

MASTER

ROOMS

ROlap based Occupation Measurement System

Kemps, G.C.M.

Award date:
2011

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

ROOMS: ROlap based Occupation Measurement System

by

GCM Kemps

Supervisor: dr. T.G.K Calders

Supervisor DIZ: M. Kuyck & S. van Zutphen

Master's thesis

Technische Universiteit Eindhoven

Technische Universiteit Eindhoven

August 2011

Eindhoven

The Netherlands

Abstract

As part of the Eindhoven University of Technology Campus 2020 project, the services Dienst Interne Zaken (DIZ) and Dienst Huisvesting (DH) need data about the use of lecture rooms. To obtain this data, they would like to start measuring the occupation and utilization of lecture rooms during the academic year 2011-2012. They would like to know which rooms are occupied and which rooms are empty during the lecture hours, as well as how many seats there are used in occupied rooms.

In order to obtain this data, a web-based system is designed to store, process and analyze this data. The system needs to be flexible and easy maintainable. At the core of the design we have a relational database management system to store all the information. On top of that we need a data warehouse to facilitate decision support. Reports generated by multi-dimensional queries can be extracted from the system. The code framework is based on the MVC design pattern.

The system is implemented in Java using the Struts framework. The model is constructed using the Data Access Objects and Data Transfer Objects design patterns. The view consists of JavaServer Pages. In the persistence layer MySQL is chosen as the relational database with an open source 100% Java written OLAP server called Mondrian on top. Measurement data obtained from previous projects was used to test and evaluate the system and its design.

Acknowledgements

I would like to thank the following people without whom I would not have managed to bring this project to a satisfying end:

First my gratitude goes to all my supervisors, Monique Kuyck, Sandy van Zutphen and Toon Calders. For the opportunity to do my master thesis at DIZ, your value feedback, suggestions and review of my work throughout the project.

Secondly all my colleagues at the DIZ departement for providing a pleasant and stimulating work environment.

Next I would like to thank Frans Beerens, Ad Winkels, Bert Sprong, Richard Rhemrev and Erwin Wolf for having me and proving me with their knowledge about room occupation and utilization measuring.

Finally all my friends I have met during my time at the Eindhoven University of Technology, my parents, family (in law) and girlfriend for giving all the support and motivation to (finally) finish my master's degree.

List of Definitions

Term	Definition
CSV	Comma separated values
DAO	Data Access Object
DH	Real Estate Management
DIZ	Internal Affairs
DTA	Data Transfer Object
Fontys Eindhoven	University of applied science in Eindhoven
FMIS	Facility Management Information System
Global Viewer	System to monitor and distantly control equipment college rooms
HAN	University of applied science in Arnhem and Nijmegen
JDBC	Java Database Connectivity
JSP	Java Server Pages
MDX	Multi Dimensional Expressions
Microsoft Excel	Spreadsheet application written and distributed by Microsoft
ORM	Object Relational Mapping
Randstad Nederland BV	Dutch temporary employment company
RDBMS	Relational Database Management System
Rostar Eduflex	Timetable administration system
SSV	Semicolon separated values
SQL	Structured Query Language
Syllabus Plus	Software suite for timetabling used by TU Eindhoven
TU Delft	Delft University of Technology
TU/e	Eindhoven University of Technology
Occupation	The number/percentage of rooms that are occupied
Utilization	that number/percentage of seats that are in use
Untis	Timetable administration system
UU	University of Utrecht

Table of Contents

Abstract	ii
Acknowledgments	iii
Definitions	iv
Table of Contents	vii
List of Tables	viii
List of Figures	x
1 Introduction	1
1.1 Background	1
1.2 Purpose of the project	2
1.2.1 Project goal	2
1.2.2 Project steps	2
1.3 Outline of the report	3
2 Similar projects	4
2.1 Research	4
2.2 Technische Universiteit Eindhoven	5
2.2.1 Methodology	5
2.2.2 Results	5
2.3 Fontys Eindhoven	5
2.3.1 Methodology	6
2.3.2 Results	7
2.4 TU Delft	7
2.4.1 Methodology	8
2.4.2 Results	8
2.5 Universiteit Utrecht	10
2.5.1 Methodology	10
2.5.2 Results	11

2.6	HAN	11
2.6.1	Methodology	11
2.6.2	Results	11
2.7	Comparison	12
2.8	Conclusion	12
3	Requirements	14
3.1	System description	14
3.1.1	User management	15
3.1.2	Period management	15
3.1.3	Room management	15
3.1.4	Reservation management	16
3.1.5	Measurement management	16
3.1.6	Reports	17
3.2	Functional requirements	17
3.3	Non-Functional requirements	17
3.4	Use cases	18
3.4.1	Use Case A	18
3.4.2	Use Case B	19
3.4.3	Use Case C	20
4	Design	23
4.1	Relational database	23
4.2	ROLAP	25
4.3	Architecture	29
4.4	Data import	32
4.4.1	Rooms import	33
4.4.2	Timetable import	33
4.5	Environment	34
5	Implementation	35
5.1	Struts	35
5.2	ROOMS	37
5.2.1	Controller	38
5.2.1.1	Forms	40
5.2.2	Model	40
5.2.2.1	Data Transfer Object	41
5.2.2.2	Data Access Object	41
5.2.2.3	Service	42
5.2.3	View	44
5.2.3.1	JSP tag libraries	44

5.3	MySQL	45
5.3.1	MyBatis	45
5.4	Mondrian	48
5.4.1	Mondrian Schema	50
5.4.2	Caching and Tuning	53
5.4.3	MDX	54
5.4.4	JPivot	56
5.5	Deployment	56
6	Results	57
6.1	Tests	57
6.2	Evaluation	57
6.3	Future work	58
	Appendices	58
	A Functional Requirements	59
	B Package Diagram	65
	C Class Diagrams	66
	D Struts Config File	72
	E Deployment Descriptor File	75
	F Screenshots	78
	G MySQL Structure	86
	Bibliography	88

List of Tables

2.1	Tabular comparison of occupation/utilization measurements between institutes	13
3.1	Requirement priorities	17
A.1	General requirements	59
A.2	User management requirements	60
A.3	Room management requirements	61
A.4	Timetable management requirements	62
A.5	Period management requirements	62
A.6	Measurements requirements	63
A.7	Reporting requirements	64

List of Figures

2.1	TU Delft report: occupation and utilization per week	9
4.1	ER model of the relational database	25
4.2	Cube Example	27
4.3	OLAP Drill Down/Up Example	28
4.4	OLAP Slice Example	28
4.5	OLAP Dice Example	29
4.6	Data Access Object	30
4.7	Data Access Object sequence diagram	31
5.1	Struts client request	36
5.2	Struts Architecture	37
5.3	Example Service Sequence: getCurrentDIZTime()	43
5.4	MyBatis Architecture	47
5.5	Mondrian Architecture	49
B.1	ROOMS UML Package Diagrams	65
C.1	UML Classes: Action package	67
C.2	UML Classes: DTO package	68
C.3	UML Classes: Service package	69
C.4	UML Classes: Form package	70
C.5	UML Classes: Persistence package	71

F.1	ROOMS login	78
F.2	ROOMS Main menu	79
F.3	User management	80
F.4	Room management	80
F.5	ActionForm example	81
F.6	Period management	81
F.7	Reservation management	82
F.8	Measurement management	83
F.9	Report selection	84
F.10	JPivot report example	85

Chapter 1

Introduction

1.1 Background

The number of lecture rooms available for education has been reduced by 40% since 1994. With the Campus 2020 project¹ coming up, the expectation is that number of lecture rooms will drop even more [1]. The Campus 2020 project is an extensive renewal plan for realizing a compact campus for personal use of the Eindhoven University of Technology. The compact campus is a place where the boundary between science and business is blurred by the common and flexible use of rooms and buildings. More efficient use and sharing of space means that unnecessary buildings can be disposed. The compact campus should also accommodate the growing need for small study rooms and the flexible use of lecture rooms.

In the meantime the number of students has only increased over the years. As a result, the timetable scheduling became more and more complex. Even more because of an increase of educational possibilities. In 2007 a new timetabling software system was introduced as a result of the research of Bas Ligtenberg [1]. He advised to purchase Syllabus Plus as the university wide timetabling software package. Before 2007, most faculty timetabling coordinators constructed their timetables manually using Microsoft Excel. Because no central system was used, a lot of information had to be entered in each separate system, creating a lot of superfluous work. Every coordinator had his or her own way of constructing timetables and these procedures were not well documented. Absent coordinators were

¹<http://www.tue.nl/universiteit/over-de-universiteit/tue-science-park/de-compacte-campus-campus-2020/>

therefor hard to replace.

Now that timetabling coordinators are able to tackle problems in a more structured and user friendly way, it is time to investigate how the timetable is respected. How many of the rooms are actually occupied? And if the rooms are occupied, how well are they utilized?

1.2 Purpose of the project

As part of the Campus 2020 project the services Internal Affairs and Real Estate Management (DH) need data about the use of lecture rooms. They want to determine the total amount of rooms required and its possible dimensions. To obtain this data, DIZ would like to start measuring the occupation and utilization of lecture rooms during every second week of a teaching period during the academic year 2011-2012. They would like to know which rooms are occupied and which rooms are empty during the lecture hours, and how many seats are used in occupied rooms.

1.2.1 Project goal

In order to obtain this data, a system has be designed to store, process and analyze this data. Also, data has to be imported from the Syllabus Plus timetabling system. From these data sets, reports can be generated from different points of view. These reports can serve as decision support for the DIZ management.

1.2.2 Project steps

To design such a system, the following steps are to be followed:

1. Trace universities or high schools that have done a similar investigation on room occupation and utilization. Find out why they performed such an investigation, what were their measuring methods and what were the final results. There may also be common problems that need extra attention.
2. Based on this investigation and stakeholder interviews, determine which data to store in the database, what reports are desirable and other requirements.
3. Design, build and test the system based on these agreed requirements.

4. Test, evaluate and adjust the system where necessary.

1.3 Outline of the report

After this introduction we will start with a chapter on the findings of the investigation among other educational institutes. Several universities and high schools were contacted to ask if they have been investigating their room occupation or utilization. Almost all of these institutes did, but the four most interesting cases were picked out for further research.

Next we will describe the functionality of the system in an informal way and state the agreed (non)-functional requirements. Subsequently three use cases are presented to further illustrate the functionality of the system.

Based on the investigation and the requirements we will create a design for code, database and data warehouse in Chapter 4. At the core of the design we need a relational database to store all the information needed and gathered during the measurement periods. On top of that we need data warehouse functionality to facilitate decision support. The data warehouse must support on-line analytic processing (OLAP) which enables us to do multi-dimensional analytic queries. We need this functionality to generate reports. Finally we need a code framework to handle the requests, manage the data and generate the views. Chapter 5 will elaborate on the implementation of the design. The code implementation will be based on the Struts framework. Struts uses a MVC approach and makes it easier for web developers to build web applications based on Java Servlets and Java Server Pages (JSP). MySQL will be used as the relational database. MySQL has become the most popular open source database of the world. Mondrian is a popular open source OLAP solution. It is written in 100% Java, executes queries written in the MDX query language and can read its data from a relational database.

Finally we will discuss the results of the project. We will analyse the system using data obtained from previous measurements and currently available timetable data from Sylabus.

Chapter 2

Similar projects

The Eindhoven University of Technology (TU/e) is not the first to research room occupation and utilization. Based on previous cooperation, we already know that other universities and high schools are trying to obtain, or have already obtained, some kind of occupation and utilization analyses. It is a good idea to contact these parties and benefit from their experience and learn from their approach.

2.1 Research

Several universities and high schools were contacted to ask if they have been investigating their room occupation or utilization. Almost every of these institutes did and the four most interesting cases were picked out for further research. Some key questions that were asked during the meetings were:

- What were the goals of their research? What did they intend to find?
- What was measured during the research?
- How were the actual measurements prepared?
- How were the measurement carried out? What was the methodology used?
- What were the results of the research?
- What was the conclusion of the research? Were the results used in any way?
- What problems occurred during the research? What would they do differently?

2.2 Technische Universiteit Eindhoven

A small scale project was already carried out a few years ago on a small sample of rooms. In 2006 all rooms of the Auditorium were measured during 2 different periods of time. The first period was during week 29, after a few weeks followed by the second period consisting of weeks 43, 44, 45 and 46. In 2007 another period of 5 weeks (week 6 to week 11) was added, but his time over all lecture rooms scheduled centrally by DIZ.

2.2.1 Methodology

The measurements were carried out by physically visiting the rooms during every lecture hour. This was done by DIZ employees and invigilators. They wrote down their findings on specially prepared lists. Afterwards this data was manually imported into Microsoft Excel for analysis and reporting.

2.2.2 Results

Over all Auditorium rooms during the total of 10 weeks, the following results were obtained:

- The average percentage of reserved rooms is 66% with a minimum of 59% and a maximum of 71%
- The average percentage of occupied rooms is 55% (from all rooms) with a minimum of 46% and a maximum of 65%
- The average percentage of seat utilization (occupied rooms only) is 33% with a minimum of 26% and a maximum of 42%

2.3 Fontys Eindhoven

At the Fontys Eindhoven they have been measuring utilization of college rooms for 2 years now. The measuring is performed per building and they have just finished their 15th measurement. The primary reason Fontys started with this project is to get a better insight in their utilization of rooms and buildings. The measurements mostly take place

in the first of four semesters. This is the only semester where there are no internships planned for third or fourth year students. Therefore the first semester is the busiest period of the year and is the limiting factor of the total capacity of Fontys Eindhoven.

2.3.1 Methodology

In preparation of the coming measurements, the so called *counting lists* are composed. The lists are composed by Frans Beerens, facility manager at Fontys Eindhoven. These lists are used by the counters to register their gathered data. The counting lists are composed using the Facility Management Information System (FMIS). This system is used to register which rooms are located in what building and who owns them. This information is placed alongside the timetable and this results in the final counting lists. Several timetable systems are used by Fontys, for example Untis and Rostar Eduflex. Research from Bas Ligtenberg [1] shows that these systems are particularly suited for secondary and higher education and not for university education. The educational structure these systems support does not correspond to the TU/e structure. Untis and Rostal Eduflex are more pointed towards scheduling different groups of students (minimize spare hours), while the TU/e structure is more pointed towards activity scheduling. Meeting the requirements for every activity is more important than students having spare hours in between classes. The counting lists consist of a few columns including: *room* (roomID), *capacity*, *period*, *date* (day of the week), *occupation* (number) and *scheduled* (yes/no). The capacity of the rooms is expressed in 3 categories, category A (< 20), category B (20 to 30) and category C (> 30). Eventually, the column *occupation* will be filled in by the counters.

During the measurement periods, employees will walk past the lecture rooms. Every period (lecture hour) a room is checked for its occupation and utilization. Most rooms have 10 periods a day, some have 12. One measurement period will last for exactly one week, from Monday to Friday. Usually the third week of the semester is picked, because by then most of the timetable errors are rectified and will no longer interfere with the measurement. All measurements start 10 minutes after the start of the period and 10 minutes before the end of it. Every employee is able to check 50 to 60 rooms within this half hour interval. He or she simply counts every student present in the room. The counting lists are composed and ordered in such a way, that every employee can go from room to room easily and only has to write down a single number per room.

2.3.2 Results

The data obtained by the counters are to be returned to the facility manager. He enters the data back into the digital counting lists using Microsoft Excel. One day worth of data will take two days to process. No extra tables, graphs or aggregated data is derived from the data. The interpretation of the data is directly done on the (sorted) Excel tables themselves.

The interpretations of the results are discussed with the managers and the most striking and problematic results are provided as feedback to the timetable makers. Since this feedback is made possible, Fontys Eindhoven was able to increase their occupation with 10% or 15% to a total of 40%. Because of this, Fontys was able to push off some of their buildings and therefor reduce cost.

The goal of Fontys Eindhoven is to further increase their utilization to about 60% or 70%. According to Frans Beerens, the utilization percentage has been around 80% a few years ago. But because of the switch to decentralized timetabling, this number has decreased significantly. There is also the intention to facilitate every room with the same equipment. Timetable makers should then only worry about the capacity instead of required facilities such as beamer, sound system, etc.

2.4 TU Delft

During 2007 the departement FMVG (Facilitair Management & Vastgoed) of the TU Delft has carried out a few measurements divided in two measuring periods. The first period was at the end of the academic year 2006-2007 (10th of April to 8th of June) and the second period at the start of the new academic year 2007-2008 (3rd of September to 19th of October). The research focused on the 80 lecture rooms with a centralized timetable. This resulted in 52.000 data entries. The research was started to give more insight in the use of centrally timetabled lecture rooms. There was a persistent demand for more rooms by the TU Delft board while there was a dominant feeling within the FMVG departement that rooms are often empty [6].

2.4.1 Methodology

Similar to Fontys Eindhoven, FMVG prepared counting lists. For every single room there was a list available with an eight row long table where every row represents a single period (lecture hour). In every row the counters can write down their findings.

The research (actual measurements) was performed by Randstad Nederland BV. They provided 20 counters each week. During the afore mentioned periods, counters (wearing Randstad sweaters) entered the rooms during lectures and counted the students present . The number (or percentage) of students present was written down on the counting lists.

The obtained data is then entered into a database at the end of every week. Meanwhile the data is checked for (obvious) errors manually by the editors. MS Access has been used as their database. With the help of the built-in MS Access report functionality the results are exported from the database. Some definitions were made beforehand:

$$Occupation = \frac{\text{Number of periods room is not empty}}{\text{Total number of periods}} \times 100\%$$

$$Utilization = \frac{\text{Number of students present in room}}{\text{Total number of seats in room}} \times 100\%$$

The different reports can be found in the report by Sprong and Winkels [6]. An example of one of their reports can be found in Figure 2.1.

2.4.2 Results

The following results have come forward from the investigation described above:

- The average scheduled occupation (with respect to the timetable) is 64%
- The true average occupation of the lecture rooms is 49%
- The average utilization of occupied rooms is 32%
- From all occupied rooms, only 3,4% of the rooms is 100% utilized
- From all occupied rooms, only 14% of the rooms is > 70% utilized

Some environmental factors that were taken into account while interpreting the results:

- Number of registered students (per 1-09-2007): 14.281

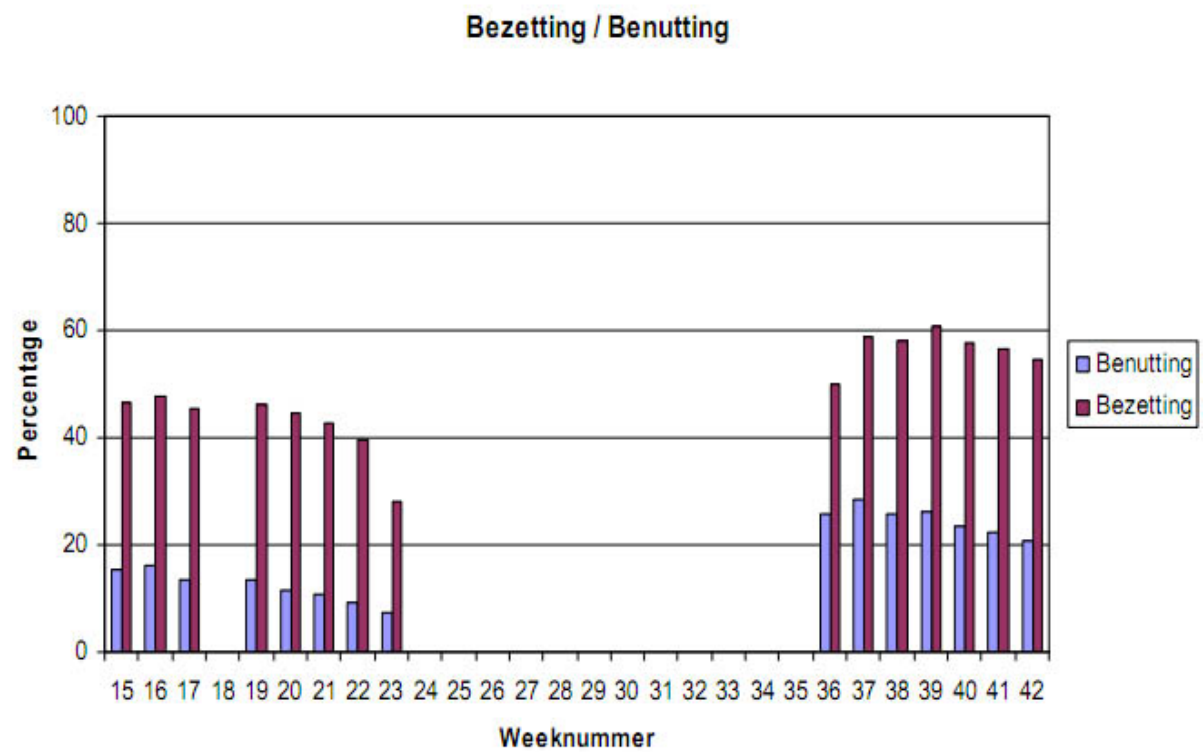


Figure 2.1: TU Delft report: occupation and utilization per week

- Further differentiation in education forms. Smaller project- and work groups result in a larger demand in rooms.
- The construction of the timetable plays an important part in increasing occupation. Aspects that play a role:
 - On time delivery of educational information to the timetable makers
 - Delivery of the right educational information
 - Determining the correct education form the teacher is going to use for his or her subject such that appropriate rooms can be booked.

2.5 Universiteit Utrecht

The University of Utrecht also performed some measurements in the past. However, these measurements were not very well structured and were carried out in an ad hoc fashion. Different sample days were chosen and not all rooms and periods were covered. They had, just as the other parties involved in this research, the feeling that rooms were more often empty than expected. They felt that a measurement sample would be enough to confirm their suspicion.

2.5.1 Methodology

From the timetabling system Syllabus Plus, counting list are exported that can be used by students to perform the measurements. In every block of 10 weeks, 2 sample days are picked to carry out these measurements. During those days students pass by lecture rooms for the whole morning and part of the afternoon. As a result not all periods are covered by the measurements.

Together with the exported list from Syllabus Plus, students walk past the lecture rooms. Whenever the room is indeed occupied, also the utilization is measured. Only the rooms present in the timetable are measured, unless it is clearly visible from the outside that the room is occupied. This can of course lead to incomplete and inconsistent measurements.

2.5.2 Results

The obtained data is processed by the students themselves in Excel. This results are then again compared with the timetable data and conclusions are drawn. Because the measurements can be incomplete, the results are interpreted on a more abstract level. Nevertheless, feedback from the measurements led to better timetabling insights which then led to an increase of the utilization.

2.6 HAN

At the Hogeschool Arnhem Nijmegen (HAN) measurements were also taken. They were curious to know about their room occupation and utilization and if some improvement could be made there. During the academic year 2008-2009, in the third of five semesters, only one building was measured. The building covered an area of $15.000m^2$ which comes down to 70 (also non-educational) rooms.

2.6.1 Methodology

No real preparation was done before the measurements. Invigilators that were already employed by the HAN during the examination periods, were asked to execute the measurements. Their timetabling system (Untis) was used to prepare the counting lists.

Similar to the methodologies described earlier, the invigilators walked passed the 70 rooms in the building. Every room was checked every period. The results were processed in Excel worksheets.

2.6.2 Results

The results obtained from the measurements show a significantly higher number then we have seen so far. This could (partly) be explained by the fact that also non-educational rooms were taken into account.

- The average occupation of the rooms in the building ($> 70\%$)
- The average utilization of occupied rooms ($> 40\%$)

Based on the results described above, HAN felt that there was room for improvement. Starting in 2010, they implemented a real time occupation measurement system, Room Management System (RMS) by Cofely [7]. Cofely RMS offers realtime insight into the actual use of space. Movement sensors detect whether someone is present. This information is then stored in a database and presented via internet and touch screens. Information on the usage of a building and current management information is easily available. Touch screens in every hallway show a map of the current floor. Every room that is not in use is colored green, occupied rooms are colored red. Touch screens can also be synchronized with the timetabling system. Rooms that are reserved but not yet in use are then colored orange. If the room is not used within 15 minutes from the start of the period, it will become available and its screen will turn green. At the moment the RMS system can only detect room occupation, but not the number of present persons. However, it is expected that this functionality will be added soon.

2.7 Comparison

An overview of the comparison between the institutes discussed above can be found in Table 2.1.

2.8 Conclusion

Every educational institute that was contacted for the sake of this research does (or did) some kind of research on room occupation and/or utilization. These researches are started because of the feeling of underutilization or the desire to increase these occupation and utilization figures. Every party counts rooms by physically visiting them. Most of them use simple Excel sheets to prepare lists and process data. Only the HAN uses specialized software while the TU Delft uses a relational database system. Measurements were taken in different intervals mostly at the beginning of the academic year but not in the first week(s). According to the interviewees the most important part of the measurement process is the preparation. Well prepared measurements give structure to the entire process, speed it up and prevent errors.

		TU/e	Fontys	TU Delft	UU	HAN
Year		2006-2007	2009-2011	2007	2008	2009-2011
Method		Counters	Counters	Counters	Counters	Counters/Automatic
Executors		Emp./Invigilators	Employees	Temporary emp.	Students	Invigilators
#Rooms		66	All	80	Scheduled only	70
Frequency		once a period	once a period	once a period	4-6 times a day	once a period
Period		10 weeks	3 rd week 1 st semester	2 × 8 weeks	2 days	7 weeks
Room occupation		yes	yes	yes	yes	yes
Room utilization		yes	yes	yes	yes	yes/no
Processing		Excel	Excel	Access	Excel	Excel/RMS
Timetable System		Decentralized	Untis/Rostar	Syllabus	Syllabus	Untis

Table 2.1: Tabular comparison of occupation/utilization measurements between institutes

Chapter 3

Requirements

In this chapter an informal description of the system is given. In the following section the functional requirements will be addressed, which were agreed on by the DIZ supervisors. These are followed by the non-functional requirements of the system. Finally we will discuss three use cases which will sketch the functionality of the system together with its user interaction.

3.1 System description

Let's start with an informal description of the system. The system will be build as a web-based system, for the sake of portability and maintainability. Different people need to work with the system, either to monitor the measurements and its results, or to actually enter the measurement into the system. When the web-based system is installed on a central server, there is no need to install software on different computer systems except for an internet browser. Maintainability is also important. Because the system will be used for the first time starting September 2011, it might be adjusted based on user experience. When the system provides good decision support, it might also be fit for further expansion in functionality.

The system will be split up into six different parts:

- User management
- Period management
- Room management

- Reservation management
- Measurement management
- Reports

3.1.1 User management

The user management section will take care of the users of the system. Users can be either *Administrator* or *Counter*. The exact rights of each user become clear in Section 3.2, but basically the Administrator is allowed to add, edit or delete different items such as periods, rooms or reservations. Counters are allowed to see all information, but are only allowed to add or update measurement data.

The functionality of this part of the system, as well as most others, comes down to the four basic persistent storage operations: create, read, update and delete (CRUD) users. Every user has a *username* and a *password* to be able to log in. Administrators can assign passwords, after logging in everybody can change his or her own password. Furthermore every user has a *full name*, *user type* (as mentioned above) and an *active* flag. The active flag can be used to (temporarily) deactivate a user. This can be useful when deleting a user is not desirable, as there might be other objects referring to a user entry.

3.1.2 Period management

This section will manage the lecture hours or periods. Every university has a different period timetable and they are often likely to change over time. The same CRUD operations are provided. Periods have a *rank*, *start time*, *end time* and an *active* flag.

3.1.3 Room management

The measurements that are going to be taken, are about room utilization and occupation. So we need a section that will administer the rooms we will be measuring. Again, the CRUD operations will be necessary. Because the adding and updating of rooms will be a cumbersome activity, there should also be an import function. Rooms are already stored in the Syllabus timetable system after all. Syllabus provides an exporting mechanism through spreadsheets or CSV/SSV text files. Importing these files should result in adding missing rooms and updating existing ones.

Rooms are described by their *name* (full name and abbreviated short name), *building*, *size*, *capacity*, *type* ('Flat' or 'Amphitheater'), *active* and *notebookReady* flags. The size will be represented by a character in the range of A-E. A being smallest category of rooms and E being the largest. The exact definitions can be found in Table A.3 in Appendix A.

3.1.4 Reservation management

In order to relate our measurements to made reservations, we need functionality to import data from Syllabus. Of course it is also possible to create, update and delete single reservations, but taking into account that thousands of reservations are made every academic year, most work should be done by automatic imports. As mentioned earlier, Syllabus provides us with an exporting facility in its software. For every activity scheduled in Syllabus, we are able to export its module *name* or *subjectID*, the *room* it takes place in, its *date*, *start time*, *end time* and responsible *lecturer(s)* and *faculty*. These are the attributes we need to keep as well.

The reservations can be viewed in a tabular fashion. The end user can pick a date and period and as a result a table will be shown with as many rows as there are active rooms. For every room that has a reservation on this particular day and time, the above mentioned attributes will be shown on its row as well. Administrators will have the possibility to update or remove these reservations, counters can only retrieve them.

3.1.5 Measurement management

This section is responsible for the most important data in our system: measurements. It will have the same interface as reservation management, with a tabular overview of all the rooms on a given day and period. In every row the number of occupied seats can be entered. The capacity of the room is also present, to give an indication of its size. If the room is reserved, the row is supplemented with reserving faculty, responsible teacher and subjectID. If the room is not occupied (no seats are used), no action has to be taken. Together with the *occupied seats*, also the *measurement taker* (logged in user) and the *timestamp* of the measurement will be stored. The rows in the table will be sorted on room names, with the reserved rooms first. The end user is able to change the sorting of the table by clicking on the header of a column.

3.1.6 Reports

Now that all our data is stored and manageable, we are ready to generate some reports. There are two types of reports. Reports about *occupation*, i.e. the number of occupied rooms with respect to the reservations. The second type is about utilization, which means the average number of used seats in occupied rooms. Both type of reports will be generated directly, resulting in a 2 dimensional grid. On one axis we will find a time dimension which is either scaled in weeks, weekdays or periods. On the other axis we will find a room dimension, which will occur in the form of room buildings, room types, room sizes or room names. This means that there will be $2 \times 3 \times 4 = 24$ different reports.

3.2 Functional requirements

Now that we have seen an informal description of the system, we will try to capture the system in a number of requirements next. The requirements were established during interviews with stakeholders, mainly with the DIZ supervisors. The requirements are again divided into 6 sections, the same sections as used in Section 3.1. The requirements can be found in tabular fashion in Appendix A. The requirements can have different priorities assigned. The priorities are explained in Table 3.1.

Priorities	
Priority 1	Must be filled in
Priority 2	Should be filled in
Priority 3	Could be filled in
Priority 4	Might be filled in

Table 3.1: Requirement priorities

3.3 Non-Functional requirements

The system will be web based. This makes it accessible from every system, regardless of the operating system or other software installed. Only a web browser is needed. The system has to be installed only once and the data can be stored at one single place. The system should be available at all time from every work station with a web browser installed. Maintainability and flexibility of the system are important for future use, evolution and

expansion. Additional software needed should preferably be open source, since there are no or limited funds available for this project. Response time is not very important, but you should be able to click through the system conveniently. Importing large data sets and generating reports should take as little time as possible but may take several seconds up to a minute depending on the size.

3.4 Use cases

To give more insight in how the system will be functioning and how it will interact with its end-users, some use cases were set up to initially give the supervisors more insight in their future system. Use cases A, B, and C are shown below.

3.4.1 Use Case A

1. Brief description

This use case will describe how an Administrator will add a user to the system

2. Actors

- Administrator

3. Pre-conditions

- An administrator is successfully logged in to the system

4. Basic flow of events

- The use case starts with presenting the main menu of the system
- The administrator clicks to enter *User management* section
- The system will show a list overview of present user records
- The administrator clicks the add-new-user option
- The system presents a blank form with following fields:
 - Username (required)
 - Password (required)
 - Re-enter password (required)

- iv. Real name (optional)
- v. User type (required, presented as dropdown box filled with user types)
- (f) The administrator enters/selects the appropriate information into the fields
- (g) The administrator clicks the save button
- (h) The system will display a confirmation message
- (i) The system returns to the user overview screen from step 3 (which now includes the new user)

5. Alternative flows

- In step f of the basic flow, at least one of the fields is filled in incorrectly =>
 - After step g, the system will display an error message indicating the actual error
 - The user solves the error and resumes at step f

6. Post-conditions

A user is added to the list and the list is shown on the screen.

3.4.2 Use Case B

1. Brief description

This use case will describe how to enter measurements of Monday 12 September 2011, 1st and 2nd period.

2. Actors

- Counter (or Administrator)

3. Pre-conditions

- A counter is successfully logged in to the system

4. Basic flow of events

- (a) The use case starts with presenting the main menu of the system
- (b) The counter clicks to enter *Measurements section*

- (c) The system will show the current day and period in a small form, followed by a list of all rooms currently present in the database that are set to active. The list is ordered by reservations first and ordered further by room ID. Every row will show:
 - i. Room ID (read only)
 - ii. Room capacity (read only)
 - iii. Room occupation (optional, integer > 0 , input field)
 - iv. Reserved? (read only)
 - v. Subject code (read only)
 - vi. Faculty (read only)
- (d) The counter will select Monday 12 september from the date select box and 1st period from the period select box. After pressing the ‘Get measurements’ button, the room list will be refreshed according to the new set timestamp.
- (e) The counter enters the room occupation figures obtained from, for example Global Viewer, in every row necessary.
- (f) The counter confirms its contribution by clicking the save button.
- (g) The system will save the measurements without error warnings.
- (h) The counter will proceed with step d again (but now selecting the 2nd period) and finally exits the measurements section and returning to the main menu.

5. Alternative flows

- In step e of the basic flow, one of the occupation figures is not an integer =>
 - After step f, the system will not save, but will show error message(s) indicating something like ‘digits only’.
 - The user resolves the error and retries the save button

6. Post-conditions

The measurements of the first two periods of the Monday are entered into the system

3.4.3 Use Case C

1. Brief description

This use case will describe how to generate the report described in REP-01: *Show*

average number/ percentage of reservations per week vs. average number/percentage room occupation per week.

2. Actors

- Counter (or Administrator)

3. Pre-conditions

- A counter is successfully logged in to the system

4. Basic flow of events

- (a) The use case starts with presenting the main menu of the system
- (b) The counter clicks to enter *Reporting section*
- (c) The system will provide a menu where the user can choose between 6 report types:
 - i. Reservations vs. Occupation (per week)
 - ii. Reservations vs. Occupation (per weekday)
 - iii. Reservations vs. Occupation (per period)
 - iv. Seat Utilization (per week)
 - v. Seat Utilization (per weekday)
 - vi. Seat Utilization (per period)
- (d) The user will pick an option by clicking the appropriate button (in this case option i)
- (e) In the next step the system will show the final step of the report selecting procedure by presenting the following options:
 - i. Individual rooms
 - ii. Buildings
 - iii. Types
 - iv. Size
- (f) The user will pick an option by clicking the appropriate button (in this case option i)

(g) The report will be generated with on top a form to adjust the time interval (start and end week). The last 4 weeks is chosen as the standard interval. The standard report is presented in a tabular fashion.

(h) If desired the graphical version can be generated

5. Alternative flows

No alternative flows

6. Post-conditions

A tabular and optionally graphical report is generated.

Chapter 4

Design

Given the requirements, the next step is to come up with a robust, flexible and easy to maintain design. At the core of the design, we need a relational database to store all the information needed and gathered during the measurement periods. On top of that we need data warehouse functionality to facilitate the decision support. The data warehouse should be maintained separately from the relational database. It must support on-line analytic processing (OLAP) which enables us to do multi-dimensional analytic queries. We need this functionality to generate our reports. Finally we need a code framework which will combine all pieces. It will handle the requests, manage the data and generate views.

4.1 Relational database

The relational database can be derived from the requirements almost directly. The ER-model of the relational database can be found in Figure 4.1. Underlined attributes are primary keys, while bold attributes are required for every entry. Every part of the system, as described in Section 3.1, has its own table, except for the reports section.

The *users* table can be derived from the USER-01 requirement. An extra attribute *active* has been added, to be able to (de)activate a user, instead of deleting and re-inserting. The *userID* attribute is the primary key, which should be a unique integer. The *username* and *fullName* fields are strings, the *userType* field is either 1 (Administrator) or 2 (Counter) and the *active* field can be either 1 (true) or 0 (false).

The *periods* table can be derived directly from the PER-01 requirement. The primary key *periodID* represents the, obviously unique, lecture hour. The *start* and *end* field are expressed in time format *hh:mm:ss*.

The *rooms* table can be derived from the ROOM-01 requirement. An extra attribute *shortName* is necessary to be able to synchronize with the rooms available from Syllabus, more on that later in Section 4.4. The fields *shortName*, *fullName*, *building* and *type* are strings, *size* is a character between A-E according to requirement ROOM-03 and *seats*, *active* and *notebookReady* are integers. Where *active* and *notebookReady* can be either 0 or 1.

The measurements table is where the actual measurements are stored. This table can be derived from requirement MEAS-01. However, instead of *date* and *period* we will introduce a *timeID* field, which is a foreign key to the *time* table. This table is necessary to be able to make various aggregations in time. More on that in Section 4.2 about ROLAP. The fields *roomID* and *userID* are also foreign keys to their corresponding tables, where *roomID* points the room being measured and *userID* is the id of the logged in user carrying out the measurement. *Occplaces* represents the number of occupied seats measured. Since measurements only take place on occupied rooms, this field can not be empty. Finally the date and time will be saved in the *timeOfEntry* field, to be able to trace back the measurement.

To relate the measurements to the timetable, the *reservations* table is created. Similar to the measurements table it is also related to time and place and has therefor foreign keys *timeID* and *roomID*. The faculty that reserved the room, the subject that is given there and the responsible lecturer(s) are stored in *faculty*, *subjectID* and *teacher* respectively. The extra field added, with respect to requirement TIME-01, is *yearID*. This is the foreign key inherited from the *academicYear* table. This enables us to connect every reservation to a academic year. This makes the importing and emptying of reservations by academic year possible and easier.

As mentioned above the *academicYear* table is introduced to distinguish reservations by academic year. The description field should be used to indicate which academic year it concerns (ie. 2010-2011), but can also be used to create a custom entry (ie. ‘planning’ or ‘congress’), which might be useful for testing or non-academic timetables. The *start* and *end* fields indicate the time span of the record. The field *locked* indicates if the reservations related to this record can be imported or erased.

As mentioned earlier, a special table *time* is created to store timestamps. If we model time information as a hierarchical dimension, it is possible to analyze measurements by year, month, week, weekday, period etc. All the attributes of the *time* table are stored as integers, except *weekday_label*, which stores the day of the week (Monday, Tuesday, etc.) as a string. The meaning of the *year* field is trivial, *month* represents the month of the year (value between 1 and 12), *week* represents the week of the year (week 1 to 52), *weekday* is the day of the week (1 is Monday and 7 is Sunday) and *day* is the day of the month (value between 1 and 31). The *periodID* field is a foreign key pointing to the *period* table.

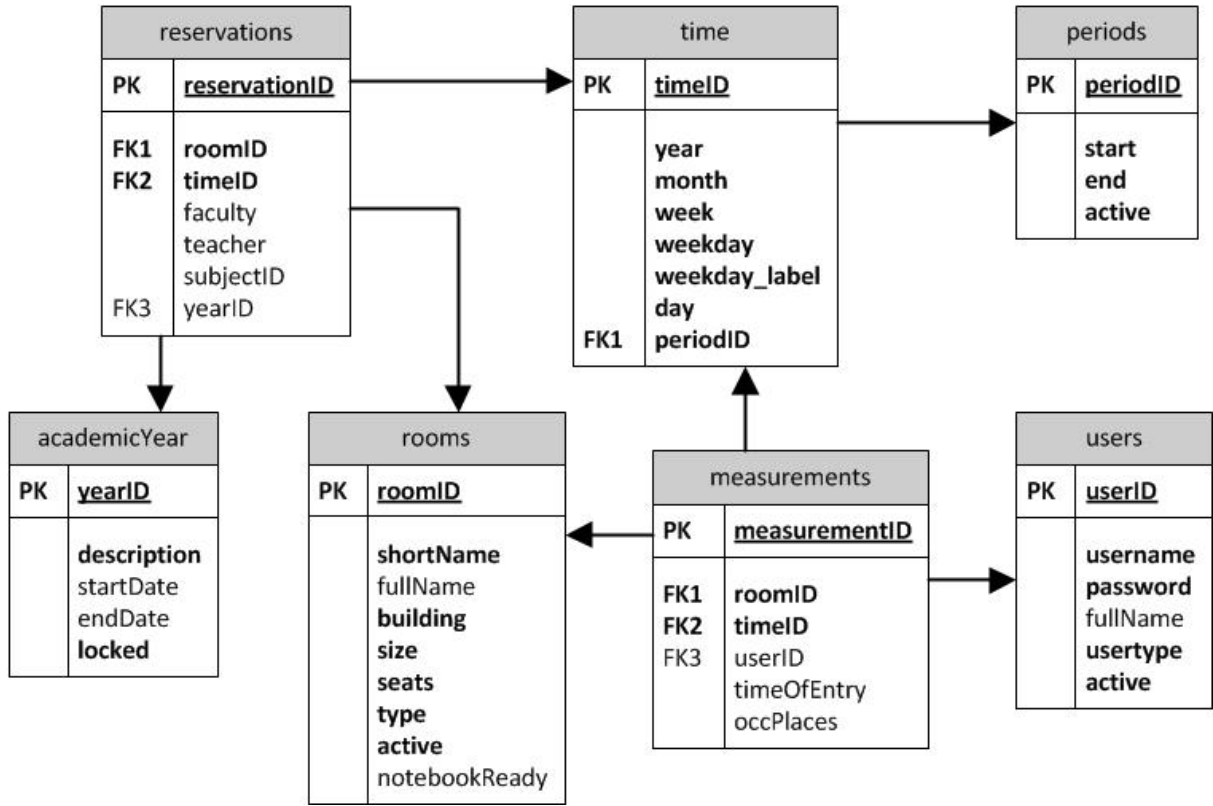


Figure 4.1: ER model of the relational database

4.2 ROLAP

On top of the relational database described in the previous section, we need a data warehouse with OLAP functionality to facilitate the report functionality of the system. According to Chaudhuri and Dayal [5], data warehousing is a collection of decision support

technologies, aimed at the knowledge worker (executive, manager, analyst) to make better and faster decisions. The report section of ROOMS should also provide decision support to the DIZ executives to make better and faster decisions with respect to timetabling and room management.

Data warehouses are typically maintained separately from the relational database(s). It supports on-line analytical processing (OLAP), while relational databases support on-line transaction processing (OLTP). OLTP databases are characterized by short transactions such as UPDATE, DELETE or INSERT. It focusses on the typical day-to-day operations in most organizations. In OLAP on the other hand, historical, summarized or aggregated data is more important than detailed, individual records. Therefore the size of an OLAP database can become quite large. Accessing an OLAP database usually involves more complex queries, up to millions of records and perform lots of joins. Because OLTP or relational databases are designed and tuned to perform short transactions, trying to execute OLAP queries against it would probably result in unacceptable performance. Other reasons to separate these type of databases is that OLTP supporting databases only store current data and OLAP databases require historical data. Finally, supporting OLAP functionality often requires a special data organization which is not compatible with the data organization of an OLTP targeted database.

To facilitate the complex analyses of OLAP, the data in a warehouse is modeled multi-dimensionally. For example in a sales warehouse, time-of-sale, place-of-sale or sold products might be some interesting *dimensions*. Every dimension is described by a set of *attributes*, these attributes can be hierarchical organized. A good example of a *hierarchical dimension* is time. As we have seen in the previous section, we have split up timestamps in different attributes in the time table according to its hierarchy: year > month > week > day. The time dimension is of particular significance to decision support, for example in trend analysis. The multidimensional data structure of OLAP is often visualized as a cube, as we can see in Figure 4.2.

Once we have our multidimensional model or *cube*, we need a set of numeric *measures* that will be our object of analysis. In our previous example this might be the turnover, revenue or inventory. All the dimensions together uniquely determine the measure, or in other words, the measure is a value in the multidimensional space.

A distinctive feature and key operation of OLAP is the *aggregation* of measures by one or more dimensions. For example the total turnover of a product, of a region or combination of both. Comparing two measures is also possible, as long as they are aggregated over the

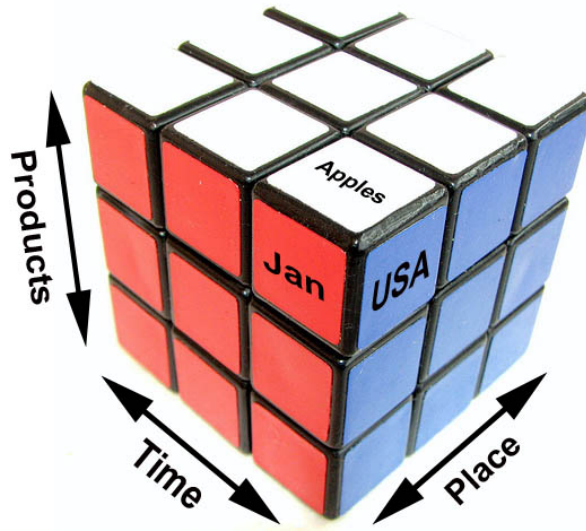


Figure 4.2: Cube Example

same dimension. So in our example you can, for example, calculate product revenue by subtracting product cost from product turnover.

Another important feature or operation is pivoting. An analyst might want to view the data, or ‘pivot’ the cube, in various ways. For example, if one wants to display all the regions on one axis and all the products on the other. If the analyst wants a different view, the cube can be re-oriented in the desired position. The resulting view is a grid where each value with a (x, y) coordinate corresponds to the aggregated value of the measure, where x is the value of the first dimension and y the value of the second. If we once again use our example, we can use products on the one dimension and regions on the other. The point (x, y) will then represent the aggregated sales of product x in region y .

Operators related to pivoting are *roll-up* and *drill-down*. The roll-up operation corresponds to doing a (further) group by on one of the dimensions of the measure. Drill down (and up) lets you go down in the hierarchy levels of a dimension. An example is shown in Figure 4.3. In this figure we see a cube with the sales figures of a large fruit company. The cube has three dimensions, *time*, *place* and *product*. The drill down operation drills down the hierarchy of the product dimension to another level. We now have a better understanding of the sales figure of the apple product, since we now see the sales figures of the different apple varieties. The drill up operation is the inverse of the drill down operation. *Slice* and *dice* operations correspond to taking a projection of the data on a subset of dimensions for selected values of the other dimensions. An example of the slice operation can be found

in Figure 4.4 and an example of the dice operation can be found in Figure 4.5. We see that in Figure 4.4 the time dimension is restricted to a single year and in Figure 4.5 we see that the product dimension is restricted to a subset.

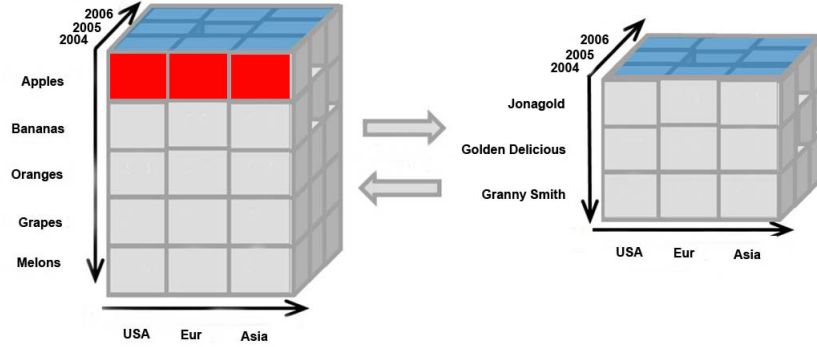


Figure 4.3: OLAP Drill Down/Up Example

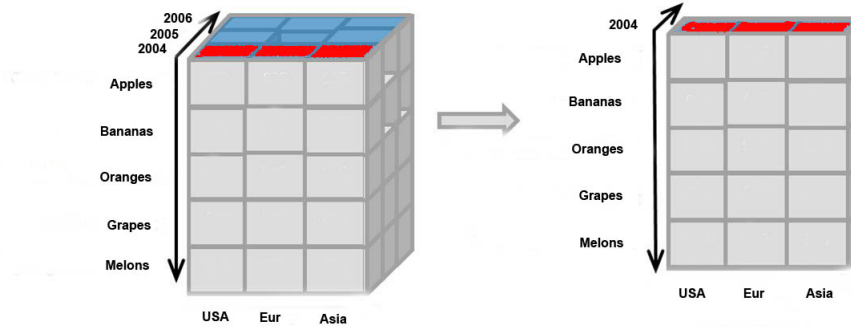


Figure 4.4: OLAP Slice Example

Data warehouses implemented on top of standard relational databases, as we intend in this project, are called Relational OLAP (ROLAP) servers. On the other side multidimensional OLAP (MOLAP) servers, are servers that store multidimensional data in own specialized data structures. ROLAP servers support extensions to SQL, while MOLAP servers implements operations over its special data structures.

Now that we have seen the architecture of ROLAP, we can apply it to our own project. Looking at requirements REP-01 to REP-06, we are going to need at least three measures. First the *occupation* measure, which counts the number of occupied rooms. Second, the *utilization* measure, which takes the average number of seats of the occupied rooms. Finally, we need to measure the amount of *reserved* rooms. All these measures will depend on

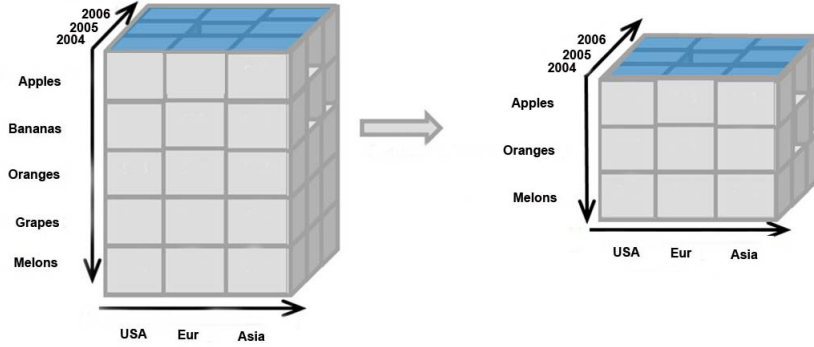


Figure 4.5: OLAP Dice Example

the dimensions *time* (requirement REP-01 to REP-06) and *rooms* (requirement REP-09). The time dimension will be hierarchical and will have at least the attributes *week*, *weekday* and *period*. The room dimension will consist of the non-hierarchical attributes *room type*, *room size*, *room building* and *room name*. The time, room and measures dimension will form a cube.

4.3 Architecture

Now that we have seen how we store our data in the fact tables in the relational database with a data warehouse on top, we need to design a code framework to tie everything together. As we have seen in Chapter 3 most of the workload of the code framework (excluding data warehouse) will involve CRUD operations. The Data Access Object (DAO) design pattern [8] helps an application to perform various CRUD operations on databases. DAO classes provide methods for inserting, deleting, updating and finding desired data in the storage medium. The advantages of using DAO classes are the loose coupling and less code repeating. Almost every part of the system needs to perform CRUD operations on the relational database, this would lead to repeating the same code fragments. It is needless to say that this repeated code fragments would not be easy to maintain if the interaction with the database would change.

DAO was invented to overcome the problems mentioned above. The DAO pattern is supposed to have a DAO interface, DAO class and DAO factory corresponding to each table in the database. Loose coupling arises through the simple separation of business logic and persistence logic. Business logic can still depend on the same DAO interface,

while changes in the persistence logic can take place in the DAO classes.

The DAO pattern is often used in combination with the DTO pattern [9]. Data Transfer Objects, sometimes also called Value Objects, are used to transfer data between different subsystems. Only one single method call is used to retrieve the object, instead of numerous remote calls to retrieve single attribute values. Usually more than one attribute from an object is required, so the number of remote calls can be significantly reduced in this way. The business logic receives the complete Transfer Object and can then use its getter and setter methods to retrieve its properties. A DTO does not have any extra behavior except for storage and retrieval of its own data. They are transferred to the business logic by value, so all calls to the instance are local.

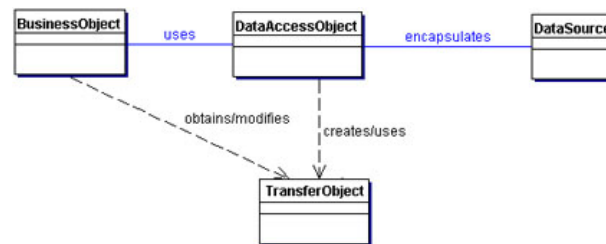


Figure 4.6: Data Access Object

In Figure 4.6 we can see how the DAO and DTO design patterns relate to each other. In Figure 4.7 we see a sequence diagram of a typical scenario of interaction between the DAO and DTO design patterns. The controller wants to update a user by setting its new name and address. First the controller creates a new user Data Access Object **UserDAO** (step 1). The DAO is responsible for talking to the database and retrieves the specific user (step 2.1). Instead of returning the individual user attributes to the controller, a Data Transfer Object (**UserDTO**) is created by the DAO (step 2.2). With this DTO the controller can set or get every user attribute it wants (step 3 & 4), in this case the user's name and address. When finished, it sends the **UserDTO** back to the **UserDAO** (step 5). The DAO now stores the user data in the database (step 5.4) using the just adjusted UserDTO (step 5.1, 5.2 & 5.3).

Now that we have abstracted our data source form the business logic, we also need to abstract the user interfaces from the underlying data. Here comes the Model View Controller (MVC) design pattern into play [12].

- **View:** displays information to the user and provides a mechanism to interact with the system. Should not check, process or calculate data.

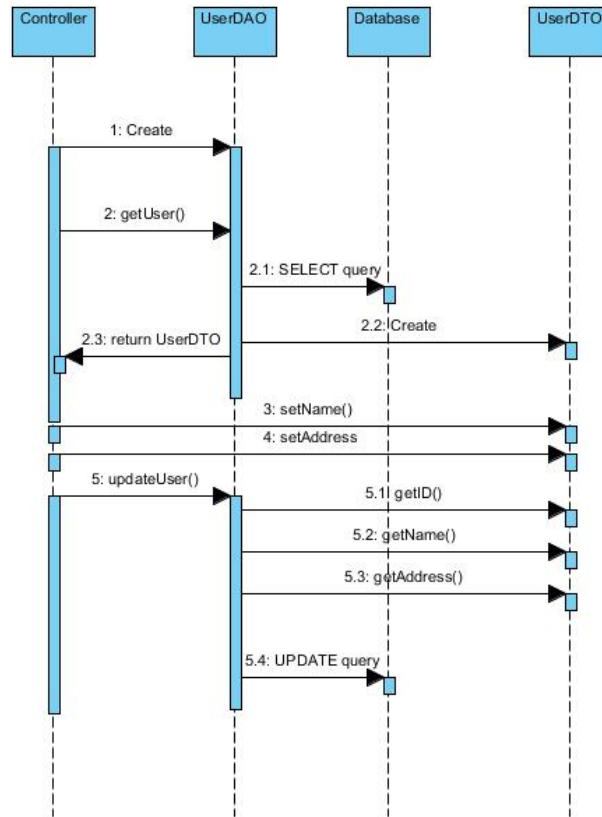


Figure 4.7: Data Access Object sequence diagram

- **Controller:** processes events caused by user interaction
- **Model:** part of the application that both contains the information shown by the View and the logic that changes this information based on the events received. The actual storage of data is done in the persistence layer of the application, in our case a RDBMS. The persistence layer is not necessarily part of the MVC pattern. We have already seen how to transport data from and to the Model using the DAO and DTO patterns.

The MVC pattern makes its easier to develop and maintain the system. The look and feel of the application can now change and evolve without having to make changes to data structures or business logic. The application is also able to maintain several interfaces such as multiple languages.

Because web-applications are interactive software systems, they can really benefit from the MVC design pattern. A possible problem for using MVC in web-applications might

be that web-applications are partitioned by server and client by nature. The View is always shown on the client side, but the Controller and Model can, theoretically, be placed at either side. The developer is forced to make a partition, while the MVC design pattern should be partition-independent, meaning Model, View and Controller should reside in the same address space. Location dependent issues should not drive architecture design or influence design decisions. However, since we really want to avoid partitioning and portability being one of our important features, there is only one solution possible. Controller and Model should reside on the server side, this is also known as the ‘thin client’ variant. This results in minimal customization on the client side. The downside is that for every action a request to the server is needed, but this should not cause any problems since the traffic to our system is expected to be rather limited. Only a few users will work on the system at once.

4.4 Data import

Entering and managing data about rooms and timetables can be a cumbersome process. Luckily, this data is already available from another system at DIZ, Syllabus Plus. Syllabus Plus is a worldwide used software package for planning, timetabling and resource management in higher professional education. It is developed by Scientia and is used at over 450 universities worldwide (dated 2006) [1]. The most important application of the Syllabus Plus package is the Timetable application. With this application educational timetables can be produced. Because multiple Timetable applications can be connected with a central Scientia Database (SDB), different timetable coordinators can work at the same time while keeping data consistent. The SDB hosts all data relevant to timetabling. Think of teachers, faculties, rooms, courses, etc. From this Timetable application we can extract all kinds of data from the SDB we need for ROOMS. Data we need is all the rooms associated to DIZ and the timetable of all the activities taking place in these rooms [1].

Syllabus Plus allows to create templates for special reports and output files. To print a data report, or save data to an output file in a certain way a *printing template* is used. It goes beyond the scope of this report to describe in detail how to construct these templates. But it is fair to say that is possible to extract available data in any format desired. Two new templates have been constructed, one to extract rooms and one to extract timetables.

4.4.1 Rooms import

Ideally, we would like to fill our room table in the relational database completely from Syllabus. Unfortunately, no building name, room type (Amphi or Flat) or notebook readiness is currently available from Syllabus. That leaves us with the following remaining attributes to import. Every bullet mentions a database field from the rooms table to be imported, followed by the matching Syllabus field in brackets:

- shortName (Zalen.naam)
- fullname (Zalen.beschrijving)
- seats (Zalen.capaciteit)
- size (calculate from seats)

When importing rooms the remaining table fields (those not mentioned above) will be set to their default value. The default room type value will be ‘Flat’. Imported rooms are also set to active and notebook ready. Changes to these attributes can be made afterwards manually. We will use *shortName* as a unique identifier (next to its primary key *roomID*). If a room is imported with a *shortName* that already exists in the database, the attributes *fullname*, *seats* and *size* are updated instead of added as a new row. The remaining attributes of the updated row will be left untouched.

4.4.2 Timetable import

Luckily, the data we need to fill our reservation table is completely available from Syllabus. We can make the following matching with the SDB:

- roomID (Activiteiten.Zalen.Naam)
- timeID (Activiteiten.activiteitsdatums + Activiteiten.Geroosterde begintijd + Activiteiten.Geroosterde eindtijd)
- faculty (Activiteiten.Afdeling.Naam)
- teacher (Activiteiten.Docent.Naam)
- subjectID (Activiteiten.Module.Sleutelveld)

As we can see above we have to derive the *roomID* from Activiteiten.Zalen.Naam as we did in the previous section. If the *roomID* can not be found in the rooms table based on

the data gathered from Syllabus, we will discard the activity altogether. We only want the activities of the rooms we are interested in. So it is wise to first import the rooms before importing the timetable. The timeID can be retrieved using the date, start and end time of the activity in Syllabus. If we can not find the corresponding time entry, we have to add it to our time table. In this way we only store the time entries we need. The rest of the attributes can be copied one on one.

4.5 Environment

Finally we need to create an environment to support the architecture described in this chapter. Since the software needs to be web-based, we will at least need a web server. Since the implementation of the system cannot count on a high budget, we need to be as creative as possible. The most limiting factor is finding the right software for our data warehouse. There are a lot of commercial products available on the market, but most of them are rather expensive. A popular and open source alternative is called Mondrian, also known as Pentaho Analysis Services Community Edition [2].

Another popular and widely used implementation of RDBMS is MySQL. It is easy to use, has a good performance and is reliable. MySQL runs on more than 20 platforms including, Linux, Windows and MacOS. The world's largest and fast growing companies such as Google and Facebook use MySQL in many of their products [11].

Because Mondrian is written in 100% Java and requires a web container anyway, it seems wise to also chose Java as our programming language. JavaServer Pages (JSP) technology provides a simplified, fast way to create dynamic web content. JSP technology enables rapid development of web-based applications that are server- and platform-independent [4].

Chapter 5

Implementation

Now that we have outlined our architectural foundation discussed in Chapter 4, we can fill in the implementational details. First we will have a look at the code implementation based on the Struts framework. The accompanying UML package diagram can be found in Appendix B and UML Class diagrams of the individual packages can be found in Appendix C. Next we will have a look at the implementation of our databases. MySQL is picked as our relational databases, with Mondrian as open source OLAP server on top.

5.1 Struts

Jakarta Struts is an open source framework, which was designed to make it easier for web developers to build web applications based on Java Servlets and Java Server Pages (JSP). Developers can use this as a solid framework to base the rest of their design upon. They can shift their focus more towards business logic design instead of infrastructure design. The Struts framework is one of many well-known and successful Apache Jakarta projects. It was created by Craig R. McClanahan and donated to the Apache Software Foundation in 2000. The mission of the project nowadays is to provide a commercial-quality server solutions in an open and cooperative fashion [4].

A Java Servlet is a Java class in Java EE that conforms to the Java Servlet API, a protocol by which a Java class may respond to requests. Java Servlets provide a component-based, platform-independent method for building web applications. Because they are written in Java they are not bound to a platform or operating system. Java also comes with an entire suite of application programming interfaces (APIs). As we have noticed in Section 4.5,

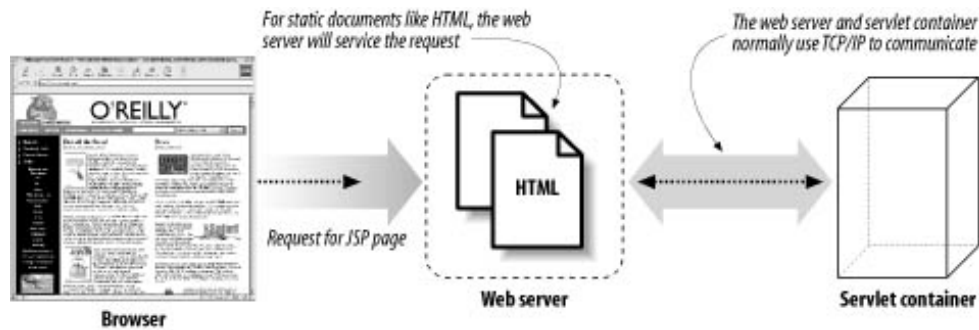


Figure 5.1: Struts client request

servlets are not executed on the web server itself. They need a web container. In Figure 5.1 we see how the web server and the web container cooperate in handling client requests [4].

One of the disadvantages of Servlets was the fact that adding HTML text to a Servlet as the output of a request has some limitations. Java Server pages were introduced to overcome these limitations. Java Server Pages are a natural extension to the Java Servlet technology. JSP files (which have .jsp extension) are text files containing HTML mixed with XML-like tags and scripts. The tags and scripts are the logic that generate the page content. After preprocessing and Java compilation, the files can be executed by the Servlet container. We will be using JavaServer Pages to create our user interface.

As we can see in Figure 5.2, the Struts architecture uses a MVC approach. There is a clear separation of business logic, presentation and request processing. The views within the MVC pattern typically consist of HTML and JSP pages. HTML is for static content, JSP for dynamic and static content. Almost all of the dynamic content is generated in the web tier with the exception of some client-side Javascript. Javascript is used to sort tables on the client side or to generate warning and confirmation messages. The controller in our MVC approach is a Java Servlet. It takes care of the following duties:

- Intercepts HTTP requests from a client
- Translates each request to business operations
- Helps to select the next view to display to the browser
- Returns the view to the browser

Because all traffic goes through the controller, we have a central point where we can control the application. If we need to change or add functionality to the way client requests are

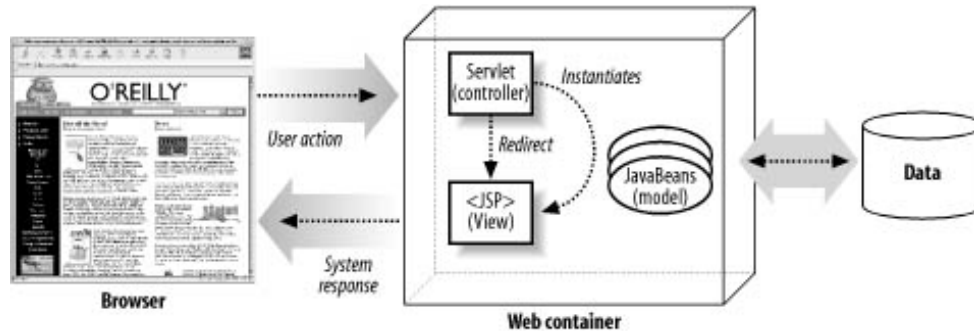


Figure 5.2: Struts Architecture

handled, we can do that at a single point instead of changing several or all JSP pages. We already saw that the controller plays a crucial part in decoupling the presentation from the business logic.

5.2 ROOMS

Now that we have studied the Struts framework the ROOMS application is based on, we can have a look at the application itself. In Appendix B we can find a package diagram of the application. The following packages are present:

- **nl.tue.diz.rooms**: Central package of the application.
- **nl.tue.diz.rooms.action**: Package which contains the controller servlets.
- **nl.tue.diz.rooms.forms**: Package with form beans for every form used in the application. Most of the user interaction goes through forms.
- **nl.tue.diz.rooms.DTO**: Package with Data Transfer Objects for every table in the database.
- **nl.tue.diz.rooms.persistence**: Package with Data Access Objects for every table in the database.
- **nl.tue.diz.rooms.constants**: Contains single class where constants used across the application are defined.

5.2.1 Controller

The controller is responsible for translating user input into actions to be performed by the model. Based on this user input and the output of the model, the next view is determined. The controller is implemented by a Java Servlet, the centralized point of control for the application. The controller is implemented by the `ActionServlet` class. All incoming requests are mapped to the central controller in the deployment descriptor. Java web applications use a deployment descriptor file to determine how URLs map to Servlets. This file is named *web.xml* and can be found in Appendix E. The following snippet is responsible for mapping actions to the `ActionServlet` class:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

The `servlet-mapping` tag makes sure that all URLs with the pattern `*.do`, are mapped to this `ActionServlet`. The class `requestProcessor` will further process the request. Once it receives a request it delegates it to a helper class. In the Struts framework these helper classes are (or descendants of) `Action` classes. They function as a bridge between user actions and business operations. Next we need to create a one-to-one mapping from user requests (URIs) to `Action` classes. This is done in the struts configuration file, *struts-config.xml*. This file is included in Appendix D. Below you can find a single mapping to illustrate how this works:

```
<action-mappings>
    <action
        path="/login"
        name="loginForm"
        type="nl.tue.diz.rooms.action.DIZLoginAction"
        scope="request"
```

```

        parameter="dispatch">
        <forward name="success" path="/index.jsp" redirect="true" />
        <forward name="failure" path="/login.jsp"/>
    </action>
</action-mappings>

```

The above action mapping example illustrates the login functionality in ROOMS. If the login form (called `loginForm`) is sent to the controller via the URL `http://rooms.diz.tue.nl/login.do` the `DIZLoginAction` `Action` class is called as a helper class to further process the request. This class will call the necessary business operations from the model and determine if the user is authorized to log in or not. If he or she indeed is, the forward action "success" is chosen by the controller and the client is redirected to the main menu in the *index.jsp* view. Otherwise the user is directed back to the login form i.e., *login.jsp* page, possibly with some error message indicating the authorization problem.

Also notice that in the example above we also send an extra parameter `dispatch` along with the request. This is because instead of the regular `Action` helper classes, the `DispatchAction` class is used which is a standard built-in action that provides a mechanism to collect related functionality in a single action. This way we don't need to create single independent actions for every function. The creation, update and deletion of a user can then be captured in a single `Action` class, ie `DIZUserAction`. The collection of `Action` classes used in the ROOMS application can be found in the action package shown in Appendix B. The UML class diagram of the action package can be found in Appendix C Figure C.1. The following `Action` classes are defined:

- **DIZAcademicYearAction**
- **DIZLoginAction**
- **DIZMeasurementAction**
- **DIZPeriodAction**
- **DIZReportAction**
- **DIZReservationAction**
- **DIZRoomAction**
- **DIZUserAction**

The names of the `Action` classes speak for themselves. Every `Action` class is responsible for the request coming from its own section as described in Section 3.1. The `DIZLoginAction`

is an extra class to handle log in and log out requests, this also could have been integrated in the `DIZUserAction` class but has its own class for separation of concerns. The `DIZAcademicYearAction` is called to life for the reasons explained in Section 3.1.4.

5.2.1.1 Forms

Forms, or `ActionForm` objects are used in the Struts framework to pass data back and forth from the user to the controller. The framework automatically collects the input from the request and passes this data to an `Action` using a `JavaBean`. A `JavaBean` is used to encapsulate many objects into a single object, so that they can be passed around as a single bean object instead of multiple individual objects. To keep the presentation layer decoupled from the business layer, we should not transfer the form bean to the business layer, but we should create a DTO using the form data instead. For every form in the view an `ActionForm` should be used. The same `ActionForm` can be used for multiple pages if necessary, as long as the form fields and `ActionForm` properties match. The `ActionForm` used in ROOMS can be found in the forms package. The forms package can be found in Appendix C Figure C.4. An `ActionForm` class should have all the form fields as properties. All properties must have type `String`. The properties can be retrieved or set using getter and setter methods.

5.2.2 Model

Now that the controller is set up to process user requests into business operations, we can have a look at how these business operations are implemented in the model. As we have seen in Section 4.3, we are using the DAO and DTO design patterns to structure our model. However, it is not ideal to connect the `Action` classes directly to the Data Access Objects. DAOs are designed to perform atomic actions such as insert, update and delete. There might be more complex business operations required. Therefore we introduce an intermediate class between the Struts `Action` class and the persistence layer, implemented by the Data Access Object. In this so called *service* class we do our business logic before we hand off our Data Transfer Object to the persistence layer.

5.2.2.1 Data Transfer Object

Let us first have a look at the Data Transfer Object or Value Object since these are used to transfer data (in)between the Controller and Model. Data Transfer Objects simply map a Java class to a database table. For every field in the table, there is a (private) field in the DTO class. Every field can have its own type, even the type of an other DTO. To set and retrieve these fields, getter and setter methods are created for every field. This is the only functionality a DTO class is allowed to have, no extra behavior is defined. As can be seen in Appendix C the following DTOs are defined:

- DIZAcademicYear
- DIZMeasurement
- DIZPeriod
- DIZReservation
- DIZRoom
- DIZTime
- DIZUser

5.2.2.2 Data Access Object

Now that we have defined our DTOs, we can implement our DAOs, to retrieve and send the data objects to and from the persistence layer. We have defined 7 DAO interfaces, again one for each table.

- DIZAcademicYearDAO
- DIZMeasurementDAO
- DIZPeriodDAO
- DIZReservationDAO
- DIZRoomDAO
- DIZTimeDAO
- DIZUserDAO

These interfaces define the basic CRUD operations we need. Every interface is implemented for our current storage medium MySQL. So for every interface `DIZTableDAO` we have an implementing class `DIZTableDAOMySQL`. These implementing classes extend the base class `DIZBaseDAOMySQL`. The base class starts a MySQL session, using an Object-relation mapping (ORM) framework called MyBatis. We will have a closer look at MyBatis in Section 5.3.1.

5.2.2.3 Service

To tie up loose ends, we need to create an extra layer between the Controllers' `Action` classes (Controller) and the Data Access Objects. This will be the Service layer. This layer is intended to perform the more complex business operations in the model. The service layer will break down the operations to simple CRUD operations and calls the DAO layer. In the current status of our project, the operations in the service layer are not much more complicated than the CRUD operations in the DAO layer. But for future reference it might be a good idea to create an intermediate service layer for flexibility. If for example, we need to immigrate the Controller layer to a completely different framework, we do not need to refactor all the DAO calling code, since it can remain in the service layer.

We introduced 8 service interfaces, one for each part of the system and extra services for academic years and security (log in and log out) for separation of concerns:

- `DIZAcademicYearService`
- `DIZMeasurementService`
- `DIZPeriodService`
- `DIZReservationService`
- `DIZRoomService`
- `DIZSecurityService`
- `DIZTimeService`
- `DIZUserService`

Every service interface `DIZConcernService` is implemented by the class `DIZConcernServiceDAO`. This class codes the business logic with calls to other services and corresponding Data Access Objects where appropriate. For example in Figure 5.3 we see an example of a typical service sequence. In this example we see the implementation of the method `getCurrentDIZTime()` from `DIZTimeServiceDAO` which is called from the controller. This method requests the current time object (`DIZTime` DTO) corresponding to the current timestamp. First the current period is requested from the period service. Next the date is retrieved from the server. This date and period are then translated to a `DIZTime` object. Then the time service calls the `DIZTimeDAO` to find the current time object in the database. If not found (`result == null`) it will insert it using the DAO again and tries to retrieve it again to obtain the `DIZTime` DTO including its unique primary key generated by the database.

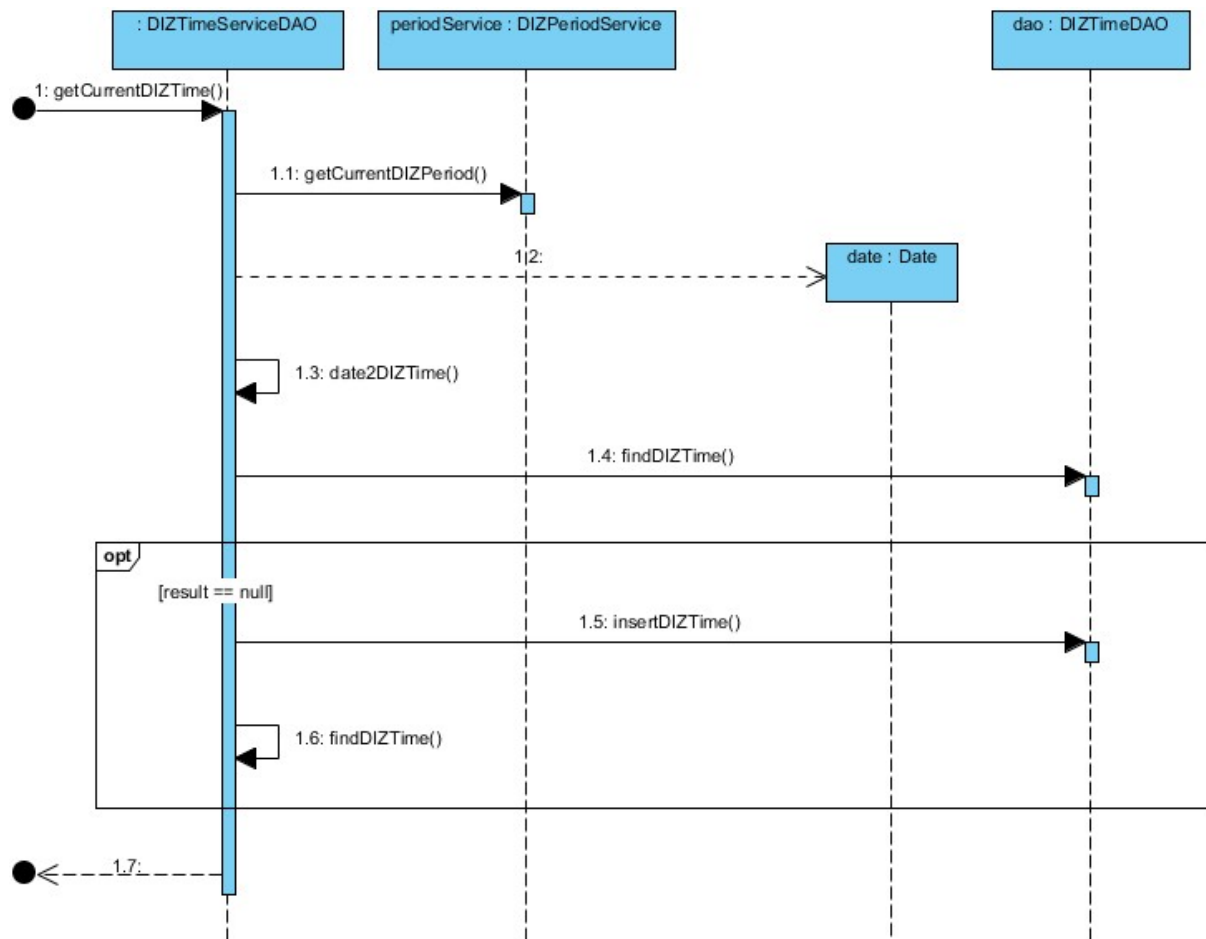


Figure 5.3: Example Service Sequency: `getCurrentDIZTime()`

Both the service and the persistence package are written using interfaces. Interfaces define a contract between two classes and only a contract, nothing more. It helps in making explicit how classes are coupled together. It also helps in separating out concerns. By writing an interface for every concern before writing the implementing class, size and complexity can be reduced. In the persistence package especially, if the storage medium is changed, an alternative implementation for this medium can then easily be added or placed instead of the old implementation while the contract remains the same. In general, programming interfaces is low cost and have great advantages that go beyond polymorphism.

5.2.3 View

The last part of the MVC pattern we still have to address is the view. The components we typically encounter in a Struts view are HTML, JavaServer Pages, DTOs and Struts `ActionForms`. Screenshots of the various screens in ROOMS can be found in Appendix F. An aspect of JSP pages worth while looking further into are JSP tag libraries.

5.2.3.1 JSP tag libraries

The Struts framework provides six core tag libraries that applications can use. Each one has a different purpose and can be used individually or alongside others. You can also create your own custom tags when additional functionality is required. The custom tag libraries that are included with the framework are the *HTML*, *Bean*, *Logic*, *Template*, *Nested*, and *Tiles* tag libraries. Only the first three are used by the ROOMS application.

Some examples of tasks that can be performed by custom tag actions include form processing, accessing beans and using iterations over data sets. Before the availability of custom actions, JavaBeans components in conjunction with scriptlets were the main mechanism for performing such processing. The disadvantage of using this approach is that it makes JSP pages more complex and difficult to maintain. JSP tag libraries declare modular functionality so that any JSP page can reuse it. Tag libraries reduce the need to embed large amounts of Java code in JSP pages by moving the functionality of the tags into tag implementation classes.

You declare that a JSP page will use tags defined in a tag library by including a *taglib* directive in the page before any custom tag is used:

```
<%@ taglib uri="/tlt" prefix="tlt" %>
```

The `uri` attribute refers to a URI that uniquely identifies the tag library. This URI can be relative or absolute. The `prefix` attribute defines the prefix that distinguishes tags provided by a given tag library from those provided by other tag libraries.

JSP custom actions are expressed using XML syntax. They have a start tag and end tag, and possibly a body:

```
<tlt:tag>
    body
</tlt:tag>
```

A tag with no body can be expressed as follows:

```
<tlt:tag />
```

5.3 MySQL

The MySQL relational database has become the most popular open source database of the world. It is easy to use, has a good performance and is reliable. MySQL runs on more than 20 platforms including, Linux, Windows and MacOS. Worlds largest and fast growing companies as Google and Facebook use MySQL in many of their products. MySQL was originally founded and developed in Sweden by two Swedes and a Finn: David Axmark, Allan Larsson and Michael Widenius [11].

MySQL is written in C and C++, but many of the programming languages include libraries for accessing MySQL databases. Since we are using Java, we will be using the Java Database Connectivity (JDBC) interface. The translation of the ER Model given in Section 4.1 is pretty straightforward. The MySQL structure dump file can be found in Appendix G.

5.3.1 MyBatis

As we have seen, the persistence layer of the code framework (DAOs) talks to the actual storage medium, the MySQL database. An easy way to facilitate this is by using an Object-relation framework (ORM). MyBatis is one of the open source solutions of an (sort

of) ORM. Unlike most ORMs it does not map the object model to the relational database, but it maps objects to SQL statements. It makes it easy to use relational databases with object-oriented applications such as ROOMS. Simplicity is its biggest advantage. Often, only one line is enough to execute a SQL statement. It saves time and prevents errors.

The architecture of the MyBatis framework is depicted in figure 5.4. SQL statements are stored in XML files. An XML mapper file was created for every table (and therefor DAO). Let's look at an example, the measurement mapper *measurementMapper.xml*:

```
<resultMap id="measurements" type="DIZMeasurement">
  <id property="measurementID" column="measurementID"/>
  <result property="occPlaces" column="occPlaces"/>
  <result property="timeOfEntry" column="timeOfEntry"/>
  <association property="room" column="measRoomID" javaType="DIZRoom"
    resultMap="rooms"/>
  <association property="time" column="measTimeID" javaType="DIZTime"
    resultMap="time"/>
  <association property="user" column="measUserID" javaType="DIZUser"
    resultMap="users"/>
  <association property="reservation" javaType="DIZReservation"
    resultMap="reservations"/>
</resultMap>

<select id="getMeasurement" parameterType="int" resultMap="measurements">
  SELECT * FROM measurements AS m
  LEFT JOIN (rooms as r, time as t, users as u) ON
    (m.roomID = r.roomID AND m.timeID = t.timeID AND m.userID = u.userID)
  WHERE m.measurementID = #{id} LIMIT 1
</select>
```

At the bottom of the example we see a simple select statement with an incoming parameter `#{id}` of type `integer`. The output is a result map of type `measurements`. This result map is defined on top. There the DTO `DIZMeasurement` is mapped onto the query result. Inside the `resultMap` we find an `id` element, explicitly identifying the `id` which improves the overall performance. The `result` tags are normal results directly injected into DTO properties. The `association` tags indicate complex type association with other

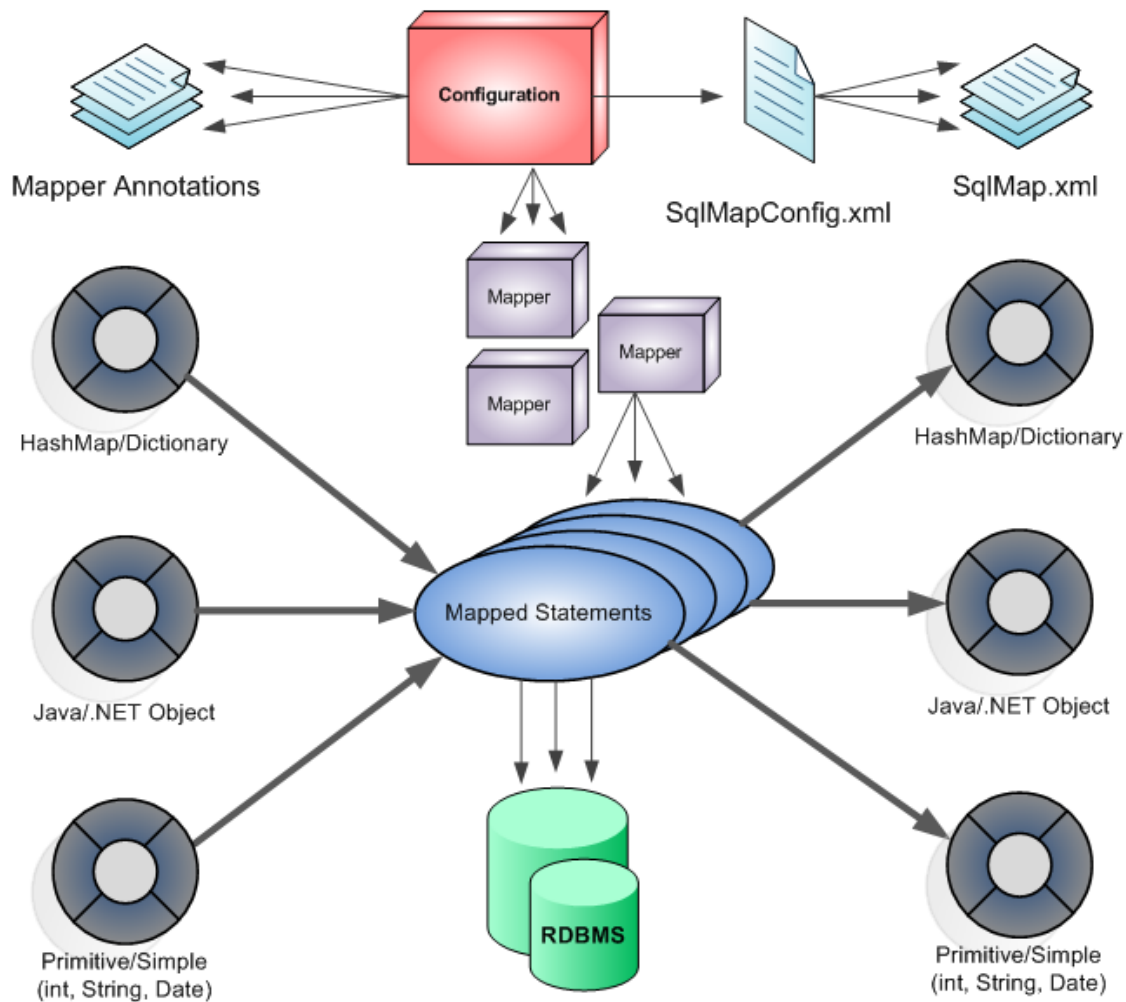


Figure 5.4: MyBatis Architecture

resultMaps. In this way DTOs containing other DTOs can be mapped to directly without any supplementary code.

DAOs can call various SQL statements, using a MyBatis SQL Session, by its id ("get-Measurements" in the example). The result of the queries will be, thanks to the result mappers, DTOs which can then be transported directly to the service layer. Not in every case a DTO as output is desirable. MyBatis can also output basic Java types.

5.4 Mondrian

There are a lot of commercial software OLAP products on the market. Big names are Microsoft SQL Server Analysis Services and Hyperion, which had more than half of the market share together in 2006 ¹. Since there were not enough funds available to purchase a commercial OLAP product and because there are good open source alternatives we chose for the latter. Probably the most popular open source OLAP solution is Mondrian.

Mondrian, also known as Pentaho Analysis Services Community Edition is an OLAP server written in 100% Java. It executes queries written in MDX query language, reads its data from a relational database and presents it in a multidimensional format through a Java API. The Mondrian OLAP system architecture, as we can see in Figure 5.4 consists of four layers, these are: the presentation layer, the dimensional layer, the star layer, and the storage layer.

The first one, the *presentation layer* determines what the end-user will eventually see on his or her screen. There are many ways of presenting the multidimensional data that is returned from the server. You can use pivot tables, charts, and more advanced visualization tools such as JPivot. We will see more on JPivot in Section 5.4.4.

The second layer is the *dimensional layer*. The dimensional layer parses, validates and executes MDX queries. We will have a closer look at MDX queries in Section 5.4.3. An MDX query is evaluated in multiple phases. The axes are computed first, later the values of the cells within the axes.

The third layer is the *star layer*, and is responsible for maintaining the aggregate cache. An aggregation is a set of measure values, which are as we have seen in Section 4.2, determined by a set of dimension values. The dimensional layer sends requests for sets of cells. If the requested cells are not in the cache, or derivable by rolling up an aggregation in the cache, the aggregation manager sends a request to the storage layer.

The final layer, the *storage layer*, is the relational database (RDBMS). It is responsible for providing the aggregation data and for providing data about dimensions. Mondrian uses RDBMS as its storage manager and aggregated data is read by submitting *group by* queries to the RDBMS. If the relational database system supports materialized views and these views are created for particular aggregations, Mondrian will use them explicitly. The general idea is to delegate as much as possible to the database itself [2].

¹<http://www.1keydata.com/datawarehousing/olap-market-share.html>

Pentaho Analysis Services: Mondrian Project Architecture

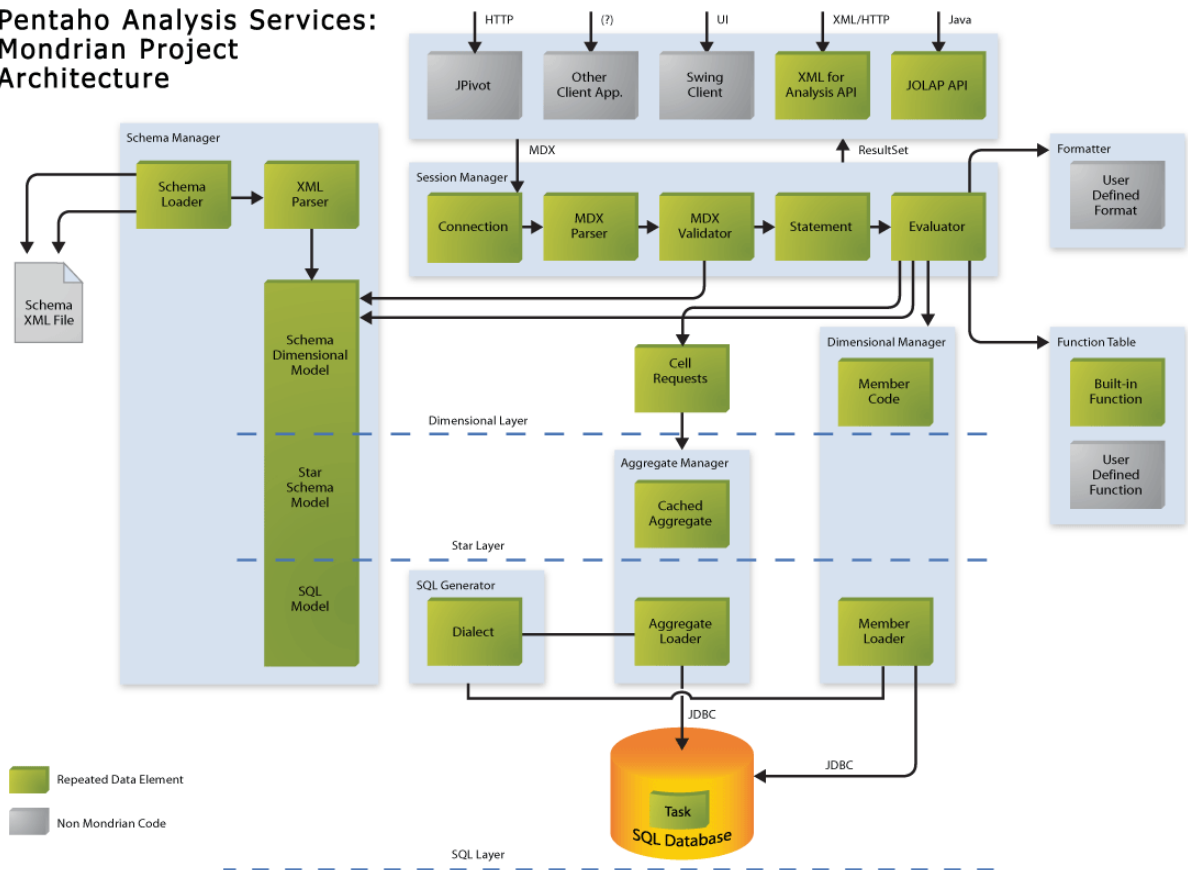


Figure 5.5: Mondrian Architecture

5.4.1 Mondrian Schema

A Mondrian schema defines a multi-dimensional database. It consists of two models, a logical model and a mapping of this model onto a physical model. The logical model will consist of cubes, dimensions, hierarchies, levels and members. The physical model is the source of the data, in our case a star schema of the relations database as presented in Figure 4.1.

Mondrian schemas are presented in XML. The most important components of a schema are cubes, measures and dimensions:

- A cube is a collection of dimensions and measures in a particular subject area.
- A measure is a quantity that you are interested in measuring. (ie. unit sales of a product)
- A dimension is an attribute, or set of attributes, by which you can divide measures into sub-categories. (ie. color of the product)

A *cube* is a named collection of measures and dimensions. The fact table holds the columns from which the measures are calculated and holds references to the tables of the dimensions. A cube can have several measures. A *measure* has a column in the fact table and an aggregator. The aggregator can be, among others, a summation, average, minimum or maximum. For the sake of uniformity, a measure is a member of its own dimension called ‘Measures’.

Dimensions are also a set of hierarchies which discriminate on the same fact table attribute. For example the day a measurement took place. A hierarchy is a set of members organized in a structure for convenient analysis. For example a time hierarchy can consist of year, quarter, month, week and day. These positions in the hierarchy are also called levels. Members in a level have the same distance to the root of the hierarchy. A hierarchy allows you to easily calculate subtotals. The total sales of a quarter, are the sum of the sales of its months.

Now that we know the terms and definitions of the Mondrian schema, let us start with the dimensions we need for our measures. As discussed in Section 4.2 we need two dimensions, *rooms* and *time*. Let us start by defining the first dimension, rooms:

```
<Dimension type="StandardDimension" name="Rooms">
  <Hierarchy name="types" hasAll="true" allMemberName="All types" primaryKey="roomID">
    <Table name="active_rooms" alias="" />
  </Hierarchy>
</Dimension>
```

```

<Level name="roomType" column="type" type="String" uniqueMembers="true" />
</Hierarchy>
<Hierarchy name="sizes" hasAll="true" allMemberName="All sizes" primaryKey="roomID">
  <Table name="active_rooms" alias="" />
  <Level name="roomSize" column="size" type="String" uniqueMembers="true" />
</Hierarchy>
<Hierarchy name="buildings" hasAll="true" allMemberName="All buildings" primaryKey="roomID">
  <Table name="active_rooms" alias="" />
  <Level name="roomBuilding" column="building" type="String" uniqueMembers="true" />
</Hierarchy>
<Hierarchy name="names" hasAll="true" allMemberName="All names" primaryKey="roomID">
  <Table name="active_rooms" alias="" />
  <Level name="roomName" column="shortname" type="String" uniqueMembers="true" />
</Hierarchy>
</Dimension>

```

The dimension is named "Rooms" and is of type `StandardDimension`. A dimension's type may be one of `StandardDimension` or `TimeDimension`. Inside the dimensions tag we see a collection of hierarchies. The hierarchies are named, *types*, *sizes*, *buildings* and *names*. We see that all members relate to the primary key `roomID` to identify its members. Also they all have the `hasAll` attribute set to true. This means that every hierarchy is completed by an extra element which represents the whole group, as a grand total. This element is indicated by the name specified by the `allMemberName` attribute. Inside the hierarchy element we find a `<table>` element. This specifies the table where the hierarchy is build on. The `<level>` tag specifies which column in the table holds the unique identifier of this level. You can specify the type of the members inside the level and if they are all unique or not. The members of the `roomType` level are either 'Amphitheater' or 'Flat'. The members of `roomSize` are in the region A-E. The `roomBuilding` level hosts all distinct building names, while in the `roomName` level all rooms are distinguished by room name. All the hierarchies specified here only have one level, so are not much of a hierarchy. We will see more hierarchy in the time dimension:

```

<Dimension type="TimeDimension" name="Time">
  <Hierarchy name="week" hasAll="true" allMemberName="All Time" primaryKey="timeID">
    <Table name="active_time" alias="" />
    <Level name="Year" column="year" uniqueMembers="true" levelType="TimeYears" />
    <Level name="Week" column="week" uniqueMembers="false" levelType="TimeWeeks" />
    <Level name="Day" column="day" uniqueMembers="false" levelType="TimeDays" />
    <Level name="Period" column="periodID" uniqueMembers="false" levelType="TimeHours" />
  </Hierarchy>
  <Hierarchy name="weekday" hasAll="true" allMemberName="All Weekdays" primaryKey="timeID">
    <Table name="active_time" alias="" />
    <Level name="Weekday" column="weekday" ordinalColumn="weekday" nameColumn="weekday_label" uniqueMembers="true" levelType="TimeDays" />
    <Level name="Period" column="periodID" uniqueMembers="false" levelType="TimeHours" />
  </Hierarchy>
  <Hierarchy name="period" hasAll="true" allMemberName="All Periods" primaryKey="timeID">
    <Table name="active_time" alias="" />
    <Level name="Period" column="periodID" uniqueMembers="true" levelType="TimeHours" />
  </Hierarchy>
</Dimension>

```

This dimension is named "Time" and is of the type `TimeDimension`. This type will allow the use of the MDX time functions. Inside this dimension, three hierarchies are specified,

the *week* hierarchy, the *weekday* hierarchy and the *period* hierarchy. We need these three hierarchies to, be able to aggregate on these levels. We will see how this aggregation works in the next section about MDX. Because of this dimension being a time dimension, the levels can be specified as special time levels. You can indicate if a level is concerned with *TimeYears*, *TimeMonths*, *TimeWeeks*, etc. The `nameColumn` attribute lets you label ordinal columns for the sake of readability of the results.

Now that we have got our dimensions specified, we can define our measures. We already saw in Section 4.2 that the measures are uniquely determined by its dimensions. First we will need the measurements for room utilization and occupation. These measures depend on the two previous determined dimensions. We can then create our first cube in the following way:

```
<Cube name="ROOMS_meas" cache="false">
  <Table name="active_measurements" alias="" />
  <DimensionUsage source="Time" name="Time" foreignKey="timeID" />
  <DimensionUsage source="Rooms" name="Rooms" foreignKey="roomID" />
  <Measure name="Utilization" column="occPlaces" aggregator="avg" visible="true" formatString="#,##"/>
  <Measure name="Occupation" column="measurementID" aggregator="count" visible="true" />
  <Measure name="MaxSeats" column="seats" aggregator="avg" visible="true" formatString="#,##"/>
</Cube>
```

The cube named `ROOMS_meas` is defined to not cache the fact table in Mondrian, since the facts are likely to change often during measurement periods. The next element, the `<table>` element defines the fact table where the cube is based on. The fact table here is the `active_measurements` table which is a view in the MySQL database. The view is actually a join on the `measurements` and `rooms` table and will discard measurements of inactive rooms. What follows are the `<DimensionUsage>` tags that refer to our previously defined dimension. The `foreignKey` attributes point towards the foreign key fields in the fact table where the dimension should be matched on. Finally we can define our measures. The *Utilization* measure is based on the field `occPlaces` and has the aggregator *avg*. This means that when the cube is rolled up or drilled down/up, the result of the aggregated measure is the average of the `occPlace` field entries. The *Occupation* measure will only *count* the number of measurements, as it should count the number of occupied rooms. The last measures take the average of the maximum amount of seats available. This we will need to calculate the percentage of used seats with respect to the total amount of seats.

Next we also need a measurement to calculate the number of reserved rooms. This is stored in a different fact table, so we need to define a different cube for that. We find the definition of this cube below:

```

<Cube name="ROOMS_res" cache="false">
  <Table name="active_reservations" alias="" />
  <DimensionUsage source="Time" name="Time" foreignKey="timeID" />
  <DimensionUsage source="Rooms" name="Rooms" foreignKey="roomID" />
  <Measure name="Reserved" column="reservationID" aggregator="count" visible="true" />
</Cube>

```

The definition of the `ROOMS_res` cube is analog to the first cube. The fact table `active_reservations` is again a join on the reservation and rooms table where the reservations made to inactive rooms are discarded. The same dimensions are used and the measurement counts the number of reserved rooms in the fact table.

Now we have two cubes, but we still need to use them both for some reports. Since they have the same dimensions, Mondrian provides us with the opportunity to define a virtual cube to combine them. The definition of the virtual cube is as follows:

```

<VirtualCube name="ROOMS">
  <CubeUsages>
    <CubeUsage cubeName="ROOMS_meas" />
    <CubeUsage cubeName="ROOMS_res" />
  </CubeUsages>
  <VirtualCubeDimension name="Time" />
  <VirtualCubeDimension name="Rooms" />
  <VirtualCubeMeasure cubeName="ROOMS_meas" name="[Measures].[Occupation]" />
  <VirtualCubeMeasure cubeName="ROOMS_meas" name="[Measures].[Utilization]" />
  <VirtualCubeMeasure cubeName="ROOMS_meas" name="[Measures].[MaxSeats]" />
  <VirtualCubeMeasure cubeName="ROOMS_res" name="[Measures].[Reserved]" />
</VirtualCube>

```

The above cube can now be seen as a 3-dimensional cube with `Time`, `Rooms` and `Measures` as its dimensions.

5.4.2 Caching and Tuning

Mondrian's cache ensures that once a multidimensional cell, for example the measurements of room Auditorium 1 during September 2011, has been retrieved from the RDBMS using an SQL query, it is held in memory for subsequent MDX calculations. That cell may be used later during the execution of the same MDX query, and by future queries in the same session and in other sessions. The cache is a major factor ensuring that Mondrian is responsive for quick analysis. If the contents of the RDBMS change while Mondrian is running, Mondrian's implementation must overcome some challenges. The end-user expects a quick query response time and an up to date view of the database. Response time requires a cache, but this cache will become out of date as the database is modified. Mondrian cannot deduce when the database is being modified, so an API is introduced

so that the container can tell Mondrian which parts of the cache are out of date. We can tell Mondrian the cache is out of date when for example measurements are entered or reservations imported.

The performance of Mondrian is determined by a combination of design, hardware, database and other configuration tuning. For really large cubes, the performance issues are driven more by the hardware, operating system and database tuning than anything Mondrian can do. As part of database tuning process the following steps are advised by Mondrian:

- Indexes on primary and foreign keys
- Consider enabling foreign keys
- Ensure that columns are marked NOT NULL where possible
- If a table has a compound primary key, experiment with indexing subsets of the columns with different leading edges. For example, for columns (a, b, c) create a unique index on (a, b, c) and non-unique indexes on (b, c) and (c, a).
- Analyze tables, otherwise the cost-based optimizers will not be used

In MySQL, indexes are already defined on primary keys. In order to create indexes on the foreign keys in the important fact tables *measurements* and *reservations* we define the *foreign key* constraints on the fields *roomID*, *timeID* and *userID*. This should speed up the joins necessary on these fact tables. Almost all fields in the database are set to not allow NULL values. There are no compound primary keys in our design, so we can skip this tuning step. Finally we can apply the **Analyze** operation on tables. This operation analyses and stores the key distribution for the table. MySQL uses the stored key distribution to decide the order in which tables should be joined when you perform a join on something other than a constant.

5.4.3 MDX

MDX stands for multidimensional expressions. It is the query language in the Mondrian OLAP database. MDX was introduced by Microsoft in 1998 with the launch of Microsoft SQL Server OLAP Services. Microsoft proposed MDX as a standard and its adoption is steadily increasing [2].

The syntax of MDX looks like the syntax of SQL, but the structure is quite different. A basic MDX query is structured in a fashion similar to the following example:

```
SELECT [<axis_specification>
      [, <axis_specification>...]]
FROM [<cube_specification>]
[WHERE [< slicer_specification>]]
```

This means that a basic MDX query contains a SELECT clause and a FROM clause with an optional WHERE clause. The SELECT clause determines the axis dimensions of a statement. Axis dimensions determine the edges of a multidimensional result set. Each `axis_specification` value defines one axis dimension. The number of axes is equal to the number of `axis_specification` values in the MDX query. A MDX query can support up to 128 specified axes, but very few MDX queries will use more than 5 axes.

`<axis_specification> ::= <set> ON <axis_name>`

`<axis_name> ::= COLUMNS | ROWS | PAGES | SECTIONS | CHAPTERS | AXIS(<index>)`

Each axis dimension is associated with a number, the first 5 axis have aliases as can be seen in the rule above. The FROM clause determines which multidimensional data source (or Cube) is used to populate the result set of the MDX query. The WHERE clause optionally determines which dimension or member to use as a slicer dimension. A slicer dimension restricts the result set to a specific dimension or member.

We can take a query of one of the reports as an example. Let's take a look at the query that is responsible for generating the room occupation report for individual rooms per period:

```
WITH
  MEMBER Measures.R AS 'Measures.Reserved'
  MEMBER Measures.O AS 'Measures.Occupation'
  MEMBER Measures.[%] AS 'Measures.O / Measures.R', FORMAT_STRING = '#.##'
  SET [mySet] AS {Time.[${start_year}].[${start_week}]:Time.[${end_year}].[${end_week}]}
SELECT
  Crossjoin({[Time.period].Members}, {Measures.R, Measures.O, Measures.[%]}) ON COLUMNS,
  [Rooms.names].Members ON ROWS
FROM [ROOMS]
WHERE [mySet]
```

We see that the SELECT clause is preceded by a WITH clause. The WITH keyword is used to define calculated members and calculated sets. We calculate the percentage of occupied rooms with respect to the reservations. Furthermore we calculate the set of weeks

we need as our interval. The variables indicated by dollar signs, are filled in at runtime by the JavaServer Pages. On the COLUMNS axis we create a **Crossjoin** between all the members of the **Time.period** hierarchy and the 3 measures create in the WITH clause. The **Crossjoin** return the cross product of the two sets. On the ROWS dimension we find the members of the **Rooms.names** hierarchy. The FROM clause takes our main cube **ROOMS** as the data source and the WHERE clause restricts the result set to the members of the time dimension interval.

5.4.4 JPivot

JPivot is a JSP custom tag library that renders an OLAP table and let users perform typical OLAP operations like slice and dice, drill down and roll up. It uses Mondrian as its OLAP Server. It it used to visualize the queries that generate the reports. For every report an MDX query is written. This query is fed to the JPivot custom tag to generate the table. In Figure F.10 in Appendix F we can see output generated by JPivot for the following MDX query. This query is responsible for the report that shows the occupation of every single room versus the reservations of these rooms. The interval can be set using the variables indicated by the dollar signs using an **ActionForm**.

```
<jp:mondrianQuery id="query01" jdbcDriver="com.mysql.jdbc.Driver" jdbcUrl="jdbc:mysql://localhost/rooms" catalogUri="/WEB-INF/queries/diz.xml">
WITH
    MEMBER Measures.R AS 'Measures.Reserved'
    MEMBER Measures.O AS 'Measures.Occupation'
    MEMBER Measures.[%] AS 'Measures.O / Measures.R', FORMAT_STRING = '#.#%'
    SET [mySet] AS {Time.[${start_year}].[${start_week}]:Time.[${end_year}].[${end_week}]}
SELECT
    Crossjoin(mySet, {Measures.R, Measures.O, Measures.[%]}) ON COLUMNS,
    [Rooms.names].Members ON ROWS
FROM [ROOMS]
</jp:mondrianQuery>
```

5.5 Deployment

After implementation, the application was deployed on a virtual web server hosted by XLS Hosting. The virtual server runs on CentOS 5.6, has a 2.2 GHz processor and has 512 MB RAM. Installed is Java 1.6.0_26, MySQL 5.5.10 and Mondrian version 3.2.1. Tomcat 5.5 was installed as the servlet container.

Chapter 6

Results

6.1 Tests

As described in Chapter 2 the Eindhoven University of Technology performed an occupation and utilization research 5 years ago. The data obtained from this research was used to test the system. From the 2006-2007 measurements, data of the first period (week 43 to 46) was used because it was fine grained enough to populate the database. Data from other weeks were only kept on an aggregated level.

When turning on SQL tracing in the relational database we observe that it is only queried when running the reports for the first time. When requesting the same report on the same interval, the data is retrieved from Mondrian cache. When the interval is extended, only the weeks unknown from the cache are retrieved from the MySQL database. Because the Mondrian cache is used when appropriate the response times will decrease. However, with this data set and hardware this is hardly noticeable. The Mondrian cache will start to pay off when the data set increases significantly.

6.2 Evaluation

During and after implementation of the system several evaluation sessions were held with different stakeholders. Stakeholders consisted of DIZ management, potential system administrators and counters. The earlier sessions were used to evaluate implemented functionality and to gain new insights. The feedback received during these sessions was used to

adjust the specifications of the system where necessary. Most of the feedback concerned user interface issues or issues where the requirements do not provide in.

All but a few requirements were implemented. Requirement MEAS-04 in Table A.6 was adapted to the fact that the field *occupied seats* will not be left empty when the room is not occupied, but will not be stored at all. Storing empty requirements is not needed, because if no measurement exists for a given period and room, we will assume it was not occupied. ROOM-07 in Table A.3 was not implemented.

6.3 Future work

The first measurement period will start in the second week of the first semester of the academic year 2011-2012. The measurements will be carried out by the Services department of DIZ. They are already in the possession of the Global Viewer system which enables them to monitor the different college rooms from distance. The Global Viewer system provides real time video images of the college rooms. This makes it unnecessary to physically visit the rooms. The measurements can be carried out from behind a desktop (or any other device with a internet connection) using Global Viewer.

In preparation of this first measurement period, a test measurement period will take place in the interim exam period starting from August 15th 2011. Based on the experience gained and the feedback received from the service employees, adjustments can be made to improve or to ease future measurements.

Appendix A

Functional Requirements

General requirements		
Requirement ID	Requirement	Priority
GEN-01	The system will consist of 6 different subsections: <ul style="list-style-type: none">• User management• Room management• Timetable management• Period management• Measurements• Reporting	1
GEN-02	Every user has to successfully login to use the system	1
GEN-03	There are 2 different user types: <ul style="list-style-type: none">• Administrators• Counters	1
GEN-04	After successfully logging in the user will be presented with the main menu. The main menu represents the 6 subsections from GEN-01. Only the authorized sections are shown	1

Table A.1: General requirements

User management requirements		
Requirement ID	Requirement	Priority
USER-01	A user record consists of the following fields: <ul style="list-style-type: none"> • User ID • Username • Password • Real name (optional) • User type 	1
USER-02	<i>Administrators</i> can create, read, update and delete user records.	1
USER-03	<i>Counters</i> don't have access to user management	1
USER-04	Users are shown in a simple table with in every row the fields described in USER-01. At the end of every row an option to edit or delete the user is presented.	1
USER-05	Users can order the user overview on every column either ascending or descending.	3

Table A.2: User management requirements

Room management requirements		
Requirement ID	Requirement	Priority
ROOM-01	A room record consists of the following fields: <ul style="list-style-type: none"> • Room ID • Full room name • Building • Type • Size • Seats • Active • Notebook ready 	1
ROOM-02	The type of the room can either be 'amphitheater' or 'flat'	1
ROOM-03	The size of the room is divided into 5 categories: <ul style="list-style-type: none"> • A: 20-40 persons • B: 40-70 persons • C: 70-120 persons • D: 120-250 persons • E: >250 persons 	1
ROOM-04	The field Seats will indicate the exact number of seats available in the room. This is necessary for utilization calculations.	1
ROOM-05	<i>Administrators</i> can create, read, update and delete room records	1
ROOM-06	<i>Counters</i> can only read room records	1
ROOM-07	<i>Administrators</i> can extract all room records to a Microsoft Excel file.	2
ROOM-08	<i>Administrators</i> can import room records by uploading a Microsoft Excel print-template exported by Syllabus	2
ROOM-09	Rooms are shown in a simple table with in every row the fields described in ROOM-01. At the end of every row an option to edit or delete the room is presented.	1

Table A.3: Room management requirements

Timetable management requirements		
Requirement ID	Requirement	Priority
TIME-01	A timetable record consists of the following fields: <ul style="list-style-type: none"> • Record ID • Date • Period • Room ID • Faculty • Course ID • Lecturer 	1
TIME-02	<i>Administrators</i> can create, read, update and delete timetable records.	1
TIME-03	<i>Counters</i> can only read timetable records	1
TIME-04	<i>Administrators</i> can import timetable records by uploading a Microsoft Excel print-template exported by Syllabus	2
TIME-05	Timetable is shown in a simple table with in every row the fields described in TIME-01. At the end of every row an option to edit or delete the record is presented.	1

Table A.4: Timetable management requirements

Period management requirements		
Requirement ID	Requirement	Priority
PER-01	A period record consists of the following fields: <ul style="list-style-type: none"> • Period number • Start time • End time • Active 	1
PER-02	<i>Administrators</i> can create, read, update and delete period records.	1
PER-03	<i>Counters</i> can read period records	1
PER-04	Period records are shown in a simple table with in every row the fields described in PER-01. At the end of every row an option to edit or delete the record is presented.	1

Table A.5: Period management requirements

Measurements requirements		
Requirement ID	Requirement	Priority
MEAS-01	<p>A measurement record consists of the following fields:</p> <ul style="list-style-type: none"> • Record ID • Date • Period • Room ID • #Occupied places • User ID • Date + time of entry 	1
MEAS-02	The <i>date</i> field must support all days of the week, so also weekends and holidays.	2
MEAS-03	The <i>period</i> field should correspond to a period record in the <i>Period Management</i> section.	1
MEAS-04	The number of <i>occupied places</i> will be empty if the room is not occupied. Otherwise it will indicate how many seats were indeed occupied.	1
MEAS-05	Given a start and end date, measurements within the given period can be exported to a Microsoft Excel file	2
MEAS-06a	Measurement records are shown in a simple table with in every row the fields described in PER-01.	1
MEAS-06b	The table described in MEAS-06a is shown for every single period. So every table represents a period.	1
MEAS-06c	Every row in the table described in MEAS-06a is extended with its corresponding timetable record when applicable	1

Table A.6: Measurements requirements

Reporting requirements		
Requirement ID	Requirement	Priority
REP-01	Show average number/ percentage of reservations per week vs. average number/percentage room occupation per week	1
REP-02	Show average number/percentage of reservation per week-day vs. average number/percentage room occupation per weekday	1
REP-03	Show average number/percentage of reservations per period vs. average number/percentage of room occupation per period.	1
REP-04	Show average number/percentage of seat utilization per week.	1
REP-05	Show average number/percentage of seat utilization per weekday.	1
REP-06	Show average number/percentage of seat utilization per period	1
REP-07	The requirements REP-01 to REP-06 should show numbers and percentages at the same time.	1
REP-08	The requirements REP-01 to REP-06 aggregate the measurements over a given period of time. Given by a start and end date.	1
REP-09	The requirements REP-01 to REP-06 can be generated for 5 different situations: <ul style="list-style-type: none"> • Individual rooms • Rooms in the same building • Rooms of the same type • Rooms in the same size range • All rooms together 	1
REP-10	REP-01 to REP-06 can be viewed in tabular modus and in graphical modus.	2
REP-11	Reports can be generated by both <i>Administrators</i> and <i>Counters</i>	1
REP-12	Beside the standard reports in REP-01 till REP-06, there can also be generated more flexible reports by dynamically determining cube axis. A row axis, column axis and a filter axis.	2

Table A.7: Reporting requirements

Appendix B

Package Diagram

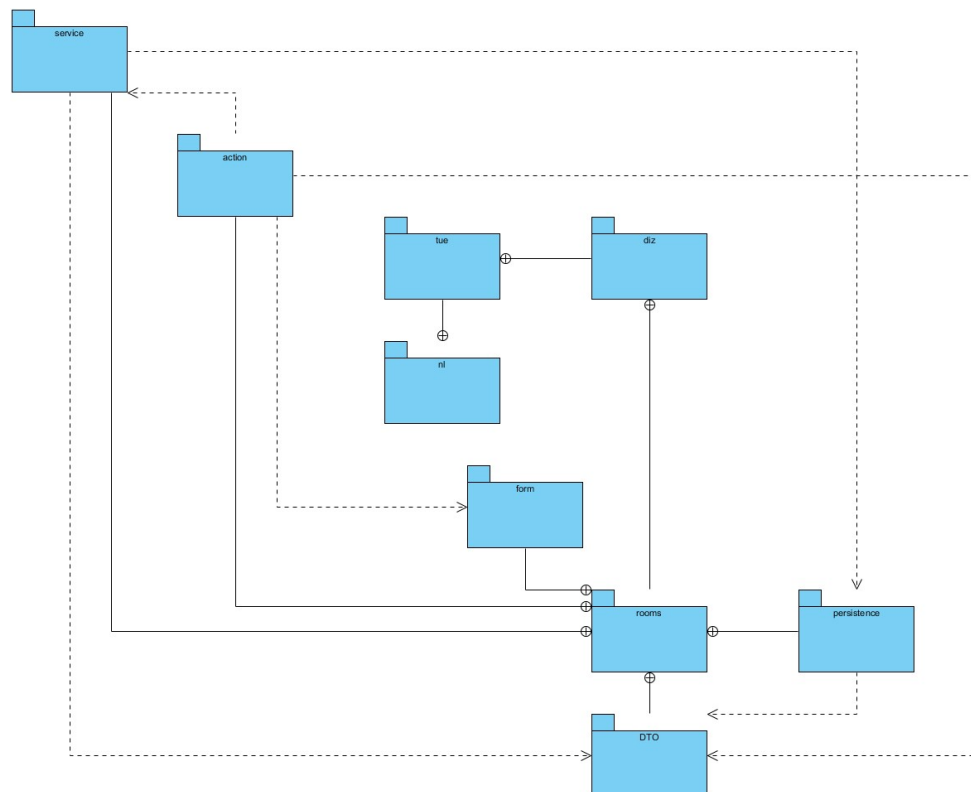


Figure B.1: ROOMS UML Package Diagrams

Appendix C

Class Diagrams





Figure C.2: UML Classes: DTO package

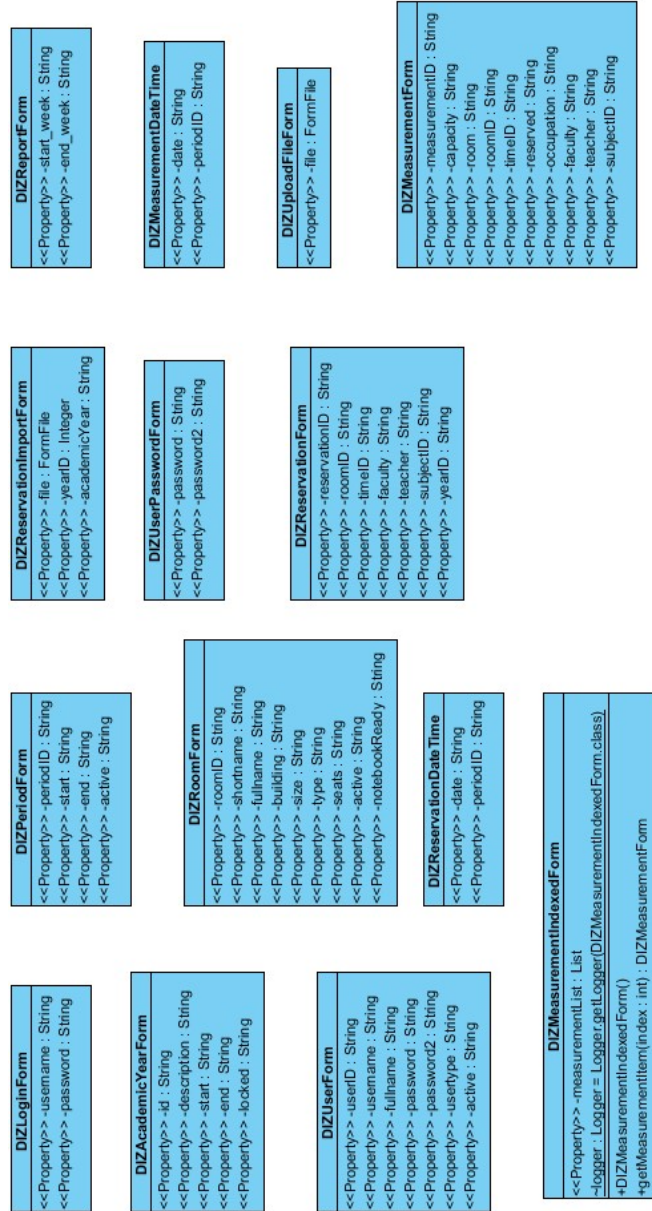


Figure C.4: UML Classes: Form package

Appendix D

Struts Config File

```
<struts-config>
<form-beans>
  <form-bean name="userForm" type="nl.tue.diz.rooms.form.DIZUserForm"/>
  <form-bean name="periodForm" type="nl.tue.diz.rooms.form.DIZPeriodForm"/>
  <form-bean name="roomForm" type="nl.tue.diz.rooms.form.DIZRoomForm"/>
  <form-bean name="reservationForm" type="nl.tue.diz.rooms.form.DIZReservationForm"/>
  <form-bean name="reservationDateTime" type="nl.tue.diz.rooms.form.DIZReservationDateTime"/>
  <form-bean name="reservationImportForm" type="nl.tue.diz.rooms.form.DIZReservationImportForm"/>
  <form-bean name="measurementIndexedForm" type="nl.tue.diz.rooms.form.DIZMeasurementIndexedForm" />
  <form-bean name="measurementForm" type="nl.tue.diz.rooms.form.DIZMeasurementForm" />
  <form-bean name="measurementDateTime" type="nl.tue.diz.rooms.form.DIZMeasurementDateTime" />
  <form-bean name="measurementsImport" type="nl.tue.diz.rooms.form.DIZUploadFileForm" />
  <form-bean name="reservationsImport" type="nl.tue.diz.rooms.form.DIZReservationImportForm" />
  <form-bean name="roomsImport" type="nl.tue.diz.rooms.form.DIZUploadFileForm" />
  <form-bean name="loginForm" type="nl.tue.diz.rooms.form.DIZLoginForm" />
  <form-bean name="academicYearForm" type="nl.tue.diz.rooms.form.DIZAcademicYearForm" />
  <form-bean name="reportForm" type="nl.tue.diz.rooms.form.DIZReportForm" />
</form-beans>

<action-mappings>

<action
  path="/login"
  name="loginForm"
  type="nl.tue.diz.rooms.action.DIZLoginAction"
  scope="request"
  parameter="dispatch">
  <forward name="success" path="/index.jsp" redirect="true" />
  <forward name="failure" path="/login.jsp"/>
</action>

<action
  path="/academicYearSetUp"
  name="academicYearForm"
  type="nl.tue.diz.rooms.action.DIZAcademicYearAction"
  scope="request"
  parameter="dispatch">
  <forward name="success" path="/academicYearForm.jsp"/>
</action>

<action
  path="/academicYearProcess"
  type="nl.tue.diz.rooms.action.DIZAcademicYearAction"
  name="academicYearForm"
  scope="request"
  parameter="dispatch">
  <forward name="failure" path="/academicYearForm.jsp"/>
```

```

        <forward name="success" path="/academicYears.jsp"/>
    </action>

    <action
        path="/userSetUp"
        name="userForm"
        type="nl.tue.diz.rooms.action.DIZUserAction"
        scope="request"
        parameter="dispatch">
        <forward name="success" path="/userForm.jsp"/>
    </action>
    <action
        path="/userProcess"
        type="nl.tue.diz.rooms.action.DIZUserAction"
        name="userForm"
        scope="request"
        parameter="dispatch">
        <forward name="failure" path="/userForm.jsp"/>
        <forward name="success" path="/users.jsp"/>
        <forward name="redirect" path="/userSetUp.do"/>
        <forward name="index" path="/index.jsp"/>
    </action>

    <action
        path="/periodSetUp"
        name="periodForm"
        type="nl.tue.diz.rooms.action.DIZPeriodAction"
        scope="request"
        parameter="dispatch">
        <forward name="success" path="/periodForm.jsp"/>
    </action>
    <action
        path="/periodProcess"
        type="nl.tue.diz.rooms.action.DIZPeriodAction"
        name="periodForm"
        scope="request"
        parameter="dispatch">
        <forward name="failure" path="/periodForm.jsp"/>
        <forward name="success" path="/periods.jsp"/>
    </action>

    <action
        path="/roomSetUp"
        name="roomForm"
        type="nl.tue.diz.rooms.action.DIZRoomAction"
        scope="request"
        parameter="dispatch">
        <forward name="success" path="/roomForm.jsp"/>
    </action>
    <action
        path="/roomProcess"
        type="nl.tue.diz.rooms.action.DIZRoomAction"
        name="roomForm"
        scope="request"
        parameter="dispatch">
        <forward name="failure" path="/roomForm.jsp"/>
        <forward name="success" path="/rooms.jsp"/>
    </action>
    <action
        path="/roomsImport"
        name="roomsImport"
        type="nl.tue.diz.rooms.action.DIZRoomAction"
        scope="request"
        parameter="dispatch">
        <forward name="redirect" path="/roomProcess.do?dispatch=getRooms" redirect="true"/>
    </action>

    <action
        path="/reservationSetUp"
        name="reservationForm"
        type="nl.tue.diz.rooms.action.DIZReservationAction"

```

```

        scope="request"
        parameter="dispatch">
        <forward name="success" path="/reservationForm.jsp"/>
    </action>
    <action
        path="/reservationProcess"
        type="nl.tue.diz.rooms.action.DIZReservationAction"
        name="reservationForm"
        scope="request"
        parameter="dispatch">
        <forward name="failure" path="/reservationForm.jsp"/>
        <forward name="success" path="/reservations.jsp"/>
        <forward name="redirect" path="/academicYearProcess.do?dispatch=getAcademicYears" redirect="true" />
    </action>
    <action
        path="/reservationDateTime"
        name="reservationDateTime"
        type="nl.tue.diz.rooms.action.DIZReservationAction"
        scope="session"
        parameter="dispatch">
        <forward name="success" path="/reservations.jsp"/>
    </action>
    <action
        path="/reservationsImport"
        name="reservationsImport"
        type="nl.tue.diz.rooms.action.DIZReservationAction"
        scope="request"
        parameter="dispatch">
        <forward name="success" path="/reservations_import.jsp"/>
        <forward name="redirect" path="/academicYearProcess.do?dispatch=getAcademicYears" redirect="true" />
    </action>

    <action
        path="/measurementProcess"
        type="nl.tue.diz.rooms.action.DIZMeasurementAction"
        name="measurementIndexedForm"
        scope="request"
        parameter="dispatch">
        <forward name="failure" path="/measurements.jsp"/>
        <forward name="success" path="/measurements.jsp"/>
        <forward name="redirect" path="/measurementProcess.do?dispatch=getMeasurements" redirect="true"/>
    </action>
    <action
        path="/measurementDateTime"
        name="measurementDateTime"
        type="nl.tue.diz.rooms.action.DIZMeasurementAction"
        scope="session"
        parameter="dispatch">
        <forward name="success" path="/measurements.jsp"/>
        <forward name="redirect" path="/measurementProcess.do?dispatch=getMeasurements" redirect="true"/>
    </action>

    <action
        path="/reportsSetType"
        type="nl.tue.diz.rooms.action.DIZReportAction"
        name="reportForm"
        scope="session"
        parameter="dispatch">
        <forward name="success" path="/reports_type.jsp"/>
        <forward name="redirect" path="/jpivot_controller.jsp" redirect="true"/>
    </action>

</action-mappings>

<message-resources parameter="nl.tue.diz.rooms.MessageResources" null="false"/>

</struts-config>

```

Appendix E

Deployment Descriptor File

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <filter>
    <filter-name>JPivotController</filter-name>
    <filter-class>com.tonbeller.wcf.controller.RequestFilter</filter-class>
    <init-param>
      <param-name>errorJSP</param-name>
      <param-value>/error.jsp</param-value>
      <description>URI of error page</description>
    </init-param>
    <init-param>
      <param-name>busyJSP</param-name>
      <param-value>/busy.jsp</param-value>
      <description>This page is displayed if a the user clicks
        on a query before the previous query has finished</description>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>JPivotController</filter-name>
    <url-pattern>/pivot_controller.jsp</url-pattern>
  </filter-mapping>

  <!-- resources initializer -->
  <listener>
    <listener-class>com.tonbeller.tbutils.res.ResourcesFactoryContextListener</listener-class>
  </listener>

  <listener>
    <listener-class>mondrian.web.taglib.Listener</listener-class>
  </listener>

  <servlet>
    <servlet-name>MDXQueryServlet</servlet-name>
    <servlet-class>mondrian.web.servlet.MdxQueryServlet</servlet-class>
    <init-param>
      <param-name>connectString</param-name>
      <param-value>@mondrian.webapp.connectString@</param-value>
    </init-param>
  </servlet>

  <servlet>
    <servlet-name>MondrianXmlaServlet</servlet-name>
    <servlet-class>mondrian.xml.impl.DefaultXmlaServlet</servlet-class>
  </servlet>

</web-app>
```

```

<!-- jfreechart provided servlet -->
<servlet>
  <servlet-name>DisplayChart</servlet-name>
  <servlet-class>org.jfree.chart.servlet.DisplayChart</servlet-class>
</servlet>

<!-- jfreechart provided servlet -->
<servlet>
  <servlet-name>GetChart</servlet-name>
  <display-name>GetChart</display-name>
  <description>Default configuration created for servlet.</description>
  <servlet-class>com.tonbeller.jpivot.chart.GetChart</servlet-class>
</servlet>
<servlet>
  <servlet-name>Print</servlet-name>
  <display-name>Print</display-name>
  <description>Default configuration created for servlet.</description>
  <servlet-class>com.tonbeller.jpivot.print.PrintServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>DisplayChart</servlet-name>
  <url-pattern>/DisplayChart</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>Print</servlet-name>
  <url-pattern>/Print</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>GetChart</servlet-name>
  <url-pattern>/GetChart</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>MDXQueryServlet</servlet-name>
  <url-pattern>/mdxquery</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>MondrianXmlaServlet</servlet-name>
  <url-pattern>/xmla</url-pattern>
</servlet-mapping>

<!-- optional? now in JPivot by default -->
<context-param>
  <param-name>contextFactory</param-name>
  <param-value>com.tonbeller.wcf.controller.RequestContextFactoryImpl</param-value>
</context-param>

<context-param>
  <param-name>connectString</param-name>
  <param-value>@mondrian.webapp.connectString@</param-value>
</context-param>

<taglib>
  <taglib-uri>http://www.tonbeller.com/wcf</taglib-uri>
  <taglib-location>/WEB-INF/wcf/wcf-tags.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>http://www.tonbeller.com/jpivot</taglib-uri>
  <taglib-location>/WEB-INF/jpivot/jpivot-tags.tld</taglib-location>
</taglib>

<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>

```

```

        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>detail</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>login.jsp</welcome-file>
</welcome-file-list>

<context-param>
    <param-name>javax.servlet.jsp.jstl.fmt.localizationContext</param-name>
    <param-value>MessageResources</param-value>
</context-param>

</web-app>


```


Appendix F

Screenshots

ROOMS
ROlap based Occupation Measurement System

Enter your name:
Enter your password:


Log in
Log in using your username and password.

Dienst Interne Zaken @ Tue - 2011 Log out

Figure F.1: ROOMS login



Figure F.2: ROOMS Main menu

ROOMS

ROlap based Occupation Measurement System

Users

Rooms

Periods

Reservations

Measurements

Reports

+

Add new user

user ID	User name	Full name	Usertype	Active?		
1	admin	Administrator	Administrator	Yes		
2	counter	Counter	Counter	Yes		

Users

Add, edit or delete users. Users can be administrators or counters.

Dienst Interne Zaken @ Tue - 2011

Log out

Figure F.3: User management

ROOMS

ROlap based Occupation Measurement System

Users

Rooms

Periods

Reservations

Measurements

Reports

+

Add New Room

room ID	Short name	Full name	Building	Size	Seats	Type	Active?	Notebook ready?		
1	AUD.01	AUDITORIUM 1	AUDITORIUM	D	230	Amphi	Yes	Yes		
2	AUD.02	AUDITORIUM 2	AUDITORIUM	D	147	Flat	Yes	Yes		
3	AUD.03	AUDITORIUM 3	AUDITORIUM	E	300	Flat	Yes	Yes		
4	AUD.04	AUDITORIUM 4	AUDITORIUM	D	170	Flat	Yes	Yes		
5	AUD.05	AUDITORIUM 5	AUDITORIUM	D	170	Flat	Yes	Yes		
6	AUD.06	AUDITORIUM 6	AUDITORIUM	E	280	Flat	Yes	Yes		
7	AUD.07	AUDITORIUM 7	AUDITORIUM	D	167	Flat	Yes	Yes		
8	AUD.08	AUDITORIUM 8	AUDITORIUM	D	200	Flat	Yes	Yes		
9	AUD.09	AUDITORIUM 9	AUDITORIUM	C	88	Flat	Yes	Yes		
10	AUD.10	AUDITORIUM 10	AUDITORIUM	C	88	Flat	Yes	Yes		
11	AUD.11	AUDITORIUM 11	AUDITORIUM	B	70	Flat	Yes	Yes		
12	AUD.12	AUDITORIUM 12	AUDITORIUM	B	70	Flat	Yes	Yes		
13	AUD.13	AUDITORIUM 13	AUDITORIUM	B	70	Flat	Yes	Yes		
14	AUD.14	AUDITORIUM 14	AUDITORIUM	B	70	Flat	Yes	Yes		
15	AUD.15	AUDITORIUM 15	AUDITORIUM	C	78	Flat	Yes	Yes		
16	AUD.16	AUDITORIUM 16	AUDITORIUM	C	78	Flat	No	Yes		

Rooms

Add, edit or delete lecture rooms. Inactive rooms have no influence on Reports.

You can import a Syllabus export file below.

Input file:

Choose File

No file chosen

Upload

Dienst Interne Zaken @ Tue - 2011

Log out

Figure F.4: Room management

ROOMS

ROlap based Occupation Measurement System

Users

Rooms

Periods

Reservations

Measurements

Reports

Update Room

Short name:

AUD.04

Full name:

AUDITORIUM 4

Building:

AUDITORIUM

Size:

D

Seats:

170

Type:

Flat

Notebook ready?:

☒

Active?:

☒

Submit

Cancel

Update Room

The fields short name, fullname, building and seats are required

Dienst Interne Zaken @ Tue - 2011

Log out

Figure F.5: ActiveForm example

ROOMS

ROlap based Occupation Measurement System

Users

Rooms

Periods

Reservations

Measurements

Reports

+

Add New Period

period ID	Start	End	Active?		
1	08:45:00	09:45:00	Yes		
2	09:45:00	10:30:00	Yes		
3	10:45:00	11:30:00	Yes		
4	11:45:00	12:30:00	Yes		
5	13:45:00	14:30:00	Yes		
6	14:45:00	15:30:00	Yes		
7	15:45:00	16:30:00	Yes		
8	16:45:00	17:30:00	Yes		
9	17:45:00	18:30:00	No		
10	18:45:00	19:30:00	Yes		

Periods

Add, edit or delete periods. Periods define the lecture hours of the Technische Universiteit Eindhoven. Inactive periods do not influence reports.

Dienst Interne Zaken @ Tue - 2011

Log out

Figure F.6: Period management

ROOMS

ROlap based Occupation Measurement System

[Users](#)
[Rooms](#)
[Periods](#)
[Reservations](#)
[Measurements](#)
[Reports](#)

05-09-2011

period 3

Get reservations

Reservation ID	Room	Faculty	Teacher	Subject ID		
	AUD.01					
22227	AUD.02	FSA_EE	PEMEN A.J.M.	5EE20		
26558	AUD.03	FSA_W	SCHREURS P.J.G.	4A330		
26889	AUD.04	FSA_IEIS	BERENDS J.J./REYMER I.M.M.J.	1ZZ25		
26024	AUD.05	FSA_WI	SIDOROVA N.	2II65		
27708	AUD.06	FSA_B	JANSSEN H.J.M.	7P060		
27626	AUD.07	FSA_TN	WEGER B.M.M. DE	2DN11		
28685	AUD.08	FSA_B	VOORTHUIS J.C.T.	7X700		
22483	AUD.09	FSA_TN	PEL L.	3MN40		
	AUD.10					
	AUD.11					
	AUD.12					
24149	AUD.13	FSA_IEIS	GELLECUM J.F.B.	2DD20		
26305	AUD.14	FSA_ST	KOETS P.P./VEN A.L.M. VAN DE	6BP10		
22513	AUD.15	FSA_TN	BRUSSAARD G.J.H.	3NB40		

12

Reservations

Edit or delete reservations.
Reservations are related to
Academic Years. There you can
import schedules from Syllabus.

Dienst Interne Zaken @ Tue - 2011

Log out

Figure F.7: Reservation management

ROOMS

ROlap based Occupation Measurement System

Users

Rooms

Periods

Reservations

Measurements

Reports


05-09-2011

period 3

Get Measurements

room ID	Capacity	Occupation	Reserved?	Faculty	Subject ID
AUD.08	200	198	yes	FSA_B	7X700
AUD.06	280	111	yes	FSA_B	7P060
AUD.07	167	23	yes	FSA_TN	2DN11
AUD.04	170		yes	FSA_JEIS	1ZZ25
AUD.03	300		yes	FSA_W	4A330
AUD.14	70	56	yes	FSA_ST	6BP10
AUD.16	78		yes	FSA_B	7X560
AUD.05	170		yes	FSA_WI	2II65
AUD.13	70		yes	FSA_JEIS	2DD20
AUD.15	78		yes	FSA_TN	3NB40
AUD.09	88		yes	FSA_TN	3MN40
AUD.02	147		yes	FSA_EE	5EE20
AUD.01	230		no		
AUD.10	88		no		
AUD.11	70		no		
AUD.12	70		no		

Save



Measurements

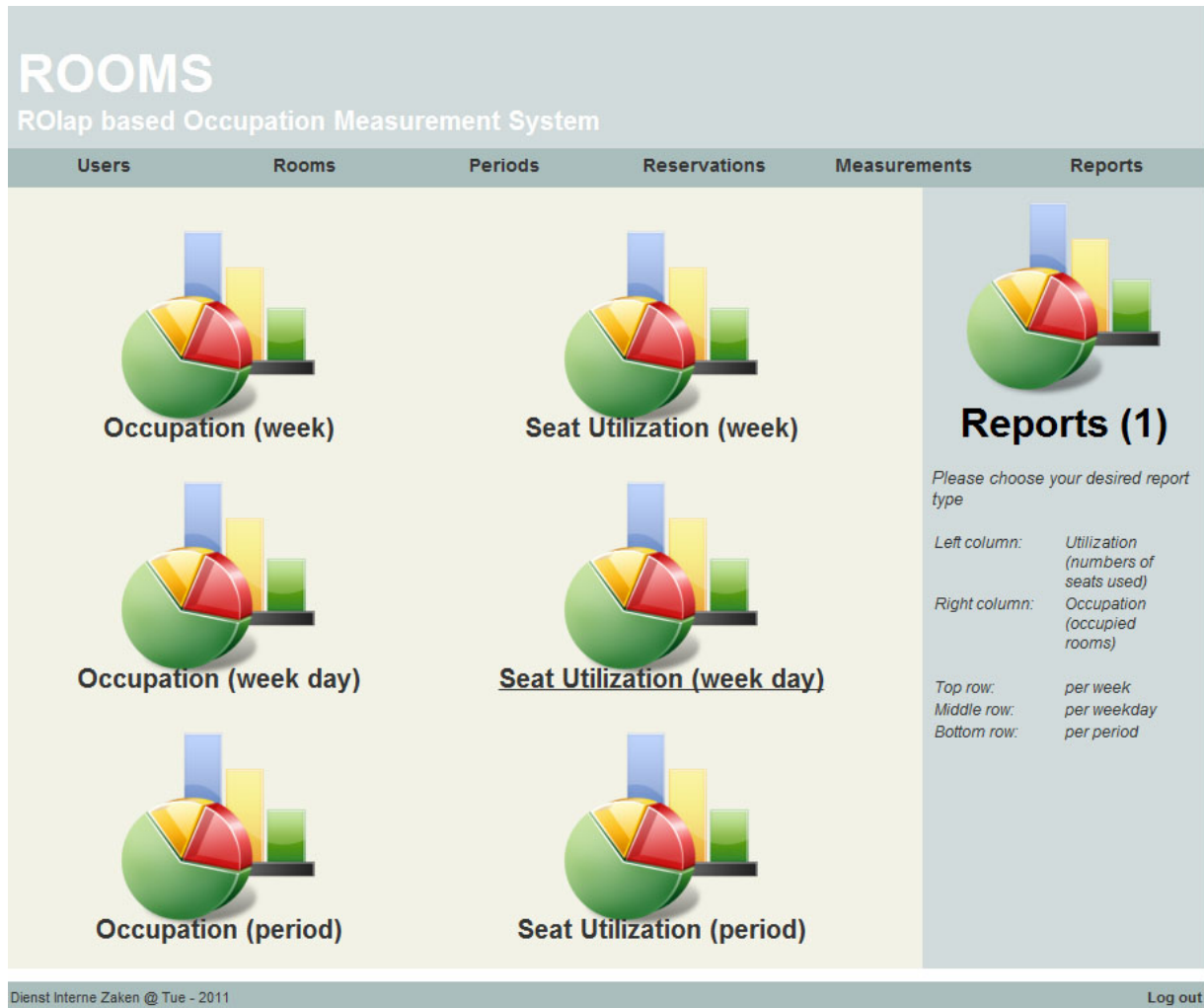
Here you can enter your measurements. Please take the following steps:

- Initial date and period is today
- Check the date or select desired date and period
- Press 'Get measurements' to get the latest status
- List is sorted by building (reserved first)
- Enter your measurements in the green boxes
- Watch the boxes turn red when you enter your data
- Click 'Save'
- Watch the boxes turn green again: successfully saved!
- You can sort on every column by clicking on top of it

Dienst Interne Zaken @ Tue - 2011

Log out

Figure F.8: Measurement management



From: 2006 - week 43 ▼ Until: 2006 - week 46 ▼ Reload

Occupation per week by individual room



	week											
	+43			+44			+45			+46		
	Measures			Measures			Measures			Measures		
names	R	O	%	R	O	%	R	O	%	R	O	%
-All names	423	358	84.6%	424	360	84.9%	423	384	90.8%	423	308	72.8%
AUD.01	24	20	83.3%	24	24	100.0%	24	23	95.8%	24	23	95.8%
AUD.02	33	21	63.6%	33	26	78.8%	33	29	87.9%	33	27	81.8%
AUD.03	32	31	96.9%	32	34	106.3%	32	32	100.0%	32	19	59.4%
AUD.04	17	11	64.7%	17	8	47.1%	17	9	52.9%	17	10	58.8%
AUD.05	16	14	87.5%	16	17	106.3%	16	16	100.0%	16	11	68.8%
AUD.06	31	29	93.5%	31	24	77.4%	31	31	100.0%	31	21	67.7%
AUD.07	34	27	79.4%	34	27	79.4%	34	31	91.2%	34	26	76.5%
AUD.08	27	23	85.2%	27	16	59.3%	27	20	74.1%	27	18	66.7%
AUD.09	36	28	77.8%	36	33	91.7%	36	36	100.0%	36	28	77.8%
AUD.10	31	23	74.2%	31	18	58.1%	31	27	87.1%	31	21	67.7%
AUD.11	30	26	86.7%	30	27	90.0%	30	30	100.0%	30	21	70.0%
AUD.12	22	26	118.2%	23	27	117.4%	22	14	63.6%	22	25	113.6%
AUD.13	28	31	110.7%	28	28	100.0%	28	27	96.4%	28	23	82.1%
AUD.14	33	32	97.0%	33	31	93.9%	33	31	93.9%	33	27	81.8%
AUD.15	29	16	55.2%	29	20	69.0%	29	28	96.6%	29	8	27.6%

Figure F.10: JPivot report example

Appendix G

MySQL Structure

academic_year			
	Field	Type	Extra
P	id	int(11)	Auto Increment
	description	varchar(255)	
	start	date	
	end	date	
	locked	int(1)	

measurements			
	Field	Type	Extra
P	measurementID	int(11)	Auto Increment
	roomID	int(11)	
	timeID	int(11)	
	userID	int(11)	
	occPlaces	smallint(6)	
	timeOfEntry	datetime	
	Index	Fields	Extra
	meas_roomID	roomID	
	meas_timeID	timeID	
	meas_userID	userID	

periods			
	Field	Type	Extra
P	periodID	tinyint(1)	Auto Increment
	start	time	
	end	time	
	active	tinyint(1)	

reservations			
--------------	--	--	--

	Field	Type	Extra
P	reservationID	int(11)	Auto Increment
	roomID	int(11)	
	timeID	int(11)	
	faculty	varchar(255)	
	teacher	varchar(255)	
	subjectID	varchar(255)	
	yearID	int(11)	
	Index	Fields	Extra
	roomID	roomID	
	timeID	timeID	

rooms			
-------	--	--	--

	Field	Type	Extra
P	roomID	int(11)	Auto Increment
	shortname	varchar(10)	
	roomname	varchar(255)	
	building	varchar(100)	
	size	varchar(255)	
	seats	smallint(6)	
	type	varchar(255)	
	active	tinyint(4)	
	notebookReady	tinyint(4)	
	Index	Fields	Extra
	shortname	shortname	Unique

time			
------	--	--	--

	Field	Type	Extra
P	timeID	int(11)	Auto Increment
	year	smallint(4)	
	month	tinyint(2)	
	week	smallint(2)	
	weekday	tinyint(2)	
	weekday_label	varchar(255)	
	day	smallint(2)	
	periodID	int(11)	

users			
-------	--	--	--

	Field	Type	Extra
P	userID	int(6)	Auto Increment
	username	varchar(255)	
	password	varchar(100)	
	fullname	varchar(100)	Allow Null
	usertype	tinyint(1)	
	active	tinyint(1)	

Bibliography

- [1] Bas Ligtenberg *Geautomatiseerd Roosteren: een Verbetering van de Roostersystematiek op de Technische Universiteit Eindhoven*, TU Eindhoven, 2007
- [2] Pentaho *Pentaho Mondrian Project*, Available at: <http://mondrian.pentaho.com/>
- [3] MyBatis *MyBatis: new home of the world's most popular SQL mapping framework.*, Available at: <http://www.mybatis.org/>
- [4] Chuck Cavaness *Programming Jakarta Struts*, O'Reilly Media, June 2004, Second Edition.
- [5] Surajit Chaudhuri and Umeshwar Dayal *An Overview of Data Warehousing and OLAP Technology*, ACM SIGMOD Record, March 1997, Volume 26 Issue 1.
- [6] B.H.M. Winkels, G.J.M. Sprong *Onderzoek Bezetting en Benutting van centraal geroosterde onderwijszalen*, TU Delft, 25 januari 2008, Versie 3.0.
- [7] Cofely *Optimaal en flexibel ruimtegebruik*, Available at: <http://www.cofely-gdfsuez.nl/nl/markten/gebouwen/ruimtemanagementsysteem.html>
- [8] Sun Developer Network *Core J2EE Patterns - Data Access Object*, Available at: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>
- [9] Sun Developer Network *Core J2EE Patterns - Transfer Object*, Available at: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html>
- [10] MySQL *MySQL :: The world's most popular open source database*, Available at: <http://www.mysql.com/>
- [11] Microsoft *MDX - SQL Server 2000*, Available at: <http://msdn.microsoft.com/en-us/library/Aa216767>
- [12] Avraham Leff, James T. Rayfield *Web-Application Development Using the ModelNiewlController Design Pattern*, Enterprise Distributed Object Computing Conference, 2001, p. 118 - 127