports are not changed during the merge since both reports still exist.

### 5.3.4 Comment

A COMMENT is always attached to a single REPORT and acts as a description of the reported defect or just as a note, for instance if various solutions are discussed.

The *body* attribute contains the text of the comment and the creation date is stored in the *created date* attribute. The *origin report* attribute holds information of where the comment originated. This information becomes useful if the comment is moved to another report under a merge since the system shows where the comment came from.

The only special operation is *move* for moving a comment to another report. This is only used when merging two reports.

### 5.3.5 User

The USER class represents users. Each user has a unique *username*. The *password* is stored in an encrypted form and is used during the verification process. The *email address* is used for notification when reports are changed.

The DEVELOPER class from the analysis turns out to be empty so it is not necessary anymore. Instead the responsibility association is placed directly on the USER class.

Supported operations are *add permission* and *remove permission* for administrating the individual user's permissions and *check permission* for checking if the user has the rights to perform a given task. To be able to give other users the same permissions as oneself, the administrator permission is required.

### 5.3.6 Permissions

The security of the system is implemented through a number of permission classes each inheriting from the abstract PERMISSION class. Each user has a number of the concrete permissions READ, WRITE, SHOW STATISTICS, EDIT REPORT and ADMINISTRATION as described in section 2.1. The READ, WRITE and SHOW STATISTICS permissions are granted on the basis of a system and allows the user to use the permissions on that system and all of its subsystems.

### 5.3.7 Database design

Figure 5.6 shows how the data of the model component is modeled in the database. Some tables have been added without a corresponding class in the model component to implement the many-to-many relations. The permission hierarchy has been collapsed into a single table with a column to separate the different types (this is one of the three possibilities discussed in [10]).

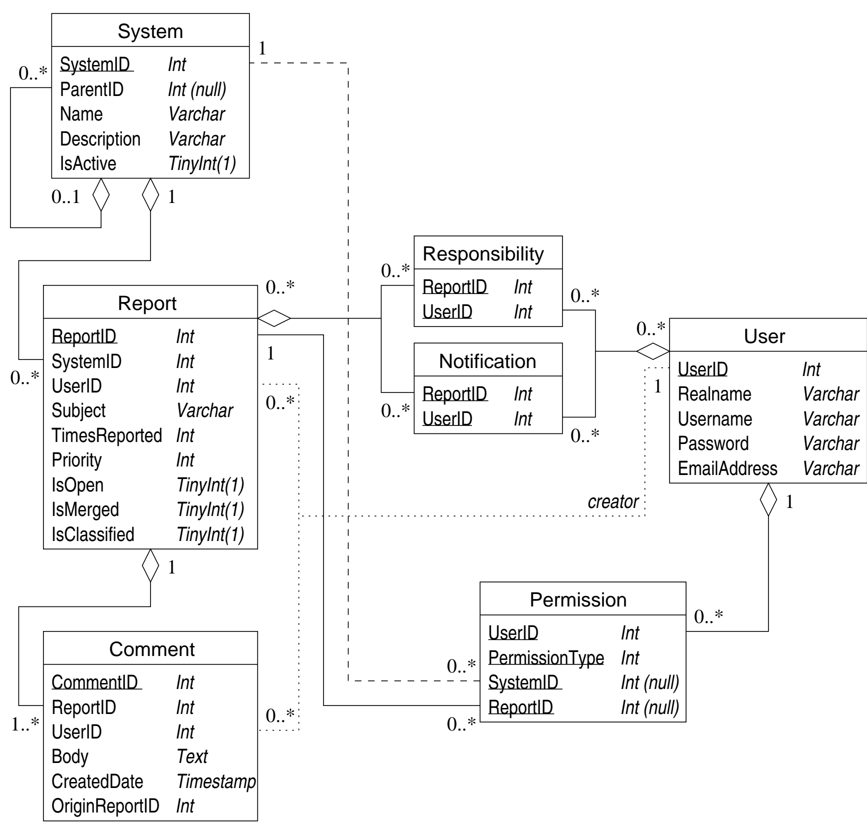



*Figure 5.6: The arrangement of the model classes into a set of tables for a relational database using UML. The association and aggregation lines connect the various IDs (some are dashed to provide a better overview). Underlined columns constitute the keys for the tables. All searchable columns are indexed.*

### 5.3.8 Concurrency

The client-server architecture creates a problem with concurrency since multiple users may be editing the same model object at the same time. The request-response architecture of the web is not well suited for updating changes instantaneously at all clients. Thus some clients may be seeing and committing old data.

There are two possible ways to solve these concurrency problems. Either the objects can be locked when it is possible to write to them, e.g. when the “manage report” page is entered. Then only one person may change the data at a time. Or the objects can be made to check upon saving themselves whether the data has been changed behind their back – using a timestamp – this way it is at least possible to reject the user’s changes with a warning that data may be lost.

The first possibility requires changing both the interface to the model component and the user interface. It also has problems of it own, in particular concerning the user interface semantics. Locking too much will make the system awkward to use when many users are accessing it.

Checking that the data has not changed in the database before saving an object is unfortunately somewhat awkward in the browser-based environment since the Java objects only live shortly during a request; so it requires transferring an extra set of attributes to the client to make it possible to compare with what the database contains.

However, the risk of two users colliding is small since modifying attributes is relatively uncommon and quickly finished compared to e.g. browsing the reports and adding new comments. Also, the potential troubles are limited; overwriting another user’s changes, and not an inconsistent state that could lead to a breakdown.

Hence, in compliance with the rating of the reliability criterion, which is less important, the time is invested in improving other aspects of the system and nothing is done to prevent concurrency problems.

## 5.4 Function component

The function component contains functions that are not associated with or specific to any interfaces to the system, and that do not rely on encapsulated parts of the model classes. Placing these outside the classes decreases the dependencies in the system since it reduces the amount of code that have access to the encapsulated parts. The classes are shown in Figure 5.7.

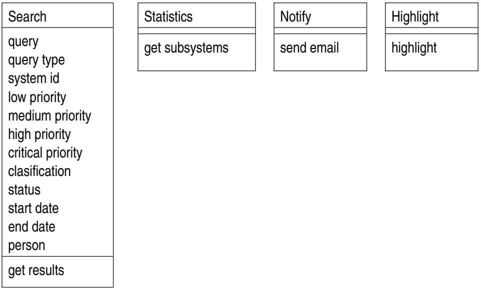



*Figure 5.7: Classes in the function component. Trivial set and get methods are not shown in the diagram.*

### 5.4.1 Notify

The notify function takes as input a report and either the new and previous values of an attribute or a comment. From these data, a message object is created and added to an internal notify queue. After a message has been in the queue for a specified amount of time, all messages concerning a specific report are collected and used to compose a unified email message to all users that have a notification association to the report.

The purpose of the queue is to reduce the amount of email that the system sends out. A lot of changes to a single report in a short period of time only generates one actual email message. It also reduces the complexity of the user interface components since they do not have to try to combine calls to notify, thus making them more maintainable in compliance with the criterion, which is important.

### 5.4.2 Search

To facilitate easy communication with the user interface, the search function is modeled as a class where the attributes are the search criteria. These attributes are combined using logical operations to restrict the search space. The search class then has the method *get result* for performing the query in the database with an SQL statement and returning a list of found reports. Reports that the user does not have permission to are excluded from the list.

### 5.4.3 Statistics

The statistics function has the method *get subsystems* for generating statistical of open and closed defects, with respect to their priority, for each subsystem. The method takes a SEARCH as parameter and the reports from the search result is used

for generating the statistical. Reports in a system that the user does not have show statistics permission to are not included in the statistical.

### 5.4.4 Highlight

To create a better user interface for the search page and the view report page, the words searched for are highlighted. To facilitate this, this function takes as parameter a text string which is the text to be highlighted and two text strings which defines the start and end tag for highlighting; these are placed before and after every found pattern.

## 5.5 Browser client component

The browser client is structured into three components: a view component which is implemented with Java server pages, a controller component which is implemented with servlets and a client-side validation component which is implemented with Javascript. The next sections discuss these components as a whole.

### 5.5.1 The commonly applied controller/view pattern

A Java server page (JSP) is a HTML page with embedded special tags. The web server translates the JSPs into special servlets; these are simply Java classes run by the web server. In [8], it is suggested that both are used in creating an interface. We have designed a particular pattern for the user interface as shown in Figure 5.8.

With the add report page as an example, our pattern works as follows. The user has a browser process running and when the “add defect” link is clicked, a request is sent to the system. In the general case, the request may be accepted by a servlet that does some preparations and then forwards the request to a JSP. If no preparations are necessary, the request is handled directly by the JSP – this is the case for the add report page. In any case, the JSP constructs an ordinary HTML page which is sent back to the browser as the response. So the browser renders the add report page.

Afterwards, when the user clicks a link or a button on the HTML page, a new request is sent. A servlet then processes the result (this may involve updating the model component) and decides what URL to forward to next – for the add report page, this is either back to the start page or to a page saying thank you. After this, the process repeats. In case no processing is needed and the decision of where to forward to is fixed, the latter servlet may be omitted too.

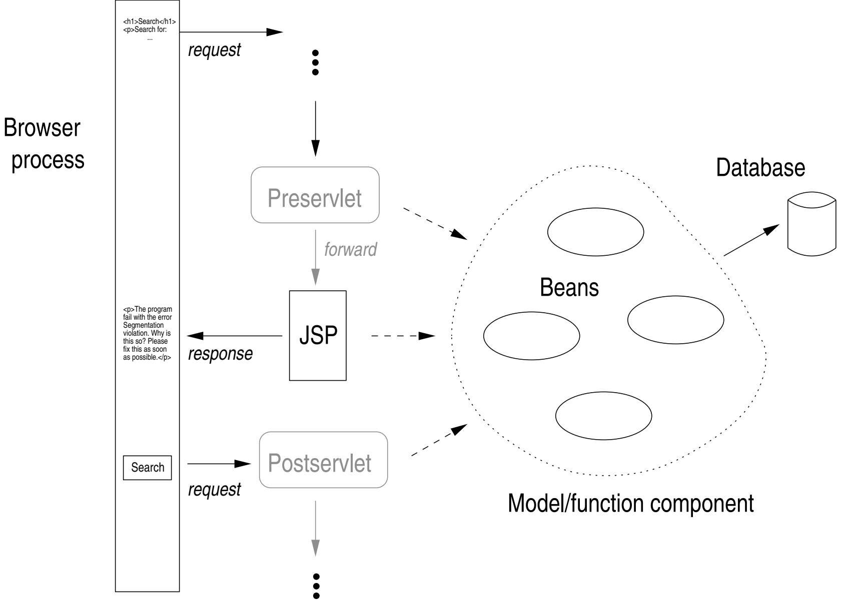



*Figure 5.8: The interactions between the user's browser and the server pages and servlets. The server page and the servlets can call the model and function component beans directly. As hinted, the servlets are unnecessary in some cases.*

This division of labour is an application of the generic model-view-controller pattern. The JSPs provide the views and the servlets serve as controllers. Since the servlets are just Java classes they are well suited for being controllers. Likewise, the JSP architecture makes it easy to construct the views since the pages look like ordinary HTML pages except for the dynamic contents. Furthermore, the JSP architecture allows easy transfer of HTML form fields to bean objects. This of course requires the model component classes to be written as beans. A bean is basically a Java class following certain naming conventions to facilitate ease-of-use and common features e.g. introspection, setting and getting fields.

Our split between controller and view is not a completely clear cut since, in the simple case, the JSPs decide where to forward to. However, the benefit of having completely independent JSPs does not seem to be worth the added complications of the source code.

### 5.5.2 Overview of view and controller components

The Figures 5.9 – 5.14 show the relationship between the individual parts of the view and the controller components. These figures can assist in maintaining the system, as the dependencies between views and controllers provide information

about which parts can be affected by a given change in another part. For an overview, refer to Figure 3.9 on page 34. The start page is marked with a bold box. Servlets are marked with "(servlet)" after the name, the other pages are server pages.

A description of each element is given in Table 5.1 below.

| Element | Functionality | Figure |
|---|---|---|
| default | Welcomes the actor to the system and provides an option for quickly adding a report. | 5.9 |
| login | Prompts the actor for a username and password and check these against the current users in the database. If valid information has been entered the actor will be granted access to the system, otherwise the actor will be asked to try again. | 5.9 |
| create user | Lets the actor create a new user by filling out the following information: name, email address, username and password. | 5.9, 5.14 |
| create user (servlet) | This servlet has two uses, the first is saving submitted information on a user in the database. And the other is updating the user information. If an administrator has requisted the password to be changed, a new password is generated and sent to the user. The password in then stored in a secure manner in the database. | 5.9, 5.14 |
| send mail (servlet) | This servlet is used in two different places. One is sending an activation message containing an activation URL to the email address of a newly created user. The actor must click this to activate the user. The other one is assigning a new random generated password to a user who has forgotten his username or password. A new password together with his username is then sent to the user. | 5.9 |
| wait for mail | A confirmation screen informing that the message has been sent. | 5.9 |
| activate user (servlet) | Activates a newly created user from URL-passed id. | 5.9 |
| send password | The actor is prompted for his email address. | 5.9 |

| | | |
|---|---|---|
| send password ok | Confirmation screen informing that the password and username has been sent successfully. | 5.9 |
| choose system step | A system tree structure enabling the actor to choose an appropriate system. | 5.9, 5.10 |
| add report | Collecting the subject and nature of the defect. | 5.10, 5.11 |
| add report (servlet) | Takes the user submitted information and adds it to the database together with the appropriate relations. | 5.10, 5.11 |
| thank you | Confirmation screen thanking the actor for his participation in adding a defect. | 5.10 |
| log off | Terminates the user session. | 5.10 |
| start page | The main page which gives the actor an overview of the functionality he has access to. | 5.9, 5.11 |
| my involvements | A summary of a user's different involvements in the defect tracking system. | 5.11 |
| search | Search is equipped with an input line which lets the actor narrow down the results by adding extra criterias to the request. Search is also constructed to show search results as links to the reports when available. | 5.11, 5.12 |
| search result | Replaces any previous search results in scope and redirects the search request to the search servlet. | 5.12 |
| search (servlet) | Prepares result objects and executes a search with the submitted criteria and query string. The results are loaded into the session scope for display. | 5.12 |
| search refresh | Sends the previous search request again. | 5.12 |
| prepare report (servlet) | When a report is chosen from the search result, the prepare report servlet accesses the database and prepares the report to be displayed. | 5.12 |
| view report | Shows the attributes and comments of a report. The actor can submit additional comments from this page. | 5.12 |
| manage report | Provides the possibility for a developer to edit a chosen report. This includes deleting comments. | 5.12 |

| | | |
|---|---|---|
| add comment (servlet) | This servlet is responsible for adding a submitted comment to the database. | 5.12 |
| manage systems | An overview of the system structure is presented in a tree form making it easy to locate the desired system. When one is chosen, there are multiple possibilities; move, edit, delete and add subsystem. | 5.11, 5.13 |
| manage system (servlet) | This takes care of sending the submitted data to the right place and making the necessary preparations. | 5.13 |
| delete system (servlet) | Deletes a system and its subsystems. This includes all reports and comments. | 5.13 |
| edit system | Displays the information on the desired system to edit and lets the actor to modify this information. This page is also used to add a subsystem. | 5.13 |
| edit system (servlet) | Saves the changes or creates a system in the database. | 5.13 |
| move system | Prompts the actor with a system tree enabling him to select a destination for the system that is being moved. | 5.13 |
| move system (servlet) | Moves a system to a new parent. | 5.13 |
| prepare users (servlet) | This servlet retrieves all users of the defect tracking system from the database. | 5.11, 5.14 |
| manage users | Displays all users and enables an administrator to edit, delete or change permissions of a particular one. The page also includes an option to create a new user. | 5.14 |
| postmanage users (servlet) | Redirects the actor to the right destination after preparing the chosen user. | 5.14 |
| edit user | Allows an administrator to change information on a user, this includes changing the name, email or generating a new password for the user. | 5.14 |
| edit permission | Shows a users permissions together with options for changing or deleting these. | 5.14 |
| postpermission (servlet) | Saves changed permissions to the database, preserving the user relations. | 5.14 |

| | | |
|---|---|---|
| statistics | Information on which system and between which dates the statistics should be generated is provided here. | 5.11, 5.15 |
| statistics result | Redirects the data in the request scope using introspection. | 5.15 |
| statistics (servlet) | Generates data for a statistical overview of a selected system. These data are generated from the submitted criteria. | 5.15 |
| statistics subsystems | Displays statistical data using bar diagrams. | 5.15 |

*Table 5.1: Descriptions of all the user interface elements.*

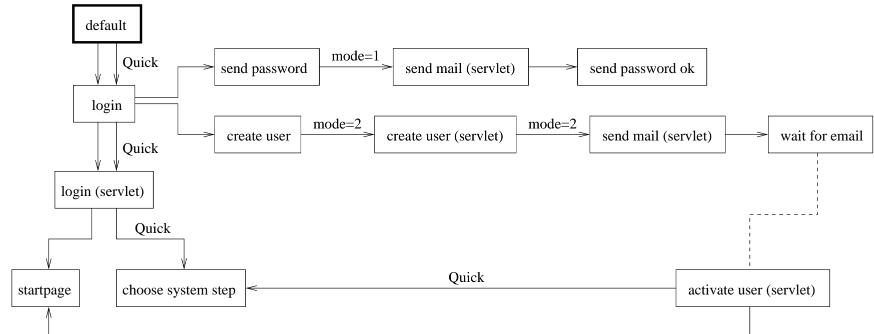



*Figure 5.9: The startpage. From the default page a user can choose between a quick mode and a non-quick mode. The choice dictates which pages he will be redirected to and which pages he will be able to choose from (the arrows marked with "quick"). Choose system step is continued in Figure 5.10 and startpage is continued in Figure 5.11*

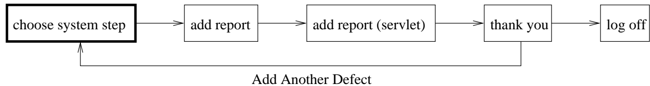



*Figure 5.10: The quick guide.*

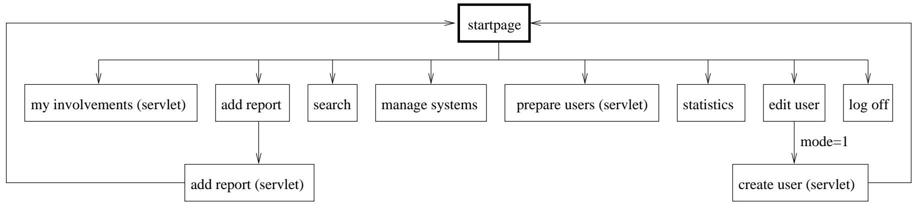



*Figure 5.11: The search and manage defects page. The pages my involvements and search are continued in Figure 5.12, manage systems is continued in Figure 5.13, prepare users is continued in Figure 5.14 and statistics is continued in Figure 5.15*

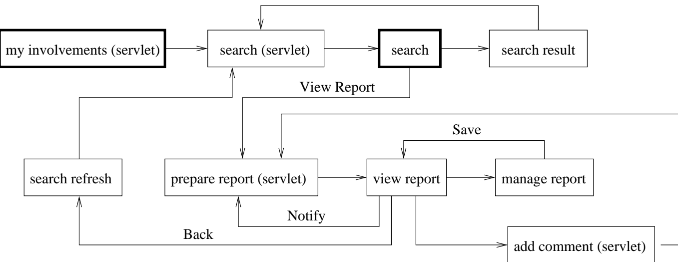



*Figure 5.12: There are two possible entrances to searching, one is my involvements, which automatically searches for the user's defects, and the other one is search, which is the normal search page. Note that searching is the only way to find a specific report.*

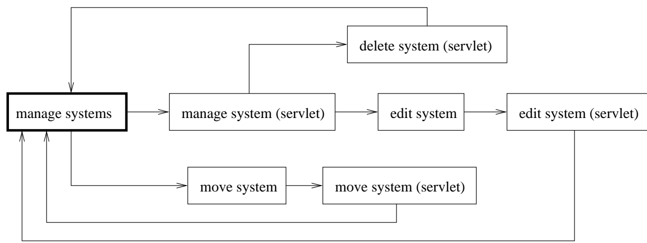



*Figure 5.13: Manage Systems.*

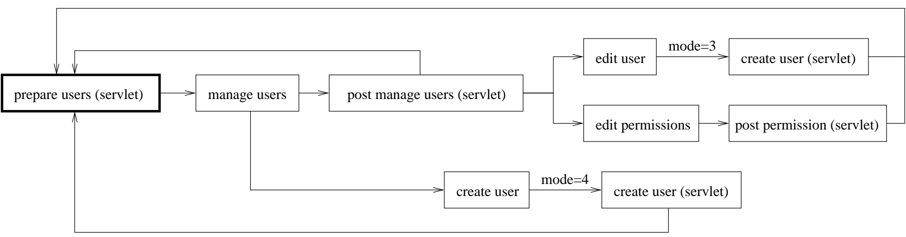



*Figure 5.14: Manage users.*

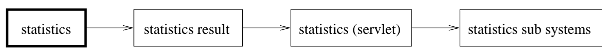



*Figure 5.15: The statistical overview.*

### 5.5.3 Overview of client functions

Table 5.2 gives an overview of the functions in the client function component.

| Function | Functionality |
|---|---|
| is date valid | Returns true if the parameter is a valid date on the form yyyy-mm-dd. |
| is empty error check | Returns true if the field is empty and prints an error message to the user. |
| is valid email | Returns true if the parameter is a sensible email address. |
| insert system | Prints the HTML code for showing a specific system in a system tree. |
| browse systems | Opens a pop-up window with a system tree. |

*Table 5.2: Functionality of the client-side Javascript.*

### 5.5.4 Error handling

There are two possible causes of errors: either something in the system went wrong because of a defect somewhere, or the user tried to perform an action that would put the system in an invalid state.

System errors detected in the model and function component are handled by exceptions. Since these exceptions represent errors in the system, it is not possible easily to work around them. So instead they are catched by a JSP handler that outputs an apologising HTML error message.

Some of the user errors can be catched before an actual request is submitted to the server. To facilitate this, all relevant pages include the above Javascript library. The library will print an error message under the title and color relevant fields red in case an invalid action is performed.

Other user errors can only be detected on the server. The general pattern is then to send the user back to the original page with an error message.

### 5.5.5 Security

All server pages and servlets check the user's session for a valid user object. This user object is created upon login where the person trying to login goes through an authentication process. Thus it should not be possible to access the system without being authorized.

# Part III

# Implementation

# Chapter 6

# Implementation

This chapter documents how certain parts of each component from the design has been implemented, using the following technologies: Java beans [12], servlets [15], Java server pages [14], Javascript [7], DHTML [18] and ordinary HTML [19].

## 6.1 Documentation

Documentation of the implemented beans is provided with cross-references and overviews generated automatically by Javadoc [13]; refer to Appendix A.4.

The code for the individual servlets and Java server pages are very specific for their tasks, so only overviews are provided in Section 5.5.2 of the design document. Further information can be retrieved directly from the commented code. Likewise, the Javascript code is documented by the overview in 5.5.3; the code itself provides further information.

The code can be found by following the reference in Appendix A.3.

## 6.2 Unimplemented features

Although the defect tracker is working, there are still some missing features and reported defects left in the system. These can be found using the guide shown in Appendix A.1.1.

The most fundamental ones are:

- Notification of changes to reports is not implemented.
- Responsibilities are shown, but there is no interface for adding or removing them.

- The automatic reporting interface is not implemented.

Smaller problems include:

- The description field of systems is not utilized.
- The times reported field of report is not utilized.
- There is no browse button for merging which makes the interface awkward.
- Submission on enter key-press is not implemented. This makes usage of the interface less efficient.
- Configuration of some features, e.g. where to locate the database, requires editing the source code and recompilation of the beans.

## 6.3 Database component

The database component is implemented as an abstract Java class, containing a method for retrieving a CONNECTION and methods for converting data to MySQL syntax (the latter are not shown):

```
1  abstract public class Database
2  {
3    public static Connection getConnection()
4      throws ...
5    {
6      String dsn = "jdbc:mysql://localhost/defecttracker?
7                                user=aaa&password=bbb";
8      try{
9        Class.forName("com.mysql.jdbc.Driver");
10     }
11     catch(ClassNotFoundException e) {
12     }
13
14     return DriverManager.getConnection(dsn);
15   }
16   ...
17 }
```

Line 6–7 creates a string representing the location of a MySQL database (*aaa* and *bbb* should be replaced by the appropriate user and password). Line 9 loads the MySQL JDBC driver and line 14 returns the connection to the database.

## 6.4 Model component

The model component classes is implemented using Java beans as prescribed in section 5.1 of the design document. The classes in the model component communicates directly with the database class and are aware of persistence and security issues themselves, which is the topic of this section. In the following, code for the beans regarding the above issues is presented with the report bean as an example.

### 6.4.1 Obtaining database connectivity

Database connectivity is obtained by accessing the *getConnection()* method of the abstract DATABASE class. Once a database connection is obtained, a STATEMENT must be created before SQL statements can be executed. Executing a SQL query returns a RESULTSET containing zero or more records that match the query. If the purpose of the query was not to retrieve data but to delete, insert or update, an empty result set is returned. After the given task is performed the database connection is closed in a exception-safe manner.

```
1   Connection con = Database.getConnection();
2   ResultSet rs = null;
3
4   try {
5     Statement stm = con.createStatement();
6     rs = stm.executeQuery('SELECT * FROM report');
7   }
8   finally {
9     con.close();
10  }
```

### 6.4.2 Ensuring persistence

When a new report is created, it does not have persistent identify before it has been be saved to the database. If a report with a persistent identity is modified, the changes must be updated in the database using this identifier. The *save* method must distinct between these two cases; this is facilitated by having two constructors in REPORT.

The *Report(int id, User u)* constructor sets the *reportID* and then calls *load(u)*, which checks that the user has the appropriate permissions to view the report and if so fills in the attributes with the values from the database. The *Report()* constructor for new reports sets the *reportID* to -1, indicating that the report is not persistent yet.

```
1  class Report {
2    ...
3    public Report(int id, User u)
4    throws ...
5    {
6      reportID = id;
7      load(u);
8    }
9
10   public Report()
11   {
12     reportID = -1;
13   }
14   ...
15 }
```

If the report is not persistent when it is saved, it is assigned an identifier – a reportID – generated by the reportID auto-increment field in the database. After this, comments can be added to the report since it then has an identity.

### 6.4.3 Removing persistent objects

A persistent object is removed from the database by instantiating it and then calling its *delete* method as listed below. First the user's permissions are checked in line 5 to verify that he has write permission to the containing system. If this is not the case a permission denied exception is thrown in line 6.

Due to the aggregation relation between a report and its comments, all the comments must also be deleted when a report is deleted. This is implemented by instantiating the comments and calling their *delete* method. It would be possible to delete all of them with a single, and faster, SQL query, but this conflicts with the object-oriented design, reducing the flexibility.

Line 11 fetches a LINKEDLIST containing the comments. Line 15-16 retrieves the next comment in the list and deletes it, by delegating the deleting responsibility to the COMMENT class where the user's permissions are also checked. Once all the comments have been deleted, the report itself is deleted in line 20-22.

```
1  class Report {
2    public void delete(User u)
3    throws ...
4    {
5      if (!PermissionChecking.checkWrite(u, this))
6        throw new PermissionDeniedException();
```

```
7
8       Connection con = Database.getConnection();
9
10      try {
11        LinkedList comments = getComments();
12        ListIterator commentsIterator = comments.listIterator();
13
14        while (commentsIterator.hasNext()) {
15          Comment c = (Comment) commentsIterator.next();
16          c.delete(u);
17        }
18
19        Statement stm = con.createStatement();
20        String SQLStatement = 'DELETE FROM report WHERE reportID='
21                              + reportID;
22        stm.execute(SQLStatement);
23      }
24      finally {
25        con.close();
26      }
27    }
28  }
```

### 6.4.4 Verifying permissions

A user's permissions are checked by accessing appropriate methods in the abstract PERMISSIONCHECKING helper class. The *checkWrite(User u, Report r)* method checks whether the user has write permission to a report; this includes checking the system containing the report and all of its parents for write permission. The check permission request is propagated up through the system tree until the requested permission is found or the root system is reached without finding the permission. In the latter case false is returned, else true is returned.

In line 5 the user is checked for an edit report permission to the report. If that permission is found in the user's list of permissions he is granted write access to the report. If not, the *checkWrite(User u, TrackedSystem s)* is called in line 8, which begins the propagation.

```
1 public abstract class PermissionChecking {
2   public static boolean checkWrite(User u, Report r)
3     throws ...
4   {
5     if (u.checkPermission(new EditReportPermission(r.getReportID())))
```

```
6         return true;
7
8       return checkWrite(u, r.getContainingSystem());
9     }
10
11    public static boolean checkWrite(User u, TrackedSystem s)
12      throws ...
13    {
14      if (u.checkPermission(new WritePermission(s.getSystemID())))
15        return true;
16
17      while (!s.isRoot()) {
18        s = s.getParent();
19
20        if (u.checkPermission(new WritePermission(s.getSystemID())))
21          return true;
22      }
23
24      return false;
25    }
26    ...
27  }
```

When a user's *load* method is called, all attributes are filled in, including the list of permissions. The *checkPermission* method in the USER class looks up the requested permission in the list of permissions and checks if an equal permission exists, returning the appropriate boolean value.

```
1  public class User {
2    ...
3    private LinkedList permissions = new LinkedList();
4    public boolean checkPermission(Permission p)
5    {
6      return permissions.contains(p);
7    }
8    ...
9  }
```

## 6.5 Function component

The function component is implemented using the same technology as the model component. However, there is one important difference between the classes in the

two components; the classes in the function component are never required to be persistent. This simplifies the design. Also, the code of the function component is very similar to that of the model component, so no further examples are given.

## 6.6 Browser client component

The graphical interface is implemented using JSP and servlets on the server side and HTML, Javascript and DHTML on the client-side. Javascript and dynamic HTML is needed for error handling and for drawing and expanding the system tree. A stylesheet is used to facilitate a central control of the layout.

### 6.6.1 Controllers

The controllers in the browser client component are implemented as servlets running on the server as prescribed in Section 5.5 of the design document. The servlets have different areas of responsibility: all those accessing the model component are responsible for verifying the current session, and many of the servlets are also responsible for preparing objects for the views. Common for all servlets is that they must catch exceptions caused by invoking methods in the beans.

Hence, logging in, verifying the session, handling exceptions and preparing objects are the topics of this section, illustrated with example servlet code.

#### Logging in

When a user logs into the system by entering his username and password and clicks the appropriate button, control is transfered to the login servlet. The login servlets receives a USER object with the submitted username and password and must verify that this user actually exists in the database.

```
1  public class LoginServlet extends HttpServlet
2  {
3    public void doPost(HttpServletRequest request,
4                       HttpServletResponse response) {
5      ...
6
7      try {
8        User u = (User) request.getAttribute("loginRequest");
9
10       HttpSession session = request.getSession(true);
11
12       User u2 = new User(u.getUsername());
```

```
13
14          if (ServletHelpFunc.generateHash(u.getPassword()).
15                                        equals(u2.getPassword())){
16            if (u2.getActive()){
17
18              session.setAttribute("login", u2);
19              session.setMaxInactiveInterval(3600); // 30 minutes
20            }
21            else{
22              throw new IsNotActiveException(u.getUsername());
23            }
24          }
25          else{
26            throw new LoginNotFoundException(u.getUsername());
27          }
28          ...
29        }
30        catch (Exception e) {
31          ...
32        }
33      }
34    }
```

Line 8 retrieves the submitted user object from the request scope and in line 10 the current session is fetched; if the session does not exist, it is created. Line 12 tries to retrieve the user with the submitted username from the database. If the user does not exist, an exception is thrown. Line 14-15 verifies that the two user objects have identical passwords and that the user is activated. If these checks evaluates to true the user bean is stored in session scope and is thereby made available to the entire system.

### Verifying the session

Once the session is established, it must be verified for existence of a login entry everytime a new task is performed. The servlets handles this by inheriting a *check login session* method from the MASTERSERVLET, which provides shared functionality for all servlets.

```
1  public class MasterServlet extends HttpServlet {
2    protected User currentUser;
3    protected HttpSession currentSession;
4
5    protected void checkLoginSession(HttpSession session)
```

```
 6        throws LoginSessionNotFoundException
 7    {
 8        this.currentSession = session;
 9
10        currentUser = (User) session.getAttribute("login");
11
12        if (currentUser == null)
13          throw new LoginSessionNotFoundException();
14    }
15    ...
16  }
```

Line 9 tries to fetch the login entry in the session and line 11 checks whether a user object was fetched or not. If a user object did not exist an exception is thrown, else the session is valid.

The *check login session* must be embedded manually in the servlets that needs to check the session. This is done by adding the following lines at the top:

```
try {
  checkLoginSession(request.getSession());
}
catch(LoginSessionNotFoundException e) {
  handleException(request, response, e);
}
```

#### Handling exceptions

Servlets handle exceptions the same way they check for a valid login session, by using inherited methods from the MASTERSERVLET, in this case the *handle exception* method.

```
 1  public class MasterServlet extends HttpServlet {
 2    protected RequestDispatcher exceptionRequest;
 3
 4    public void init()
 5    {
 6      exceptionRequest = getServletConfig().getServletContext().
 7                         getRequestDispatcher("/exception.jsp");
 8    }
 9
10    protected void handleException(HttpServletRequest request,
11                                   HttpServletResponse response,
12                                   Exception e)
13    {
```

```
14        request.setAttribute("exceptionCaught", e);
15
16        try{
17          exceptionRequest.forward(request, response);
18        }
19        catch(Exception eForward){
20          System.err.println("exception in forward");
21        }
22      }
23      ...
24  }
```

Line 6-7 initializes *exception request* which represents a JSP page where the exception is forwarded to and can be handled. Line 15 sets the received exception as an attribute in request scope and forwards to the JSP page in line 17.

### Preparing objects

One of the advantages of using controllers – the servlets – is that these can concentrate on preparing objects for the Java server pages, and other tasks that do not require any presentation. This approach makes the presentation easier because the Java server pages simply receives a list of objects and can focus on presenting them. This was chosen whenever it was possible and convenient, if this was not the case a mixture was implemented instead.

```
1   public class PrepareReportServlet extends MasterServlet
2   {
3     public void doGet(HttpServletRequest request,
4                       HttpServletResponse response) throws ...
5     {
6       ...
7
8       String reportID = request.getParameter("reportID");
9       String systemPath = "";
10      Report report = null;
11      TrackedSystem system = null;
12      User reporter = null;
13      LinkedList comments = null;
14      LinkedList commentsUsers = null;
15      LinkedList responsibilities = null;
16      Boolean notify = null;
17
18      try {
```

```
19         report = new Report(Integer.parseInt(reportID), currentUser);
20
21         ...
22
23         system = report.getContainingSystem();
24         systemPath = system.getPath();
25         reporter = report.getCreator();
26
27         notify = new Boolean(report.getNotification(currentUser));
28
29         comments = report.getComments();
30         responsibilities = report.getResponsibleUsers();
31
32         ListIterator commentIterator = comments.listIterator();
33
34         commentsUsers = new LinkedList();
35
36         while (commentIterator.hasNext()) {
37           Comment c = (Comment) commentIterator.next();
38           commentsUsers.add(c.getCreator());
39         }
40
41       }
42       catch(Exception e) {
43         handleException(request, response, e);
44       }
45
46       request.setAttribute("commentsUsers", commentsUsers);
47
48       ...
49
50     }
51   }
```

Line 8-16 declares the objects required by the Java server page and line 19 and 23-39 prepares them. Line 46 stores a prepared object in request scope, this is done with all the objects required by the view but only one example in given to avoid cluttering the code example. Objects in request scope are available to the Java server page once the request is forwarded (in the not included part at the bottom).

### 6.6.2 Views

The views in the browser client component are implemented as Java server pages running on the server as prescribed in Section 5.5 of the design document. The views are responsible for presentation and providing interaction possibilities for the user, both are fulfilled by generating HTML.

The views interact with objects, either created by themselves or a controller, in both the model and function component. This section illustrates how the views interact with the objects, through examples from the search bean. Furthermore, as each view is required to verify the session, an example of using page includes is given.

#### Object introspection

For handling multiple input fields in a HTML form, JSP provides a feature called introspection. This feature was used throughout the implementation phase to match the input fields with the appropriate set methods in a bean. Introspection would for an input field named “person” call the setPerson() method with the new value as a parameter. This cuts down on the amount of necessary code, reducing the potential for defects.

The SEARCH RESULT JSP receives search criterias from the SEARCH JSP and creates a SEARCH object in the session scope. Introspection is performed with the received criterias and then control is forwarded to the SEARCHSERVLET where the newly created object is available with the criterias as attributes.

```
<jsp:useBean id="searchRequest" class=
  "defecttracker.beans.Search" scope="session"/>
<jsp:setProperty name="searchRequest" property="*"/>
<jsp:forward page="/Servlet/SearchServlet"/>
```

#### Page includes and using beans

Code reuse is achieved by using page includes. The include pages can be anything from other Java server pages to HTML pages. The below code, taken from the VIEW REPORT JSP, line 1 includes “loginChecker.jsp” which checks the current session for a login entry.

Line 4-5 assigns an identifier to a recieved object in request scope. Once a object is assigned an identifier – with the JSP useBean tag – it is accessible throughout the Java server page as an Java object.

Line 11-25 outputs the necessary HTML and on line 22 a JSP tag is inlined to dynamical generate the HTML page.

```
1  <%@ include file="inc/loginChecker.jsp" %>
2  <%@ page import="defecttracker.beans.*, java.util.*" %>
3
4  <jsp:useBean id="commentsUsers" class="java.util.LinkedList"
5   scope="request"/>
6
7   ...
8
9  <%@ include file="inc/topInclude.jsp" %>
10
11 <h1>View report</h1>
12 <small>You are here: <a href="startPage.jsp">Start page</a>
13                        - <a href="/defecttracker/search.jsp">Search</a>
14                        - View report</small><br><br>
15
16 <table width="500" align="left">
17   <tr>
18     <td>
19       <table width="500">
20         <tr>
21         <td width="100"><strong>System:</strong></td>
22         <td width="400" align="left"><%= systemPath %></td>
23          </tr>
24
25          ...
```

### 6.6.3 Client-side

The client side is implemented using Javascript and HTML/DHTML.

One of the most commonly used features in the browser client component is the component responsible for drawing the system structure as a tree and making it possible for the user to select a system. The graphical layout is shown in Figure 6.1 on the next page.

Different kinds of trees is used in the implementation, each with special functionality. With the permission tree, for instance, more than one system can be selected and selections are inherited from the ancestors. These trees are specialisations of the simple tree type which is described here. Only one system can be selected so a new selection deselects the previous selection.

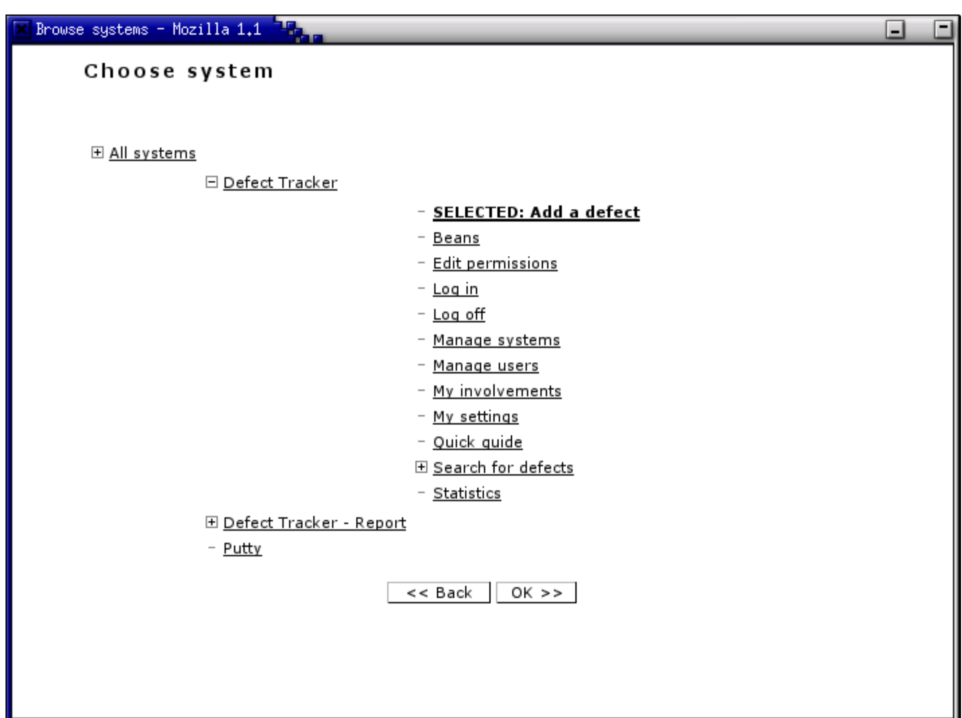



*Figure 6.1: The system tree. In this case the user is adding a defect and must choose a system.*

### The tree in HTML

There are several requirements the tree must obey; it must draw the entire tree and be able to change the visibility of each leaf. Also it must provide functionality to select a single system and to expand or collapse parts of the tree.

These requirements can be satisfied by changing the properties of HTML tags during runtime. First an example of a simple static tree in HTML with one parent and two children:

```
<table>
 <tr>
  <td>This is a parent (id=1)</td>
  <td><table id="systemArea1">
        <tr><td> </td></tr> <!-- Empty: children below parent -->
        <tr>
          <td>This is child1 (id=2)</td>
        </tr>
        <tr>
          <td>This is child2 (id=4)</td>
        </tr>
      </table>
  </td>
 </tr>
</table>
```

By using the <**table**> tag, the browser itself takes care of hierarchical placement of parents and children. Also, placing every subsystem of a system in the same table makes it easy to hide or show them all simple by changing the visibility of the table corresponding to the system. By nesting the tables, the children can have children themselves.

### Augmenting the HTML tree with Javascript

The dynamic functionality needed by the tree uses DHTML which provides functionality to change the visibility of elements, and Javascript which controls the DHTML, handling events such as clicks.

The HTML itself is built by the Javascript. When a system is added, the script only needs to know whether the system contains subsystems and the trivial attributes name and unique id of the system. Also, in order to close a table of a given system, the Javascript must know when all children of a system has been inserted.

So the interface consists of two functions: *insertSystem(systemName, systemID, hasSubSystem)* and *insertEndOfSystem()*. Only if a system contains one or more subsystems does it make sense to close the system, otherwise the system is closed automaticly. A tree showing a system structure is drawn when calling these functions as shown below:

```
insertSystem('This is a parent (id=1)', 1, true);
 insertSystem('This is child1 (id=2)', 2, true);
  insertSystem('This is a child to child1 (id=3)', 3, false);
 insertEndOfSystem(); // child1 ends here...
 insertSystem('This is child2 (id=4)', 4, false);
insertEndOfSystem(); //The parent ends here...
```

A simplified version of the Javascript is listed below:

```
1  function insertSystem(systemName, systemID, hasSubSystem)
2  {
3    document.write('<tr><td>');
4
5    if (hasSubSystem) {
6      document.write('<a href="javascript:reSpan('+systemID+')">');
7      document.write('<img src="opened.gif"
8                          id="spanImg'+systemID+'">');
9      document.write('</a>');
10   }
11   else
12     document.write('<img src="noExpanding.gif">');
13
14   document.write('</td><td>');
15   document.write('<a href="javascript:reSelect('+systemID+')">');
16   document.write(systemName);
17   document.write('</a></td>');
18
19   if (hasSubSystem) {
20     document.write('<td>');
21     document.write('<table id="systemArea'+systemID+'">');
22     document.write('<tr><td> </td></tr>');
23   }
24   else {
25     document.write('</tr>');
26   }
27 }
```

Line 4 in *insertSystem()* is only executed if the system contains subsystems and provides a link to expand the tree. This is done by *reSpan()* by switching the current visibility of the subsystems and changing the the image from “opened.gif” to “closed.gif”. This is illustrated with this simplified version:

```
function reSpan(systemID)
{
  var currentTable = 'systemArea'+systemID;
  var currentImage = 'spanImg'+systemID;
```

```
  if (currentTable.visibility == 'hidden') {
    currentTable.visibility = 'visible';
    currentImage.src = 'opened.gif';
  }
  else {
    currentTable.visibility = 'hidden';
    currentImage.src = 'closed.gif';
  }
}
```

Line 11 in *insertSystem()* provides the link to select a system by calling *reSelect(systemID)*. In this implementation *reSelect(systemID)* changes the system name to bold and holds the current selected systemID in a global Javascript variable. A simplified version of this function is:

```
function reSelect(systemID)
{
  // let currentSelection be a global variable
  if (systemID != currentSelection) {
    // switch bold from old to new
    systemID.bold();
    currentSelection.unBold();

    currentSelection = systemID;
  }
}
```

So a JSP using a tree can call the two functions *insertSystem(systemName, systemID, hasSubSystem)* and *insertEndOfSystem()* in order to build the tree. When the page is loaded the user can use the tree until some button is clicked; this button must copy the currentSelection variable to a HTML form and submit it.

# Part IV

# Test

# Chapter 7

# Test

This chapter describes the two means we have used to test the system: unit tests and system tests. Unit testing has been used as a tool to test part of the system while it was being written so that errors were detected early and effectively when the programming issues were still fresh in memory, whereas the goal of the system testing was to ensure that the completed system fulfills the requirements from the analysis.

## 7.1 Unit tests

Unit tests can provide a rigorous means for testing the system since the tests are programmed instead of just performed, and consequently can be repeated easily to catch regressions (errors that occur as a consequence of an error correction). We have thus created unit tests for all public methods of the classes in the database, model and function components. The JUnit framework [6] has been utilized for this since it provides a convenient method for writing and running the tests.

In total, there are 63 test so we will not list all of them here. Instead, the JUnit framework supports generating an HTML-formatted report of the test results. A reference to these results is available in Appendix A.5. Furthermore Appendix A.3.5 provides a reference to the source code of the test implementation.

To illustrate how the unit tests have been designed, the following section explains the test of the *delete* operation in SYSTEM.

### 7.1.1 Testing *delete*

The *delete* operation should delete the system from the database. However, to maintain a valid state of all objects, it must also remove objects that references

it, such as subsystems and reports. The subsystems in turn need to delete their subsystems and reports, and the reports need to delete their comments.

Since the purpose of the test is to test the delete operation of the system, the delete operations of the reports are however assumed to be correct; they are tested separately. Also, the delete operations of the subsystems are assumed to function correctly. This is derived from the fact that delete works in a recursive manner. So from a theoretical point of view, if delete correctly deletes the subsystems of a system and also deletes the system itself, it follows by induction that it will correctly delete any system tree.

Hence, a test plan for the unit test looks like this:

1. Set up phase:
   (a) add a test system to the root system
   (b) add two subsystems to this system
   (c) add two reports to this system
2. Action phase:
   (a) call delete on the test system
3. Test phase:
   (a) check that the test system does not exist
   (b) check that the two subsystems do not exist
   (c) check that the two reports do not exist

### 7.1.2 Test results

The unit test results consist of a success or a failure for each test. These results were used during implementation to quickly locate errors once a test did not succeed. Most of the tests failed at least once at some point, but by fixing the code all were made to pass eventually.

A reference to these results is in Appendix A.5.

## 7.2 System tests

The system test serves partly as an integration test and a test of the user interface, and partly as a means of ensuring that the goals that have been set forth for the

system are fulfilled. The parts of the system that has been implemented must fulfill the requirements from the analysis and the design criteria.

Hence, the system test has been performed by going through the use cases and checking that all described functionality is available and working. The relevant design criteria are:

**Usable** This has been evaluated by inspection during the test of the use cases.

**Secure** Due to the design of the system, it should be secure. That it is in fact secure is difficult to test without the help of a skilled cracker. But we have tried entering URLs directly to circumvent the restrictions in the HTML-pages. Some errors were found this way.

**Efficient** During the use cases test, none of the operations took excessively long time – however, a proper test requires a much larger amount of data and simultanous users to see how the system scales.

**Correct** That the system is correct is the primary goal of performing the use cases test.

**Reliable** The robustness has been tested by deliberately trying to cause the system to malfunction, e.g. by entering invalid information in fields, and test whether it reacts sensibly.

**Comprehensible** This has been evaluated by inspection.

**Portable (frontend)** The system has been tested with both Internet Explorer and Mozilla as required as the technical platform.

**Portable (backend)** We have not tried to test this.

### 7.2.1 Test of reporting a defect

To illustrate how the system test has be done systematically we describe here as an example the test of reporting a defect. Figure 7.1 shows how the dedicated window looks like in the final system.

As shown in Figure 3.5 on page 26, it must be possible to enter the add report window. So the first step of the test is to verify that this is possible and easy to do. Clicking the “add a defect” link verified this.

Next, it must be verified that it is possible to choose a system and that it is possible to enter the relevant information (which is the subject and the description of the defect). Entering some information and clicking the “browse” button to select

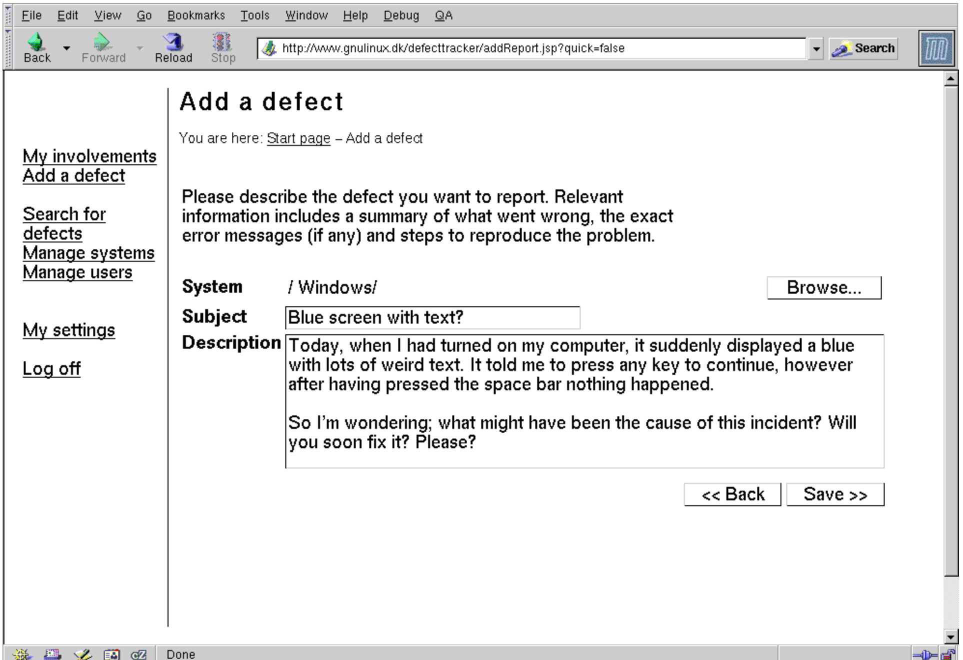



*Figure 7.1: The add report window.*

a system ensured that this was working – in the system tree it was also tested that it was only possible to select active systems. The final step of the test is to ensure that it is possible to submit the report, and that it is then saved. Clicking the "save" button and then searching for and viewing the report verified this.

It must also be tested that the system replies with helpful error messages when no system has been chosen or when one of the fields has been left empty. The various combinations of this were tried, each producing a red error message below the title, explaining the problem. Finally, the navigational contents must be checked, i.e. that the back button returns one to the previous page and that "you are here:" link is working. This was verified, too.

### 7.2.2 Test results

The conclusion after the system tests was that the system was working as intended, though with some flaws. These flaws were reported in the system itself and given priorities. Then focus was devoted to correcting defects carrying critical or high priorities.

The result is shown in Figure 7.2, which is the output from the statistical page of the system. Note that many of the defects are closer to suggestions for improving the system rather than actual errors. Appendix A.1.1 provides a guide for finding

known errors in Defect Tracker.

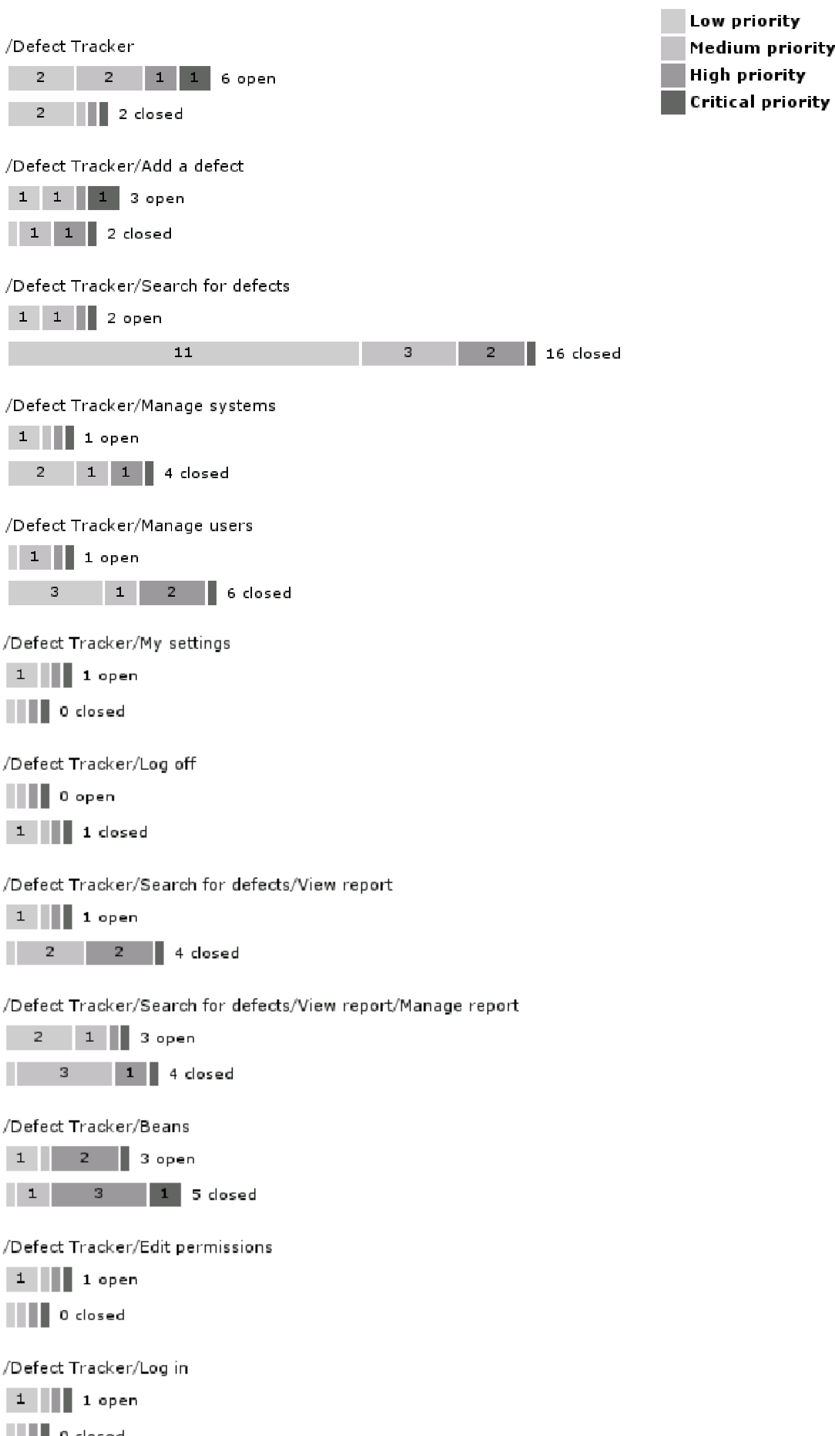



*Figure 7.2: System test results with open and closed defects generated by the statistical page.*

# Part V

# Study report

# Chapter 8

# The development process

This chapter reflects on selected aspects of the development process. Special attention is given to the algorithmic problem of searching for substrings efficiently, which the highlighting function of our system needs to do.

## 8.1 The analysis phase

A problem with the detail level of the analysis prolonged that phase.

The OOA&D method prescribes an extensive analysis before the actual design of the system. However, a substantial part of the method is devoted to converting the analysis of the problem domain into an implementable model component design, with special focus on how to identify further needed classes. Thus it seems that it is actually not necessary to include minor classes during the analysis as long as all events are still included. The events will then be transformed into the classes that are necessary to capture the information about the events.

In our analysis, some minor classes are included, e.g. the comment class and the permission hierarchy which could have been modeled with events only.

The problem with treating them as classes is that it makes the model more complicated without capturing more information about the problem domain. Apart from making the structure of the model more complicated, it also adds extra, rather trivial, descriptions, behavioural patterns, etc. which accompany the classes, making the analysis document longer and more difficult to get an overview of. For instance, Figure 2.4 and 2.7 are both trivial – the amount of new information captured in them is very low.

So, although our extensive analysis of the structure did save some time in the design phase since the events were already represented as classes, it would have been better to simplify the analysis by omitting these classes.

## 8.2 The design phase

Two problems were encountered during the design phase – one has to do with the design of the user interface and the other has to do with the problem of combining object-oriented design with relational databases.

### 8.2.1 Designing the user interface

The design phase as it is described in [10] is quite stringent. Although the OOA&D method emphasizes fitting the strategy to the particular situation, there is no recipe for how to actually proceed when the uncertainty is high as it was when we were to design the user interface architecture; noone in the group had previous experience with the servlet and server page architecture.

Our first approach was to attack the problem by trying to design analytically the entire interface on paper by describing what would be needed for each page from the user interface section in the analysis document. However, this quickly turned out to be very laborious and with such a high level of uncertainty that we could not be confident that the design was good enough.

According to *the principle of limited reduction* as stated in [11], this is a well-known problem:

> Relying on an analytical mode of operation to reduce complexity introduces new sources of uncertainty requiring experimental countermeasures. Correspondingly, relying on an experimental mode of operation to reduce uncertainty introduces new sources of complexity requiring analytical countermeasures.

The first part of principle is based on the observation that the problem is not properly understood through the analytical mode of operation which focuses on abstraction; some of the left out details may turn out to be of major importance. Conversely, an experimental mode can reduce this uncertainty but also produces new knowledge that needs to be dealt with analytically.

This view is supported by [16] which argues that though it is still worthwhile to try to approximate a rational, stagewise design process, it is never completely possible due to the uncertainties involved.

Hence, it appeared that it would be better to suspend the analytical mode of operation and instead proceed with experiments until the uncertainty had reached an acceptable level. So our course of action was to ensure that the decisions which would require the largest amount of time to change were carefully thought through, such as general patterns and where to put an extra effort to increase code reuse

and avoid copy-pasting. And then we began implementing the user interface. The design criteria that were considered, are:

**Usable** Most of the visual part of the interfaces were already designed in the analysis (Sec. 3.4). However, attention to the left-over details was needed to ensure user-friendly pages.

**Secure** Security was handled by the common pattern for checking the user session object. The security of the model component would catch any mistakes.

**Reliable** By taking non-sensible user input into account, the system was made more robust.

**Maintainable** Better maintainability was achieved by trying to reuse existing components instead of copy-pasting, and writing readable code.

**Comprehensible** To be comprehensible, attention was focused on making the visual part of the interfaces correspond to what actually happens inside the system to let the users form correct mental models of the program behaviour.

**Portable (frontend)** Portability was ensured since some in the group used Internet Explorer and some used Mozilla during the development of the system.

During the implementation it was found that most of the remaining user interface design was most effectively carried out by the individual programmer. Perhaps this can be attributed to the complexity of the interface. It is difficult to anticipate all the details that emerge when a coherent and sensible page is being implemented. This creates a high level of uncertainty which makes it hard and unproductive to try to design the pages beforehand.

On the other hand, the experimental mode of operation also causes problems. The structure of the code may end up quite diverse. This can be alleviated by increasing the communication between the involved parties, i.e. sometimes going to analytical mode. We did this sometimes.

### 8.2.2 Combining OOP with relational databases

The defect tracker utilizes a relational database for achieving persistence of the data. The language used for communicating with the database, SQL, is however very different from the object-oriented language of Java and the object-oriented nature of the OOA&D method. The clash between these two different paradigms makes it more difficult to design the model and function components. The problems

with the different type models and the necessary conversions are known as the *impedance mismatch* [4].

The problems are partly due to differences between the physical datatypes of the two languages as would be the case for most programming languages (e.g. how many bits are used to represent numbers), but mostly due to different organisations of the data. A relational database clusters the data in open tables whereas the object-oriented approach encourages distributing the data into encapsulated object networks formed by aggregations and associations.

The difference is most visible when retrieving the data. In an object-oriented environment it requires traversing the object network through the predefined accessor methods, whereas the relational enviroment joins the tables freely with no notion of class encapsulation.

Although it is possible to use the relational database as a simple storage means by saving and retrieving the individual attributes of the classes one at a time (support for this can easily be modeled by a single database class), it may be very inefficient since many values may need to be copied back and forth between the object-oriented program and the database. A search could be done entirely within the database instead of instantiating a lot of objects and retrieving a large part of the data of the database, even though most of the information is really not needed.

A more efficient approach has, in general, two possible resolutions:

1. Encapsulate the parts of the system that communicates with the database in a set of classes. The internal design of these cannot follow the object-oriented paradigm; however, apart from perhaps a slightly changed interface, the rest of the system can still pretend it is purely object-oriented.

2. Modify the programming environment so that it bridges the gap between the two paradigms: [20]

   (a) Extend the database so that it supports the object-oriented notions directly. This is known as an object-oriented database.

   (b) Extend the programming language so that it supports persistence and some efficient means of performing queries. The serialization approach in Java is a small step in this direction.

Unfortunately, the Java language does not properly support efficient persistence, and object-oriented databases are not yet in wide-spread use so a richer programming environment was not an option within the time frame of the project.

So the way we have chosen to alleviate the problem is to break encapsulation by letting some classes in the server component above the model component access

the database directly, and to introduce the persistent pattern for each class, thus changing the semantics of how the instantiations of the classes can be used. This makes the design less intuitive, but with the benefit of reducing the amount of data copied back and forth between our system and the database.

## 8.3 The test phase

The test phase brought up two subjects: whether to cover a greater amount of the system with unit tests, and how to handle the discovered defects.

### 8.3.1 Unit testing the user interface

Our unit tests only covered the classes of the database, model and function components. It would have been possible to create unit tests for the other components too, thereby being able to catch more regressions easily.

However, writing tests for the user interface is usually complicated by the fact that it is very difficult to foresee the combinations of actions that the users may perform, and it is difficult to simulate these actions since the interfaces are geared towards humans and not programming languages.

Since our system is using HTTP, the latter problem can be alleviated. There is even a framework for doing this [17]. Still, the investment of time needed to actually write down the tests is high, whereas a few mouse clicks can cover many cases quickly. Also, the benefit of testing the user interface with strict unit tests is smaller than the benefit of testing the model and function component in the same manner since errors in the former by their nature are more localised – a defect in the model component is likely to affect all components above it. Finally, manual system testing is still needed to ensure that the system is usable and that it fulfills the analysis requirements.

Hence, creating unit tests for the user interface is not such a good idea as it may seem – unless the correctness criterion is rated very high, which we have not done.

### 8.3.2 Handling discovered defects

During the implementation of the system a lot of issues emerge: an interface turns out to be suboptimal or some part of the functionality does not work. It is important that these issues are taken care of, but it is also important that the implementation effort stays focused. Thus some means of keeping track of the issues is needed. In other words, we needed a defect tracker.

So as soon as our system was running we began using it to track its own defects. The experiences have been positive; apart from the obvious benefit of being able to manage the defects easily (see section 1.1 and 1.2), it also helps the management of the project since it is easier to track what needs to be finished and also get an overview of what should be postponed.

Actually being a user of the system also helps discovering usability problems.

## 8.4 Faster highlighting

In the user interface, the search words are highlighted in the subject lines and in the comments. The comments may, however, contain quite large amount of texts so that much time is spent on searching for the positions of the words in the comments. Thus it is worthwhile looking into whether the simple algorithm that has been used for finding the positions can be improved.

First, we will take a step back and approach the problem in a more formal manner, describing two alternative algorithms; then the section ends with an empirical comparison of the described algorithms to conclude which is more suited for our context.

### 8.4.1 The string-matching problem

The problem of finding substrings in a string can be formalised as follows [5, Ch. 32]. The text is assumed to be an array $T[0, \ldots, n-1]$ of length $n$ and the pattern which is being searched for is an array $P[0, \ldots, m-1]$ of length $m$. The elements of the array are symbols drawn from a finite alphabet $\Sigma$, e.g. $\Sigma = \{a, b, \ldots, z\}$.

The pattern $P$ occurs with *shift* $s$ in the text $T$ if $0 \leq s < n - m$ and $P[i] = T[s + i]$ for $0 \leq i < m$. If $P$ occurs with shift $s$ in $T$ then $s$ is a valid shift, else $s$ is an invalid shift.

So the *string-matching problem* is the problem of finding the valid shifts with which a pattern $P$ occurs in a text $T$.

There are many different approaches and algorithms for the problem [3]. The most simple algorithm has a worst case running time of $\Theta(nm)$ and a best case running time of $\Theta(n)$. A better algorithm, the Knuth-Morris-Pratt algorithm, focuses on reducing the worst case running time (at most $2n$ character comparisons are performed). The Boyer-Moore algorithm tries to reduce the average case running time, still with the bound $\Theta(nm)$ in the worst case but now performing less than $n$ comparisons in the average case ($\Theta(n/m)$ in the best case).

Some other definitions are necessary. A string $p$ is a *prefix* of the string $x$ if

$x = py$ for some string $y$ that may be the empty string. This is denoted by $p \sqsubset x$ – e.g. *'ab'* $\sqsubset$ *'abcd'*. Likewise, a string $s$ is the *suffix* of a string $x$ if $x = ys$ for some string $y$ that may be the empty string. This is denoted by $s \sqsupset x$. The prefix of length $q$ of a string $P$ is denoted by $P_q$.

### 8.4.2 The simple approach

The most simple algorithm for performing the search is to run through all the shifts $s$ of the string $T$, comparing the characters in $P$ with $T[s, \ldots, s + m - 1]$:

```
1  for s := 0 to n − m − 1
2    for i := 0 to m − 1   // test this shift
3      if P[i] = T[s + i]
4        if i = m − 1
5          match found with shift s!
6      else
7        break    // try next shift
```

So in the worst case the algorithm runs through $m$ characters for each of the $n - m - 1$ possible valid shifts, giving a running time of $\Theta(m(n - m - 1))$ or $\Theta(nm)$. In the best case, the first character of the pattern never matches so that each execution of the inner loop stops immediately, resulting in $n - m - 1$ character comparisons and a running time of $\Theta(n)$.

### 8.4.3 The Knuth-Morris-Pratt algorithm

Obviously, the simple algorithm is sometimes very inefficient when a part or the whole of the pattern has been found since it then tries to match the pattern once again at the shift next to the just examined. For most patterns, this cannot result in a new match. For example, consider the text *'wisdom is not to be laughed at'* and the pattern *'laugh'*; when the *'laugh'* has been found, it is a waste of time to try to match it against *'aughe'*.

Instead the algorithm may jump over the match and start from *'ed at'*. In the general case, it is not as simple, though. For instance, the pattern *'aba'* and the string *'ababa'* has two valid shifts. It explicitly needs to be computed how many characters it is possible to skip.

The Knuth-Morris-Pratt algorithm [5, Sec. 32.4] involves an efficient way of doing this. A prefix function $\pi(q)$ is introduced. When a character that does not match the next character of the pattern is encountered, $\pi(q)$ returns the greatest

number of matching characters of the pattern that may still give a valid shift given that only the $q-1$ first characters are considered.

For example, given the text *'hoho, he laughed'* and the pattern *'hohoho'* the first part *'hoho, he...'* of the text can be matched. Trying the next character does not yield a match, though, so instead $\pi(4)$ is computed which gives 2 since the two first characters, *'ho'*, is the longest prefix less than 4 of the pattern that still matches the read characters (i.e. *'hoho'*) and may result in a valid shift.

Consequently, the Knuth-Morris-Pratt algorithm shifts searching position in the text so that the first two characters are matched (*'ho, he...'*) and continues from that position. This way the algorithm has avoided one comparison compared to the simple approach – trying to match *'oho, he...'* with the pattern is predetermined to be fruitless.

So more formally, the prefix function of a pattern $P$ of length $m$ is the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ that is length of the longest prefix shorter than $q$ which is a suffix of $P_q$ [5]:

$$\pi(q) = \max\{k \,|\, k < q \wedge P_k \sqsupset P_q\}$$

To actually save any comparisons, the prefix function of a given pattern must be precomputed. This is not a problem since the domain of the function is of size $m$, and $m$ is typically much smaller than $n$.

It is possible to compute each value of $\pi(q)$ for a pattern $P$ by comparing $P_{q-1}$, $P_{q-2}$, ..., $P_1$ with $P_q$ backwards. For example, for $q = 4$ comparing the prefixes *'hoh'*, *'ho'* and *'h'* with *'hoho'* (no match), *'hoho'* (match) and *'hoho'* (no match), so $\pi(4) = 2$. However, this gives a running time of $\Theta(m^2)$ in the worst case.

By observing that if $P_i \sqsupset P_q$ for some integer $i$ where $1 \leq i < q$ then $P_{i-1} \sqsupset P_{q-1}$, it is possible to reuse $\pi(q-1)$ to reduce the running time, see Figure 8.1.

Shifting $P_{q-1}$ so that $\mu = \pi(q-1)$ characters are matched, the algorithm just need to compare $P_{q-1}[\mu]$ with $P_q[q-1]$. If it matches, the computation stops, else a shift of $P_{q-1}$ for $\pi(\mu) = \pi(\pi(q-1))$ characters is tried instead, and so on until $\pi(\ldots)$ is zero, indicating there is no further possibilities. When the computation stops if the characters compared last matched then $\pi(q)$ must be $\mu + 1$, else $\pi(q)$ is 0.

Written in pseudo code, the algorithm can be expressed as:

```
1  π(1) := 0
2  for q := 2 to m
3    μ := q − 1
4    repeat                // loop through each possible shift
5      μ := π(μ)
```

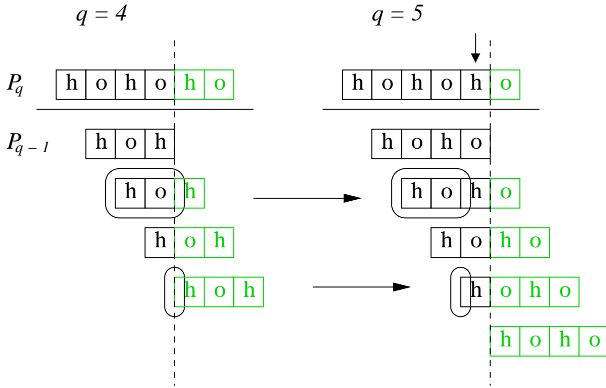



*Figure 8.1: Illustration of how the solution of $\pi(4)$ can be used to compute $\pi(5)$. For $q = 5$, the possible matches (marked) must be present in the matches for $q = 4$. The algorithm just needs to compare the fifth character (as indicated by the small arrow) of $P_q$ with, in the worst case, one character from each of the selected shifted patterns; in this case, 'hoho<u>h</u>' and 'ho<u>h</u>' are matched immediately. So $\pi(5) = \pi(4) + 1 = 3$ for this particular pattern.*

```
6     while μ ≠ 0 and P[μ] ≠ P[q − 1]
7     if P[μ] = P[q − 1] // found a valid shift
8       π(q) := μ + 1
9     else
10      π(q) := 0
```

With the prefix function precomputed, the Knuth-Morris-Pratt algorithm for solving the string matching problem can be presented. It works by running through the characters in the text ($i$), keeping track of how many characters of the pattern that have been matched ($q$) and shifting the pattern each time a mismatch is found. Each time the match count equals $m$, the pattern has been found:

```
1  i := 0
2  q := 0
3  while i ≠ n              // examine characters one−by−one
4    if P[q] = T[i] then
5      q := q + 1           // one more character matched
6      if q = m then
7        match found with shift i − (m − 1)!
8        q := π(q)
9      i := i + 1
10   else
11     if q ≠ 0 then
```

```
12          q := π(q)        // slide the pattern to try again
13        else
14          i := i + 1
```

The observation [9] that at each step of the algorithm, a character comparison is made and either the pattern is shifted to the right or the position in the text $i$ is incremented makes it clear that at most $2n$ character comparisons are performed since it is the pattern and the position can only be shifted right at most $n$ times. On the other hand, at least $n$ comparisons are always performed since each iteration begins with a character comparison and the counter runs through $0, 1, \ldots, n-1$.

### 8.4.4 The Boyer-Moore algorithm

Although the Knuth-Morris-Pratt algorithm improves the worst-case running time for the string matching problem from $\Theta(mn)$ to $\Theta(2n)$, it does not affect the lower bound on at least $n$ comparisons. The algorithm of Boyer and Moore [2] is based on another approach and improves this bound.

The key observation is that starting the matching process from the right end of the pattern provides more information if a mismatch is encountered. Consider once again the pattern *'laugh'* and the text *'wisdom is not to be laughed at'* with the pattern positioned at shift 0 – since *'o'* (from *'wisdom'*) does not occur in the pattern, the shifts 0–4 are immediately known to be invalid and the pattern can be shifted 5 characters without comparing with any of the first 4 characters of the text. Thus it is possible to do with less than $n$ comparisons.

The algorithm works by comparing the characters of the pattern from right to left. When a mismatch occurs, the pattern is shifted a number of places to the right and the algorithm begins trying to match it once again from the right end. The pattern shift $s$ is at least 1, but may usually be safely enlarged at most mismatches by the help of two auxiliary precomputed functions, $\delta_1$ and $\delta_2$. In pseudo code it looks like this ($i$ is the examined text position and $q$ the position in the pattern):

```
1  i := m − 1
2  while i < n
3    q := m − 1
4    while T[i] = P[q]        // this is a possible match
5      if q > 1 then          // continue matching
6        q := q − 1
7        i := i − 1
8      else
9        match found with shift i − 1!
```

```
10         break
11      i := i + max(δ1(T[i]), δ2(q))  // jump to next possible shift
```

The $\delta$ functions each compute how long the shift at least must be for it to be a valid shift so to save as many comparisons as possible the largest value is chosen.

The $\delta_1(c)$ function returns the distance between the rightmost occurrence of a character $c$ and the right end of the pattern, or $m$ if $c$ is not in $P$. More precisely, it can be defined as

$$\delta_1(c) = \begin{cases} \max\{k \mid P[k] = c\} & \text{if for some } i,\, P[i] = c \\ m & \text{if for all } i,\, P[i] \neq c \end{cases}$$

where $c \in \Sigma$ and $0 \leq i < m$.

It based on the observation that it is safe to slide the pattern so that the rightmost character is aligned with the mismatching character from the text, without having to check for matches. As mentioned, if the character is not in the pattern, it is possible to slide the pattern all the way past the character in the text. Likewise, if the rightmost character is to the left of the current position in the pattern, the algorithm can skip the comparison of a number of characters. For example, consider the pattern '*axaaaa*' and the text '*... bbbbxaaaa...*'. With the following setup where '*axaaaa*' and '*... bbbbxaaaa...*' have been compared, it is possible to slide the pattern three characters:

```
         ↓                       3
     axaaaa                    ──→
  ...bbbbxaaaaa...          axaaaa
                         ...bbbbxaaaaa...
```

If the rightmost character is to the right of the current position in the pattern, aligning the pattern with the text would require sliding it to the left instead of right which is redundant since that part of the text has already been checked. So only in some circumstances is the computation of $\delta_1$ beneficial.

After having slided the pattern, the algorithm should start trying to match it once again from the right end. So the number of characters it needs to move the current position in the text is actually $\delta_1$; the pattern itself is "moved" by resetting its counter (*q* in the algorithm) to its rightmost character. For example, the position in the text above ends up as this:

```
         ↓                             ↓
     axaaaa                    axaaaa
  ...bbbbxaaaaa...          ...bbbbxaaaaa...
                                 ──→
                                  4
```

The $\delta_1$ function can be precomputed for a pattern easily by finding the rightmost character of each character in the pattern:

```
1  for c ∈ Σ          // initialize
2     δ1(c) := m
3  for i := 0 to m − 1
4     δ1(T[i]) := m − 1 − i
```

The fastest way of storing the precomputed values is with an array; however, if $|\Sigma|$ is large, the cost of initialising the array may dwarf the actual string matching. Another possibility is to use a hash table with only those characters that are actually present in the pattern.

The $\delta_2(\mu)$ function returns the distance between the right end of the pattern and rightmost plausible reoccurrence of the $\mu$ last characters of the pattern. A plausible reoccurrence is roughly speaking an occurrence of the $\mu$ characters for which the character to the left of the occurrence is different from the character to the left of the $\mu$ last characters. For example, the rightmost plausible occurrence for $\mu = 2$ for the pattern '*xabcab*' is '*xabcab*' ($\delta_2(2) = 5$) whereas for the pattern '*xabcabcab*' it is '*xabcabcab*' ($\delta_2(2) = 8$).

The reason the above explanation is roughly speaking, is that a plausible reoccurrence may also start off the left end of the pattern (without matching all of the $\mu$ characters), e.g. for the pattern '*bcabcab*' the reoccurrence is '*_bcabcab*' ($\delta_2(2) = 8$). More formally, the $\delta_2$ function can be expressed as

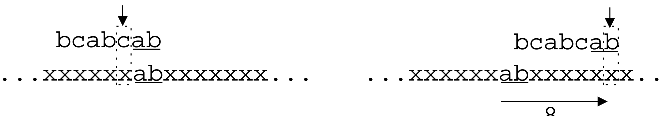
$$\delta_2(\mu) = \min_{-\mu \leq i < m-\mu} \left\{ m - i \;\middle|\; \begin{array}{c} P[i, \ldots, i+\mu] = P[m-\mu, \ldots, m-1] \\ \wedge\, P[i-1] \neq P[m-\mu-1] \end{array} \right\}$$

where $i$ runs through the possible shifts and the resulting substrings are compared with the last $\mu$ characters of the pattern, $P[m-\mu, \ldots, m-1]$. To make the above expression well-defined, comparisons of characters to the left of the leftmost character of the pattern are defined always to be true, i.e. $P[-j] = c$ and also $P[-j] \neq c$ for any positive integer $j$ and any character $c$.

The observation that $\delta_2$ is based on is similar to that of $\delta_1$: when a mismatching character is encountered, it is safe to slide the pattern so that the already matched $\mu$ characters are aligned with the next plausible reoccurrence in the pattern (which, according to the above definition, may be completely off the left edge of the pattern). For example, a trivial situation ($\delta_2(2) = 5$):

```
     xabcab                        xabcab
...xxxxxxabxxxxx...        ...xxxxxabxxxx...
                                  --->
                                   5
```

And an example of how the next plausible occurrence starts off the left edge ($\delta_2(2) = 8$):




As with $\delta_1$, after having slided the pattern, the algorithm should continue trying to match it from its right end, so the text position can simply be incremented by $\delta_2$ and the pattern counter $q$ set to the rightmost character, as the above examples illustrate (the current position is moved 5 and 8 characters to the right, respectively).

Precomputing the $\delta_2$ function can be done in the obvious way by starting from the right end of the pattern and trying to match the last $\mu = 1, 2, \ldots, m$ characters with the shifts $m - \mu$ to $-\mu$, stopping as soon as a plausible occurrence is found. This turns out to have the running time $\Theta(m^3)$ in the worst case, see Figure 8.2 – instead, Knuth [9] gives a more complex algorithm similar to the precomputation of the $\pi$ function of the Knuth-Morris-Pratt algorithm, with the running time $\Theta(m)$.

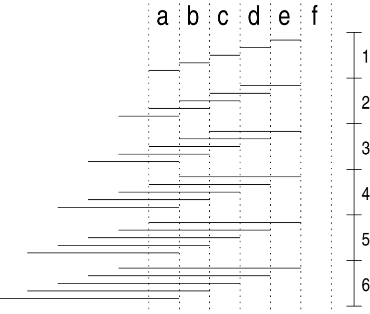



*Figure 8.2: When precomputing the $\delta_2$ function, the simplest approach is to try to match the rightmost character with the possible shifts, then the two rightmost characters with the possible shifts, etc. Since there are $m - 1$ such shifts and $m$ substrings that needs to be checked ($1, 2, \ldots, m$ rightmost characters), with each check using on average $m/2$ character comparisons, the total number number of comparisons is $\Theta(m^3)$.*

The running time of the algorithm itself depends on the pattern and the text. In the best case, all characters read from $T$ do not occur in $P$ so that it is always possible to move forward $m$ positions (with the help of $\delta_1$), consequently using only $\lfloor n/m \rfloor$ character comparisons.

In the worst case, the algorithm however exhibits the same behaviour as the worst case of the simple string matching algorithm. Consider the pattern '*aaaa*' and the text '*aaaaaaaaaa*'. The first match is '*aaaaaaaaaa*', and the algorithm must then start matching from the character to the right of the match, comparing the characters '*aaaaaaaaaa*', since shift 1 is also a match. And so on with '*aaaaaaaaaa*',

'*aaaaaaaaaa*' etc. Thus most matches involves comparing characters that have already been compared several times, and the algorithm has degenerated to a reverse form of the simple matching algorithm with the running time $\Theta(nm)$.

The average case behaviour depends on the probability of a character from the pattern and a character from the text matching [2] and on the length of the pattern. If the probability of a pattern character and a text character matching is low, the chance of being able to skip portions of the text safely is higher, whereas a longer pattern can make it possible to skip larger amounts. A large alphabet may lower the probability of characters matching.

Under some simplifying assumptions, a theoretical analysis in [2] suggests, however, that the number of character comparisons in the average case is well below $n$ (e.g. $\frac{1}{5}n$ for patterns of length 8 with a probability that resembles that of English text). This is supported by empirical evidence, and is also confirmed by our results in the next section.

### 8.4.5 Comparison of the algorithms

The difference between the simple algorithm and the Knuth-Morris-Pratt and Boyer-Moore algorithms is that the latter have a preprocessing phase where their behaviour is optimized based on information extracted from the pattern only. The information which the Knuth-Morris-Pratt algorithm extracts is used to optimize the cases when either the whole or most of the pattern is found, whereas the information extracted by the Boyer-Moore algorithm is most useful when matches are seldom.

This has a major impact on the efficiency of the algorithms when searching in ordinary text, such as this report. The algorithms can only save a few comparisons if the pattern is short, and longer patterns usually occur rarely. Hence, the problem of the simple algorithm with redundant comparisons in the event of a partial match is seldomly encountered and the number of comparisons is close to $n$.

Trying to enhance the search by using the Knuth-Morris-Pratt algorithm is futile – the added complexity may even make it more difficult to further optimize the algorithm when implementing it for a specific architecture, thus rendering the matching process slower in practice. Conversely, the best or nearly-best situation will often happen for the Boyer-Moore algorithm, and the worst case behaviour is never encountered since long sequences of the same short pattern are very uncommon.

To support these assertions, we have empirically tested the three algorithms on a large corpus of text (318 kb) from an existing defect tracker system [1]. The

three algorithms are versions of already written reference implementations from [3], slightly modified to count the total number of character comparisons done during the matching phase. 1000 patterns of a given length was picked randomly from the corpus and searched for, averaging the resulting number of character comparisons. The result of running the three algorithms with different pattern lengths is shown in Figure 8.3.

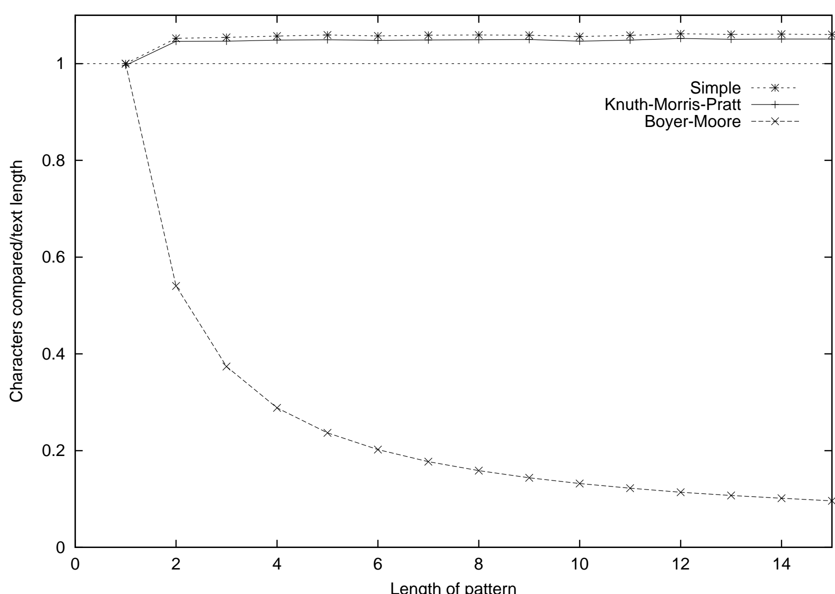



*Figure 8.3: A comparison of the efficency of the three algorithms on English text in terms of the number of compared characters per character in the text. The simple algorithm and the Knuth-Morris-Pratt algorithm both examine each character a little more than once, on average, with the simple algorithm comparing about 1% more characters. On the other hand, The Boyer-Moore algorithm, with a pattern length above* $3$*, compares only less than* $1/4$ *of the characters, on average.*

The running times of the three algorithms are summarized in table 8.1.

| | Simple | KMP | BM |
|---|---|---|---|
| Worst | $\Theta(mn)$ | $\Theta(2n)$ | $\Theta(mn)$ |
| Best | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n/m)$ |
| Average | $1.06n$ | $1.05n$ | $0.10n$-$0.37n$ |

*Table 8.1: Overview of the three algorithms with the average values empirically determined by our test (with pattern lengths of 3–15).*

### 8.4.6 Conclusion

Clearly, using the Boyer-Moore algorithm would be an improvement over the simple algorithm for finding words to highlight in our system.

The only potential problem with implementing it is the dependence on the alphabet size when precomputing $\delta_1$. As the system is implemented now, it is not a problem since only 8-bit characters are supported so $|\Sigma| = 2^8 = 256$. However, if the system is extended to e.g. 16-bit characters, filling a table of $|\Sigma| = 2^{16} = 65536$ entries is impractical.

Since only at most $m$ values actually have to be stored (all characters from the alphabet not appearing in the pattern are mapped to the value $m$), some sort of hashing would be an option. It has to be extremely efficient with respect to time, though.

Instead it would be simpler to split the 16-bit characters into two 8-bit parts and perform an 8-bit search, although this increases the probability that a character from the text and a character from the pattern match, worsening the $\delta_1$ and $\delta_2$ estimates. In case of two 16-bit characters matching, it also doubles the number of necessary comparisons.

The problem with the worst case running time of the Boyer-Moore algorithm is that it does not notice that it has already matched some characters when the pattern is slided to the right. By modifying the algorithm so that it remembers this, the so-called Turbo-BM algorithm [3], it is possible to achieve a worst case running time of $\Theta(2n)$. Since the situation is unusual in ordinary text, the question is whether the modification is an improvement.

# Bibliography


[1] Bugzilla – the Mozilla bug database. `http://bugzilla.mozilla.org/`.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communication of the ACM*, 20(10), October 1977.

[3] Christian Carras and Thierry Lecroq. Handbook of exact string-matching algorithms, 2002.

[4] J. Chen and Q. Huang. Eliminating the impedance mismatch between relational systems and object-oriented programming languages. The 6th International Hong Kong Database Workshop, 1995.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[6] Erich Gamma and Kent Beck. Junit. `http://junit.sourceforge.net/`.

[7] ECMA Standardizing Information and Communication Systems. Ecmascript language specification, 1997. `http://developer.netscape.com/docs/javascript/e262-pdf.pdf`.

[8] jGuru. Java server pages fundamentals, September 2000. `http://developer.java.sun.com/developer/onlineTraining/JSPIntro/`.

[9] Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[10] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object-oriented Analysis and Design*. Marko, 2000.

[11] Lars Mathiassen and Jan Stage. The principle of limited reduction in software design. *Information Technology & People*, pages 171–185, 1992.

[12] Sun Microsystems. Javabeans(tm). `http://java.sun.com/products/javabeans`.

[13] Sun Microsystems. Javadoc. `http://java.sun.com/j2se/javadoc/`.

[14] Sun Microsystems. Javaserver pages(tm) technology. `http://java.sun.com/products/jsp/index.html`.

[15] Sun Microsystems. Java(tm) servlet technology. `http://java.sun.com/products/servlet`.

[16] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, pages 251–257, 1986.

[17] The Jakarta Project. Cactus. `http://jakarta.apache.org/cactus/`.

[18] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Dynamic modification of documents, 1999. `http://www.w3.org/TR/1999/REC-html401-19991224/interact/scripts.html#h-%18.2.4`.

[19] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Html 4.01 specification, 1999. `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[20] F. D. Rolland. *The essence of databases*. The essence of computing series. Prentice Hall, 1998.

# Appendix A

# References to Defect Tracker

## A.1 Running version

Defect Tracker is running at

`http://www.gnulinux.dk/defecttracker/default.jsp`

By using the “Create new user” link on the login page, it is possible to create a new user account, or one can use the following super-user account:
Username: test
Password: test

### A.1.1 Finding the open defects in our system

Steps to see the known defects in Defect Tracker:

1. Go to the running version as explained above.
2. Click “Search and manage defects”.
3. Login using the super-user account.
4. Select “Search for defects” in the menu.
5. Click on the “Browse...” button and select the system “Defect Tracker”.
6. Click on the “More options...” button and select the “Open” radio-button under “Status”.
7. Click on the “Search” button and all unfixed defects in Defect Tracker should be displayed.
8. Click on a subject to see details of a specific defect.

## A.2 System demonstration

A live demonstration of the system is available at:

`http://www.cs.auc.dk/~jasper/dat1/demo/`

## A.3 Source code of defect tracker

The source code available in two formats. One file containing all source files (`http://www.cs.auc.dk/~jasper/dat1/package/`) and all the files in pretty-printed HTML format:

### A.3.1 Overview of our source code

- `http://www.cs.auc.dk/~jasper/dat1/sc/`

### A.3.2 Source code of the model component

- `http://www.cs.auc.dk/~jasper/dat1/sc/model/`

### A.3.3 Source code of the function component

- `http://www.cs.auc.dk/~jasper/dat1/sc/function/`

### A.3.4 Source code of user interface components

- `http://www.cs.auc.dk/~jasper/dat1/sc/ui/`

### A.3.5 Source code for unit tests

- `http://www.cs.auc.dk/~jasper/dat1/sc/unitTest/`

## A.4 JavaDoc

The documentation is available in JavaDoc at:

`http://www.cs.auc.dk/~jasper/dat1/javaDoc/`

## A.5 Test results

The test results from the unit test are available at:

`http://www.cs.auc.dk/~jasper/dat1/testResults/`