Group s304a
SW3, 2009
Department of Computer Science
Aalborg University

presents

Project Management System

Front page contains photos and graphics from [RAW09], [gui09], [Cob09] and [maj09].

Title:
Project Tool for Agile Development

Theme:
Programming

Semester:
SW3, Fall 2009

Group:
s304a

Students:

Mette Bank

Dianna Hjorth Kristensen

Jacob Bang

Lasse Linnerup Christiansen

Jens Kristian Stride Geyti

Christoffer Hjortlund Heder

Jeppe Pihl

Supervisor:
Darius Sidlauskas

Copies: 9
Pages: 108
Published: December 18th 2009

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East
http://www.cs.aau.dk

Synopsis:

This report documents the development of a project management tool for agile development aimed to be used by student groups. The emphasis is put on analyzing requirements, designing an appropriate architecture, and programming using the object oriented paradigm. Chapter one introduces the report and raises the initiating questions. Chapter two and three are based on the method of Object Oriented Analysis and Design. In these chapters the requirements for the management system is found and described. This has been done to meet both the requirements from the study regulation and project proposal, as well as to design a suitable architecture. The fourth chapter concerns the implementation of results from analysis and design. The attention is put on the architectural pattern MVC, as it is the foundation of this development and supports the analysis and design from the previous chapters. Chapter five encompasses different testing approaches, with unit testing of selected functions and an overall stress test of the system. In the last chapter, difficulties faced during the development is discussed. Subsequently a reflection of the process and improvement for future versions is presented, and a conclusion drawn. Finally the management system will be put into a larger context.

Preface

This report is written to document the 3rd semester project, compiled by group s304a - Software Engineering students from Department of Computer Science at Aalborg University. The report covers the development of a software project management system including the theories, methodologies and technologies this imply.

As a precondition for reading this report, the reader is expected to have basic knowledge about C#, ASP.NET, SQL and the object oriented method of system analysis; OOA&D.

The report is divided into six chapters. The first chapter gives an introduction to the project and defines the initial problems and the focus. The following two chapters concerns system analysis and design of the system to be developed. The two chapters are structured by using the approach to system analysis approved by OOA&D. Chapter four concerns the implementation of the system. In this chapter, the analysis is taken into account, and used for describing how and why the system has been implemented as it has been. Tests of the system will be concerned in chapter five, and finally, the report ends by a recapitulation in chapter six.

When the acronym *PMS* is used throughout the report it refers to the developed Project Management System. When the term *he* has been used to specify a person in the report, it refers to *he/she*.

References are denoted using the following syntax: [InitialsYear]. For instance, if the authors: *Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen* and *Jan Stage* have written a book, which has been published in year 2001, a reference to this book will be denoted: [MMMNS01]. In some cases, references will be followed by a chapter and or a page indication. Using the same example as before, a such reference will be denoted [MMMNS01, chapter 9, page 173].

The included DVD contains the source code of the developed system and the tests, as well as the SQL-queries for creating database tables with test data. Furthermore the test results and the report in PDF-format is included on the DVD.

Group s304a would like to thank supervisor Darius Sidlauskas, Department of Computer Science at Aalborg University to contribute with constructive feedback and technical experiences.

Enjoy reading the report.

Group s304a

Approach

The approach for writing this project has been inspired by the use of project related courses in System Analysis and Design (SAD) and Object Oriented Programming (OOP). As we were expected to learn about the method of Object Oriented Analysis and Design (OOA&D) through the lectures in SAD, we have chosen to build the project upon the use of this method. By doing so, we have approached the project by emphasizing the analysis.

Using this approach, the project aims to analyze and describe the requirements and architecture of the final system. When it comes to programming the outcome of the analysis, we have approached this, by the use of an agile development method. As the main part of the group had earlier experiences with eXtreme Programming (XP), it was reasonable to choose this method.

In the following, the basics of XP will be explained. This is not meant as a thorough walkthrough but solely to display how the development phase of the final system has been inspired by some of the rules in XP.

In XP, the basic development process is concerning user stories, tasks and iterations. First some user stories are created, reflecting the customer's function requirements for the software to be developed. Then the development team estimates how long it will take to code the created user stories. Followed by that the customer defines the priority of the user stories, and in cooperation with the developers, an overall release plan is defined. A release plan often consists of small releases based on iterations. For each release plan the customer defines which user stories should be implemented [Wel99d].

An iteration is the period of time where selected user stories are coded and implemented. At the beginning of an iteration, the customer defines which user stories are to be implemented. Next, the developers starts writing tasks for the user stories, which have been planned for the current iteration. In the next period of time, the developers are working on implementing the user stories, by solving the tasks which have been defined for each single user story. When the iteration has reached its end, the team releases a small piece of functional software which fulfills the requirements of the user stories. At last the customer is invited to take a look at the result, clarifying if he is happy with the solutions or not [Wel99b].

The rules of XP are tangible tools to help developing well-considered software. In the following, the rules applied in our development phase, will be described.

Release Planning is a meeting where the overall release plan for the system is defined. User stories are written and the developers estimate how long it will take them to implement each particular user story. Then, each user story is prioritized by the customer and release plans are specified on behalf of this. A release plan defines which user stories must be implemented for each particular release of the system [Wel99d].

Iteration Planning follows the release planning. A release is divided into iterations, and at the beginning of each iteration, an iteration planning meeting must be called. Holding this meeting lets the customer define which user stories are most valuable and which must be implemented in a particular iteration. In our case, the user stories would reflect the project requirements. The customer, which is us in cooperation with our supervisor, is able to choose between the user stories reserved for the particular release. If some previous user stories have not passed the acceptance test, that is, the customer was not satisfied with the implementation, those user stories may be planned to be re-visited in a new iteration. Using iteration planning is good, as it allows us to verify that we are doing right, and that we are reaching our goal of solving the project requirements [Wel99b].

Pair Programming is a technique where two programmers sit together at one computer solving a specifik task. One is typing code while the other reviews each line of code typed, trying to think about possible errors. Using pair programming increases the learning outcome, as prior knowledge among developers is easily passed on [Wel99c].

Refactoring can be defined as Martin Fowler says in [FBB+99, Preface, page xvi]: *'Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.'* Refactoring is usually done in small steps, where cleaning up code is done by removing or rewriting parts of the code without changing the functionality. Refactoring has been used to make the code more simple and easier to understand. After refactoring, it is also easier to change the functionality if needed [CC08].

Acceptance Tests belongs to user stories. Running an acceptance test includes the customer to specify scenarios for when the particular user story has been solved. Acceptance tests must be validated by the customer, and it is his responsibility to define when a user story has passed its acceptance test. Usually, acceptance tests are built upon the concept of black-box testing that is, the functionality representing the user story is given some specific input, and if the output is similar to what is expected, then the user story has passed its acceptance test [Wel99a].

Continuous Integration is about integrating newly created code into the rest of the system as quick as possible. In that way, developers avoid integrating big chunks of code, and thereby minimizes the risk of code conflicts [Bec02].

Contents

Introduction 1

Collaboratively developing larger software projects can be quite a daunting task. The fact that each developer's individual contributions to a project must all come together as a complete working software system, induces strong challenges to team collaboration. Ending up with a system that works as intended is obviously the main goal of software development. However, knowing how to get there within a deadline or on a budget while maintaining good code quality and proper documentation is what calls for a well-organized development process.

Depending on what is being developed, who is developing and who the customer is, there will be different demands and considerations as to what is necessary in order to get from initial idea to final product. Factors like security, product cost, code efficiency and code maintainability must be known and addressed differently depending on this.

Software development within typical student groups introduces considerations not commonly found in commercial programming, as learning outcomes and documentation of the actual development process is strongly favored to the quality of the final product.

This project aims at finding and analyzing problems within a student group's development process in order to produce a software design, and product, that solves these problems. Emphasis is placed on thorough analysis and design in order to document how to actually generate a complete common foundation with a possible customer, as to what the final software solution should be. Instead of simply programming software from a customer's initial requirements, a thorough analysis of both the customer's problem- and application domain ensures common understanding of exactly what and how a software solution should solve the found problems. The questions that will be analyzed and solved throughout this initial process are the following:

- How does a student group administer software development, without using a general purpose bug- and development tracking system?
- How can this process be supported by such a system, without supplying unneeded functionality?

1.1 Focus

This project will focus on developing a project management system for student groups. The object of the system will be to support several groups and projects, making it possible for Universities, for instance, to make them publish a project management system, as a service to their students. The typical student group in mind is heavily inspired by the one writing this project. These decisions have been reached for numerous reasons. Developing a project management system that would be better than existing alternatives is highly unlikely, due to the deadline, the learning outcome of the project as well as the programming experience among the developers. The predominant learning outcome of the semester in which this project is written, has been to learn how to

> *"gain fundamental techniques in solving software technical problems, and gain experience with programming of large systems, division of labor and quality control including testing"*[Uni09a].

Finding a suitable student group to use for analysis and testing is omitted in favor of focusing on proper and correct analysis and programming techniques.

This means that the project management system is being developed to student groups that:

- work in a smaller group of five to seven developers.
- have fundamental ideas of how to use an agile system development process.

Analysis 2

The purpose of this chapter is to determine the overall system characteristics of the system to be built. Further the chapter will concern identifying the parts of the system context which can be administrated, controlled and monitored by a system. This will be done by examining and model the problem domain. Finally the chapter will concern the usage requirements of the system to be build, by describing the application domain. The object for this chapter is to create a foundation to build a system architecture upon. This chapter has been written with [MMMNS01] as the primary source.

2.1 Pre-Analysis

At the beginning of a software development project, it is often difficult to measure which challenges are most important. Some challenges must be addressed early in the project to prevent them leading to other problems down the road; other challenges might not be too important in the beginning, and may be implemented later on. When working with definition of challenges, it is important to get all involved to agree at a specific level of interpretation.

To do so, the creation of a system definition can be a helpful tool. A system definition is a concise description of a computerized system expressed in natural language. The definition is helpful when trying to compare and distinguish different interpretations of challenges in a system. It describes what information the system should contain, which functions to provide, where the system should be used and finally, it serves as a foundation for continuing analysis.

A system definition consist of two subactivities; describing the situation where the system is to be used, and an activity concerning an innovative part, where ideas are created to come up with useful features, which not yet have been concerned. This section will consider these two subactivities and end up with a final system definition.

2.1.1 Rich Picture

The idea of starting the development process with drawing a rich picture, is to inspire the development team into thinking about usage patterns and problems in a more creative way, and not just solely by writing and discussing. Since the development of a rich picture is done in a non-technical way, the process will be ensuring that all involved think the same way, as developers and end users are less likely to misunderstand one another when forced to actually explain problems through drawings.

The developed rich picture will be used as a pre-analysis tool for understanding and agreeing upon problems in a specific situation. In this case, the situation will be concerning students and their typical approach to an agile development process using XP. For this project, a rich picture is a great chance of explaining exactly what problems might occur when students are developing software in their groups.

A rich picture concerning this project has been developed in figure 2.1. The rich picture is illustrating a student groups' process and problems of developing using an agile development method managed manually. A project and its requirements to a program gets translated to user stories (first arrow, starting from the top left corner). For each user story, a set of tasks is written (second arrow). Developers read and update their task list continuously, a process which is reflected on a manually updated time schedule and burn down chart. When all tasks are done, the team present a small release of the program to their supervisor, and the process may continue for another iteration if required. Crossed swords points out problems in this development process.

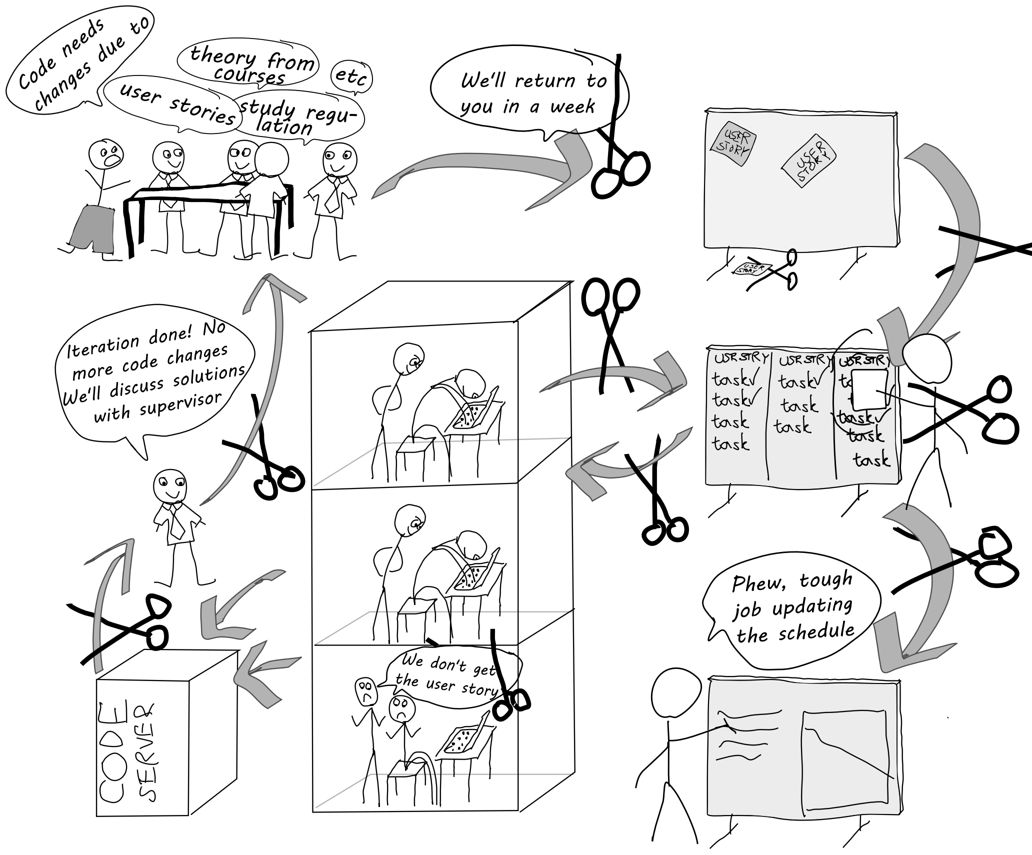

Code needs changes due to
user stories
theory from courses
study regulation
etc
We'll return to you in a week
USER STORY
USER STORY
USR STRY
task✓
task✓
task
task
USR STRY
task✓
task
task
USR STRY
task
task
task
Iteration done! No more code changes We'll discuss solutions with supervisor
Phew, tough job updating the schedule
We don't get the user story
CODE SERVER

Figure 2.1. The rich picture illustrating the situation where a student group is manually managing an agile software development process.

The picture itself is constructed of different elements. People and objects are black while processes and work flow is visualized with grey arrows. This way the situation gets both explained and outlined. At this point, however, looking at the drawing, it becomes possible to locate specific problems in the way projects are currently developed. These problems, which have been marked with crossed swords in the rich picture, are listed below:

- Whenever a request for certain functionality has been made (by the supervisor for instance), hence starting an iteration, it is difficult for the supervisor to monitor the

project development progress, as all planning happens internally, and manually, at the development team's location.
- Data loss may occur when managing user stories and task lists manually – post-it notes with user stories may be lost and tasks may be accidentally wiped.
- Reflecting task list changes manually onto time schedules and burn down charts, is a cumbersome and error-prone process.
- As developers almost solely work by solving tasks assigned to user stories, it may be difficult to gain required information about how to understand and solve a certain problem.
- All code changes are committed to a centralized server, but as it is difficult for the supervisor to monitor both the tasks completion status as well as the solutions themselves, the only viable solution is to show the supervisor a small release at the end of every iteration.

Besides both explaining, outlining and specifying the situation, the rich picture shows a set of objects that can be used within the problem domain, such as developer, user story and task list. Both the initial agreement upon the situation, as well as the objects within it, are subjects that will be used directly in the following analysis of the problem- and application domain.

2.1.2 Create Ideas

This activity concerns the process of acquiring new ideas for the system to build. There might have been features which not have been considered yet, and therefore, this process is used to ensure, that new ideas might get an impact on the final system. This is done by considering other exemplars of similar systems and by studying metaphors.

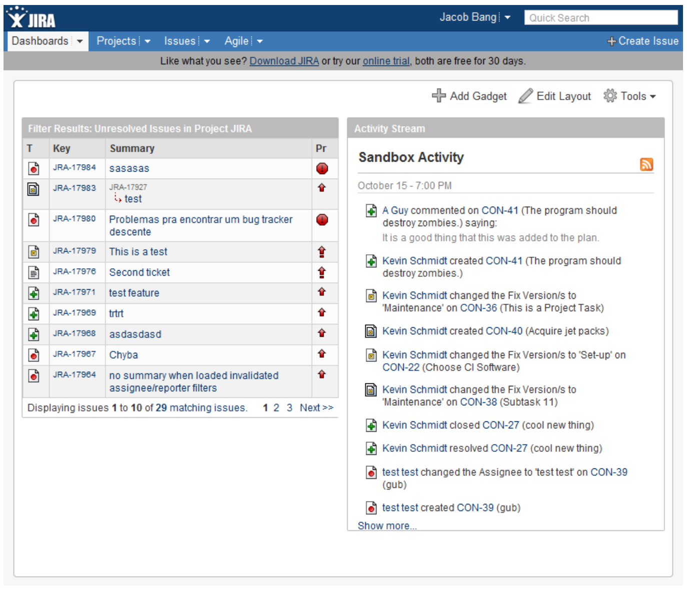
2.1.2.1 Exemplars

There are already many project management tools for software development on the market today. This makes it an advantage to look at some of these systems and consider these as exemplars for the system to be developed. These examples could be used to look at what functionality is generally in project management tools, but also to find great features which must be considered in the system to be built. In a survey of the market, two systems have been selected to serve as examples and will be described in the next sections. Each description will include a brief analysis of the systems and describe the pros and cons of these.

A general thing that can be said about the systems is that they are both web-based. As will be mentioned later in the FACTOR criteria, the system to built will be a web-based solution. The advantage of this choice is, that the system can be easily accessed without the need for installing additional software on the clients.

Jira

Jira is an administration tool developed by Atlassian and is commonly used to manage bug tracking, issue tracking and agile software development [Ltd09]. It is based on a web interface and use lots of AJAX[1] to make it more pleasant to work with. For example, it is possible that different tables on the page are automatically updated in regular intervals without rest of the page to be updated. In Jira this functionality is called dashboard and is shown in figure 2.2.



Like what you see? Download JIRA or try our online trial, both are free for 30 days.

Filter Results: Unresolved Issues in Project JIRA

T	Key	Summary	Pr
●	JRA-17984	sasasas	●
▣	JRA-17983	JRA-17927 ↳ test	⇧
●	JRA-17980	Problemas pra encontrar um bug tracker descente	●
▣	JRA-17979	This is a test	⇧
▤	JRA-17976	Second ticket	⇧
✚	JRA-17971	test feature	⇧
✚	JRA-17969	trtrt	⇧
✚	JRA-17968	asdasdasd	⇧
●	JRA-17967	Chyba	⇧
●	JRA-17964	no summary when loaded invalidated assignee/reporter filters	⇧

Displaying issues **1** to **10** of **29** matching issues. 1 2 3 Next >>

Activity Stream

Sandbox Activity

October 15 - 7:00 PM

- A Guy commented on CON-41 (The program should destroy zombies.) saying:
  It is a good thing that this was added to the plan.
- Kevin Schmidt created CON-41 (The program should destroy zombies.)
- Kevin Schmidt changed the Fix Version/s to 'Maintenance' on CON-36 (This is a Project Task)
- Kevin Schmidt created CON-40 (Acquire jet packs)
- Kevin Schmidt changed the Fix Version/s to 'Set-up' on CON-22 (Choose CI Software)
- Kevin Schmidt changed the Fix Version/s to 'Maintenance' on CON-38 (Subtask 11)
- Kevin Schmidt closed CON-27 (cool new thing)
- Kevin Schmidt resolved CON-27 (cool new thing)
- test test changed the Assignee to 'test test' on CON-39 (gub)
- test test created CON-39 (gub)

Show more...

Figure 2.2. The dashboard functionality is very handy when a user wants to get a quick overview of the development progress. It is possible to add and remove Gadgets.

Beside the default settings, it is possible for all users to create their own preferences for viewing the dashboard. The elements on the dashboard are called Gadgets and can be placed anywhere on the dashboard. There are many different Gadgets and almost all features in Jira can be deployed as a Gadget. An example of a gadget can be seen on figure 2.2 where the right part of the dashboard is a log of recent activities on the project. This applies, as an example, the creation of new issues and status changes of issues.

Another great feature is the project pages. On these pages the user can find all information associated with each project. Each project has a summary page, where the latest updates and status-changes are shown. In addition, it is possible to choose what information you want to be seen in detail. An example of this feature is the option to show how many issues remaining to be fixed until the next release for publication.

Jira use 'issues' to handle tasks. Issues can be categorized as types (bug, new feature, task improvement etc). Besides the type there is also a number of other details such as priority.

Jira is a system with lots of complex functionality and contains nearly everything needed

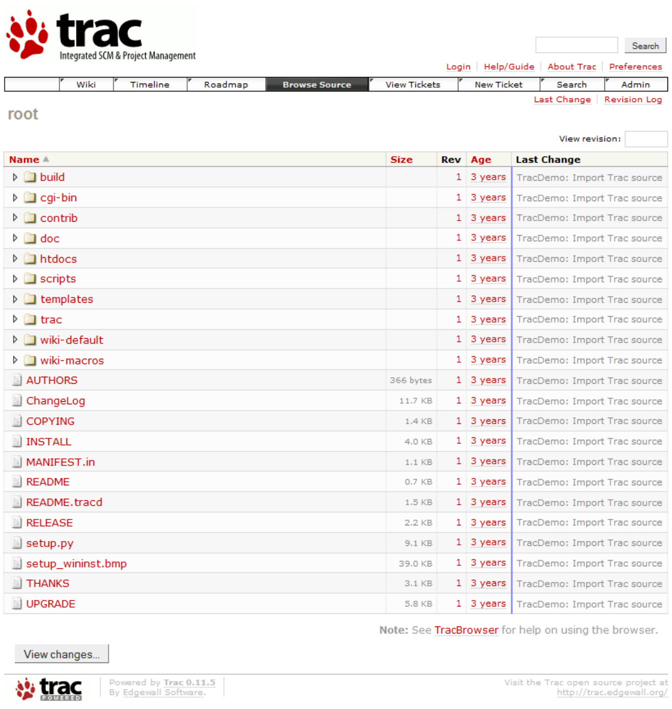
[1]Asynchronous JavaScript and XML - is based on JavaScript and HTTP requests. Used to make a webpage more dynamic by allowing JavaScript to communicate with the Web server without having to retrieve a new page [W3S09].

to keep track of software development. The interface does, however, that the system is easy to use for less experienced users. Users wishing to use the more advanced features will be able to access them without major problems and may otherwise change the dashboard to show the relevant information. The idea of showing recent information at one place, and the way of handling issues, are some of the functionalities that would be a good idea to use as examples when developing the product.

Trac

The idea of Trac is to link software development, wiki pages, issue control and version control together. This is done by integrating all components into a single web system where all these can communicate together. The web system is like Jira but is made more like a traditional page without a lot of interactive AJAX [Sof09].

A key feature of Trac is the ability to integrate source code sharing with the issue control. To communicate with the source code, the version controlled source has been integrated into the system. An example of this can be seen in figure 2.3. Subversion (SVN) is officially supported but it is also possible to use other tools like GIT[2]. By using the feature of integrated version controlled source, makes it possible to close an issue, with a description of what has been done, while linking the description to exactly the SVN version, that fixes the issue. It is then possible to follow the link and see exactly what changes the update has made to the system. It is also possible to browse the whole source and look in every file and see all the changes there have been done in every version.


trac
Integrated SCM & Project Management
Search
Login | Help/Guide | About Trac | Preferences
Wiki
Timeline
Roadmap
Browse Source
View Tickets
New Ticket
Search
Admin
Last Change | Revision Log

root

View revision:

Name ▲	Size	Rev	Age	Last Change
build		1	3 years	TracDemo: Import Trac source
cgi-bin		1	3 years	TracDemo: Import Trac source
contrib		1	3 years	TracDemo: Import Trac source
doc		1	3 years	TracDemo: Import Trac source
htdocs		1	3 years	TracDemo: Import Trac source
scripts		1	3 years	TracDemo: Import Trac source
templates		1	3 years	TracDemo: Import Trac source
trac		1	3 years	TracDemo: Import Trac source
wiki-default		1	3 years	TracDemo: Import Trac source
wiki-macros		1	3 years	TracDemo: Import Trac source
AUTHORS	366 bytes	1	3 years	TracDemo: Import Trac source
ChangeLog	11.7 KB	1	3 years	TracDemo: Import Trac source
COPYING	1.4 KB	1	3 years	TracDemo: Import Trac source
INSTALL	4.0 KB	1	3 years	TracDemo: Import Trac source
MANIFEST.in	1.1 KB	1	3 years	TracDemo: Import Trac source
README	0.7 KB	1	3 years	TracDemo: Import Trac source
README.tracd	1.5 KB	1	3 years	TracDemo: Import Trac source
RELEASE	2.2 KB	1	3 years	TracDemo: Import Trac source
setup.py	9.1 KB	1	3 years	TracDemo: Import Trac source
setup_wininst.bmp	39.0 KB	1	3 years	TracDemo: Import Trac source
THANKS	3.1 KB	1	3 years	TracDemo: Import Trac source
UPGRADE	5.8 KB	1	3 years	TracDemo: Import Trac source

Note: See TracBrowser for help on using the browser.

View changes...

Powered by Trac 0.11.5
By Edgewall Software.

Visit the Trac open source project at
http://trac.edgewall.org/

Figure 2.3. Trac allows the user to browse source code directly from version control.

[2]Git is a free & open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency [HPT09].

Trac also has a Timeline function which is used to get an overview of the latest changes of the project. This applies to changes in issue handling, updates in version control and changes in the wiki. This function is very similar to Jira's Dashboard described in 2.1.2.1 on page 13. Another function which delivers a good overview too, is the Roadmap in Trac. This function shows how far through each release (or iteration) the development has reached. The progress is shown by comparing how many issues have been flagged for a release and how many of these are finished.

In general Trac is a very good all-round tool for project management. The good thing about Trac is the flexibility because it is not specially designed for a specific model of development, but can be used for almost all projects.

2.1.2.2 Metaphors

Considering a tailor as a metaphor for the software project management system. At a tailor, the customer defines what he/she wants the tailor to produce. The tailor might have some ideas for the product, and after discussing the customer's needs and preferences, the tailor is going to work. At any time the customer is able to visit the tailor and check how for instance the new wedding dress is developing. Further, the tailor would have to schedule meetings with the customer in order to verify, that the tailor is producing a product the user wants and which agrees upon the initial talk between the customer and the tailor when they first met. The process of designing and assembling the dress might therefore be an iterative process, where several steps could be repeated in order to satisfy given conditions.

When comparing this metaphor to the rich picture, a new idea has occurred. The supervisor should be granted access to the system, and thereby have the opportunity to follow how the students progress in the development of the system. For instance by being able to view burn down charts and statistics concerning the number of open issues, implemented user stories and so on.

2.1.3 System Definition

To define a system definition the FACTOR criterion can be used to ensure that important aspects are covered. **F** is for functionality and describes the systems functions that support the application domain tasks. Thatx is, defining what the system is able to do. **A** is for application domain and concerns those parts of an organization which administrate, monitor, or control a problem domain. Next is **C**, which covers conditions under which the system will be developed and used. Technology is abbreviated by **T**, and covers the technology used to develop the system and the technology on which the system will run. **O** is for objects and describes the main objects in the problem domain. Final is **R** for responsibility which covers the systems overall responsibility in relation to its context. That is, how the system would interact with the tasks to be solved using the system.

Using the rich picture and the process of generating ideas leads to the following FACTOR criterion for the system to be built:

Functionality In general; plan and document the development process. Keep track of user stories and tasks. Define schedules for implementation of user stories. Describe status and priority for user stories and tasks. Offer the opportunity of internal communication among the development group and their supervisor. Let users and supervisor be able to comment on user stories and tasks. Provide statistics and project status by use of burn down chart. Grant access to supervisor with the purpose of monitoring the development progress. Administrate users and provide contacts details.

Application domain Multiple study groups and their related supervisor involved in a smaller software development project.

Conditions The system is applied to a number of developers with familiarity and knowledge about using agile development, and with experience of using computers and basic software systems. Furthermore the system has a fixed deadline. That is, any adjustment will have to be in the content of the system as the deadline cannot be postponed.

Technology The system should run on an ASP.NET MVC web server and be accessible through the Internet using a web client. For data storing, the system should use a relational database such as MySQL.

Objects Student developers, supervisor, small software project, iterations, user stories, tasks and comments.

Responsibility The system should follow the agile spirit and preserve the work, creativity and ideas of each individual in the software development group.

Using the FACTOR as support leads to the following system definition:

A web-based system used to aid multiple study groups administrating and planning a smaller software development project. Any affiliated supervisor must be able to monitor the progress of the software development. The system should inspire to plan, document and develop using agile development, with special emphasis on structuring the work of each users contributions to the project. Statistics concerning implementation of user stories should be implemented using burn down charts and progress bars. It should be clear that programming takes place in another environment, such as Visual Studio. As the system is seen as a project management system for multiple study groups, it should be possible to manage who is able to join a specific project. In addition, each user should have their own profile, containing contact information. The system must be programmed using ASP.NET MVC and be accessible using a web-client. For data storing, the system must use a relational database management system (RDBMS) such as MySQL for instance.

2.2 Problem Domain

The problem domain is concerning that part of the system, which is administrated, monitored and controlled by a system. It outlines the future users understanding of the problem domain, and the aim of describing it, should be creation of a model, describing the problem domain. Using this model it would later on be possible to design and implement a system that can process, communicate, and present information about the problem domain in a appropriate and usable manner. The key in this section is to describe what information the system should deal with. This will be done through defining classes and events in the system, and by structuring the relations between classes. Furthermore some state chart diagrams should support the description. Usually this process of defining the problem domain is approached bottom-up, defining all the parts such as classes, events, structures and behavior. This description will consider the final choices and not completely reflect the bottom-up approach. The section is based on [MMMNS01, chapter 3, 4, and 5].

2.2.1 Classes and Events

In OOA&D a class is a description of a collection of objects sharing the same structure, behavioral pattern, and attributes. For instance, instead of working with several developer objects in a system, those are grouped together in a developer class, and then referred to as instances of that particular class. Thus the objects will still have their own identity, state, and behavior.

Events are related to classes. More specific; an event is defined as an instantaneous incident involving one or more objects. The event is used to describe and characterize problem domain objects. In reality all events have a duration in time, but in defining and using events, it only matters if the event has been triggered or not. An event might involve several objects which leads to common events. The common event is used to express a dynamic relation between objects.

Both classes and events are defined using an iterative process. Here, candidates for classes are first considered and then evaluated. Then, events corresponding to the chosen classes are found using the same process: write down, consider and evaluate. The final result is a list of classes and events describing which events and objects should be registered and controlled by the system.

In the process of defining classes and events, considerations about each class and event are written down and used as input when evaluating what to keep and what to purge. These considerations are useful, not only in the definition, but also to use as a note for how exactly different classes and events were meant to be described and handled in the system.

In the following, some of the more important considerations related to the definition of classes and events are listed.

Class: Developer
This class contains basic contact information about a developer. Each developer is assigned to work on specific tasks.

Event: Task assigned
This event occurs in the current iteration, when developers have picked out some tasks they want to solve.

Class: Supervisor
This class contains basic information about a supervisor.

Event: No events
The class supervisor has no events. It is a generalization of a user, and thereby inherits all events on the user class. The supervisor has fewer remedies than a developer, and consequently the two classes have to be split. Another way to solve this could be to make a user class encompass a supervisor and make a generalization to a developer class which could be broadened. However, even though the supervisor class does not have any events it is kept to make the model remain conceptual and realistic.

Class: User story
This class contains a description of the user story, and a combined implementation-status inferred from tasks related. An instance of this class must be given a priority to indicate the importance of implementing a particular user story.

Event: Iteration scheduled
When scheduling an iteration, its start- and end date should be defined, as well as the user stories to be implemented in the particular iteration. To ensure agile development this event must be included.

Class: Task
Each task has its own description. It is written in a technical manner for the developers to understand what exactly should be done to complete this certain task. Further, each task has its own status, priority and information about which developers are assigned to it. At last, a task can be related to a set of comments. The comments could be notes about how a particular task has been handled, and thereby serve as a kind of documentation when specific functions have to be described in the related project report.

Event: Re-assign task to developer
In some situations, for instance when using pair programming, it might be necessary to re-assign a task to another developer. This is allowing a team to switch between different tasks and take full advantage of pair programming.

Class: Iteration
Each iteration-object has a number of user stories related. It also contains a schedule defining the deadline for the specific iteration.

Event: Close iteration
This event occurs automatically when an iteration has reached its deadline. Without this event it would not be possible to end an iteration.

To get a full overview of the relations between classes and events in the system, that is which classes an events are involved, an event table has been constructed in table 2.1 on the next page. Furthermore, the purpose of the event table, combined with its related class diagram in figure 2.4 on page 20, is to depict a conceptual and realistic model of the system. Therefore several events has been outlined to ensure, that the most important and common events of the system are covered.

	Project	User	Supervisor	Student group	Developer	Part	Comment	Issue	Task	User story	Iteration
Added						*					
File attached		*					*				
Part commented		*				*	*				
Issue commented		*					*	*			
Iteration deleted											+
Iteration finished											+
Iteration updated											*
Project created	+			*							
Project deleted	+										
Project updated	*										
Reported		*						+			
Task assigned					*				*		
Task deleted									+		
Task solved									*		
Task re-opened									*		
Task updated									*		
Issue assigned					*			*			
Issue solved								*			
Issue re-opened								*			
User story deleted										+	
User story solved										*	
User story updated										*	
User assigned to project	*	*									
User removed from project	*	*									

Table 2.1. Event table describing how classes are related to events in the system. The + symbol indicates that an event will either not happen or happen once. The * symbol means that an event can either not happen or will happen several times.

In general, common events shown in table 2.1 implies that when a certain common event is triggered, the event will involve an instance of each of the related classes. Such events will be handled as associations in section 2.2.2, where the structure of the problem domain will be described.

2.2.2 Structure

Next step in analyzing the problem domain is to describe the structural relations between classes and objects. By doing so, it becomes easier to get an idea of how information should be related in the final system. The process of defining relations has been done on the basis of the classes defined in section 2.2.1 on page 17. The result is the class diagram, depicted at figure 2.4.

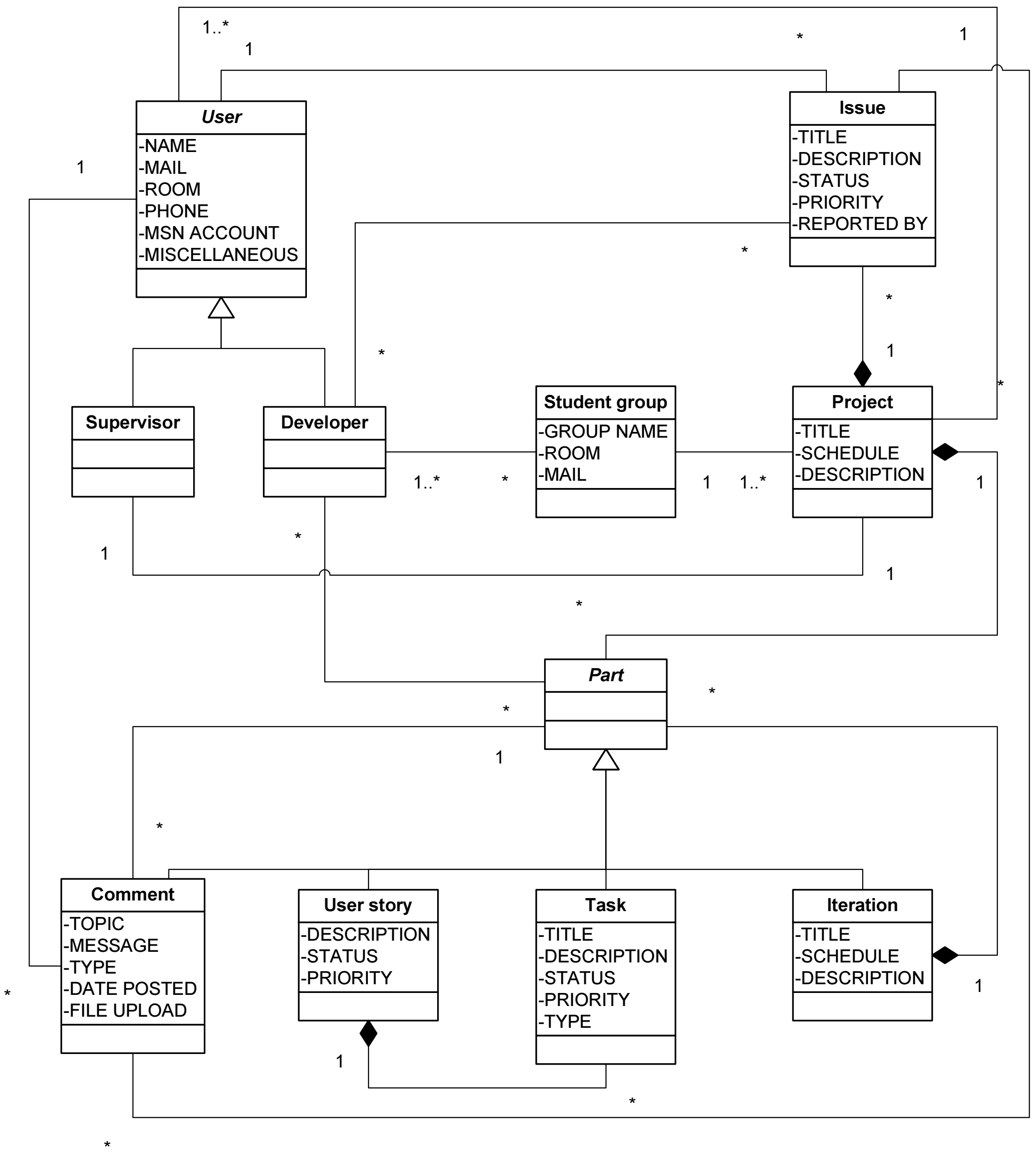


Figure 2.4. Class diagram depicting the object-oriented structure of classes and objects in the problem domain.

First of all, in general the diagram is depicting a structure, which helps dealing with some of the issues defined in the rich picture from the system definition. For instance, by letting user stories, tasks and other information be stored in the system, the problems caused to use of a dashboard in a regular group environment, has been taken into account.

The class diagram is using some different kinds of object-oriented patterns. First of all, the project class is aggregated into several parts. These parts are assembled in a hierarchy, which is defined using the composite pattern. This pattern is allowing the hierarchy to be composed in several ways, corresponding to how the system will need to structure its information. For instance, concerning a student group using a system based on the class diagram shown in figure 2.4. The student group will be using the system together with an agile development method such as XP. At first, this group needs to define their software development project. This project will then be the root in a three structure, consisting of an iteration assembled by an arbitrary amount of user stories, tasks, and comments to the progress. When the iteration has come to an end, a new iteration can be instantiated as a new branch in the three, and then be assembled by related user stories, tasks and so on. This kind of recursive structure can be illustrated by figure 2.5. Here, the project is concerned as the root of a semester tree, involving an unknown amount of iterations composited by an arbitrary number of leafs, which is user stories, tasks and comments.

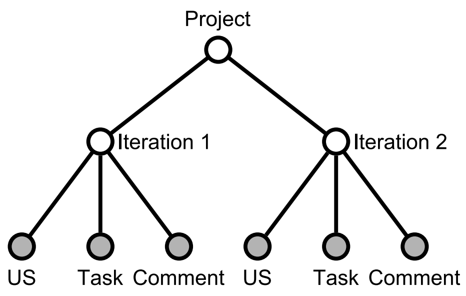


Figure 2.5. The composite pattern used when composing a hierarchy of project elements in the system to be built.

The key of using this pattern is, that the handling of information in the system gets dynamic, and at this level in modeling the system, there has not been set any restrictions on how this dynamic should be used. Some would claim that it is not agile development if one iteration leads to a new iteration, by which two iterations will be related. The composite pattern allows this structure, but it have been chosen to restrict this behavior, such that a project only consists of iterations which do not have iterations as their leafs. That is, no iterations will never be able to have an iteration as its child. This is similar to what is depicted in figure 2.5.

Another thing to notice in the composite pattern is the relation between the user story class and the task class. Without their association, it would not be possible to track which user story each individual task is related to. Same story appears concerning the comment class. Here a comment is associated to the abstract part class, implying that a comment can be related to either an iteration, task, user story or to another comment. By allowing a comment to relate to another comment, lets the system handle information of dialogues between the comment authors. Furthermore, a comment is able to be related to an issue, allowing the developers to discuss a given issue. In this context, it should be noticed that a supervisor is able to report an issue, depicted by the association between the abstract user class and the issue class.

At last it should be mentioned that a student group is aggregated by a set of student developers, each of them related to parts of the project. Further, the group as a unit is associated to their supervisor through the project. The supervisor is able to comment on the work of the students, symbolized by his association to the comment class.

Summing up this diagram leads to a list of quality attributes, which the system will be able to handle using the described way of handling information:

- Information such as user stories, tasks and issues is stored and structured by the system (will not fall down on the floor and get lost).
- The system will be able to handle information fitting to different sizes of student development projects (does not require large dashboards and several post-it notes).
- New kind of supervisor-interaction. The supervisor is able to supervise and comment on different project components without being invited to a meeting with the group (the process is not 'hidden' in periods between meetings).
- Comments and thereby also documentation makes it easier for non-experienced developers to get track of what has been made on a task or user story for instance (it becomes easier to understand the project).

2.2.3 The Dynamics

To model the dynamics of the problem domain, a state chart diagram can be a useful tool. The idea is to describe the behavioral patterns and attributes of the classes defined in the class activity in 2.2.1 on page 17. By doing this, each class description is extended, and it becomes more clear, how each individual class should behave in the problem domain.

One of the more important behavioral pattern of a class in the problem domain, is the behavior of the 'issue' class. This class has several states and can be related and used in different ways. This has been depicted in figure 2.6.

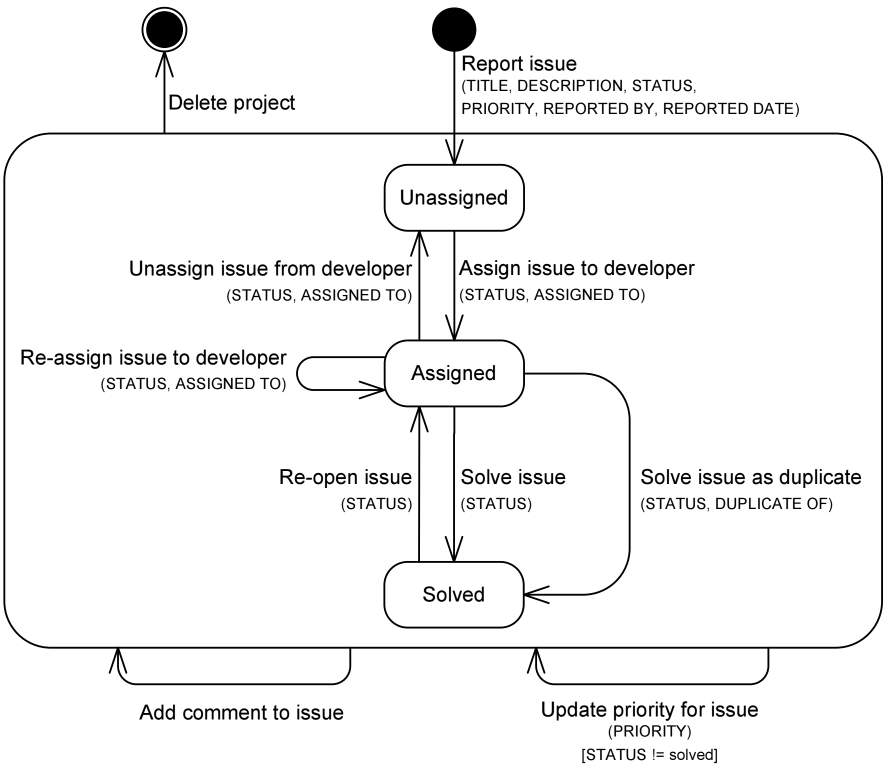


Figure 2.6. State chart diagram for the 'Issue' class.

As shown in figure 2.6 all instances of class 'issue' can behave in their own specific way. The states and events are related in a pattern, ensuring that the developer gets all relevant information needed about an issue. For instance, a developer is able to see the status and priority of a single issue. Further, each developer and their supervisor, is able to comment on a single issue. This helps the student group keeping track of different aspects related to each issue, and further, it is allowing the supervisor to give feedback on his students work. An issue can change priority and be re- and unassigned to different developers if needed. Finally, it should be noticed that each instance of the 'issue' class has its owns title and description. Combining this with status and priority-attributes makes it easier to sort, separate and browse through a large amount of issues.

2.3 Application Domain

When analyzing the application domain, there is focus on the question: *how will the system be used?* Answering this question leads to requirements for the system usage, functions and its interfaces which in addition, is the three main keys of the application domain. The application domain analysis interacts with the analysis of the problem domain described in section 2.2 on page 16. There, the question was: *what is the target-system about?* Answering that question lead to definitions of requirements for the systems model. Thus, the problem domain analysis can be used as the vocabulary for this section. Even though that interfaces is one of the three key concepts in defining the application domain, this area will not be covered, as this project is not supposed to concern how to design user interfaces. [MMMNS01, chapter 6 and 7] will be used as primary source for this section.

2.3.1 Usage

When looking at usage of a system being developed, it is important to determine who the users of the system will be. Users can both be individuals, or systems that the target-system should cooperate with. In OOA&D users are referred to as actors. An actor is an abstraction of users or other systems that interact with the target-system. Thus, an actor describes someone or something that interacts with the system, such as an employee or a third-party program for instance. When defining the usage of a system, questions such as *who will use the system?* and *how will it be used?* can be used in terms of defining actors and use cases.

In this section, actors have been identified by studying the goal and characteristics of different persons related to the system. The goal describes the actor's role in relation to the target system whereas the characteristics concerns the significant aspects of the actors use of the system. In the following the actors of the system to be built, will be defined.

Login actor

Goal: The login actor can access the system using a valid account. When logged in, the login actor can log out of the system.

Characteristics: The login actor is anyone with access to the system.

Examples: A user logs in to the system, and then later, he logs out of the system.

Reviewer

Goal: A reviewer is a user with interest in following the progress of the project. He is also a user, who is interested in commenting on the work of the users.

Characteristics: The reviewer is anyone who got a valid login account and at the same time, is related to a given project.

Examples: The supervisor related to a project observes the progress of his student groups work. Further he observes an issue in the program, and reports it.

Developer

Goal: The developer is responsible for creating and updating project related items. Further, a developer is responsible for administrating other users' access to the developers' current project. Finally, the developer has rights to remove any other developer from the project he is a member of.

Characteristics: All group members can act as developers.

Examples: A developer grants access to a new pending user.

Each actor is interacting with the system using different types of patterns. In OOA&D these patterns are described in use cases. More formally, an use case is an abstraction of an interaction with the system. It describes a delimited use of the system. Thus, a set of all use cases concerning one system, will describe all possible interactions with the system. Defining use cases is an activity involving both users and developers. The coming users of the system formulates needs and contribute with application domain insights, whereas the developers formulate use cases and contribute with technical knowledge. The contribution from the upcoming users can be seen as the part of XP, where user stories are being defined. Thus, user stories only describes a certain functionally which the users want the system to concern. In contradiction, use cases are defining a certain pattern of interaction with a system, not just the functionality needed itself. This means that use cases can be far more technical and contain a lot more information than user stories.

As mentioned in the approaches to this project on page 2, the target-system will be developed using parts of the agile development method XP. Therefore, the contributions of the coming users of the target system, which in this project will be the authors of this report, have written down user stories, serving as the users contribution to the process of defining use cases for the system. In the following, these contributions will be grouped and linked to their related use case.

Below is the list of user stories concerning the system to be built. The user stories are grouped by their related use case, which has been marked with bold letters. As an exception, the first group of user stories is not related to any use case. These user stories are written by the developers, with the purpose of having a set of user stories, defining the basic stuff to do, when a new development project is starting up.

Initiating/design tasks

- Start up an ASP.NET MVC project.
- Check that the system can compile and run at the host.
- Set up MySQL RDBMS.
- Hard code tables in DB using the Data Definition Language (DDL).
- Design Cascading Style Sheet (CSS).

Use case: Overview

- There should be an overview of which user stories are assigned to the current iteration.
- It should be possible to get an overview of the status of all user stories.
- Dashboard, that is the index site of the system, should provide a list of recent activities and the latest project status.
- Any user should have access to statistics, that is burn down charts and process overview of the current iteration.
- All users must have their own profile, enabling other users to see contact information.
- It should be possible to get an overview of who is working on which tasks.

Use case: Iteration planning

- User stories should be able to be created and exist in the system.
- Development must be based on iterations.
- It should be possible to specify which user stories should be implemented in the current iteration.
- A task can be created and edited.
- A task should be able to be attached to any user story.
- A developer must be able to be assigned, re- or unassigned to a task.
- It should be possible to update the status and priority of the implementation of a certain user story and task.

Use case: System login

- The user should be able to login to the system.

Use case: Comment

- Supervisor and developers must be able to comment and attach additional information to comments, iterations, user stories, tasks, and issues.

Use case: Documentation

- All actions and changed statuses should be logged to generate a history/documentation of the development process - similar to a 'change log'.

Use case: Error report

- It should be possible to report an issue.

Use case: Error-handling

- Any issue must be editable.
- It should be possible to update the status and priority of the solution of a certain issue.
- An issue must be able to be re-opened.
- It should be possible to assign, re- and unassign developers to an issue.

Use case: Apply for a project

- It should be possible to apply for any project.

Use case: Start project

- It should be possible to create and schedule a project.

Use case: User administration

- Developers should be able to remove users.
- Developers should be able to grant or reject users' access to a project.
- Different user-levels: supervisor and developer.

To give a deeper insight in the use case contribution of the developers and to elaborate the essence of some of the more important use cases, this will be done in the following using state charts and use case descriptions.

The first use case being described is concerning the pattern of applying for a project. This pattern is modeled in figure 2.7.

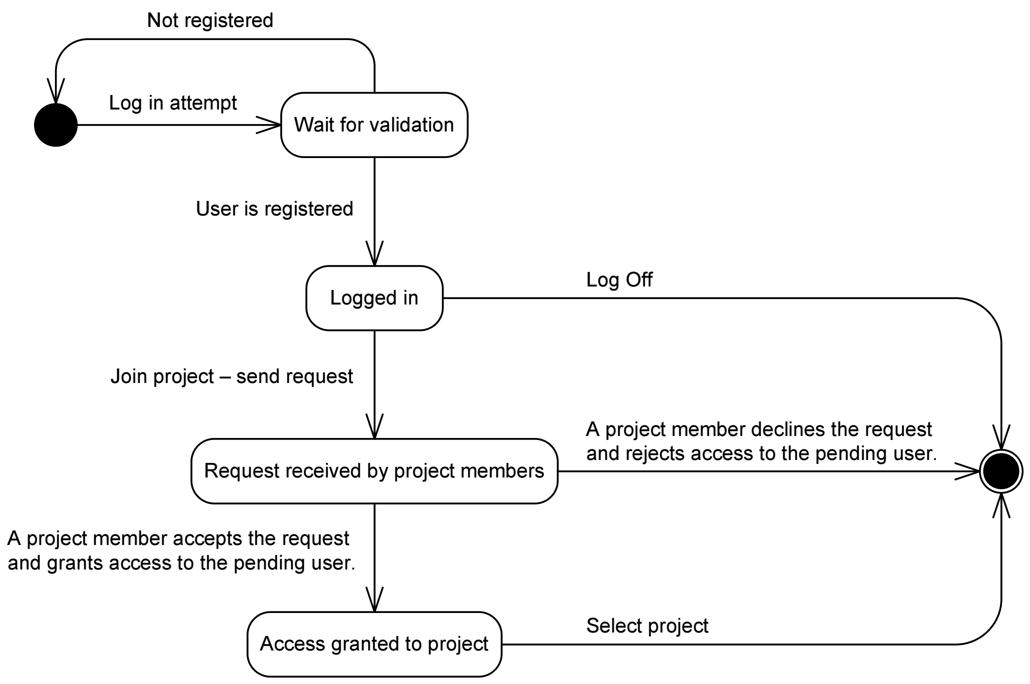

Not registered
Log in attempt
Wait for validation
User is registered
Logged in
Log Off
Join project – send request
Request received by project members
A project member declines the request and rejects access to the pending user.
A project member accepts the request and grants access to the pending user.
Access granted to project
Select project

Figure 2.7. State chart diagram for use case 'Apply for a project'.

As shown in figure 2.7 on the facing page, when applying for a project, the login actor starts by trying to log in to the system. If the user has no valid account in the system, then the user will be rejected. Else, if the user is registered in the system, he will be logged in. When logged in to the system, the user can choose to log off or send a pending request for any project managed by the system. By sending a request, the developers in the project, to which the user is now pending, will get an entry in their administration section of the system. This entry will tell them that a user is pending to this project, and that he can be granted or rejected access to the current project. If the user is rejected, he will no longer have status as pending to the current project, and will have to try pending again if he wants to. Else, if the user has been granted access to the project, he is now a part of it and will no longer be registered as pending to this particular project. Further, the user will now be able to select the project from the project list, and work inside it any time.

The next use case is concerning iterations. The part of the use case, where a user story is created, is modeled in the statechart diagram at figure 2.8.

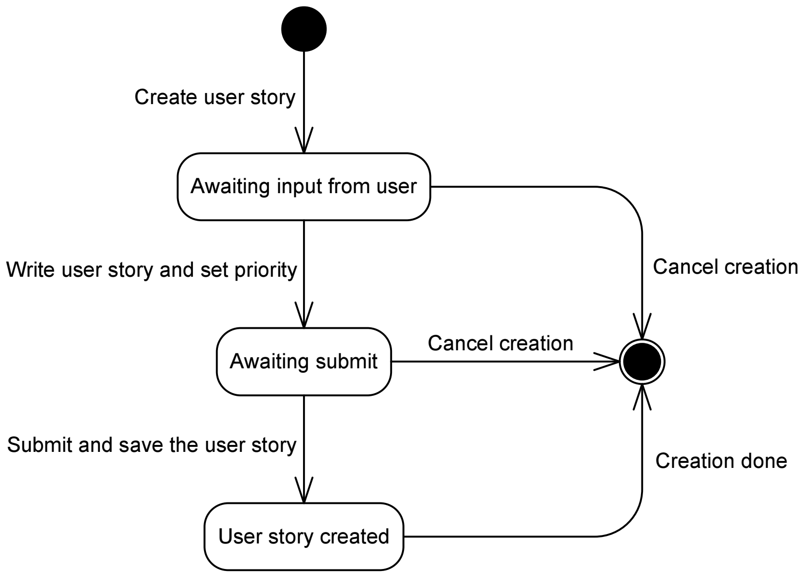


Figure 2.8. State chart diagram for use case 'Iteration planning', concerning the part where a user story is being created.

When a developer actor wants to save a 'customers' user story for instance, in the system, the first step is to tell the system that he wants to - this is shown in figure 2.8. On the screen the actor must choose 'Create User Story', and the system will be awaiting the user story description. The developer can either choose to cancel and end the interaction, or the actor can continue and type in the user story. The system will be waiting for the developer to either submit or cancel the creation. If canceled, the interaction will end. Else, if the creation is submitted, the user story will be saved in the system. Then, the user story will only disappear if it is being deleted.

Finally, in figure 2.9, the use case concerning 'documentation' has been depicted. In this use case, the main idea is that all relevant actions and status changes in the system, must be logged. Doing this, should make it easier to document the development process.

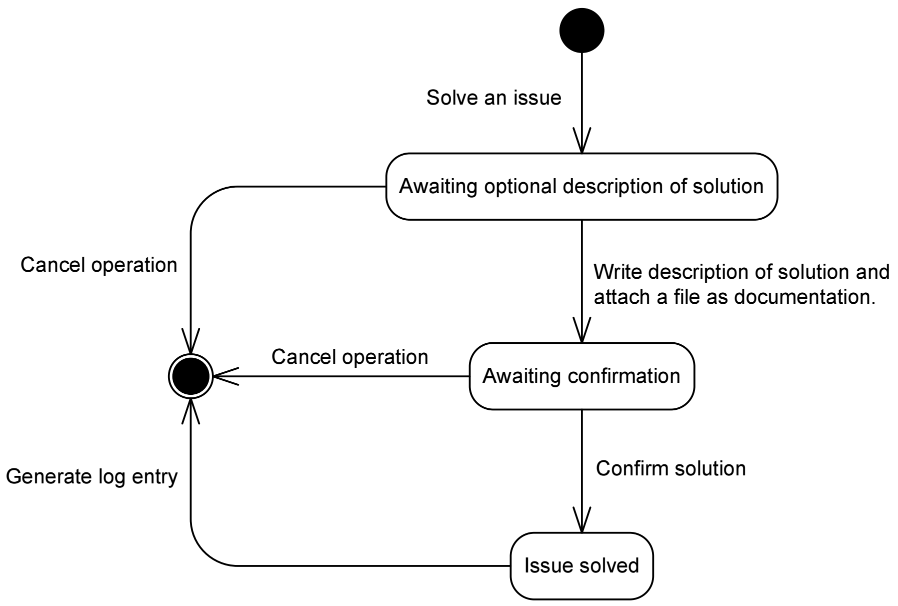


Figure 2.9. State chart diagram concerning the use case 'Documentation'.

As illustrated in figure 2.9, the developer actor has solved an issue and wants to reflect this in the system. By selecting the issue in the system and by choosing the solve action, the system is now awaiting an optional description of the solution. This description will be a part of the log entry generated, if the user confirms his solution. Now, the solution has been described and a screenshot has been attached too, and the system is now awaiting the users confirmation to mark the issue as solved. At this point, the user is allowed to cancel the operation, leaving his description, or he is able to confirm his description and the new state of the issue. The user confirms that the issue has been solved, and automatically, a log entry will be generated, telling all project members, who has solved the issue, and how it has been solved. This log entry will further be related to the current issue, enabling other developers to re-open the issue or to add comments to the solution.

Excerpts from a number of different use cases have now been described, but to get an overview of how these use cases are related to actors, a use case diagram is needed. The use case diagram in figure 2.10 on the next page gives an overview of the connections between actors and use cases. When a use case has two pointers, that is two actors related, both actors must be involved in the use case. For instance, the use case 'Apply for a project', involves both the login actor and the developer actor. The reason is, as described in the statechart diagram in Figure 2.7 on page 26, that when a login candidate has send his request for being granted access to a project, then a developer, which is member of that particular project, must either grant or reject the pending user. In that way, both type of actors will be involved.

A common thing when describing actors, is that any particular user may appear in different roles. This means, that when a particular user is accessing the system, the user will appear as a login actor. If the user then starts to error-handle, the user will appear as a developer, and so on. As the user story in use case 'system login' says that a user should be able to login to the system, it implies that any user must appear as login actor before they are able to use the system any further.

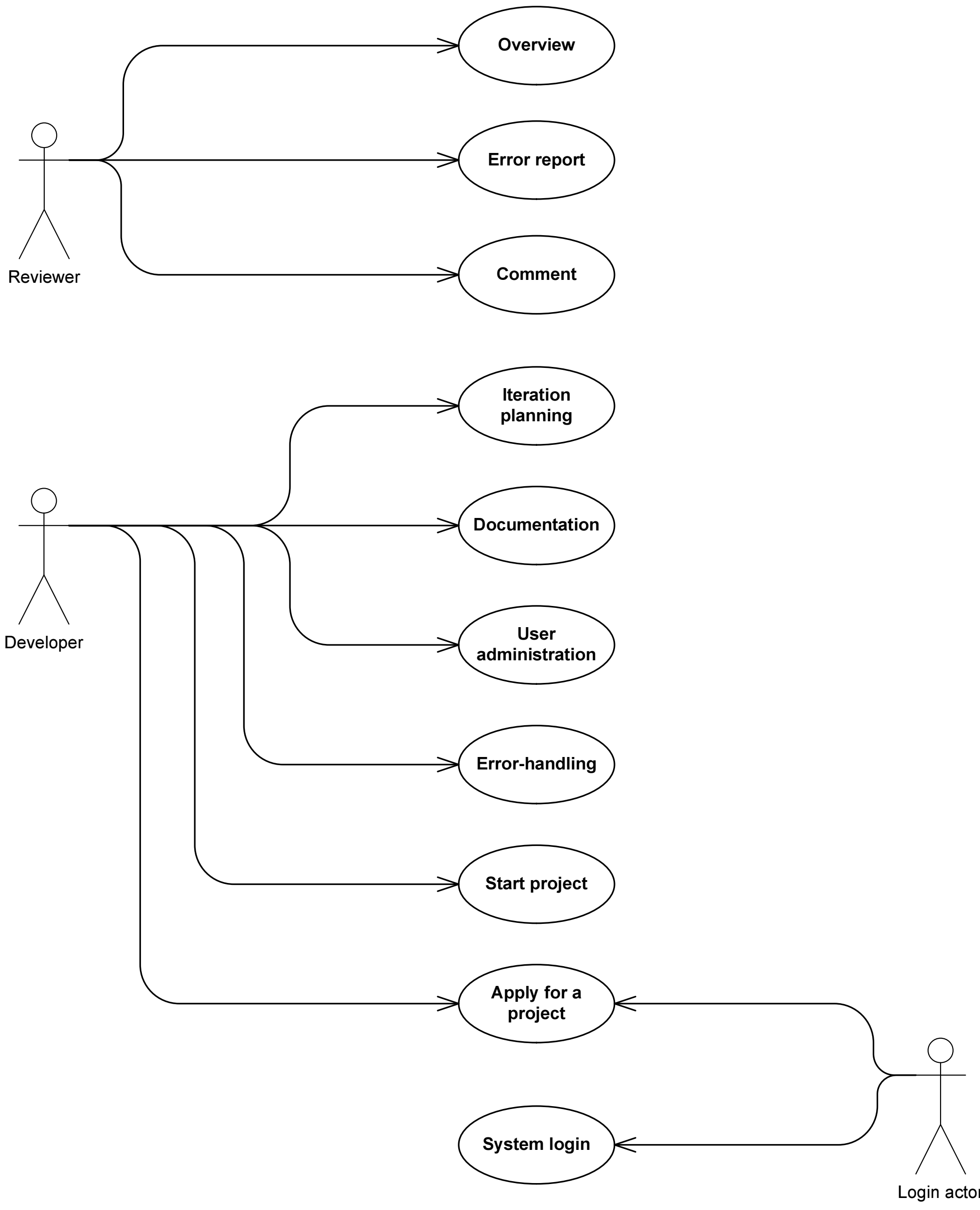


Figure 2.10. The use case diagram provides an overview of how actors are related to use cases in the system.

2.3.2 Functions

From an analytic perspective, functions are very useful as they are intended to elaborate the objective of the system. When defining functions it is asked, *what is the system supposed to do?* In the usage part at subsection 2.3.1 on page 23, it was concerned *how* the system should be used. This makes the usage and functions closely connected, since it is difficult to talk about how a system is being used without discussing what it should do.

When analyzing functions the object is to get a complete list of all the functions the system must implement. The goal is not to describe every single function in detail, quite the contrary the goal is to identify the functions.

Functions can be grouped into types. There are four different types of functions:

Update is a type defining functions which are activated by a problem or application domain event, and which results in a change in the model's state.

The type *Signal* defines a function which is triggered by a change in the model's state. Running the function always results in a reaction to its surroundings: that is either the reaction is a display to the actors in the problem domain or else the reaction is a direct intervention in the problem domain.

When an actor has need for information the function of type *Read* is activated. The function is displaying the relevant data of the model to the actor.

Finally, the *Compute* function is activated when an actor provides information, which should be included in a computation which also involves data from the model. The function returns its computed result to a display.

To identify functions it is essential to review the classes and events from the problem domain, and the use cases from the application domain. Typically, classes causes the discovery of read and update functions, as it is here, the model is structured. In addition the related events often involves update functions, as it is most likely that an event will affect the model by updating some attributes. When it comes to use cases, all types of functions can be relevant.

When all functions are found they must be defined by their type and complexity. OOA&D recommends that functions in the analysis are specified as short as possible, as the intention is to identify the functions. If a function is very complex, that is involving mathematical expressions for instance, it might be relevant to post some sort of math or pseudo code explaining it.

For the PMS the list of functions, seen in table 2.2 on the facing page has been identified.

Title	Complexity	Type
A user should be able to see profile details	simple	read
It should be possible to see profile details for all users in the chosen project	simple	read
It should be possible to see a list of all user stories	simple	read
It should be possible to see a list of all iterations with start/end-time	simple	read
It should be possible to see a list of all open issues in the system	simple	read
It should be possible to see details about the project	simple	read
It should be possible to see the latest recent activities in the project	medium	read
It should be possible to create a project	simple	update
It should be possible to edit the project description	simple	update
A user should be able to join to a project	simple	update
A user should be able to select a project	simple	update
A user should be able to edit profile details	simple	update
A user should be able to log on to the PMS	simple	update
A user should be able to log off the PMS	simple	update
It should be possible to create a user story	simple	update
It should be possible to edit a user story	simple	update
It should be possible to delete a user story	simple	update
It should be possible to create a new task	simple	update
It should be possible to create a new iteration	simple	update
It should be possible to report, solve and re-open an issue	simple	update
A new user should be able to register to the PMS	medium	update
It should be possible to attach a developer to a task/issue	medium	update
It should be possible to post a comment on any iteration, user story, task and issue.	medium	update
It should be possible to attach a file to a comment.	medium	update
It should be possible to attach a user story to an undone iteration	medium	update
It should be possible to get an overview of a current iteration with time estimates, statistics and related user stories	medium	compute/read
It should be possible to see a burn down chart of the current iteration as well as the entire project	complex	compute/read

Table 2.2. Functions related to the PMS.

The only complex function in the PMS is 'It should be possible to see a burn down chart of the current iteration as well at the entire project'. The more detailed description of this function is given in section 4.7 on page 72.

Design 3

This chapter concerns the part of the project where the system architecture, for the program to be developed, is defined. This is done through defining important design criteria and considering different architectural patterns. This will be the foundation for the implementation of the system.

3.1 Design Criteria

When considering architectural design, the starting point is the system requirements, derived from the analysis of the problem- and application domain. Further, describing the system architecture also includes system-operation requirements. Describing these requirements leads to considerations about, to which degree quality criteria, such as security, usability and efficiency for instance, should be prioritized in the final system. This section will be based on [MMMNS01, chapter 9].

3.1.1 Design Conditions

In the context of the architectural design part of OOA&D, the concept 'conditions' features which possibilities and limitations the development will be subjected to. This is similar to the 'conditions' element in the FACTOR criterion, mentioned in subsection 2.1.3 on page 15 - though, when considering conditions in this part of OOA&D, the conditions are described a bit more in detail. It is important to specify the conditions, which are linked to the development of the software to gain knowledge about, how the system should be structured. In this subsection the typical conditions will be explained.

There are two primary conditions for the design of the PMS. The first is the fact that the PMS is being developed in an educational context. Learning is a primary objective, and the group will also have to adhere to the goals of the study regulations for the SW3 semester. This means that the PMS is not intended for professional use, meaning that a design criterion as 'reliability' for instance, would not get a maximum rating of importance. Moreover, the final product of the PMS has to 'compete' with writing the report in terms of allocating resources. This could mean sacrificing a bit of testability (which has been rated important) in order to both squeeze out the last bit of functionality, but also to meet the deadline of writing the report.

The second main condition for designing the PMS, is the experience (or lack thereof) of the group members in developing systems in general, and web development in particular. Some group members have previous experiences with web development, whereas this is an entirely new domain to others. Using ASP.NET MVC is new to all in the group and will therefore naturally affect some of the design choices and priorities. In other words, we cannot use what we do not know about, so we might be unaware of some possibilities that seasoned ASP .NET MVC developers would have in their arsenal.

3.1.2 Prioritizing Conditions

The conditions described in subsection 3.1.1 on the preceding page will form the basis for prioritizing the design criteria, which is to be examined in this subsection. These criteria will be placed into five different categories; very important, important, less important, irrelevant, and easily fulfilled. By creating this prioritizing checklist, seen at table 3.1, the development will be leveled at the more important parts of the design.

Criterion	Very important	Important	Less important	Irrelevant	Easily fulfilled
Usable			X		
Secure		X			
Efficient			X		
Correct		X			
Reliable			X		
Maintainable			X		
Testable		X			
Flexible	X				
Comprehensible	X				
Reusable					X
Portable				X	
Interoperable				X	

Table 3.1. Checklist for prioritizing design criteria for the PMS.

There are two viewpoints on usability. One has to do with how easy it is for users to use the system. The other has to do with how well the system utilizes the technical platform without causing any bottlenecks.

Usability is deemed less important for the PMS, though the PMS is targeted exclusively at students studying computer science, thus being able to handle systems with less than optimal usability. Furthermore, as the PMS should not manage large amounts of data there will not be problems with bottlenecks.

Security is deemed important for the PMS. Since the PMS is the only place where data about user stories, tasks, deadlines, etc will be saved, it is important that outsiders cannot freely gain access and change or delete data. A precautionary measure could be to create a login requiring a strong password. The reason why this is not categorized as 'very important' is that the information stored will probably not benefit any external persons. Therefore attacks will be highly unlikely.

Efficiency is less important as the system will only manage a rather small amount of data.

Correctness refers to fulfillment of requirements and is deemed important to the PMS. The PMS is developed as a student project, and it is important to fulfill the requirements from the study regulations.

Reliability is the fulfillment of the required precision in function execution. The goal is of course to have reliability as close as possible to 100% and avoid catastrophic failures, which can render the system unavailable for extended periods of time. In the PMS, reliability is rated less important. That is, reliability is primarily considered as a by-product from trying to produce high-quality code (which will avoid serious errors).

Maintainability has to do with how well a program handles changes. This can arise from changing customer requirements or a change in the computing environment. Maintenance usually accounts for the most effort in a given software project due to changing or evolving requirements. The software system in this project however differs substantially from a normal software system. The system is not going to be maintained beyond the lifetime of the SW3 semester and there is no customer which can require new or changed functionality. As a result, maintainability is prioritized as less important, but will be derived as a side effect from the rating of flexibility as 'very important'.

Testability refers to how easily a computer program can be tested. Testability has a significant impact on the quality of the resulting software and the degree of testability also directly influences most other quality criteria. Testability encourages breaking a system up into manageable pieces which can be tested in isolation. This means that errors can be quickly identified and fixed before they propagate into the rest of the system. This obviously also helps in relation to a system's maintainability and reliability (and others). Testing is an integral part of XP and due to this, the quality criterion of testability has been rated important for learning purposes. Though this criterion has been rated important, it should be mentioned that web-applications are inherently more difficult to test than regular terminal based programs for instance. This topic will be discussed further in chapter 5 on page 83.

Flexibility refers to how easy it is to modify the deployed system and is deemed very important for the PMS. The PMS is developed in an educational context, and flexibility will therefore be rated as very important. This will be done by using different design patterns e.g. the composite pattern and the architectural Model View Controller (MVC) pattern enabling the system to be more dynamic and modifiable.

Comprehensibility has been rated very important for the PMS. As the system is developed as a group effort it is important that all group members know what is going on. This is important from an educational point of view where group members are going to defend their knowledge of the system at the exam. It will also encourage good programming practices, e.g., using agreed upon coding standards, as well as communication between group members. Pair programming and technical memos will be used to ensure a high degree of understandability.

Reusability, portability, and interoperability are not relevant for this project. It is not the intention to reuse the PMS (or parts of it) in other applications, nor is it an issue to be able to move the PMS to other platforms, and the PMS is also not going be integrated into other systems.

3.2 Architecture

Using the design criterion from section 3.1 on page 32, this section will consider the design of the system architecture. The main source for this section will be [MMMNS01, chapter 10] .

When using the OOA&D approach, system architecture is processed as component architecture. A component is defined as a collection of program parts that constitutes a whole and has well-defined responsibilities. Using components, a system structure is transferred into a component architecture composed of interconnected components. To define the component architecture of the PMS, some architectural patterns have been studied. The first pattern used, is a general pattern, concerning the basic decomposition of the system into components, such as model, function and interface. By doing this, allows to take a look at other patterns that can help structure these components into a relation, which fits the goal of the PMS.

The layered architecture pattern is a classic and is used as a basis pattern in other architectures. The idea of the layered architecture is to designate components as layers. A layer may consist of several components. This design lets each component define its responsibilities as well as which interfaces it provides upwards and downwards. Downward interfaces concerns which operations the current layer is able to access from the underlying layer. Contrary to this, the upward interface describes which operations the current layer is providing to the layer above. Internally in each layer, it is important that the design of its components is complying with the interfaces, which the layer is designed to provide. Else, layers below and above will not be able to use important functions which might impair a system. In the structure of layered architectures, it is implicitly assumed, that a layer uses the layer below, thus changes in a layer pointed to, may affect the layer pointed from.

One of the patterns which is using this layered architecture, is the generic architecture pattern. The primary use of this pattern is to elaborate the structure of basis systems which incorporates interface-, function- and model-components. The model-component, which contains the information of the problem domain, is placed as the lowest layer. The function-component is on top of it, and followed by that is the interface. The function-layer provides the operations possible to execute on the model. Those functions are passed on to the users by the interface layer.

The architecture of the PMS is constructed using the client-server architectural pattern. The client-server pattern is a widely used pattern in the software industry. The main advantage using this pattern is, that it easily solves the problem of getting a variety of personal computers (PC) to communicate with a central mainframe server - allowing a lot of clients to use the server without being a part of a common whole, see figure 3.1. The components in client-server architecture are several clients and a single server. The client-server pattern provides an asymmetric design. That is, the server does not presume any kind of knowledge about its clients. Instead, its clients must know which functionality the server is providing.

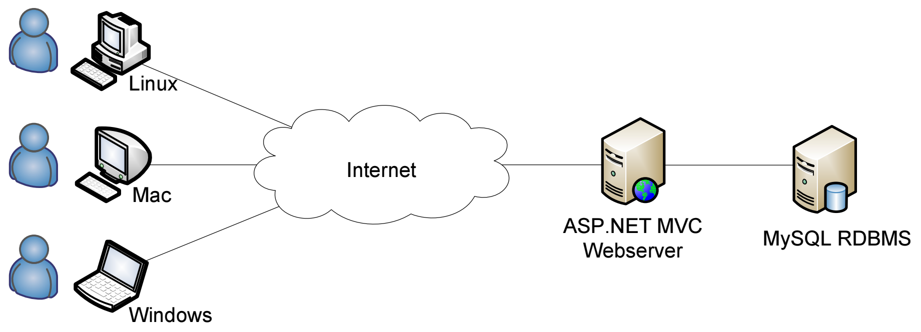


Figure 3.1. The client-server architectural pattern.

In client server architecture, the server is responsible of providing what is common to different groups of clients using it. That is, the server must provide functionalities and data related to specific user groups. This means, that it is the servers liability to ensure, that all data relating the model is saved - this could be done, using a database for instance. The functionalities of the server are most commonly distributed to the clients over a network, letting the clients be responsible of providing its own interface for its users. This means, that if you have a server, providing the functionality of managing a software development project, the server will provide code to its clients, typically using Hyper Text Markup Language (HTML), to let the clients be responsible of rendering an interface to its users.

To get the structure of the PMS to fit the architectural pattern of client-server, the idea of the generic architecture pattern and the architectural MVC-pattern has been used. This also complies with the design criterion 'flexibility' in section 3.1.2 on page 34. As seen in figure 3.2 on the next page, the architecture of the PMS is built up using three major layers. The top layer consists of all the clients connected to the server providing an interface to its user using the functions provided by the View interface. A client is concerned as a PC running any sort of Operating System (OS) with access to a web-browser, letting any kind of users, such as Windows, Linux and Mac be able to use the PMS. The primary responsibility of the View interface is to allow each client component to handle the interaction between the PMS-actors and the PMS-functionalities. The interface layer encompasses the view component. This component is responsible for creating the GUI to send to the clients based on objects retrieved from the model through functions in the function component. Also, the view states which function in the function component should handle the users' requests. As an example, a user clicks on a link in the management system and the view forwards the request to the appropriate controller in the function layer. The function layer then takes actions based on the request. As a result a new view will be rendered to the user. As the design criteria 'usability' in section 3.1.2 on page 33 is deemed less important, the interface will not be further investigated.

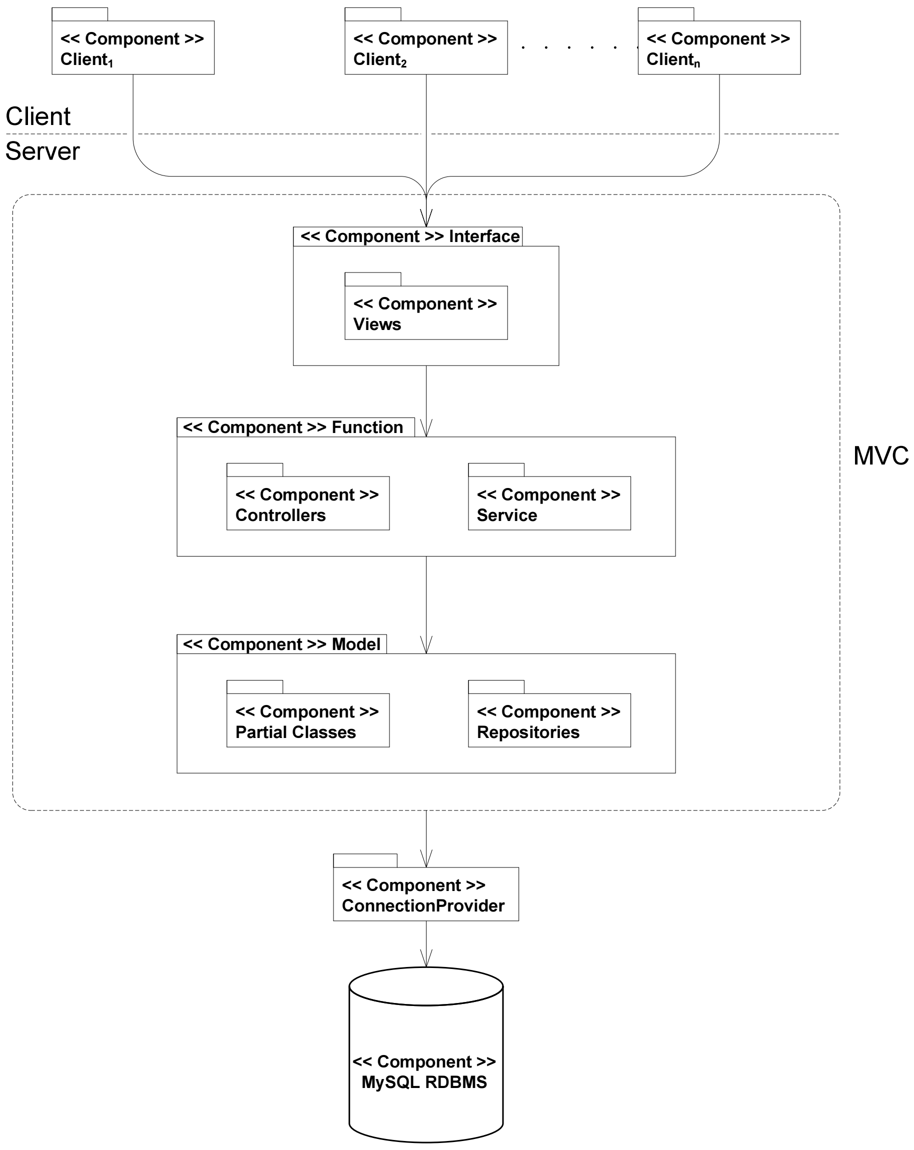


Figure 3.2. The architecture of the PMS.

The middle layer is the functionality layer, containing the controller- and service component. These two components are responsible for providing functions to the view layer. That is, if the user for instance, is submitting a form in a view, then the controller must handle the input, and decide what to do next - it could be some calculations combined with interventions in the model resulting in a returned view, showing the computed results. Concerning the service component, this is responsible for providing additional functions to different parts of the system. More on these two components in section 4.3.1.3 on page 56.

The final major layer is the model, consisting of partial classes. The model component is responsible for keeping track of the objects which are concerning the problem domain, and further it is responsible of managing the data in a consistent way. This will be revisited in section 4.3.1.1 on page 53.

The MySQL Relational Database Management System (RDBMS) is required for the PMS to be able to store its data. The database is not required to run on the same server running the PMS, thus it should be available over a network.

Finally, to let the clients of the server, being able to connect to the PMS from anywhere in the world, the server is connected to the internet. Between the server and the database, a data access layer has been placed. Inside this layer, repositories for querying data between the server and the database have been placed. The idea of using such repositories is to abstract the data access away from the function layer, and into a separate data access layer. By doing this, all data querying is managed in this layer. This means, that if the database for some reason should be changed to another type, the only place to correct data access will be in the data access layer. As regards to the design criteria in section 3.1 on page 32, this choice of design is agreeing upon the high prioritization of flexibility in the system.

Client-server architecture can be distributed in different ways. An approach could be, having 'fat' clients, making the application logic, such as the model and functionalities, reside at the clients [Bur98]. Another approach could be using 'thin' clients. That is, the server is doing all the processing and data-management instead of the clients, letting the thin-clients only be responsible of invoking which functionalities the server should process [Uni09b]. The PMS is using thin-clients, giving the server the responsibility of handling all the work relating to data-processing and data-management. This can also be referred to, as using a 'local-presentation' when it comes to defining the distribution of client-server architecture.

The pros of choosing this architecture:

1. Multiple users have access to the same data, as it is stored at one location. This means for instance, that the same data concerning a specific project managed by the PMS can be distributed to each of the clients connected to the PMS-server. Thus, the system allows a group of several students to use the system simultaneously.
2. Any kind of browser, OS and internet connection is capable of using the system. This is an important advantage, as students might have their own preferences when it comes to OS and web-browsers.
3. The model is one unit, and is stored in a RDBMS, ensuring that all relevant project data is consistent and updated when a client accesses and uses the system.
4. The functions, which adds different types of functionalities to the model, can be separated according to the type of client connected. This means, that if a supervisor is connected to the PMS, the supervisor can be restricted to some segments of functionalities - for instance he might only be able to post comments and not be able to create and edit tasks.

The cons of choosing this architecture:

1. The University hosting and providing the PMS must have access to a server, providing ASP. NET MVC and a RDBMS.
2. In case the server has a breakdown the system cannot be used, and the users cannot reach the information stored in the database.

3. If many users try to gain access to the system at the same time this could cause the system to succumb to the pressure. This will be discussed in more detail in 5.3 on page 87.

3.3 Model Component

The model component is where the result from the problem domain is implemented. The basis for designing the model component is primarily the class diagram, event table, structures and dynamics from the object oriented analysis. The object of this activity is to get at class diagram for the model component to use for implementation of the system. The design class diagram depicting the model component of the PMS can be seen in figure 3.3.

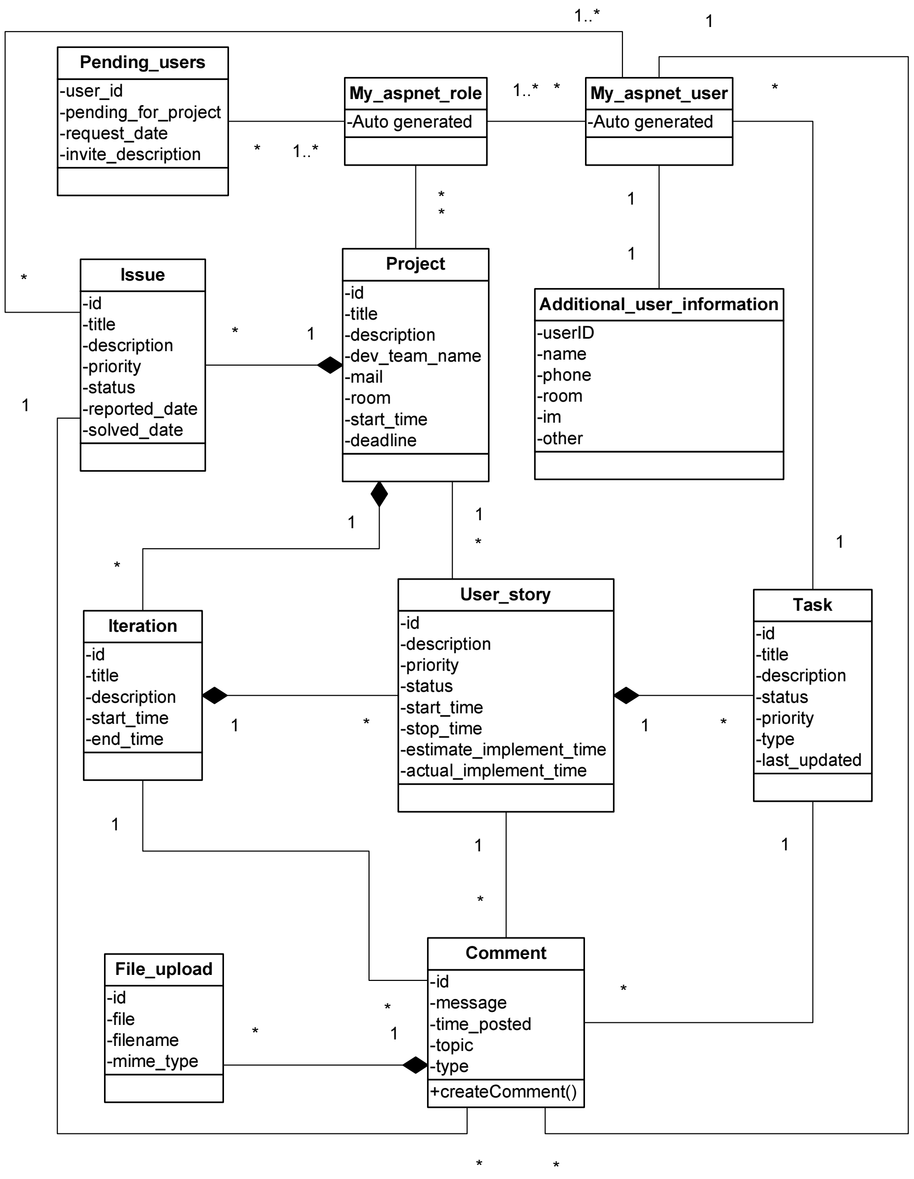


Figure 3.3. Design class diagram for the PMS.

To identify new classes for the design class diagram it is often necessary to examine each event from the analysis to find 'events that occur in iterations'. This means, if an event appears on a class more than once, and data regarding it, needs to be stored, then a new class would be added using an aggregation structure.

When analyzing the PMS, the system was already available 'on paper'; meaning experience from previous projects provided us with knowledge about which data needed to be stored. Thereby a class such as 'comment' was already identified in the early analysis.

In the following, each class from the design class diagram depicted in figure 3.3 on the preceding page will be reviewed.

Project This class is defined in the analysis and kept as it is needed to store e.g. a description of the project, a deadline, the name of the developer team etc. The class is related to roles allowing any user which is a member of the specific project role, to enter the project. Further, the class has an aggregation to iteration and issue. The reason why an issue is not an aggregation of iteration instead, is that an issue is not attached to a particular iteration.

Iteration This class is defined in the analysis, and is retained to save e.g. the title of an iteration. The class is aggregated into user stories, and is related to a comment, as it is possible to comment on iteration.

User_story This class is defined in the analysis, and is kept to save e.g. the priority and description of a user story. The class is aggregated into tasks, and is related to comments, as it is possible to comment on a user story.

Task This class is defined in the analysis, and is kept to save e.g. the title, description and priority of tasks. The class is related to comments as it is possible to place a comment on a task. Furthermore, it is related to a user which for instance will make it possible to see which users are assigned to a particular task.

Issue This class is defined in the analysis, and is retained to save e.g. title, priority and status of an issue. It is associated with the comment- and user class, which makes it possible for any user, which is either a supervisor or developer, to either report or comment on an issue. It is only developers who are able to solve issues though. Using this structure makes it possible to tell who has reported an issue as well as who is assigned to a specific issue. The issue class is an aggregation of project. The reason why an issue is not an aggregation of an iteration or user story is that it should be possible to bug report an issue without having it to be implemented/fixed during an iteration.

Comment This class is defined in the analysis, and is kept to store data about e.g. the message and the time when a comment was posted. It is related to Iteration, User Story, Task and Issue as it is possible to place a comment on instances of those classes. The comment class has as well an aggregation to the file_upload class as it is possible to attach a file to a comment. To make it possible to track who has posted a particular comment, a relation between comment and my_aspnet_user is made.

File_upload This class is derived by examining the problem and application domain. In the event table, the event *'File attached'* has been defined. As this event happens more than once, the attached files along with the user who attached it, must be stored, and to do so a class must be defined. The File_upload class is an aggregation to the comment class, and will therefore in the PMS always be related to a comment. Hereby it will be possible to see who attached the file, as the comment class has a relation to my_aspnet_user.

My_aspnet_user As defined in the FACTOR criterion the PMS should run on an ASP.NET MVC web server, which will make it possible to automatically generate (when creating login) a default class to hold the most basic user information, such as id and name. This class is related to five other classes. The relation to the issue class makes it possible to identify who has reported, and who is assigned to a particular issue. The relation to the comment class makes it possible to see who posted a comment (and thereby a possible attachment). The relation to additional_user_information makes it possible to add further information about users without altering the system-generated my_aspnet_user class. The relation to the task class, makes it possible to see who is attached to a specific task. The relation to the my_aspnet_role class helps to manage which roles a user have, making it possible manage restrictions in the system.

My_aspnet_role This class is derived by examining the problem domain, and has been auto generated by the system. The class is used to manage the distribution of roles in the system according to the logic decided. The general logic is, that if a user is registered with a role name, corresponding to the id of a particular project, then the user has access to that specific project. By using this setup, the system will be able to manage exactly which projects a specific user is granted to access. Further, using this role-setup also allows the system to distinguish between developers and supervisors, as the logic has been composed to require a user to be either a supervisor or a developer.

Additional_user_information This class contains the information from the abstract *user* class from the analysis. It is related to the automatically generated my_aspnet_user class and makes it possible to add further data to a user.

Pending_users This class is defined on account of the need to control who is joining a project. A pending user is a user who have registered and applied for a project but still not granted access. The user, when sending his request, is able to write a short explanation of why he should be granted access to a particular project. And as this event is able to appear more than once, a new pending_users class has been made.

In the analysis a class Student Group was defined. The need for this class will be unnecessary as users will take a role and apply for a project. In this way users who are in the same project will be considered as a student group. In chapter 4 on page 45, the design class diagram will be taken into account, as this chapter will walk through how the system has been implemented.

3.4 Function Component

The function component is the layer of the system that links the users of the system together with the model. Following the approach of OOA&D, all functions defined in section 2.3.2 on page 30 must be transformed into operations, which can be activated on the basis of user input or requests. In general, an operation can be seen as a procedure; it is activated by a message, it executes a prescribed data processing and finally, it returns to the place from where it was activated either with a result or not. As many of the functions in the system to be built, shares a similar structure, this section will consider a couple of distinct functions, and describe how they should be implemented as operations and thereby, how the objects involved should behave. This will be done using sequence diagrams and models from [MMMNS01, chapter 7].

As one of the design criteria prescribes, that the system must be built flexible using ASP.NET MVC, the 'Controller' layer provided by this architectural pattern, will be the place to handle the input from the user. This means, that when a user sends a request to the system, the 'Controller' component will handle the request by invoking the appropriate operations in the system. By doing so, the model will be separated from that part of the system, which handles the interaction with the user, keeping the model simple and flexible. This further applies to the 'flexible' design criterion, highly prioritized in section 3.1 on page 32.

The first function to consider is defined as: *It should be possible to report an issue.* This function is a simple update function concerning the users opportunity to report issues found in a project, managed by the system to be built. As illustrated in figure 3.4 the function will be activated from the problem- and application domain by the Reviewer Actor that was defined in subsection 2.3.1 on page 23. That is, the Reviewer triggers a specific function trying to edit the state of the model.

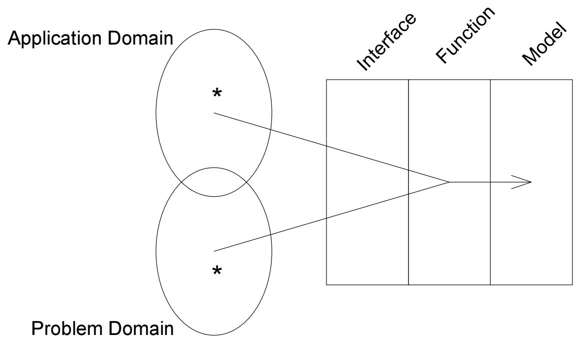


Figure 3.4. Update function activated by an event in the application domain, which is trying to update the state of the model in the problem domain [MMMNS01, chapter 7]. The asterisks shows the initiator of the function while the arrow shows the effect of the process.

As the particular update function will be placed in a controller in the function layer, the actor from the application domain sends his request directly to the function layer. An action in the controller which corresponds to the user's request will then change the state of the model. Back to the user will be returned an updated output.

The controller taking care of the issue reported by the Reviewer, will be named 'IssueController'. Inside this controller, the action responsible for the needed model interventions will be named 'Report'. As depicted in the sequence diagram in figure 3.5, the 'Report' action starts by creating a new instance of the issue class. Next a request for a new connection is sent to mysqlconnection. A new connection is created and returned to the IssueController. The new issue object is then updated to reflect the issue described by the Reviewer. Followed by that, an instance of the IssueRepository class is created and returned. On this new object, the Add() operation is invoked, taking the new issue as parameter. The IssueRepository will then invoke the AddToIssues() method on the mysqlconnection object to add the new issue to the list of system issues. A success message will be returned to the IssueRepository and forwarded to the IssueController. Subsequently the IssueController will call the Save() operation in the IssueRepository which will invoke the SaveChanges() operation on the mysqlconnection object to save the list of issues to the database. A success message will be returned to the IssueRepository and forwarded to the IssueController.

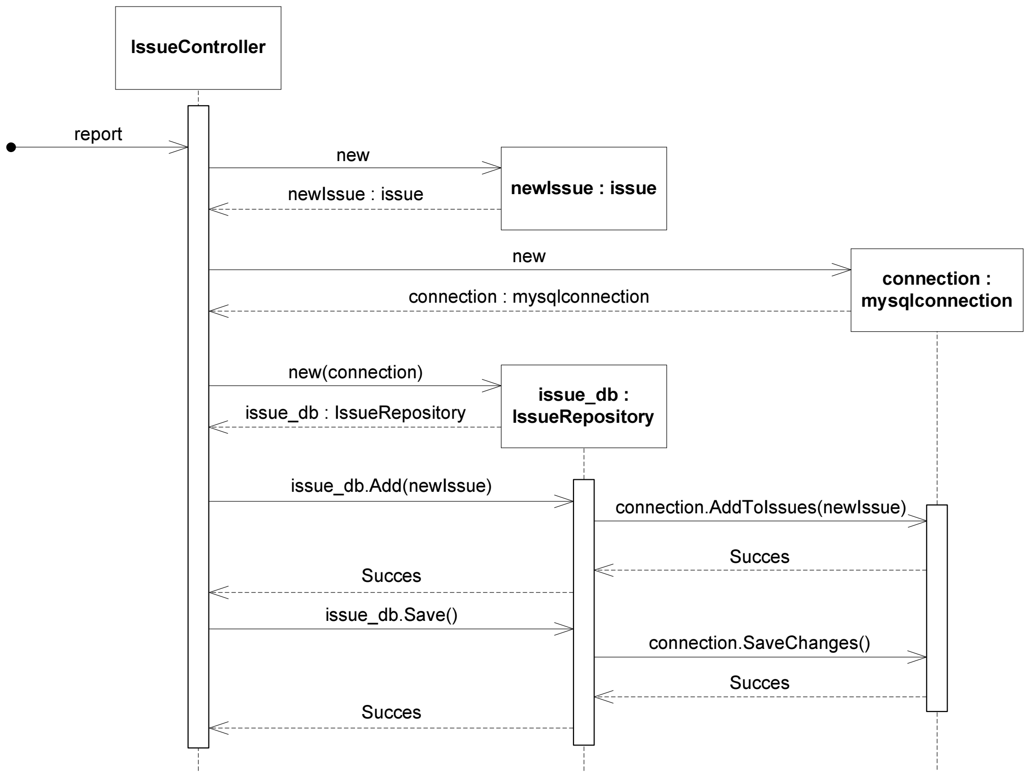


Figure 3.5. The behavior of the involved objects, in reporting and saving an issue, is depicted in this sequence diagram.

The next function to consider is of type 'Read', and is defined as follows: *A user should be able to see profile details.* The function considers the Reviewer's opportunity to get an overview of which details are registered about him. As depicted in figure 3.6 on the next page the function is invoked by the Reviewer in the application domain, requesting some information which is gathered by the Controller from inside the Model.

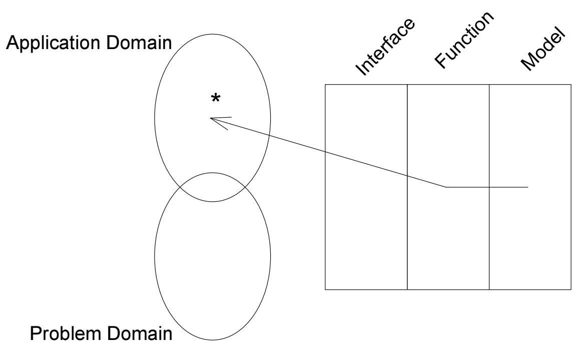


Figure 3.6. Read function activated by an actor in the application domain [MMMNS01, chapter 7]. The asterisk shows the initiator of the function while the arrow shows the effect of the process.

'MyProfileController' will be the name for the controller taking care of the Reviewer's request for reading his profile details.

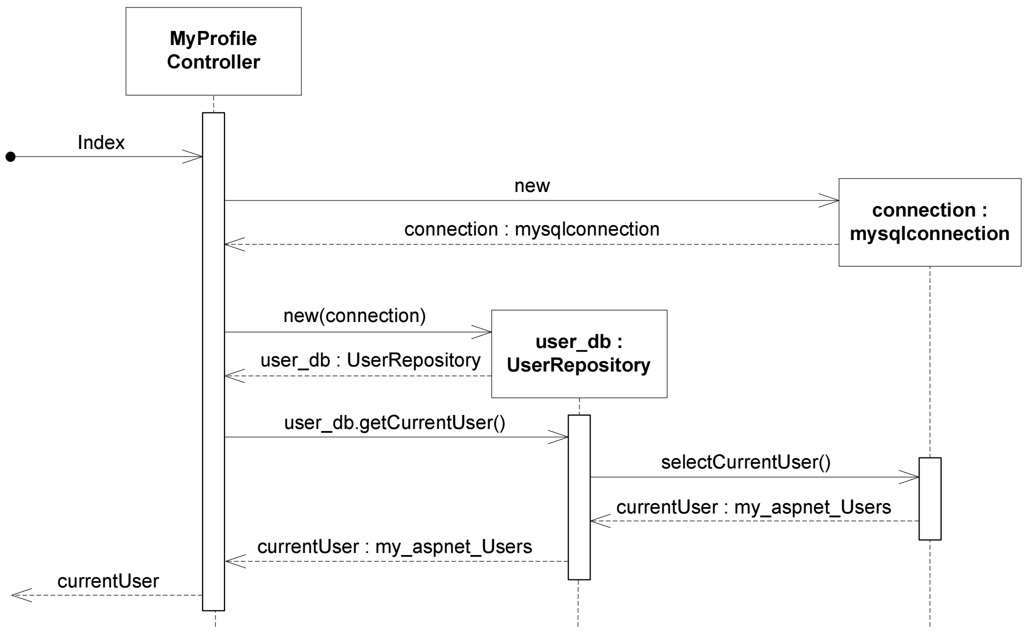


Figure 3.7. Sequence diagram illustrating the behavior of the object concerning the display of profile details.

As depicted in figure 3.7 the read function starts in the MyProfileController. A request is sent to mysqlconnection for a new connection. A new connection is created and returned to the MyProfileController. A new instance of the UserRepository is created and returned from the model. On this object, the getCurrentUser() operation is invoked. The UserRepository invokes the SelectCurrentUser() operation to retrieve user information from the mysqlconnection. Subsequently the currentUser information will be returned from the mysqlconnection as an object and forwarded to the MyProfileController. Finally, this object will be returned to a view, and displayed to the user.

Implementation 4

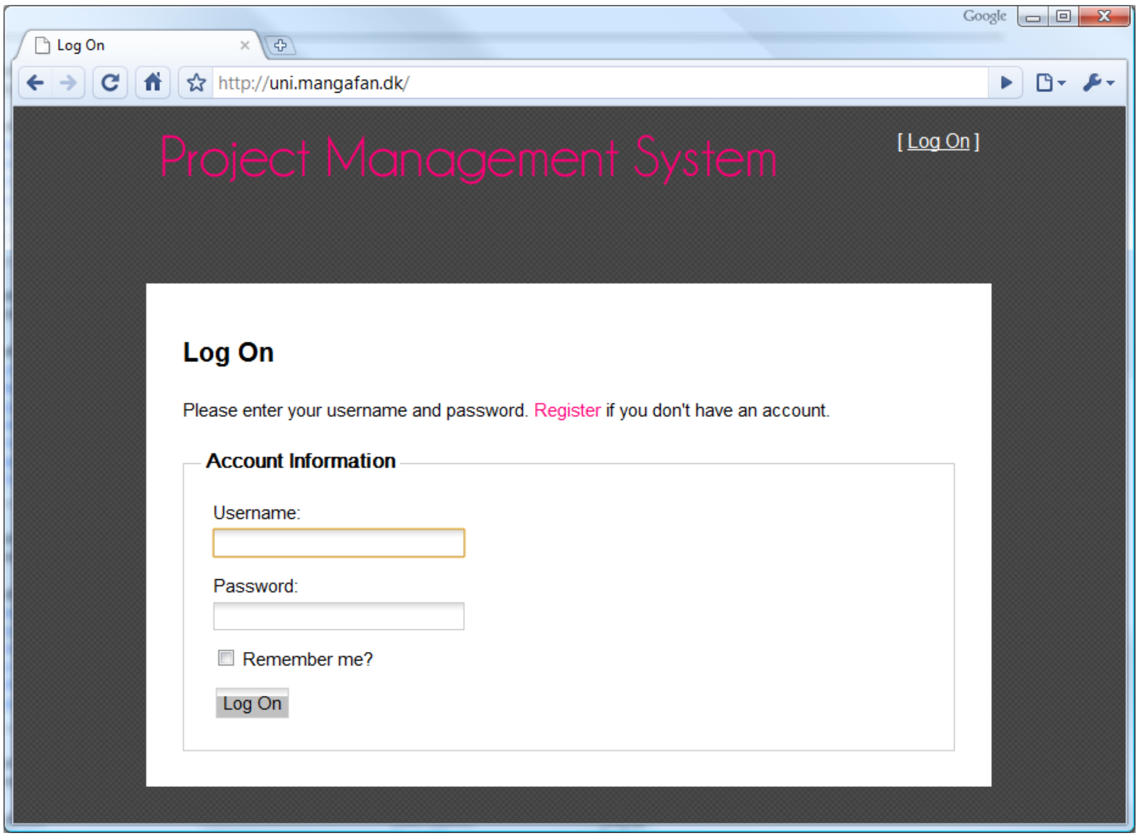
Using the previous two chapters as underlying basis this chapter will cover the implementation of the PMS. The chapter has been divided into several parts, each describing a specific module, functionality or segment of the system.

4.1 Program Presentation

This section will give an introduction to the developed program. The main concern will be presenting how we have implemented some of the requirements specified by the system definition in subsection 2.1.3 on page 15.

4.1.1 Login

When a user wants to use the PMS, he starts by opening a web client such as Google Chrome. In this client an URL pointing to the place where the PMS is hosted, must be invoked. For instance, the URL hosting our version of the PMS is: `http://uni.mangafan.dk`. When the web client has finished rendering the HTML output, the user will be presented for a login screen as depicted in figure 4.1.



Log On

Please enter your username and password. Register if you don't have an account.

Account Information

Username:

Password:

Remember me?

Log On

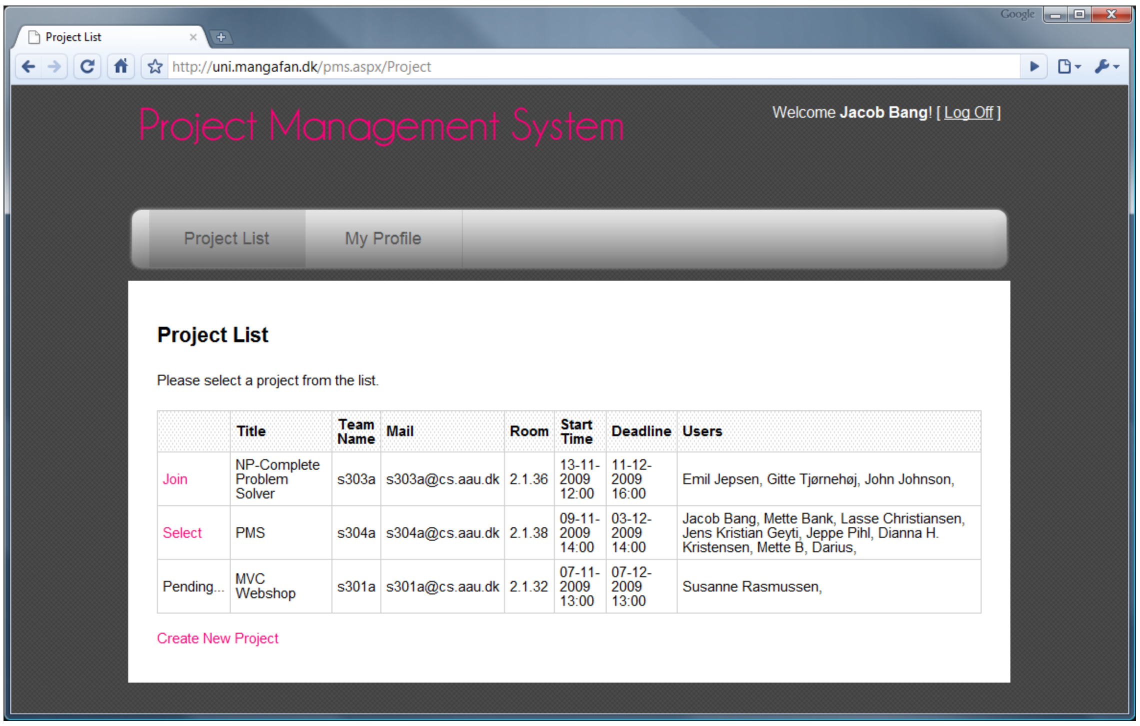
Figure 4.1. The login screen of the PMS.

If the user is new, he needs to create a new account by using the *Register* function. Creating a new account is simple, and the user only needs to provide basic information such as username, email and most important a strong password of minimum ten characters. This design choice has been made to comply with the design criterion of *Security* which has been deemed important in section 3.1 on page 32.

After providing the required information, the user may click *Register*. Now, the system verifies the input, and if everything is valid, the user new account is created, and the user will be signed in. If some of the information provided was invalid, such as no email, the user is not registered, and will get a warning indicating what to correct.

4.1.2 Project List

To comply with the system-definition, the PMS supports multiple study groups and thereby several projects. The project list depicted in figure 4.2 is the first element presented to the user when signed in. In this list, the user can see all projects managed by the PMS for a particular University or institute for instance. The user is able to select those projects in which he is registered as a member. If the user wants to join a specific project, he may click *Join*, write a short 'why-should-I-have-access' message, and then wait for the members of the project to handle the request. As a third option, the user is also able to start a new project.


Project List
http://uni.mangafan.dk/pms.aspx/Project
Project Management System
Welcome **Jacob Bang**! [Log Off]
Project List
My Profile

Project List

Please select a project from the list.

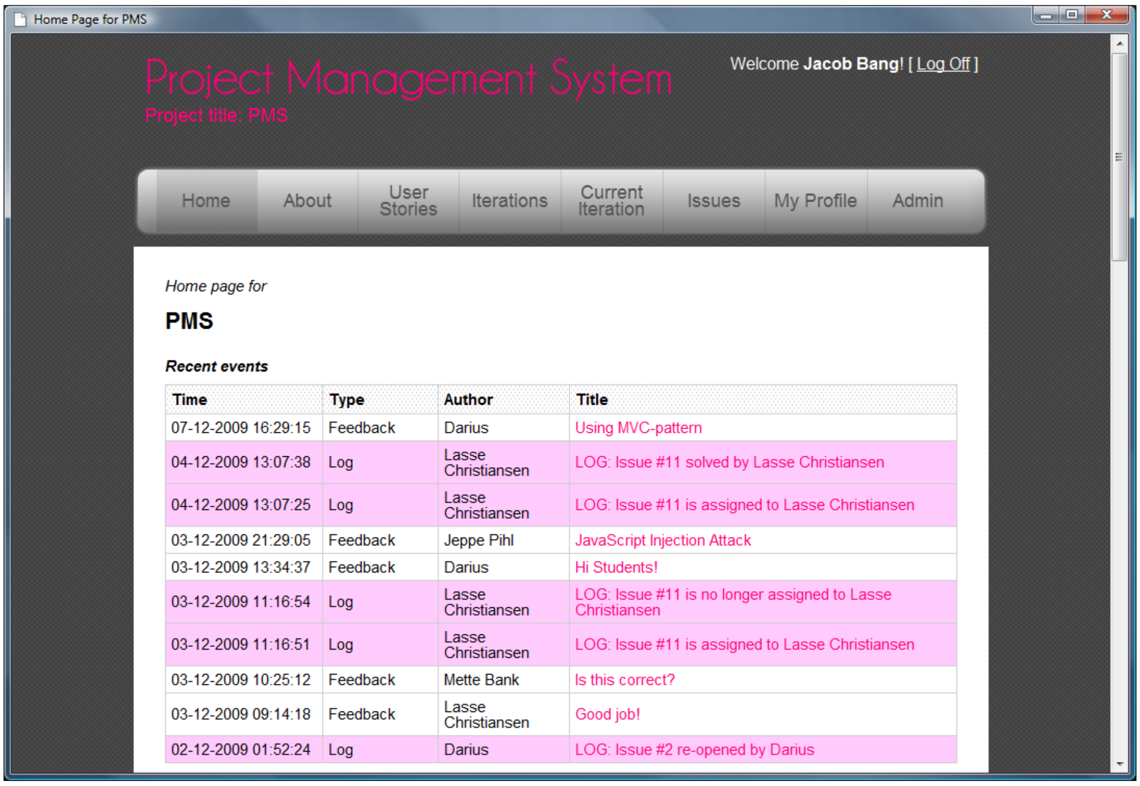
	Title	Team Name	Mail	Room	Start Time	Deadline	Users
Join	NP-Complete Problem Solver	s303a	s303a@cs.aau.dk	2.1.36	13-11-2009 12:00	11-12-2009 16:00	Emil Jepsen, Gitte Tjørnehøj, John Johnson,
Select	PMS	s304a	s304a@cs.aau.dk	2.1.38	09-11-2009 14:00	03-12-2009 14:00	Jacob Bang, Mette Bank, Lasse Christiansen, Jens Kristian Geyti, Jeppe Pihl, Dianna H. Kristensen, Mette B, Darius,
Pending...	MVC Webshop	s301a	s301a@cs.aau.dk	2.1.32	07-11-2009 13:00	07-12-2009 13:00	Susanne Rasmussen,

Create New Project

Figure 4.2. The project list of the PMS.

4.1.3 Home

One of the approaches made to comply with the requirement of enabling the supervisor affiliated with a project, to get an overview of how a project is progressing, is the implementation of log and in particular the burn down chart. These approaches is the first shown to the user when a project has been selected from the project list. As depicted in figure 4.3 the log shows the ten recent events, such as if an issue has been solved or re-opened. The burn down chart gives an overview of how the student group is doing when it comes to solving user stories. The chart provides statistics about how many user stories have been 'burned' per day, how the students are following the ideal burn down and how many user stories are left. The implementation of the burn down chart also complies with the part of the system definition, saying that *statistics concerning implementation of user stories should be implemented using burn down charts and progress bars.* More about burn down chart in section 4.7 on page 72.



Home page for

PMS

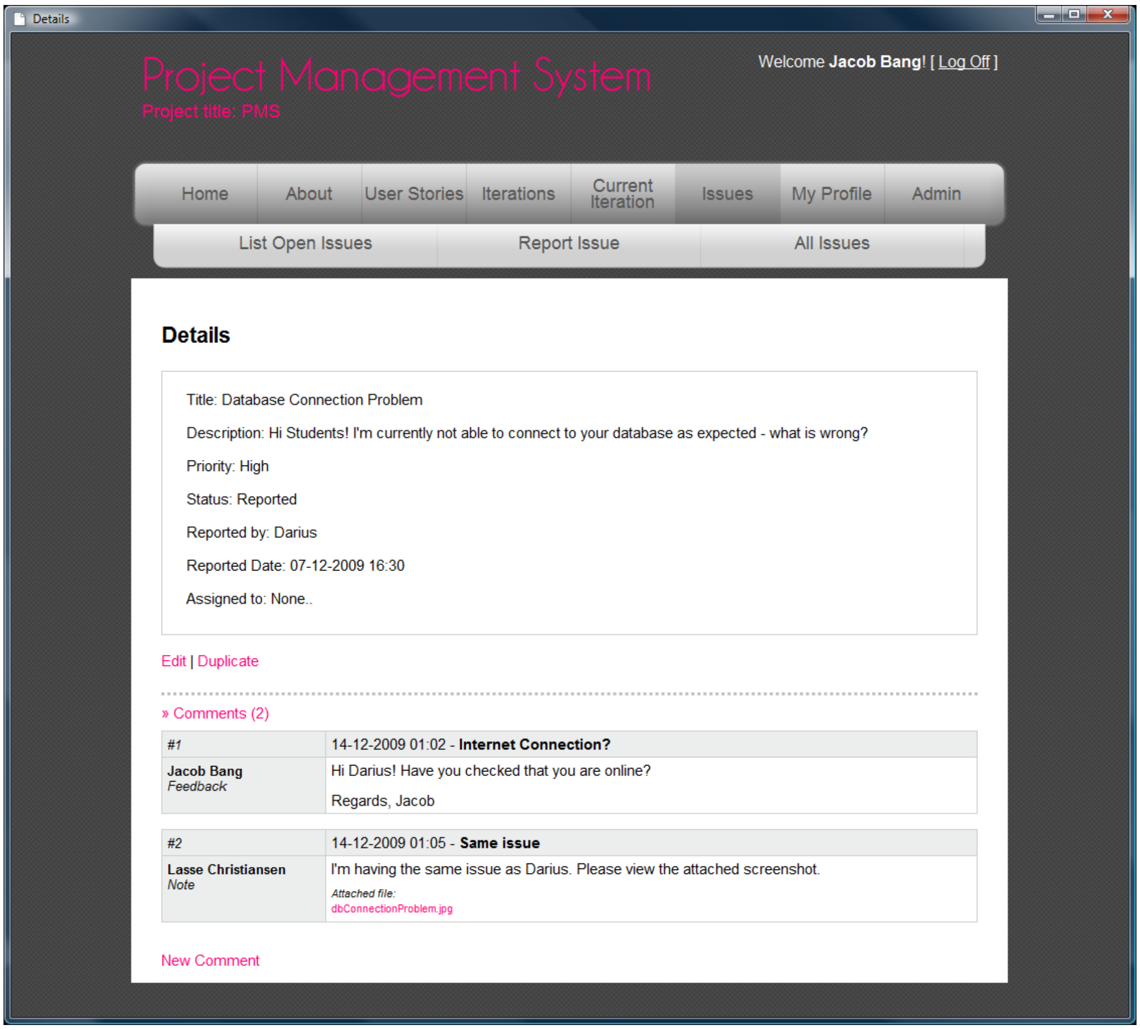
Recent events

Time	Type	Author	Title
07-12-2009 16:29:15	Feedback	Darius	Using MVC-pattern
04-12-2009 13:07:38	Log	Lasse Christiansen	LOG: Issue #11 solved by Lasse Christiansen
04-12-2009 13:07:25	Log	Lasse Christiansen	LOG: Issue #11 is assigned to Lasse Christiansen
03-12-2009 21:29:05	Feedback	Jeppe Pihl	JavaScript Injection Attack
03-12-2009 13:34:37	Feedback	Darius	Hi Students!
03-12-2009 11:16:54	Log	Lasse Christiansen	LOG: Issue #11 is no longer assigned to Lasse Christiansen
03-12-2009 11:16:51	Log	Lasse Christiansen	LOG: Issue #11 is assigned to Lasse Christiansen
03-12-2009 10:25:12	Feedback	Mette Bank	Is this correct?
03-12-2009 09:14:18	Feedback	Lasse Christiansen	Good job!
02-12-2009 01:52:24	Log	Darius	LOG: Issue #2 re-opened by Darius

Figure 4.3. A segment of the home screen of the PMS, showing the recent events of the particular project.

4.1.4 Comments

As defined in the system definition *the system should inspire to plan, document and develop using agile development...* To accomplish the part concerning documentation, we have made it possible to place a comment on any iteration, user story, task or issue. The feature of viewing comments has been implemented as a hidden list, which is related to each specific item. That means, that if the user is viewing details about a specific item, an issue for instance, a link named *Comments* will be placed below its details. If the user clicks the link, a list of all comments related to the particular item will then drop down, and the user will be able to view all posted comments. If the user wants to add a new comment, he might do so by invoking the *New Comment* function. In figure 4.4 a list of comments has been invoked by the user.



Details

Title: Database Connection Problem

Description: Hi Students! I'm currently not able to connect to your database as expected - what is wrong?

Priority: High

Status: Reported

Reported by: Darius

Reported Date: 07-12-2009 16:30

Assigned to: None..

#1	14-12-2009 01:02 - **Internet Connection?**
Jacob Bang *Feedback*	Hi Darius! Have you checked that you are online? Regards, Jacob

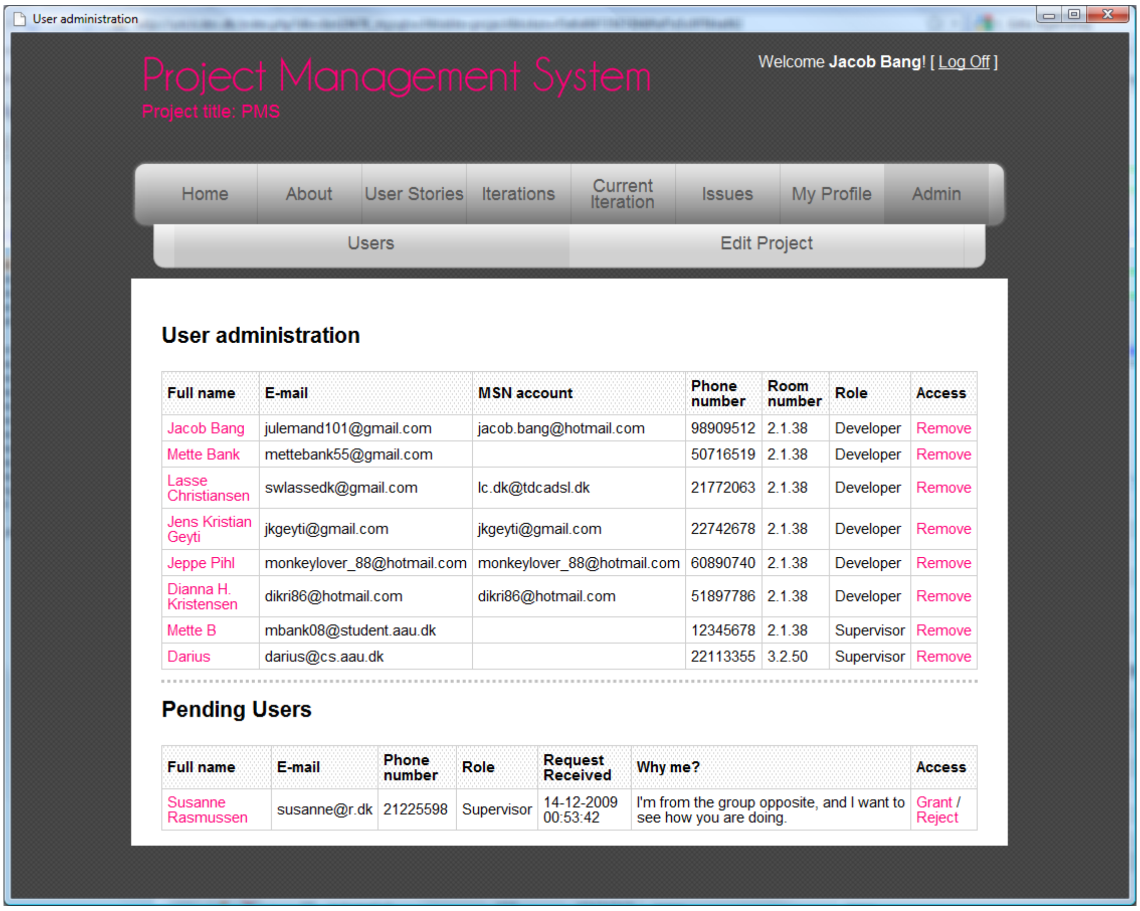
#2	14-12-2009 01:05 - **Same issue**
Lasse Christiansen *Note*	I'm having the same issue as Darius. Please view the attached screenshot. *Attached file:* dbConnectionProblem.jpg

Figure 4.4. Issue details, with user viewing the comments related.

Along with a comment, the user is able to attach a file such as a screenshot for instance. This might be a helpful feature if a user wants to be more specific about a certain issue, or if a user have made some drawings, supporting a comment regarding an algorithm and so on.

4.1.5 User Administration

As the PMS is built to manage multiple projects, a need for some kind of user administration was a matter of course, which also is reflected in the system definition. The main idea of the user administration is to divide all users into roles. When a user is being created, he specifies if he is a developer or a supervisor. That means, a user is always either related to the developer role or the supervisor role as outlined in the model component in section 3.3 on page 39. When it comes to projects, the user has access to a given project, if he has the role, which is related to the particular project. That if, is the user is member of a project with `id=7`, then the user has the role named `7`. Using this system, the PMS is able to manage exactly who has access to which projects, and further determine which privileges they should have. This also makes it possible for all members in a project to administrate if a user should be removed from a particular project. The user administration is build upon trust, that is any developer has privileges to remove any user in a project. When a user is pending for a project, developers are also responsible for granting or rejecting access to the pending users. An example of the user administration panel can be seen in figure 4.5.



User administration

Full name	E-mail	MSN account	Phone number	Room number	Role	Access
Jacob Bang	julemand101@gmail.com	jacob.bang@hotmail.com	98909512	2.1.38	Developer	Remove
Mette Bank	mettebank55@gmail.com		50716519	2.1.38	Developer	Remove
Lasse Christiansen	swlassedk@gmail.com	lc.dk@tdcadsl.dk	21772063	2.1.38	Developer	Remove
Jens Kristian Geyti	jkgeyti@gmail.com	jkgeyti@gmail.com	22742678	2.1.38	Developer	Remove
Jeppe Pihl	monkeylover_88@hotmail.com	monkeylover_88@hotmail.com	60890740	2.1.38	Developer	Remove
Dianna H. Kristensen	dikri86@hotmail.com	dikri86@hotmail.com	51897786	2.1.38	Developer	Remove
Mette B	mbank08@student.aau.dk		12345678	2.1.38	Supervisor	Remove
Darius	darius@cs.aau.dk		22113355	3.2.50	Supervisor	Remove

Pending Users

Full name	E-mail	Phone number	Role	Request Received	Why me?	Access
Susanne Rasmussen	susanne@r.dk	21225598	Supervisor	14-12-2009 00:53:42	I'm from the group opposite, and I want to see how you are doing.	Grant / Reject

Figure 4.5. The user administration panel of the PMS.

4.2 Applied Technologies

There have been used different types of technologies in this project. In the following a brief overview will be given of them all. Some of the technologies will be revisited later with the purpose of describing how they have been used in relation to the PMS.

ASP.NET is a web platform built on the .NET Framework. Using ASP.NET, a developer will be able to program web platforms targeted at smaller companies or private web sites. Coding ASP.NET can be done using any language which is compatible with the runtime environment of the .NET Framework, that is Visual Basic or C# for instance [Net09]. Coding ASP.NET makes it easy for drag and drop developers to quickly get a website running, but when it comes to test and larger scale data binding, the quality of the ASP.NET is not remarkable [Tav08]. ASP.NET will be revisited in section 4.4 on page 64.

ASP.NET MVC is an architectural pattern which is used to divide an application into three main components: Model, View and Controller. ASP.NET MVC can be seen as an alternative to the ASP.NET Web Forms pattern, making it possible for developers to create MVC-based ASP.NET web applications. The ASP.NET MVC framework is integrated with existing ASP.NET features, such as master pages and membership-based authentication [Cor09c]. This will be revisited in section 4.3 on page 52.

MySQL is an acknowledged free and open source RDBMS, providing multiple user access and Structured Query Language (SQL) [MM09]. The use of the MySQL RDBMS in the PMS will de described further in section 4.5 on page 67.

ADO.NET is a base class library in the .NET framework, designed to provide .NET developers with access to different types of data-sources. With ADO.NET comes .NET Framework data providers, giving the .NET developer an opportunity for connecting to databases, on which commands can be executed and query results received. By using the *DataSet* object, provided by ADO.NET, relational data can be handled as in-memory data, which is represented as objects in the system. This is a very useful feature which has been beneficial in the PMS [Cor09b] [Cor01].

MySQL Connector/NET is an ADO.NET driver written to enable .NET developers to create applications that require connectivity with a MySQL RDBMS. The driver is written in C# and implements the required interfaces into the ADO.NET library [MM08].

LINQ is an abbreviation for *Language Integrated Query.* LINQ is a .NET component that adds the capability for querying in a variety of data sources like arrays, enumerable classes, XML, relational databases and third party data sources [Mic09].

C# is an object-oriented type-safe programming language. We have been introduced to this language through the Object Oriented Programming (OOP) course related to this SW3-semester. The language has been used together with Microsoft's C# compiler and the .NET Framework. Further, the language has been chosen as it allows us to follow the object oriented paradigm [Cor07].

Microsoft Chart Controls is a library that enables developers to rapidly develop visual representations of datasets. Microsoft Chart Controls is well known for its use in Microsoft Excel. The library supports a large amount of different chart types, such as Bars, Columns, Lines, Areas, Pies, Points and Ranges. Charts can be created from various data sources such as XML-, Excel- and `.csv` files or from arrays and `IEnumerable` objects. Microsoft Chart Controls will be revisited in section 4.7.3.1 on page 75.

HTML stands for *Hyper Text Markup Language* and is a markup language for web pages, providing the means to create structured documents. HTML only gives the document its structure such as headings, paragraphs, and links etc.

CSS is short for *Cascading Style Sheet.* CSS is a style sheet language meant to format the look of any given document from the XML family. CSS is mostly used in combination with HTML where it provides the HTML-document with its layout, colors, background, fonts and much more.

jQuery is a free JavaScript library which makes it easy to implement event handling, animations and different kinds of interactions into HTML documents. It supports a different range of browsers ranging from Internet Explorer 6.0 to Google Chrome, making it desirable for implementing, as almost any client will be able to view the effects powered by the jQuery library [RtjT09]. The library has been used for implementing a Date and Time picker and a hide / show animation into the PMS.

SVN is short for *Subversion.* SVN is an open-source system for revision control, making it possible to manage different versions of files located in a SVN-repository. During the development of the PMS, SVN has been used together with our Integrated Developer Environment (IDE) - more on this in the following.

PHP has been used for developing various scripts for stress testing the system. In conjunction with cURL, a library for connecting and communicating with different types of servers with various protocols (including HTML), PHP provides an easy way of emulating user actions on websites (loading websites and images, handling session cookies, posting forms etc).

When developing the PMS, we have used two major tools; an IDE and a database interface. These will be presented in short:

Microsoft Visual Studio 2008 (MSVS) is the IDE used to develop the PMS. It is a product licensed by Microsoft Corporation, and supports the developer with a variety of different tools. In the development of the PMS the primary tools used has been debugging, different explorers, code auto-completion, built-in documentation etc. MSVS supports different plug-ins - one of the plug-ins we installed in MSVS was AnkhSVN, enabling us to revision control the source code of the PMS.

phpMyAdmin is a free and open source tool for managing MySQL databases through a simple web interface programmed in PHP. Tables, cascades, primary- and foreign keys etc becomes easier to manage using this GUI tool. We deployed a version of phpMyAdmin on a webhost, and used the tool almost any time when changes was to be made inside the MySQL RDBMS.

4.3 MVC

The PMS has been developed from an architectural pattern called Model-View-Controller (MVC). This section will describe what MVC is and why it has been chosen for this project. The description will go into detail about the individual elements of the MVC-pattern, and include code samples from the PMS, showing how the individual elements of how this pattern has been used in the system. Further, the description will go through how ASP.NET MVC has been used in the development of the program.

4.3.1 Overview of the MVC

MVC is an architectural pattern that describes how the structure should be composed in a system. It is widely used in systems such as web shops, as it provides a usable way of separating various components. Further it adds the functionality of verifying user access, determining if a user should be granted access to specific components or not. This whole thing is done by splitting the system up into three components: Model, View and Controller, where each component has its own purpose and responsibility specified in the documentation of the pattern. This means that, by using the MVC pattern, it becomes possible to structure a system in accordance with a well-defined documentation. This gives the development a solid framework from the beginning, defining how the structure should be created [Cor09c].

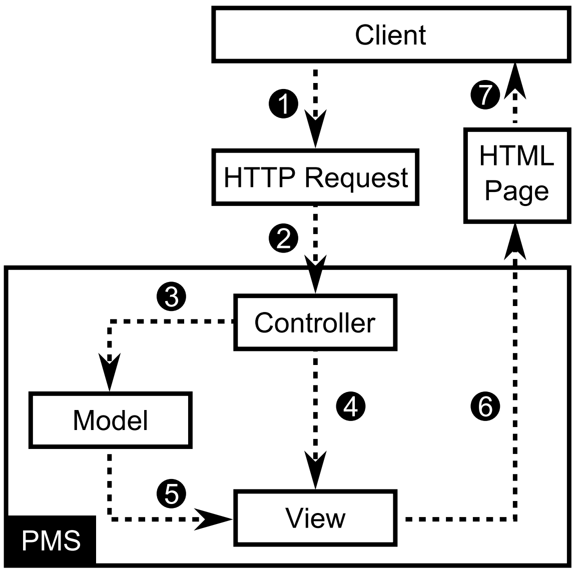


Figure 4.6. Example of how the MVC architecture is used in the PMS.

Figure 4.6 does not only show how the MVC is used in the PMS, it also gives an insight into how communication takes place in the PMS. By showing this, makes it easier to understand the communication which takes place between the system and the user (client). As an introduction to the description of the individual components of MVC, a description of how a simple request will be processed by the PMS will first be explained. The goal is here to provide a better understanding of how the components are linked together, combined with a description of which purpose each component has in the system.

1. First the client (user) goes into the PMS system or performs an action inside the system (e.g. clicks on a link). This allows the browser to make a HTTP request.

2. The system tries to forward the request to a controller which satisfies the particular client-request. This is done by the URL mapping system and is one of the features that ships with ASP.NET MVC. The URL mapping will be described in further detail in the section about URL mapping in subsection 4.3.2.1 on page 60. After the controller has received the data from the HTTP request it takes care of the information from the request, and may forward some of the information to the model. Then the controller can choose to send a view to the user (which is the most common option), output a file, switch user to another page or choose to do nothing (uncommon).

3. The model receives an order from the controller and makes the necessary changes in the model. This could be saving data from the user into the database for instance. More about the model in subsection 4.3.1.1.

4. If the controller chooses to send a view to the user, the controller typically sends a reference to an object from the model, to the selected view.

5. As described before there is often an object reference related with a view. The purpose of such a reference is, that a view can ask the model for data to be used in the view, and thus it becomes possible to print out data based on the particular object reference - data which will be visible to the user.

6. The view's task is to create the GUI of the system and send it back to the client. In a Web system it usually implies that there must be returned HTML code out to the client.

7. The user's internet browser translates the HTML code to a GUI and prints it to the user's screen. It is then possible for the user to perform a new function in the system and start over in the process.

4.3.1.1 Model

The purpose of the model is to implement the logic into the system. The term logic should be understood as a concept covering how the program is intended operate, what rules should be applied between objects and how data should be stored in the system. An example might be a Task object which receives its information from a database, modifies this information and then updates the information in the database to reflect the objects current state [Cor09c]. In short, the model should contain all the application logic that is not contained in a view or a controller. This is business logic, validation logic, and database access logic [Cor09f].

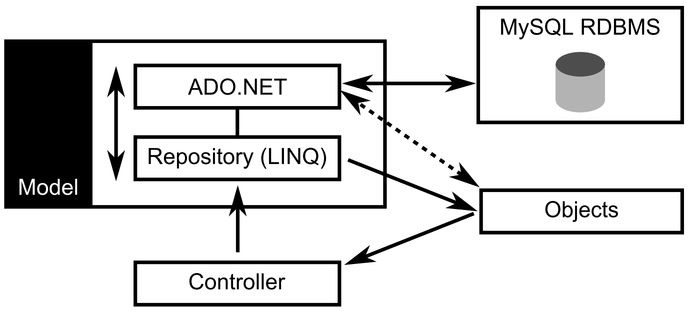


Figure 4.7. Example of how the Model is used in the PMS.

In the PMS the model is primary used to communicate with the database. The actual communication with the database is not directly but through a data access layer, generated by ADO.Net. Figure 4.7 shows the internal structure of the Model component in the PMS.

The first thing that happens in the figure, is that a controller sends a request to a repository to get some data (e.g. a Task). The Repository contains the code to obtain information from the ADO.Net component. This code is in the PMS system mostly written in LINQ (e.g. a method which gets all tasks in a selected user story). In the ADO.Net layer there is auto generated classes of all the types of information contained in the database. This means that the repository will get back an answer in the form of objects. These objects are then passed out as response to the controller. When the system needs information from the objects, the ADO.Net layer will then communicate with the database and retrieve the necessary information. The controller can also change, add or remove information from the objects. If updated information should be saved in the database, the controller needs to send the objects back to the repository which contains the necessary commands to get ADO.Net to save the information in the database.

4.3.1.2 View

The view's job is to make a GUI which can be routed to the user. Often a view receives an object from the model component and thereby being able to retrieve necessary information from this object. This could e.g. be a view which receives a Task object. If the view has been programmed to use that kind of object, it is now able to get all the properties related to that particular Task object. This means that a controller can choose to send any Task object to this view, knowing that the information about the task is sent back to the user. It is also possible to create views that take a list of objects as input (e.g. a list of task objects). In such a case, it would be normal if the view handles the list of objects by printing out a table which is listing the information of each Task object.

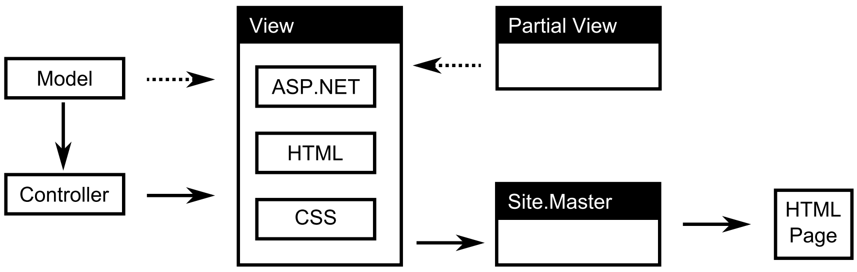


Figure 4.8. Example of how the view is used in the PMS.

In the PMS, the view is not the only component responsible of creating the GUI. By using ASP.NET and ASP.NET MVC, it is possible to use technologies like master pages and partial views. As shown in figure 4.8 it is possible to make views which contains references to one or more partial views. A partial view is like a view which accepts an object as parameter. Therefore it is possible to move some functionality away from a view and into a partial view. It is also possible to allow multiple views to share the same partial view.

An example of this can be seen in code sample 4.1 line 1 showing how to refer and send an object as parameter to a partial view. In this case the *Model* is an issue and contains a list of comments in the variable *comment.* The list is forwarded to the partial view which generates the HTML code to display the comments.

```
1  <% Html.RenderPartial("~/Views/Comment/View.ascx", Model.comment); %>
2  <br />
3  <%= Html.ActionLink("New␣Comment", "CreateComment", "Comment", new { returnUrl = Request
       .RawUrl, commentOnType = "issue", commentOnId = Model.id }, null)%>
```

Listing 4.1. Example on how the partial view is used in the PMS. The code comes from *Views/Issue/Details.aspx.*

Line 3 in code sample 4.1 shows how to generate a link to the comment method in the PMS. In this way it is easy to show all the comments and add a link that makes it possible to write a new comment. The `Html.ActionLink` method is used to create a link to a specific method in a Controller. In this example the method creates a link with the name `New Comment` which points to the `CreateComment` method in the `Comment` controller. The additional parameters are used to send information to the `CreateComment` method:

returnUrl is a parameter used by `CreateComment` to know what page the user should be directed to after a new comment has been posted. In this example, the user will be redirected to the page on which he was at the time, clicking the `New Comment` link.

commentOnType defines what type of element the user is trying to comment - an issue for instance.

commentOnId is needed to tell the `CreateComment` method to which object the comment should be attached. In this example, ID will be the ID of the selected issue.

In total, by using partial views for this purpose, we do not need to create a new `CreateComment` method every time we want to add the functionality of placing a comment somewhere in the system. We only need to paste the code in listing 4.1 on the previous page and modify the parameters.

After the view has been generated (including the partial views if any) the page will be forwarded to a master page (in this system called `Site.Master`). The purpose of the master page is to split a page, making code, which always should be loaded, to be located in one place. This applies to the HTML header and a number of design elements such as the menu and includes of CSS and JavaScript. The master page takes all the elements and builds a complete HTML page and then sent the page out to the user.

4.3.1.3 Controller

The controller is the component handling the interaction with the user, working with the model, and finally it is the component which is responsible for selecting a view to render and send back to the user [Cor09c]. In ASP.NET MVC it is possible to send variables to the controller, either by the URL or by using POST variables. In section 4.3.2.1 on page 60 there will be a description about URL-mapping and how the system knows which controller to invoke. That section will also describe how variables are sent within the URL.

The normal job of the controller is to get input from the user, use it to manipulate the model, and then send back a view. The controller is a class with methods as shown in figure 4.9. In the figure, the controller to which the HTTP Request is pointing to, have four methods and these methods points to four views. It is also possible for a method to point on more than one view, nothing or something else as files like pictures.

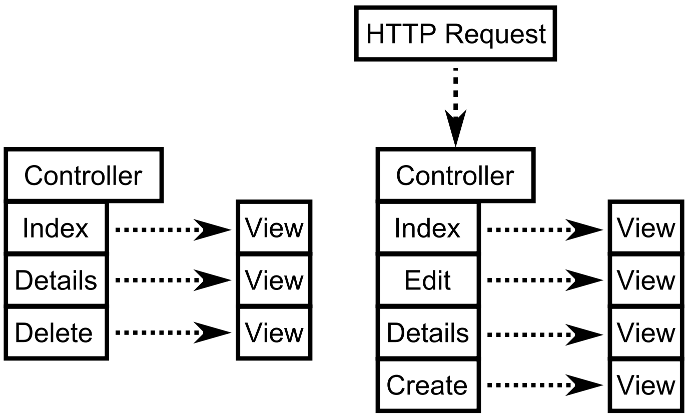


Figure 4.9. Example of how controllers are used in the PMS.

A Controller often consists of several methods and in the PMS the same applies. These methods are called actions in ASP.NET MVC. In figure 4.9 there is four actions: `Index`, `Edit`, `Details` and `Create`. As an example, the selected controller is a `TaskController` and is used to administrate all the actions, the user can do with tasks in the system. This example can be seen on figure 4.10 on the next page and will be reviewed in the following text. The aim of the example is to explain the communication between the user and the components in the system, when the user is trying to edit information in a selected task.

This example does not come directly from the PMS but the principle is used several places in the PMS.

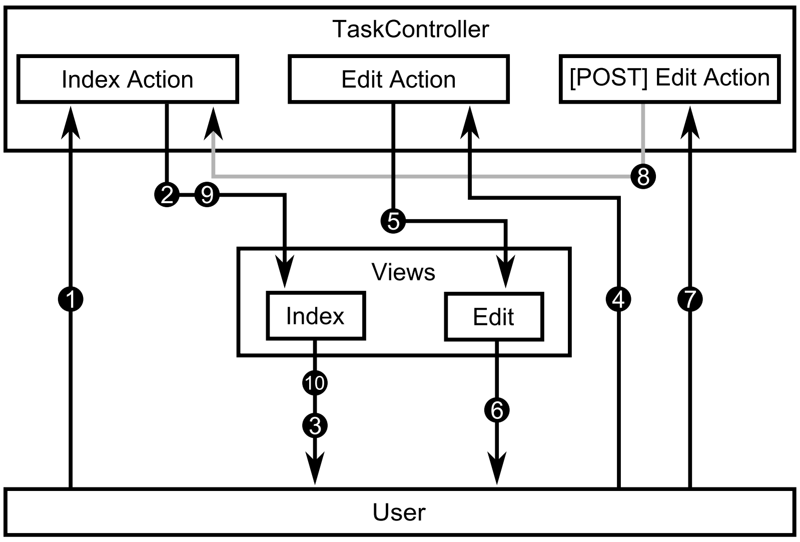


Figure 4.10. Shows an example of the communication between the actions, a user and the views.

1. The user sends a request to the `Index` action in `TaskController`.
2. `Index` action returns a list of tasks to the `Index` view.
3. The view generates a HTML page which lists all the tasks. It also adds a link with every task pointing to the `Edit` action. This makes the user able to click edit on a specific task and thereby getting the opportunity to update some of the information in the task.
4. The user clicks edit on a task and sends a request to the `Edit` action. The task ID is sent together with the request.
5. The `Edit` action returns the selected task to the `Edit` view. In this way, the user can see the updateable information already registered for the particular task.
6. The `Edit` view returns a HTML page with a form, where the fields contains the updateable information of the selected task.
7. User updates some information in the task and clicks on the submit button. The browser sends the form information to a special `Edit` action taking POST variables as parameters. More about POST variables in section 4.3.2.1 on page 60.
8. After the `Edit` action (POST version) has updated the model with the new task-information, it forwards the user to the `Index` action. If the user has made an invalid input when updating the task, the `Edit` action is also capable of sending the user back to the `Edit` view, highlighting the invalid fields.
9. The `Index` action returns a list of tasks to the `Index` view (the same as step 2).
10. The user is back to the list of tasks and can select another task to edit.

A remarkable thing about the controller is, that all user requests needs to be send to a controller. This implies that the controller can check if the particular user has access to current controller. This check can be done before one single line of controller code has been executed, making it possible to distinguish among who should be granted access to different parts of the system. This feature is used in the integrated security system from ASP.NET MVC as seen in section 4.8.2.1 on page 79.

It is important to understand the task of the controller component in MVC. The controller should only contain minimum of logic required to return the correct view (or to redirect to another action). Everything else should be controlled in the model. In principle, developers should aim at fat models and thin controllers. If the controller becomes too large the developer should strongly start thinking about moving some of the logic away from the controller and in to the model [Cor09f].

Code example of a controller
The code in listing 4.2 shows how simple an action in a controller is coded. This example also shows how actions work when it comes to communicating with the model and how replies are sent back to a view. In the following text the code will be reviewed.

```
1  namespace PMS_dev.Controllers
2  {
3      public class ProjectController : Controller
4      {
5          [Authorize]
6          public ActionResult Index()
7          {
8              mysqlconnection connection = Connection.newConnection();
9              ProjectRepository Project_db = new ProjectRepository(connection);
10             return View(Project_db.Select());
11         }
12         ...
13     }
14 }
```

Listing 4.2. Example on how a Controller with an Action is used in the PMS. The code comes from *Controllers/ProjectController.cs*.

Line 1 defines the namespace of the controller. All controllers are placed in the same namespace and also at the same directory. **Line 3** defines the `ProjectController` class that extends the `Controller` class. All controllers extends this `Controller` class which provides lots of functionality, such as returning views, redirecting to another action and so on. **Line 5** is used to define user restrictions and is described in section 4.8.2.1 on page 79. **Line 6** declares the `Index` action in the `ProjectController` and takes no input parameters. In section 4.3.2.1 on page 60 there will be examples on actions taking input such as POST and GET variables.

In **line 8** the action gets a connection to the MySQL database. This connection is made because it is important that all communication to the MySQL database is using the same connection in the same action. More about this topic in section 6.1.1 on page 91. The connection is sent as a parameter in **Line 9** to the `ProjectRepository`. All communication with the `ProjectRepository` object `Project_db` will then use the created MySQL connection. `ProjectRepository` and the connection to the MySQL database are classes in the model component.

Line 10 is using the `View()` function to return a `ActionResult`. There are many different variations of the `View()` method, but in this example it takes an object as parameter. This object is forwarded to the view, which uses the object go generate its HTML code. The object in this example comes from the `Select()` method in the `Project_db` object that returned a list of all projects in the MySQL database.

4.3.1.4 Services

The service component is not a part of MVC but has been found necessary building during the development of the PMS. As described earlier in section 4.3.1.2 on page 54, the views have the role to generate HTML code. However, there are a few situations where it has been necessary to write several lines of code to achieve a desired functionality. To avoid having a lot of duplicated code to float around in the system, it was necessary to make a service component that could contain the static functions.

Also, in the `ChartController` there has been need for placing some main functionality in static functions inside the service component. The need occurs, as huge complex functions should not be placed inside the controllers. Instead they, according to MVC, should have been placed in the model to keep the controller thin and simple. However, the only affected controller is the `ChartController`, which uses some methods from the service component concerning chart generation.

To describe the problem in detail; the reason why this problem has occurred is, that chart generation is a grey zone when it comes to ASP.NET MVC design. The question is where to place the chart functionality in the MVC model. The controller is a good option because it takes the request from the user, uses the model to get the relevant data and is able to return an image. But the method that actually creates the image should be a view component according to MVC. In ASP.NET MVC the view component is made in ASPX files and is responsible for generating HTML code. It would be strange if the chart functionality should be placed between these files, as the chart is generated by C# code and do not contain any HTML. It would also be a little strange if the model should contain the code, because the chart is just a graphical representation of data in the model. Therefore it was decided to place the chart generation methods into the service component. It has been done by placing all the chart generation classes into a separate namespace in the service component as illustrated in figure 4.11.

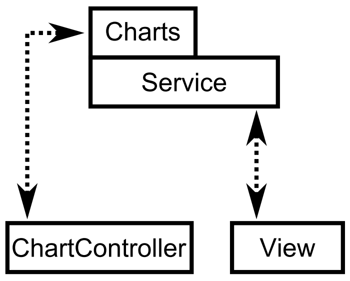


Figure 4.11. The figure shows how the service component is used by the `ChartController` and views. The service component only contains static classes with static methods.

4.3.2 Introduction to ASP.NET MVC

It is possible to develop a system directly from the MVC architectural pattern, but since the PMS has been chosen to be programmed in ASP.NET and as the design criterion of flexibility has been prioritized very high, it was natural to use the ASP.NET MVC framework. A brief description of ASP.NET MVC can be found in section 4.2 on page 49.

ASP.NET MVC comes with a variety of features which automate some of the things behind the communication between components in MVC. One of the features that has been used a lot, is the possibility to manipulate the URL and use it to send variables that makes the URL looks more simple. This is an advantage because it allows the user to look at an URL and see what it refers to. In the following section 4.3.2.1 the URL-mapping will be described [Cor09c].

The main advantage of ASP.NET MVC is that it is enormously flexible. It is therefore possible to include other technologies and these may be more or less integrated with ASP.NET MVC. It also adds some functionality to Microsoft Visual Studio 2008 which makes it easy to create controllers and views and navigate between them. But ASP.NET MVC also requires that developers know what MVC is when it comes to software development; where to place code, why using models and so on. Else, they would not be able to take full advantage of the architectural pattern making the system architecture inconsistent and not functional in the MVC way. There is a lot of documentation on the homepage belonging to the MVC pattern, but initially it can be difficult to find information about all the important details.

4.3.2.1 URL-mapping

As described previously URL-mapping is one of the major features in ASP.NET MVC. It operates as a system, responsible for activating controllers and actions according to which URL has been invoked. It is able to take and pass on variables sent via the URL. The PMS uses two URL-maps which both will be described in this section.

The first URL-map is based on the standard URL-map that comes with an ASP.NET MVC project. The following example uses figure 4.12 to explain the elements in the URL and how ASP.NET MVC URL-mapping use them.

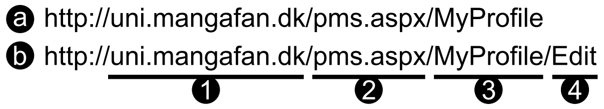

a http://uni.mangafan.dk/pms.aspx/MyProfile
b http://uni.mangafan.dk/pms.aspx/MyProfile/Edit
1
2
3
4

Figure 4.12. Two PMS-compatible URL strings using the first URL-map.

In the following, the integers displayed in figure 4.12 on the preceding page will be explained in detail:

1. This is the hostname or address to the server. The URL-mapping do not use that part of the URL, as it must be static.
2. This part has been added as a standard URL element, as the server is running an older version of IIS. In brief the reason why this is needed, is due to that the server has trouble understanding, that it can render the page without an .aspx file extension is present in the URL. In the PMS, there is no `pms.aspx` file present, which makes this URL-element being a fix to get the PMS running on the older IIS server. How this has been solved in the URL-map can be seen in listing 4.3 on the following page.
3. This place specifies the name of the controller that should be activated in the system. In this example it is the MyProfileController (the system automatically adds 'Controller' to the name).
4. This is the name of the action to be invoked. The action should be defined in the controller pointed to in item 3.

In figure 4.12 on the preceding page it is also shown, that it is not mandatory to specify all parameters in an URL used by the PMS. In the figure, the first URL 'a' will point to the standard action defined, as no action has been given. This can be seen in line 4 in listing 4.3 on the following page, which in general shows the URL-map code that is used to interpret the URLs shown in figure 4.12 on the preceding page.

```
1 routes.MapRoute(
2     "NoProject",
3     "pms.aspx/{controller}/{action}/{id}",
4     new { action = "Index", id = "" }
5 );
```

Listing 4.3. Shows the URL-map for URL strings that do not specify the ID of a project. The code comes from *Global.asax.cs.*

This code shown in listing 4.3 is not the most used map route in the PMS, as it is only used when the user tries to access the PMS main page, where projects can be selected or the user profile can be updated. The first parameter (line 2) is the name of the map route. This is used if a view needs to make a link to another place in the PMS, as such an operation requires the use of a specified map route. The next parameter (line 3) is where the map route is specified. First the string `pms.apsx` is specified as outlined before - to make the PMS compatible with older versions of IIS. The next tags defines how the URL should be split into the variables: `controller`, `action` and `id`. If these elements are not specified in the URL request, the URL-map takes the defaults values from line 4. In this example, the default action is set to `Index` and `id` to nothing.

The next URL-map is used when the user has selected a project, and thereby this is the primary URL-map used in the PMS. In addition to the URL-map in listing 4.3, the project title and id is added. The title is only present to inform the user about, which project the current URL will point to. Besides that, the title has no effect to the URL. Instead, the project id is the one used to identify which project the user has currently selected.

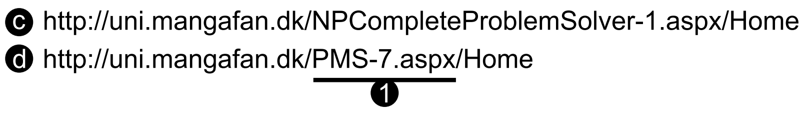

c http://uni.mangafan.dk/NPCompleteProblemSolver-1.aspx/Home
d http://uni.mangafan.dk/PMS-7.aspx/Home

Figure 4.13. Displays two URL strings using the second URL-map.

As seen in figure 4.13 the URL marked with 'd' and the part underlined by the integer '1' contains the project title: *PMS*, the project id: *7* and an `.aspx` extension. The code behind this, available in listing 4.4, is made nearly the same way as the URL-map in listing 4.3. The only thing changed is the addition of the `title` and `project` parameters.

```
routes.MapRoute(
  "Default",
  "{title}-{project}.aspx/{controller}/{action}/{id}", //Added .aspx to fix the problem
  new { action = "Index", id = "" }
);
```

Listing 4.4. Shows the URL map for URL strings that specifies the ID of project. The code comes from *Global.asax.cs.*

The URL mapping is not only a great feature to make user friendly URL strings, but also a way to specify extra variables that should be a part of the URL. In ASP.NET MVC it is easy to use these variables, as the URL mapping just calls the given action in the right controller, with the variables as parameters. This can be seen in listing 4.5 on the next page. The name of the input variables, in an action, needs to be the same as the names in the URL-map, as this is how ASP.NET MVC identifies the parameters.

```
public ActionResult Index(int project, int id)
{
...
```

Listing 4.5. Example of an action taking the `project` and `id` variables from the URL-mapping as parameters.

4.3.2.2 POST and GET Variables

This section explains what POST and GET variables are, and how they are handled in ASP.NET MVC. In web-based systems like the PMS, there are two types of variables: POST and GET. GET variables are data placed in the URL and is used by the URL-mapping in section 4.3.2.1 on page 60 [Kor03]. In the PMS, GET variables are primarily used in order to enable the user to copy a direct link to some specific element in the PMS. In contrast, POST variables are not a part of the URL, but can be sent e.g. by HTML forms when the user clicks *Submit*. In [Kor03] there is a simplification of the definition of GET and POST where:

> 'GET' is basically for just getting (retrieving) data whereas 'POST' may involve anything, like storing or updating data, or ordering a product, or sending E-mail.

In ASP.NET MVC it is also possible to get the POST variables from a request. This can be seen in the following example in listing 4.6. It shows how the action takes all the POST variables as parameters and at the same time, is converting the POST variables to a specific type, like `String` or `DateTime`.

```
1 [AcceptVerbs(HttpVerbs.Post)]
2 public ActionResult Create(String title,
3                            String dev_team_name,
4                            String mail,
5                            String room,
6                            DateTime start_time,
7                            DateTime deadline,
8                            String description)
```

Listing 4.6. Shows an action taking POST variables as parameters. This action is located in *Controllers/ProjectController.cs.*

To make an action listening for POST variables there is need for one line of code (line 1). An action that takes POST variables also takes GET variables - there is no GET variables in this example though. The opportunity of getting POST and GET variables is an important and necessary feature in web-based development, as it is the only way to get information from the user and thereby making the system interactive.

4.3.3 The Purpose of Choosing ASP.NET MVC

This section aims to make it clear why MVC and ASP.NET MVC have been chosen in this project.

The first reason is the possibility to develop after a certain type of architectural pattern. This makes it possible to learn how to turn a design into a good and flexible design. Furthermore, a solid structure is also an advantage when it comes to developing software in a group. This can be seen as the MVC pattern clearly defines where to place different parts of the program, making misconceptions and confusions more rare.

Another feature in the ASP.NET MVC is the possibility to execute Unit Tests and use Test-Driven Development (TDD). ASP.NET MVC includes the possibility of making tests in the program and integrate with the test tools in Microsoft Visual Studio 2008 [Cor09c]. According to test, this topic will be revisited in chapter 5 on page 83.

ASP.NET MVC supports C#. This might not be a permanent reason for selecting ASP.NET MVC in general, but since we have been offered lectures in learning the Object Oriented Paradigm by the use of C# through the OOP-course, it was even more reasonable to choose ASP.NET MVC.

Further, ASP.NET MVC is a good way to make use of object oriented programming, as the model component make use of objects. This was outlined in section 4.3.1.1 on page 53.

Finally ASP.NET MVC is very flexible and fits well with the design criterion; *flexibility*, described in section 3.1.1 on page 32.

4.4 ASP.NET

The Internet is a widely used communication media making it easy for everybody to look for and share information. As the demands on homepages on the Internet are rising, so are the demands on more advanced software for developing homepages. A relatively new technology is ASP.NET developed by Microsoft. ASP stands for Active Server Pages [Cor09a] and .NET is an abbreviation for .NET framework. ASP.NET supports developing websites, web applications and web services, and supports mainly two programming languages; Visual Basic and C# [Net09]. In this chapter a basic introduction to ASP.NET's page architecture will be given. The page architecture serves to separate presentation and functionality, and makes extensive use of design template known as master page to ensure a uniform appearance. These concepts are foundational to the new ASP.NET MVC framework.

4.4.1 Page Architecture

A normal ASP.NET page has the file extension `.aspx` for example `mypage.aspx`, which contains the presentation of the page, supporting normal HTML/XHTML and also ASP.NET server controllers. An ASP.NET server controller start with the `<asp:` and have

the attribute `runat="server"`, which indicates it is an ASP.NET server controller and should be compiled into HTML. The ID attribute is important to manipulate the controller. To manipulate the controllers ASP.NET supports two ways of doing it. One is to add in the top of the page `<script runat="server"></script>` and write the functionality inside, the other is by separating the functionality from the presentation in another file. This file must be named exactly as the `.aspx` page, but with the extension `.cs` if the language is C# or `.vb` if it is Visual Basic. This file is normally referred to as a code-behind file of the `.aspx` page. In this project we use another way to separate the functionality and presentation, by the MVC principles. Each `.aspx` page starts with a `<\%@ Page \%>` directive that contains all the information regarding the `.aspx` page. An example of a @page directive is showed in listing 4.7.

```
1 <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="
      PMS_dev._Default" %>
2 <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="
      System.Web.Mvc.ViewPage<PMS_dev.Models.task>" %>
```

Listing 4.7. Two example of a @page directive. Line 1, the actual @page directive from the **`default`**`.aspx` in the PMS. Line 2, a @page directive from a view in the PMS.

The attribute `Language` defines which programming language is being used. The attribute `MasterPageFile` contains, if the page uses a master page, the path to the master page, which will later be discussed in section 4.4.2. The attribute `CodeFile` contains the path to the code-behind file and the `Inherits` attribute in line 1 contains the name of the class in the code-behind file, where `PMS_dev` is the namespace the class belongs to. In line 2 the `Inherits` attribute defines the relationship between the application and the view, and can pass an object, which will be defined inside the `<>`.

ASP.NET also supports embedding code in the `.aspx` file with the `<% %>` block, which indicates that the code should be executed. An example of the use is shown in listing 4.8.

```
<%if (Roles.IsUserInRole("Developer")){ %>
  <% Html.RenderPartial("submenu"); %>
<%} %>
```

Listing 4.8. An example of embedding code in a `.aspx` page.

A `<% %>` block is mostly used for data binding in the PMS, especially when listing objects with the **`foreach`** loop, but also to restrict display of 'action' components, like buttons and links.

4.4.2 Master Page and Content Page

When designing a website and the same design is wanted on all the pages, copy-pasting the same code into all files is both tiresome and redundant. Instead it would be practical and time saving with a template of the design that would be loaded for every page on the website. In ASP.NET these are called master pages and content pages. The master page contains everything that will be repeated on all the website's pages, and then includes the content of the actual page from the content page. Master pages are defined with the file extension `.master` and instead of `<%@ Page %>` it uses `<%@ master %>`. Otherwise it behaves and follows the rules as another `.aspx` page. The way to include the content from content pages is to use a `ContentPlaceHolder` controller. An example of a `ContentPlaceHolder` is shown in listing 4.9.

```
<asp:contentplaceholder id="MainContent" runat="server" />
```

Listing 4.9. An example of a ASP.NET `ContentPlaceHolder` controller

The `ContentPlaceHolder` is then inserted in the master page where the content should be placed. A master page can contain many `ContentPlaceHolder` controllers, which holds the content of the content pages. The content page is recognized by the attribute `MasterPageFile` in the `<%@ Page %>`, which was showed in listing 4.7 on the preceding page and only contains content blocks that refer to the `ContentPlaceHolder` in the master page by the `contentplaceholder id` attribute. The content controller contains the page's content in between the start and end tag. The content can be plain text, HTML or an ASP.NET server controller. An example of a content block is shown in listing 4.10.

```
<asp:content ID="Content1" ContentPlaceholderID="MainContent" runat="server">
  ...Content...
</asp:Content>
```

Listing 4.10. An example of a Content controller

In listing 4.11 is an example on a complete master page with three content place holders and in listing 4.12 is a content page which includes the master page from listing 4.11.

```
 1  <%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD␣XHTML␣1.0␣Strict//EN" "http://www.w3.org/TR/xhtml1/
        DTD/xhtml1-strict.dtd">
 3  <html xmlns="http://www.w3.org/1999/xhtml">
 4  <head runat="server">
 5      <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
 6      ...
 7  </head>
 8
 9  <body>
10      <div class="page">
11          <div id="header">
12              ....
13              <asp:ContentPlaceHolder ID="submenuPlaceHolder" runat="server" />
14              <div id="main">
15                  <asp:ContentPlaceHolder ID="MainContent" runat="server" />
16              </div>
17          </div>
18      </div>
19  </body>
20  </html>
```

Listing 4.11. A segment of the PMS's `Site.Master`, containing the `ContentPlaceHolder`, `TitleContent` in line 6, `submenuPlaceHolder` in line 14, and `MainContent` in line 17.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="
    System.Web.Mvc.ViewPage<PMS_dev.Models.task>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
  ...
</asp:Content>

<asp:Content ID="SubMenu" ContentPlaceHolderID="submenuPlaceHolder" runat="server">
  ...
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  ...
</asp:Content>
```

Listing 4.12. A segment of the PMS's AddTask.aspx, showing the use of content page with content controllers [HKSU06].

In the PMS the use of standard ASP.NET has been at a minimum. The reason is that ASP.NET MVC uses a different approach, where ASP.NET server controller is replaced by Form and HTML helper. The use of master page and content page instead, is used through the PMS. The reason for this is that it works easily with MVC framework, and makes it a lot easier for the developers, as they do not have to spend time on placing the content in the right place, and copy the design in all the pages. This also makes it easier to change the design on the website.

4.5 Database Systems

This section will concern the considerations made about which type of data storage to apply in the PMS. Even though the system definition prescribes, that the PMS must use a RDBMS, this section will consider alternatives and argue why a RDBMS is a reasonable choice.

Data for websites can be stored in numerous ways. The technically simplest way is obviously to store everything as static HTML files. However, if updating and calculating data manually is not an option, data must be stored in a manner that allows automated processes to store and calculate data. Data can be stored in syntactically formed operating system files that, by the use of various executables, allows users as well as automated processes to view and manipulate these data. While storing data in files makes it possible to organize and manipulate data in a structured way, it brings major disadvantages:

Data redundancy and inconsistency. The lack of strict rules as of how to structure data in files combined with the fact that files may be accessed by numerous programmers writing applications in different programming languages, may make data structures and manipulation procedures extremely inconsistent. Data may also be duplicated in several files, possibly differently structured. Updating data (e.g. a user's telephone number) in one file, does not automatically mean it will be updated in other files, making the data not only redundant, costing extra storage space as well as processing, but also inconsistent.

Data access. Searching in files without any index or the like, makes it difficult to search for specific data within them. This is especially the case when combining multiple search criteria (e.g. searching for the amount of tasks (stored in one file) the user with first name 'Peter' and last name 'Nielsen' (stored in another file) has solved).

Data integrity. Consistency constraints within a file-based storing system must be handled entirely by the programs accessing the data within them. As multiple constraints are added over time, it becomes difficult to make sure that all programs accessing data complies with all data constraints.

Atomicity and concurrent access. The file storage engine may somehow fail, or allow multiple executables to manipulate data concurrently. Both may cause failure or incorrect results that are not solved or checked for automatically.

Security. As data access restrictions are difficult to apply directly to files, most access restrictions must be done by the executables extracting or manipulating the data, making it difficult to keep a consistent data access scheme across many executables.

Database systems consists of information stored interrelatedly, and programs that allows users to access and modify this information. These programs provides an abstract level of access and manipulation possibilities, that allows the Database System to handle many of the before mentioned problems automatically, hence managing both data constraints, data relations, assertions and authorization. [SKS06, Chapter 1, Page 4]

4.5.1 Relational Databases and SQL

A relational database stores data in tables with multiple uniquely named columns, and any number of rows with data corresponding to these columns. Each column defines what type of values can be stored in the associated rows, such as integers no larger than eight digits `INT(8)`, character-fields no larger than 255 characters `CHAR(255)` or `VARCHAR(255)`, binary data `BLOB` etc. table 4.1 gives an example of how some of the data types can be used.

firstname `VARCHAR(100)`	**lastname** `VARCHAR(255)`	**email** `VARCHAR(255)`	**age** `INT(3)`
John	Doe	john@doe.com	33
Peter	Griffin	pg@tv.com	25
Jack	Daniels	jd@alco.com	34

Table 4.1. A table illustrating the use of different data types.

The abstract access to data allows the Database Systems to provide a query language to access these data. Most relational databases, including MySQL which is used in this project, uses the Standard Query Language (SQL) to extract or manipulate data. A simple example of an SQL query would be to get the first and last name of the user with e-mail `john@doe.com` from table 4.1. This example can be seen in listing 4.13 on the next page:

```sql
SELECT firstname, lastname FROM users WHERE email = 'john@doe.com'
```

Listing 4.13. Simple SQL select statement.

The select statement in listing 4.13 would return table 4.2.

firstname	**lastname**
John	Doe

Table 4.2. The result from executing the SQL query in listing 4.13.

Rows can be uniquely identified by a **primary key**. In the above example, the `email` column could be used as primary key. Creating a primary key in a table ensures that rows are only valid, if all column values are unique and not null.

Multiple tables relates through foreign keys. An `inventory` table, for instance, might contain information of what cars users own. By adding foreign key relations, data redundancy and inconsistency is eliminated, as information occurs only once. The following tables shows an example of how foreign keys are used.

Table: employees		
id *(PK)*	**firstname**	**lastname**
1	John	Doe
2	Peter	Griffin
3	Jack	Daniels

Table 4.3. The employees table.

Table: inventory		
id *(PK)*	**owner** *(FK)*	**item**
1	2	Ford Ka
2	3	Mazda RX8
3	2	Lotus Elise

Table 4.4. The inventory table.

The default behavior is to disallow changing and deleting fields used as foreign keys in other tables (i.e. you cannot delete or update the id of the Peter Griffin tuple as tuples in the inventory table has foreign key constraints pointing to that user). This behavior can be changed by the use of cascading. By adding `ON UPDATE CASCADE` to a foreign key constraint, you allow editing of the parent key, as you define what will happen with the child element. If updating Peter Griffin's id to 4, all foreign keys pointing to this key will be updated (all Peter Griffin's cars will have owner id updated to 4). By adding `ON DELETE CASCADE` to a foreign key constraint, all data related to a given tuple will be deleted, if the parent is deleted. That is, if Peter Griffin is deleted from the `users` table, all referenced tuples in the `inventory` table will be deleted (all Peter Griffin's cars will be deleted). Foreign key and cascading is an effective way of ensuring no information is unreachable because of missing or broken relations [SKS06, Chapter 4, Page 130].

Whenever a specific action occurs within a database, one might wish to trigger a series of events. Triggers can be added to tables to achieve this. A trigger is code that is automatically executed upon `INSERT`, `UPDATE` or `DELETE` events. Common usage is logging changes or enforcing creation of required tables (e.g. creating a salary tuple in a different table whenever a new employee is added to an employees table.) [SKS06, Chapter 8, Page 330].

Commonly used series of events, or events that must always be fired in conjunction, can be expressed using stored procedures, that can be easily executed through SQL queries. Apart from simplifying complicated queries, procedures has the major advantage that they are atomic - if the procedure does not execute fully, or an unexpected result occurs, the entire procedure will roll back. This is an advantage to developing programs that extracts data, manipulates the data and submits it back to the database. Procedures guarantees that no one else has manipulated the data in the time it takes to manipulate them and resubmit them, resulting in invalid results [SKS06, Chapter 4, Page 146].

The decision of using a RDBMS, was already taken in the system definition, but as stated before, the RDBMS have at lot of advances over file-based data storing, of keeping it simple to store data and management data, so the logical choice was using a RDBMS. One RDBMS is MySQL, a free open source database, which is supported by most web-based development environments and servers on the internet. For this project, the MySQL database have been chosen as it is most commonly used for web-application and what the PMS host supports. So with that in mind, the design of the database can begin, now that it is known what the database can handle and the possibilities it provide.

4.6 Database Design

This section concerns the design of the database storing the PMS model. The topic of this section will mainly be to describe how the database has been created using the data-definition language (DDL), and further how statements like cascades and foreign key constraints have been used to ensure data consistency and data integrity.

DDL is a database language used to create database schemas and structures inside the database [SKS06, chapter 1, page 10]. In the following a segment of the DDL commands used to create the database schemas and the structures of the PMS's SQL database will be explained.

```sql
CREATE TABLE table_name
    (attribute_name domain_type(length) optional_integrity_contstraint,
     attribute_name domain_type(length) optional_integrity_contstraint,
     ...
     optional_integrity_constraint(attribute_name));
```

Listing 4.14. Simple SQL DDL command to create a table.

The SQL DDL command `CREATE TABLE` has obviously been used, as this is the basic command for creating tables. This has been done by using the syntax shown in listing 4.14 [Ass09b, Slide 4]. Using the DDL as outlined in listing 4.14 the tables in the PMS have been created.

```
1  CREATE TABLE project (
2    id int(10) NOT NULL auto_increment,
3    title varchar(255) NOT NULL,
4    description text,
5    start_time datetime NOT NULL,
6    deadline datetime NOT NULL,
7    dev_team_name varchar(255) NOT NULL,
8    room varchar(255) NOT NULL,
9    mail varchar(255) NOT NULL,
10   PRIMARY KEY  (id)
11 ) ENGINE=InnoDB;
```

Listing 4.15. SQL DDL commands used to create the `project` table in the PMS.

The code-example in listing 4.15 shows how the `project` table has been created. The table is responsible for storing data about each project managed by the PMS, and includes the following attributes; `id`, `title`, `description`, `start_time`, `deadline`, `dev_team_name`, `room`, and `mail` from line 2 - 9. At line 10, the `id` attribute is set as the primary key, meaning that this is the key to uniquely identify a project within a relation. The `id` is a simple auto increasing integer, as easily derived from line 2 of listing 4.15. This means that whenever a project is created it will automatically be given a unique id (within the limits of the integer type).

To create relationship between two tables a foreign key constraint has to be added. This has been used consistently in most of the tables in the PMS to ensure that the model from section 3.3 on page 39 has been correct associated. As the model in the PMS application converts the relational data to in-memory data, applying foreign keys was very important, as LINQ / ADO.NET otherwise would not catch, how data should be related in the model. The only table, besides the user related ones, without foreign keys is the `project` table, as this is used as a parent to all other tables, for instance the iteration table, as shown in listing 4.16.

```
CREATE TABLE iteration(
    id int(10) NOT NULL auto_increment,
    ...
    project_id int(10)  NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (project_id) REFERENCES project(id) ON DELETE CASCADE ON UPDATE CASCADE
)ENGINE=InnoDB
```

Listing 4.16. SQL DDL commands used to create a foreign key to the `project` from the `iteration`.

An important thing to notice about the use of foreign keys constraints is the additional use of `CASCADE`. By using this in combination with the foreign key constraint, we have integrity constraints, which ensure that a change to the database always maintains data consistency [Ass09a, Slide 6]. The reason why we are using these integrity constraints is that it enables us to control the relation between the data stored in the database and make it correspond to how the model has been designed. In figure 4.14 on the following page, the use of `CASCADE` has been illustrated.

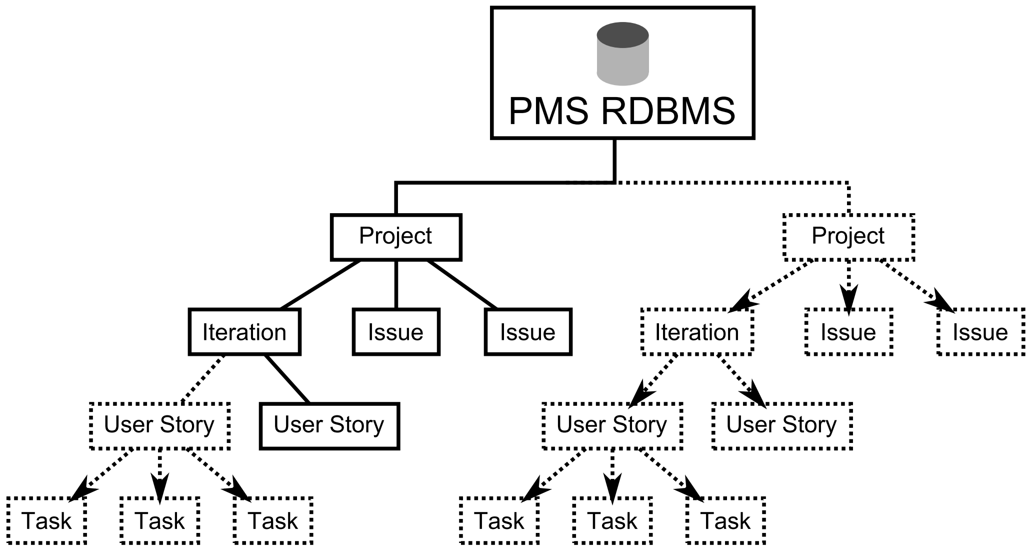


Figure 4.14. Illustration showing how cascades have been advantaged in the PMS RDBMS.

As can be seen in 4.14, one of the project branches has been deleted (the dotted lines shows this). This results in a cascade deletion of all related data to this project. This is caused due to the use of line 6 in listing 4.16 on the preceding page. Here, `ON DELETE CASCADE` has been added to the foreign key constraint, meaning that if the project is deleted, the iteration must be deleted too. This construction has been used elsewhere in the PMS, as the design class digram in figure 3.3 on page 39 has prescribed that specific classes needs to be associated with other classes. This is also seen in figure 4.14, where a user story is deleted in the left project branch. Here, the tasks related to the user story has a foreign key constraint pointing at the specific user story id. Combined with the `ON DELETE CASCADE` statement, this is making the tasks being deleted together with the user story.

4.7 Burn Down Chart

In an agile development methodology, burn down charts are used to give a simple overview of the work done. This gives developers an idea of how they are doing, and how far they are behind or ahead of schedule.

4.7.1 Burn Down Chart Overview

There are many different types of burn down charts. Some are simple, and some are more advanced, depending on which plot is being displayed. The x-axis in a burn down chart normally represents the time or date. The y-axis represents the user stories or work left. By plotting the amount of user stories at day one, and gradually removing solved user stories on a daily basis, the chart will display the work history, and hence make it easy to derive how healthy a team's work pace is. The ideal work pace is represented by a straight line from day one, at total work, to deadline date, at zero. If the plot which shows how much work has been done each day follows this line, the work is distributed evenly over each day, which classifies a healthy work pace.

4.7.2 The PMS Burn Down Charts

In the PMS two different burn down charts has been developed; one concerns user stories, the other concerns 'effort', or man hours. The charts extract information about user stories from the database.

Both burn down charts are able to reflect either the entire project or the current iteration. This means that there is a total of four different burn down charts in the PMS.

4.7.2.1 Burn Down Chart - User Stories

This section concerns the burn down chart depicting the implementation progress based on the number of user stories left. The rather simple burn down chart can be seen in figure 4.15.

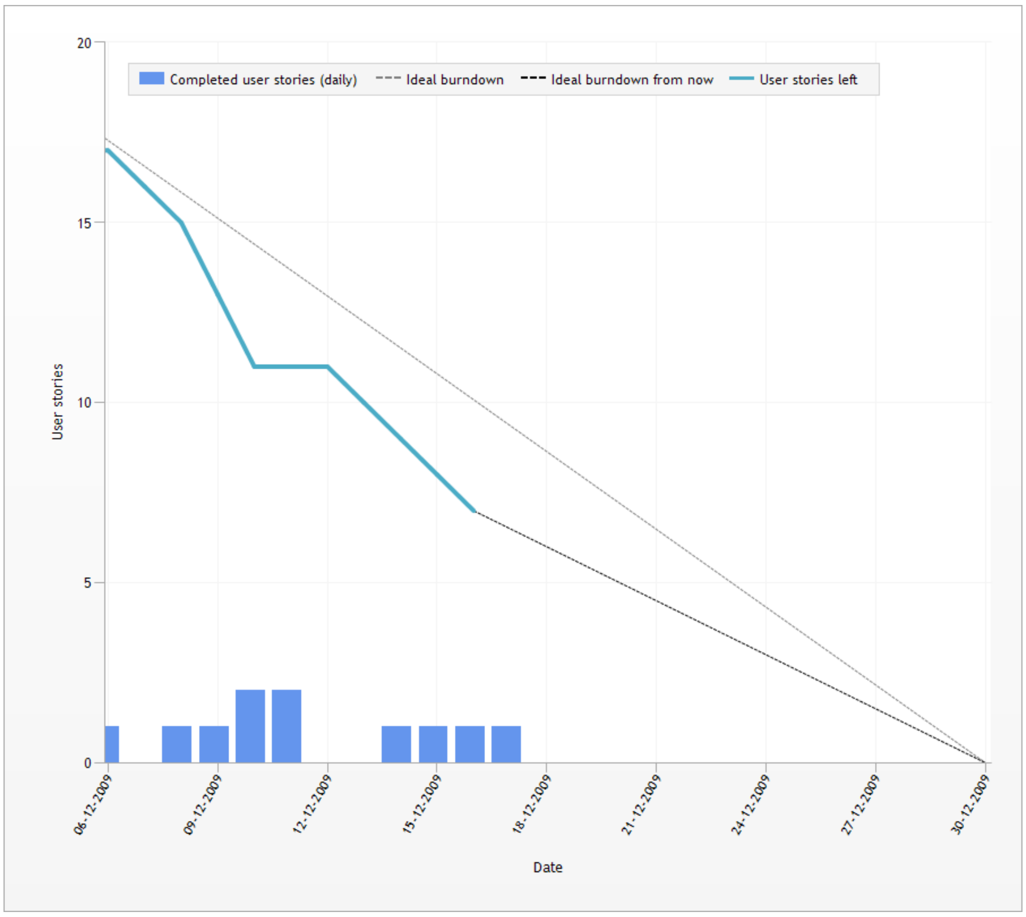


Figure 4.15. User story burn down chart for a project.

The chart consists of the following data series:

User stories left shows how many user stories are left, and was left at all past days (as described in section 4.7.1).

Ideal burn down is a linear line from the amount of user stories at project start to zero at project end, showing the ideal work pace.

Ideal burn down from now is essentially the same as the above line, except its starting point is plotted at the amount of user stories left 'today'. This gives the developers a hint to how fast they will need to work from now to meet the deadline, as well as the possibly of deriving how many user stories should be solved in future iterations.

Completed user stories (daily) displays how many user stories has been solved any given day.

4.7.2.2 Burn Down Chart - Effort Left

The effort chart is more complex than the user story chart. By plotting effort on the y-axis instead of simply the number of user stories, developers are able to see how many hours of work remains before the project or iteration is finished. This chart can be seen in figure 4.16.

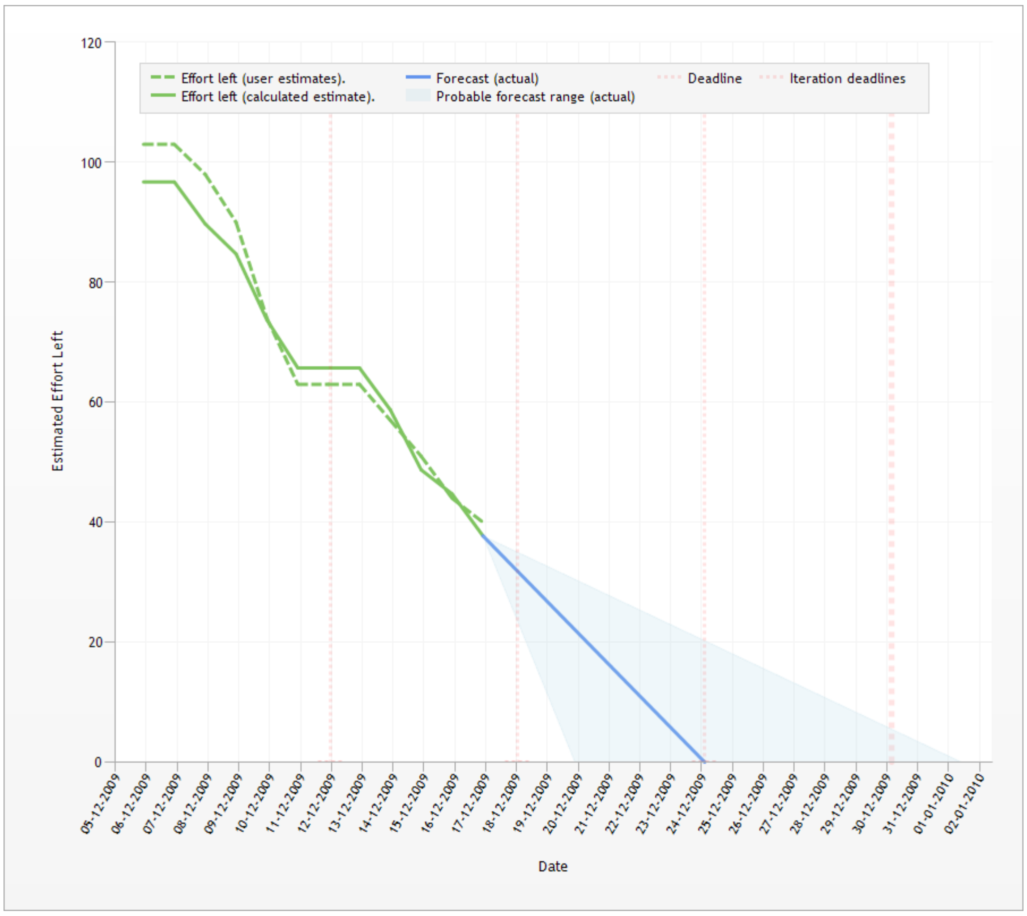


Figure 4.16. Effort burn down chart for a project.

The chart consists of the following series:

Effort left (user estimates) is a plot of how the users' estimates for user stories are being burned throughout the project or iteration.

Effort left (calculated estimates) is a plot of how the actual effort for implemented user stories is being subtracted from a calculated estimate time based on the relation between the users' estimate and the actual implement time. In other words, this series describe how the PMS predicts the effort left.

Forecast shows the most likely future progress based on the current speed ('effort per day').

Probable forecast range shows the same prediction as the above series, but calculates the predicted progress by using the users' best and worst estimates as basis.

Deadline is a self-explanatory series, which simply shows the deadline of a project or iteration.

Iteration deadlines shows the deadlines specified for each iteration.

Showing both effort left based on user estimates and the effort left based on the calculated estimates, makes it possible to analyze how well a group is estimating. The deviation between these two series can also be used to explain why the probable forecast might be narrow or wide.

4.7.3 Implementation of the PMS Charts

This section will give an insight into how the PMS charts have been implemented, particularly the more complex parts concerning the forecast. As most of the series are a mere representations of data, only the forecast series will be described in detail.

4.7.3.1 Microsoft Chart Controls

The charts in the PMS are drawn using the Microsoft Chart Controls. To be able to use the Microsoft Chart Controls API, the .NET Framework version 3.5 SP1 is required.

When working with the API (see listing 4.17 on the next page), charts are created as objects on to which series and labels are added.

```
Chart chart = new Chart();                    //Create chart
chart.Width = width;                          //Set chart width
chart.Height = height;                        //Set chart height
Series series = new Series();                 //Create new series
series.ChartType = SeriesChartType.Line;      //Make the series a line
series.Name = "Series␣name";
series.Points.Add(new DataPoint(4, 2));       //Add point to series
series.Points.Add(new DataPoint(7, 6));       //Add point to series
chart.Series.Add(series);                     //Add series to chart
chart.Legends.Add(new Legend(series.Name)); //Add legend
```

Listing 4.17. Shows how to modify the `Chart` and `Series` objects.

As four charts are created, some of which contains variations of the same series, a static MVC service has been been developed to ease the creation of charts and avoid redundant copies of code. A general structure of this can be seen in figure 4.17.

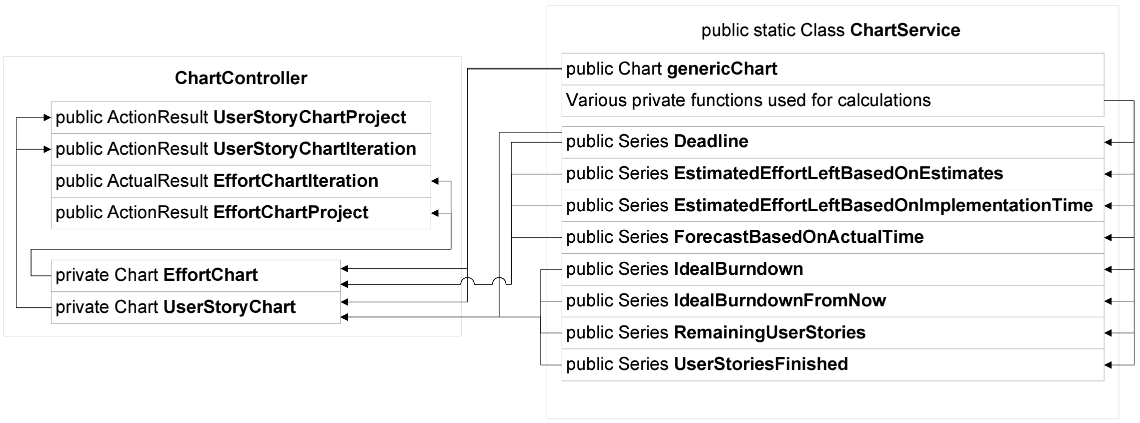

ChartController
public ActionResult UserStoryChartProject
public ActionResult UserStoryChartIteration
public ActualResult EffortChartIteration
public ActionResult EffortChartProject
private Chart EffortChart
private Chart UserStoryChart
public static Class ChartService
public Chart genericChart
Various private functions used for calculations
public Series Deadline
public Series EstimatedEffortLeftBasedOnEstimates
public Series EstimatedEffortLeftBasedOnImplementationTime
public Series ForecastBasedOnActualTime
public Series IdealBurndown
public Series IdealBurndownFromNow
public Series RemainingUserStories
public Series UserStoriesFinished

Figure 4.17. The ChartController creates a generic Chart (contains default values defining layout). Based on variables from the URL (URL mapping) the ChartController requests either an effort or user story functions. The start day and end date passed to these private functions defines whether it shows the entire project or an iteration. These two private functions adds the appropriate Series to the Chart by requesting them from the ChartService.

The controller does not rely on a view in normal sense. The controller is accessed through four different URL maps:

```
http://uni.mangafan.dk/[Projectname]-[projectid]/EffortChartProject
http://uni.mangafan.dk/[Projectname]-[projectid]/UserStoryChartProject
http://uni.mangafan.dk/[Projectname]-[projectid]/UserStoryChartIteration/[iterationid]
http://uni.mangafan.dk/[Projectname]-[projectid]/EffortChartIteration/[iterationid]
```

The ActionResult returned from the controller is the chart formatted as a `.png` image. This way, the charts can be accessed, not through a regular view, but directly by invoking it through a HTML `<img>` tag which can be embedded in other views, such as:

```
<img src="http://www.site.com/myProject-4/EffortChartIteration/12"/>
```

4.7.3.2 Forecast Algorithm

The forecast consists of the forecast line and the probable forecast range. These are generated from three different calculations: The most likely end date and the end date in best and worst case.

The most likely end date (the forecast line) consists of two coordinates (x_1, y_1) to (x_2, y_2):

x_1 is the date today (`DateTime.Now.Date.ToOADate()`);

y_1 is the calculated effort left. This is calculated by `effortTotal - effortDone`. `effortDone` is the summed 'actual effort' of all finished user stories. `effortTotal` sums `effortDone` and a calculated estimate of the effort required to implement the remaining user stories.

x_2 is found as the calculated remaining effort divided by the speed ('effort per day') value of user stories that have been implemented. This is added to the value of x_1 to get the date at which the project is most likely to be finished.

y_2 is zero (no more effort required when everything is implemented).

The left- and rightmost end dates of the probable forecast range are calculated slightly different. The calculated progress is now based on the users' best and worst estimates.

Estimated time	5	6	4	3
Actual time	15	18	12	9

Table 4.5. Example data

Estimated time	5	6	4	3
Actual time	3	4	6	5

Table 4.6. Example data

Even though user stories are always finished three times later than estimated in table 4.5, the estimates are very consistent (always 3 times too small). This means that no forecast range will be visible, as the relationship between estimate and actual time is constant.

In table 4.6 the estimates may be closer to the actual implementation time, but they are very inconsistent compared to the implementation time. This is reflected in the now visible probable forecast range, by multiplying the most-likely prediction with the worst and best guess deviation compared to the average variation. This computation can be seen in listing 4.18.

```
 1 //foreach finished user story {
 2   double usEstActRelation = (double)us.actual_implement_time.Value / (double)us.
         estimate_implement_time.Value;
 3   if (usEstActRelation > maxRelation)
 4     maxRelation = usEstActRelation;
 5   if (usEstActRelation < minRelation)
 6     minRelation = usEstActRelation;
 7 //} (...)
 8 //note: averageRelation = (sumEstimates / sumActualImplementationTimes)
 9 double minRelativeRelation = minRelation / averageRelation;
10 double maxRelativeRelation = maxRelation / averageRelation;
```

Listing 4.18. Private function `MinMaxEstActRelativeRelation` in `ChartService`. Called by `ForecastBasedOnActualTime` in `ChartService`.

These 'relative relations' (minRelativeRelation and maxRelativeRelation seen in line 9-10), are in listing 4.19 multiplied with the most likely amount of days left, and then added to today, which gives us the estimates of when the project or iteration will end if we guess as positively and negatively as we have done in the past:

```
//Find out how much our estimates are vary from our actual implementation times.
double[] deviationEstimateAndActualTime = MinMaxEstActRelativeRelation(USList); //min,
    max
//Find out, in best and worst case, how many days are needed to solve the remaining user
     stories
//note: effortLeft = calculated estimate of the remaining effort.
double bestCase = effortLeft * deviationEstimateAndActualTime[0]; // [effort]
double worstCase = effortLeft * deviationEstimateAndActualTime[1]; // [effort]

//Based on our implementation speed so far (effort per day), calculate how many days it
    will take to solve the remaining user stories, in best and worst case
//note: MostLikelyFinishGuess = (effortLeft * 1) / speed
double DaysLateFinishGuess  = worstCase / speed; // [days]
double DaysEarlyFinishGuess = bestCase / speed; // [days]
```

Listing 4.19. Public function `ForecastBasedOnActualTime` in `ChartService`. Called by `EffortChart` in `ChartController`

4.8 Security

To insure the safety of the data in the projects managed by the PMS, some precautionary measures have been taken. This includes; a user login, project access granting, and user roles. These precautionary measures will be described in this section.

4.8.1 Overview

This subsection will give a short introduction to each of the precautionary security measures. This is done to substantiate the prioritization of the design criterion *Secure* introduced in section 3.1.2 on page 33.

The user login requires the user to register, before creating or applying for projects.
This is done to ensure the identity of the user. Furthermore, the minimum password

length has been set to ten, combined with the distinction of using lower and upper case letters. In this way, it becomes more complicated to guess a particular password, which makes the user login an easy way to improve the security of the system.

User roles include two types of roles, both a 'Developer', and a 'Supervisor'. The developer has unlimited access whereas the supervisor's access is limited to reviewing; such as monitoring the development progress, commenting on iterations, and reporting issues.

The project access granting ensures that only users with an association to a given project are able to enter it. When a user creates a new project he or she will automatically be granted access to it. Other users are then able to apply for access to the project. These appliances are evaluated by already granted users, and they can choose to either grant or reject the pending users. Granted users are also able to exclude other already granted users from the project, this means that the security inside projects is based on trust between the granted users.

4.8.2 Use of ASP.NET MVC Built-In Security

This subsection will describe how some of the built-in functionalities in ASP.NET MVC have been used, and in some case modified, to make them meet our needs.

4.8.2.1 Authorization

The following will describe how we have applied the built-in ASP.NET MVC login and user authorization. This feature makes handling users an easier task. Combined with the Visual Studio ASP.NET MVC functionality for user login, a lot of the work regarding user handling is done already when a new MVC project has been created. By using the Visual Studio 2008 Web Site Administration Tool, user roles are easily implemented and administered too. The authorization of actions are accomplished by decorating the actions, which requires user login. This is done by using the attribute shown in line 8 in listing 4.20.

```
 1 using System.Web.Mvc;
 2
 3 namespace MvcApplication.Controllers
 4 {
 5   [Authorize]
 6   public class Controller : Controller
 7   {
 8     [Authorize]
 9     public ActionResult ControllerFunction()
10     {
11       return View();
12     }
13     [Authorize (Roles = 'role_name')]
14     public ActionResult RoleRestrictedControllerFunction()
15     {
16       return View();
17     }
18   }
19 }
```

Listing 4.20. Simple MVC controller with authorize functionality.

The attribute can also be added to the whole controller, by placing the attribute on the class, as showed in listing 4.20 line 5. Another feature which is displayed in this listing, is role restriction. This can be seen at line 13, where the `role_name`, specifies which role is needed to access the particular action [Cor09d].

4.8.2.2 Login and User Handling

When using the standard login-functionalities as described in section 4.8.2.1, a `my_aspnet_Users` table is automatically generated in the database. This table contains the most basic login information, such as user name and password. Regarding the PMS, this is insufficient as we need to store additional information about the user, like full name, E-mail, phone number etc. As described in the model component in section 3.3 on page 39, this has been solved by creating a separate `additional_user_information` table. This table is related to the users stored in the `my_aspnet_Users` table, by the use of a foreign key. Using a structure like this enables us to relate more information to each particular user.

This solution was chosen, rather than modifying the `my_aspnet_Users` system-generated table, as we did not want to risk, that altering the table would impair the system in some way. Extending the foreign key referential integrity constraint between the `my_aspnet_Users` table and the `additional_user_information` table, by the `ON DELETE CASCADE` clause, ensures that additional user informations is only stored if the user exists in the system.

As the PMS is designed to always require the user to be logged in, adding user authentication to the system was crucial. There were two different approaches to accomplish this. The first approach concerned adding the `[Authorize]` attribute to every controller, but as the authorization check has to be made every time a user is using the system, this would result in duplicated code that is difficult to maintain. Further, each of us would have a great responsibility of remembering to place the `[Authorize]` attribute whenever it was needed. Instead we applied the restriction by editing the `web.config` file. In this file, listing 4.21 has been added. By using this solution, we reduced duplicated code, and removed the responsibility away from each of us to remember placing the `[Authorize]` attribute when needed.

```
1  <location path="Default.aspx">
2    <system.web>
3      <authorization>
4        <deny users="?" />
5      </authorization>
6    </system.web>
7  </location>
8  <location path="Account">
9    <system.web>
10     <authorization>
11       <allow users="*" />
12     </authorization>
13   </system.web>
14 </location>
```

Listing 4.21. Shows how the PMS is only accepting logged in users. This code is located in the *web.config* file.

In the first part of listing 4.21 (line 1 - 7), access is denied to the `Default.aspx` file if the user is not logged in. Because the `Default.aspx` is the file which handles all requests in the system, this leaves the users without access to anything. This is compromised by the next part of listing 4.21 (line 8 - 14), which adds an exception to the previous statement, and allows any user to access the `AccountController` (and all its actions). This enables new users to register and already registered users to log in, and thereby gain access to the rest of the system.

4.8.2.3 Roles

User authorization. One of the things that ASP.NET MVC has adapted from ASP.NET is the user authorization. This element is in the PMS used to prevent the supervisors taking the *Developer* role. That is, the supervisors must only be able to review a project by seeing statistics, error reporting and placing comments on different project elements such as iterations. To keep this separation, the `[Authorize (Roles = 'role_name')]` attribute has been used. The use of this attribute is similar to our first approach of applying authorization to the system. The attribute is placed above a particular action or controller making it possible to specify which roles are authorized to execute specific actions. Placing `[Authorize (Roles = 'Developer')]` above an action would imply that only users of type *Developer* would be authorized to use the particular action. In this way, supervisors can be restricted access to specific elements in the PMS.

Project Restriction. In ASP.NET MVC there is no built-in way to restrict registered user's access to projects in the system. To realize this, the code in listing 4.22 on the next page has been used. The code extracts the project id from the URL, and then checks to see if the user has access to the given project. If not, he or she will be directed to the project list. By placing this code in a content place holder in the Site.Master file, it will be loaded with every view (unless it is overwritten), and thus restrict any project to only allow its members to enter it.

```
<asp:ContentPlaceHolder ID="ProjectSelectCheck" runat="server">
  <% String project = URLResolver.GetRouteValue(ViewContext, "project");
    if (project == null || Roles.IsUserInRole(project) == false)
  {
    Response.Redirect("~/pms.aspx/Project");
  } %>
</asp:ContentPlaceHolder>
```

Listing 4.22. The check which insures that a user has access to the project he is requesting.

4.8.3 Known Security Issues

In an attempt to track security issues in the PMS, it has been discovered that the project restriction described in section 4.8.2.3 is vulnerable to manipulated URLs. The problem occurs when the user requests a page - then an action in the appropriate controller is called. The action invokes a view which then includes the `Site.Master` page containing the project restriction check. This means that if a malicious user sends a request using a modified URL, the action in the controller will be executed with the given GET variables, and the user will be redirected to the project list page.

By replacing the `name` element in this URL: `http://[site_name]/[project_name]-[project_id].aspx/[Controller_name]/[Controller_action_name]/[GET_variable]` it will be possible for any registered user to exploit the system without being member of any particular project. An exploitation could be removing all users from a project for instance. To accomplish this 'hack', the URL must be manipulated using the elements described in table 4.7.

URL-element	Description	What to insert
`site_name`	Server address / domain	`uni.mangafan.dk`
`project_name`	(Non-essential)	
`project_id`	Id of project to attack	`7`
`Controller_name`	Controller name	`Admin`
`Controller_action_name`	Target action	`Remove`
`GET_variable`	GET-variable (id of user to remove)	`18`

Table 4.7. The URL elements along with the values necessary to exploit the 'remove users' vulnerability.

The vulnerability can of course also be exploited to execute all controller actions that does not require a view. Further it is possible to exploit actions requiring POST-variables, either by faking the form data using e.g. the `webClient.UploadValues` function from the `System.net` namespace in C# or by making a simple HTML form.

A way to fix the vulnerability would be to move the project-restriction check inside the controllers, as no actions then will be executed if the user is not authorized to do so. This is a rather simple way to fix the issue, but as it will require some lines of duplicated code and as the *Security* design criterion in section 3.1.2 on page 33 has not been rated 'very important' we have discarded the solution.

Test 5

This chapter will describe the process of testing in general as well as based on the PMS. As described in subsection 3.1.1 on page 32 concerning the design conditions, the members of this project had little or no experience with web development beforehand. This resulted in a development with more time used on trial and error, and less time using the test first development method. There were specific problems affiliated with testing the components on the website itself, so all the tests which had been done, were on static functions concerning for instance the plotting of charts.

5.1 Approaches

5.1.1 Black and White Box Testing

Black and white box testing is two different kinds of testing methods. Other and more descriptive terms for these methods are respectively 'behavioral' and 'structural' testing. Black box testing is usually connected with testing functional requirements as it is discouraged to 'look inside' the box/code. This means that the testing is done by giving the function to be tested some input, and then evaluating the output - without ensuring that all code paths has been visited. On the contrary white box testing is focusing on testing the internal code paths in the function or class. Depending on the strictness, different test coverage is required. Unfortunately a test with complete code coverage would most likely result in an unmanageable number of tests [Nør09].

As outlined in [Bor02] the following test coverage is common:

- All statements must be executed at least once.
- All branches are to be executed at least once (an 'if' statement will cause a branching, even if it does not have an else clause).
- All individual conditions out of larger conditionals are to be tested (a conditional statement like `(a == b && c == d)` will require four test cases where to cover the outcomes. One where `a == b` and `c == d` and another where `a == b` and `c != d` and one where `a != b` and `c == d`, and a final one where `a != b` and `c != d`).
- All possible paths are to be tested (not practical for larger classes or functions).

An example of black and white box testing can easily be explained by moving the context to the world of cars. A black box test could be stated as the following: A car is required to be able to start, drive two kilometers, and then stop. The test would then be to drive two kilometers, and the test is then considered successful if the car has moved the 2 kilometers. If a car was to be tested using the white box method, a different approach would be used; all the different parts of the car would have to be tested separately, the cylinders, carburetor, tires, etc. If all these are working the test is considered successful.

5.1.2 Unit Testing

To confirm or reconfirm that software is working as intended, unit testing is a useful tool. Unit testing is done by testing the units[1] in an application separately. By doing so, the process of finding errors will be eased. [BSM07] Unit testing is a highly recommended practice in the world of software development, as it benefits the development. It benefits the developers, first of all because it forces them to keep the functions and classes testable, and thereby more simple and palpable, and secondly, because it clearly shows if an error is present and where this error is. It benefits the possible future developers, because they will be able to refactor, and easily see if a function still producing the wanted output. And finally, of course, it aids the quality of the code as it has been tested thoroughly [Cra07].

5.1.3 Test Driven Development

XP uses a test-first development method called test driven development or TDD, where planning and implementation of the test is done before the development has started. The tests are implemented as unit tests (described in the previous section), meaning that every unit of the development must pass a number of tests before it can be declared implemented and solved. In that way it is easy to validate whether a function or class is working as intended. This is both practical when developing from scratch as well as when refactoring code. First, when working from scratch it will be clear when the code is working, even though it might not meet any coding conventions. Second, when refactoring it ensures that the developer is not unknowingly breaking the code. It is especially useful when combined with a version control system as this means the code will always be revertible to a state where the unit test passes [Bec02].

5.2 Testing of Static Functions

This section will concern the testing of two static functions in the PMS. Both of these functions are related to the charts, one is concerned with the actual data, and one with the drawing of the chart. The tests have been done using the built-in testing tool in Visual Studio. This made the initial creation of the tests easy as it just requires a simple right click on the wished function to test. There are no specific reason for the choice of the two functions, other than they are both rather simple, and easily explained, though not so simple that they cannot contain errors.

5.2.1 Testing of SolveY

This function is used when drawing the chart, specifically the forecast range chart. To explain why this function is needed, the way the 'range' chart type has to be explained. The data series to draw a range is a little different from the one to draw for instance a line, as the range needs two y values whereas the line only needs one. This is to get the

[1]Units - The functions and classes of a system

range to cover a span on the y axis (between the two y values). This is where the solve y function comes in. There are three points in the forecast range each with three values (x, y_a, y_b). Two of these points are known, the first and last, whereas the third (the one in the middle) only has the x and y_a value. One thing which is known though, is that the unknown y_b value is on the line between the first and last point. The problem is visualized in figure 5.1 To find this value, the function must create the linear equation ($y = ax + b$) from the two points' y values, and then insert the known x value into the equation. This is as seen in listing 5.1 exactly what it does.

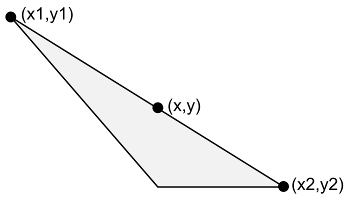


Figure 5.1. x1, y1, x2, y2 and x is known. The solveY function calculates the y-value necessary to generate a straigth line from (x1, y1) to (x2, y2)

```
private static double SolveY(double x1, double y1, double x2, double y2, double forX)
{
  double a = (y2 - y1) / (x2 - x1);
  double b = y1 - (a * x1);
  return (a * forX + b);
}
```

Listing 5.1. Shows the function to be tested.

To test this function the valid input data has to be found. The obvious invalid data would be if the x_1 and x_2 values are equal. This is due to the fact that the function would then cause a divided by zero exception. In the PMS it is also known that the function will never receive an y_1 smaller than the y_2 value, and the only time both y values could be equal would be if they both were zero. Therefore those data has been excluded from the test, whereas other more likely input has been tested in the test in listing 5.2.

```
[TestMethod()]
[DeploymentItem("PMS-dev.dll")]
public void SolveYTest()
  {
  //Test data
  double[][] testData = new double[][]{
    //     expected  ,x1, y1, x2, y2,  x
    new double[] {5   , 0, 10, 10,  0,  5},
    new double[] {7.5, 0, 15, 10,  0,  5},
    //both y values zero
    new double[] {0   , 0,  0,  9,  0,  6},
    //scaleablity
    new double[] {0.4, 0,0.8,0.4,  0,0.2},
    new double[] {  4, 0,  8,  4,  0,  2},
    new double[] { 40, 0, 80, 40,  0, 20},
    new double[] {400, 0,800,400,  0,200}
  };
  foreach (double[] test in testData)
  {
    double expected = test[0];
21      double actual = ChartService_Accessor.SolveY(test[1], test[2], test[3], test[4],
            test[5]);
22      Assert.AreEqual(expected, actual);
23    }
24  }
```

Listing 5.2. Simple test of the likely input.

In line 8 and 9 some standard input has been inserted, where the result comes in both integer value, and a value with decimal points. In line 11 the lower bound of what input the function might receive is inserted. Here both y values are set to zero, creating the line $y = 0$, which therefore obviously will give 0 for any x value. The next test data (line 13 - 16) is concerning scalability, here the test shows that the function handled at least input from 10^{-1} to 10^{2}. If the function passes this it suggests that it is scalable.

5.2.2 Testing of TotalEffortBasedOnEstimates

This function is used to calculate how much work, or effort, the project contains in total. It is, as the name implies, based on the estimate inserted by a user when assigning user stories to iterations. The function goes through the complete list of user stories, which it takes as input. This sum is then used to calculate an average, and this average is then multiplied with the number of user stories not estimated. The likely data for this function has been listed below here.

- An empty list.
- A list of user stories without estimates.
- A list of user stories where all user stories has estimates.
- A list where some user stories have estimates and some does not.

It should not matter if the list of user stories contains finished user stories or not. Based on this information the test in listing 5.3 has been made.

```
1   [TestMethod()]
2   [DeploymentItem("PMS-dev.dll")]
3   public void TotalEffortBasedOnEstimatesTest()
4   {
5     EntityCollection<user_story>[] testData = new EntityCollection<user_story>[]{
6       //Empty list
7       new EntityCollection<user_story>(),
8       //List without estimates
9       new EntityCollection<user_story>{
10        UserStoryGenerator(),
11        UserStoryGenerator(),
12        UserStoryGenerator()
13      },
14      //List only with estimates
15      new EntityCollection<user_story>{
16        UserStoryGenerator(3),
17        UserStoryGenerator(6),
18        UserStoryGenerator(9)
19      },
20      //A mixed list with user stories with and without estimates
21      new EntityCollection<user_story>{
22        UserStoryGenerator(3),
23        UserStoryGenerator(6),
24        UserStoryGenerator(),
25        UserStoryGenerator()
26      }
27    };
28    double[] expected = new double[] {0,0,18,18};
29
30    for (int i = 0; i < testData.Length; i++)
31    {
32      double actual = ChartService_Accessor.TotalEffortBasedOnEstimates(testData[i]);
33      Assert.AreEqual(expected[i], actual);
34    }
35    //Two lists, one with finished user stories and one without, but otherwise equal
36    EntityCollection<user_story> USList1 = new EntityCollection<user_story>{
37      UserStoryGenerator(3),
38      UserStoryGenerator(6)
39    };
40    EntityCollection<user_story> USList2 = new EntityCollection<user_story>{
41      UserStoryGenerator(3, 5, DateTime.Now.AddDays(-1) ),
42      UserStoryGenerator(6, 4, DateTime.Now.AddDays(-2) ),
43    };
44    //These should give equal results
45    double actual1 = ChartService_Accessor.TotalEffortBasedOnEstimates(USList1);
46    double actual2 = ChartService_Accessor.TotalEffortBasedOnEstimates(USList2);
47    Assert.AreEqual(actual1, actual2);
48  }
```

Listing 5.3. Simple test of the likely scenarios for the TotalEffortBasedOnEstimates function

The `UserStoryGenerator` first introduced at line 10 in listing 5.3, is a simple function to create test user stories easily. The function has three overloads. Without parameters the function will create an empty user story. With one parameter, it will create an estimated user story. And with three it will create a finished user story with the estimate as the first, the actual time as the second, and the finish date as the last.

5.3 Stress Testing the PMS

While the PMS is not built with large scalability in mind, and mostly developed for use by few people, stress testing the system still provides valuable information about where code could be optimized. And if nothing else, a hint to at what level of usage problems start to occur.

Three different stress tests have been made. The first is a test that communicates directly with the MySQL database, adding projects with large numbers of valid data (iterations, user stories etc.). The second is an "emulated usage test" of the PMS itself, emulating a large number of users using the PMS (viewing pages and images etc.). The third is a test of how well the PMS works when large amounts of data is present in the database.

5.3.1 Testing the Database

The test was developed in PHP, which was chosen because of the rapid development speed possible when code effectiveness is of less importance, and clear code structure is irrelevant.

A single connection is created to the database, which is used throughout the script. Various random parameters are then generated, before generating a project.

- Project start and end time (in a range of 7 to 51 days)
- Amount of iterations (4 to 10)
- Amount of user stories in each iteration (2 to 8)
- Amount of tasks in each user story (3 and 30)
- Amount of issues in each project (20 to 200)
- Amount of comments in each iteration, user story and task (5 to 8)

The script then executes a large amount of queries (as many as 5700) to create the project with the appropriate data. The generated data is valid and can be viewed in the PMS the same way as for normal projects. The results of the test are seen in table 5.1.

The exact same test was done, only 30 copies of the script were executed at the same time. The result of this test can be seen in table 5.2.

	Run 1	**Run 2**	**Average**
Start time	20:14:46	20:08:37	20:07:20
End time	20:15:09	20:09:54	20:07:52
Queries	1557	5656	3551
Queries per second	68	73	111

Table 5.1. Results of running the test one at a time. The smallest and largest test (in terms of amount of queries) are depicted together with the average of all seven runs.

	Run 1	**Run 2**	**Average**
Start time	19:53:18	19:53:18	19:53:18
End time	19:54:54	19:58:45	19:57:48
Queries	942	5871	3860
Queries per second	10	18	14

Table 5.2. Results of running the 30 tests at the same time. The smallest and largest test (in terms of amount of queries) are depicted together with the average of all 30 runs.

The results show a 87% slowdown when executing 30 connections at the same time compared to one at a time. The slowdown may be due to client throughput limitations as well as extended server load. What is interesting, however, is not how much slower the database server may be at executing queries when the server is busy, as this test does not simulate actual use of the PMS. The interesting result is that 30 concurrent connections can be made to the database, executing 14 queries per second, without any errors occuring or waiting time being excessively long. While the amount of connections (30) was limited due to considerations as to how many connections a single host could create to the database without being the limiting factor, it still executes a hugely excessive amount of queries compared to what real users would be able to. This shows that the database itself is unlikely to cause problems in case of increased usage of the PMS.

5.3.2 Testing the PMS

The database test only shows that the actual database is geared towards heavy usage. The test relies on connections made from the PHP script directly to the database. The PMS relies on connections made with ADO.NET, and queries are made in an entirely different way in the PMS than in the former PHP test. To accommodate this, another test has been developed.

The test is also written in PHP, but relies on cURL for communicating with the PMS website. By requesting an URL and including an authorization cookie, it was possible to create a test that does the following:

1. Load the Project List 5 times.
2. Load the Home page 5 times.
3. Load the Current Iteration page 5 times.

Notice that cURL does not read files included through HTML `src` attributes (such as JavaScripts, CSS files and images) which improves the loading speed compared to loading pages in a browser. Browsers do however cache many of these static files which minimizes the difference in loading.

The test was executed in the same manner as the former test, testing with both one request at a time as well as multiple simultaneous requests. The results are depicted in the graph in figure 5.2.

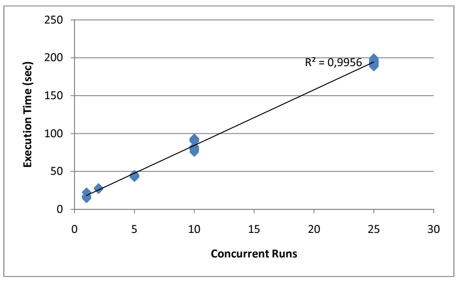


Figure 5.2. Results of running the automated PMS usage test. The graph shows the amount of seconds it took to execute the script 1 (4 results depicted), 2 (2 results depicted), 5 (5 results depicted), 10 (20 results depicted) and 25 (25 results depicted) times concurrently.

From this test it is possible to see that the execution times are proportional with the amount of concurrent requests. This may either be due to the way the PMS serves the pages or limitations in the network connection of the machine on which the tests were executed. If it is indeed the PMS that gets slower for each concurrent request, this could pose a problem if many users access the PMS at the same time. Further testing from separate connections would however be required to investigate this further.

What can be concluded from the test is that the PMS is able to handle multiple concurrent requests without failing. The returned results are, while delivered slower, all correct and valid. Further similar tests have been done to ensure adding data to the PMS through the POST forms on the website is also possible with concurrent connections, which also turned out succesful.

The test was not able to execute correctly when using more than about 30 concurrent connections, as the MySQL database itself limits the amount of connections allowed. This does not mean 30 users cannot use the website at once - MySQL's built-in connection pool allows users to share connections when they are not used concurrently. As 30 online users are unlikely to query the database at the very same time, connections will be available for more users.

5.3.3 Load Test

As a final test, the project-creation-script has been executed numerous times in order to fill the database with 300 projects, 2114 iterations, 10029 user stories, 118370 tasks, 33129 issues and 724821 comments. The automated PMS usage script has been executed in order to compare the results with the earlier run on a much smaller database (30 projects). The results are seen in table 5.3

	Run 1 (30 projects)	Run 2 (300 projects)
Show project list	0,58 seconds	13,5 seconds
Show home	1,6 seconds	1,7 seconds
Show current iteration	0,6 seconds	1,8 seconds

Table 5.3. Automated PMS usage test compared for 30 projects in the database (run 1) and 300 projects in database (run 2). Results are the average loading time in seconds.

5.3.4 Stress and load test conclusion

Except for when showing the project list, the PMS does not get significantly slower by adding large amounts of data to the database. When browsing the website in a normal browser, no noticeable difference was found. The problem with the project list taking long to load could be easily solved by only showing a limited number of projects on each page and adding the possibility to search for your project.

A slowdown was expected when pages are requested simultaneously, as this is an obvious downside of the server-client pattern. The fact that the loading times seem to be linearily dependent of the amount of results, provides an interesting possibility to predict loading times based on the amount of users connecting to the system. The test was however not found to be conclusive enough to actually use as of basis of such a calculation.

While the tests were all highly unscientific and the results doubtful at best, the tests still show tendencies of how the PMS and MySQL database responds to increased usage. The results shows that the code is written in a way that supports concurrent results and is able to handle at least a couple of hundred projects without any noteworthy problems.

Recapitulation 6

This chapter will sum up the project by reflecting on the process and the outcome. The chapter will also conclude upon the project, and verify if the PMS complies with the system definition. Further the chapter will take a look at future work, that is improvements which could extend the functionality of the PMS. Finally, some perspective will be drawn for the PMS and the environment it is affecting. First, the chapter will discuss one of the major elements that had influence on our project and which made us come up against a brick wall several times.

6.1 Discussions

This section will concern some of the difficulties uncovered during the implementation of the PMS. Two main issues have been selected as these have caused significant problems. We will describe which problems they have caused and how we have decided to solve them.

6.1.1 Communication Between the PMS and MySQL

In the development of the PMS there has been lots of problems in how the communication between different components and technologies has taken place. It is not all problems that have been discussed in this report as not all of them are especially interesting to discuss. Though, this section will discuss one of the more interesting problems there has been in the development of the PMS system, that is the communication problem between the MySQL RDBMS and the running PMS. This problem is interesting because there has been lots of different approaches for solving the problem during the development of the PMS. Every time the problem seemed to be solved, it implied that another problems arrived in the same domain. This section will focus on some of the solution approaches.

6.1.1.1 The Main Problem

The PMS is using a MySQL RDBMS to retrieve and store data and thereby, the PMS needs to communicate with the database server. This is done using ADO.NET that makes it possible to retrieve relational data and handle it as in-memory data represented as objects in the PMS. To abstract the retrieval of data away from the controllers, this code has been located in repository classes in the model layer. Instead of creating new instances of the database object when repositories were to be used, instances of a particular repository were created instead. This made the repositories responsible for retrieving and saving data, as these repositories knew how to communicate with the database object for getting and saving data. That means, if a project from the database was needed, then an instance of the `ProjectRepository` had to be instantiated. The problem was now, how these repositories should get a database connection to use for communicating with the database.

6.1.1.2 Solutions

The first solution was to use a singleton design pattern to make sure that only one connection was created and that all repositories used the same single connection. The idea behind this design choice was to prevent the creation of unnecessary connections to the database. At the same time this made it easier to use more than one repository, as the constructor method in all repositories asked the singleton for a connection ensuring that only one connection was to be used in the system. The use of the singleton design pattern can be seen in listing 6.1.

```
public static mysqlconnection instance
{
    get
    {
        if (connection == null)
        {
            connection = new mysqlconnection();
        }
        return connection;
    }
}
```

Listing 6.1. The singleton pattern used in the `Connection` class was used to have only one `mysqlconnection` in the system. The code is located in *Model/Connection.cs.*

Using this singleton pattern leads to a structure that can be seen in the left illustration in figure 6.1 on the facing page. The representation shows that many users only use one connection between the PMS and the database. The problem by applying this solution arose when the system was tested with more than one user, letting multiple users trying to access a specific page at the same time. Problems also occurred under the development of the PMS, as sometimes the system did not register changes made by other developers in the database.

The reason why these issues occurred, can be described by investigating the `mysqlconnection` object. This `mysqlconnection` object can be seen as an instance of the database, when it first has been created. That means, if changes were made to the database, during the users use of the PMS, the `mysqlconnection` object did not register those, as it did not try to check if updates had been applied to the database or not. The reason for this behavior, can the traced to ADO.NET, which is making use of cache instead of communicating with the database when needed data was requested. This implied that users were not able to register their changes right away.

There are several ways to bypass this by manipulating the `mysqlconnection` object, but these attempts leads to a new problem. When lots of users were using the same `mysqlconnection` it was impossible to determine exactly when the system should execute save and reload commands. The point is, that if user *A* has invoked a function *A1*, which tries to create a new issue with a comment attached to it for instance - then if a user *B* is saving some data another place in the system, while function *A1* only has added the comment to the `mysqlconnection` object, then the system will crash, giving user A a warning, saying that a foreign key constraint has been violated. Further, by using the same instance of the `mysqlconnection`, we were not able to run database requests simultaneously. This affected the generation of charts, as the code responsible for this, were trying to execute SQL-statements in parallel - but since we only had one connection, this attempt failed and the chart generation was not successful.

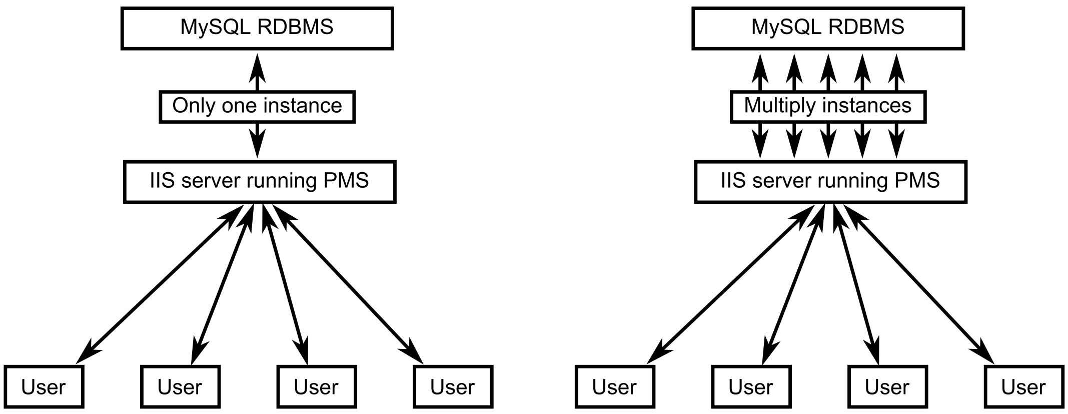


Figure 6.1. The figure on the left shows how the PMS handled MySQL connections with use of singleton. The right figure shows the problem, if each repository creates a separate instance of the `mysqlconnnection`.

The first solution to this problems was to remove the singleton and just make it create a new `mysqlconnnection` every time it was asked for. This operation happens every time a repository was instantiated and lots of `mysqlconnnection` objects could then be made. In theory this should create a `mysqlconnnection` for every user that used the system, but as there are lots of places in the code, which needs more than one repository for executing a method, the use of the system created even more connections per user. In figure 6.1 (right) this situation is depicted, and shows that four users for instance can contribute to the establishment of five `mysqlconnnection` objects. The charts in the system also creates multiple connections as they are loaded as images. This means, that as the most web-browsers today are simultaneously loading images on a page, multiple `mysqlconnection` objects were needed.

It is important to know that the actual numbers of instantiated connections to the database, is not extreme as a built-in connection pool in the ADO.NET [Cor09e] takes care of this. The connection pool remembers all the connections and recycles them, so the actually number of connections is the number of simultaneous requests from users. This can be seen on figure 6.2 on the following page (right).

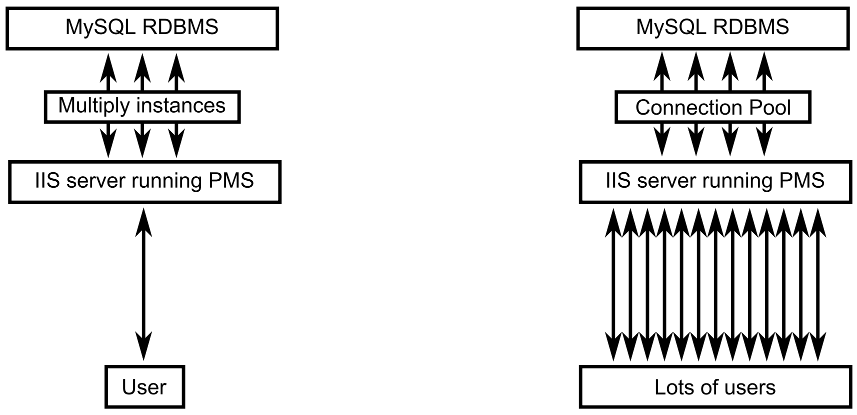


Figure 6.2. The figure on the left shows how a user may lead to the establishment of multiple connections to the database. Right figure shows that many users do not necessarily involve lots of connections to the database.

The main problem with this solution is, that in the theory it is a good idea, but in practice there is a main problem in how ADO.NET handles relations in the database. Relations cannot be made between objects that comes from different `mysqlconnnection` objects. These relationships are used many places in the PMS, such as where a user story must be put together with a particular iteration. The issue occurs because the PMS searches for 'user story' objects by use of an instance of `UserStoryRepository` and uses an instance of `IterationRepository` to search for 'iterations'. These two objects will in this situation use different instances of `mysqlconnnection` and relating the user story together with the iteration cannot be handled by ADO.NET.

6.1.1.3 Final Solution

The final solution to the problem comes with a combination of the two ideas. In the first solution there was not created enough necessary `mysqlconnnection` objects and in the other solution we got too many `mysqlconnnection` objects. The solution is therefore to establish the appropriate number of connections needed, while making sure that the different repositories are using the same connection but only in the same web request. The solution can be seen in listing 6.2.

```
1  public ActionResult Index(int project)
2  {
3      mysqlconnection connection = Connection.newConnection();
4      UserRepository user_db = new UserRepository(connection);
5      ProjectRepository project_db = new ProjectRepository(connection);
6      ...
```

Listing 6.2. Shows how a `mysqlconnnection` is created and sent as parameter to the repositories in line 4-5. The code comes from *Controllers/AboutController.cs.*

The conclusion to this problem may seem very simple but have required many hours to figure out. The reason is, as finding information about how ADO.NET manage its relations to the database and how to normally operate effectively with these, is quite a comprehensive task. This might not be the best approach to a solution to the problem, but it solves the problems mentioned and there has so far been no indication of, that new problems has occurred after this deployment.

The solution also has some disadvantages such as the connection is closed after each web request. Some systems needs the connection to remain open, but this often relates to non web-based systems - and therefore, this is not a problem to further investigate. Though, it should not be a major problem to develop a solution to this if needed.

6.1.2 SQL Discussion

The two primary concerns in the original database design was to automate as many processes as possible in the database, as well as having a fully cascaded tree structure as shown in figure 4.14 on page 72. Cascading was a design criterion because it ensures no data in the database is left unreachable or unrelated to appropriate tables. Automating processes in the database ensures that developers do not have to remember what queries to execute successively, such as making sure a developer is actually assigned whenever a user story's status is changed to 'assigned'. By letting the database ensure that correct procedures are followed, the risks of generating invalid data structures become minimal. Avoiding data redundancy was another primary concern of the database design.

The first design idea was to relate everything through junction tables. This means that every table containing data e.g. `user_story` would be associated to other tables, e.g. `iteration`, through a junction table in the manner seen in figure 6.3.

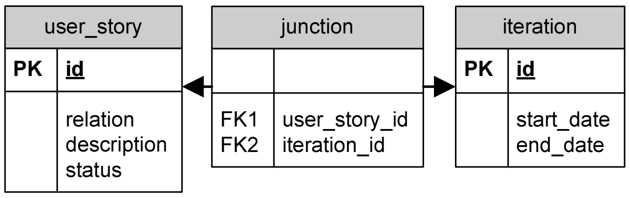


Figure 6.3. Relations between tables using two foreign keys connected through a junction table: `rel_us_iteration.user_story_id` to `user_story.id` and `rel_us_iteration.iteration_id` to `iteration.id`.

This design requires a tuple to be added to the junction table every time a user story is added, requiring developers to always execute two queries. To simplify the procedure of creating two tuples, a trigger would be added to automate the creation of required junction tuples.

The design idea was abandoned for a number of reasons. The junction table is unnecessary when a 1 to 1 relation exists between tables. In the above example, the user story table might just as well contain a nullable foreign key directly to the iteration table. Also, the database host used does not allow creating procedures, checks nor triggers, hence the design would require developers to always remember to add the appropriate tuples.

The revised design idea was to use directly related foreign keys to parent tables in one-to-one relationships (iteration to project for instance), and using junction tables only for one-to-many, many-to-one and many-to-many relationships.

Another design problem was tables that would have multiple possible parents. A comment may for instance be associated to either a task or a user story. The idea was to have only one comment table with multiple possible foreign key constraints (see table 6.1). Having only a single comment table is beneficial if editing the structure of the comment table is ever required. The idea was to add a check constraint to the table, making sure only one foreign key is set (i.e. `foreign_us` or `foreign_task`) is set. But since check constraints are not supported for any storage engines in the version of MySQL installed on the server (5.0.51), this idea had to be abandoned in favor of having multiple comments table for each possible foreign key. However, the original idea (a single table) is actually used in the implemented database design, but must be considered an unreliable design decision, as it is possible to add tuples with multiple foreign keys, making it possible to add a comment to multiple parents.

id (PK)	**text**	**foreign_us** (FK)	**foreign_task** (FK)
1	abc	1	NULL
2	bca	2	NULL
3	cab	NULL	1

Table 6.1. Comments would be associated with appropriate tables through different foreign key columns, of which only one must be set for each tuple.

In general, the limitations in the MySQL engine and the limited functionality provided by the host making it impossible to add checks, procedures and triggers, massively limited the amount of design ideas that could be implemented. This means that many of the checks for valid queries have to be done in the PMS source, which may cause data integrity errors if not done correctly.

6.2 Reflection

The primary goal for this project was to gain knowledge about OOAD and OOP.

The approach to this project was that we initially identified some problems in relation to managing student projects.

According to the goals we prioritized an analysis of requirements and design of the architecture higher than previous projects. This have caused, that we have been very focused on making a correct analysis and thus de-emphasized the matter of whether there really existed a demand for such a project management system. Consequently we have not spent time studying the market, but instead we served the role as a customer ourselves asking demands according to the project proposal and study regulation.

We started relatively late programming parts of this project as a result from the thorough analysis and design stage, as well as the fact that we had to learn many new technologies.

Many of the applied technologies were new to us all, which have caused some hurdles. For instance the majority of the group had a prior knowledge of PHP and were accustomed to that way of developing and found it hard to alter the approach of developing a web page using the ASP.NET MVC.

The combination of using so many new technologies and starting out late with the programming part, made it difficult to foresee the future issues clearly. In order not to stand still for too long, we had to make some assumptions of which technologies were qualified for this project. Some of those assumptions caused problems later, as we late in the process discovered that LINQ did not support cascading for instance.

The testing of the program has been postponed to the last stage of the project. We decided early on, that we would not practice Test Driven Development due to our lack of necessary experience in that field. Though, we have tried to use some of the outcome from the lectures in Object Oriented Programming, by applying a couple of unit tests to some of our service methods. The drawback of this approach have been the difficulty of fulfilling the high prioritizing of the design criteria *Test*, as complying test to an existing program does not impart the same effect on the system as Test Driven Development. However, we tested a couple of methods to verify their correctness and to emphasize, that testing is important.

During the development phase we have used some parts of XP, to help us organize the development process. Every day we held a short morning meeting, introducing the agenda of the day. This helped us structure the division of labor and to get an overview of is someone was struggling with any kind of problems. In that way it became easier to ensure that people were working on appropriate tasks, and to share knowledge about problems discovered throughout the development phase. By using a release plan, divided into short iterations of approximately one week, we once per week started a new iteration holding an iteration planning meeting. At this meeting, we and sometimes in cooperation with our supervisor, selected which user stories to implement next. If some previous user stories have not passed the acceptance test, that is, if the supervisor or our selves was not satisfied with its implementation, then such user stories could be revisited in a new iteration too. Using the iteration planning meeting helped us verifying that were doing right, and that we were reaching our goal of solving the project requirements.

To help sharing knowledge and help each other learning the new technologies, pair programming was practiced. At the beginning, the discipline of only developing in pairs was high, but as the different technology-problems started to arise, it was difficult to keep pair programming working. However, in the period when pair programming was used, the outcome was beneficial.

The use of OO&AD has been crucial for our project. In the begging, after being introduced to the method through the course in SAD, we decided to weight the analysis and the use of OOAD as very important for this project. A main reason for this decision was the study regulation requiring us to analyze demands for a system and to use a suitable architecture. Using OOAD as approach to accomplish this has been beneficial, as it helped us to think about a lot of different structures, patterns, components, architectures and so on through the phase of analyzing what kind of system should be built. Additionally, the method was very suitable when it came to accomplish some sort of structure and flexibility inside the PMS. This is e.g. reflected in the way, it helped us divide the problem- and application domain into two several parts, and by the way it supported us in making a system architecture assembled of components. The method also supported us in structuring our classes, defining their responsibility, attributes and relationships. Seen as an entire whole, the approach of OOAD supported us to get a fundamental knowledge about system analysis, though it sometimes was difficult to survey the particular design, and think about design consequences. As we had been learned through the lectures of SAD, the method requires experience if someone wants to take full advantage of it. Though, the approach by using the method helped us structuring a flexible system design.

In addition to OOP and SAD we followed a course in Software Architecture (SWA). This course introduced us to databases, SQL and gave us a brief introduction to design choices, which supplemented the project well.

In total, the courses of the semester were useful and a good fit for the theme of the semester.

6.3 Conclusion

In this project we have developed a project management system for use in student groups, as a supporting tool for managing the project planning and the administration of persons related to a software development project.

The PMS is capable of handling users with different roles including their affiliations to a project. It is further developed to handle iterations, user stories, tasks, issues and their relation to users in a given project. Plus, the PMS has been built to handle the schedule of a project including computing of work-forecasts and the generation of a burn down chart.

By these functions the PMS is capable of administrating a software development project for several student groups without introducing any unneeded functionality.

We have analyzed the problem- and application domain to define the structure of the system, strongly inspired by the problems sketched in the rich picture and the system definition. From that we have derived an architecture that is based on splitting up the function and model to achieve a flexible structure. The use of ASP.NET MVC directly supports this partitioning and division of responsibility, into a model, view and controller layer.

During the design phase of this project, some design criteria were set to ensure an appropriate architecture and thereby a suitable structure for the PMS. The conditions in this project defined in section 3.1 on page 32 are an important factor as the primary objective is to gain experience on developing a system that fulfills some requirements, in this case controlled by the study regulation. *Comprehensibility* is one of the design criteria, and has been obtained by the use of pair programming and frequently arranged meetings. *Flexibility* has been achieved as mentioned, by the partitioning of components and by making use of ASP.NET MVC. *Correctness* has been obtained by the thorough analysis approached by OOA&D and the study regulation which established the main requirements for the PMS. *Testability* has been achieved by achievement of the criteria *flexibility*; the system has been divided into smaller pieces making them easier to test. The unit test has been implemented on a segment of functions.

The learning outcome of this project has been obtaining knowledge about analyzing and designing a system architecture using the object oriented paradigm. Taking advantage of this architecture helped us to develop a running program implementing the requirements elicited in the analysis. Through our gained knowledge of testing and by applying it to the system, ensured that some functions were tested to verify that they were working as intended. This entire project has supported us in learning and utilizing concepts in the object oriented paradigm.

6.4 Future work

A future version of the PMS could be improved by supporting subversion (SVN) repositories where project code is stored. This would make it possible for the users to make connections between their user stories, tasks, issues etc and a specific element of their source code. This also improves the possibility of documenting a project, as specific actions could be related to particular revisions of source. This also makes it possible to track in exactly which revision a given task has been solved, when an issue has been reported and so on. Finally, implementing SVN would to a greater extent invite the supervisor to take a look at specific source, comment it, and report issues and so on.

Another thing to consider in a future version, could be giving users the possibility to attach specific issues to an iteration. By doing that, it would be possible to schedule solutions of given issues, improving the overall administration of all elements in a project.

A final example of future work, could be implementing E-mail notifications about relevant updates, changes, comments etc. In our development of the PMS, we actually succeeded implementing this feature, but due to host-restrictions, we were not allowed to send E-mails using their SMTP-server. We tried to solve that issue by making a PHP-script, which took GET variables from the URL as parameters. The script was placed at another host, and used the parameters such as who the email should be sent to, the message etc to send an email using a less strict SMTP-server. That functionality was working when testing it as local host, but when we tried to send the HTTP-request from our PMS-host to the host where the PHP-script was located, the PMS-host denied us to do so. That means that if we have had access to another host, with a less strict security policy, we might have been able to implement this feature.

6.5 Perspective

Even though we have developed a project management system, the question for a given project is whether a project management system will actually be beneficial. Why digitalize something that is already manually working?

A project management system is not necessarily the solution to any project, but it depends on factors such as the type of project and the number of participants in the project.

A project management tool will help automate some tasks but this in turn can make the execution of these tasks less flexible as they have to adhere to a format that is predefined by the project management system. This also raises the question of suitability of a project management system. A project management system that follows the waterfall model will be of little use to an agile project. The concepts for an agile project are simply missing from the tool which means that tasks have to be artificially fitted to suit the tool.

Another challenge for a project management system is to capture, share and archive things that are not necessarily easy to digitalize e.g. drawing on a blackboard. In an optimal solution external devices, like digital whiteboards, would come in handy allowing the team to discuss freely without the risk of losing information. Having essential data stored digitally avoids the potential loss of information that could easily occur from only having it on a blackboard (it might be accidentally erased), or on post-its (which could disappear).

In short, a project management tool has to help optimize working procedures, not work against them. Depending on the type of project and the team size a project management tool might be a good idea to use for sharing information, giving a better overview and documentation. But it is essential that the system adapt to the users and not the other way round.

Bibliography

[Ass09a] Ira Assent. Advanced sql. https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2009/SWA/L5_AdvSQL.pdf, 2009. Downloaded from site: December 15th 2009.

[Ass09b] Ira Assent. Sql. https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2009/SWA/L4_SQL.pdf, 2009. Downloaded from site: December 15th 2009.

[Bec02] Kent Beck. Introduktion til extreme programmering, 2002. ISBN: 87-7843-509-9.

[Bor02] Andreas Borchert. White box tests. http://www.cs.rit.edu/~afb/20012/cs4/slides/testing-26.html, 2 2002. Downloaded from site: December 13th 2009.

[BSM07] Mads Bach-Sørensen and Mikael Malm. Automated unit testing - a survey of tools and techniques. Technical report, Aalborg University - Department of Computer Science, 2007.

[Bur98] Dr. Ron Burback. Current distributed architectures. http://infolab.stanford.edu/~burback/dadl/node103.html, December 1998.

[CC08] Inc. Cunningham & Cunningham. What is refactoring. http://c2.com/cgi/wiki?WhatIsRefactoring, October 2008. Downloaded from site: October 25th 2009.

[Cob09] Cobrasoft. Meeting: Better results. http://www.sxc.hu/photo/1131288, January 2009. Downloaded from site: December 12th 2009.

[Cor01] Microsoft Corporation. Ado.net for the ado programmer. http://msdn.microsoft.com/en-us/library/ms973217.aspx, December 2001. Downloaded from site: December 8th 2009.

[Cor07] Microsoft Corporation. C# language specification version 3.0, page 1. http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharpLanguageSpecification.doc, 2007. Downloaded from site: December 8th 2009.

[Cor09a] Microsoft Corporation. Active server pages. http://msdn.microsoft.com/en-us/library/aa286483.aspx, 2009. Downloaded from site: December 14th 2009.

[Cor09b] Microsoft Corporation. Ado.net overview. `http://msdn.microsoft.com/en-us/library/h43ks021.aspx`, December 2009. Downloaded from site: December 8th 2009.

[Cor09c] Microsoft Corporation. Asp.net mvc overview. `http://www.asp.net/learn/mvc/tutorial-01-cs.aspx`, December 2009.

[Cor09d] Microsoft Corporation. Authenticating users with forms authentication (c#). `http://www.asp.net/Learn/mvc/tutorial-17-cs.aspx`, 2009. Downloaded from site: December 6th 2009.

[Cor09e] Microsoft Corporation. Sql server connection pooling (ado.net). `http://msdn.microsoft.com/en-us/library/8xx3tyca.aspx`, December 2009.

[Cor09f] Microsoft Corporation. Understanding models, views, and controllers. `http://www.asp.net/learn/mvc/tutorial-02-cs.aspx`, December 2009.

[Cra07] Bob Cramblitt. Alberto savoia sings the praises of software testing. `http://searchsoftwarequality.techtarget.com/news/article/0,289142,sid92_gci1273161,00.html`, 9 2007. Downloaded from site: December 13th 2009.

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1 edition edition, 1999. ISBN: 02-0148-567-2, Can be viewed at `http://kortlink.dk/78vv`.

[gui09] guitargoa. Graph1. `http://www.sxc.hu/photo/1239215`, November 2009. Downloaded from site: December 12th 2009.

[HKSU06] Chris Hart, John Kauffman, David Sussman, and Chris Ullman. Beginning ASP.NET 2.0 with C#. Wiley Publishing, Inc, 1 edition edition, 2006. ISBN: 0-470-04258-3.

[HPT09] Junio C Hamano, Shawn O. Pearce, and Linus Torvalds. About git. `http://git-scm.com/`, 2009. Downloaded from site: December 4th 2009.

[Kor03] Jukka "Yucca" Korpela. Methods get and post in html forms - what's the difference? `http://www.cs.tut.fi/~jkorpela/forms/methods.html`, September 2003.

[Ltd09] Atlassian Pty Ltd. Bug tracking, issue tracking & project management software - jira. `http://www.atlassian.com/software/jira/`, October 2009. Downloaded from site: October 15th 2009.

[maj09] majaFOTO. Sunrays. `http://www.sxc.hu/photo/1243996`, November 2009. Downloaded from site: December 12th 2009.

[Mic09] Microsoft. Linq). `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`, 2009. Downloaded from site: December 8th 2009.

[MM08] MySQL and Sun Microsystems. Mysql connector/net. http://dev.mysql.com/doc/refman/5.0/en/connector-net.html, 2008. Downloaded from site: December 8th 2009.

[MM09] MySQL and Sun Microsystems. Mysql 5.4 reference manual, chapter 1. http://dev.mysql.com/doc/refman/5.4/en/introduction.html, June 2009. Downloaded from site: December 8th 2009.

[MMMNS01] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. Object-oriented analysis & design. Marko Publishing, 1st edition, 2001. ISBN: 87-7751-150-6.

[Net09] Microsoft Developer Network. Asp.net web applications. http://msdn.microsoft.com/en-us/library/ms644563.aspx, 2009. Downloaded from site: December 8th 2009.

[Nør09] Kurt Nørmark. Test of object-oriented programs. http://www.cs.aau.dk/~normark/oop-09/html/notes/test-book.html, 2009. Downloaded from site: December 14th 2009.

[RAW09] RAWKU5. Deer. http://www.sxc.hu/photo/1138596, January 2009. Downloaded from site: December 12th 2009.

[RtjT09] John Resig and the jQuery Team. jquery: write less, do more. http://jquery.com/, December 2009. Downloaded from site: December 8th 2009.

[SKS06] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. Database system concept. McGraw-Hill, 5 edition edition, 2006.

[Sof09] Edgewall Software. Welcome to the trac open source project. http://trac.edgewall.org/, October 2009. Downloaded from site: October 24th 2009.

[Tav08] Chris Tavares. Building web apps without web forms. http://msdn.microsoft.com/en-us/magazine/cc337884.aspx, March 2008. Downloaded from site: December 8th 2009.

[Uni09a] Aalborg Universitet. Study regulation 2009 (studieordning for bacheloruddannelsen i sofware, 3. - 6. semester). http://fsn.aau.dk/fileadmin/fsn/dokumenter/7_studieordninger/f09/softwbach_sept2009.pdf, September 2009. Quote is translated from danish to english.

[Uni09b] Charles Sturt University. Thin client. http://www.csu.edu.au/division/dit/services/service-catalogue/thin-client/what-is-thinclient.htm, November 2009.

[W3S09] W3Schools. Ajax introduction. http://www.w3schools.com/Ajax/ajax_intro.asp, October 2009. Downloaded from site: October 15th 2009.

[Wel99a] Don Wells. Extreme programming - acceptance tests. `http://www.extremeprogramming.org/rules/functionaltests.html`, 1999. Downloaded from site: December 12th 2009.

[Wel99b] Don Wells. Extreme programming - iteration planning. `http://www.extremeprogramming.org/rules/iterationplanning.html`, 1999. Downloaded from site: December 11th 2009.

[Wel99c] Don Wells. Extreme programming - pair programming. `http://www.extremeprogramming.org/rules/pair.html`, 1999. Downloaded from site: December 11th 2009.

[Wel99d] Don Wells. Extreme programming - release plan. `http://www.extremeprogramming.org/rules/commit.html`, 1999. Downloaded from site: December 11th 2009.