



Norwegian University of
Science and Technology

Lecture Quiz 2.0

A service oriented architecture for educational games

Erling Andreas Børresen
Knut Andre Tidemann

Master of Science in Computer Science

Submission date: June 2010

Supervisor: Alf Inge Wang, IDI

Co-supervisor: Bian Wu, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Lecture Quiz is a game similar to Sony's Buzz game used in lecture halls for rehearsing theory and making learning more fun for the students. The game consists of a server with questions and statistics, a student client running on iPhones/iPod Touch, mobile phones or lap-top PCs, and a teacher client running the game shown on a video projector.

The goal of this project is to continue the development of a flexible platform and software architecture for Lecture Quiz games that can provide various game modes and is easy to extend. Quality focus in this project is on "fun-ability", usability, and modifiability. This project can be classified as a mixture of software engineering, software architecture and creativity. It will also be important to develop various game-modes and run user-tests.

Assignment given: 15. January 2010
Supervisor: Alf Inge Wang, IDI

Abstract

This project takes the work done by Mørch-Storstein and Øfsdahl [1] and builds a stable and flexible framework for the Lecture Quiz platform. This platform is a game-like system where teachers can hold quizzes in lectures to increase interactivity with students. The previous prototype was a proof of concept application and the functionality was hard coded to get a working application as fast as possible. This resulted in a system that was unstable and difficult to expand.

With this in mind, we have created an architecture with the focus on easy expansion and modifiability to let new developers interested in the Lecture Quiz platform create new and different content with ease. The applications we have built consist of a server and multiple clients. Each game has a teacher client running on a computer attached to a projector and multiple student web clients accessible by laptops and smart phones. The students log on to the web client and answer the questions shown on the projector in a game-like manner.

We have performed an experiment in a lecture to get feedback from students on how they liked the system. This experiment was performed by running through a quiz in the class and the students delivered a questionnaire afterwards. The students were positive to the system and found it easier to use than the application created by the previous project[1] and many would like to use this system as a recurring element in lectures.

The main result of this project is the completed framework including both clients and the server. It enforces few restrictions on new game play mechanics new developers would like to implement and the base is a stable system. The limitations found in the first Lecture Quiz prototype have been solved and the framework is ready to be taken to the next step which includes further development of the clients to create a spectacular gaming experience.

Preface

This report is a master thesis at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). It is based on the work done by two students in the master thesis Game Enhanced Lectures[1] in 2007.

We would like to thank our supervisor Alf Inge Wang for his great guidance and enthusiasm for this project. We would also like to thank the students of the course *Software Architecture* for participating in our lecture experiment.

Trondheim, 11.06 2010

Erling A. Børresen

Knut A. Tidemann

Contents

I	Introduction	1
1	Introduction	3
1.1	Problem Definition	4
1.2	Project Context	5
1.3	Reader's Guide	5
II	Research Questions and Method	7
2	Research Questions	9
3	Research Method	11
3.1	Development Method	12
3.2	Experiment Method	13
3.2.1	System Usability Scale (SUS)	14
3.2.2	Subjective Assessment	14
III	Prestudy	15
4	Previous Work	17
4.1	Game Theory	17
4.2	State of the Art	19
4.3	Lecture Quiz 1.0	22
5	Technologies	27
5.1	Programming Languages	27
5.1.1	Java	28
5.1.2	C# - Microsoft .NET Framework	28
5.2	Graphics Library	28
5.2.1	OpenGL	29
5.2.2	Direct3D	29
5.3	Web Deployment	29
5.4	Communication Frameworks	30
5.4.1	Web Services	31
5.4.2	Own Protocol	32
5.5	Databases	33
5.5.1	MySQL	33
5.5.2	PostgreSQL	33

IV	Own Contribution	35
6	Requirements	37
6.1	Functional Requirements	37
6.2	Quality Requirements	38
6.2.1	Modifiability	39
6.2.2	Usability	40
6.2.3	Performance	42
6.2.4	Other attributes	43
7	Architecture	45
7.1	Logic View	45
7.1.1	Game Server	46
7.1.2	Teacher Client	46
7.1.3	Student Client	47
7.2	Process View	48
7.2.1	Teacher Client	48
7.2.2	Student Client	50
7.3	Deployment Views	52
7.3.1	One Server	52
7.3.2	Three Servers	52
7.3.3	Multiple Game Servers	54
8	Chosen Technologies	57
8.1	Programming Language	57
8.2	Graphics Library	58
8.3	Web Deployment	58
8.4	Communication Framework	58
8.5	Database System	59
9	Implementation and Design	61
9.1	Lecture Quiz Game Service	61
9.1.1	Configuration	63
9.1.2	Game Modes	63
9.1.3	Quiz Editing	64
9.2	Database design	65
9.3	Teacher Client	66
9.3.1	Game Modes	68
9.3.2	Quiz editor	70
9.3.3	Configuration	71
9.4	Student client	72
9.4.1	Model View Presenter	73
9.4.2	Application control	74
9.4.3	Configuration	75
9.4.4	Game Modes	75
10	User's and Developer's Guides	77
10.1	Deployment	78
10.1.1	Lecture Quiz Service	78
10.1.2	Database Setup	79
10.1.3	Teacher Client	79
10.1.4	Student Client	80

10.1.5	Developer Environment	81
10.2	The Lecture Quiz Service API	81
10.3	Creating a Game Mode	82
10.3.1	Lecture Quiz Service	82
10.3.2	Teacher Client	93
10.3.3	Student Web Client	101
V	Evaluation	105
11	Lecture Experiment	107
11.1	Experiment Delimitation	107
11.2	Experiment Context	108
11.2.1	Participants and Environment	108
11.2.2	Success Criteria	108
11.3	Experiment Execution	109
11.4	Experiment Results	110
11.5	Experiment Evaluation	115
11.6	Experiment Conclusion	118
12	Evaluation	121
12.1	Research Method	121
12.2	Development Method	122
12.3	Requirements	123
12.3.1	Quality Requirements	126
VI	Conclusion	129
13	Conclusion	131
14	Further Work	135
14.1	Improve Clients	135
14.2	Implement Tags and Searching	136
14.3	Improve Quiz Editor	136
14.4	Security	137
14.5	Additional Game Modes	137
14.6	Run Larger Empirical Tests	138
VII	Appendices	139
A	Questionnaire	140
B	Database SQL file	143
C	The Lecture Quiz Service API	147
C.1	Web service Procedures	147
C.1.1	authenticate	148
C.1.2	endQuiz	149
C.1.3	getAvailableGameModes	150
C.1.4	getAvailableQuizzes	151
C.1.5	getCurrentGameStatus	152
C.1.6	getCurrentQuestion	153

C.1.7	getGameModeInfo	155
C.1.8	getOverallStatistics	156
C.1.9	getQuestion	157
C.1.10	getQuestionList	159
C.1.11	getQuestionStatistics	160
C.1.12	getQuiz	162
C.1.13	getServiceVersion	163
C.1.14	joinQuiz	164
C.1.15	newQuiz	165
C.1.16	saveQuiz	167
C.1.17	startNextQuestion	169
C.1.18	submitAnswer	170
C.2	Exported data types	172
C.2.1	Answer	172
C.2.2	QuestionInfo	172
C.2.3	QuizInfo	173
C.2.4	FullQuestionInfo	174
C.2.5	FullQuizInfo	174
C.2.6	StatisticsEntry and ParameterEntry	175
C.2.7	GameModeInfo	175
C.2.8	GameStatus	176

List of Figures

3.1	The Scrum process	13
4.1	Genre relevance for theoretical knowledge [1]	19
4.2	Ez ClickPro sensor and remote controls [1]	21
4.3	Buzz! cover and controllers [1]	21
4.4	Screenshots of the application developed at University of Mannheim [2]	24
4.5	Screenshot of the mobile client of Lecture Quiz 1.0 [1]	25
4.6	Screenshot of the teacher client of Lecture Quiz 1.0 showing a question [1]	25
4.7	Screenshot of the teacher client of Lecture Quiz 1.0 showing statistics for a question [1]	26
7.1	Client-server architecture	46
7.2	Game server architecture	47
7.3	A typical teacher client session	48
7.4	A student client session	50
7.5	A simple setup of Lecture Quiz	53
7.6	A setup of Lecture Quiz with three servers	54
7.7	A setup of Lecture Quiz with multiple game servers	55
9.1	Core class dependencies of the Lecture Quiz Server	62
9.2	ER diagram of database	66
9.3	Core class dependencies of the Lecture Quiz Teacher Client	67
9.4	Screenshot of a question in the teacher client	70
9.5	Screenshot of question statistics in the teacher client	71
9.6	Core class dependencies of the Lecture Quiz Student Client	73
9.7	Screenshot of the login user interface of the student client	74
9.8	Screenshot of the user interface for displaying questions in the student client	76
11.1	Q1 : I think that I am an experienced computer user	111
11.2	Q2 : I think I payed closer attention during the lecture because of the system	112
11.3	Q3 - I found the system had a distracting effect on the lecture	112
11.4	Q4 - I found the system made me learn more	113
11.5	Q5 - I think I learn more during a traditional lecture	113

11.6	Q6 - I found the system made the lecture more fun	114
11.7	Q7 - I think regular use of the system will make me attend more lectures	114
11.8	Q8 - I feel reluctant to pay 0.5 NOK in data transmission fee per lecture to participate in using the system	115

Part I

Introduction

Chapter 1

Introduction

Today, lectures at university level are still very traditional. The use of slides and electronic notes has taken part of the lectures, but they are still mostly one way communication. The teacher will talk about the subject and the students will listen and take notes. This technique may be thought of as boring and have few ways to keep the students concentrated.

The technology has now evolved, and smart phones, laptops and wireless networking has become normal for many university students. This brings new opportunities for interaction in the lecture. With game technology becoming more important on NTNU, Mørch-Storstein and Øfsdahl [1] proposed in 2007 a way to make the lecture more fun and interactive. They made a prototype of a quiz game called Lecture Quiz, that made it possible for the students to participate in a group quiz using their mobile phone or laptop to answer. The questions were presented on a big screen and the teacher had the role as host in the game show. This prototype was created in a hastily manner to prove that the concept was viable. Due to the nature of the development part of the project, much of the features was hard coded and not easy to expand. This resulted in an unstable application with many limitations.

This master thesis will use the idea from [1] and make a new version from ground

up. The outcome will hopefully be used as a basis for further development and make a change in how students and teachers interact in the future.

1.1 Problem Definition

The implementation of Lecture Quiz 1.0 was clearly a prototype that was made as a proof of concept and lacked good methods for extension and modifiability. With everything hard coded it was difficult to extend this prototype into something that could be used in a larger scale. There were issues with an unstable application and the architecture it self was not built for actual usage. This could be easily identified by the many limitations, such as only one session allowed per server and no ability to edit quiz data from a user perspective. In the light of this, the aim of this master thesis is to come up with a good architecture that supports both extensions and modifiability in a good way, and on the same time has a focus on the possibility that the application may be used in many, and big, lectures, and thus must be scalable.

When the architecture is decided we will choose the best fit technologies for this architecture and implement a working solution. The main focus will be to give a good and solid base for others to extend and further develop the system in the future, and thus hopefully making it a regular part of university lectures.

After the implementation phase we will test our solution in a real lecture. This experiment will be compared to the similar experiment of Lecture Quiz 1.0 in 2007. Such test will give vital feedback on how well the concept is received by regular university students.

1.2 Project Context

During the later years the Norwegian University of Science and Technology has had an increasing focus on computer games in their research. There is now established a new research program at the Department of Computer and Information Science and thus several master theses have had focus on computer games. One of the focus areas has been how computer games can affect and improve lectures.

The master thesis of Mørch-Storstein and Øfsdahl [1] in 2007 proposed a game concept for a group quiz designed for use in university lectures. In this thesis Knut A. Tidemann and Erling A. Børresen will, under the supervision of associate professor Alf Inge Wang, look at this concept and create a suitable architecture for this quiz game. Based on this architecture there will be implemented a working solution with focus on modifiability and extendability. The proposed solution will be tested in an experiment taking place in a regular lecture at the Norwegian University of Science and Technology.

1.3 Reader's Guide

This report is divided into seven parts. The first Part gives an introduction to this document. It describes the problem definition, the context of the project, as well as this reader's guide. In Part II we define the research questions and give an overview of the research methods we have used. The research questions define what goals we have for this project. Part III is about our prestudy. The first chapter in this part describes the literature study we have done to give a summary of relevant game theory, and earlier work on educational software. The second chapter in the prestudy part is about possible software solutions for our system.

In Part IV we describe the work we have contributed to this project. First we define a set of requirements based on our findings in the literature study. After the requirements are defined, we will present our chosen architecture. Then we

will explain our chosen technologies suited for this architecture. The next chapter will present the implementation and design of the system. Here we will look at how each of the components in the architecture are built. The last chapter in Part IV will present a set of user's and developer's guides. This chapter is meant for people that are either deploying our software or will extend it. These guides give detailed explanations of how this is done.

Part V is about the evaluation of our project. First we present and evaluate the experiment done in a real lecture situation, and then we evaluate our work in general. In Part VI we conclude our project and give our thoughts of what could be done in the future. The last Part is Appendices, and include the questionnaire used in the experiment, as well as the database SQL file and a detailed description of the Lecture Quiz Service API.

Part II

Research Questions and Method

Chapter 2

Research Questions

From the problem definition in Section 1.1 we have picked out the most critical aspects of it and created a few *research questions* to cover the problem. We will try to answer these questions in this report:

RQ1 What architecture is best suited for the Lecture Quiz game?

We will build an architecture that is well suited for a game where students can join in with ease to participate in games run in lectures. We will also look at some of the specific tasks involved in this in the sub-questions of this research question.

a) How should data be exchanged between the clients and the game server?

We will go through different methods of communication and see which form is most suited for this type of architecture with weight on both ease of use and modifiability.

b) How does this architecture scale when the number of users increases?

How does our chosen architecture hold up with scenarios where many users will use the system at the same time and how can we make the architecture scalable to support this?

c) How do we design the architecture flexible in terms of game modes¹?

We want to find out how we can make the architecture allow developers to create new ways to play the game without the need to rewrite large parts of the software.

RQ2 What technologies are best fit for the chosen architecture?

We will use our findings in **RQ1** and try to find out what existing technologies would be best suited for the implementation of our architecture.

RQ3 Will students consider the software easier to use if it is web based?

We will compare our architecture to previous work and find out if students prefer a web based system for the client applications by doing a usability survey.

RQ4 Do students enjoy playing educational games during lectures?

With the help of the study performed in **RQ3** we will get feedback from students on the fun parts of the game and see if they enjoy playing educational games during lectures.

¹A game mode defines the rules of a quiz game. It determines how questions are answered and how scoring is done as well as other aspects of the gameplay.

Chapter 3

Research Method

When doing research in software engineering there are often problems to be solved that has more than one solution, and that none of the solutions could be pointed out as the correct one. To achieve one of the possible solutions there should be applied an approved research method. Basili [3] lists three recommended methods to be used in software engineering, these are shown below.

The engineering approach: In this approach the domain of the problem is examined to find existing solutions, for then to propose new and better solutions. New solutions are constructed and tested until no more improvements may be found.

The empirical approach: Based on a proposed model of the problem domain, statistical/qualitative methods are developed. These are applied to case studies and then validated against the proposed model. Hence this solution will give more reliable results.

The mathematical approach: When using this approach, a set of axioms or a formal theory is proposed. Based on this, results may be derived, and then it could be possible to compare them with empirical data.

Based on these three approaches we have decided to go with the engineering approach. We will, using a literature study, find and evaluate the previous work on the field of a lecture quiz game and then propose a new and improved solution. This solution will then be tested in a small experiment in a real lecture situation. The method for developing the new solution will be described in the following section. And then the method used for the experiment will be presented in Section 3.2.

3.1 Development Method

Agile and iterative development methods are popular in software development these days. Where Scrum, as described in [4], is one of the most used. Normally, iterative processes have iterations lasting from a few weeks to a few months depending on the size of the project. Since this project is fairly small, we have chosen to set the size of each iteration to two weeks.

The overall scrum process is shown in Figure 3.1¹. An iteration in scrum terms is called a sprint. At the beginning of each sprint we would create a sprint backlog with the set of tasks that are planned to be solved in the sprint. Once every day there is a scrum meeting where each participant need to answer the following questions [4].

- What have you completed since the last meeting?
- What obstacles have you met during this work?
- What are you planning to do until the next meeting?

Answering these questions each day will help the other participants to be aware of each others work and the participants may easier help each other if it is discovered that someone has a problem.

¹Picture from Wikimedia Commons with GNU Free Documentation License

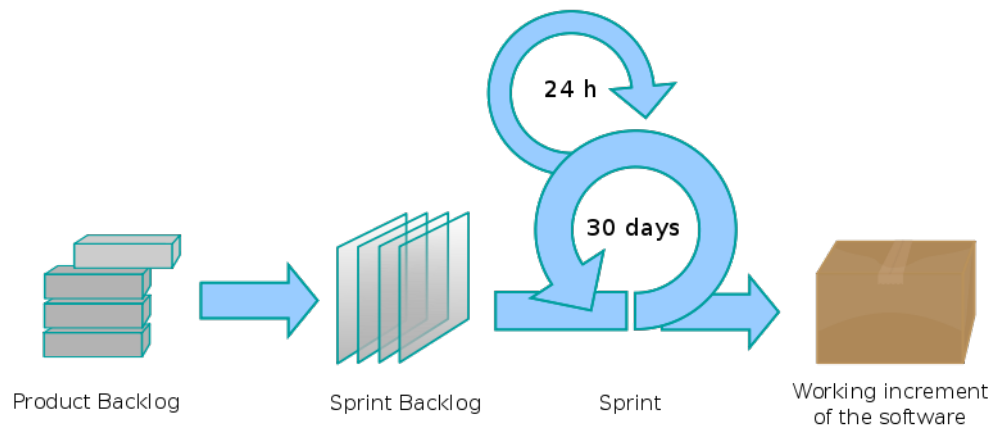


Figure 3.1: The Scrum process

At the end of each sprint there is built a working copy of the software and the sprint is evaluated. This evaluation will work as a basis for deciding what will be in the next sprint backlog.

Although we have chosen to use Scrum as our development method, the report is presented more like a waterfall project. This is done to give the report a less complex structure, and to ease the reading.

3.2 Experiment Method

After developing a prototype, there were held an experiment in a real life situation. As this experiment was going to be compared with the experiment of 2007 [1], we tried to use mostly the same methods as that experiment. As defined in ISO 9241-11 [5] there are three main goals of usability and we will try to measure our usability within these three goals:

- *Effectiveness - The ability of users to complete tasks using the system, and the quality of the output of those tasks*

- *Efficiency - The level of resource consumed in performing tasks*
- *Satisfaction - Users' subjective reactions to using the system*

3.2.1 System Usability Scale (SUS)

System Usability Scale (SUS) [5] is a scale used to measure the usability of software based on a questionnaire of 10 questions. The scale goes from 0 to 100, where 100 is the best score. All the ten questions are publicly available and SUS has proven to give similar results as more complex and time consuming usability tests.

Each question in a SUS test may be rated by the user from 0 to 5, where 0 is strongly disagree, and 5 is strongly agree. To calculate the final SUS score every even question gets a score of 5 minus the average rating for that question. While the odd questions score is calculated by subtracting 1 from the average rating of that question. The overall SUS score will then be the sum of all the scores of the individual questions multiplied by 2.5. The result and calculation of our SUS score is included in Section 11.4

3.2.2 Subjective Assessment

In addition to the SUS questions, we added a set of questions to our questionnaire that were specific to our software. Some of the questions were taken from the questionnaire used in 2007 and some were completely new. These additional questions gave us answers on how well the software worked in a lecture setting and how the students thought of using it in future lectures. All questions will be discussed in a non-formal evaluation to get a grip on how our software was perceived by the students.

Part III

Prestudy

Chapter 4

Previous Work

In this chapter we will look at some of the previous work done in the field of quiz games and gaming in lecture situations. Most of this theory and research was done in the master thesis of Ole Kristian Mørch-Storstein and Terje Øfsdahl [1] in 2007, and are summarized here.

4.1 Game Theory

In [1] they present eight important characteristics of good educational games. The following list of characteristics is meant as a reference for people designing educational games. They argue that missing one of the characteristics may not mean that the game will be unpopular or unsuccessful, but including the missing characteristics in the game concept may make it better.

- **Variable instructional control** - How the difficulty is adjustable or adjusts to the skills of the player
- **Presence of instructional support** - The possibility to give the player hints when he or she is incapable of solving a task

- **Necessary external support** - the need for use of external support.
- **Inviting screen design** - The feeling of playing a game and not operating a program
- **Practice strategy** - The possibility to practice the game without affecting the users score or status.
- **Sound instructional principles** - How well the user is taught how to use and play the game.
- **Concept credibility** - Abstracting the theory or skills to maintain integrity of the instruction.
- **Inspiring game concept** - Making the game inspiring and fun.

The authors of [1] also present a taxonomy of educational games. Using three criteria, listed below, they group educational games in a set of game genres.

- **Player interaction** - Is it possible for several players to interact with the system in some way?
- **Fantasy and skills interaction** - Is the fantasy of the game extrinsic¹ or intrinsic²?
- **Game concept type** - In what genre of computer games does the game concept belong?

Based on this taxonomy they present a model, shown in figure 4.1, that shows how each game genre provides theoretical knowledge. We may classify the Lecture Quiz game as a group quiz, and according to this model it is an effective and simple genre. We will during the design and implementation of our system try to achieve as many of the presented characteristics as possible, and thus hopefully make a good system for educational purposes.

¹Extrinsic fantasies are fantasies that are independent of the application of skills in the game [1]

²Intrinsic fantasies are fantasies that are inherent and essential to the game concept [1]

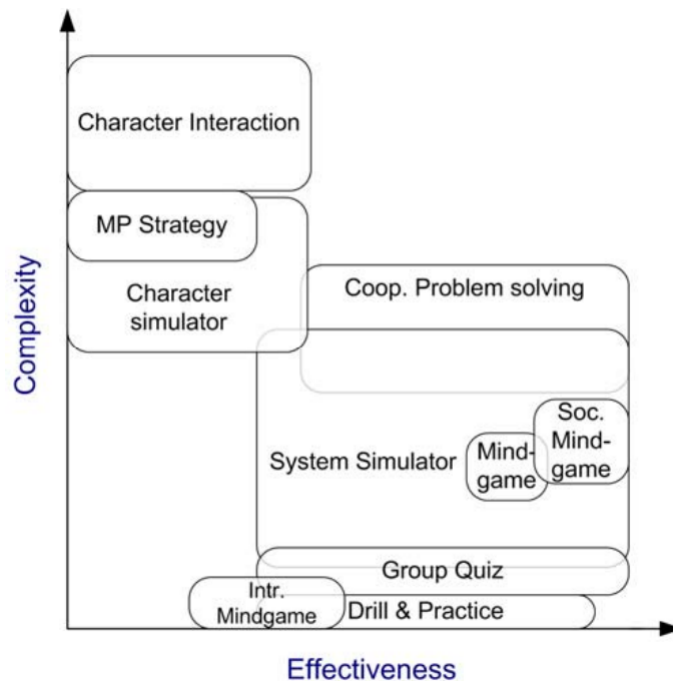


Figure 4.1: Genre relevance for theoretical knowledge [1]

4.2 State of the Art

In [1] they present six existing solutions of educational software. Each providing some kind of support to the lecture environment.

- **TVREMOTE Framework** - Designed at Darmstadt University of Technology, Germany[6]. Supports polling of student opinions and question submissions. The teacher may also broadcast notes, links and multiple choice questions. Feedback is read by the teacher on a private display, and may show selective data on a second, public screen. Uses GPRS for data transmission.
- **Classroom Presenter** - A plug-in to Microsoft PowerPoint developed at University of Washington, USA. Allows public writing on slides in a presentation. The plug-in is open source and used in courses teaching software engineering and algorithms[7, 8].

- **WIL/MA** - Developed at the University of Mannheim, Germany. Is a Java implementation for digital hand raising, spontaneous comments and multiple choice questions. Using PDAs with J2ME support, and WLAN for data transmission.
- **ClassInHand** - Developed at Wake Forest University, USA. Uses PDAs with Windows Mobile and features a presentation controller, student/teacher interaction and real time quizzes. The teacher also uses a PDA to interact with the system[9, 10].
- **Ez ClickPro** - Commercial classroom polling application developed for teaching in elementary school by Avrio Ideas. Uses infrared light for data transmission to the custom handheld controls, shown in Figure 4.2. The teacher uses a PC to interact with the program. Supports a multiple choice quiz game with presentation on a projector or TV. The questions can be supplemented with both pictures and videos[11].
- **Buzz! The Schools Quiz** - Commercial game for the PlayStation 2. Developed with government funding for schools in the UK[12], it works as a game show featuring nice graphics and an enthusiastic game host. The game is shown on a TV or projector using wired custom controls, shown in Figure 4.3, limiting the game to four players.

The features of the listed applications are summarized in Table 4.1. As shown, only TVRemote has no need for custom hardware, and only Ez ClickPro and Buzz! has animated graphics. The most common feature is the quiz mode, but it is mainly for the teacher to monitor the knowledge of the students and not for competition.

In addition to the applications described in [1] there was made a prototype application at the University of Mannheim, Germany [2], to discover if such an application could make the lectures more interactive. The application was run on PDAs using custom software to allow the students to report questions to the lecturer, give feedback and for the lecturer to run a quiz on the selected topic. A number



Figure 4.2: Ez ClickPro sensor and remote controls [1]



Figure 4.3: Buzz! cover and controllers [1]

of questions would then be transferred to all of the PDAs and the students used some time to answer the questions. After all answers was submitted the results

Feature	TVRemote	Cl.Pres.	WIL/MA	ClassInHand	Ez ClickPro	Buzz!
Digital student comments	×		×	×		
Teacher info broadcast	×	×	×	×		
Quiz mode	×		×	×	×	×
Public feedback display	(×)	×			×	×
Animated graphics					×	×
No custom HW needs	×					

Table 4.1: Features of the previous solutions

where reviewed and displayed both on the big screen in the lecture room, and on the PDAs. Data transfer between the PDAs and the main server was done using a custom protocol on top of TCP. Wireless network was available in the lecture room, so no cables were required for the PDAs to communicate. Screenshots of the application are shown in Figure 4.4. Compared to the idea behind the system presented in this report, the application from Mannheim is in many ways similar, but it has no focus on being a game and is more like a tool for the lecturer to see what the students have learned during the lecture.

4.3 Lecture Quiz 1.0

In this section we are presenting the prototype of the Lecture Quiz game developed in [1]. This was done as a part of a master thesis on NTNU where they focused on the impact of using games in a lecture environment. The main focus of this prototype was to develop a quiz game where students were asked questions and each student used his or her own mobile phone or laptop to answer.

The developed prototype consisted of one main server, a teacher client and a student client. Communication between the different parts were done through a custom built protocol on top of TCP.

The student client was developed using J2ME, the Java implementation for mobile devices. To begin a session each student had to download the software to their phone using wireless network, bluetooth or the mobile network(GPRS/EDGE/3G).

After the download was finished, the software had to be installed before the students were ready to participate. This was seen as a bit of a cumbersome process. Screenshots of the student client are shown in Figure 4.5

The teacher client was implemented in Java and used OpenGL to display graphics on a big screen, as shown in Figure 4.6 and Figure 4.7.

The prototype implemented two game modes. One plain game mode where all the students answered all the questions as they were asked. Each question had its own time limit, and the students had to answer within that time. After each question a screen with statistics was displayed, providing information on how many students that answered on each option. At the end of the quiz, the teacher client displayed a list of the students having the most correct answers.

The other game mode was last man standing. The questions were asked in the same way as with the plain game mode, but if a student answered incorrectly he or she was removed from the game. The game continued until all but one student had answered a question wrong, and that student was crowned as winner.

One of the main drawbacks of this prototype is that it lacks a good architecture, making it hard to extend, modify and maintain. It also lacks good documentation and to add a new quiz or a question you have to manually edit the data in the database. The time spent on downloading and installing the software on the students devices also made it less interesting for regular use in lectures.

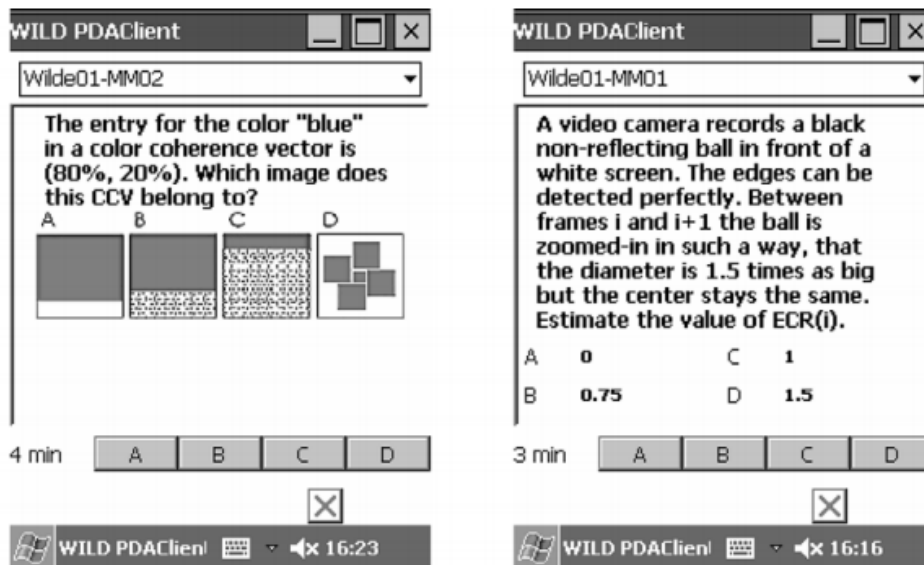


Figure 4.4: Screenshots of the application developed at University of Mannheim [2]

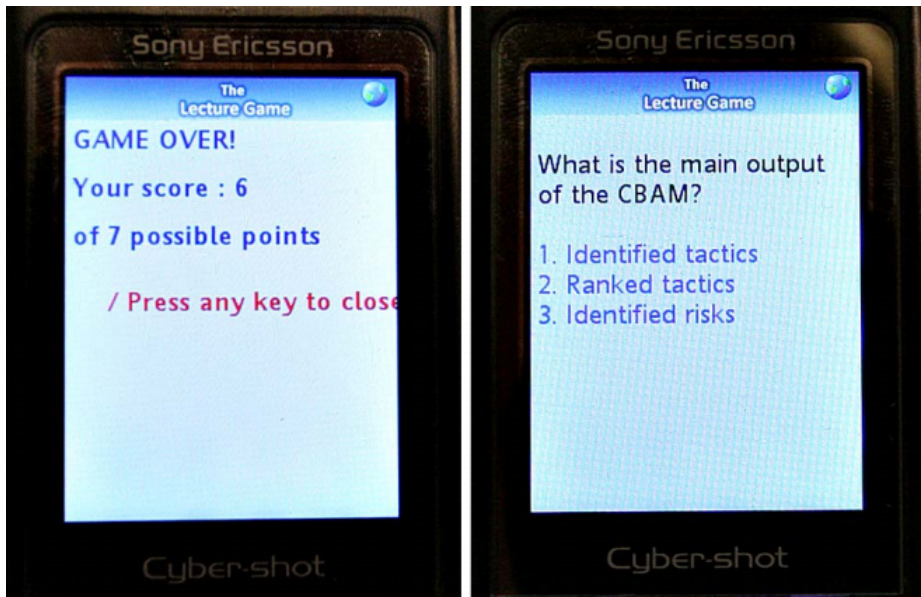


Figure 4.5: Screenshot of the mobile client of Lecture Quiz 1.0 [1]

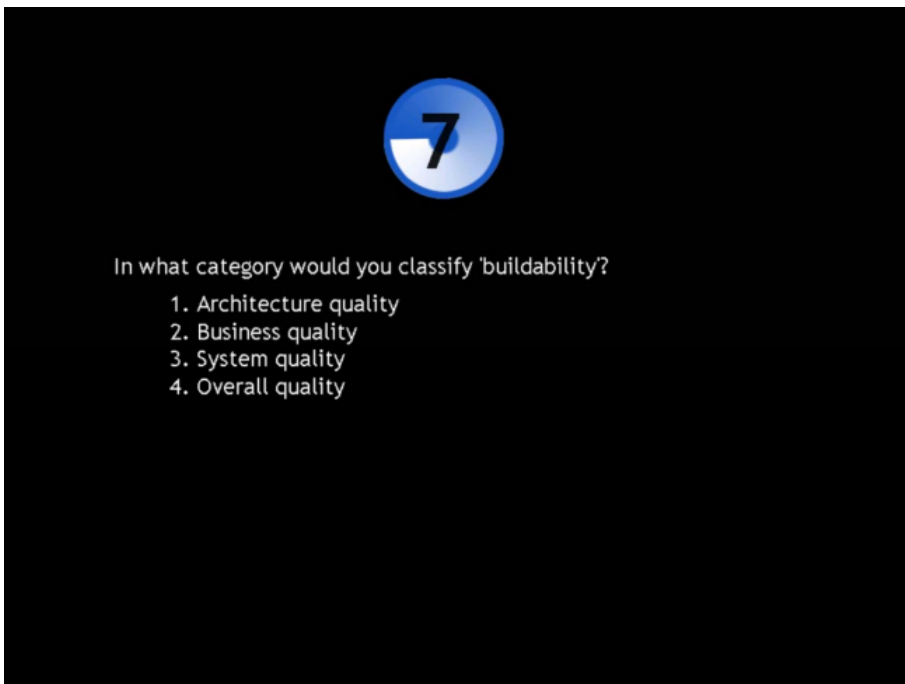


Figure 4.6: Screenshot of the teacher client of Lecture Quiz 1.0 showing a question [1]

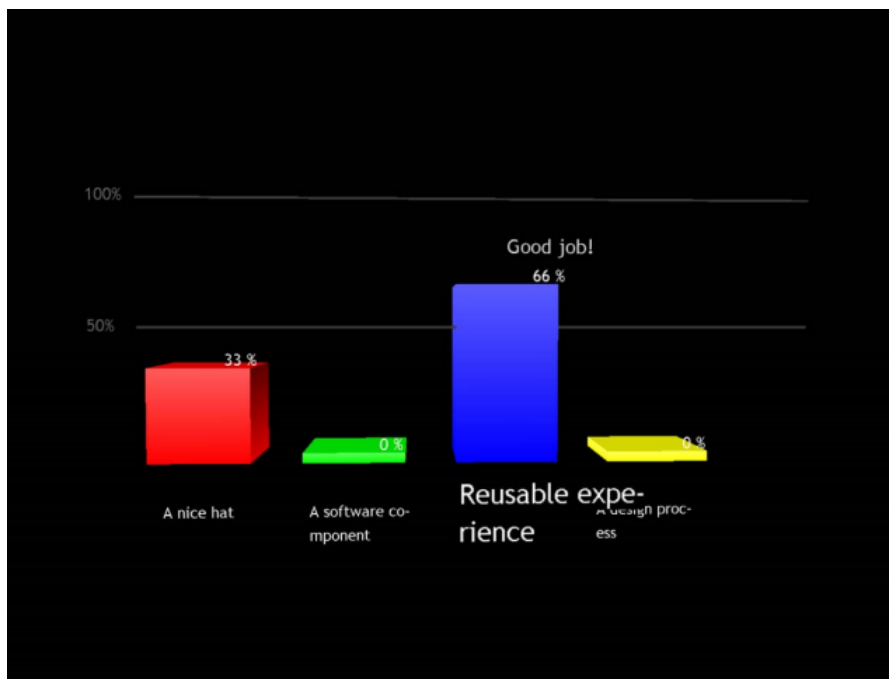


Figure 4.7: Screenshot of the teacher client of Lecture Quiz 1.0 showing statistics for a question [1]

Chapter 5

Technologies

In this chapter we will discuss the various available technologies we can use to solve our problem. We will describe the most relevant pieces of technology that can help us create a better and more scalable architecture. We will evaluate and choose the appropriate technologies for our architecture in Chapter 8.

5.1 Programming Languages

In this project, we have three separate components that all perform different tasks. As the project is required to reach multiple platforms, including a standalone PC client and easy to access clients for mobile phones and PCs, we are looking for a programming language that can work well for the cases we require. The two largest programming languages out there that support this are Sun's Java and the Microsoft .NET Framework. While we could split up the project in different languages, we decided that it would be best to have all parts written in the same programming language, as this would ease up the learning curve for any new contributors and unite the project as one entity.

5.1.1 Java

Java is a programming language created by Sun Microsystems. In 2006 most of the implementation code was released under the GNU General Public License[13]. The Java platform consist of a bundle of programs which allow for developing and running applications written in the Java programming language. The Java platform runs on many types of devices through different implementations of the platform. It is also possible to write Java Servlets, which is a way to publish Java applications on the web. This makes it possible to write web services and web pages in Java.

5.1.2 C# - Microsoft .NET Framework

The Microsoft .NET Framework was released in 2002 by Microsoft and is currently at version 4.0[14][15]. The framework is available for the Windows platform as well as on Linux and other Unix/X11 platforms in a limited version through the help of the Mono project[16]. Many programming languages are supported by the framework, but C# is the most frequently used and has a very similar syntax to Java. The .NET Framework can be used to develop standalone applications, web services and web pages.

5.2 Graphics Library

Our teacher client will be displaying the quizzes on a projector or similar equipment in a large class room, and have the possibility to display advanced graphics. The two most used graphics libraries today are the OpenGL library and Direct3D which is a part of the Microsoft DirectX library. By basing our client on one of these widely known libraries we make it easier to extend the client for new developers.

5.2.1 OpenGL

The OpenGL graphics library is a standard that is developed by the OpenGL Working Group[17]. OpenGL is an open standard and is available for many platforms, including Windows, Linux, Mac OS X and embedded devices. The API is originally written for the C programming language, but bindings can be installed for many other languages. The graphical features of OpenGL is being extended all the time and is kept up to date with the latest improvements from graphic card manufacturers. It is capable of rendering both advanced 3D and 2D graphics. Installation of the library it self is not needed as it comes with graphic card drivers, however bindings to languages other than C and C++ must be packaged with the application if they are used.

5.2.2 Direct3D

The Direct3D API is a part of the Microsoft DirectX library. It is the main competitor to the OpenGL library and comes pre-installed on all Windows versions. The API is for C/C++ and C# and runs only on the Windows platform. It is developed by Microsoft alone, with the exception drivers from graphic card manufacturers. Direct3D can render both advanced 3D and 2D graphics and is widely used on the Windows platform. While the library it self comes with Windows, updates might need to be installed to ensure a full feature set when using the newest Direct3D API. Using Direct3D or DirectX locks you to the Windows platform.

5.3 Web Deployment

To publish web services written in Java or the .NET Framework on the web, a server is required. There are many out there that all suit different needs. Services and web sites written in the .NET Framework can be deployed on the Microsoft IIS server or on Apache with the help of the Mono Project. Java services on the

other hand are deployed to a web container. The three largest web containers for Java are JBoss¹, Apache Tomcat² and GlassFish³.

JBoss was developed by JBoss, now a division of Red Hat. As of JBoss 5.1, released in 2009, it operates as a Java EE 5 application server, which implements the Servlet 2.5 and JSP 2.1 specifications from Sun Microsystems. Since it is written in Java, JBoss is platform independent and can operate on any system that Java supports.

Tomcat is developed by the Apache Software Foundation and implements the Servlet 2.5 and JSP 2.1 specifications. It is also cross-platform and written in Java.

The GlassFish application server is developed by Sun Microsystems and is the Java Enterprise Edition reference implementation, and therefore the first to implement Java EE 6, which includes the Servlet 3.0 specification. As with the other application servers, it is also written in Java and is cross-platform.

All of the Java application servers are free and open source. For the .NET platform Apache and Mono are both free, open source and cross-platform while Microsoft's IIS is proprietary and only runs on Windows.

5.4 Communication Frameworks

In this section we will look at different ways components may communicate with each other to make requests and sharing data.

¹<http://labs.jboss.com/jbossas/>

²<http://tomcat.apache.org/>

³<https://glassfish.dev.java.net/>

5.4.1 Web Services

Web services is a way to make an API accessible on the internet. This is typically done over HTTP, but there are other options as well. A web service makes it easy for a data provider to publish the data in a way that makes it less complicated for a 3rd party to write client software. One of the main benefits of using web services is that it makes the services independent on what programming language the clients are written in, thus making it easy to make a new client for your favorite platform.

One of the downsides with web services, is that the client have to initiate the requests. This means that for the server to send data to a client, it has to answer a request from the client, and thus making it hard to use patterns like listeners.

There are two main styles of publishing web services, SOAP and REST. These are described in more detail in the following sections.

SOAP

SOAP is a W3C⁴ recommendation for web services. SOAP was originally an acronym for Simple Object Access Protocol, but as of version 1.2 it is just a name[18]. SOAP relies on XML for formatting messages, and on a protocol for message transfer. HTTP is the most common used transport protocol[19], but using SMTP or sending SOAP messages directly over TCP are other options.

To make SOAP easy to use for the client developers, there is a standard way of contracting the provided services. WSDL⁵ is used for this purpose. The WSDL document defines the services provided in an XML file so it may be read by software for a number of platforms. With the WSDL document in hand it is possible to create client-support code, making classes and methods available in the chosen programming language.

⁴World Wide Web Consortium

⁵Web Services Description Language

REST

REST⁶ is an acronym defined by Roy Fielding in his Ph.D. dissertation[20]. REST is about providing an interface to a set of resources, and each resource is identified by a unique identifier (URI). REST is restricted to the HTTP protocol, but takes use of the powers of that protocol to do operations on the resources. HTTP has several verbs that describe operations on a resource, REST uses POST, PUT, GET and DELETE, whereas SOAP only use POST and GET.

Where SOAP is a messaging protocol, REST is seen as a style of software architecture for distributed systems on the web. REST does not have one standard way of encoding the data that is transferred, the developers may choose by themselves. Examples of encodings are XML and JSON.

Compared to SOAP's WSDL, REST has no standard for contracting the service provided. The WADL⁷ document is an effort towards a standard, but it is not yet official nor widely accepted, thus not provided in a lot of frameworks for creating or consuming REST services.

5.4.2 Own Protocol

Instead of relying on an already implemented way of communicating it is possible to write an own application protocol on top of TCP or UDP. Doing so will give full flexibility on how to make the communication work, but that will also require more time spent on things that already exist in other protocols. Session management is one such example.

The server implementing this protocol will also have to run on a non standard TCP port, making it harder to pass through firewalls, thus giving more job to system administrators.

⁶Representation State Transfer

⁷Web Application Description Language

5.5 Databases

In our application we need a persistent storage to store information such as quizzes, users, questions and more. For this task we will consider a database back end. In the following section we will discuss a two DBMS's⁸ that can work as our back end to persistent information storage. We will take a look at MySQL and PostgreSQL, but other solutions exist such as Oracle Database and Microsoft SQL Server.

5.5.1 MySQL

MySQL is an open source relational database system currently owned and developed by Oracle. It is the worlds most popular open source database software and has high speed, great reliability and is easy to use[21]. It has two licenses, one commercial and one open source license. The commercial license is required if you only distribute your application in binary form to end users. MySQL has a client API for many languages, including C, C++, Java and C# .NET. It runs as a server, and both the server and client applications run on many systems, including Windows, Linux and Mac OS X.

5.5.2 PostgreSQL

PostgreSQL is a powerful open source object-relational database system running on all major operating systems, including Windows, Linux and Mac OS X[22]. It is released under the PostgreSQL license which is a liberal open source license similar to the BSD and MIT licenses. With native programming interfaces to many languages such as C, C++, Java and C# it is easy to integrate in applications. It runs as a server and have clients connect to it to make query requests.

⁸Database Management Systems

Part IV

Own Contribution

In this part we will describe our own contribution to the Lecture Quiz system. We will outline our architecture and explain why we have chosen the technologies we use. The details of our implementation of the architecture will be laid out in Chapter 9. There is also included a set of user's and developer's guides that will help both administrators and new developers make use of the Lecture Quiz system.

Chapter 6

Requirements

By taking a look at the previous prototype of the Lecture Quiz platform we created a set of functional and quality requirements. We looked at the previously identified issues with the prototype from 2007, and made requirements that would fix many of these issues. The main focus was to move the Lecture Quiz system from a proof of concept system to a modifiable application framework for further development.

6.1 Functional Requirements

In this section we will describe a set of functional requirements. These are listed in Table 6.1. Most of the requirements are of high priority, which means that they all of them should be implemented. Medium prioritized requirements are requirements that would be nice to have, but should only be implemented if all the other requirements are fulfilled.

ID	Description	Priority
FR1	The game shall consist of a teacher client and a number of student clients	High
FR2	The teachers need to authenticate to use the client	High
FR3	It must be possible to extend the game with new game types	High
FR4	It must be possible for the teachers to store questions for later use	High
FR5	It must be possible to tag questions for easier reuse and grouping	High
FR6	A question shall consist of four options	High
FR7	Questions must be able to have an individual time limit	High
FR8	It must be possible to run several quizzes at the same time	High
FR9	Statistics should be shown after a question has been answered and after the quiz has been completed	High
FR10	A quiz game may group the students into groups	High
FR11	The teacher decides when to start a quiz	High
FR12	The students must identify them selves with a user name when joining a quiz	High
FR13	The students must answer questions before the time limit is up	High
FR14	The students must supply a quiz code to join a quiz	High
FR15	The teacher must be able to pause between questions	High
FR16	The teacher must be able to save the statistics from a quiz round that has just ended	Medium

Table 6.1: List of functional requirements

6.2 Quality Requirements

As well as functional requirements, the non-functional requirements are important. In this section we describe these requirements using scenarios, as shown in *Software Architecture in Practice*[23], grouped by quality attributes.

6.2.1 Modifiability

The overall goal of this thesis is to make the Lecture Quiz applications easy to modify and extend. This section describes some scenarios regarding modifiability.

M1 - Deploying a new game mode for a client	
Source of stimulus	Game mode developer
Stimulus	The game mode developer wants to deploy a new game mode for one of the Lecture Quiz clients or the server
Environment	Design time
Artifact	One of the Lecture Quiz clients or the game server
Response	A new game mode is deployed and should be ready for use
Response measure	The new game mode should be possible to be deployed in a couple of hours

M2 - Creating a new client	
Source of stimulus	Client developer
Stimulus	The client developer wants to create a new client for the Lecture Quiz game
Environment	Design time
Artifact	The Lecture Quiz service
Response	A new client supporting to play the Lecture Quiz game.
Response measure	The server communication part of the client should be complete within two days

M3 - Adding support for a new database back end	
Source of stimulus	Server developer
Stimulus	The server developer wants to add support for another database back end
Environment	Design time
Artifact	The Lecture Quiz server
Response	A new option for database storage in the server
Response measure	The new back end should be finished in two hours

6.2.2 Usability

This section describes some scenarios for the usability attribute. Usability is normally associated with how the systems supports the tasks it is supposed to support and how easy it is to do those tasks.

U1 - Changing server settings	
Source of stimulus	Server administrator
Stimulus	The server administrator wants to change some of the default settings (e.g. database host, username and password)
Environment	Deploy time
Artifact	The Lecture Quiz server
Response	The new settings in effect
Response measure	The new settings should be in effect without rebuilding the project. The server should only need a restart.

U2 - Getting started	
Source of stimulus	Student user
Stimulus	The student wants to join a quiz game for the first time
Environment	Run time
Artifact	The Lecture Quiz Student Client
Response	The student is ready to play
Response measure	The student should be able to join a quiz game in less than two minutes

U3 - The game should be fun	
Source of stimulus	Student user
Stimulus	The student user is playing a game of Lecture Quiz
Environment	Run time
Artifact	The Lecture Quiz game
Response	The student participating in the game
Response measure	At least 80% of the students should think the game is fun.

U4 - Deploying the Lecture Quiz Server	
Source of stimulus	Server administrator
Stimulus	The server administrator wants to deploy the game server for the first time
Environment	Deploy time
Artifact	The Lecture Quiz server
Response	The server is up and running
Response measure	The server should be up and running in less than one hour

U5 - Adding question	
Source of stimulus	Teacher
Stimulus	The teacher wants to add a question to a quiz
Environment	Run time
Artifact	The Lecture Quiz Teacher Client
Response	The question is added to the quiz
Response measure	The teacher should be able to save and add the question to a quiz in less than 1 minute

6.2.3 Performance

In this section we will describe some scenarios grouped under the performance quality. By performance we think of how well the software is running under normal conditions, as well as how the response time and other measurable performance options scale when the user mass gets bigger.

S1 - Multiple game servers	
Source of stimulus	The server administrator
Stimulus	The server administrator wants to add more game servers to ease the load of the servers
Environment	Deploy time
Artifact	The Lecture Quiz Server
Response	Multiple game servers are running with the same database for data storage
Response measure	A system should get at least a 50% performance improvement if the clients are split between two servers

S2 - Number of users	
Source of stimulus	The teacher and student users
Stimulus	The teacher wants to run the Lecture Quiz in a big class of 100 students
Environment	Run time
Artifact	The Lecture Quiz Game
Response	The game is running smoothly
Response measure	The game should run with a response time of no more than one second

6.2.4 Other attributes

As well as the previous mentioned attributes there are other attributes that are often used for quality requirements. Such attributes may be availability, testability and security. In this project these attributes are not regarded as very important and we will therefore not define specific scenarios and requirements for the three attributes. Although they are not regarded as important, they are still thought of in the design of the system.

Chapter 7

Architecture

We have designed an architecture that is built up of multiple parts. The documentation of the architecture is divided into views based on the ideas by Phillippe Kruchten in [24]. To explain how the three parts fit together we have shown the different parts in form of a logic view. The process flow and the communication between the parts are outlined in the the process view shown in Section 7.2. In addition, the physical set up is displayed in the deployment view shown in Section 7.3.

7.1 Logic View

The architecture we have decided to build our system on is based around a three part system with one game server and two different clients as seen in Figure 7.1. The two clients are separated by roles, namely the teacher client and the student clients.

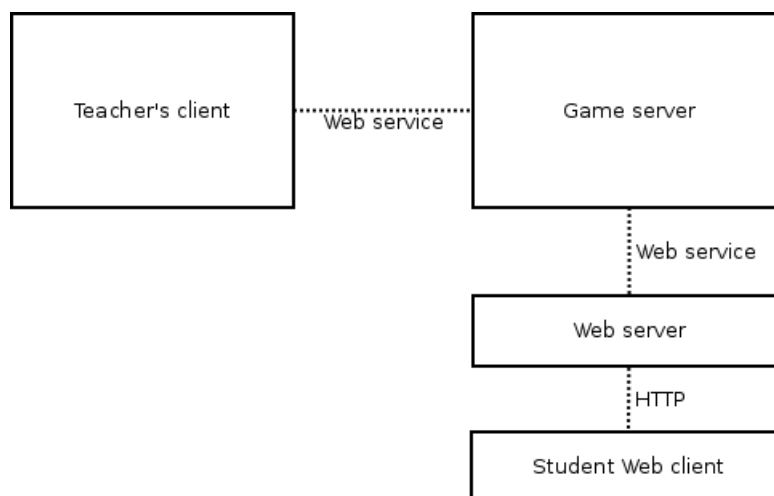


Figure 7.1: Client-server architecture

7.1.1 Game Server

As seen in Figure 7.2, the game server takes requests from clients through the web service layer of the application. Before the client is actually associated with a game, things like authentication is checked against the database. Once a client has joined or started a game, the requests are forwarded to the appropriate game mode instance and processed there.

7.1.2 Teacher Client

The teacher client runs natively on a laptop or another computer, and controls the flow of a quiz. Every quiz is started on the command from a teacher client, and this client tells whenever a new question will be started on the server. The main purpose of this client is to present the quiz and its questions to the audience which will interact with the game through student clients. A teacher is given full command over when the questions will start so he can be given time to explain any answers after a question has been answered by the students.

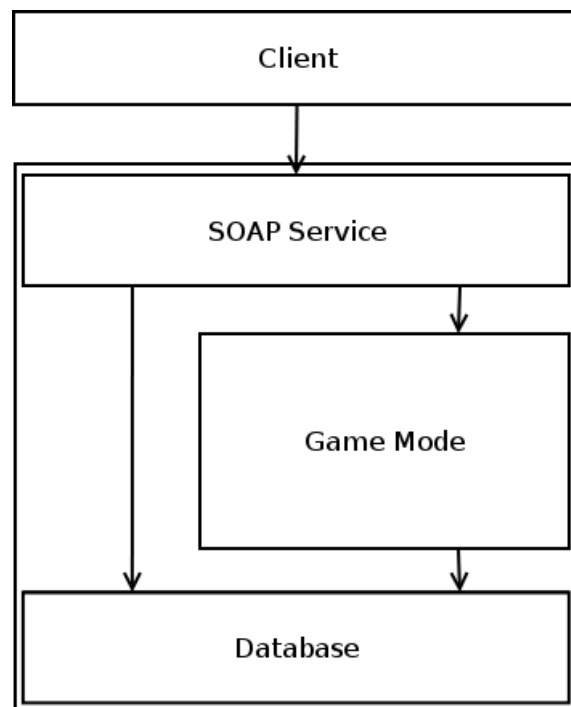


Figure 7.2: Game server architecture

7.1.3 Student Client

The student client is on the other hand a much smaller client. All it needs to do is to let the students answer the questions given by the quiz. Depending on the size of the device the student client is running on, the amount of information displayed can range from everything, including the quiz and answer alternatives, to simply 4 buttons representing each answer. To help distributing the clients, we have decided to run them on a web server. This will in practice give us two student clients, one which runs on the students' cell phones, laptops and other devices, and another one in form of the web server handling these web clients. The web server handling requests will basically forward calls to the game server and render the results in the form of a web page to the students.

7.2 Process View

All the communication between the clients and the game server goes through the web in form of SOAP requests. On the server these are received by the web service container and processed by the server application. All communication, including threading and session handling on the server, is managed by the web service container.

7.2.1 Teacher Client

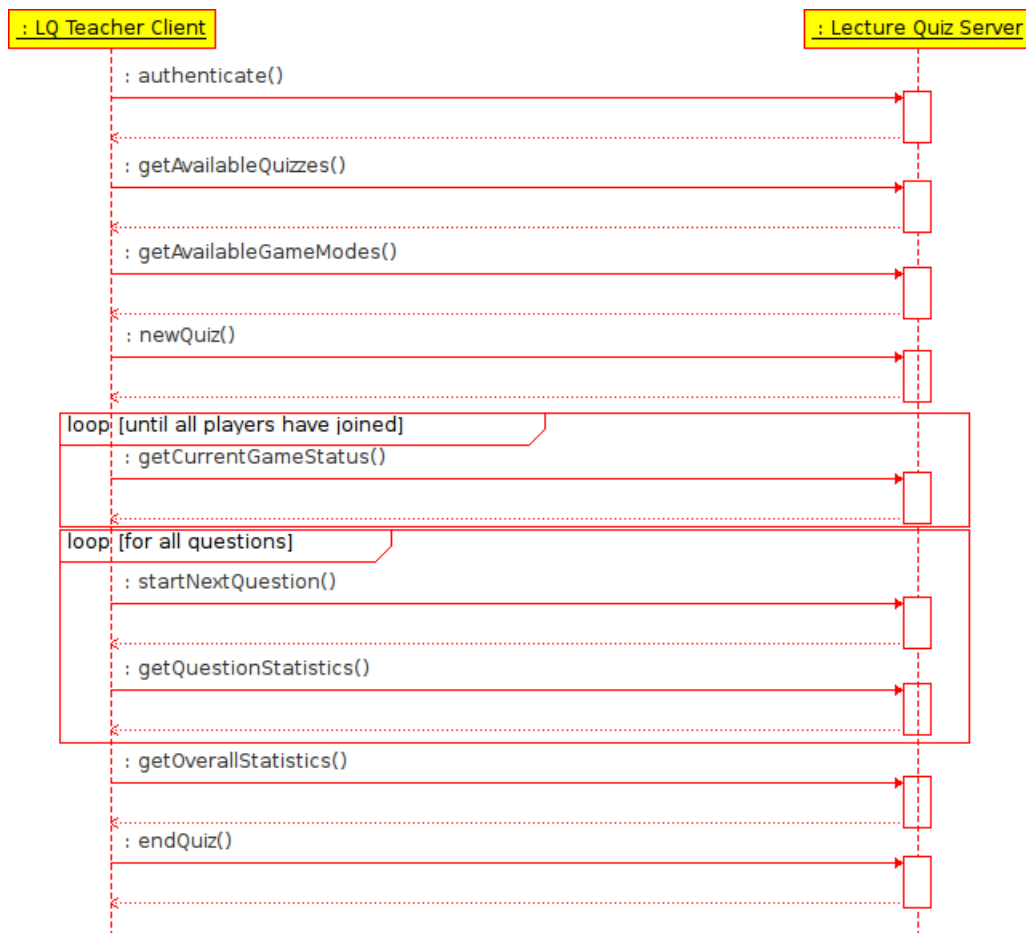


Figure 7.3: A typical teacher client session

Figure 7.3 shows the communication between the Lecture Quiz service and a teacher client. As shown, the session is started by a call to `authenticate`. This tries to validate the supplied username and password before it sets up a valid session. If the authentication fails, an error code is returned and the communication halts.

The next step is to get a list of all available quizzes and game modes by calling `getAvailableQuizzes` and `getAvailableGameModes`. These steps can be omitted if the calling teacher client already knows the IDs of the quiz and game mode they want to start.

As a final step before the game begins, the teacher client calls `newQuiz` which starts a new quiz on the server with the supplied game mode and quiz. A quiz code is also supplied as a reference for student clients that want to join this game. If for some reason the quiz or game mode does not exist on the server, an error code is returned and the creation of the quiz is aborted.

Once a game has been started, the teacher client should typically display a welcome screen and can call `getCurrentGameStatus` to get the current number of players that have joined as well as information on how many questions the quiz has and its quiz code. This call can be repeated to get updated information. When enough players have joined the client calls `startNextQuestion`. This starts the first question in the quiz, and starts the count down on the server. Any answers submitted must be submitted before this countdown has been reached. The teacher client renders the question returned from the call to `startNextQuestion`, and waits until the timeout is reached.

Once a question has timed out, the teacher client calls `getQuestionStatistics`. This will return general statistics on the answers of the previous question, like how many selected the different answers and what the correct answer was. If `getQuestionStatistics` is called out of order, an error code is returned. This can happen if it is called before the question timeout has been reached.

The procedure of calling `startNextQuestion` and `getQuestionStat-`

istics is repeated until the quiz is out of questions. When this happens `startNextQuestion` will return an error code telling the client that there are no more questions available for this quiz. The client then calls `getOverallStatistics` which will return general statistics for the entire quiz.

When the teacher client is done, it will call `endQuiz` to free up resources on the server and abandon the quiz. This will prohibit any other student clients to access the quiz as well. If the teacher wants to start a new quiz the sequence is repeated from `newQuiz`.

7.2.2 Student Client



Figure 7.4: A student client session

The session of a student client starts in the same way as a teacher client session, as

shown in Figure 7.4. The client first calls `authenticate`, however the student client does not supply a password and doesn't ask for real authentication, just a name check. This will check if the supplied name is available and not already in use. If the name is usable, a valid session is created, otherwise an error is returned.

After the client is authenticated with the server, it can now join a quiz. To be sure that the game mode ran by the quiz is also available on the student client, we call `getGameModeInfo`. This will return information of the game mode of the given quiz. After this check the client is ready to join the quiz. This is done by calling the `joinQuiz` procedure and supply the quiz code of the desired quiz. If the quiz is available and open, the client will be registered with the quiz, otherwise an error will be returned.

When the client has joined a quiz, it should start to poll for a question. Since this architecture is based around a web service and the student clients are run on the web, we cannot directly notify them when the quiz has moved passed the welcome screen and started a question, so the student clients will have to poll by calling `getCurrentQuestion`. This will either return the current question or return an error code telling the client that the question is not available yet and it should try again in a few seconds. If another error occurs, the client should abort.

After the question is received, the client has to submit an answer within the time limit of the given question. This is done by calling the `submitAnswer` procedure. If the answer is accepted, it will be registered on the server, otherwise an error will be returned.

After the question is answered, the client should then start to poll for statistics. This is done by calling `getQuestionStatistics`. As with the question call, this will return an error code telling the client that statistics are not available yet until the question has timed out on the server. The client should keep on calling it until the question has reached its time limit. The statistics are then returned and will tell if the user answered correctly or not.

As with the teacher client, the procedure of calling `getCurrentQuestion`

and `getQuestionStatistics` are repeated for all questions. When the last question has been reached the appropriate error code will be returned and the `getOverallStatistics` method is called to display overall statistics for the client. At this point the client can be given the ability to join another quiz by entering a new quiz code.

7.3 Deployment Views

In this section we will describe how the Lecture Quiz can be deployed on different server and client hardware. We will point out three ways the Lecture Quiz Game may be deployed. It is also possible to use variants of these models when deploying.

7.3.1 One Server

The easiest way of deploying the Lecture Quiz game is to use one physical server as shown in Figure 7.5¹. Both the main service and the server part of the student client will run on this server, as well as the database system. This will make the delay of transporting data between these parts as small as possible, though this one physical server will take all the load from running the game. The clients will connect to this central server.

7.3.2 Three Servers

The next way to deploy the Lecture Quiz Game is to use one physical server for each part of the game. That means that there is a need for three servers, as shown in Figure 7.6. One hosting the database system, one hosting the main

¹All the pictures used in the deployment figures are collected from iconseeker.com, being licensed under LGPL, Creative Commons or other free licenses.

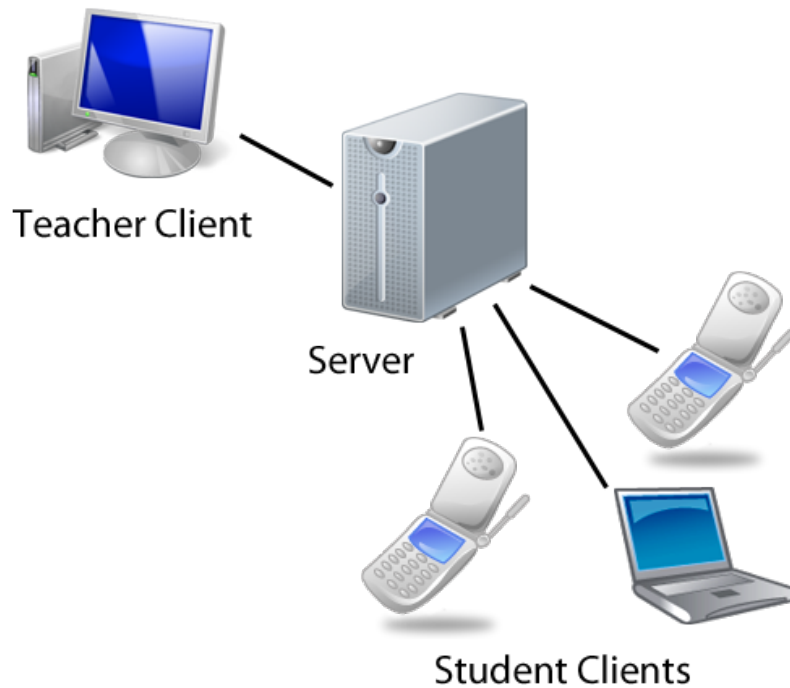


Figure 7.5: A simple setup of Lecture Quiz

service and one hosting the server side of the student client. The teacher client will then connect to the game server using the web service, and the student clients will connect to the web server using plain HTTP and HTML or AJAX². This solution will split the load in some way, but increasing the time spent on data transfer. Many organizations already have their own database server, so this way of deployment will make use of this advantage.

²Asynchronous JavaScript and XML

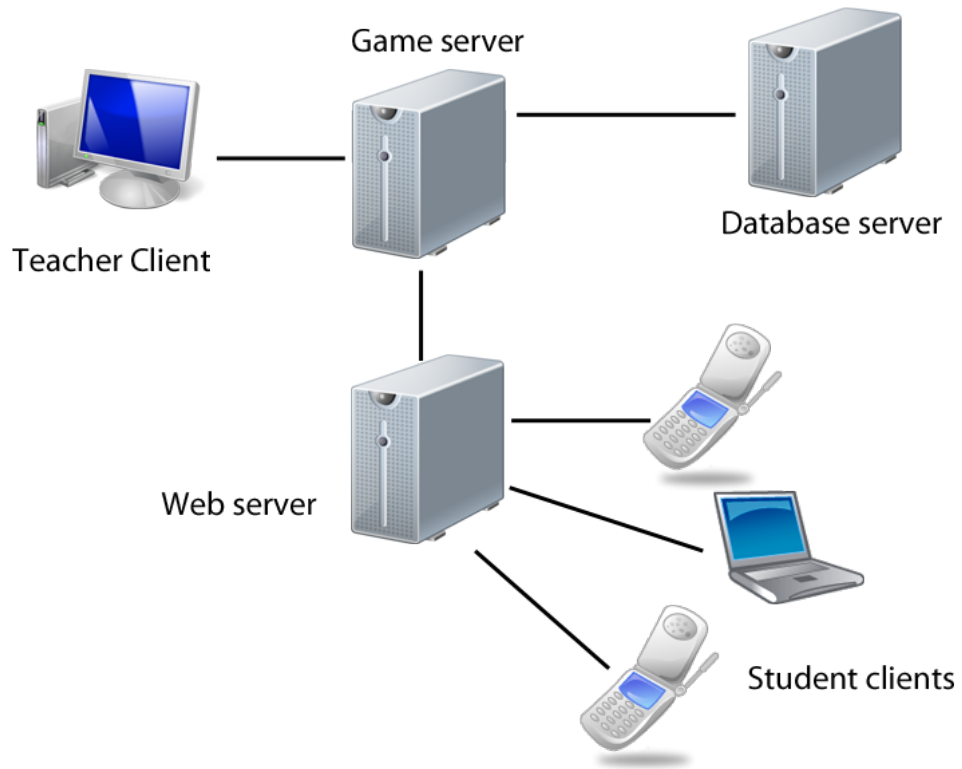


Figure 7.6: A setup of Lecture Quiz with three servers

7.3.3 Multiple Game Servers

If the Lecture Quiz Game is to be used in more large scale situations, it may be smart to divide the load even more than in the three servers solution. In this solution, shown in Figure 7.7, we have introduced multiple game servers using the same database server. With this solution it is possible to let a few quiz games run on different game servers, and still use the same database server. The clients then have to connect to the corresponding game server or web server. As the game server collects all data about a quiz from the database once a game is started, the Lecture Quiz Game will be able to handle a lot more simultaneous games than the other deployment solutions.

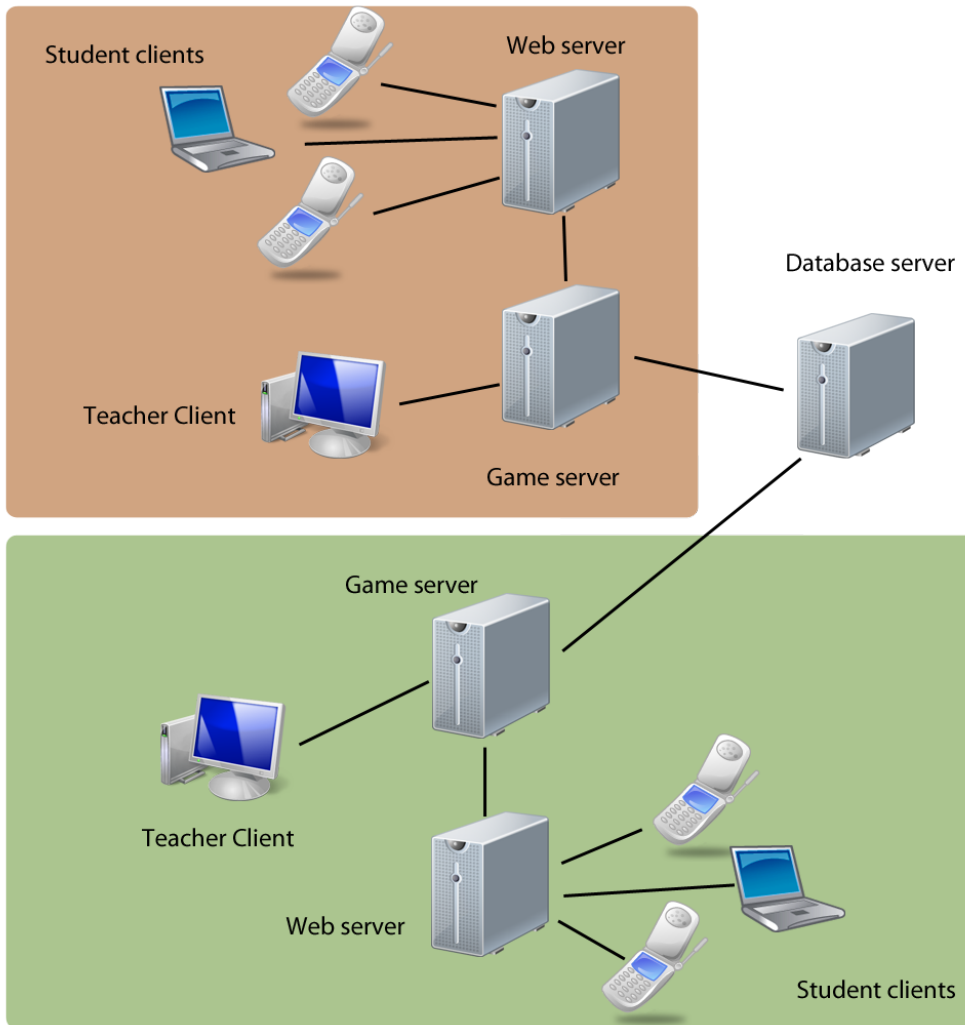


Figure 7.7: A setup of Lecture Quiz with multiple game servers

Chapter 8

Chosen Technologies

In this chapter we will look at the technologies presented in chapter 5, and argue about which of them to choose for our solution. These technologies will be used in the implementation described in Chapter 9.

8.1 Programming Language

Microsoft's .NET platform has been gaining popularity strongly over the last years, but our choice still falls on the Java platform. The main reasons for this is the usage of the language on NTNU, both among courses and the students. By choosing Java, it will be easier for students who do not have the experience with the .NET platform to continue development on this project. Java is the language taught and used in many courses at our institute, IDI¹. An other reason for our choice of Java is the multiplatform support. Even though the .NET platform can run on multiple platforms with the help of projects such as Mono, the Java platform is more widespread and has better support and maturity in this matter.

¹Institutt for Datateknikk og Informatikk (Department of Computer and Information Science)

8.2 Graphics Library

Once the choice of programming language fell on Java, our graphics library mostly chose it self. Even though you can access the Direct3D library from Java with the help of 3rd party wrappers, the multiplatform aspect is lost if this approach is chosen. This is why we went for the OpenGL library and the Java implementation JOGL.

8.3 Web Deployment

As our institute IDI already have Tomcat servers running and the fact that we already had some experience using Tomcat, this was our choice of web deployment software during the implementation. The application will most likely be easy to deploy on the other mentioned web containers as well.

8.4 Communication Framework

Developing a new communication protocol for the Lecture Quiz game would probably had taken to much time compared to the advantages we had gained. Choosing a web service as our communication framework lets us use a lot of the provided parts already from day one, and it will make it a lot easier for others to extend and implement new clients for our system. As we chose to use web services for communication, we are left with two possible options, REST or SOAP.

Using REST and RESTful development is popular these days, but the support for REST has not grown as far as we had hoped. The support for SOAP in libraries, programming environments and other supporting tools makes our choice of communication framework quite clear. SOAP's use of the WSDL file is a huge advantage and the support for auto-generation of java classes based on the WSDL

makes development far easier.

8.5 Database System

While both database systems have pros and cons we went with the MySQL database system. This is mostly due to the fact that it is MySQL we have the most experience with and NTNU is already running MySQL on some servers. Despite our choice, the architecture is designed to make it easy to implement support for other DBMS's.

Chapter 9

Implementation and Design

In this chapter we will describe how we have implemented the architecture presented earlier. The main component in this architecture is the Lecture Quiz Game Service, and it is therefore the component we have had the most focus on. The clients are implemented as flexible components that are easy to extend and further develop, although they are regarded as proof of concept and are meant as a guidance for others in making of new and richer clients.

9.1 Lecture Quiz Game Service

The Lecture Quiz Game Service is the server component that handles all the game logic. Both teacher and student clients connect to this server through its web service. The server itself is implemented in Java EE 6 and was running on the Apache Tomcat application server during development, but should be able to run on any Java web container.

Figure 9.1 shows the class dependencies in the Lecture Quiz Server. All requests are handled by the `LectureQuizService` class which does authorization checking and passes on the requests to the proper classes. The four manager

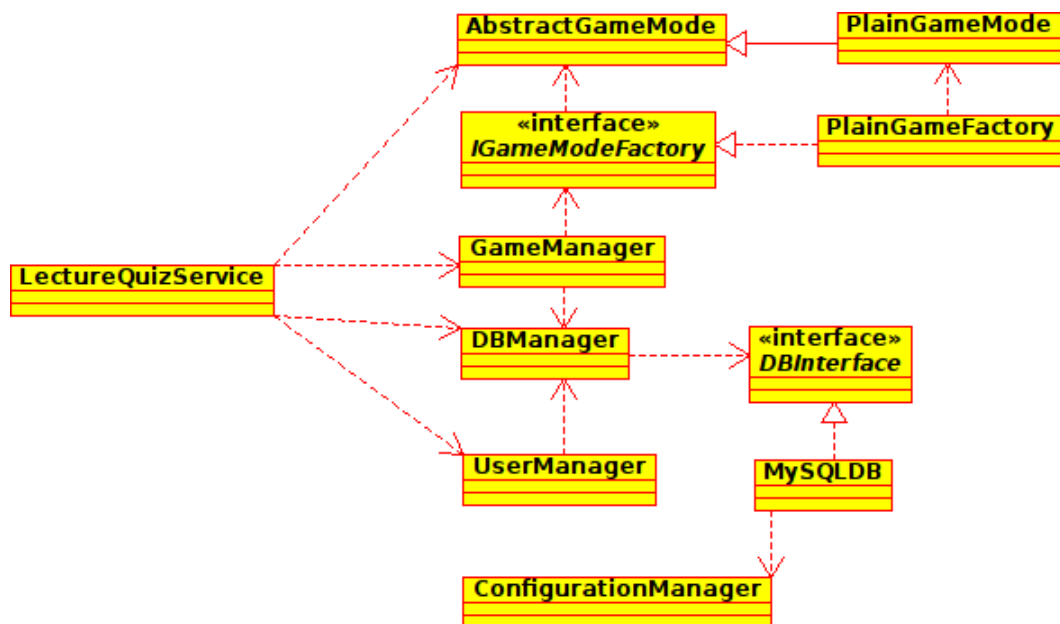


Figure 9.1: Core class dependencies of the Lecture Quiz Server

classes, GameManager, UserManager, DBManager and ConfigurationManager, are singletons with a shared instance between all clients. Most of the procedures exported by the LectureQuizService class requires an established session with the server through the authenticate procedure, with the exception of authenticate itself, getAvailableGameModes, getGameModeInfo and getServiceVersion.

The UserManager class is responsible for keeping track of all the users logged in on the server. It is implemented very simplistic with a list of all users and methods to manipulate this list.

The creation of new games is handled by the GameManager. When a request for a new game comes through the LectureQuizService class, GameManager checks for availability of game modes and quiz codes before it creates an instance of the appropriate game mode class. The instantiation of a game is done by the game mode factory of the selected game mode. Every game mode registers a factory with the GameManager class, which lets it create new instances when it needs to. Every game mode is identified by a string ID. Except for the requirement of being

unique, this ID has no restrictions on format.

Access to the persistent database is done through the DBManager class. This class creates a new instance of a class implementing the DBInterface interface. This makes switching back ends pretty easy, all that is needed is to implement this interface in a new class and tell DBManager to create an instance of the newly created class instead. When a database connection is requested, a new instance is created for the request. This makes sure that a connection is always available.

9.1.1 Configuration

The ConfigurationManager is just a simple class for holding information found in the configuration file. This is mainly where the address to the database server is located. The configuration is read from *configuration.xml* file in the the *WEB-INF/classes* directory from where the web application was deployed.

9.1.2 Game Modes

All game modes derive from the class AbstractGameMode. This class includes both abstract and implemented methods, easing some of the work to implement a completely new game mode. Many operations found in LectureQuizService are passed along to the running game mode of the client that sent the request.

The base implementation of game modes handles the question time limit by checking the elapsed time since the question was started on each call from a client. This removes the need for an active timer to mark when the time limit has been reached. The most noticeable use of this time limit check is the `getCurrentQuestion`, `getQuestionStatistics` and `submitAnswer` functions exported by LectureQuizService. All of these can behave differently whether the time limit has been reached. The server will not automatically move on to the next question before the teacher client tells it to do so. It is therefore the responsibility of the

teacher client to keep track of when the time limit is reached so it can ask for statistics and start the next question when appropriate. This can be done by reading the time left value when calling `startNextQuestion` and start a timer locally, or by calling `getCurrentQuestion` to get an updated time left value.

Plain Game Mode

In this framework we have created a very simple game mode that we call “Plain game” which is implemented by the `PlainGame` class. This game mode works as a very simple quiz where all questions are asked in order and the participants have a time limit to answer each question. When a question is answered, statistics are generated to show how the answers were distributed and which answer that was the correct one. At the end of the quiz, overall statistics are generated to represent the answers to all of the questions.

9.1.3 Quiz Editing

The ability to edit quizzes is mainly implemented by the `MySQLDB` class. The first call comes from a teacher client through the web service asking for a specific quiz to edit. The user is checked to make sure it has permissions to view a quiz by checking the session, as only teachers and administrators are permitted to retrieve entire quizzes from the server. If the authorization passes, the quiz is returned to the teacher client in form of a `FullQuizInfo` object. This object contains all the questions and answers, as well as the quiz name and other similar information. After the teacher client has done all the modifications needed, the same `FullQuizInfo` structure is sent back. This structure is then passed on to the `MySQLDB` class, which processes the information before it saves it into the database. Again, permission checking is done, and on save the permissions are even stricter. Users logged in as a teacher are only allowed to save quizzes they are the owner of or entirely new quizzes. Administrators are able to save to all quizzes. A quiz is identified as new when the ID field of `FullQuizInfo` is set to zero. This tells the

database back end class to create a new quiz instead of updating an old one. The same logic is applied to questions who are either created as new if the ID is zero or updated if it has a valid ID.

9.2 Database design

The database design can be seen in Figure 9.2. It shows the five main tables in the database. In addition to these five, we have created two reference tables that help us perform the needed relations between quizzes and questions, as well as the tags. These two reference tables are named *ref_quiz_question* and *ref_tagged*. The question to quiz relation is done by a simple table that holds both the ID of the question and the quiz in a single row. The tagged relations are a bit more advanced as we use the same table for both quiz tags and question tags, since they can both be tagged with the same tag. To perform this we have three columns, the a tag ID, a quiz ID and a question ID. In this table we always want either the quiz or the question ID to be null and only use one of them when we are searching for tags.

The *user* table contains all users that are able to log in to the teacher client. There are two notable fields here, the *password* field and the *role* field. The *password* field stores the SHA-1 hash sum of the actual password and the *role* field can be either 1 or 2. A role value of 1 means the user is an administrator and a value of 2 means the user is a teacher. An administrator has higher privileges than a teacher, most notable the ability to change quizzes the user it self is not the owner of.

All quizzes are stored in the *quiz* table. This table simply holds the name and ID of a quiz, and questions are linked up to this quiz by the use of the previously mentioned *ref_quiz_question* table. A quiz can have as many questions as it wants, and is always tied to a single owner.

The *question* table hold all questions in the database. Every question has a set of answers and these are stored in the *answer* table. Each answer holds a reference to the question they belong to and the question holds a single reference to the correct

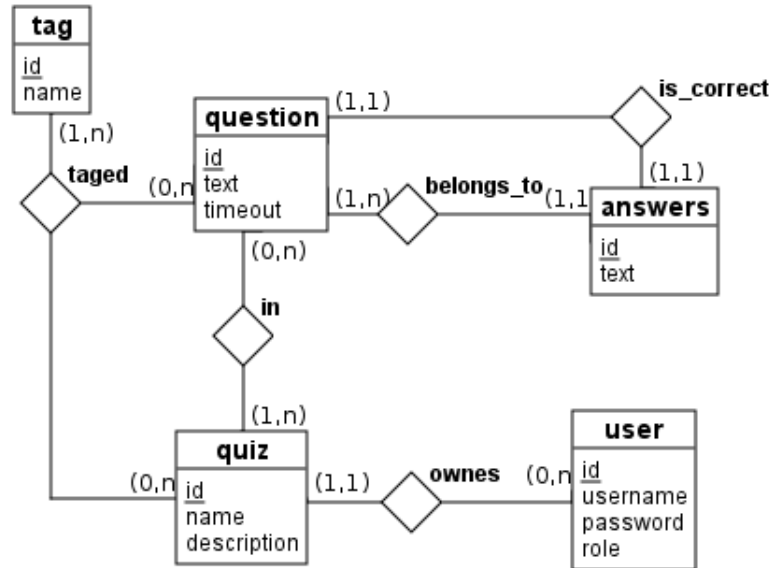


Figure 9.2: ER diagram of database

answer id. A question can be in many quizzes but answer options are specific to a single question.

The *tag* table holds a list of tags. These are just simple single words that somehow relate to the quiz or question who was tagged with it. A single tag is only stored once and all quizzes or questions refer to this tag through the *ref_tagged* table.

9.3 Teacher Client

The teacher client is developed in Java SE 6. The development has mainly focused on the functional parts of the client, leaving some of the visual aspects pretty basic. Implemented in the client is a simple menu system, a quiz editor to create and edit quizzes and questions, and a single game mode. When the teacher client is

started, a connection check is performed to make sure the application can reach the Lecture Quiz web service. This is done by querying the web service version information through the `getServiceVersion` function call. If the connection failed, the user is presented with the options to retry, cancel or open the configuration window where the server address can be changed.

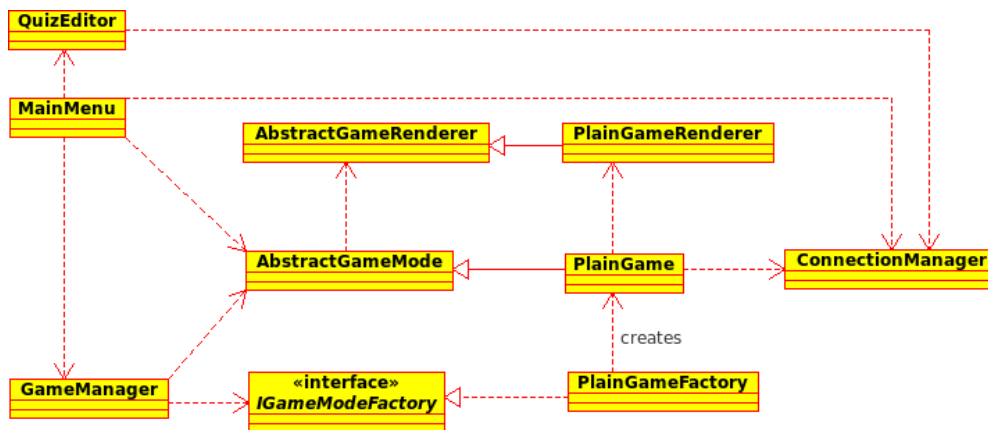


Figure 9.3: Core class dependencies of the Lecture Quiz Teacher Client

Upon a successful connection to the Lecture Quiz web server, the user is presented with the Main Menu view, handled by the `MainMenu` class. This class is responsible for handling the teacher login, as well as selecting which quiz to start. Quiz selection is done by retrieving a list of available quizzes on the server, as well as getting a list of supported game modes the server is capable of starting. The client then compares the server's game mode list with its own, and displays a list to the user where only the game modes available to both the teacher client and the server are shown. The `MainMenu` class is created as a singleton so game mechanics classes can easily call up this class and return to the main menu when the game is complete, should the connection be broken or another error occurs.

All communication with the Lecture Quiz Server goes through the web service `LectureQuizService`. The connection to this service is handled by the `ConnectionManager` class. When this class is accessed, a connection to the service is established with the help of compile time generated classes from the service's

WSDL¹ file.

9.3.1 Game Modes

As a user selects a quiz it wants to start, the `GameManager` class is called up to handle the creation of the new game. This class has all the information on which game modes the teacher client supports, and will only try to start games which are supported. A request to start a new game is sent to the Lecture Quiz server by calling the exported `newQuiz` procedure. If the server allows the creation of the game, the `GameManager` looks the current game mode up in the list of supported game modes to get the game mode factory so a new instance can be created.

All game modes on the teacher client must register a factory with the `GameManager` class. This procedure is very similar to the registration of game modes on the server. These factories are implementations of the `IGameModeFactory` interface and their purpose is to identify the game mode by name and id, as well as creating a new instance of this game mode.

The game modes are split into two different parts, the game mode it self that extends the `AbstractGameMode` class, and the game mode renderer extending `AbstractGameRenderer`. The game mode class queries the server for information about questions, statistics and other information that is needed by the specific game mode. It then processes this information before it is passed to the renderer class. The renderer class simply displays what it gets from the game mode class. These two classes are split up to make it easier for developers to extend existing game modes, and separates the logic and rendering parts of a game mode. When the renderer and the game mode is split up, a developer can choose to only extend the functionality of one of these classes and use a previous implementation for the other. This gives the opportunity for many different game modes to share the same renderer or the other way around.

¹Web Service Description Language

Plain Game Mode

As with the Lecture Quiz server, we have implemented the very simple game mode *Plain game*. The class implementing this game mode in the teacher client is also called `PlainGame`. An important part when implementing game modes for both the server and clients is to make sure they all have identical game mode IDs. If they do not, the game will not start as it cannot find a common game mode on the server and clients. This game mode simply calls `startNextQuestion` when the teacher moves to a new question and passes the information to the `PlainGameRenderer` class. It also starts a timer matching the number of seconds left on the question retrieved. The `PlainGameRenderer` then renders the question with a simple screen, consisting of four different boxes with each answer alternative and the question text at the top. A time left counter is also displayed at the top of the screen. Figure 9.4 shows a screenshot of a question in the teacher client.

Once the time limit has been reached, the `PlainGame` class requests statistics for the previous question. It then passes this information on to the `PlainGameRenderer`, which displays this in the form of four histograms representing the submitted answers for all alternatives. The text of each answer alternative is shown to the right as seen in Figure 9.5 and the correct answer is lit up in bright green. The next question is not started until the user either presses the space bar key or clicks the *Next question* item in the teacher client *Game* menu.

The `PlainGameRenderer` uses the JOGL library. The JOGL project is the development version of the JavaTMBinding for the OpenGL[®]API detailed in the JSR-231²[25]. This library provides full access to the APIs found in the OpenGL 1.3 - 3.0, ≥ 3.1 , ES 1.x, and ES 2.x specifications as well as nearly all vendor extensions[26], however due to the simple nature of the graphical user interface of this game mode only OpenGL 1.x is used. This makes the requirements to run this game mode very low and it can be run by most of today's computers.

²Java Specification Request

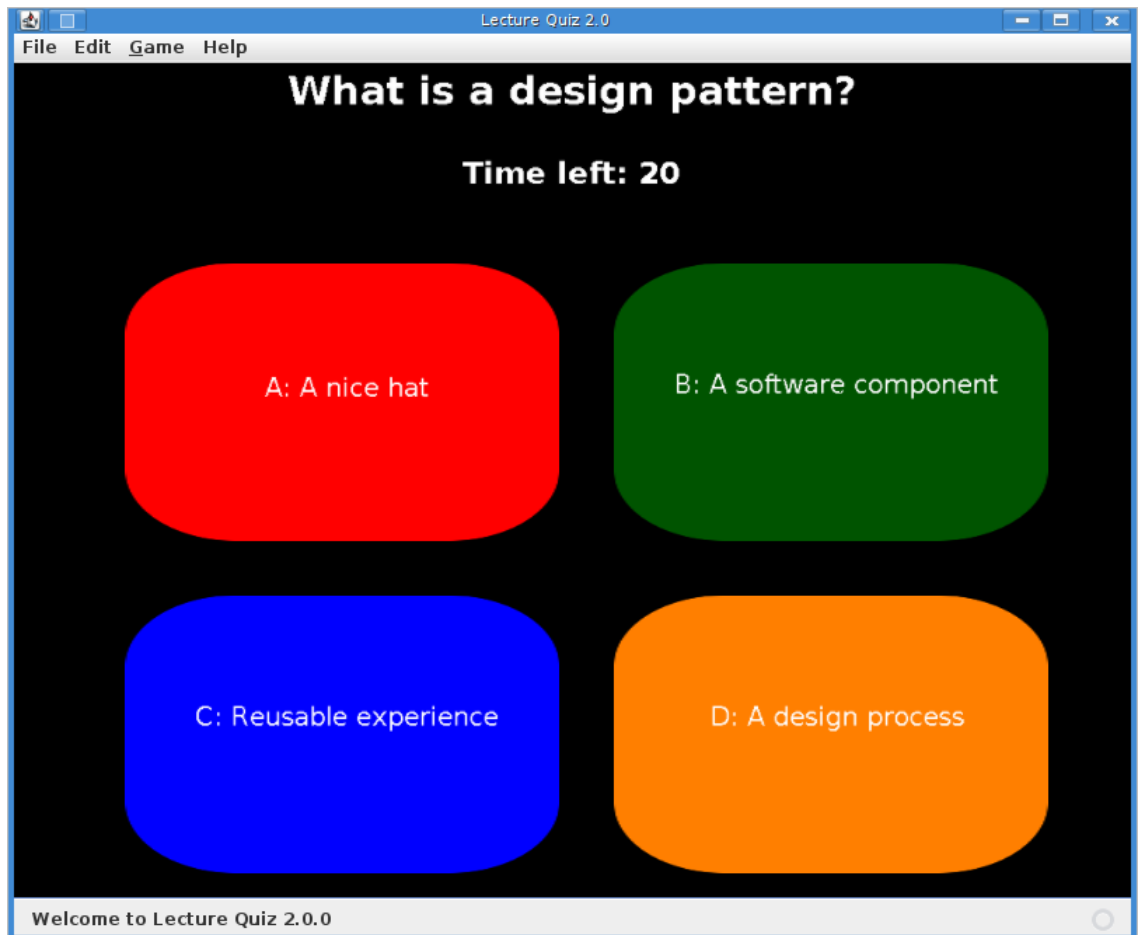


Figure 9.4: Screenshot of a question in the teacher client

9.3.2 Quiz editor

The quiz editor is implemented in the `QuizEditor` class. On load, it retrieves a list of all editable quizzes and display them in a list so the user can pick one they want to edit. A user can also choose to create a completely new quiz. The `QuizEditor` class is assisted by the `AddQuestionDialog` class to simplify adding new or existing questions to a quiz. When editing a quiz, the entire quiz is requested from the server and returned through the `getQuiz` web service procedure. The modifications are then done by the GUI in the `QuizEditor` class before the entire quiz is sent back to the server for saving by calling `saveQuiz`. Any new questions have

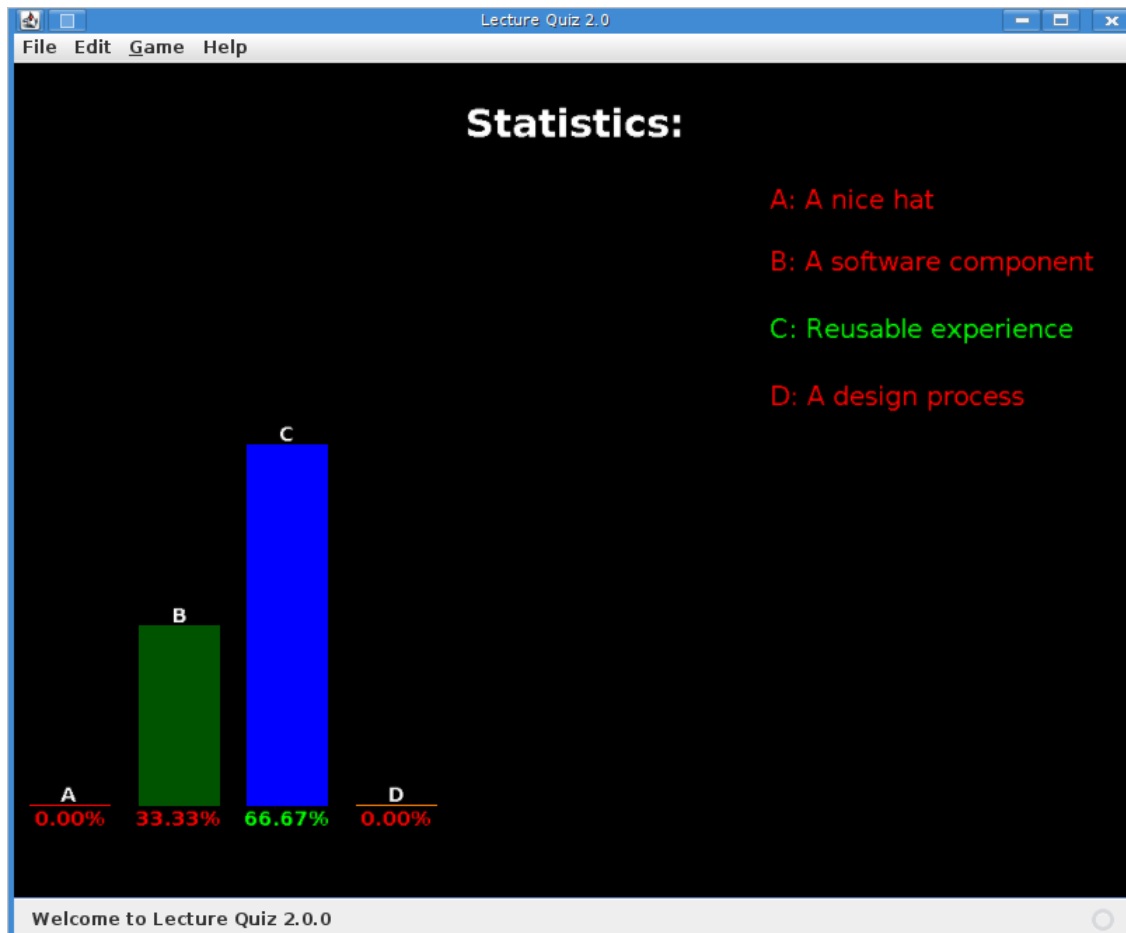


Figure 9.5: Screenshot of question statistics in the teacher client

the ID of 0 and will be created when saving. If a new quiz is created the same structure is sent, but the quiz ID is set to zero.

9.3.3 Configuration

Configuration of the teacher client is mainly for the web service connection. The configuration element consists of two classes, the ConfigurationManager class and the ConfigurationDialog. Configuration editing is done by the ConfigurationDialog class, which can be accessed through the *Edit* menu in the main application

window. The only editable field is the web service URL. When configurations are saved, a connection test is done by the `ConnectionManager` before the actual configuration is saved to disk so the user can make sure the web service URL is correct and a connection can be established. Saving and loading the configuration data is done by the `ConnectionManager` class with the help of the Java Preferences API, which automatically finds the correct save path for the operating system the application is running on. The `ConnectionManager` class is a singleton and configurations are read from the disk on application load and saved when they are changed by the configuration dialog. Any class can request a configuration variable from the connection manager.

The other configuration option found is the *text renderer mipmap*³ option, which tells the graphical part how to render text. This option is for advanced users only and cannot be changed from the configuration dialog.

9.4 Student client

The student client is developed in Java using the Google Web Toolkit⁴ (GWT) as AJAX⁵ framework. As with the teacher client, the main focus of this implementation has been on functionality and providing a reference as of how a client can be implemented. Hence, the graphical design is somewhat minimalistic.

In Figure 9.6 an overview of the classes in the student client is displayed. Each class will be described in the following paragraphs. The classes are divided into two main parts, one part running as a servlet on a java web container, and one part running in the user's web browser supported by javascript, HTML and CSS. Communication between the two parts is provided by GWT-RPC. In the class diagram, the `LQService` interface, its implementation, and the `ConfigurationManager`, are the only classes running on the web container. All other classes are compiled to

³Mipmapping is a texture filtering method used in 3D graphics

⁴<http://code.google.com/webtoolkit/>

⁵Asynchronous JavaScript and XML

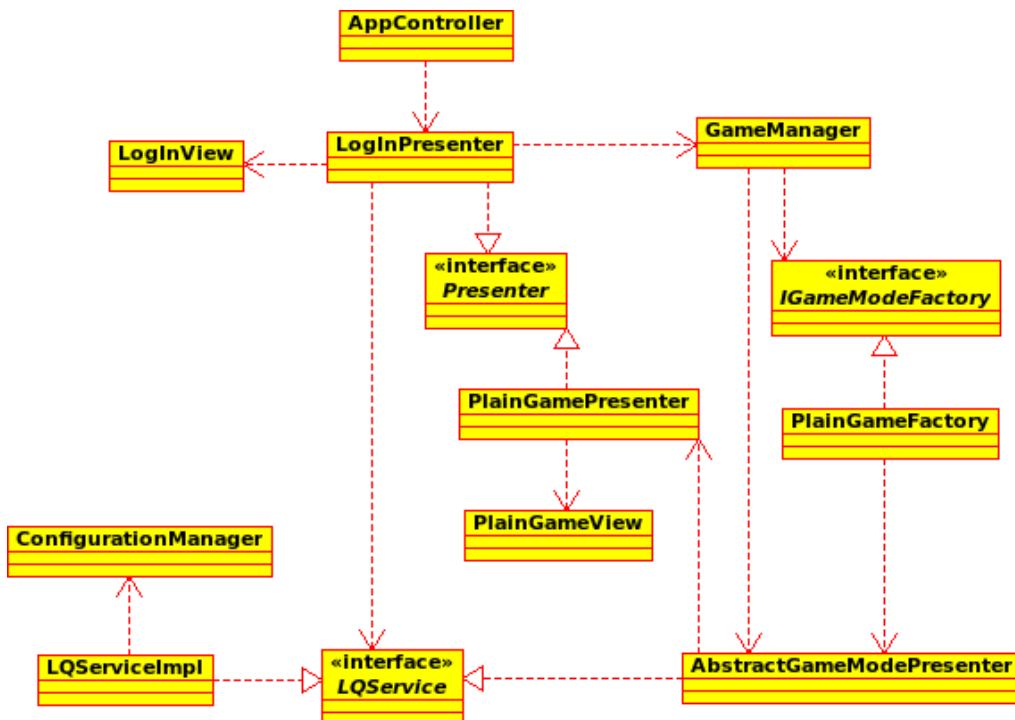


Figure 9.6: Core class dependencies of the Lecture Quiz Student Client

javascript for use in the web browser. As GWT lets us write code in Java and compile it to javascript, we will only present the java implementation in this section.

The server part of the student client is quite basic. It is mainly a proxy for the Lecture Quiz Service, providing support for GWT-RPC. The LQService interface is used by the client part for making asynchronous RPC calls to the methods exposed by this interface. The LQServiceImpl class handles the incoming requests and relays most of them to the Lecture Quiz Service.

9.4.1 Model View Presenter

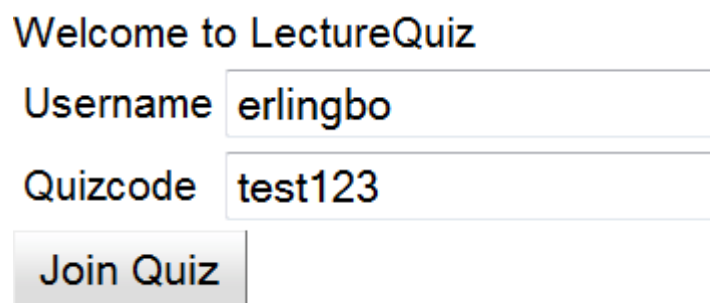
In the implementation of the Lecture Quiz student client, we have followed the guidelines from Google [27] regarding use of the Model View Presenter (MVP) architecture. In this way it is possible to keep the logic separated from the data and

display, and it is therefore possible to change one part without the need to change the others, as long as the interfaces are the same. Each presenter implementing the Presenter interface, is responsible for doing some logic and deliver the computed data to a view to be displayed. The view may be different depending on the user's device, e.g. mobile phone, smart phone or computer.

9.4.2 Application control

When a user opens the student client in a web browser, the ApplicationController is started. This class is responsible for the main flow of the application. As our implementation is quite basic, the ApplicationController only starts the log in process by starting the LogInPresenter. In a richer client with several parts in the user interface, the ApplicationController will have a bigger role.

The LogInPresenter handles the login sequence using the LogInView to display input fields for nickname and quiz code, as shown in Figure 9.7. After the user has successfully joined a quiz, it asks the GameManager for the correct subclass of AbstractGameModePresenter, and starts it. This is done by asking the server for the identifier of the running game mode, and providing this id to the GameManager. This is described in detail in Section 9.4.4.



Welcome to LectureQuiz

Username

Quizcode

Figure 9.7: Screenshot of the login user interface of the student client

9.4.3 Configuration

The ConfigurationManager in the Student Client works in exactly the same way as the same class in the service implementation described in Section 9.1.1. The ConfigurationManager class is a singleton that reads from the configuration.xml file located in the *WEB-INF/classes* directory from where the application was deployed. The student client configuration holds information about the address to the service and how long the session timeout should be.

9.4.4 Game Modes

The game modes in the student client are all subclasses of AbstractGameModePresenter. As with the other Presenter classes, they process the logic of a game mode and gives the data to a corresponding view for it to be displayed.

The GameManager and GameModeFactory in the student client works mostly in the same way as the corresponding classes in the service described in Section 9.1. The main task of the GameManager is to return an implementation of AbstractGameModePresenter from a given id. The GameManager uses the correct implementation of GameModeFactory to create a new instance of a GameModePresenter. All supported game modes needs to have a GameModeFactory that is listed in the GameManager class.

PlainGame

The student client has support for one game mode, namely the implementation of the PlainGame described in Section 9.1.2. This is implemented in PlainGamePresenter which uses the PlainGameView as user interface.

After the PlainGame has started, it starts to ask the server for the current question of the given quiz game. This polling is done with an interval of 2 seconds. When a

question is returned from the server, it displays the question text and four colored answer buttons, as shown in Figure 9.8. When a user submits his answer the game mode starts to ask for statistics for the current question, with the same 2 seconds interval. After statistics are shown it starts to poll for the next question.

What is a design pattern?



Figure 9.8: Screenshot of the user interface for displaying questions in the student client

This procedure continues until there are no more questions and the user will get an option to display end statistics. Then the game mode asks for end statistics, displays it, and the game has ended. They user may then refresh the web page to join a new quiz game.

Chapter 10

User's and Developer's Guides

In this chapter we will explain how an end user can get the system up and running for use in lectures. In Section 10.1 we go through the needed steps to get the Lecture Quiz service, the teacher clients, and the student clients up and running.

We will also explain how a developer can use this architecture and extend it with new functionality. This includes a guide explaining how to create a new game mode for the Lecture Quiz server as well as for the clients in Section 10.3. With the explanations in this chapter, a developer wanting to continue the development of this project should have all the information needed to write new game modes for all clients. The developer should also be able to write completely new clients should the need arise.

As the ability for further development and expansion of the Lecture Quiz framework is an important part of our work we have decided to include detailed information on how this can be done. This information is intended for new developers wanting to pick up the Lecture Quiz system and continue development on the many aspects of it.

10.1 Deployment

This deployment section explains how all Lecture Quiz system components can be deployed in order to get a fully working system up and running. Some previous knowledge of MySQL and the Apache Tomcat server is preferred. The first component that should be deployed is the Lecture Quiz service and its database, before the clients are deployed.

10.1.1 Lecture Quiz Service

The deployment of the Lecture Quiz service is done on a Java web container. During the development of this service we used Apache Tomcat, however the application should work on other Java web containers as well.

The deployment of the Lecture Quiz service application on to a Apache Tomcat server consists of two steps, WAR file deployment and configuration. First open up the Tomcat Web Application Manager. This is usually found as the */manager* application on a default installation of Tomcat. For more information about configuring access to the manager web interface see the Apache Tomcat web page¹. In the management web interface select the *LectureQuiz.war* in the *Deploy* section to upload and deploy the war file to the server. This *LectureQuiz.war* file can be found in the attachments under the *dist/LectureQuiz* directory.

When the Lecture Quiz application is deployed, the configuration file needs to be changed on the server. This file is located in the path *WEB-INF/classes/configuration.xml* under the deployed application. To find out where your application is deployed, refer to the Tomcat server configuration. There are four options in this file that needs to be changed in order to get the web service to work. The field *DatabaseHost* tells where the web service can locate the database server. The *DatabaseDB* field tells which database to use. *DatabaseUser* and *DatabasePa-*

¹<http://tomcat.apache.org/>

ssword sets the user and password to log in with respectively. Once these four settings are configured correctly, the service can be used. For more information see the following section on how to setup the database.

10.1.2 Database Setup

The database server we have chosen for our back end is the MySQL server. Another server can be used, but it requires a new database back end class in the Lecture Quiz server to be written, so we will only discuss MySQL in this section. It is suggested to create a separate user that the web service will use to connect to the database, but this is not required.

The creation of the database structure can be done by importing the SQL file found in Appendix B. This file can also be found as *database.sql* in the attachments under the *dist/LectureQuiz* directory. By importing this file, a database called *lecturequiz* will be created and all the tables needed will be created under this database. If this is not the desired database name, it can be changed by modifying the two first SQL lines in the SQL file. These two lines can be removed if the importing is done directly into an existing database. In addition to tables, an administrator user will also be created with the username “Administrator” and the password “admin”. The password can be changed by updating the SQL. It is important to remember that the *password* field in the *user* table is the SHA-1 sum of the password. There is currently no custom user interface to create new users, so this must be done by directly inserting users into the database the same way the administrator is inserted in the Appendix B file. For more information about the table fields see Section 9.2 on database design.

10.1.3 Teacher Client

The teacher client needs Java 1.5 or later to be installed on the target machine. In addition to this it needs the JOGL library, however this is bundled with the Win32

and Win64 ZIP packages. These ZIP packages can be found in the attachments as the *dist/LQTeacherClient/LQTeacherClient_win32.zip* and *dist/LQTeacherClient/LQTeacherClient_win64.zip*. The Win32 package is to be used on 32-bit Windows systems, while the Win64 is meant for 64 bit systems. The last file in that directory is the *LQTeacherClient_all.zip*. This file is meant for Linux, Mac OS X and other users but does not contain the binary JOGL files. The latest JOGL binaries can be found at their download page². The only operation needed for the Windows install is to extract the correct ZIP file for the target architecture and start the application with the BAT-file that will be extracted. This should start the application.

A small window saying “Connecting to server” should pop up and will probably hang for a few seconds while the application tries to connect to the default *localhost* server. If the connection fails, a warning box will appear and give the option to open the configuration window. Here the correct web service url can be inserted. To get the correct url open the page where the Lecture Quiz web service has been deployed. On this page there is a link to the WSDL file and it is this link that should go in the configuration box. Once the correct URL is entered, pressing OK should return the view to the login window. At this point the application is ready to be used.

Should the application start up correctly, but not start a quiz when the *Start quiz* button is pressed, it is most likely due to missing JOGL binaries. Make sure the binaries can be found by the application by having them in the current or system path.

10.1.4 Student Client

The deployment of the Student client works in the same way as for the Service as described in Section 10.1.1. To get the student client up and running you have to deploy it from the included *LQWebClient.war* located in the *dist/LQWebClient*

²<http://download.java.net/media/jogl/builds/archive/>

directory in the attachments.

After the deployment you should edit the configuration located in *WEB-INF/classes/configuration.xml* in the path of the deployed application. The most important configuration is the address to the Lecture Quiz Service, which should point to the provided WSDL file. You may also edit the time limit before sessions expire.

10.1.5 Developer Environment

In this project we have used Netbeans as our primary development environment. In the attached code there are Netbeans project files included which should give an up and running development environment without much configuration except for adding a few libraries. These libraries are the JOGL library and the GWT library. Once these libraries are installed on the system the projects can be opened in Netbeans and the missing library problems can be solved by pointing to the installed JOGL and GWT libraries. For more information on installing these libraries see their respective documentation.

10.2 The Lecture Quiz Service API

The Lecture Quiz Service API is an important part of our architecture, as all communication between the server and clients go through it. We will therefore provide a detailed description of all exported procedures³ and data structures, as well as examples on how to use these in Appendix C.

This API is used both when creating new game modes on the clients as well as entire new clients. The game modes on the Lecture Quiz service responds to calls through this API, so information about the expected behavior of these procedures are preferred when creating new game modes both the server and the clients.

³A procedure is similar to a Java method

10.3 Creating a Game Mode

This section will go through the creation of a new game mode for all components of the Lecture Quiz architecture. The most important part here is the creation of the actual Lecture Quiz Server game mode, as this is where most of the logic is. The most of the work on the clients go towards showing the data returned from the server in a meaningful way.

10.3.1 Lecture Quiz Service

When creating a game mode for the Lecture Quiz Service, the most essential part is the `AbstractGameMode` class. All game modes derive from this class or a subclass of it. We will explain how this class can be used to create a new game mode.

The factory class

In addition to the actual game mode class, we need to create a factory class that can instantiate a new instance of our game mode class. To create a factory we implement the `IGameModeFactory` interface shown in Listing 10.1.

Listing 10.1: `IGameModeFactory`

```
public interface IGameModeFactory {  
    public String getGameModeID();  
    public String getGameModeName();  
    public AbstractGameMode createInstance(Quiz quiz,  
        String quizCode);  
}
```

The two naming methods `getGameModeID` and `getGameModeName` return the string ID that will identify the game mode and a string representation of the

game mode name respectively. The ID is used when requesting a game mode through the web service layer, while the name is shown in the clients when picking or querying game modes. These two are usually implemented to return static strings identifying the game mode the factory creates.

The `createInstance` method creates a new instance of your game mode and returns it. The arguments **quiz** and **quizCode** are usually passed to the constructor of your game mode class and not touched by the factory class. A simple factory class can be viewed in Listing 10.2

Listing 10.2: A simple factory class

```
public class ExampleFactory implements IGameModeFactory {  
    public String getGameModeID()  
    {  
        return "ExampleGameMode";  
    }  
  
    public String getGameModeName()  
    {  
        return "Example_Game_Mode";  
    }  
  
    public AbstractGameMode createInstance(Quiz quiz,  
        String quizCode)  
    {  
        return new ExampleGameModeClass(quiz, quizCode);  
    }  
}
```

After creating a factory for your game mode, you need to register the factory with the `GameManager` class. This is currently done by modifying the `GameManager` constructor, as no plug-in system has been made. The registration is done by adding an instance of your factory class to the `supportedModes` list, which is a private member of the `GameManager` class. The example in Listing 10.3 shows the already existing `PlainGame` factory, as well as our new `ExampleFactory` being

added to the list of supporting modes.

Listing 10.3: Registering a game mode factory

```
// Private singleton constructor
private GameManager() {
    games = new ArrayList<AbstractGameMode>();

    supportedModes = new ArrayList<IGameModeFactory>();
    supportedModes.add(new PlainGameFactory());
    supportedModes.add(new ExampleFactory());
}
```

The game mode class

When implementing a new game mode you, can derive your class from `AbstractGameMode` or any subclasses of it. You might want to derive from `PlainGame` if you're only making small changes to the game mode. In this guide we will explain how to create a game mode by deriving from the `AbstractGameMode` class. This class contains some abstract methods that needs to be implemented, as well as some others that can be overridden if the base functionality is not sufficient for the new game mode.

Protected variables

There are a number of protected variables in the `AbstractGameMode` class. Some of these should be filled out by the deriving class and some are used by methods that can be overridden. These variables are listed in Listing 10.4 and will be explained as the methods using them are explained.

Listing 10.4: `AbstractGameMode` variables

```
protected String id;
protected String name;
protected String gameModeCode;
protected String quizCode;
```

```
protected User owner;  
protected ArrayList<User> players;  
protected Quiz quiz;  
protected boolean canJoinStarted;  
protected Question currentQuestion;  
protected int currentQuestionIndex;  
protected Date questionStartTime;  
protected boolean quizHasEnded;
```

The constructor

Listing 10.5: AbstractGameMode constructor

```
public AbstractGameMode(Quiz quiz, String quizCode)
```

The `AbstractGameMode` constructor definition is shown in Listing 10.5. This method sets the protected variables *quiz* and *quizCode* from the method arguments. These variables hold the quiz object and the quiz code sent from the teacher client that started the game respectively. It also initializes the players array and sets the variables *quizHasEnded* to false. The deriving class should call this constructor and set the *id* and *name* member variables. These represent the game mode ID and game mode name and should have the same values as the factory class for this game mode would return. The boolean *canJoinStarted* should also be set to true or false whether students can join the game after the first question has been started or not.

The setParameters method

Listing 10.6: setParameters method definition

```
public boolean setParameters(List<ParameterEntry>  
    parameters)
```

The `setParameters` method is called upon game mode creation and the parameters are sent from the teacher client when it calls the web service procedure

`newQuiz`. The default implementation in `AbstractGameMode` simply ignores these parameters, but they can be used by new game modes to take per game configuration options from the teacher client. It is not required to implement this method if the new game mode does not need any configuration parameters on creation. This method should return true if the parameters are accepted or false if something is missing and the game cannot start without it. Be aware that the **parameters** argument can be null.

The startQuiz method

Listing 10.7: startQuiz method definition

```
public void startQuiz();
```

The `startQuiz` method of the `AbstractGameMode` class is called when the teacher client starts the first question through a call to `startNextQuestion` in the web service. The base method simply sets the *started* variable to true, the *currentQuestionIndex* variable to -1 and calls the `startNextQuestion` member method. This method can be overridden to make initializations that need to be done before the quiz starts and should call the base class method or set *started* to true and call the member method `startNextQuestion` manually.

The startNextQuestion method

Listing 10.8: startNextQuestion method definition

```
public abstract boolean startNextQuestion();
```

The `AbstractGameMode` class' method definition of `startNextQuestion` can be seen in Listing 10.8. This is an abstract method and needs to be implemented by all new game modes. The purpose of this method is to mark the next question in a quiz as the active question, set the *questionStartTime* time stamp variable, point the *currentQuestion* variable to the next question in the quiz and bump the *currentQuestionIndex* up to the next value. The *currentQuestionIndex* variable holds the

index of the current question in the quiz. Depending on how you want your game mode to store and create statistics, answers from the last question might be stored as well for later use. The method should return true if the game mode was able to set the new question and false if there are no more questions in the quiz. It is also important that the protected member variable *quizHasEnded* is set to true should there be no more questions available. This method is called by `startQuiz` and the web service procedure `startNextQuestion`.

The `getCurrentQuestion` method

Listing 10.9: `getCurrentQuestion` method definition

```
public Question getCurrentQuestion()
```

The `getCurrentQuestion` method returns the current question the game is currently running. It should only return a question that is running, meaning it has been started by `startNextQuestion` and the time limit has not been reached. If there is no current question running, it should return null. The default implementation in `AbstractGameMode` returns the *currentQuestion* variable if the time limit is not reached, otherwise it returns null. It should not be necessary to override this method unless the new game mode requires special handling of the current question.

The `joinGame` method

Listing 10.10: `joinGame` method definition

```
public int joinGame(User player,  
    List<ParameterEntry> parameters)
```

The `joinGame` method is called when a player wants to join a game. The **player** argument is the user object of the player that wants to join and the **parameters** argument is custom arguments coming from the student client. It has three different error codes, 0 for success, 1 if the player is already in the game, and 2 if the join

is denied for other reasons.

The default implementation of this method checks that the player has not already joined and checks if the *canJoinStarted* and *started* member variables agree on whether the user can join or not. If everything is in order the player is added to the list of players, namely the *players* member variable. The **parameters** argument is ignored. A new game mode does not need to implement this method unless it wants to use the custom parameters sent from the student client or handle joining in any other way. If this method is overridden the new method should behave in the same way as the base method in `AbstractedGameMode`.

The receiveAnswer method

Listing 10.11: receiveAnswer method definition

```
public abstract int receiveAnswer(int answerID,  
    User player);
```

The `receiveAnswer` method is called when a student submits an answer. This is an abstract method and the new game mode is required to implement it. The **answerID** parameter is the ID of the answer submitted and the **player** parameter is the player that submitted it. This method can return four different error codes, 0 for success, 1 if the answer is not a valid answer for this question, 2 if the player has already answered and 3 if the question time limit has been reached or the first question has not been started yet. The answer should be discarded in all cases where the method does not return success.

Listing 10.12: Example receiveAnswer implementation

```
@Override  
public synchronized int receiveAnswer(int answerID, User  
    player) {  
    if(currentQuestion == null)  
        return 3;  
  
    if (!currentQuestion.hasAnswer(answerID)) {
```

```

        return 1;
    }
    else if (currentAnswers.containsKey(player)){
        return 2;
    }
    else if (timeoutReached()){
        return 3;
    }
    else{
        currentAnswers.put(player, answerID);
        return 0;
    }
}

```

The implementation of `receiveAnswer` in the `PlainGame` class can be seen in Listing 10.11. As it can be seen, it checks if the current question has not started yet and returns error code 3 if the member variable *currentQuestion* is null which. It then checks if the answer is a valid answer and if the player submitting it has already submitted an answer. At last it checks if the time limit has been reached. If it has not, the answer is stored in a member variable called *currentAnswers* in `PlainGame`. This member variable is not from the `AbstractedGameMode` class and answers need to be handled by the new game mode class. In this case *currentAnswers* is a hash map of the user and the id of the answer. This makes looking up already existing answers easy.

The `getQuestionStatistics` methods

Listing 10.13: `getQuestionStatistics` method definitions

```

public abstract ArrayList<StatisticsEntry>
    getQuestionStatistics();
public abstract ArrayList<StatisticsEntry>
    getQuestionStatistics(User user);

```

In Listing 10.13 we see the definitions of the abstract method `getQuestionStatistics` in `AbstractGameMode`. A new game mode is required to implement both definitions for this method. After each question has reached the time limit, the clients ask for statistics. This method returns the statistics in form of key and value pairs of strings. The keys are decided by the game mode and is interpreted by the clients in their game mode implementation.

The method with the **user** argument should return the statistics for the single user that is passed as the argument, while the method without arguments should return general statistics for all submitted answers of the last question. The methods can signal failure by returning *null* instead of a list.

In the `PlainGame` implementation, the method with the **user** argument only returns the correct answer ID with the key *correctAnswer*. The definition without arguments counts all the submitted answers and returns the percentage of answers for each answer option as well as the correct answer. For more details see the `PlainGame` class' full implementation of the `getQuestionStatistics` methods in the code for the `LectureQuiz` project found in the attachments.

The getOverallStatistics methods

Listing 10.14: `getOverallStatistics` method definitions

```
public abstract ArrayList<StatisticsEntry>
    getOverallStatistics();
public abstract ArrayList<StatisticsEntry>
    getOverallStatistics(User user);
```

The `getOverallStatistics` methods shown in Listing 10.14 behave very similar to the `getQuestionStatistics` methods. The purpose of these methods are to deliver the overall statistics of the entire game after the last question has been answered. These are abstract methods and any new game modes need to implement both of these definitions. The method with the **user** argument returns overall statistics for the user supplied, while the method without arguments

returns overall statistics for all users in the game. An implementation of any of these methods should fail if it is called before the last question has reached its time limit. The return value is a list of key and value pairs of strings, much in the same way as `getQuestionStatistics`. The result is read on the client who requested it and is interpreted by the game mode implementation running on the client. The function signals failure by returning *null*.

The `PlainGame` implementation of this method with the `user` argument returns the percentage of correct answers that user had. The key field of this entry is *correct* and is a value between 0.0 and 1.0. The method without an argument returns the percentage of all submitted answers which were correct with the *correct* key and the total number of questions in the quiz with the *questionCount* key. It also returns the user who had the most correct answers. The user has the key *mostCorrectUser*, while the number of correct answers this user had is stored with the key *mostCorrectCount*. For more details see the `PlainGame` class' full implementation of the `getOverallStatistics` methods in the code for the `LectureQuiz` project found in the attachments.

The removeUser method

Listing 10.15: `removeUser` method definition

```
public void removeUser(User user)
```

The `removeUser` method seen in Listing 10.15 removes a user from the game. The parameter `user` represents the user to remove. This method is called when a game is shut down to clean up users and also when a user session times out so there are no dead users left in the quiz. If a user decides to authenticate twice during a session this method is called if the user is in an active game.

The `AbstractGameMode` class implements this method by removing the user from the *players* protected member variable. A new game mode might want to override this method to clean up statistics if a user leaves in the middle of a quiz. In this case the method overriding the `AbstractGameMode` implementation should either

call the parent method or remove the user from the *players* list manually. The PlainGame implementation removes the user from the answer statistics then calls the parent method.

The `getGameStatus` method

Listing 10.16: `getGameStatus` method definition

```
public GameStatus getGameStatus()
```

The `getGameStatus` method returns the current status of an ongoing game. The return type `GameStatus` is a class with some information about the current state of the game. The `AbstractGameMode` implementation fills out this information and it should not be necessary to override this method unless the player list or question handling is done in a special manner. For more information on the `GameStatus` class see Appendix C.2.8.

Other getter and setter methods

None of the methods in Listing 10.17 should require overriding when making a simple game mode. The only getters that needs special handling is the `getNumberOfPlayers` and `getPlayers` and, this is only if the new game mode chooses to handle the player list in another way than `AbstractGameMode` does. For more information on these methods see the full method documentation found in the JavaDoc pages in the attachments.

Listing 10.17: The other getter and setter methods

```
public String getId()
public String getName()
public int getNumberOfPlayers()
public User getOwner()
public ArrayList<User> getPlayers()
public String getQuizCode()
public int getTimeLeft()

public boolean canJoinStarted()
```

```
public boolean hasQuestionTimedOut ()  
public boolean hasQuizEnded()  
public boolean isStarted()  
  
public void setOwner(User user)
```

10.3.2 Teacher Client

Creating a new game mode from the teacher client is all about presentation. There is very little logic in the teacher client, as most of this work is done on the Lecture Quiz server. When creating a new game mode for the teacher client there are three classes and one interface of importance, `AbstractGameMode`, `AbstractGameRenderer`, `ConnectionManager` and `IGameModeFactory`.

The `AbstractGameMode` and `AbstractGameRenderer` classes are almost entirely abstract, with the exception of a few protected variables and initialization in the constructors. One or both of these two classes needs to be subclassed to create a new game mode. The `ConnectionManager` class is a singleton that has one method of interest, the `getPort` method. This method returns a reference to the web service object and with it the web service procedures can be called. To get the singleton instance of this class call the `getInstance` method. The `IGameModeFactory` interface exposes a factory for each game mode the client supports.

The game mode code is split in two between the `AbstractGameMode` and `AbstractGameRenderer` classes. This is done so a new game mode can reuse the renderer of a previous game mode without the need to subclass the renderer. This will also make sure that rendering code and logic code are separated.

The factory class

Like the Lecture Quiz server game modes, the teacher client game modes also need a factory to create an instance of the correct game mode when a game is started. This interface can be seen in Listing 10.18 and work very much in the same way as the Lecture Quiz server factories.

Listing 10.18: IGameModeFactory interface

```
public interface IGameModeFactory {  
    public String getGameModeID();  
    public String getGameModeName();  
    public AbstractGameMode createInstance(JComponent parent  
        );  
}
```

A new game mode should create a factory that implements this interface and then register this game mode with the `GameManager` class. The `getGameModeID` method should return a string representation of an ID that identifies the new game mode. The `getGameModeName` method should return a human readable string representation of the game mode's name. These two methods are usually implemented to return static strings. The last method `createInstance` should create an instance of the new game mode. An example implementation of a factory can be seen in Listing 10.2 in Section 10.3.1. That example implementation shows a factory for the Lecture Quiz service, but a factory for the teacher client would be identical with the exception of the `createInstance` parameters.

After the factory has been created, it needs to be registered with the `GameManager` class. This can be done by adding an instance of it to the *supportedModes* member variable in the `GameManager` class. This is usually done in the constructor. An example adding the `PlainGameFactory` as well as a new `ExampleGameFactory` can be seen in Listing 10.19.

Listing 10.19: Register a game mode factory

```
private GameManager() {
```

```

supportedModes = new ArrayList<IGameModeFactory>();
supportedModes.add(new PlainGameFactory());
supportedModes.add(new ExampleGameFactory());

currentGame = null;
}

```

The game mode class

The main game mode class is the `AbstractGameMode` class. A new game mode should derive from this class. The responsibility of this class is to handle input from the teacher client GUI by responding to events. It should also handle the time limit of questions, get the question data as well as statistics from the Lecture Quiz service. Most of the work in this class goes towards getting data from the service and deliver it to the renderer. The definition of the `AbstractGameMode` class can be seen in Listing 10.20. This shows the abstract methods that needs to be implemented for the game to work.

Listing 10.20: `AbstractGameMode` class definition

```

public abstract class AbstractGameMode {
    protected JComponent parent;
    protected AbstractGameRenderer renderer = null;

    public AbstractGameMode(JComponent parent);
    protected abstract void createRenderer();
    public abstract void startQuiz(int quizId,
        String quizCode);
    public abstract void nextQuestion();
    public abstract void cleanup();
}

```

To communicate with the Lecture Quiz web service we use an object we call the *port*. This object is an instance of a class that has each web service procedure

mapped to an identically named method in this class, so calling any of the methods on the *port* object will call the procedure on the other end of the connection. We can get the *port* instance from the `ConnectionManager` class. An example of this can be seen in Listing 10.21.

Listing 10.21: Get an instance of the port object

```
LectureQuizService port =  
    ConnectionManager.getInstance().getPort();
```

The constructor

The `AbstractGameMode` constructor implementation sets the *parent* member variable to the one passed through its parameter. It also sets the parent's layout to `BorderLayout`⁴. It then calls the abstract method `createRenderer`. A subclass should make sure to either do these operations on its own or call this constructor from the new game mode's constructor.

The createRenderer method

This method is automatically called by the `AbstractGameMode` constructor and needs to be implemented in the new game mode. The objective of this method is to create an instance of a class extending the `AbstractGameRenderer` class and set the protected *renderer* member variable. It should also add this renderer to the parent in a way that is appropriate for the game mode. This is usually done by calling the `add` method on the *parent* member variable. This method is of interest if a new game mode is created by only switching out the renderer of a previously created game mode, as the new game mode can extend the previously created one and just override this method to create a different renderer.

The startQuiz method

The `startQuiz` method is called when a quiz is started from the teacher client GUI. At this stage the teacher client would usually tell the renderer to display a welcome screen of some sorts and display how many users that have joined the quiz. A game mode is not required to call the `startQuiz` web service procedure as this has already been done by the teacher client GUI in the menu screen.

⁴`BorderLayout` is a layout type in Java Swing

The PlainGame implementation of `startQuiz` starts a timer that updates the welcome screen every 4 seconds with new game information retrieved from the Lecture Quiz service. The renderer is told to show the welcome screen by calling the `showWelcomeScreen` method on the *renderer* member object.

The nextQuestion method

When a teacher clicks the *Next question* element in the teacher client's menu bar, the `nextQuestion` method of the running game mode is called. This method should call the web service procedure `startNextQuestion` and tell the renderer to display this new question on the screen by calling the `showQuestion` method of the *renderer* member object. A timer should also be started to count down and fire when the question's time limit is reached. When this timer fires, the question statistics should be requested from the Lecture Quiz service by calling the `getQuestionStatistics` procedure.

Should the `startNextQuestion` procedure call signal that the last question has been answered by returning the error code 5, the overall statistics should be retrieved from the server with a call to the `getOverallStatistics` procedure and the renderer should be told to display it by calling the `showOverallStatistics` method on the protected *renderer* member object.

For more information see the full implementation of the teacher client PlainGame class in the LQTeacherClient project code found in the attachments.

The cleanup method

The `cleanup` method is called when a game mode is shutting down. The typical task of this method would be to remove the renderer object from the protected *parent* member object. The `cleanup` method on the renderer should also be called to make sure it cleans up after it self. Other needed cleanup tasks can also be done in this method.

The renderer class

As with the `AbstractGameMode` class, the `AbstractGameRenderer` class shown in Listing 10.22 is mostly built up of abstract methods. The `AbstractGameRenderer` class also extends the `GLCanvas` class and implements the `GLEventListener` interface. This is done to enable the use of OpenGL in this class and its subclasses. This class handles calls from the `AbstractGameMode` class in order to display that information on screen. A new game mode renderer needs to implement all these abstract methods, as well as the methods defined by the `GLEventListener` interface shown in Listing 10.23. An introduction to programming with OpenGL in Java can be found at the JOGL site[26]. It is important to notice that the actual rendering does not happen in any of the abstract methods in `AbstractGameRenderer` but rather in the `display` method implemented through the `GLEventListener` interface.

Listing 10.22: `AbstractGameRenderer` class definition

```
public abstract class AbstractGameRenderer extends
    GLCanvas implements GLEventListener {
    public AbstractGameRenderer(int width, int height);
    public abstract void showWelcomeScreen(GameStatus
        gameStatus);
    public abstract void showQuestion(QuestionInfo info);
    public abstract void showStatistics(
        List<StatisticsEntry> stats);
    public abstract void showOverallStatistics(
        List<StatisticsEntry> stats);
    public abstract void cleanup();
}
```

Listing 10.23: `GLEventListener` interface

```
public interface GLEventListener {
    public void init(GLAutoDrawable glad);
    public void display(GLAutoDrawable glad);
    public void reshape(GLAutoDrawable glad, int i,
```



```
    int i1, int i2, int i3);  
    public void displayChanged(GLAutoDrawable glad,  
        boolean bln, boolean bln1);  
}
```

The GLEventListener methods

The methods in the GLEventListener interface is not a special set of methods for this application, but rather methods used by all applications who use the JOGL library. We will go through a short usage scenario of these methods in a game mode bellow.

The first method that is called is the `init` method. In this method initialization of any OpenGL buffers and other adjustments should be done. This is the same initialization that is done in most applications utilizing OpenGL. It is also typical to create and start an FPSAnimator. The FPSAnimator makes sure that `display` is called at a certain rate of frames per second. For more information on the inner workings of the FPSAnimator and how to initialize OpenGL correctly see the JOGL tutorial page at their homepage[26].

The `display` method is the method that does all the actual drawing on the screen. It is usually called repeatedly many times per second. It is a smart idea to keep a tracker variable inside the new game renderer class to make sure this method always knows what to draw, should it be the welcome screen, a question or statistics.

When a user resizes the the teacher client window or any other GUI element that affects the OpenGL area, the event system will send a call to the `reshape` method. The usual response to this would be to call the `glViewport` method of the GL object with the new size to resize the current viewport. Again there is nothing special with this method compared to other applications who use OpenGL.

The last method in the GLEventListener interface is the `displayChanged` method. This method is called when the display mode or display device changes. For more information on how to implement this method see the JOGL tutorial

site[26].

The constructor

The `AbstractGameRenderer` base constructor simply calls the `GLCanvas` constructor and adds it self as a listener for GL events. It also sets it's own size to what the parameters specify. Any class that derives from `AbstractGameRenderer` should make sure to call this constructor.

The `showWelcomeScreen` method

The renderer class listen to multiple events that come from the game mode class. The `showWelcomeScreen` method is one of those. It signals that the game mode wants the renderer to display a welcome screen and the argument **`gameStatus`** tells information about the current state of the game. For more information about the `GameStatus` class see Appendix C.2.8. There should be no actual rendering done in this method, but it should signal the class that the next rendered frame should be the welcome screen with the information passed in the argument.

The `showQuestion` method

The `showQuestion` method is called when the game mode wants to display a question on the screen. The parameter **`info`** contains all the information needed to display the question. Detailed information on the `QuestionInfo` class can be found in Appendix C.2.2. As with the `showWelcomeScreen` method no rendering should be done here, but it should signal that the next frame should be the question.

The `showStatistics` method

After a question has reached the time limit, the game mode class should retrieve the statistics from the server and send it to the renderer with a call to the `showStatistics` method. The statistics passed to this function is a list of key and value string pairs representing various aspects of the statistics for the previous question. This method is called after each question and should not be confused with the `show-OverallStatistics` method.

The `showOverallStatistics` method

When all the questions have been answered and the statistics have been shown

for the last question, the game mode requests the overall statistics for the entire game from the Lecture Quiz server. These statistics are then sent to the `showOverallStatistics` method which signals the renderer that the next rendered frame should be the overall statistics. The statistics in the parameter **stats** is a list of key and value pairs of strings in the same way the parameters of the `showStatistics` method work.

The cleanup method

The last method to be called to a renderer is the `cleanup` method. This signals that the game is over and the renderer should clean up anything that is needed after the initialization of the class. If no extra cleanup is needed this method can be implemented as a blank method.

10.3.3 Student Web Client

Game mode creation in the student web client work in a similar way as the other two parts, but the important parts are different. As described in Section 9.4, the student web client uses the MVP architecture suggested for use with the Google Web Toolkit. This means that the implementation of a new game mode mainly consists of a Presenter and a least one corresponding View.

The Presenter

The game mode presenter must extend the `AbstractGameModePresenter` seen in Listing 10.24. This Abstract class has a constructor that takes a reference to the **rpcService** used by the GWT application and the **eventBus** of the application. The **eventBus** is not implemented in the student web client by now, but the reference is used through out the application so that it may be easily implemented at a later stage. By now it may be ignored in the game mode presenter.

The `go` method in the abstract class is used to start the presenter. It takes one

argument; the **container** the corresponding view should be occupying. From this point the developer is free to do what he or she wants regarding implementation, but it is advisable to use at least one separate view to be used for display purposes. In both PlainGamePresenter and LogInPresenter we have specified an interface for the corresponding views to implement. By doing it this way, it is easy to support different views for different devices without changing the logic of the presenter.

Listing 10.24: AbstractGameModePresenter

```
public AbstractGameModePresenter(LQServiceAsync
    rpcService, HandlerManager
eventBus) {
    this.rpcService = rpcService;
    this.eventBus = eventBus;
}
public abstract void go(HasWidgets container);
```

Calls to methods of the `rpcService` are done as regular method calls, but takes one additional argument because AJAX requires asynchronous method calls. An implementation of `AsyncCallback` is needed, as the method is non blocking and returns once it is called. This class is instantiated once the service method have finished. The `onSuccess` method is called if the service method succeeded, otherwise the `onFailure` method is called.

The View

The View class of a game mode implementation is responsible for all the parts of the game mode's graphical user interface. As the user interface may change for each game mode, we have not designed an own interface or abstract class for this purpose. As the user interface may be much richer in the future, this may be something that could be done, so that e.g. questions are displayed in a similar way in different game modes.

As the view does not have to implement any interfaces or extend any classes the

developer of a new game mode may do the implementation the way he or she wants.

The Factory

The game mode factory functionality of the student web client works in the same way in the the other two parts. The GameManager is exactly the same and looks for an implementation of the IGameModeFactory matching the id or name that is asked for. Each game mode should make a new implementation of IGameModeFactory, as seen in Listing 10.25, and add it to the GameManager. The task of the factory is to return a new instance of the corresponding GameModePresenter. For further details on how to use the GameManager, see Section 10.3.1.

Listing 10.25: IGameModeFactory

```
public String getGameModeID();
public String getGameModeName();

public AbstractGameModePresenter createInstance(
    LQServiceAsync rpcService,
    HandlerManager eventBus);
```

The Service

The LQServiceImpl class works as a proxy for the Lecture Quiz Service. As most of the web browsers prevent AJAX calls to different domains, for security purposes, and we are using GWT, it is not possible to call the Lecture Quiz Service directly from the Presenter.

Implementation of the parts of the service used by PlainGame is provided in this class, but additional functionality may be needed in the future. To implement new functionality in LQServiceImpl you may create a new method like a standard java method. To access the Lecture Quiz Service use the code snippet shown in

Listing 10.26. After the implementation is finished, the interfaces, LQService and LQServiceAsync, needs to be updated correspondingly.

Listing 10.26: Getting access to the Lecture Quiz Service

```
LectureQuizService port = (LectureQuizService)
((User)getSession().getAttribute("User")).getPort();
```

Part V

Evaluation

Chapter 11

Lecture Experiment

In this chapter we will present an empirical experiment where our applications were tried out in a realistic environment. The method is described in Section 3.2. In the first part of this section, the context of the experiment will be presented. Then the results will be presented and analyzed briefly.

11.1 Experiment Delimitation

The goal of this experiment was to get an overall picture of how the Lecture Quiz service and clients worked in a real life setting, and to compare it to the similar experiment ran on Lecture Quiz 1.0 in 2007 [1]. As this is just a small test, we will only point out and discuss trends we see in the results. Statistical analysis and thorough psychological analysis are out of the scope of this report. For details on the method used in this experiment see Section 3.2

11.2 Experiment Context

This experiment tested the usability and functionality of Lecture Quiz 2.0 by Knut André Tidemann and Erling Andreas Børresen. The test was prepared by the software creators and lead by associate professor Alf Inge Wang. The experiment took place on May 11th 2010 at lecture room G1 at NTNU, Trondheim, Norway. The purpose of this experiment was to collect empirical data about how well our prototype worked in a real life situation, especially regarding usability and functionality.

11.2.1 Participants and Environment

The experiment was conducted in a lecture in the course TDT4240 Software Architecture, and all the participants were students taking this course. 21 students took part of this experiment, where 81% were male and 19% were female. As the test was conducted in a class of computer science students, most of the students look at themselves as experienced computer users, but none of the participants had tried the software before the experiment. The test was lead by the teacher, and he controlled the progress of the game with the teacher client running on a laptop and displayed on a big screen by a projector. The students used their own mobile phone or laptop to participate through a web browser supporting javascript. The Lecture Quiz server was running on a computer located outside of the lecture room.

11.2.2 Success Criteria

In this section we will present a set of hypotheses describing what we think of as important criteria of our experiment. Some of these success criteria are the same as for the experiment of 2007 [1], and some are new. Basic confirmation of these hypotheses will be regarded as success, although statistical confirmation

of these are out of the scope of this report and this experiment.

- **H1** - The system is not conceived as intrusive on the lecture.
- **H2** - The system is easy to getting started with and use
- **H3** - The system works as it should
- **H4** - The system has high usability
- **H5** - The students find the system inspiring and fun

11.3 Experiment Execution

21 of the students in class chose to participate in the experiment. The lecture was a summary lecture in the course TDT4240 Software Architecture. In the first part of the lecture, theory from this semester was summarized and discussed. The students were allowed to ask questions.

The experiment took place in the second part of the lecture, after a short break. The teacher client was started and an URL to the student client was shown on the projector. Each student logged in on the web client using a desired username and the quiz code displayed in the teacher client. None of the students reported any problems with this part, but some asked if they needed to use their NTNU username or they just could choose one by themselves.

The experiment went on very well. Everyone was able to answer the questions and there was a relaxed atmosphere in the room. Some of the answer options made the students laugh a bit. In one of the questions the teacher client was not able to display the statistics and correct answer. This was displayed correctly on the student client. The problem was solved by going directly to the next question, and all the software seemed to handle this well.

All of the 21 students that took part of the experiment did also answer the questionnaire. The questionnaire is found i Appendix A

11.4 Experiment Results

In this section we will present the results of the questionnaire. First we will calculate the SUS score as described in Section 3.2.1. Thereafter we will present the result of the other important questions. Most of these had five options for the user to answer. From *strongly disagree* to *strongly agree*. These options are displayed in the graphs as values from 1 to 5 respectively, and -1 means that this question was not answered by a user.

To calculate our SUS score we had to discard 6 of the 21 returned questionnaires as they had not answered all of the questions included in the SUS part of the questionnaire. That made it 15 valid questionnaires for our SUS calculation.

Our software got a SUS score of 84 out of 100. This is displayed in table 11.1, and shows how we scored on each question. For a description of how SUS is calculated see Section 3.2.1.

The results of the other questions in our questionnaire are displayed as graphs in the Figures 11.1 through 11.8. Graphs shows how many who answered on each option of a question. Where 5 means strongly agree, 1 means strongly disagree and -1 means did not answer.

Question	Average	SUS/Question
I think that I would like to use this system frequently	3.53	3.53
I found the system unnecessarily complex	1.40	3.6
I thought the system was easy to use	4.53	3.53
I think that I would need support of a technical person to be able to use this system	1.13	3.87
I found the various functions in this system were well integrated	3.73	2.73
I thought there was too much inconsistency in this system	1.73	2.73
I would imagine that most people would learn to use this system very quickly	4.73	3.73
I found the system very cumbersome to use	1.73	3.27
I felt very confident using the system	4.33	3.33
I needed to learn a lot of things before I could get going with this system	1.27	3.73
	SUS Score	84.00

Table 11.1: Results of System Usability Scale

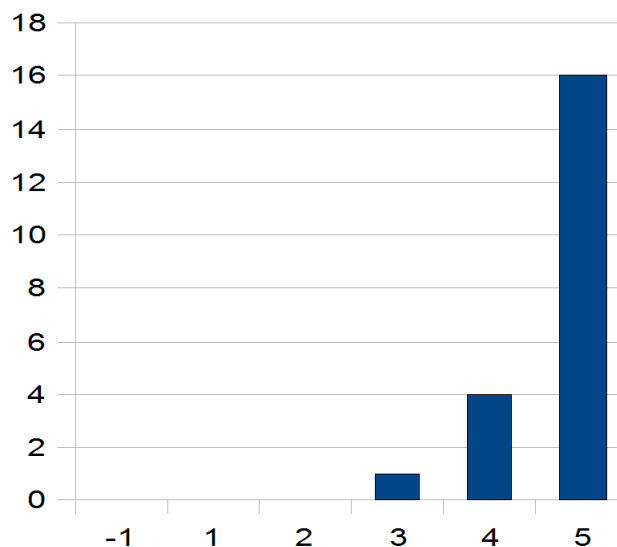


Figure 11.1: **Q1**: I think that I am an experienced computer user

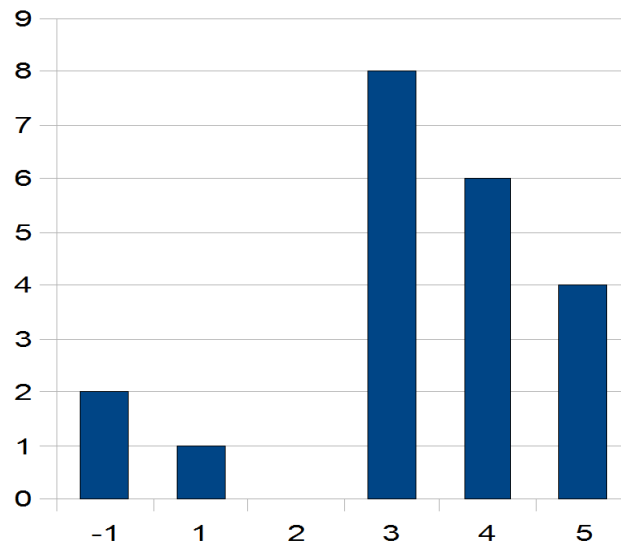


Figure 11.2: **Q2**: I think I paid closer attention during the lecture because of the system

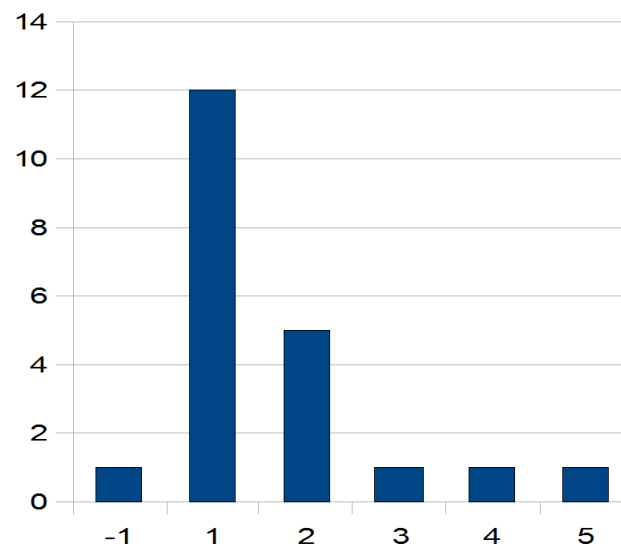


Figure 11.3: **Q3** - I found the system had a distracting effect on the lecture

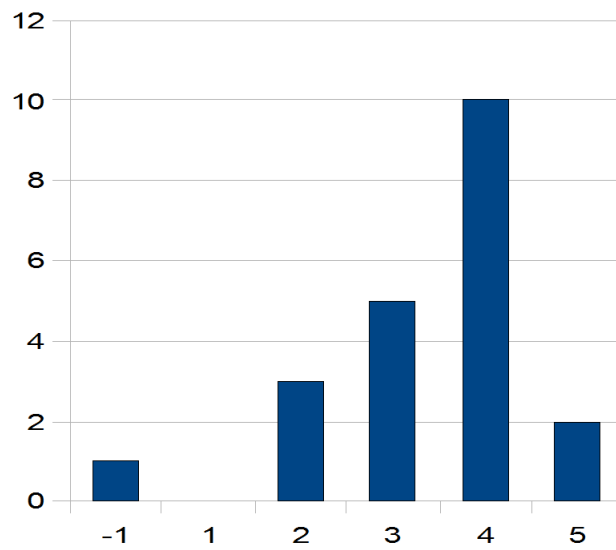


Figure 11.4: **Q4** - I found the system made me learn more

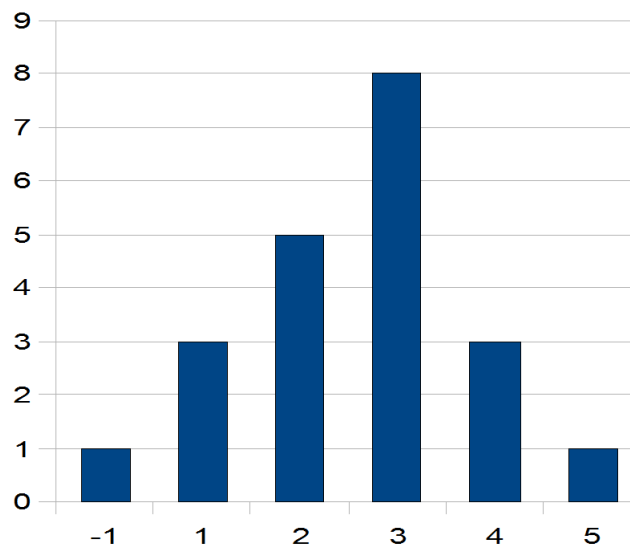


Figure 11.5: **Q5** - I think I learn more during a traditional lecture

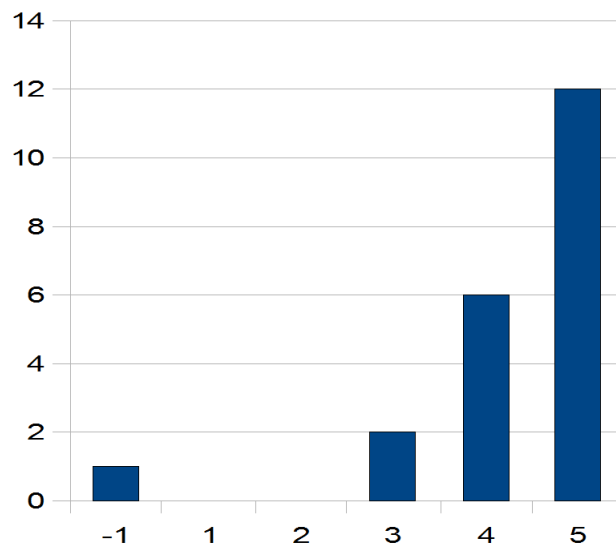


Figure 11.6: **Q6** - I found the system made the lecture more fun

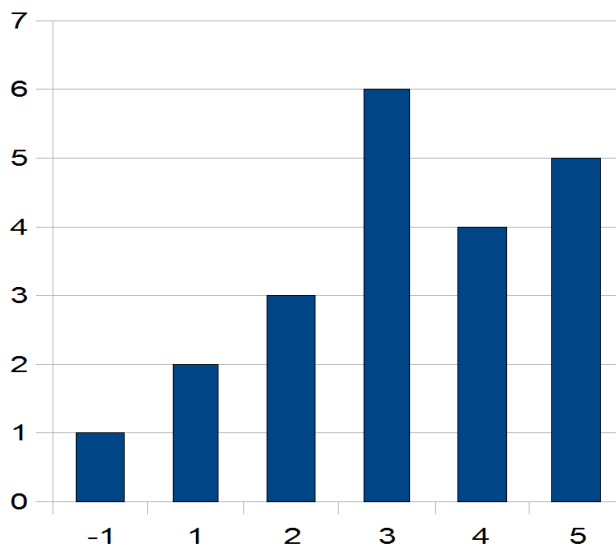


Figure 11.7: **Q7** - I think regular use of the system will make me attend more lectures

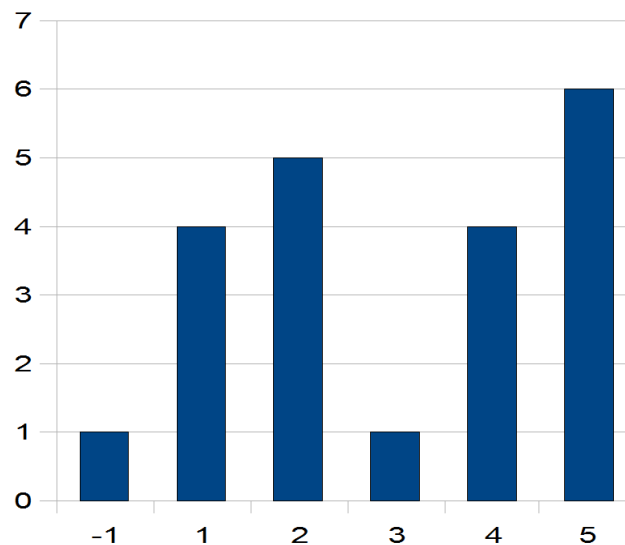


Figure 11.8: **Q8** - I feel reluctant to pay 0.5 NOK in data transmission fee per lecture to participate in using the system

11.5 Experiment Evaluation

In this section we look at the trends and opinions in our survey. These are evaluated along with the feedback we got on the comment part of the questionnaire. The evaluation is presented as answers to the success criteria listed in Section 11.2.2.

- **H1** - The system is not conceived as intrusive on the lecture.

As we can see in Figure 11.3, over 80% disagreed in some way that the system had a distracting effect on the lecture, where 57% strongly disagreed. While only 2 persons agreed to this statement. This is a slightly better result than in 2007, where 70% disagreed to this statement in some way. Having the quiz at the end of the lecture, and not having to change lecture room, as in 2007, may be factors changing this result.

In 2007 the majority thought that regular use of the system would make them attend more lectures. In this experiment the distribution of answers was more even, as seen in 11.7. This proves that more research is necessary before we can make a valid result on this question.

Most of the students thought they paid closer attention during the lecture because of the system. While only one person disagreed to this statement. Although 8 did not have a clear opinion on this and 2 persons did not answer, we find this as a positive result, as this was more evenly distributed in 2007.

- **H2** - The system is easy to getting started with and use

Although these questions are a part of the SUS questions, and therefore describe the general usability of our system, we think they are so important for our prototype that they are included as a separate success criteria. One of the main focuses of our software is to let the students run the client in a web browser and it should therefore be fairly easy to start and use.

There are several questions in SUS addressing these questions.

- I thought the system was easy to use
- I think that I would need support of a technical person to be able to use this system
- I would imagine that most people would learn to use this system very quickly
- I needed to learn a lot of things before I could get going with this system

With an average score on these questions of 4.53, 1.13, 4.73 and 1.27, respectively, we find the results relatively clear. The people that took our questionnaire find our system both easy to use and easy to getting started with. All of these results are somewhat better compared to the experiment in 2007.

Although this is a great result, some of the students commented that the graphical design of the software was not good. Many students complained that the answer buttons were too small, although this could be solved using the zoom function in their web browser. We are fully aware that we are not graphical designers, and that major improvements could be done on this part, but our main focus in this assignment was to get the technical things on the back end done right.

There were also some complaints about the color chosen as a background on option two on the teacher client. This color was displayed differently on the big screen than on a standard computer screen and this made the text almost unreadable. In the experiment the teacher read out all the options, so that all the students did get the information they needed. In the submitted version of the teacher client the background color for this option was made darker to improve readability.

From the teacher's perspective, this was an important part of the experiment. As the time needed to start a quiz was so little, you only needed to put up an URL on the big screen and let the students log in, he thought it was much easier to use the system in a lecture. There was less time wasted compared to the experiment of 2007, and he felt that the system was not breaking the lecture apart in the same way.

- **H3** - The system worked as it should

Out of the 21 returned questionnaires, 18 reported that the software worked as it should on their device. 1 meant that the software did not work because of the problem with small buttons described earlier, 1 did not answer, and 1 did complain that the software did not work in Opera Mini. The reason for the problem in Opera Mini is that our web software is built on AJAX and therefore needs javascript support in the browser. In Opera Mini the requests are compressed and handled on a central server before being sent to the mobile device, and thus javascript does not work. The class was informed of this before the experiment started, and the person reporting problems switched browser to use our software without any further problems.

During the experiment the teacher client failed to show the statistics for one of the questions. The statistics were displayed correctly on all the student clients, and all answers were stored as normal. The quiz continued as usual when the teacher pressed the button to start the next question. We think this is only a minor bug in the teacher client and that the rest of the system works as expected. We were not able to reproduce this bug.

- **H4** - The system has high usability

With a SUS score of 84 we think that our system has high usability. The SUS score of the experiment in 2007 were 74.25. Our system does mainly the same things, except for our student client being web based. Thus we may see the web based approach as a success. Although some of the technical problems of 2007 may have had an effect on that score.

- **H5** - The students found the system inspiring and fun

Figure 11.6 shows this quite well. 18 out of 21 think that the system made the lecture more fun. 1 did not answer and 2 persons did not have a clear opinion on this statement. This is close to the result in 2007, and we see a clear trend that students think using the lecture quiz system in lectures make them more fun.

11.6 Experiment Conclusion

During this experiment we had less technical problems than the comparable experiment in 2007, thus probably making the users more friendly in their evaluation of the system. The results of this experiment are mostly positive and in some extent better than last time. The results of this experiment indicate the following trends:

- Most students do not find the system intrusive on the lecture

- The students found the system made the lecture more fun
- The system was easy to getting started with and use
- The system has high usability
- Our system worked mostly as expected

Chapter 12

Evaluation

In this chapter we are going to discuss and analyze our research and development method and how it was executed according to the given guidelines. We will also look at the requirements described in Chapter 6 and discuss how well these are met in our system. The experiment is already analyzed and evaluated in Chapter 11.

12.1 Research Method

As described in Chapter 3, we decided to go for the engineering approach in our research. In the beginning we started to define a set of research questions. This was followed by searching for previous work, both regarding game theory and previous educational software, which gave us a good historical basis for our further work.

Based on the findings in the previous work we came up with a set of requirements for the our system, and based on these requirements a software architecture was designed. Then there was selected appropriate technologies, followed by implementation of the system.

The chosen solutions and design choices in the architecture and the choice of technologies are mainly based on our subjective assessments and the findings in the previous work. The engineering approach has made it possible to experiment with the combination of our own ideas and things that have been done earlier.

At the end of our project we arranged an empirical experiment to test the usability of the system. This has shown some trends and indications of how well the system was received. Ideally there should be held a larger and more statistically valid experiment. Such an experiment would probably uncover room for more improvements and give the system more integrity.

12.2 Development Method

In this project we chose to go for the Scrum process with short iterations. This has worked out well. There have been some days where the scrum meetings were missing, but the communication between the two of us has been good all along.

We used the requirements from Section 6 as the product backlog. Taking parts of the requirements and implementing them in pieces has been successful. It has been good to see different functions evolve and to see the whole product as a sum of smaller parts.

At the end of each sprint we have not, as described in scrum, had an evaluation and testing with external stake holders, but we have done this internally. As this is not a project driven by a commercial customer, and there were no need to deploy each sprint in production, this was an acceptable solution for our project.

12.3 Requirements

When we designed the architecture we created a set of requirements for our Lecture Quiz system. In this section we will go through how we have completed these requirements.

FR1 The game shall consist of a teacher client and a number of student clients

Our architecture allows both a teacher client and multiple student clients to be connected at the same time in the same game.

FR2 The teachers need to authenticate to use the client

Authentication with a username and password through a call to the web service procedure `authenticate` is required before a user can use any procedure that require teacher privileges.

FR3 It must be possible to extend the game with new game types

New game types can be created by extending the already existing game modes in both the server and clients.

FR4 It must be possible for the teachers to store questions for later use

A teacher or administrator can save questions in the Lecture Quiz system through the quiz editor and these questions can then be used by new quizzes later.

FR5 It must be possible to tag questions for easier reuse and grouping

While the support for question-tagging is implemented in the database, it is not implemented in the Lecture Quiz server or the teacher client quiz editor.

FR6 A question shall consist of four options

A question in the implemented game mode does consist of four options, but this is not a hard limit. Any new game mode can allow fewer or more answer options.

FR7 Questions must be able to have an individual time limit

The questions have the ability to have an individual time limit which is set in the quiz editor.

FR8 It must be possible to run several quizzes at the same time

The system is capable of running both several different quizzes at the same time as well as multiple instances of the same quiz in different lectures.

FR9 Statistics should be shown after a question has been answered and after the quiz has been completed

The statistics are shown after each question and after the quiz is completed. The running game mode determines what and how these statistics are displayed.

FR10 A quiz game may group the students into groups

The possibility for a game mode to group students into teams is available but the implemented Plain Game mode does not support this feature. A new game mode can implement this without restrictions.

FR11 The teacher decides when to start a quiz

To start a quiz the teacher has to actively start each question after the previous one has completed. However, it is possible to create a game mode that overrides this behavior.

FR12 The students must identify themselves with a user name when joining a quiz

Before a student can join a quiz it needs to identify him or herself with a call to the `authenticate` procedure. If the user name is already taken, authentication will fail and the user is asked to supply another one.

FR13 The students must answer questions before the time limit is up

If a student tries to answer the question too late, the server returns an error. In the student client the question is automatically skipped if the timer runs out and the student loses his or her ability to submit an answer to the question.

FR14 The students must supply a quiz code to join a quiz

To join a quiz a student has to supply the quiz code given out by the teacher. If a wrong quiz code is inserted an error is returned.

FR15 The teacher must be able to pause between questions

The game automatically pauses when the statistics are shown and the teacher needs to start the next question manually. This behavior can be overridden by a game mode if required.

FR16 The teacher must be able to save the statistics from a quiz round that has just ended

The teacher cannot save the statistics in the teacher client at the moment, but the ability to implement this feature is already there by saving the statistics data returned from the server.

12.3.1 Quality Requirements

In this section we will go through all the quality requirements and discuss if they are fulfilled by our system. As there was little time for thorough testing, some of the requirements are hard to validate. This is left for future tests and experiments.

M1 - Deploying a new game mode for a client

This is not tested as there currently only exists one game mode for the system. But the system is designed and implemented in a way that this should most definitely be a requirement that is possible to reach.

M2 - Creating a new client

This is very hard to test without making a completely new client. This requirement depends in many ways on how rich the new client should be. We have made a clear and well documented service, and a small client could be possible to complete within two days

M3 - Adding support for a new database back end

Again, this is not tested, as we have not implemented support for more than the MySQL database back end. The system is designed in a way that should make it easy to implement support for new back ends. We think that if the person implementing the support are familiar with the new database system, this should absolutely be a requirement that could be reached.

U1 - Changing server settings

This requirement has been fulfilled by our systems use of a configuration file. The use of the configuration file is described in Section 10.1.1.

U2 - Getting started

This is not tested by use of a stopwatch, but based on the feedback we got from the experiment, this requirement is perfectly fulfilled.

U3 - The game should be fun

As 18 out of the 21 answered that they in some extent found the system made the lecture more fun, thus we consider this requirement as fulfilled.

U4 - Deploying the Lecture Quiz Server

By testing this requirement by our selves, we used around 10 minutes. Taking into account that we are familiar with the system, this makes it plausible that a system administrator should complete this task in less than one hour.

U5 - Adding question

By simple testing done by us as users that are familiar with the system, fulfilling this requirement is not a problem. But there should be done more testing to validate this with inexperienced users.

S1 - Multiple game servers

As the system has really low performance needs in our small test of 21 users, this requirement is hard to validate. Further experiments and performance testing should be done to achieve this.

S2 - Number of users

In our test with 21 users our home computer had no problem serving the system. This makes it likely that 100 users should not cause any problems on more suited server hardware. Though this should be validated by more extensive testing.

Part VI

Conclusion

Chapter 13

Conclusion

In our thesis we have tried to make a scalable architecture for use in the education game Lecture Quiz. By looking at the previous work done by Mørch-Storstein and Øfsdahl[1], we have identified requirements and research questions which we have solved to create a better and more modifiable architecture. The results of the research questions are:

RQ1 What architecture is best suited for the Lecture Quiz game?

We have created an architecture based on three separate components and we found that using a web service as our server eased the development and the modifiability of the architecture. By using a student web client we could reach more students in an easier way.

The main features of our architecture is extendable game modes, the ability to run multiple game servers on the same database and run many different quizzes on the the same server. Answers from students are delivered via a web application while the teacher client runs as a native Java application.

a) How should data be exchanged between the clients and the game server?

For communications the SOAP protocol fits the needs of the system and is also easy to implement and use. It gives the ability to develop clients on many different platforms without issues and the integration of this protocol is very good in many libraries and platforms. This protocol has many advantages to the other alternatives, including automatic session handling on the server. By using a remote procedure protocol we did not have to debug any of the actual protocol communication our selves.

b) How does this architecture scale when the number of users increases?

The architecture is made scalable by allowing multiple game servers to use the same database. This means that the quizzes and questions are all stored at the same place and should the need for more computing power be there, the requirements for installing additional Lecture Quiz servers are low. The only component required is an installed Java web container. This way the load can be distributed between multiple servers in an environment where there are many lectures running quizzes at the same time.

c) How do we design the architecture flexible in terms of game modes?

The game modes are designed with a base class that implements the core functionality required for a game mode. These classes can then be extended to allow new rules in the game play and should the need be there, all the already implemented functionality can be overridden by new code. This allows both for easy implementation of new game modes, as well as the power to implement complex new ways to run games. The web service procedures are very generic in their way and by supporting custom parameters on some of them, a new game mode can implement things we did not think of at the time of design.

There is also a handshake between clients and the server to make sure only game modes supported on the server are started on both ends. This ensures compatibility with clients that do not support certain game modes.

RQ2 What technologies are best fit for the chosen architecture?

From both a usability and development perspective, we have concluded that a web based technology is the most suited and we have therefore selected the Java platform for our server and SOAP as the communication protocol. The choice of Java was partially made because of its multiplatform properties and the usage of it on NTNU.

We could not conclude a definite best database system, so we went with MySQL which both NTNU and we have experience with. In light of this, we made an interface for all database calls which makes it easy to switch the MySQL back end to another database system.

With the overall desire of supporting multiple platforms and devices we think the Java platform with the use of OpenGL was the best technology to use when creating the teacher client.

RQ3 Will students consider the software easier to use if it is web based?

To find out if the students preferred the web based student client we performed a survey and compared the results to the previous study done with the Lecture Quiz 1 prototype by Mørch-Storstein and Øfsdahl[1]. Our results showed a higher SUS score and students had little problems getting the application up and running. This shows that the students found it easier to use and participate than they did with the previous Lecture Quiz prototype, which required application installation on the used devices.

RQ4 Do students enjoy playing educational games during lectures?

We can conclude from the results of our experiment in Chapter 11 that the students thought the use of the Lecture Quiz system made the lecture more fun. We therefore think the students enjoy playing educational games during lectures.

We consider the task of developing a scalable and modifiable architecture for the Lecture Quiz system to be very successful. Our employer is also very pleased with the results after reviewing the design and having participated in the lecture experiment.

Chapter 14

Further Work

Our project has focused on building a strong and easily modifiable architecture for the Lecture Quiz game and we have had less focus on the visual aspects. We would also have liked to perform larger usability tests on a larger audience to get as much feedback and testing of the system as possible. We have outlined some of the changes we think would improve the Lecture Quiz system.

14.1 Improve Clients

When looking at our teacher client it is pretty obvious that we did not prioritize graphical design. We did what we could to in the timeframe we had, but this area of the Lecture Quiz system is one that needs the most improvements for the game to be more spectacular. By improving the design we can make the game more fun and interesting for the students who play it. We have some graphical data included in the attachments that were made for us after the code freeze, so we did not have enough time to implement these. The files can be found under the *art/* directory. These graphic files include a logo and a splash screen as well as some icons that can be used.

We also had a requirement to be able to save statistics from played games, but we did not manage to complete this in time. This should be an easy addition by dumping the statistics received from the server in an appropriate format.

The student client can also be improved in numerous ways. There are some improvements in usability and error handling that can be implemented to increase the robustness of the application as well as the user experience for the students. One of these issues would be to handle the manual refreshing of the client and make sure the current game is not lost in this case.

14.2 Implement Tags and Searching

When we designed the architecture we wanted the questions and quizzes to be tagged with keywords that made searching for them easier and faster. This was implemented in the database but never made it into the other Lecture Quiz components. Implementing this aspect should not be hard, but it might require the addition of a new web service procedure that the teacher client can use to input search keywords and only get matching quizzes in return. The search functionality is not implemented in the teacher client either and we would want to have the possibility to search and filter for quizzes when starting a quiz.

14.3 Improve Quiz Editor

The quiz editor implemented in the teacher client works, but is pretty basic. It has the possibility to edit what questions that should be a part of a quiz and the ability to create new quizzes and questions. What it does not have is the ability to edit already existing questions. The user interface could also be improved with the addition of tags which will let users filter the quiz list. Another feature that would suit this editor is to give a warning when a user classed as a teacher tries to

open and edit a quiz he or she is not an owner of. Currently this will only give an error when the quiz is saved. The implementation of this feature could be done at the web service side by implementing a check in the `getQuiz` procedure.

14.4 Security

While we have taken notice of many intrusion possibilities, we have not focused on security. A person with enough interest would probably be able to exploit the applications in several ways we have not yet identified. The user roles are pretty much secure at this point, but tampering with data supplied to various web service procedures could lead to unexpected behavior in the web service. We have not looked at possible denial of service attacks either.

14.5 Additional Game Modes

The solution delivered by us only have one available game mode. While this is sufficient for basic usage, adding more game modes can make the game more fun and challenging for the students. By having multiple game modes available to them, teachers can pick and choose depending on the lecture and student attendance.

We have thought up a couple of different game modes that would be interesting to implement. One is a game mode were students are split up in different teams and points are given to teams depending on how many correct answers they had. The team distribution could either be random or selected on join.

Another interesting game mode would be an elimination game where students are allowed to answer until they get a question wrong and are then eliminated from the game. This goes on until there are just a single student left and that student will be declared the winner. This could require either a long quiz or the ability to

select secondary quizzes that will be run if the first one is out of questions.

14.6 Run Larger Empirical Tests

While we have done testing on a small group of 21 people in a single lecture and performance was not an issue, we were not able to do a large scale test with a filled auditorium. It would be beneficial to test the architecture on a larger group of people to make sure it holds performance up to par. Based on the results of these tests the system can be optimized to increase performance.

Part VII

Appendices

Appendix A

Questionnaire

Questionnaire Lecture Quiz – Software Architecture May 10th 2010

Gender Male Female

What brand is your mobile/computer?

What operating system did you use?

What browser did you use during this test?

What connection type did you use? WLAN Cable 3G GPRS/EDGE Other

	Strongly disagree				Strongly agree
I think that I am an experienced computer user	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I think that I would like to use this system frequently	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I thought the system was easy to use	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I think that I would need the support of a technical person to be able to use this system	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I found the various functions in this system well integrated	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I thought there was too much inconsistency in the system	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I would imagine that most people would learn to use this system very quickly	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I felt very confident using the system	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I needed to learn a lot of things before I could get going with this system	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I think I paid closer attention during the lecture because of the system	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I found the system had a distracting effect on the lecture	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I found the system made me learn more	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I think I learn more during a traditional lecture	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
I found the system made the lecture more fun	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I think regular use of the system will make me attend more lectures

I feel reluctant to pay 0.50 NOK in data transmission fee per lecture to participate in using the system

I found the system unnecessarily complex

I found the system very cumbersome to use

Did the client software work properly on your phone/computer? Yes No

If no; please describe the problem

Are there other things you would like to comment?

Appendix B

Database SQL file

```
-- SQL Dump
```

```
SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";
```

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=  
    @@CHARACTER_SET_CLIENT */;
```

```
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=  
    @@CHARACTER_SET_RESULTS */;
```

```
/*!40101 SET @OLD_COLLATION_CONNECTION=  
    @@COLLATION_CONNECTION */;
```

```
/*!40101 SET NAMES utf8 */;
```

```
--
```

```
-- Database: `lecturequiz`
```

```
--
```

```
CREATE DATABASE `lecturequiz` DEFAULT CHARACTER SET utf8  
    COLLATE utf8_general_ci;
```

```
USE `lecturequiz`;
```

```
-----
```

```

--
-- Table structure for table `answer`
--

CREATE TABLE IF NOT EXISTS `answer` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `text` text NOT NULL,
  `question` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `question` (`question`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=90 ;

-----
--
-- Table structure for table `question`
--

CREATE TABLE IF NOT EXISTS `question` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `text` text NOT NULL,
  `timeout` int(11) NOT NULL,
  `correctAnswer` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=24 ;

-----
--
-- Table structure for table `quiz`
--

CREATE TABLE IF NOT EXISTS `quiz` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(64) NOT NULL,
  `description` text NOT NULL,
  `owner` int(11) NOT NULL,

```

```
PRIMARY KEY (`id`),
KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=4 ;
```

```
-----
--
-- Table structure for table `ref_quiz_question`
--
```

```
CREATE TABLE IF NOT EXISTS `ref_quiz_question` (
  `quiz` int(11) NOT NULL,
  `question` int(11) NOT NULL,
  KEY `quiz` (`quiz`,`question`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

```
-----
--
-- Table structure for table `ref_tagged`
--
```

```
CREATE TABLE IF NOT EXISTS `ref_tagged` (
  `tag_id` int(11) NOT NULL,
  `question_id` int(11) NOT NULL,
  `quiz_id` int(11) NOT NULL,
  KEY `question_tag` (`tag_id`,`question_id`),
  KEY `quiz_id` (`tag_id`,`quiz_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
-----
--
-- Table structure for table `tag`
--
```

```
CREATE TABLE IF NOT EXISTS `tag` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
```

```
    `name` varchar(64) NOT NULL,  
    PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;
```

```
-----  
--  
-- Table structure for table `user`  
--
```

```
CREATE TABLE IF NOT EXISTS `user` (  
    `id` int(11) NOT NULL AUTO_INCREMENT,  
    `username` varchar(32) NOT NULL,  
    `password` varchar(40) NOT NULL,  
    `role` tinyint(4) NOT NULL,  
    PRIMARY KEY (`id`),  
    KEY `user_pass` (`username`,`password`)  
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

```
-----  
--  
-- Create administrator user  
--
```

```
INSERT INTO `user` (  
    `username`,  
    `password`,  
    `role`  
) VALUES (  
    'Administrator',  
    SHA1('admin'),  
    1  
) ;
```


Appendix C

The Lecture Quiz Service API

C.1 Web service Procedures

The examples in the sections bellow call web service procedures from an object known as the *port*. This port is the link between the client and the web service and its class and support classes are generated by the development environment. When coding in the teacher client you can get this port by calling up the `ConnectionManager` class and get an instance from the `getPort` method. A detailed example of this can be seen in Section 10.3.2. The student web client stores the port objects for each connection in the session of each user. An example on how to retrieve this object can be seen in Section 10.3.3.

Many of the web service procedures have parameters that is of the `Holder`¹ type. This `Holder` class makes it possible to return data as a parameter to a web service procedure. We return data this way so that we can also return error codes at the same time.

These procedures are listed in alphabetical order.

¹`javax.xml.ws.Holder`

C.1.1 authenticate

The `authenticate` procedure works in two different ways depending on whether the **auth** parameter is true or false. If this parameter is true, the client is considered a teacher client and the server will try to authenticate the user with the password supplied in the **password** parameter. This is done by the `UserManager` class and if authentication is successful, a session is created with the user object returned from the `UserManager` class. During authentication the role of the user will be set to either teacher or administrator, depending on the role set in the database.

If the **auth** parameter is false, the calling client tries to log in as a student client. In this case the **password** parameter is not used and can be null. The server only checks if another student has already picked the same user name. If the username is available the authentication succeeds, if not the error code 2 is returned.

Procedure definition:

Listing C.1: Getting access to the Lecture Quiz Service

```
public int authenticate(String username, String password,  
                        boolean auth)
```

Parameters:

username The username of the user trying to authenticate.

password The password of the user trying to authenticate.

auth Tells the server whether it should try to authenticate the client against the database or not. See details below.

Return codes:

0 The authentication was successful, and the session is now valid.

- 1 Authentication failed due to incorrect password in teacher client mode.
- 2 Username already in use.
- 3 Invalid parameters. Returned if any of the required objects are null or similar.

Examples

Authentication as a teacher client

Listing C.2: Getting access to the Lecture Quiz Service

```
port.authenticate ("ExampleTeacher", "ExamplePassword",  
    true);
```

Authentication as a student client

Listing C.3: Getting access to the Lecture Quiz Service

```
port.authenticate ("ExampleStudent", null, false);
```

C.1.2 endQuiz

The `endQuiz` procedure tells the server to shut down a running quiz created by the calling client. This procedure can only be called from an authenticated teacher or administrator. This will free up the quiz code used for the quiz as well as unlink all the student from the quiz.

Procedure definition:

Listing C.4: Getting access to the Lecture Quiz Service

```
public int endQuiz()
```

Parameters:

None

Return codes:

- 0 Success. The currently running quiz was terminated.
- 1 Could not find a valid session.
- 2 Not logged in.
- 3 Access denied. The user does not have permissions to call this procedure.
- 4 No active quiz. The calling user does not currently have a quiz running

Example:

Listing C.5: Getting access to the Lecture Quiz Service

```
port.endQuiz();
```

C.1.3 getAvailableGameModes

To get a list of supported game modes on the server anyone can call `getAvailableGameModes`. No authentication is required before this method is called. This procedure has no additional error codes, so if no supported game modes are found, the list is simply empty. All available game modes are stored in the `GameManager` class and retrieved from there.

Procedure definition:

Listing C.6: Getting access to the Lecture Quiz Service

```
public int getAvailableGameModes (Holder<List<GameModeInfo>>  
    gameModes)
```

Parameters:

gameModes A holder object carrying a list of `GameModeInfo` classes. This is an output parameter, all the game modes the server support will be stored in this list.

Return codes:

0 Success.

Example:

Listing C.7: Getting access to the Lecture Quiz Service

```
Holder<List<GameModeInfo>> gameModesHolder =  
    new Holder<List<GameModeInfo>> ();  
port.getAvailableGameModes (gameModesHolder);  
List<GameModeInfo>> gameModes = gameModesHolder.value;
```

C.1.4 getAvailableQuizzes

The `getAvailableQuizzes` procedure returns a list of all stored quizzes on the server. Only teachers and administrators are able to call this procedure after authorization has been done. The request is forwarded to the currently running database back end and is processed there.

Procedure definition:

Listing C.8: Getting access to the Lecture Quiz Service

```
public int getAvailableQuizzes (Holder<List<QuizInfo>>  
    quizzes)
```

Parameters:

quizzes A holder object carrying a list of `QuizInfo` classes. This is an output parameter. All quizzes stored on the server will be listed in this list.

Return codes:

- 0 Success.
- 1 Could not find a valid session.
- 2 Not logged in.
- 3 Access denied.
- 4 Error occurred on the server.

Example:

Listing C.9: Getting access to the Lecture Quiz Service

```
Holder<List<QuizInfo>> quizzesHolder =  
    new Holder<List<QuizInfo>> ();  
port.getAvailableQuizzes(quizzesHolder);  
List<QuizInfo> quizzes = quizzesHolder.value;
```

C.1.5 getCurrentGameStatus

A teacher or administrator can call `getCurrentGameStatus` to get an updated status of the game they started. This information contains the number players joined, quiz code, question count and current question number. For more information see the `GameStatus` class documentation.

Procedure definition:

Listing C.10: Getting access to the Lecture Quiz Service

```
public int getCurrentGameStatus(Holder<GameStatus>  
    gameStatus)
```

Parameters:

gameStatus A holder object carrying a `GameStatus` object holding the information about the game status.

Return codes:

- 0 Success.
- 1 Could not find a valid session.
- 2 Not logged in.
- 3 Access denied.
- 4 No active quiz.

Example:

Listing C.11: Getting access to the Lecture Quiz Service

```
Holder<GameStatus> gameStatusHolder =
    new Holder<GameStatus> ();
port.getCurrentGameStatus(gameStatusHolder);
GameStatus gameStatus = gameStatusHolder.value;
```

C.1.6 `getCurrentQuestion`

The `getCurrentQuestion` procedure retrieves the current question of the game the calling client has joined. This procedure can only be called after a game has been joined with `joinQuiz` or started by a teacher or administrator with `newQuiz` and the first question has been started with `startNextQuestion`. If the first question has not been started the procedure returns 4 telling the user that the question is not ready yet. The calling user's current game is looked up in the session and returns a valid `QuestionInfo` object which is then sent back to the client. This object does not contain the ID of the correct answer.

The procedure call will also fail if the last question of the quiz has been answered and will then return 5. This indicates that the client can now call `getOverallStatistics` as the quiz is over and there are no more questions. A client can call this procedure multiple times for each question if necessary and the time left

value found in `QuestionInfo` will always be updated. See the `QuestionInfo` class documentation for more information.

Procedure definition:

Listing C.12: Getting access to the Lecture Quiz Service

```
public int getCurrentQuestion(Holder<QuestionInfo> info)
```

Parameters:

info A holder object with a `QuestionInfo` object that has all the information needed to display a question and its possible answers.

Return codes:

- 0** - Success.
- 1** - Could not find a valid session.
- 2** - Not logged in.
- 3** - No active quiz.
- 4** - Cannot get question at this time.
- 5** - No more questions in the quiz.

Example:

Listing C.13: Getting access to the Lecture Quiz Service

```
Holder<QuestionInfo> questionHolder = new Holder<  
    QuestionInfo> ();  
port.getCurrentQuestion(questionHolder);  
QuestionInfo questionInfo = questionHolder.value;
```


C.1.7 `getGameModeInfo`

A student client must call `getGameModeInfo` before joining a game with `joinQuiz` to identify the game mode running on the sever and to make sure the game mode is supported by the student client. This procedure will try to find the game identified by the quiz code specified by **quizCode** and then fill out the `GameModeInfo` for this game. This procedure can be called without being authenticated and can therefore be called by all clients.

If no game is found with the quiz code supplied the procedure will return the error code 1. For more details see the `GameModeInfo` class documentation.

Procedure definition:

Listing C.14: Getting access to the Lecture Quiz Service

```
public int getGameModeInfo(String quizCode, Holder<
    GameModeInfo> info)
```

Parameters:

quizCode A unique identifier of the game to query the game mode info from.

info A holder object containing a `GameModeInfo` object.

Return codes:

0 Success.

1 No such quiz.

Example:

Listing C.15: Getting access to the Lecture Quiz Service

```
Holder<GameModeInfo> gameModeHolder =
```

```
    new Holder<GameModeInfo>();  
port.getGameModeInfo("ExampleQuizCode", gameModeHolder);  
GameModeInfo gameModeInfo = gameModeHolder.value;
```

C.1.8 getOverallStatistics

The `getOverallStatistics` procedure can be called after a quiz has finished and all the questions have been answered. This can be identified by the return value 5 from the `getCurrentQuestion` or the `startNextQuestion` procedures. Should this condition not be met it will return with the error code 4. When this procedure is called it makes the necessary session checks and then forwards the request to the game the client is a part of. The returned data is a list of `StatisticsEntry` objects, which are basically key and value pairs with statistics information.

This procedure is highly game mode specific, as nothing of the data is touched outside the game mode class that is running the calling client's game. The output is also different whether or not the calling client is a teacher or a student. The a call from the teacher would normally return overall statistics from all users, while a student client would receive statistical data from his own answers only. As this data are only key and value pairs of strings, it is up to the calling client to handle this data properly for the running game mode.

When running the game mode implemented by the `PlainGame` class the percentage of correct answers submitted are returned when a student calls this procedure. A teacher client receives four entries. The entry with the key "correct" holds the percentage of all submitted answers who were correct. The "questionCount" simply says how many questions there was in the quiz. The user who has the most correct answers of all participants is added by the key "mostCorrectUser". The value of this entry is the username of this user. Last we have the "mostCorrectCount" key, which tells how many correct answers this user had.

See also the `getQuestionStatistics` documentation.

Procedure definition:

Listing C.16: Getting access to the Lecture Quiz Service

```
public int getOverallStatistics(Holder<List<StatisticsEntry  
>> stats)
```

Parameters:

stats A holder object containing a list of `StatisticsEntry` objects.

Return codes:

- 0** Success.
- 1** Could not find a valid session.
- 2** Not logged in.
- 3** No active quiz.
- 4** Cannot get statistics at this time.

Example:

Listing C.17: Getting access to the Lecture Quiz Service

```
Holder<List<StatisticsEntry>> statsHolder =  
    new Holder<List<StatisticsEntry>>();  
port.getOverallStatistics(statsHolder);  
List<StatisticsEntry>> entries = statsHolder.value;
```

C.1.9 `getQuestion`

The `getQuestion` procedure is used to get all the information about a specific question in from of a `FullQuestionInfo` object. The difference between this class

and the `QuestionInfo` class is that `FullQuestionInfo` also contains the ID of the correct answer. This procedure was designed for use when editing quizzes. Only teachers and administrators can call this procedure so authentication needs to be completed before use. The question is retrieved by the running database back end on the server.

Procedure definition:

Listing C.18: Getting access to the Lecture Quiz Service

```
public int getQuestion(int id, Holder<FullQuestionInfo>
    info)
```

Parameters:

id The ID of the question to retrieve

info A holder object containing a `FullQuestionInfo` object with all the question information.

Return codes:

- 0 Success.
- 1 Could not find a valid session.
- 2 Not logged in.
- 3 Access denied.
- 4 No such question.

Example:

Listing C.19: Getting access to the Lecture Quiz Service

```
Holder<FullQuestionInfo> questionHolder =
    new Holder<FullQuestionInfo> ();
```

```
// Get the question that has the ID 1
port.getQuestion(1, questionHolder);
FullQuestionInfo questionInfo = questionHolder.value;
```

C.1.10 `getQuestionList`

To get a full list of all questions stored in the server database you can call `getQuestionList`. This will return a `FullQuestionInfo` object but it will only contain one answer, the correct one. This is done for performance reasons. To get all the answers for a single question call `getQuestion` instead with the ID of the question. This procedure is only available to teachers and administrators and is meant to be used when editing quizzes to add already existing questions to a quiz. Return codes are straight forward with the addition of 4, which merely indicates that there was something wrong when contacting the database.

Procedure definition:

Listing C.20: Getting access to the Lecture Quiz Service

```
public int getQuestionList (Holder<List<FullQuestionInfo>>
    info)
```

Parameters:

info A holder object containing a list of `FullQuestionInfo` objects.

Return codes:

- 0** Success.
- 1** Could not find a valid session.
- 2** Not logged in.

- 3 Access denied.
- 4 Could not get list from DB.

Example:

Listing C.21: Getting access to the Lecture Quiz Service

```
Holder<List<FullQuestionInfo>> questionListHolder =  
    new Holder<List<StatisticsEntry>>();  
port.getQuestionList(questionListHolder);  
List<FullQuestionInfo> entries = questionListHolder.value;
```

C.1.11 getQuestionStatistics

The `getQuestionStatistics` procedure is can be called after each question has been answered and the time limit has been reached. It is then callable until the teacher starts the next question with a call to `startNextQuestion`. If this procedure is called outside of this state error code 4 will be returned. That usually means the client called the procedure too early and the question time limit has not been reached yet. Upon a successful call, a list of `StatisticsEntry` objects are returned. These are key and value pairs of strings that tell the calling client about the statistics of the previous questions' answers.

As with `getOverallStatistics`, the behavior this procedure is highly dependent of the running game mode and the client type sending the request. As the retrieved data is just a list of strings it is up the the running game mode on the client to display this data to the user in a proper fashion. A teacher client calling this procedure would normally receive statistics for all the answers submitted, while a student will only get feedback from his own answer.

When running the game mode implemented by the `PlainGame` class only the correct answer ID returned when a student calls this procedure with the key "correctAnswer". When a teacher calls it, the response contains the correct answer ID

as well as the number of submitted answers each of the possible answers got from the students. The keys for these entries are the ID of the answer they represent.

See also the `getOverallStatistics` documentation.

Procedure definition:

Listing C.22: Getting access to the Lecture Quiz Service

```
public int getQuestionStatistics(Holder<List<
    StatisticsEntry>> stats)
```

Parameters:

stats A holder object containing a list of `StatisticsEntry` objects

Return codes:

- 0** Success.
- 1** Could not find a valid session.
- 2** Not logged in.
- 3** No active quiz.
- 4** Cannot get statistics at this time.

Example:

Listing C.23: Getting access to the Lecture Quiz Service

```
Holder<List<StatisticsEntry>> statsHolder =
    new Holder<List<StatisticsEntry>>();
port.getQuestionStatistics(statsHolder);
List<StatisticsEntry>> entries = statsHolder.value;
```

C.1.12 `getQuiz`

The `getQuiz` procedure searches the server database for a quiz with the given ID and then returns this in the form of a `FullQuizInfo` object through the `quizInfo` parameter. Inside this object is the name and description of the quiz, all the questions and possible answers as well as the which answers that are correct. For more information see the `FullQuizInfo` documentation in Section C.2.5. The operation itself is handled by the running database back end. This procedure was created for use when editing quizzes and can only be called by teachers and administrators. If the requested quiz is not found error code 4 is returned.

Procedure definition:

Listing C.24: Getting access to the Lecture Quiz Service

```
public int getQuiz(int id, Holder<FullQuizInfo> quizInfo)
```

Parameters:

id The ID of the quiz to retrieve.

quizInfo A holder object containing a `FullQuizInfo` object.

Return codes:

- 0** Success.
- 1** Could not find a valid session.
- 2** Not logged in.
- 3** Access denied.
- 4** No such quiz.

Example:

Listing C.25: Getting access to the Lecture Quiz Service

```
Holder<FullQuizInfo> quizHolder = new Holder<FullQuizInfo>
    >();
// Get the quiz that has the ID 1.
port.getQuiz(1, quizHolder);
FullQuizInfo quiz = quizHolder.value;
```

C.1.13 getServiceVersion

The `getServiceVersion` procedure returns the running version of the Lecture Quiz Web Service the client is connected to. This version is returned in the form a string though the **version** parameter. No authentication is required to call this procedure so it can be called from all clients. This procedure has no other error codes than success, as there is no dynamic logic behind it.

Procedure definition:

Listing C.26: Getting access to the Lecture Quiz Service

```
public int getServiceVersion(Holder<String> version)
```

Parameters:

version A holder object containing a string where the service version will be stored.

Return codes:

0 Success.

Example:

Listing C.27: Getting access to the Lecture Quiz Service

```
Holder<String> versionHolder = new Holder<String>();  
port.getServiceVersion(versionHolder);  
String version = versionHolder.value;
```

C.1.14 joinQuiz

When student clients want to join a quiz, they call the `joinQuiz` procedure with the quiz code of the game they want to join. Before a client can call this procedure it requires to authenticate as a student client, teachers and administrators will receive an error code 4 if they attempt to call this procedure. When a request is received by the server it tries to find a running game with the quiz code supplied by the **quizCode** parameter. If this is not found the error code 3 is returned. If the join is successful the server will mark this game as the calling users active game and the client can now try to get questions and submit answers.

The **parameters** sent by the client is handled directly by the running game mode. This lets custom game modes take parameters from clients as they join in form of key and value pairs of strings though a list of `ParameterEntry` objects. The game mode implemented by the `PlainGame` class simply ignore these parameters as no other configuration is needed.

Procedure definition:

Listing C.28: Getting access to the Lecture Quiz Service

```
public int joinQuiz(String quizCode, List<ParameterEntry>  
    parameters)
```

Parameters:

quizCode The quiz code of the quiz to join.

parameters A list of parameters to send to the running game mode on the server.
Can be **null** depending on game mode.

Return codes:

- 0 Success.
- 1 Could not find a valid session.
- 2 Not logged in.
- 3 No such quiz.
- 4 Join denied.

Example:

Listing C.29: Getting access to the Lecture Quiz Service

```
List<ParameterEntry> parameters =  
    new List<ParameterEntry>();  
ParameterEntry exampleParameter = new ParameterEntry();  
exampleParameter.key = "ExampleParameter";  
exampleParameter.value = "ExampleParameterValue";  
parameters.add(exampleParameter);  
  
// Join a quiz with the quiz code "ExampleQuizCode"  
port.joinQuiz("ExampleQuizCode", parameters);
```

C.1.15 newQuiz

The `newQuiz` procedure tries to start a new quiz on the server. This procedure requires the client to be authenticated as a teacher or administrator. When this procedure is called, the server checks if a game already exists with the quiz code supplied in **quizCode**. If a game already exists the error code 4 will be returned. Should this not be the case it will try to create a game with the game mode identified by **gameMode**. If this game mode is not supported on the server the error

code 5 will be returned. The quiz identified by **quizId** will then be retrieved from the database and the game mode will send questions from this quiz until `endQuiz` is called or the last question has been answered. If the quiz is not found the error code 6 is returned.

The **parameters** sent to the server are sent directly to the game mode requested by **gameMode** in form of key and value pairs of strings. The game mode then validates these parameters and if these are valid the game is created, if not the error code 7 is returned. These parameters can be different in all game modes and can be used to configure game modes on a per quiz basis. The game mode implemented by the `PlainGame` class ignores these parameters and will accept anything.

If the game creation is successful the calling user will be marked as the owner of the game and all subsequent calls that require an active game will send the requests to this game. The game will not automatically start the first question. This is done by a call to `startNextQuestion`.

Procedure definition:

Listing C.30: Getting access to the Lecture Quiz Service

```
public int newQuiz(String gameMode, int quizId,  
    String quizCode, List<ParameterEntry> parameters)
```

Parameters:

gameMode String ID of the game mode to start.

quizId The ID of the quiz the game mode will run.

quizCode The quiz code students need to join this quiz.

parameters A list of parameters sent to the game mode identified by **gameMode**.
Can be **null** depending on game mode.

Return codes:

- 0 Success.
- 1 Could not find a valid session.
- 2 Not logged in.
- 3 Access denied.
- 4 Quiz code in use.
- 5 Unsupported game mode.
- 6 Quiz not found.
- 7 Invalid parameters.

Example:

Listing C.31: Getting access to the Lecture Quiz Service

```
List<ParameterEntry> parameters =
    new List<ParameterEntry>();
ParameterEntry exampleParameter = new ParameterEntry();
exampleParameter.key = "ExampleParameter";
exampleParameter.value = "ExampleParameterValue";
parameters.add(exampleParameter);

// Start a new quiz with the PlainGame game mode,
// a quiz with id 1, the quiz code "ExampleQuizCode"
// and the test parameters
port.newQuiz("PlainGame", 1, "ExampleQuizCode",
    parameters);
```

C.1.16 saveQuiz

The `saveQuiz` procedure takes all the information on a quiz (through the **quiz-Info argument**) and saves it to the server's database. This procedure can be used in two ways, saving changes to an existing quiz or create a new one. It also saves

questions and creates any new questions added to the quiz. This procedure can only be called as a teacher or administrator and teachers are only able to create new quizzes or change quizzes they are the owner of. All the work of this procedure is done by the running database back end.

To be able to determine whether to save or create a quiz or question the `saveQuiz` procedure looks at the given quiz and question IDs. A quiz or question with an ID of zero is considered to be new and is created. A quiz can consist of both new and previously created questions. When creating a new question the numbering of answer IDs should range from 0 to n , where n is the number of answers minus one. The `correctAnswer` field should then reference any of these answer IDs.

For more information see the `FullQuizInfo` documentation in section C.2.5.

Procedure definition:

Listing C.32: Getting access to the Lecture Quiz Service

```
public int saveQuiz(FullQuizInfo quizInfo)
```

Parameters:

quizInfo A `FullQuizInfo` object containing all the information about the quiz to save.

Return codes:

- 0** - Success.
- 1** - Could not find a valid session.
- 2** - Not logged in.
- 3** - Access denied.
- 4** - No such quiz.
- 5** - Invalid data.

6 - Unknown error.

Example:

Listing C.33: Getting access to the Lecture Quiz Service

```
// Example will not show editation of the data structure
// Get the changed or newly created quiz info
FullQuizInfo quiz = ...;
port.saveQuiz(quiz);
```

C.1.17 startNextQuestion

When starting a new quiz with `newQuiz` the first question is not started until `startNextQuestion` is called. It is also required to call this procedure after each question has ended to start the next question. This will not be done automatically on the server. The next question in the quiz is returned through the **info** parameter so there is no need for the teacher client retrieve it through a separate procedure call. This procedure is only callable from a client authenticated as a teacher or an administrator.

When a teacher client calls `startNextQuestion` all student clients that are connected to this game is able to retrieve the question and answer alternatives though a call to `getCurrentQuestion`. The time stamp is also stored so the Lecture Quiz server knows if a submitted answer comes too late or if clients try to get statistics too early. The two notable error codes of this procedure are the codes 5 and 6. Error code 5 tells the calling client that there are no more questions in this quiz and that the client can call `getOverallStatistics`. Error code 6 tells the client that the next question cannot be started as the previous question has not reached its time limit yet.

Procedure definition:

Listing C.34: Getting access to the Lecture Quiz Service

```
public int startNextQuestion(Holder<QuestionInfo> info)
```

Parameters:

info A holder object with a `QuestionInfo` object with all the information about the next question in the quiz.

Return codes:

- 0** Success.
- 1** Could not find a valid session.
- 2** Not logged in.
- 3** Access denied.
- 4** No active quiz.
- 5** No more questions in this quiz.
- 6** Another question is currently active.

Example:

Listing C.35: Getting access to the Lecture Quiz Service

```
Holder<QuestionInfo> questionHolder =  
    new Holder<QuestionInfo>();  
port.startNextQuestion(questionHolder);  
QuestionInfo question = questionHolder.value;
```

C.1.18 submitAnswer

The `submitAnswer` procedure is called by student clients when they want to submit an answer to a question. It can only be called after a student has authenticated and joined a quiz, the teacher has started a question with `startNextQue-`

stion and the time limit for that question has not been reached. A user can also only answer once on each question.

When a user sends an answer to the server it checks for the active game of that user. If it cannot find any it returns the error code 3. The answer ID is then sent to the active game who controls what the server does with it. The handling of answers are up to the game mode of the running game.

If the time limit is reached or the first question isn't started yet a call to this procedure returns the error code 6. If the answer ID sent in the **answerId** parameter is not one of the answer alternatives for the current question the error code 4 is returned. Error 5 is returned if a user tries to answer a question more than once.

Procedure definition:

Listing C.36: Getting access to the Lecture Quiz Service

```
public int submitAnswer(int answerId)
```

Parameters:

answerId The ID of the answer the client wants to submit.

Return codes:

- 0** Success.
- 1** Could not find a valid session.
- 2** Not logged in.
- 3** Not a part of any quiz.
- 4** Not a valid answer.
- 5** User has already answered.
- 6** Quiz does not accept answers at this time.

Example:

Listing C.37: Getting access to the Lecture Quiz Service

```
// Submit an answer with the ID 1
port.submitAnswer(1);
```

C.2 Exported data types

Many data types are exported as a part of the Lecture Quiz web service. These are classes that hold data and is transferred between the server and the clients. In this section we will explain these data types in detail.

C.2.1 Answer

The Answer class is a simple class that holds a single answer alternative. The definition of this class can be seen in Listing C.38. The *id* member variable holds the ID of the answer option while the *text* variable holds the actual answer text.

Listing C.38: Answer class definition

```
public class Answer {
    public int id;
    public String text;
}
```

C.2.2 QuestionInfo

The QuestionInfo class holds most of the information about a question, with the exception of the correct answer. This is the data that gets sent between clients

when they request the currently running question on the server. The class definition can be seen in Listing C.39. The *id* member variable is the ID of the question. The *text* variable holds the actual question text. All the possible answer alternatives are stored in the member variable list *answers*. The *timeout* variable tells how many seconds the time out should be for this question and the *timeleft* variable tells how many of those seconds that are left when the request for this information was received.

Listing C.39: QuestionInfo class definition

```
public class QuestionInfo {
    public int id;
    public String text;
    public ArrayList<Answer> answers;
    public int timeout;
    public int timeleft;
}
```

C.2.3 QuizInfo

When a teacher client requests a list of all available quizzes on the server it gets a QuizInfo object in return. The *id* variable holds the ID of the quiz and *name* is the display name. A short description of what the quiz is about can be found in the *description* variable. The *owner* variable is the user name of the user who created the quiz and the number of questions this quiz contains can be found in the *questionCount* variable.

Listing C.40: QuizInfo class definition

```
public class QuizInfo {
    public int id;
    public String name;
    public String description;
    public String owner;
    public int questionCount;
}
```

```
}
```

C.2.4 FullQuestionInfo

The FullQuestionInfo class is an extension of the QuizInfo class. It only holds one extra data field and that is the ID of the correct answer. The definition of this class can be seen in Listing C.41. This class is used during the editing of quizzes and therefore requires the correct answer to be known. Adding a new question as a part of a quiz the *id* field is set to 0 and the IDs of all the answers are numbered upwards starting from 0. The *correctAnswer* field then references to one of these answer IDs.

Listing C.41: FullQuestionInfo class definition

```
public class FullQuestionInfo extends QuestionInfo {  
    public int correctAnswer;  
}
```

C.2.5 FullQuizInfo

The FullQuizInfo is not a direct extension of the QuizInfo class but they have many similar variables. The only difference is the *questions* variable. It replaces the question count with a list of all the actual questions in the form of FullQuestionInfo objects. A FullQuizInfo object holds all the information about a quiz and is used during editing of quizzes. When creating a new quiz the *id* field should be set to 0. This signals that the quiz is a new one and it is created on the server. Any questions that do not already exist in the database should also have the ID 0. See the FullQuestionInfo documentation for more information.

Listing C.42: FullQuizInfo class definition

```
public class FullQuizInfo {  
    public int id;
```

```
public String name;
public String description;
public String owner;
public List<FullQuestionInfo> questions;
}
```

C.2.6 StatisticsEntry and ParameterEntry

The two classes `StatisticsEntry` and `ParameterEntry` are identical with the exception of the type name. Both classes have two member variables, *key* and *value*. These are strings and simply represent a key and value pair. The `ParameterEntry` definition can be seen in Listing C.43.

Listing C.43: `ParameterEntry` class definition

```
public class ParameterEntry {
    public String key;
    public String value;
}
```

C.2.7 GameModeInfo

The `GameModeInfo` class holds the ID and the name of a game mode in the variables *id* and *gameName* respectively. These are both strings and are usually returned in the form of a list of objects of this class when a client asks the server for supported game modes.

Listing C.44: `GameModeInfo` class definition

```
public class GameModeInfo {
    public String id;
    public String gameName;
}
```

C.2.8 GameState

When a client asks for the updated status of an ongoing game the results are returned in a `GameState` class. The definition of this class can be seen in Listing C.45. The member variable *quizCode* holds the quiz code of the running game. The *playerCount* variable is a list of player counts, but it is built as a list of different counts. These different counts can represent players on different teams should the game mode require it. If teams are not a part of the game mode this list only contains a single entry which is the total number of players. The *currentQuestion* variable holds the number of the current question. This number starts on 1 for the first question and counts upwards. The last parameter is the *questionCount* which holds the total number of questions in the running quiz.

Listing C.45: `GameState` class definition

```
public class GameState {
    public String quizCode;
    public ArrayList<Integer> playerCount;
    public int currentQuestion;
    public int questionCount;
}
```

Bibliography

- [1] Ole Kristian Mørch-Storsteing & Terje Øfsdahl. Game enhanced lectures - an implementation and analysis of a lecture game. Master's thesis, NTNU, 2007.
- [2] A. Wessels S. Fries H. Horz N. Scheele & W. Effelsberg. Interactive lectures: Effective teaching and learning in lectures using wireless networks. *Computers in Human Behavior*, 23(5), 2007.
- [3] V.R. Basili. The experimental paradigm in software engineering. *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, 1993.
- [4] Linda Rising & Norman S. Janoff. The scrum software development process for small teams. *IEEE Software*, 2000.
- [5] P. W. Jordan B. Thomas B. A. Weerdmeester & A. L. McClelland. *Usability Evaluation in Industry*, chapter SUS - A quick and dirty usability scale, pages 189–194. CRC Press, 1996.
- [6] E. Tews Bär, H. and G. Rössling. Improving feedback and classroom interaction using mobile phones. Master's thesis, Darmstadt University of Technology, Germany, 2005.
- [7] Uw classroom presenter. <http://www.cs.washington.edu/education/dl/presenter/>.
- [8] et al. Linnell, M. Supporting classroom discussion with technology: A case study in environmental science. Master's thesis, 2006.

- [9] *ClassInHand Software User Guide*, 2003.
- [10] Classinhand. <http://classinhand.wfu.edu/>.
- [11] Ezclickpro. <http://www.aclasstechnology.com/ezClickPro/index.html>.
- [12] Government backs “buzz! for schools”.
<http://www.mcvuk.com/news/25249/Government-backs-Sonys-Buzz-for-schools>.
- [13] Free and open source java: Faq. <http://www.sun.com/software/opensource/java/faq.jsp>.
- [14] Microsoft product lifecycle search. <http://support.microsoft.com/lifecycle/search/default.aspx?alpha=.NET+Framework>.
- [15] Microsoft .net framework. <http://www.microsoft.com/net/default.aspx>.
- [16] The mono project. <http://www.mono-project.com/>.
- [17] OpenGL overview. <http://www.opengl.org/about/overview/>.
- [18] W3C Recommendation. Soap version 1.2. <http://www.w3.org/TR/soap12-part1/>, April 2007.
- [19] *Java Web Services: Up and Running*. Martin Kalin, 2009.
- [20] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [21] About mysql. <http://mysql.com/about/>.
- [22] About postgresql. <http://www.postgresql.org/about/>.
- [23] L. Bass P. Clements & R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

- [24] Philippe Kruchten. The "4+1" view model of software architecture. *IEEE Software*, 1995.
- [25] Jsr-231 java™binding for the opengl®api. <http://jcp.org/en/jsr/detail?id=231>.
- [26] Jogl - java binding for the opengl api. <http://jogamp.org/jogl/www/>.
- [27] Google web toolkit mvp documentation. <http://code.google.com/webtoolkit/articles/mvp-architecture.html>.