UNIVERSITY OF OSLO
Department of informatics

DHIS and Joly: two distributed systems under development: design and technology

Master thesis

Therese Steensen
Hanne Vibekk

May 2006

Acknowledgements

First and foremost we would like to thank our supervisor Arne Maus for his support, engagement and all the valuable discussions we have had, and for all the time he has dedicated to us during our time at the Institute of Informatics. We would also like to thank the HISP team in Oslo for their contributions during the initial phase of our study, and the other students at Ifi we have spent so many hours with at 'Termstua'.

I would like to thank my boyfriend and my family, for their support and confidence in me while studying toward the Master's Degree. I like to dedicate my contributions to this master thesis to my sister Anette, who always believed in me and told be I could do anything I put my mind to.

April 2006,

Therese Steensen

A special thanks to my fiancé, Alexander, who has been a tremendous support to me, and was able to make me laugh even in my most frustrated hours. I would also like to thank my family for always believing in me.

April 2006,

Hanne Vibekk

Table of contents

1 INTRODUCTION 1

1.1 RESEARCH OBJECTIVE 2
1.2 SUMMARY OF THE CHAPTERS IN THIS THESIS 3

2 THE DISTRICT HEALTH INFORMATION SYSTEM'S BACKGROUND 5

2.1 HEALTH CARE SERVICES IN SOUTH AFRICA UNDER THE APARTHEID REGIME 5
2.2 HEALTH INFORMATION SYSTEM PROGRAM (HISP) 6
2.2.1 AN HISTORICAL REVIEW OF HISP 8
2.2.2 DISTRICT HEALTH INFORMATION SYSTEM 10
2.2.3 STANDARDIZATION OF HEALTH CARE DATA 11
2.2.4 DATA COLLECTION AND ANALYSING 12
2.2.5 THE HISP NETWORK 13
2.3 FREE/OPEN SOURCE SOFTWARE (FOSS) 14
2.3.1 FREE SOFTWARE 14
2.3.2 OPEN SOURCE SOFTWARE 16
2.3.3 THE DIFFERENCE BETWEEN FREE AND OPEN SOURCE SOFTWARE 17
2.3.4 FOSS IN DEVELOPING COUNTRIES 17
2.4 DISTRICT HEALTH INFORMATION SOFTWARE 18
2.5 DHIS 1.3 20
2.6 DHIS 1.4 30
2.7 SUMMARY 31

3 THEORETICAL BASIS FOR THE STUDY 33

3.1 DHIS 2.0 33
3.1.1 REQUIREMENTS 35
3.1.2 TECHNOLOGY 35
3.1.3 ARCHITECTURE AND DESIGN 38
3.1.4 THE ROAD AHEAD FOR DHIS 39
3.2 FREE/OPEN SOURCE SOFTWARE EVALUATION 40
3.2.1 FOSS LICENSES 41
3.2.2 FOSS SOFTWARE DEVELOPMENT AND COMMUNITY 43
3.2.3 FOSS STABILITY AND DEVELOPMENT ACTIVITY 45
3.2.4 FOSS DOCUMENTATION 46
3.2.5 FOSS SELECTION PROCESS FOR JOLY 47
3.3 SOFTWARE MAINTENANCE 48
3.3.1 DESIGNING FOR CHANGE IN JOLY 51
3.3.2 INVESTIGATION PROCESS 52
3.4 SUMMARY 52

4 JOLY 55

4.1 SYSTEM REQUIREMENTS 56
4.1.1 FUNCTIONAL REQUIREMENTS 57
4.1.2 NON-FUNCTIONAL REQUIREMENTS 58
4.1.3 IMPLEMENTED FUNCTIONALITY 59
4.1.4 NON-RESTRICTED FUNCTION 61
4.1.5 RESTRICTED FUNCTIONS - JOLYADMIN 65
4.2 SUMMARY 71

5 ARCHITECTURE, DESIGN AND TOOLS FOR JOLY 73

5.1 ARCHITECTURE 74
5.2 JAVA 2 ENTERPRISE EDITION (J2EE) DEVELOPMENT PLATFORM 75
5.3 WEB SERVERS 77
5.3.1 JETTY HTTP SERVER AND SERVLET/JSP CONTAINER 78
5.3.2 APACHE TOMCAT SERVLET/JSP CONTAINER 79
5.3.3 OPEN SOURCE J2EE APPLICATION SERVERS 80

5.4 DESIGN PATTERNS 80
5.4.1 INVERSION OF CONTROL AND THE DEPENDENCY INJECTION PATTERN 82
5.4.2 DESIGN WITHIN THE LAYERS 85
5.4.3 DESIGN MAINTENANCE 86
5.5 FRAMEWORKS 86
5.5.1 HIBERNATE 87
5.5.2 SPRING 88
5.6 SUMMARY 92

6 DETAILED LAYER DESIGN OF JOLY 95

6.1 PERSISTENCE LAYER 95
6.1.1 DATABASE 95
6.1.2 SELECTION OF DATABASE 100
6.1.3 PERSISTENCE LAYER DESIGN 101
6.1.4 DATA ACCESS FROM JAVA 103
6.2 DOMAIN LAYER 108
6.2.1 DECOUPLING DOMAIN LOGIC 109
6.3 PRESENTATION LAYER 111
6.3.1 THE MODEL-VIEW-CONTROLLER PATTERN 111
6.3.2 WEB MVC FRAMEWORKS 112
6.4 SUMMARY 116

7 IMPLEMENTATION OF JOLY 119

7.1 DATABASE 119
7.2 IMPLEMENTATION OF THE PERSISTENCE LAYER 120
7.2.1 HIBERNATE ARCHITECTURE 121
7.2.2 CONNECTION POOL 122
7.2.3 HIBERNATE QUERY LANGUAGE - HQL 123
7.2.4 REQUIRED HIBERNATE LIBRARIES 123
7.2.5 OBJECT/RELATIONAL MAPPING 124
7.2.6 SPRING AND HIBERNATE 130
7.3 IMPLEMENTATION OF THE DOMAIN LAYER 136
7.3.1 DOMAIN OBJECTS 136
7.3.2 DOMAIN LOGIC 137

7.4 IMPLEMENTATION OF THE WEB LAYER **145**
7.4.1 JAVA SERVLETS 145
7.4.2 JAVA SERVER PAGES 146
7.4.3 JAVA SERVER PAGES STANDARD TAG LIBRARY (JSTL) 147
7.4.4 THE EXPRESSION LANGUAGE (EL) 147
7.4.5 JAVASCRIPT 148
7.4.6 ALTERNATIVE TECHNOLOGIES 149
7.4.7 IMPLEMENTATION OF THE WEB LAYER USING SPRING MVC 149
7.4.8 VIEW IMPLEMENTATION 153
7.5 SUMMARY **157**

8 EXPERIENCES **159**

8.1 EXPERIENCE GAINED BY THE ANALYSIS PROCESS OF THE DHIS SOFTWARE **159**
8.2 FREE AND OPEN SOURCE SOFTWARE EVALUATION PROCESS **160**
8.3 USING FREE AND OPEN SOURCE DEVELOPMENT TOOLS **163**
8.4 FRAMEWORK RELATED EXPERIENCES **162**
8.4.1 PROBLEMS RELATED TO THE DEVELOPMENT PROCESS 163
8.4.2 PROBLEMS RELATED TO MAINTENANCE 165
8.5 DESIGN PROCESS **166**
8.6 INCREMENTAL DEVELOPMENT PROCESS **168**
8.7 SUMMARY **169**

9 CONCLUSION **171**

10 BIBLIOGRAPHY **177**

APPENDIX A **185**

APPENDIX B **195**

APPENDIX C **227**

APPENDIX D **229**

Acronyms and glossary

API	Application Programming Interface
ASF	Apache Software Foundation
ASP	Active Server Pages
CLR	Common Language Runtime
CRUD	Create, Read, Update and Delete
DAO	Data Access Object
DBMS	Database Management System
DHIS	Acronym used to describe both District Health Information System and District Health Information Software
EAR	Enterprise Application aRchive
EDS	Essential Data Set
EJB	Enterprise Java Bean
EL	Expression Language
FOSS	Free/Open Source Software
GIS	Geographic Information System
GPL	General Public Licence
GUI	Graphical User Interface
HIS	Health Information System
HISP	Health Information System Program
HISPML	HISP Multi-Language Library
HQL	Hibernate Query Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol

IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IoC	Inversion of Control
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDO	Java Data Objects
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JRE	Java Runtime Environment
JSP	Java Server Pages
JSTL	Java Server Pages Standard Tag Library
LGPL	Lesser General Public License
MVC	Model-View-Controller
ODBC	Open Database Connectivity
ODL	Object Definition Language
OODBMS	Object-Oriented Database Management System
OQL	Object Query Language
ORDBMS	Object-Relational Database Management System
ORM	Object Relational Mapping
OSI	Open Source Initiative
OSS	Open Source Software. Has the same meaning as FOSS.
POJO	Plain Old Java Object
RDBMS	Relational Database Management System
SDK	Software Development Kit
W3C	World Wide Web Consortium
WAR	Web Application aRchive
XML	eXtensible Markup Language

Chapter 1

Introduction

In this master thesis we investigate how to develop a web application by using free and open source tools, and how to design for flexibility in regards to software change and platform-independency. By incorporating free and open source tools, the requirement of a flexible design is higher than that of a traditional “in house” development. The nature of free and open source products is a constantly evolutionary process of change in software code. For an application to use these products, and be able to evolve, it cannot rely on a free and open source product to support the future requirements of the application. A need for substituting the product in the future is likely, and therefore, the design must make the application able to adapt to the changes.

The background for our investigation is the Health System Information Program (HISP), which started out as a pilot project in a province in South Africa in 1995 to contribute to reform the country’s health system that had been influenced by the apartheid regime. HISP’s goal was to develop a district health information system (DHIS) to improve the use of health data for decision-making at the district level in South Africa’s health system. DHIS became a success, and several developing countries have implemented DHIS as their health system, and others are in the process of adapting DHIS to their health service. The development of DHIS started a decade ago, and has now reached a phase in the maintenance process that requires a complete reimplementation of the system to support new requirements, both functional and non-functional.

DHIS is based on the open source principal, which means that the software is distributed with its source code, to be freely used and modified by the users. The reimplementation of

DHIS is done by using an approach based on a community of developers sharing code and cooperating in the development process and incorporating existing open source products. The main focus of the reimplementation is to develop DHIS as a platform-independent system to avoid the constraints of vendor-specific products.

Through the development of a web application, we investigate various aspects of an open source development process using open source tools, and the impact this has on the application and the development process. By using technology based on standards, we seek to decrease the dependency to specific technological platforms. By designing for anticipated change, we seek to prepare the application for the inevitable maintenance process it will be subject to after its release.

1.1 Research objective

We started out with an objective to develop a prototype of a new free and open source platform-independent version of DHIS. It was intended that the prototype would address the main problem areas that previous versions of the health information system had, and trigger a process of developing a complete new version. A while after we had started our research, an initiative to develop such a complete new version started at the HISP team in Oslo. Several masters' degree students and researchers quickly became part of the team, and HISP teams in other countries also joined in and the development started. As our objective had been to develop only a prototype of the system, which would have been a complex task in itself considering the complexity and comprehensiveness of the system, it now became redundant since its intentions of triggering a process of development already had started.

We could have joined the HISP team and contributed to developing a new version of DHIS, but seeing that the system development process would be comprehensive and take many years - we would only had played a little part in it and would have finished our research before DHIS was fully developed. Since the beginning of a development process always includes a lot of work with developing a requirement specification and a design, our objective probably wouldn't have included programming and development of the system as our supervisor had intended.

We therefore needed to find another objective, and was partly asked and partly saw the need of exploring the challenges in developing a web application by using free and open source products and tools, and designing a system that supports the non-functional requirements of DHIS. In addition we wanted to look at how DHIS could make the process of maintaining the system after its release easier and ensure a long operating time, by making the design and architecture of the system capable of adapting to changes. To explore these challenges we have developed a smaller web application - Joly, but with the same non-functional requirements as DHIS. Our main problem addressed is therefore as follows:

Investigate how to design and develop a web application by using free and open source products and tools to support the non-functional requirements of the health information system DHIS, and being able to easily adapt to change.

The problem area identified above led to the two main theoretical topics we have considered in this thesis:

1. Evaluation of free and open source solutions.
2. Design for change to ease maintenance process.

1.2 Summary of the chapters in this thesis

Chapter 2: This chapter presents the background for the start up of the HISP project, and their goal of improving health systems at a district level in developing countries with the development of the district health information system - DHIS. We identify the problem areas that have resulted in the need of a complete re-implementation and re-design of DHIS, and possible solutions to these problem areas that should be considered during the development of a new version of DHIS.

Chapter 3: In this chapter we describe the theoretical basis for our study; HISP's development of the new version of DHIS (DHIS 2.0) and their non-functional requirements, evaluation of free and open source technology and how they differ from proprietary/commercial software, and how to design for change to ease the maintenance process of a system.

Chapter 4: Joly is presented in this chapter, and is the web application we have developed to investigate the challenges DHIS can encounter when developing a free and open source platform-independent web application. Joly is to be used for online submitting of mandatory student assignments in programming courses at the Institute of Informatics at the University of Oslo, to gather handed inn assignments and processing them in order to find attempts to cheat.

Chapter 5: This chapter presents the architecture, design and the choice of free and open source tools for the development of Joly, in order to meet the non-functional requirements HISP has chosen for DHIS 2.0. In c**hapter 6**, a presentation of the detailed layer design of Joly is given. The layers of the architecture laid out in chapter 5; the persistence layer, domain layer and the presentation layer, are described in separate sections, focusing on their assigned responsibilities. The detailed design decisions are presented, as well as the free and open source tools used at each layer. In **chapter 7**, we revisit the design decisions in a description of the implementation of the Joly application. We present how we have resolved the functional as well as the non-functional requirements with the technology and frameworks we have used.

Chapter 8: In this chapter we summarize the experiences we have gained during our investigation process, and identify the problems we have encountered and point out the most relevant findings of our investigation. We discuss our experience with the analysis of the DHIS software, the evaluation and use of free and open source tools, the experiences of designing for change and the incremental development process used. In **chapter 9** we conclude our investigation and relate our experiences to DHIS. We present our proposals for further work and the questions that still remain to be answered.

Chapter 2

The District Health Information System's background

This chapter gives a description of the Health Information System Program (HISP) and the District Health Information System (DHIS) developed by HISP. DHIS is the starting point for the investigation this thesis presents, and the system's non-functional requirements to address its problem areas in a new version, are the basis for our development of Joly. In the following, we describe the background for the start up of HISP, and their goal of improving local health systems in developing countries through the development of DHIS. We will in addition identify the problem areas of version 1.3 of the health information system that led to the process of a complete reimplementation of the system, and how these problems can be addressed by HISP when developing a new version of DHIS.

2.1 Health care services in South Africa under the apartheid regime

Health care services in South Africa under the apartheid regime reflected the racial divisiveness in the country. Although the country's level of economic development were higher than some of its neighbours, the health status was worse because the resources were used to care for a small portion of society on the basis of race and wealth [1]. 60% of South Africa's health care resources were used by the private sector, who served about 20% of the population whom were mostly white and wealthy[2].

After the fall of apartheid and the advent of democracy in 1994, the government committed itself to reforming the country, concentrating on the issues that were neglected under apartheid such as poverty, unemployment, education and health care. The African National Congress-led government adopted the Reconstruction and Development Programme (RDP) as its guiding policy framework to provide equal opportunities for all South African citizens. The RDP recognize that reconstruction and development are parts of an integrated process, and that the key to this is an infrastructure program that will provide access for all its citizens to modern and effective services such as electricity, water, education and health [3].

Improvement of the health sector was an important priority for African National Congress (ANC), and the ANC Health Department and consultants from World Health Organization (WHO) and UNICEF developed a National Health Plan with a "health vision" to improve the health of the citizens through "achievement of equitable social and economic development"[4]. The National Health Plan involved creation of a National Health System for coordination of all aspects of both public and private health care delivery, a decentralized system of health districts, and development of a national health information system (HIS) for collection and analysing data for planning and management purposes.

2.2 Health Information System Program (HISP)

After the South African government's new health service policy was established, a number of projects started up to reform the country's health information system as part of the RDP. One of those projects was the Health Information System Program (HISP), which started as a pilot project in the Western Cape province in 1995 to develop a district-based HIS. HISP is a research and development project that initially involved a collaboration of universities in South Africa (University of the Western Cape and University of Cape Town), Norway (University of Oslo) and the Western Cape Department of Health.

The project has focused on the systems used in local health institutions. By giving the health institutions the ability to analyse the locally collected health care data, they have the ability to delegate the resources after local data and knowledge. This bottom-up approach is in strong contrast to the top-down approach that is common in developing countries with a centralized HIS. With a top-down approach the result is often that the ability to evaluate the information

in connection with its original context is lost, and that variances in analyses important at a local level becomes insignificant in evaluation at a national level [5].

HISP's primary goal is to

> *Design, implement, and sustain HIS following a participatory approach to support local management of health care delivery and information flows in selected health facilities, districts, and provinces, and its further spread within and across developing countries [6].*

To reach this goal, HISP's primary focus has been the two interrelated processes of developing standards for primary health care data, and the design and development of a district health information system (DHIS) to support their implementation.

Important aspects of the processes has been [7]:

1. Participatory prototyping of both DHIS and the adoption process. The software should support the process, not drive it.
2. Bottom-up approach; give training and empowerment to local health workers to use the data, and establishment of routines and procedures for handling of information at health facilities and district levels.
3. Support the process of defining good health data standards and indicators for them.
4. Scaling and sustainability. It is important that the processes are adopted in a good fashion, and that the changes are working and evolving when the HISP team pulls out.

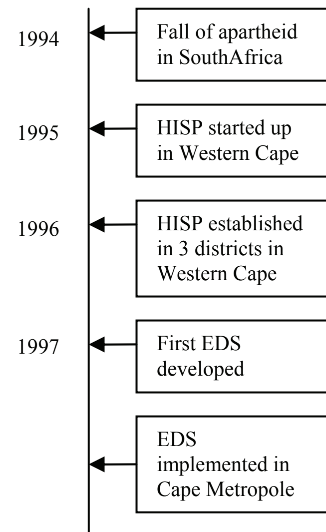
Even though the focus of the processes have been directed to the district levels, HISP has also worked on the top levels. The reason for this is that previous research has shown that working on networks, instead of singular units, improves the sustainability of a project [6]. In South Africa the work on the top levels has included gaining a political and financial support for the project and the development of a wider HIS.

HISP has been exposed to both technical and social challenges. The reason for their success in South-Africa has been the choice of correct alliances and support from key actors, combined with simple and functional software. The result is that the coverage of health data in South-

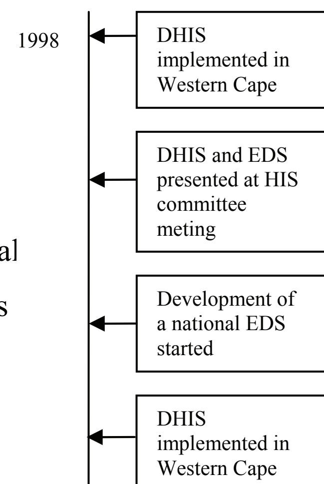
Africa never has been better than it is today, - 95 % national coverage at the end of 2001. Still, it remains a great deal of work in building an information culture, and supporting the collaboration between district level and higher levels. [7]

2.2.1 An historical review of HISP

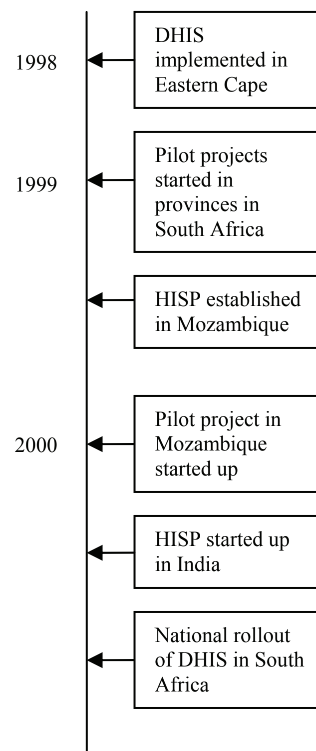
The project received funding from the Norwegian Agency of Development Cooperation (NORAD) from 1996 through 1998. During this period of time HISP was established in three health districts in and around Cape Town, with the initial focus of identifying information needs and engage the end users and local management teams in the process of developing a new health information system. The HISP team discovered the need for a national health data collection standard, since each province used different datasets, definitions and standards. HISP collaborated with local managers to develop an essential data set (EDS), and in 1997 it was implemented in Local Government health facilities in the Cape Metropole.



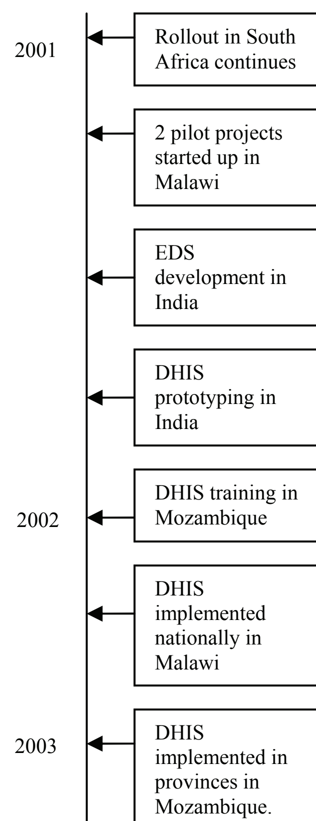
In 1998 the first prototype of DHIS was implemented and used to capture monthly data at district, regional and provincial levels in Western Cape. The same year a survey of DHIS and the EDS was presented at the national HIS committee meeting in South Africa, which resulted in the start of a process of developing the first national essential dataset. Compared to other high-tech and high-cost projects that started up at after the fall of apartheid, HISP was low-cost and fairly simple with a partly paper-based system for collecting an essential standardized dataset for primary health care [2]. While the other projects failed, the relative success of HISP made it attractive, and DHIS was implemented in all districts in Western Cape in 1998.



The Norwegian Council of Universities' Committee for Development Research and Education (NUFU) funds HISP for the period 1999-2001.



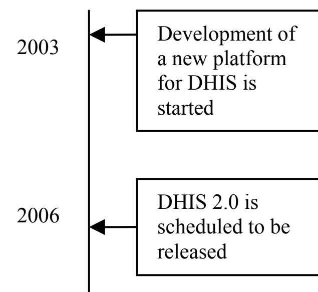
In 1999 pilot projects were started in several provinces in South Africa, and HISP was established in Mozambique as the first node in the international HISP network. The process of starting up HISP at the district level in Mozambique proved to be more difficult than in South-Africa, because the provinces wasn't self-governed and a decision to change the software of the national HIS had to be done at the governmental level. HISP started a pilot project in 2000 in the northern Mozambique despite the lack of official support, and the first version with hard-coded language translation has been operational since [6]. In 2000 HISP also started a project in India, where it was performed a situation analysis in two states. The next step for HISP in South Africa was to implement DHIS in all the provinces, so in 2000 each province developed business plans for the rollout, and some of the provinces started the implementation. The project receives funds from EQUITY/USAID[1] to support the national rollout for the period 2000-2001.



Throughout 2001-2002 the rollout continues in South-Africa, along with training of health personnel and further development and improvements of DHIS. In 2001 DHIS was adapted by Malawi and tested in two pilot districts, and in 2002 it was implemented nationally. The work in the other nodes in the HISP network also continues; in 2001 in India the development of a national EDS was carried out and the process of implementing the datasets using DHIS started in nine primary health care clinics, and in 2002 in Mozambique training and initiatives continued in the three provinces where the pilot projects had started up - and DHIS was implemented in many districts in those provinces in 2003.

[1] EQUITY is a South African Ministry of Health project financed by the United States Agency for International Development (USAID).

In 2003 the work of re-implementing the system on a new platform is started. Because the core modules in DHIS (version 1.3) is Microsoft Office based, the system is not platform-independent and it is not scalable for countries with many citizens that need to store large amounts of health data, like India (Microsoft Access database has limited storage abilities). These are just two of the reasons for why HISP chose to start the development of a "new" DHIS (version 2.0). A more in-depth description of functions and their technical requirement, along with a discussion about the problem areas in the first version (1.3) of DHIS is given later in this chapter. The first release of DHIS version 2.0 is scheduled to be finished in December 2006.



2.2.2 District Health Information System

The components that generally constitutes a health information system (HIS) is the tools and procedures that a health program uses to collect, process, transfer and use data for monitoring, evaluation and control. DHIS is an acronym used to describe both the system (District Health Information System), and the software (District Health Information Software) that is used to handle the data that is collected through the system. The term DHIS includes the procedures and formats that is used by health facilities to collect and report data, in addition to the roles and the authority that enables health workers to use their data to improve health service performance. [8]

HISP focuses on strengthening the health systems at the district level, and promotes the use of health data at a local level to make decisions in connection with the local context. DHIS is used to collect these health data. DHIS isn't reckoned as the driving factor for HISP, but as the factor that makes it possible for HISP to succeed. Nevertheless DHIS is an important factor, since the system makes changes in the institutions possible, and thereby increases the probability that the system together with its routines will have a longer operating time.

2.2.3 Standardization of health care data

The process of standardization of primary health data to be collected both globally and locally has been one of the primary goals of HISP, and they aimed at developing a national and provincial EDS and at the same time encourage the districts to develop their own additional data sets. The national EDS in South Africa, developed through collaboration and negotiations between the provinces and the national healthcare programs, consists of 60 to 70 data elements [6]. The standardization of health data is, and will continue to be, an ongoing process because it is important that it adapt to changes in e.g. the context of health services.

To ensure that the necessary data elements were collected by health care facilities at the different levels in a health care system, a hierarchy of standards was developed (see figure 2.1). The health facilities at the bottom of the hierarchy generally needs to register more detailed data of e.g. a tuberculosis patient to ensure proper treatment, while a district would only need statistics about the treatment of the patients to manage their tuberculosis program. The hierarchy of standards supports the need for registration of more detailed information in bottom levels of the health information system. The principle of the hierarchy of standards is that a dataset at one level has to include the elements from the EDS for the above level, but still have the freedom to include the data elements they think is necessary and essential in the dataset at their level. This framework functions as an interface and a gateway for communication between relatively independent actors across organizational structures and hierarchies, and enhances the integration between and within the levels. [2]

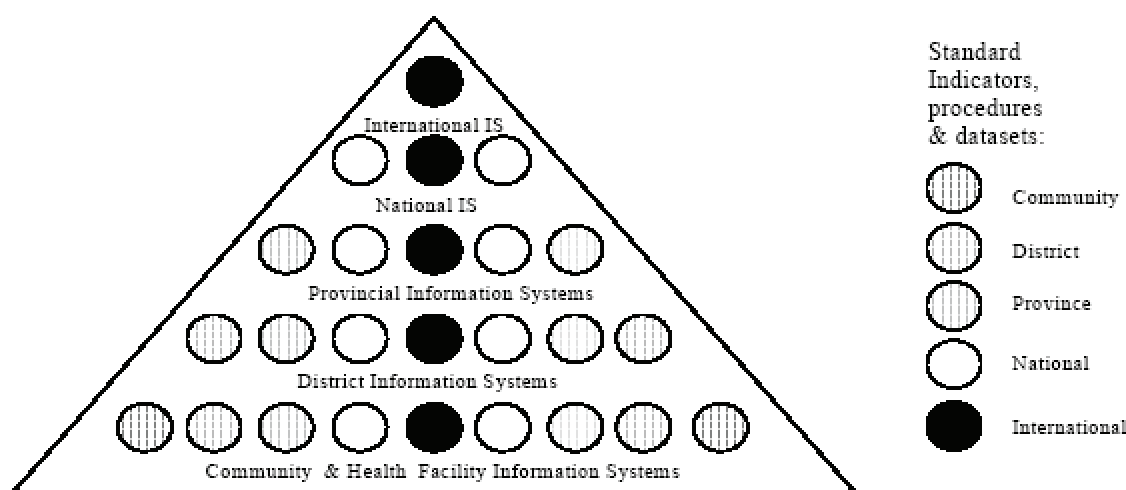


Figure 2.1 Hierarchy of standards [2].

The database in DHIS has been developed to support the hierarchy of datasets, and in version 1.3 the national part of the hierarchy is divided into five organization units; 1) national district of health, 2) provincial district of health, 3) district municipality, 4) local municipality, and 5) facility. The hierarchy is flexible, and a country that implements DHIS only has to use the parts of the hierarchy that reflects their health system, but in version 1.3 of DHIS it is not possible to extend the number of levels in the hierarchy. With restructuring in a countries health organization's structure, or for countries that have more levels in their hierarchy than five, it is a disadvantage that DHIS doesn't support *n* levels. To make it easier for countries using DHIS to use the hierarchy correctly HISP have expanded the hierarchy from five to six levels in DHIS version 1.4, and plan to support *n* levels with the release of DHIS 2.0.

2.2.4 Data collection and analysing

DHIS is designed for collection of different types of data:

- Routine data (monthly, quarterly, yearly).
- Semi-permanent data (population estimates, equipment, number of personnel).
- Survey data (ad-hoc surveys, regular surveys, client satisfaction surveys).

The demand for and supply of data can vary at different levels of a health system, and not everything needs to be known at every level of the system. The health system displayed in figure 2.2 shows an example of health data needs and sources at different levels of a health care system. At the lower levels of the system, the quantity and detail of data needed is generally greater than at the higher levels, because that is where decisions on the care of individuals are made. At higher levels, broader policy-making usually takes place, and if the health data provided those levels are too detailed it can be difficult to use them effectively. [9] To analyse the detailed/raw data collected at the lower levels in a health system, to bring more meaning and make it useful for the higher levels, DHIS uses indicators.

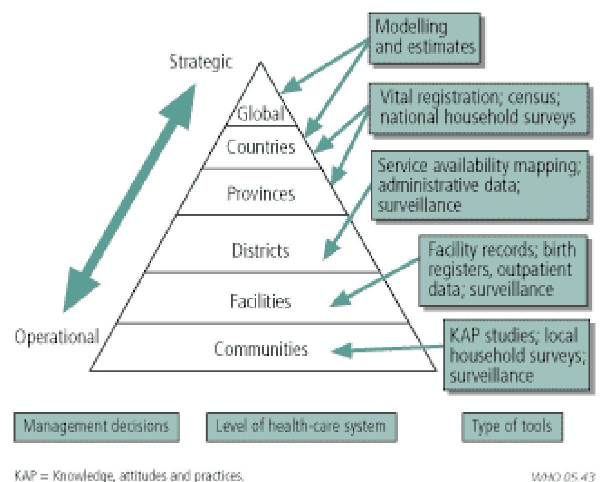


Figure 2.2 At the communities and facilities level, information is needed for effective clinical management. At the district level, health information enables decision-making regarding the effective functioning of health facilities and of the health system as a whole. At higher levels, health information is needed for strategic policy-making and resource allocation. [9]

The use of indicators is recommended by WHO for monitoring progress towards reaching the goals of implementing health policies, programmes and services[10]. The indicators can be useful for measuring and improving primary health care over time and across regions. An indicator is the difference between absolute and relative numbers, and is often the result of routine information collected (used as the numerator) and semi permanent data (used as the denominator)[7]. Indicators must be carefully selected in order to be relevant and give meaning for the various users. An example of an indicator can be the coverage factor of immunized children in a region. The number of immunized children collected as routine information (e.g. 1000) would not alone give the health planner or manager enough information to make strategic policy-making or resource allocation – the data needs to be seen in combination with the number of children that is not immunized. If the data about the population of infants in a region is present (e.g. 10 000), the indicator would be 1000/10000= 0,1, or 10% would be the coverage factor [7].

2.2.5 The HISP network

HISP has since its beginning in South Africa expanded to an international network of HISP teams. The HISP network consists of several countries in different continents, which are in different stages of implementing and using DHIS. South Africa and Malawi in Africa are the only two countries that have completely implemented version 1.3 of DHIS in all their health care facilities. Mozambique and India have only partly implemented DHIS, but both have plans for a complete rollout. Tanzania and Ethiopia are in the early stages of testing and adaptation of DHIS to their conditions in the national health systems. HISP also started projects in Cuba and Mongolia, but due to political issues in Cuba and the lack of technically competent people in Mongolia, it was difficult to sustain the projects. Other countries which also has adopted and started to use DHIS in parts of their countries are Botswana and China. [6]

There are currently HISP teams in the following countries; Ethiopia, India, Mozambique, Norway, South Africa, Tanzania, Vietnam and Zanzibar. Teams in Norway, India, South Africa, and Vietnam collaborate for the development of DHIS 2.0.

2.3 Free/Open Source Software (FOSS)

"Tout ce qui n'est pas donné est perdu."

(Anything not given away is lost.) - French saying

Before we describe the DHIS software more in depth, we need to look at the principles of Free Software and Open Source Software (FOSS) as DHIS, and Joly also, is Open Source Software. An application that employs FOSS is itself considered as a FOSS (unless the license permits otherwise). There are two movements in the world of Free and Open Source Software, the Free Software movement and the Open Source movement which jointly can be referred to as FOSS (explained below). FOSS is software that doesn't have the same restrictions on using and copying the software as proprietary[2] software, and gives the user access to human-readable source code instead of only machine-readable binary code that is common with proprietary software.

Proprietary software has been dominant over the last decades in the software market, but the development and use of FOSS has in the last 10 years increased and gained a considerably market share in some areas (e.g. Apache is the #1 web server with a three times higher market share than the next-ranked (proprietary) competitor (Microsoft), and GNU/Linux is the #2 web serving operating system on the public Internet (#1 is Microsoft Windows)[11]). Proprietary software doesn't always have the best solutions anymore, and Bernad Golden[3] stated in 2004 that "Within two years, any software selection process for enterprise environments will include the question: Is there an open source alternative?"[12]. Ergo – FOSS should be evaluated when the need for a new software solution arises. The next chapter includes a more in depth description on how FOSS differs from proprietary software, and what properties that needs to be evaluated before a FOSS product is adapted.

2.3.1 Free software

In 1984 Richard Stallman left his job at the MIT Artificial Intelligence Lab as an operating system developer, and started a project to develop free software, the GNU Project. Stallman

[2] Proprietary software means software that can be sold for money as commercial software or available for free as freeware.

[3] Gernard Golden is the CEO of Navica Inc. which is a professional services firm focused on open source, providing strategy, implementation and training services to its clients.

wanted to make a counterpart to the proprietary software that forced the users to sign nondisclosure agreements that took away the users freedom to share or change the software, and develop software that give the user's freedom to distribute the software, and change or add functionality to it to satisfy its own needs.

The term "Free Software" refers to the user's freedom, not the price of the software. Although Free Software mostly is free of charge and possible to download over the internet, there are no restrictions to sell copies of the software, it is in fact one of the rights the user's have with Free Software. Stallman defines a program as Free Software if the users have all of the following four kinds of freedom (quoted from Richard Stallman's selected essays [13]):

- Freedom 0: The freedom to run the program, for any purpose.
- Freedom 1: The freedom to study how the program works, and adapt it to your needs. (Access to the source code is a precondition for this.)
- Freedom 2: The freedom to redistribute copies so you can help your neighbour.
- Freedom 3: The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. (Access to the source code is a precondition for this.)

Stallman started the Free Software Foundation and designed the Free Software conditions because he wanted to develop an operating system that was free to use by anyone, and free to change to adapt to the user's needs. He used the UNIX operating system as a model and started developing parts of the new operating system such as GNU Emacs that soon became popular. By 1990 the GNU system was almost complete but the kernel was yet to be developed, and the work of implementing a kernel started. At the same time, in 1991, Linus Thorvald developed a Unix-compatible kernel called Linux, and the GNU Project decided to integrate that kernel in the operating system instead of continuing to develop its own. In 1992 the complete free operating system was finished, called GNU/Linux to express the composition of the GNU system with Linux as the kernel [13].

To ensure that the users of Free Software withhold with the four freedom principles, and that software developed as Free Software remained to be free if it is modified, Stallman developed the GNU General Public Licence (GPL). GPL does not restrict users to not make money out

of developing or integrating Free Software into its proprietary software, but ensures that the freedom principles are withheld.

2.3.2 Open Source software

In 1998 a group of people from the Free Software community started a new movement called Open Source, which is based on the foundations of the Free Software movement but with a slightly different philosophy with a more pragmatic view. The Open Source Initiative (OSI) was founded when the movement started up, and is a non-profit organization dedicated to promoting Open Source Software (OSS). OSI has not defined any software licence of their own, but approves software licenses as Open Source if they meet the conditions of the Open Source Definition. OSI has registered a trademark, OSI Certified, which approved licenses can use to indicate that their software is Open Source.

OSI defined an Open Source Definition, based on the Debian Free Software Guidelines. Under the Open Source Definition the licenses must meet ten conditions to be considered as an open source license [14];

1. Free redistribution: the software can be freely sold or given away.
2. Source code: the program must include the source code or make it freely obtainable.
3. Derived works: redistribution or modifications must be allowed.
4. Integrity of the author's source code: licenses may require modifications to be distributed as patches and under a different name and version number than the original program.
5. No discrimination against persons or groups: free for everyone to use.
6. No discrimination against fields of endeavour: free to be used in any field, also commercial.
7. Distribution of license: the rights attached to the program must apply to all whom the program is redistributed without the need for execution of an additional licence.
8. License must not be specific to a product: the program cannot be licensed only as part of a larger distribution.
9. Licence must not restrict other software: no restrictions on other software distributed along with the licensed software.
10. License must be technology-neutral: no provision of the license may be predicated on any individual technology or style of interface.

2.3.3 The difference between Free and Open Source Software

The Free Software movement and the Open Source movement are today separate movements with different view and goals that agree on the practical recommendations, but disagree on the basic principles. The Open Source movement has a little looser definition of what free software should be, and the source code is freely and public available but the license can restrict what the user is allowed to do with the code. With Free Software the users always have the ability to modify the source code in any way they like.

The Free Software movement is of the opinion that there should be no proprietary software, because it is a way of dominating the user's way of using the software and it takes their freedom away. The Open Source movement on the other hand doesn't mind that there is coexistence between Open Source Software and proprietary software. They look at Open Source software as an economic and technical advantage that can let the users develop software in a better manner, and guarantee the participation of programmers for creation and modification of programs. Almost all Free Software is also Open Source software, but not all of the Open Source software is Free Software.

Free Open Source Software – FOSS – (sometimes also referred to as FLOSS meaning Free/Libre/Open-Source Software) means Open Source Software that is freely available and free of charge. FOSS is a joint term that is often used when referring the Free Software and Open Source communities as a whole without differentiating between the terms and the philosophies.

2.3.4 FOSS in developing countries

The use of proprietary software in developing countries has over the last decades been the normal, but after the entry of FOSS in the software market, several developing countries, e.g. South Africa, have developed strategies for the use of FOSS in households and small businesses as well as in the government. Developing countries have a lot to gain in using FOSS, not only economical but also in the social area by developing a community of local software industry with skilled people.

The biggest advantage for developing countries in using FOSS is without doubt the cost savings. The money that was used for paying license fees and support for proprietary software can instead be used for developing skills and local capacity. The South African government has stated that the foreign currency savings of using FOSS is their main reason promoting the use of it, and the Taiwanese government has estimated that they can save nearly $300 million in royalty payments. When using a proprietary software product the developing countries are dependant on a single provider for support and maintenance, but since FOSS licenses its source code to be available to all, a number of companies can compete to provide the same services at a low cost. The use of FOSS can also contribute to supporting local software industries, if the developing countries choose to use companies inside the country's boundaries instead of being dependant on foreign software providers for the needed support.[15]

2.4 District Health Information Software

DHIS is, as described earlier, a health information system for collection, validation, analysing and reporting of health data to improve the health services in a country. The development process of DHIS has gone through several phases, and feedback from its users has always been important. With the first prototype, participatory prototyping was used to capture the user's needs and requirements. After the release of the first version of the software, informal mechanisms for reporting bugs and requesting new functionality – tightly integrated with user support – proved to be a popular way of encouraging users to provide feedback to the development team[7]. One of the reasons for DHIS' success is that through the participatory prototyping and the feedback from its users, the software has been developed to support the processes already present in the health care facilities and the health personnel's needs for an easy way of collecting and registering health data.

HISP is fully committed to the Open Source Initiative's (OSI) principles for sharing and distribution of software, and DHIS is developed as a *free and open source software* – system, which means that the system is available for everybody that wishes to use it, and the system can be modified by anyone to fit the individual user's needs. HISP retains the copyright to DHIS in order to ensure the OSI principles, and to ensure optimum innovation and future development. Microsoft Office 97 was standard software for the potential users when the development of DHIS started in 1997, and for that reason the application is developed for use

on this platform in version 1.3. The database in DHIS is implemented in Microsoft Access, Microsoft Excel is used for analyses of health data, Microsoft Word is used for preparation of reports and forms, and the graphical user interface (GUI) is developed with Visual Basic.

Even though the system is developed under the Open Source principles, each installation of the system will include costs of approximately 400-500$ for the components from Microsoft (operating system and Office package). DHIS runs on the majority of Microsoft's operating system and Office pack combinations, due to the economy in the developing countries that doesn't allow regular updates of the software[7].

DHIS has been made available to all on CD, and is free of charge like most OSSs are. DHIS is distributed with a Lesser General Public License (LGPL, explained in chapter 3), which means that the users are free to develop add-ons or derived products which not necessarily need to be distributed as Open Source. The decision to license the health information system under LGPL was made because potential users, both within the public health sector and within academia, weren't fully familiar and comfortable with the principles of Open Source, and HISP didn't want the use of GPL to become a barrier towards widespread use of DHIS[16]. All users of DHIS are granted a non-exclusive license to the software, which involves that the users can redistribute the software as they wish as long as the original license statement is included. The next chapter will include a more in-depth description of FOSS licenses.

DHIS can be used at different levels in a health information system, and the requests for improvements or additional functionality can be many. HISP doesn't have the capacity to develop and support all of the new modules the users wants integrated in the health information system. HISP logs all requests from users, but have to prioritise the functionality that the majority demands. Since DHIS is OSS the users have the ability to develop and implement desired improvements to the health information system themselves. Making improvements available as OSS contributes to maintain a network that collaborates about development and improvements of DHIS. The improvements can then be adapted by users that have an equivalent requirement.

Health care facilities and hospitals have the responsibility of exporting the collected data to health managers or planners at the level above itself in the health system hierarchy. Export or import from/to DHIS in version 1.3 is done via comma-delimited text files, either in ASCII or Unicode format. The health data are sent by e-mail or on a floppy disk by ordinary mail, since many primary health care facilities don't have access to the internet.

2.5 DHIS 1.3

DHIS 1.3 is a stable version of the health information system, and is widely used in some of the countries HISP has established teams. Which data elements to collect by using DHIS, can be configured by the user to suit the local use of data. The user also has the ability of defining customized validation-rules that can be added to the data elements to support better quality data. Validation rules can be simple minimum and maximum values for the data, or more specific ones – such as that no more vaccinations can be given than in stock. [7]

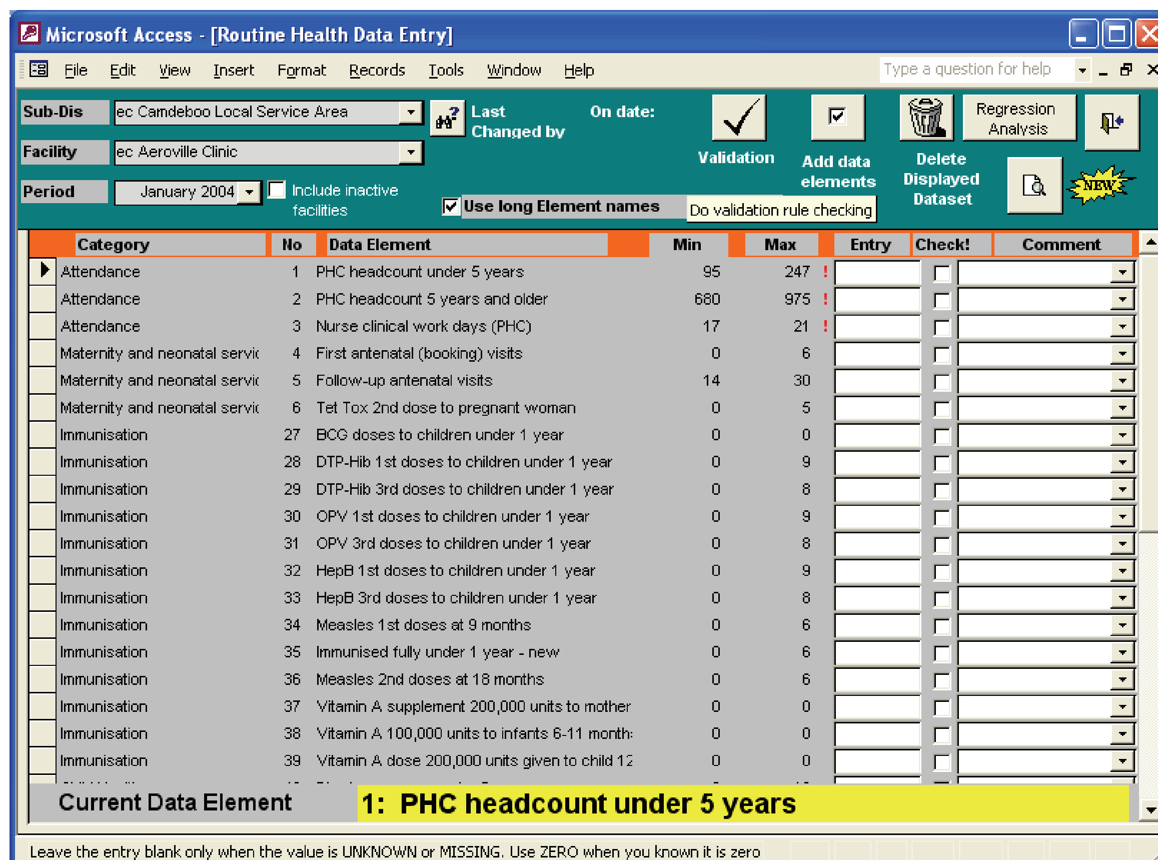


Figure 2.3 A screen capture of the form for entering data in the monthly routine data module. The user selects the sub-district and the facility the data is to be registered for, and the data elements specified for collection in that region is displayed. If the data entry violates the validation rules for a data element (minimum and maximum values are displayed in the form), the user is notified and can correct the entered data or write a comment for why the data is higher/lower than the rules allow.

DHIS consists of two main parts: a front end and a back end. The front end consists of components that give the users the ability to collect, manipulate and analyse data. The backend constitute the health data. DHIS consists of four core modules in the front end [8];

- **A routine monthly data module** – data entry (monthly, quarterly, annually and on an ad hoc basis), verification, analysis and production of standard reports for use by physical healthcare facilities and hospital services.
- **A quarterly tuberculosis (TB) module** – quarterly TB data entry and standard reports (developed for use in South Africa only).
- **A report generator module** – generating and accessing reports on the data entered into the two modules above.
- **A client satisfactions survey module** – capturing and analysing of patient satisfaction surveys (i.e. exit interviews with patients).

Each province normally has at least two database files that constitute the back end. One database file is used for the primary healthcare data and the TB data, and the other one for hospital service data. The primary health care data and data about hospital services are entered by using the routine monthly data module and the TB data by using the quarterly TB module. DHIS also stores the data elements defined in the nation's EDS in the two database files, along with the organizational structure and infrastructure that includes all the health facilities linked to the organizational hierarchy (reflects the health system hierarchy). The complexity of the architecture in version 1.3 is shown in figure 2.4.

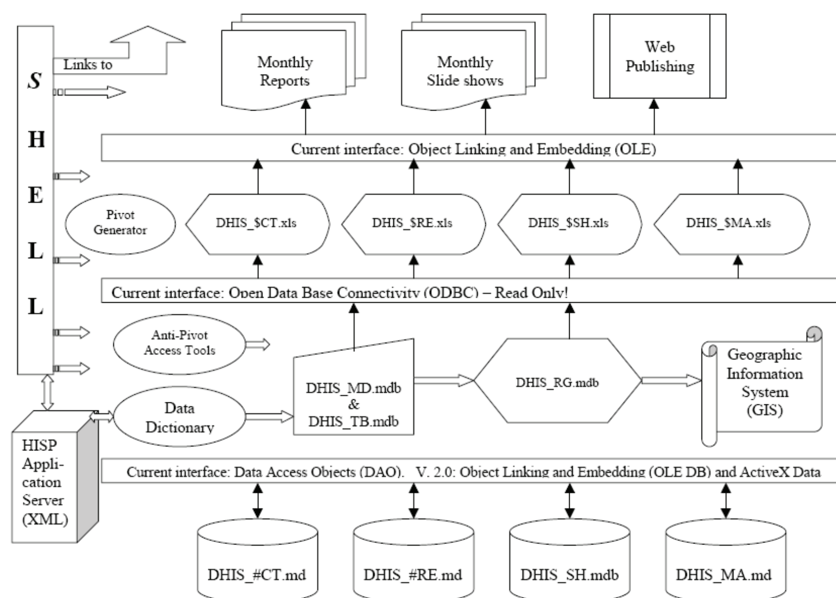


Figure 2.4 DHIS application structure [17].

DHIS has several methods for analysing and presenting the collected data. Data can be automatically processed and exported to Microsoft Excel Pivot table files that the user can generate automatically by using the Pivot generation tool. Pivot tables are Excel spreadsheets that make it easy for users to cross tabulate, filter and graph data. DHIS also gives the user the possibility of generating reports and temporary data tables consisting of elements from different database files (a data mart file) in the report generator module. The report generator assists the user in presenting the data in a way so that they easily can be analysed, and the user can also choose to export the data to GIS[4] or other similar software. Analysing and manipulation of data in the database files does not necessarily have to be done by using DHIS' tools, it can also be done by using the tools available in Microsoft Access.

Prototypes of other data analysing tools are also available with DHIS 1.3. Data Dictionary is a web browser based data dictionary (build using Active Server Pages – ASP) that stores the official names and precise definitions for all data elements included in national and provincial EDS, and will in the future also contain an overview of indicators. It has been made plan for the National Health Department to take over the responsibility of maintaining the information displayed by Data Dictionary. The Web Shell module is web based, just like the Data Dictionary, and gives the users access to all the DHIS modules. In the future, Web Shell will contain information and links to DHIS updates, health information, Windows updates and HISP online. Both Data Dictionary and Web Shell run on HISP Application Server which is a local web server.

In DHIS version 1.3 the routine monthly data, TB and the report generator modules are multilingual, and is translated to English, Portuguese, Norwegian, Swahili, Telugu, Kannada, Hindi, Spanish, Russian, Chinese, Arabian and Mongolian. Other language translations are under development. HISPML, HISP Multi-language Library, is the database that contains all the text-strings with the different language translations for use in DHIS.

Hardware requirements for running DHIS

The minimum hardware requirement for running DHIS is a PC that can run Access 97, i.e. a minimum of 16 MB RAM and 200 MB available space on the hard disk. If it is desirable to run the web applications and HISP Application Server locally, it is recommended with

[4] GIS = Geographic Information Systems

minimum 32 MB RAM. [18] There exists a lot of out-dated PC-equipment in developing countries as a result of little funding, and the requirements for running DHIS are for that reason relatively low.

Since much of the equipment is ready for replacement, it is developed recommended requirements for PCs that shall run DHIS, at both national/provincial-level and at district-level [19]:

- At district level it is recommended to use one or two PC's with a 2.0 GHz Pentium 4 processor with 512 K cache, 512 MB RAM, and a hard disk with 40-60 GB memory.
- At national/provincial level it is recommended with at technical workstation that can handle the large amounts of data that exists at this level. It is recommended to use one or more PC's with 2.4 – 2.8 GHz Pentium 4 processor with 512 K cache, 1024 MB RAM and two hard disks with 60-120 GB memory. At this level it can also be necessary with a larger sever that can share software and data between 10-100 users.

In addition to PC's that should be available at all levels in a health information system, a printer and scanner should also be available, and at national/provincial-level it can also be desirable with a digital projector for use in presentations.

Software requirements for running DHIS

DHIS 1.3 runs on all of the Windows platforms, with the exception of Windows 95A and NT 3.5x, assumed that the necessary Service Packs from Windows are installed. All users of DHIS must have installed Internet Explorer version 5, 5.5 or 6. DHIS also needs some Microsoft system files to be able to run (automatically installed together with DHIS)[18];

- Jet 4.0 – a component database engine.
- Microsoft's Data Access Objects (DAO) 3.6 – a set of objects that enables OLE[5] Automation clients to access and manipulate data in local and remote databases, in addition to manage databases, their objects and structure. DAO is used by e.g. Access and Visual Basic.
- Visual Basic 6 Runtime – necessary for execution of programs developed with Visual Basic 6.0.

[5] OLE = Object Linking and Embedding developed by Microsoft. Allows objects from one application to be embedded within another.

- Microsoft Data Access Components (MDAC) 2.6 – allows data driven applications to communicate across the Web or over a local network, includes ADO, ODBC and OLE DB.

DHIS also requires some additional applications to function optimum (distributed with DHIS);

- XML Parser.
- Snapshot Viewer 9.0 – necessary for viewing Access-reports or sending them by e-mail (free utility from Microsoft).
- Adobe Acrobat Reader – necessary for reading available material (free tool from Adobe).
- ArcExplorer 2.0 – necessary for use of the GIS interface (free desktop GIS viewer from ESRI).
- WinZip 8.1 – shareware compression utility, needed to e-mail or move larger files.

DHIS uses components from Microsoft Office and it is therefore required to have installed Microsoft Office 97/2000/XP, with the essential Service Releases. The database application in DHIS is implemented in Microsoft Access, but if Access or Office for some reason isn't available at installation, an Access Runtime version can be used. The runtime version provides a limited functionality consisting of a few of the Access components that are necessary for DHIS to function. DHIS users that use the Access Runtime version will not have the same abilities to access the data files, queries or forms, but can test out the main functionality of the health information system. [20]

Disadvantages with DHIS version 1.3 and possible solutions

DHIS version 1.3 has received critics for being out-dated, unprofessional and for having a poorly designed architecture. These criticisms speeded up the request for improving the health information system and update the technology it uses, and in 2003 the work of developing a new version of DHIS, DHIS 2.0, started.

The main criticism of version 1.3 is that the system isn't developed on a three layer architecture and that it isn't web-based. With a three layer architecture the user interface, business logic and the storage of data are separated from each other in different layers. DHIS is only separated into two layers – the user interface and the database management, with the business logic spread in those two layers. A change in the business logic forces a new

deployment of the whole system. The system is based on a decentralized concept, i.e. that the application is standalone and restricted to desktop solutions and doesn't interact with a database or other computers over a network. The database files are situated on the same machine as the user interface, and the data received at the higher levels in a health information system from different users at the lower levels, have to be manually imported from either e-mails or floppy disks sent by ordinary mail.

A modern three layer architecture is based on the use of web servers with a central database, separated business logic and clients that interact over a network. In South Africa and in several developing countries that use DHIS, the networks and telephone lines are in many areas poor, which makes it difficult for districts and provinces to connect to a central database. Even if it is desirable to make DHIS web-enabled (which is the case, and the work has started), the solution still has to make it possible for use on standalone clients where access to a network is unstable or not present at all.

DHIS database

The database engine used, Microsoft Access, is criticised for not being able to handle the large amount of data that can be registered by using DHIS. Access was originally chosen since it was already purchased and in use in South Africa, but has with the increased use of DHIS proved to be slow for handling the large amounts of health data. The formal storage capacity to Access is maximum 2 GB, but the practical limit is about 20% of that [8]. The DHIS is also criticised for not upholding the data's integrity, since changes done locally will not automatically be updated in the databases at the higher levels in the health system without the use of a central database. The latter is a consequence of DHIS being based on a decentralized concept and will continue to be a weakness of the system as long as not all districts and provinces has a stable access to a network, and it will not be improved by using a different database management system (DBMS). Nevertheless, the storage limits by using Access, and the limits of only 10-15 concurrent users if it is used in a network, makes it necessary to consider other DBMSs that can be part of DHIS on both a single-user system and multi-user system [8]. The most ideal solution would be to have a system that is independent of a single DBMS.

The internal structure of the database contains a large amount of tables (at least 38), which can make it difficult to maintain. There are created a set of tables for each of the core modules in the system and a lot of them coincide. The tables should have been designed more generally to be used by all of the modules, and decreasing the number of tables in the database. The database in a new version of DHIS should be designed to be more normalized.

Graphical user interface

The user interface in DHIS is criticised for not being very user-friendly with a poor and outdated design. The programming language used for development of the graphical user interface (GUI) is Visual Basic (VB) from Microsoft. GUI's programmed in VB gives an appearance that is already known for users of Microsoft Windows, which is the case for most of the users - at least in South Africa where the development of DHIS started. Since DHIS is Open Source Software and is used on PC's with Access, which uses VB as a programming language, it is possible to adapt the system to an individual user's request without the need for external tools. Users that have received training in using DHIS, like nurses and health personnel, has proven to handle the system well. Nevertheless it is still a lot that can and should be improved in the present GUI without having to remove the functionality the users are familiar with.

The design in the core modules of DHIS appear as poorly designed, because it is little intuitively and the guiding elements aren't used consistently. Each form also contains too much functionality, which can be an advantage for experienced users since a lot of functionality is available from one form. But too much functionality present in one form will probably be a disadvantage for new users or users that doesn't use the system on a daily basis, since it is difficult to get an overview and it can be hard to understand the purpose of all the functionality.

The Client Satisfaction Module, which was developed some time after the other core modules in DHIS, has a completely different GUI than the others. This shows that there haven't been any consistency with regard to the design of the GUI's displayed to the users, which can make it more time-consuming and difficult for users to understand the functionality present in the different modules. When development of an application is done with collaboration over a network like the HISP network with teams in different countries, a set of GUI design principles should be defined. The principles can be used when modifications or further

development of the system is done to ensure that the design is consistent. Appropriate placement of explanatory text and easily recognizable elements will also contribute to making the system more user-friendly for non-computer-competent users, that only use DHIS a couple of times a year (e.g. with entering of data quarterly or annual).

The new version of DHIS should also be made more user-friendly by including all of the different modules and tools into one single application. In DHIS version 1.3, various reports are generated by using external tools like Excel and Word. It is possible to send a report on e-mail directly from an Access-application, but only if Outlook Express is used and the receiver of the e-mail uses Snapshot Viewer for viewing the report. DHIS is quite a large system that also requires the use of other tools, and the user has to be familiar with the necessary Office-products to fully utilize the system. Including the different parts of DHIS into one application would simplify the use of the system, since the user would only have to deal with one application. In the prototype of Web Shell, an approach for gathering all the different modules available from one GUI is shown, making it more efficient for the users to employ the system.

DHIS user manual

The DHIS user manual is criticised for being poorly written and not usable for new users that wants to know how to use the system. To improve the documentation the manual is undergoing rewriting and an online help function will be included. The version of the manual we use as a basis for our discussion was distributed together with DHIS version 1.3.0.44. That version of the manual is defective after our opinion, and it isn't adjusted to the version of DHIS it is distributed together with, since some of the graphical elements described have been changed and no longer functions the way they are described.

The manual is difficult to follow and lacks an appropriate division of its content into chapters, and can hardly be used as a reference book. There has been taken no consideration to how health personnel employ the system, and there aren't included any example scenarios that describes how the system can be used. The manual consists of separate descriptions of each registration form and the tools, which appears as a description of the contents of the form. There are also three different ways of describing the functionality in a form; stepwise descriptions, point lists, and rattling off. Consistent use of graphical elements in the manual would make it easier to comprehend the described functionality. The manual is also of little

help for more advanced users that only need guidance in some areas. A table of contents is provided, but an alphabetical index describing where to find the necessary information of how to solve a problem is lacking.

HISP offers new users of DHIS courses and training, but if they encounter a problem or needs guidance when they start using the system the only documentation they have available is the manual. Rewriting of the manual to make it more structured and educational is therefore necessary.

Reporting of health data

Reports and health data are exchanged by sending them by e-mail or on a floppy disk by ordinary mail, and the receiver are required to have the necessary tools for opening and viewing the file on the receiver's computer. Word and Excel are closed systems and that makes it difficult to export data outside a Microsoft environment, even though Access includes several export tools[7]. To meet the demands of receivers that use other platforms than Microsoft, the system should start using a standardized format of the exported data. The text files generated from the data files are in either ASCII or Unicode format, but do not have a defined character set. The sender is therefore dependent on the receiver having the proper tools for interpreting the characters correctly. In the further development of DHIS it is necessary to evaluate different standards of exporting data that can be used regardless of the platforms being used.

Platform-dependency

DHIS in version 1.3 is not platform-independent, and there are costs attached to the use of Microsoft-components for each installation of the system. The costs by choosing to use an Open Source or Free Software platform like Linux instead of the proprietary Microsoft would decrease the costs considerably - if not eliminating them, and would be preferable for use in developing countries. We also described earlier that many countries are developing strategies for increasing the use of FOSS, and DHIS should therefore aim at platform-independency, or at least adaptation to several platforms than just Microsoft. The developing tools used for developing the new version of DHIS must be available on the different platforms and in the different countries, so that modification and development of the system can be done by the individual user.

Problem areas for further development

As a part of the development of DHIS 2 it is necessary to consider alternative and more efficient implementations of the system. Newer technologies may open for a more suitable development and following adjustment of the system from what it is today.

One of DHIS' problem areas is to find alternative technologies that can contribute to make the system more "up-to-date" and efficient. The alternative technologies have to comprise the storing of data, the implementation of business logic and the user interface. How to implement a three layer architecture in the system, and what design principles that are needed, have to be taken into consideration. Different DBMSs that can be used in a multiple user system, but also maintains the possibility of a single user system, have to be taken into consideration. As a three layer architecture indicates, the business logic has to be removed from the application on the user's client, and joint in one layer. This implies changing today's thick client into a thin client that interacts with the other layers over a network. In cases where the system has to operate in single user mode, the three layers have to be gathered and function as a thick client. The user interface should be improved and made more user-friendly, but at the same time the system's functionality has to remain to make the interface recognizable for the users. When developing a thin client that interacts with a web server, one should consider concepts for displaying the user interface in a web browser.

DHIS is Free and Open Source and this has to remain to make the users able to develop the system in the future. This means that the source code has to be a part of the system distribution. The development tools that are being used need to be free of cost – or at least available at low cost – to make development possible in the countries where the system is being used. Because the countries in question are developing countries, the resources present are limited. Both for the system itself and the tools for developing it, have to be compatible with the equipment and platforms the countries already have invested in. DHIS has some minimum requirements for hardware and software, but these requirements do not exceed a normal desktop standard.

To make the development and modification of the system easier, it might be a good idea to have a look at the possibilities to make the adding of components to the system easier. Newer technologies may give the possibilities for a more component based system, where each

component will have a confined responsibility that can be utilized by other components. This will make modification of parts of the system easier, with minimal influence on the rest of the system, and also make the use of other user's or supplier's components easier.

A considerable improvement to the system (and also a considerable amount of work) would be to adapt it to more than the existing platform, i.e. Linux. Defined standards – like data transmission standards – should be used to make the system as independent of specific platforms or tools as possible.

To shortly summarize the problem areas for further development of DHIS, the following needs to be considered:

- Three layer architecture with a web browser based GUI and a more component based system.
- Free and Open Source Software and developing tools.
- Database and platform independency.

The challenge is to carry the system forward and into a new technological era and at the same time adjusting it to the limited resources available in developing countries. It is important to look at the system in regard to the context it was originally developed, and to find technologies that can contribute to a more efficient development and use of the system.

2.6 DHIS 1.4

Since version 1.3 of DHIS was developed, the HISP network has focused on developing a new version of DHIS (version 2.0) to remove the disadvantages by using the system, and to solve many of the problem areas described above. DHIS 1.4 is the last version where the core modules predominantly rely on Microsoft Office and Visual Basic, and is regarded as a "bridging" version over to version 2.0 that is described in the next chapter. DHIS 1.4 only supports Office 2000/2002/2003 and later.

DHIS 1.4 contains the same features as version 1.3, but includes some new ones. One of the new features is the ability to expand the hierarchy from five to six levels, as we have described earlier. Other new features include support for user-defined data sets for more customized data entry, user-defined grouping of data elements, indicators and organizational

units, and XML-based integrated export/import of data and all resource tables. Version 1.4 also gives the user the ability to use other SQL-compliant DBMS than Access, like MySQL, PostgreSQL, SQL Server and Oracle. The database in this version has a more normalized internal structure, which results in a smaller database size. [21]

2.7 Summary

We have in this chapter presented HISP which is a research and development project that initially involved collaboration between universities in Norway and South Africa, but has in the recent years evolved to an international network with teams in several developing countries. HISP has focused on improving the health systems used in local health institutions in developing countries, and giving them the ability to analyse health data in a local context. To support this, HISP has developed DHIS for collection, reporting and analysing of health data. Currently HISP is working on improving DHIS and has started the development of a new version, described in the next chapter.

Chapter 3

Theoretical basis for the study

The starting point for our investigation is the development of DHIS 2.0, HISP's choice of a free and open source technology, and the process of transporting DHIS to a new technological platform. As we have explained earlier we chose not to be a part of the DHIS development process, but instead investigate the challenges they can encounter by using free and open source software to develop a web application, and how they can design for change.

The main focus we have had under the work of this thesis is evaluation of free and open source software to use under the development of Joly, and how we can design for change, i.e. how the application can be designed to easily adapt to changes that will occur in the system's operating time. This chapter covers the theoretical basis for our work in the evaluation process and the design of Joly, in addition to a description of the development of DHIS version 2.0 and its requirements.

3.1 DHIS 2.0

The development of DHIS 2.0 started in 2003 to address the disadvantages and problem areas of version 1.3 of the health system, as described in the previous chapter. The software development process is a global collaboration between students, researchers and experienced developers that participate in HISP teams in Norway, South Africa, India and Vietnam. The rational for developing a new version of DHIS, is to develop a fully open source, platform-independent and web-enabled version of the health information system.

It is important for HISP that DHIS continues to be a free and open source system since its users already are familiar with the benefits it has compared to proprietary software, and

because it contributes to savings of the costs a developing country spends on health care. As we explained earlier, the use of Free and Open Source Software (FOSS) in developing countries is increasing, and many governments have developed strategies that encourage the use of it. For these reasons it is important that DHIS continues to be FOSS - to ensure that it will both be widespread in use and receive support by the governments in the developing countries that employ DHIS.

The collaboration between the development teams in the different countries is mainly done by the use of a wiki[1] website (discussion and information) and a Subversion[2] repository (source code), but also by students and researchers visiting teams in other countries. JIRA[3] is used to manage the project and support the development process, and each module in the DHIS design has its own project location at JIRA which is used to define tasks to be done, requirements of the module and bugs to be fixed. HISP does not have a forum for its developers yet, but they use mailing lists to discuss problems or distribute information among its registered developers.

DHIS 2.0 has two project leaders appointed by the HISP project's leadership at the University of Oslo. The project leaders and the projects core developers (lined-up in figure 3.1) are the main decision-makers in the project, but most decisions are open to a public debate for the rest of the projects developers.[22]

Project leaders from HISP:
Knut Staring (Oslo, Norway) – faculty member with research scholarship
Ola Hodne Titlestad (Oslo, Norway) – faculty member with research scholarship
Core developers:
Kristian Nordal (Oslo, Norway) – master degree student
Torgeir Lorange Østby (Oslo, Norway) – master degree student

Figure 3.1 Organization of the DHIS 2.0 leadership at the moment.

[1] Wiki is an online collaboration model and tool that allows its users to edit the contents of the web-pages through a browser.
[2] Subversion (SVN) is an Open Source version control system.
[3] JIRA is a J2EE-based issue-tracking and project management application developed to make the development process in a team easier.

3.1.1 Requirements

DHIS version 2.0 can be regarded as a hybrid application suite, which denotes a combination of typical properties of a flexible transaction database, the properties of a data warehouse and some properties of a communication tool. The application should mainly meet the following user needs (quoted from [23]):

1. The need to rapidly design data collection tools for a variety of purposes.
2. The need to efficiently capture or import/collate this variety of data in an "integrated" manner, then to monitor the processing and flow of this data, and finally to communicate with various stakeholders in the process.
3. The need to analyse relatively large and complicated data sets quickly and efficiently.

In addition to the main requirements for the new version of DHIS, there are more detailed user requirements that include the need for a standard data entry module, indicator support, data validation and aggregation, import/export of health data, a standard report module and internationalization. In addition there need to be support for the use of more flexible or advanced modules that can be selected from the DHIS 2.0 suite, or supplemented by locally developed modules. The latter gives the users a way of cater the need of specific local requirements that are not covered in the global DHIS distribution. [23]

The requirements for technology are described below.

3.1.2 Technology

HISP has defined six requirements that their choice of technology should cater, which coincide with the requirements we concluded in the last chapter that a new version of DHIS should meet. The OSS chosen should meet the following requirements (quoted from [23]):

1. Fully Open Source Software.
2. Platform independency (operating system and database).
3. Web-enabling.
4. Modular architecture.
5. Support all existing functionality in DHIS 1.4.
6. The need to support both networked and standalone user environments.

HISP chose to re-implement the Microsoft Office based DHIS on an open source Java platform. The choice of using Java was made because it provides platform-independency since the Java Runtime Environment (JRE) supports all operative systems, and because it is an open source platform. Microsoft's .NET framework could have been a choice for HISP, but since it only supports Windows platforms, and since there are costs combined with the use of their development tool (Visual Studio), the use of .NET would violate requirements 1 and 2 above. For the .NET framework to be platform-independent, an implementation of Common Language Runtime (CLR, the equivalent to Java JRE) is required, which is the runtime environment in which .NET applications run. Until recently there has only been one implementation, which is Microsofts CLR, and it only runs on Microsoft's Windows operation system.

At the time HISP chose to use Java, an open source alternative to .NET was released – called Mono. Mono provides software for development and running of .NET clients and server applications on Linux, Solaris, Mac OS X, Windows and UNIX. Mono could have been an alternative for using Java since it meets HISP's technology requirements, but because it doesn't include all of .NET's functionality yet, the more mature Java is a better choice for a large system as DHIS.

The requirement of only using FOSS doesn't necessarily narrow down the choices of technology since there are many satisfactorily FOSS products, but it is important to evaluate the software, as we emphasize later in this chapter. Evaluation implies choosing an FOSS that is mature, has an active supporting community and a license that doesn't restrict the development or redistribution.

We will only very briefly describe the technologies used by DHIS, because it will in the following chapters be a more comprehensive description of the ones we have evaluated and used under the development of Joly.

HISP has chosen to use the following lightweight Java frameworks and tools (figure 3.2 illustrates how the architecture of DHIS and the technologies chosen play together, to support the technology requirements) [23]:

- **The Spring framework**
  A layered Java/J2EE application framework with a lightweight container that takes care of assembling of the DHIS modules, and the communication between them.
- **Hibernate**
  An object-relational mapping framework that provides database-independency for relational database management servers.
- **WebWork**
  Java web application development framework that is used to simplify the development process.
- **Velocity**
  A Java-based template engine that together with WebWork supports reuse and decoupling of the applications presentation layer from its business layer.
- **Maven**
  A build tool that supports a flexible configuration of a modular application, and is used in DHIS to couple its modules together and build a running release of the system.

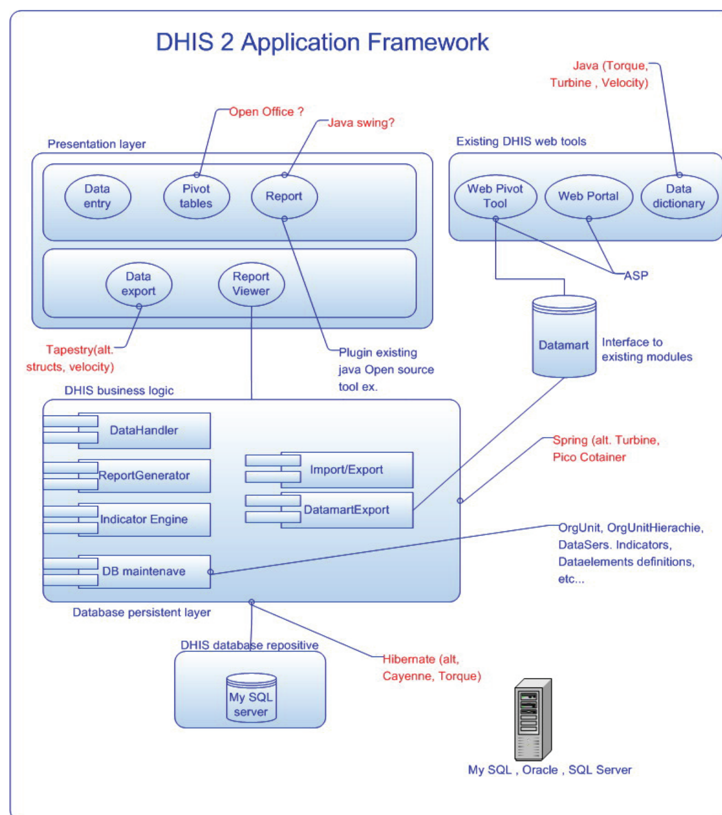


Figure 3.2 DHIS 2 Application Framework. Spring provides intra- and inter-module communication by injecting the communication links into the Java implantations. The communication links configuration is specified in XML in each module, which makes each module pluggable and replaceable. [24]

HISP has developed a set of best practises and development tips for its developers, as we advised in the previous chapter. The best practises give the developers a few tips on how to structure the code, design the GUI, and write web pages that complies with the HTML 4.01 Strict Specification[4]. The best practises advised could after our opinion have been more comprehensive and detailed, to ensure consistency and avoid messy code that could be difficult to understand and maintain by new developers.

3.1.3 Architecture and design

A challenge for HISP is to design DHIS to meet the requirement of supporting both networked and standalone users. To support this requirement they have done the following [23];

- The architecture, as shown in figure 3.2, provides a layered architecture. The layered architecture enables the use of the data and business layer by both a desktop and a web presentation module.
- The application is based on lightweight Java frameworks that only need small and simple servlet containers (e.g. Jetty or Tomcat) to run either locally by using a web browser on a standalone system without a network, or as a web application running the application.

The design of DHIS 2.0 is divided into 20 different modules which each has a limited functionality. 14 of the 20 modules are under development, in addition to a *Plug-in* module that is not shown in the overview in figure 3.3. Having a design that constitutes of a number of modules with limited functionality, instead of having all of the functionality in one place as was the case with version 1.3, makes it easier to maintain and add possibly new modules without having to redesign the whole system - as we discussed in the previous chapter.

The downside of modular design is that code can be repeated in the modules, and if the interfaces of the modules aren't clearly defined there is a risk of having to make adjustments in other modules when changes occur. However, modular design and architecture is popular in designing systems today, because they provide the flexibility to customise products for individuals and to upgrade them when better components come along [25]. In addition, modular design gives the developers the possibility of developing different parts of the system

[4] HTML 4.01 Strict Specification is developed by W3C (World Wide Web Consortium) that creates standards and specifications for the World Wide Web.

in parallel, and thus shortening the development time. These advantages of modular design would make the development process easier - and possibly shorter, in addition to making the system easier to maintain and therefore seems like a good solution for DHIS.

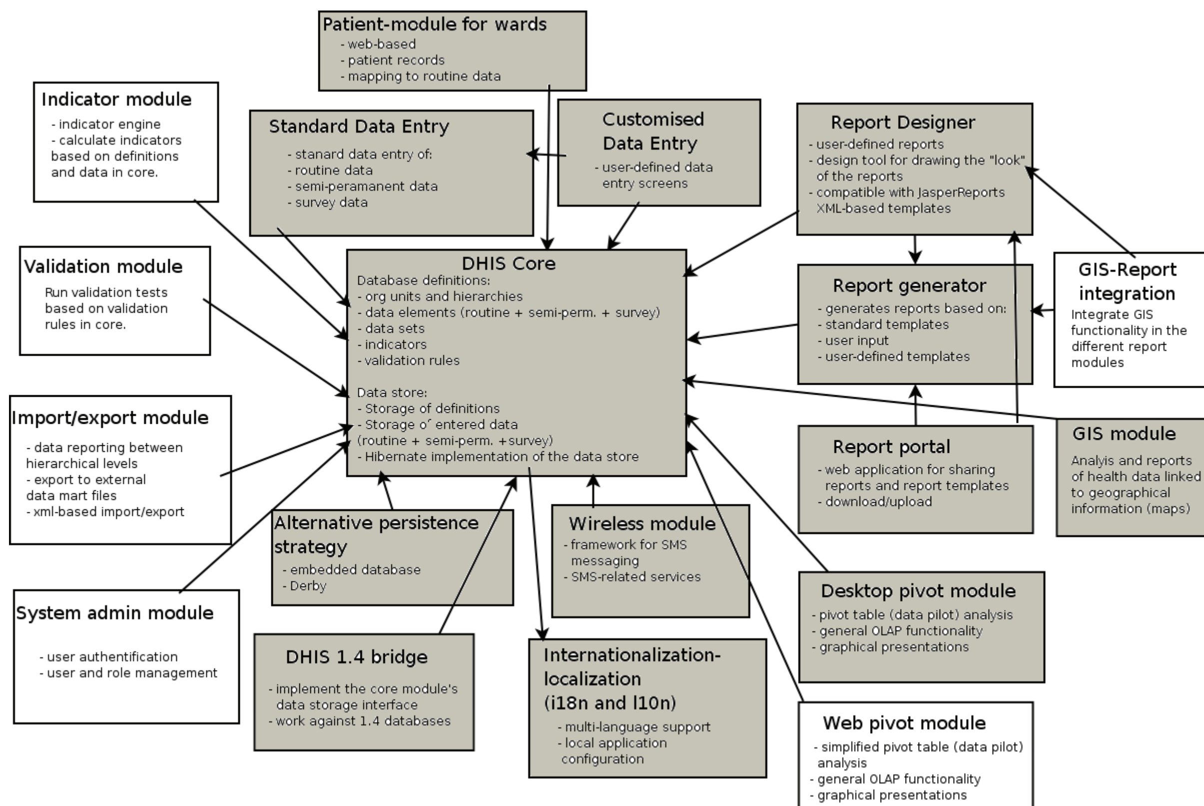


Figure 3.3 Basic overview of the modules in the DHIS 2.0 design (edited version published by [26]) - ■ = modules currently in development, □ = modules not in active development at the moment.

3.1.4 The road ahead for DHIS

At the time of writing, DHIS 2.0 Milestone 3 has been released, and it will be milestone releases each month in the time ahead until the first full release in December 2006. The full release will include a limited functionality, as not all of the modules in the design will have been developed. As we see it, it will take a long time to complete the development of the new version, due to the fact that it is a comprehensive system with a lot of functionality that needs to be in place. After DHIS is fully developed it needs to be distributed and installed, and there will be a need for new training of the health personnel in the use of the system.

A drawback by using many years to develop an application, is the risk of software providers no longer supporting or developing technology that the application rely on, and newer

versions of the technology could be incompatible with other technologies used. When the application relies on FOSS technology, the source code is available to all its users, and they can in reality continue to make necessary improvements or make changes in the technology to adapt it to their needs, but this can be a time consuming task. DHIS' architecture is modular, which makes it easier to adapt it to changes without having an impact on the system as a whole, but we fear that DHIS will meet many challenges on its way to a complete version of the system.

3.2 Free/Open Source Software evaluation

The number of FOSS available today is certainly overwhelming. At the time of writing, there are registered 118,612 projects and 1,304,427 users at the FOSS portal SouceForge. There are in addition other FOSS portals, and FOSSs are also available from individual FOSS product web sites. Many of these FOSS projects are however moribund, or haven't made much progress beyond registration of a nifty name and creating a project at one of the portals.

Although the vast majority of FOSS products are available at no cost, there also exist some commercial FOSS distributions. Usually the product is available at no cost as well, but the distribution can offer the product in a more convenient form (e.g. on a CD) or as part of a larger product that offers add-on products or services, like support or training, along with the software [27]. Redhat Linux is an example of a commercial distribution of a FOSS. Commercial distributions are today a common way for users to adopt Linux, because the installation of the operating system is easier since the user doesn't have to locate it on the internet and manually install it.

It can be hard to locate the best product in the jungle of FOSS solutions. FOSS solutions differ from proprietary software in areas like licensing and system development processes, and the maturity of the solutions aren't guaranteed in the same way as proprietary solutions. In proprietary solutions the commercial vendors take responsibility for delivering products of acceptable maturity. It is therefore important to evaluate the properties of a FOSS solution before it is adopted. The process of evaluating a FOSS product can be easier than it is for a proprietary product, since a lot of public information is available, e.g. the source code, that is lacking for proprietary products.

There are many approaches for evaluating FOSS, both processes (the most well-known is perhaps the “Open Source Maturity Model”), tips-lists and models. There is also a model, the Business Readiness Rating model, which is being proposed as a new standard model for rating FOSS that aims at determining whether a FOSS can be considered mature enough to adopt. This model is however only in a public comment period, asking for feedback from the FOSS community to help shaping this standard. It could therefore not be of any help during our assessment and evaluation of FOSS. We see that there is a need for such a rating of the different solutions, as it can give the users a quick and reliable way of evaluating a product. The other processes and models for evaluation of FOSS are essentially the same, and provides the following basic steps for evaluation of a program; identify candidates that has a solution, read existing reviews, briefly compare the program’s attributes to the ones that are required, and then perform an in-depth analyses of the top candidates [28]. Evaluation is a process that can take five minutes, or several months if a mammoth change is considered for a major enterprise.

We have chosen to evaluate the FOSS solutions for the development of Joly after the following criterions;

- Software license
- User community
- Stability and development activity
- Documentation

We will in the following describe the criterions in depth and explain how they differ from proprietary software. We will also describe how we have used these criterions in the evaluation process of finding suitable free and open source software for the development of Joly.

3.2.1 FOSS licenses

Proprietary software has licenses that are unique for each product, but they share a common characteristic in that the intellectual property of the product is reserved and held by the creator of the software. The licenses usually restrict the rights of usage and redistribution of the software, typically restricting what machine/machines the product can be used on, who can use the product, and what right the user might have to pass it on to someone else.

FOSS licenses on the other hand support the rights of usage and encourage redistribution. [27] The licenses are used for ensuring that the product remains open source. There is a variety of FOSS licenses (OSI has approved 58 open source licenses at the time of writing, and there are in addition many more), with different conditions under which a product can be used. Some of the licenses have stricter conditions for use and distribution than others (typically licenses provided by the Free Software Foundation to ensure the freedom principles for its users), and for that reason cannot be combined with other less restrictive FOSS licenses. Developers of FOSS are encouraged to use GPL-compatible licenses since it otherwise can fail to receive enough support from other developers to sustain the project, because many FOSS developers prefer the GPL (GNU\Linux are licensed with GPL)[29]. GPL is however the most restrictive license, and it is important for users to fully understand the licenses conditions and the impact it can have on a resulting product and how it can be redistributed.

Both DHIS (version 2.0) and Joly use FOSS, so we will shortly describe some of the most popular FOSS licenses that software employed by DHIS and Joly is licensed under.

GNU General Public License – GPL

GNU General Public License is the most popular but also the most restrictive license of the FOSS licenses. The license is used to guarantee the users freedom to share and change Free Software. It was developed by the Free Software Foundation, but is also supported by the Open Source Initiative. The license requires that any derivative work that incorporates a GPL licensed product, must itself be licensed as a GPL product, to ensure that the principles of freedom are preserved.

GNU Lesser General Public License – LGPL

The GNU Lesser General Public License is a less restrictive variant of the GPL, and is primarily intended for software libraries but it is also used by some stand-alone applications (e.g. OpenOffice.org). The main difference between GPL and LGPL is that the latter can be used in non (L)GPL products without imposing the condition that the resulting product must be FOSS (e.g. proprietary software). LGPL enables widespread use of useful technologies without imposing any onerous conditions on the user. LGPL is GPL-compatible.

Apache Licence Version 2.0

The Apache Licence is authored by The Apache Software Foundation, and all of their software or projects are licensed according to the terms of this license. The license requires preservation of the copyright notice and disclaimer, but allows for the use and distribution in both FOSS and proprietary software. With the release of version 2.0 of the license, The Apache Software Foundation wanted to make it compatible with other software licenses such as GPL, but the Free Software Foundation are of the opinion that it still isn't GPL-compatible because "it has certain patent termination cases that the GPL does not require" [30].

3.2.2 FOSS software development and community

Eric Raymond, in his essay "The Cathedral and the Bazaar", identifies two different models for FOSS development; the Cathedral model and the Bazaar model. The Cathedral model is the traditional software development model that Raymond compares to building a cathedral; witch relies on relatively small groups of developers that are all associated with a single institution or corporation. The source code is available with each software release as the principals for FOSS require, but between the releases the code developed is only restricted to an exclusive group of programmers. The Cathedral model is also the typical software development model for proprietary software, except for the available code at release.

In the Bazaar model, however, the code is developed over the internet in view of the public that can contribute to the system's development. The Bazaar-style of developing a system first appeared with Linus Thorvald's style of developing Linux; early and often releases, delegate as much as possible, and openness to the point of promiscuity. Raymond thought that "the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches", hence the name the Bazaar model.[31]

The central thesis in Raymond's essay is his proposition that "given enough eyeballs, all bugs are shallow". By using a Bazaar-style of developing a system the users has access to the source code during the development and can contribute to testing, scrutiny and experimentation. In contrast to the Cathedral model, where inordinate amount of time could be spent on discovering bugs since there are few developers that has access to the code, in the Bazaar model of system development the bugs could be discovered at a rapid rate if it has a large community. Early and often releases contribute to cutting down the possible duplication

of effort by the users/programmers working, unknown to each other, to identify or fix the same bug, since the number of users (and hence potential debuggers) that are exposed to the solved problem are reduced in a new release [32].

After Raymond published his thesis, many FOSS projects were convinced that the Bazaar-style of developing software was better than the Cathedral model, and it actually provided the final push for Netcape in 1998 to give away the source for Netscape Communicator which led to the start of the Mozilla project[5]. The Bazaar model of developing systems is today the most common way of developing FOSS, and is one of the advantages of using FOSS compared to proprietary software that uses the Cathedral model of systems development. This is because proprietary software does not receive the same support and contributions from its users during development.

This shows that a community with a large user base which contributes to the development is a criterion for the success of a FOSS project, not just for ensuring the quality of the code, but also for guaranteeing continuous support and further development for some time ahead. Another criterion for the success of FOSS is a large scope and broad use of the application. This is because larger scope projects require more functionality, and providing that functionality is the strength of the Bazaar model. The broader the use of a FOSS, the more likely it also is that potential users need such a product and will contribute to the development process. [33] It is therefore necessary, as it is with licenses and other properties of FOSS, to evaluate the community of a FOSS before adopting it, as we will emphasize in the next part of the chapter.

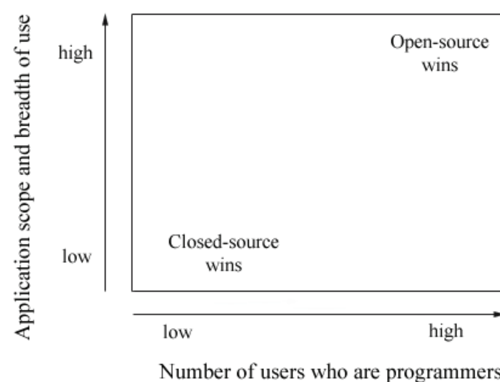


Figure 3.4 Criterions for the success of FOSS over proprietary software [33]

[5] The Mozilla project is the producer and provider of Firefox (web browser) and Thunderbird (e-mail client).

HISP used the Cathedral model under the development of DHIS version 1.3 since it was the common development model at the time the project started up. The system was developed by HISP teams in different countries and was open for everyone, but the source code wasn't available globally, e.g. on the internet, to its users under the development. At the end of the development of a new release, the system was, along with the source code, distributed to its users on CD. The software development process may have been somewhat affected by using the Cathedral model – development of new releases of the system took time, and main feedback from its users had to be done through testing of the system at the health facilities or after a release was distributed.

For the development of a new version of DHIS (version 2.0), HISP has chosen to use the Bazaar-style of system development, and the source code is available for everyone that is interested in contributing to the development process. Frequent releases should be performed so that the system can be tested, and bugs detected and fixed. Some parts of the testing can also be done by interested users downloading the source code, testing it out for them selves, and give feedback, which can reduce the overall developing time.

The system development of Joly has been done in Cathedral-style for the first version, with the only developers being the writers of this thesis. The application is developed on behalf of the Institute of Informatics at the University of Oslo, and it is their choice to decide which system development model they will use for continuous development of Joly.

3.2.3 FOSS stability and development activity

This evaluation criterion consists of two properties of a FOSS for evaluating the maturity and stability of a product; version number and ongoing effort. Proprietary software vendors have the responsibility of releasing stable and mature software, but since FOSS projects usually are developed by using the Bazaar-model, as we stated above, releases are done early and often. It is necessary to evaluate FOSS products to make sure that they have a stable and mature version before they are employed.

When there is an active development community, bugs will be discovered and fixed and patches to the software or a new version will be released. During development, a release normally has a low version number, and a product typically needs to reach its 1.0 before it can

be considered as mature and stable. Mozilla's 0.8 release of its browser Firefox was however mature and stable, so the version number should only be used as a guideline in evaluating a FOSS product. [34]

It is important that there is an ongoing effort to develop the software to fix bugs and meet the user's needs. Software projects that for some time hasn't had any ongoing effort can be in risk of being moribund, therefore it is important to choose a FOSS that has an ongoing development to ensure that the software will continue to be up to date for some time ahead. Active FOSS projects usually have regularly updated web pages and busy development lists, which indicate an ongoing effort to improve the product.

3.2.4 FOSS documentation

Documentation is a critical resource when working with software, and is important for the success of a FOSS solution. The reason is that the documentation is used by both the solution's users and developers, and inaccurate or unhelpful documentation can stop a FOSS project in its tracks if they find it hard to understand the solutions functionality and how it's developed. FOSS documentation doesn't differ very much from documentation about proprietary software. One difference is however that FOSS needs well-documented source code and documentation about how new developers can contribute. This is not necessary for proprietary software that uses the Cathedral-model for system development, because the source code isn't released. There are three types of product documentation; reference, tutorial and usage (more advanced material).[27]

There are three sources of FOSS documentation, that each occurs at different stages in the products life cycle [27];

- **The development team** – delivers reference documentation early in a products life cycle (most useful for highly technical users). If this is the only documentation available, the product is probably fairly new and not widely used, and can be considered as an immature product.
- **The user community** – typically web postings done by users that have problems or need help, and solutions are provided by users that want to share their experience with others (best suited as "fill-in" documentation, as the usefulness and quality varies). Postings are used in all stages of the products life cycle, but typically appear as the community grows

and the product matures. If there are a significant number of postings, the product can be considered as at least a fairly mature product, because there is a good-sized user base.

- **Commercial publishers** – comprehensive documentation written by authors with expertise in the subject. Commercial documentation is typically published when a product is considered as very mature with a significant current and potential user base, because it is necessary that it exists a sizable market that will purchase the book.

As we can see it is important to explore what kind of documentation that is available for a FOSS when it is evaluated, since it can be an indicator of how mature a product is.

3.2.5 FOSS selection process for Joly

Selection of FOSS products for the development of Joly have been done by evaluating the properties of the product, and going through the basic steps for evaluation as described above;

- **Identify candidates that have a solution.**
  The search for candidates was based on finding competitors to solutions we already new existed.
- **Read existing reviews.**
  We used articles, books and developers opinions about the candidates.
- **Briefly compare the programs attributes to the ones that are required.**
  The candidates had to support/fulfil the non-functional requirements for Joly.
- **Perform an in-depth analysis of the top candidates.**
  We evaluated the candidates after the criterions described above (license, community, stability and development activity, and documentation) and read articles comparing the candidates.

Evaluation and discussion of possible FOSS solutions for Joly are described in the following chapters (summarized in tables), where the need for such a solution is mentioned. During the evaluation we have considered the technology choices DHIS has made, but the selection of FOSSs have been done by choosing the product that seems like the most appropriate solution for Joly. The experiences we have made with the selected FOSSs are discussed at the end of this thesis, together with considerations about using the solutions in other projects, and in particular in DHIS.

3.3 Software maintenance

The other focus we have had during our research is designing Joly to easily adapt to changes that can occur in the software maintenance process. Change is a fundamental force that makes software development challenging, and the ability to change is an essential property of software as a medium [35] . Software is expected to change in response to changes in its environment and requirements, and the process of handling these changes is called software maintenance. Software maintenance is a way of dealing with change to prolong the aging process of the system. When the development of a new application is started, one of the requirements should be that the application is easy to maintain so that it will have a long operating time. Since our main focus has been the process of developing Joly, and not maintaining it after it was released and deployed, we have focused on designing the application so that changes that might occur at a later point in time can be easier to handle. We will therefore only shortly describe the importance of software maintenance, and how it can affect a products lifetime.

In 1968 the Nato Software Conference described the state of software development as a *software crisis*, meaning that the quality of software was generally unacceptably low and that deadlines, user requirements and cost limits were not being met. After the conference a lot of research was funded to try to solve the crises, which lead to a new engineering field – software engineering. [36] Software engineering aims at making the software fault-free, meeting the users requirements and be developed within cost and time limits. Research has shown that as much as 60-90% of software's lifecycle costs doesn't go to the initial development, but on maintenance activities [37]. Maintenance of the software to ensure that is as fault-free as possible and meets the user's requirements, is therefore a major aspect of software engineering, and lot of practises, techniques, and tools have been developed to reduce the maintenance cost.

In general, maintainability refers to "the ability to maintain a system over a period of time …[which] will include ease of detecting, isolating and removing defects" [38]. There are many definitions of software maintenance. IEEE [39] defines the term as:

> *Software maintenance is modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.*

The Eureka Software Factory project EPSOM has another definition of software maintenance that complements the IEEE definition (quoted in [36]):

> *Software maintenance is the process performed on a software system after delivery to answer a request for information, to correct faults, to improve performance or other non-functional attributes, to adapt the product to a changed environment, to add new functionality or modify the existing ones, to improve the software maintainability, or to anticipate future problems.*

Software maintenance is a process that provides an opportunity to sustain and improve the quality of a software product. The process is initiated by requests that can be divided into the following categories [40];

- **Corrective maintenance** – performed to remove a defect, is performed at unpredictable time once enough serious defects has occurred.
- **Adaptive maintenance** – a change in the software to accommodate changes in its environment (e.g. new hardware, operating system or interfaces).
- **Perfective maintenance** – addition of new functionality to improve some aspect of the system (e.g. response time, throughput, and memory size, or widen the functionality of the product).
- **Preventive maintenance** – involves changing some aspect of the system to prevent failures that can occur before the damage is done, and improve maintainability by making the software less complex.
- **User support** – provide services to answering user requests.

The Software Maintenance Life Cycle (SMLC) is a model that recognises four stages in a products life cycle. It is not inevitable that the product's life will decline, but the time it takes before it gets there is dependent on how the product reacts to changes, and if the product is designed to easily adapt to these changes. The model classifies the categories defined above into three main categories: repair, enhancement and user support, where each of the categories dominates one of the maintenance stages in a product's life and forms a maintenance regime. The four stages are (see figure 3.5) [40];

- **Introduction stage** - maintenance demand mainly comprises user support.
- **Growth stage** – maintenance demand shifts from user support to repair.

- **Maturation stage** – maintenance focus is on expanding software functions to lengthen the software's lifetime.
- **Decline stage** – the system reaches its limit of embedded technology and becomes ineffective. Users require software replacement, and the system must be integrated with other new systems, or it must be developed a replacement policy and architecture.

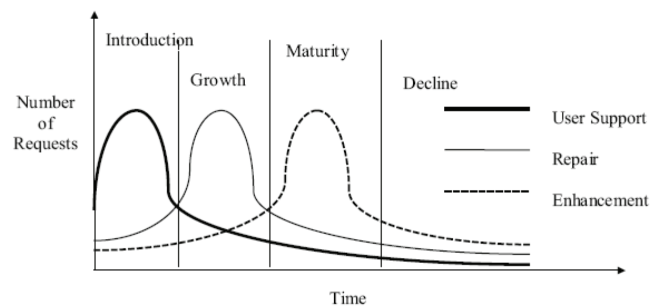


Figure 3.5 Software Maintenance Life Cycle (SMLC) [40]

The maintenance process of a FOSS differs from proprietary software in that there are no face-to-face interactions among the maintainers of the software, since many of the developers work in isolation and are not known to each other. The developers have to rely on the documentation of the source code, and on communication done through developer mail-lists or message boards. It is therefore required that FOSS solutions are highly maintainable, since insufficient documentation, the lack of proper interface definitions and structural complexity can discourage new contributions from its developers. A lot of research on software maintenance for proprietary software has been performed, but the life cycle maintenance activities in FOSS systems are processed under a different paradigm, and there are no contractual obligations for maintenance as it is with proprietary software. There has so far been done little research in how the FOSSs life cycle affects the maintenance and operating time of a system. We are of the opinion however, that when the system has an active community that contributes to the development of the software, it could increase the systems operating time as the source code is open for anyone and bugs could be detected and fixed, and new functionality added, at a more rapid rate than for proprietary software.[38]

Software ages as it evolves, and there are three main problems that causes aging [35]:

- **The stagnation problem** – stagnation occurs when modifications required to keep pace with change are deferred. When the software is finally modified, the developers are forced to deal with large amounts of change. Costs of modifying stagnant software increases non-

linearly with time, and when the costs become prohibitively high, the software becomes immutable because the software is too costly to replace or modify. The previous versions of DHIS (version 1.3 and 1.4) faced the stagnation problem and became immutable, and the start up of the development of DHIS 2.0 was therefore necessary to ensure the continuous use of the system.

- **The fatigue problem** – structural erosion caused by modification. The modifications conflict with the software's original design, e.g. modified in sub-optimal ways to quickly address new requirements from its users, and can lead to degradation of the software's operational qualities such as usability, reliability, performance and supportability.
- **The brittleness problem** – measure how much structural erosion (damage) is caused by a given change, and is generally determined by how well the software anticipates such a change. This problem is also called the preplanning problem, which indicates that it can be difficult to plan for the changes that can occurs, since its not possible to know precisely what will change.

Designing for change is the first step in controlling software aging, and how well it is done can determine a products success. The ability to design for change depends on the ability to predict the future, and a thorough analysis of future changes should therefore be a part of every product design and maintenance action. [41] We will in the following describe how we have designed Joly to prevent the problems defined above to occur.

3.3.1 Designing for change in Joly

As for any software development, we wanted to design Joly to be easily maintainable. We will only shortly describe here how the problems identified above can be addressed, as we will go thoroughly into the different methods for preventing them, like the use of design patterns and encapsulation, in the following chapters.

The problems that causes aging can be prevented, at least to some degree, by designing the system for change [35]:

- **Preventing stagnation**: By responding to changes in the environment as they occur, the amount of change that must be tackled at any one time is reduced and the resulting modifications are modest. The most common methods for preventing stagnation are:

  - Iterative development – provides periods of stability since the environment only changes between iterations, which contributes to reduce the amount of change needed to be tackled at any one time.
  - Agile development – capacity to rapidly and cheaply change the software under its development in response to changes.
- **Reducing brittleness**: Anticipating various classes of modifications, and designing the system to accommodate them, can reduce brittleness. The most common techniques for reducing brittleness are:
  - Encapsulation – hides implementation details from its clients and thus prevents changes from propagating.
  - Adaptive design – partitioning responsibility among the components according to rate of change and reason for change by using design patterns.
- **Reduce fatigue**: Reorganizing of the software and changing the systems design to accommodate to a modification can reduce fatigue, this can be achieved by:
  - Refactoring – is less expensive when done early in the life cycle, patterns and anti-patterns can be used.

3.3.2 Investigation process

Our investigation includes the following steps;

1. Evaluation of suitable FOSS products for development of a web application with the non-functional requirements that DHIS has for its version 2.0 (described in chapter 5 and 6).
2. Design of the system with regard to the non-functional requirements, as well as designing for change (described in chapter 5 and 6).
3. Development of the first working version of Joly with the FOSS products and tools identified in step 1 (implementation described in chapter 7).
4. Evaluation of the three steps above with regard to the development process, FOSS products and tools used, and the flexibility in the design to adapt to changes in the future (experiences from the investigation process is described in chapter 8).

3.4 Summary

HISP has started its development of DHIS version 2.0, and has defined a set of non-functional requirements for transporting DHIS to a new technological platform. These non-functional

requirements will be used in our investigation of developing a web application – Joly – for the Institute of Informatics at the University of Oslo, which will be described in the next chapter, in addition to Joly's functional requirements. HISP has chosen to use free and open source products under the development of the new health information system, and the system itself will also be free and open source software. When free and open source software is used in the development of an application, it is important to evaluate the products before they are adopted, to ensure that they meet the non-functional requirements for the application, and that they are mature and stable. In this chapter we have defined the evaluation process we have gone through, and the set of criterions the products are evaluated by. The following chapters describe the results of our evaluation, in addition to design and implementation of the Joly application.

We have also emphasized the importance of designing for change in this chapter, to ease the maintenance process that a system will go through after its release. By identifying certain foreseeable changes that will occur in the future, and designing the system so that it can easily be modified to meet these changes in the systems environment, we can prolong the system's operating time. Design patterns, encapsulation, and an iterative development process are the most common methods for designing for change, and an evaluation and discussion of how this can be applied to both DHIS and Joly in the best possible manner are also described in the following chapters.

Chapter 4

Joly

Joly is a web application meant for online submitting of mandatory student assignments in the programming courses at the Institute of Informatics at the University of Oslo. The courses run once or twice a year with a number of admitted students ranging between 100 and 1000. Each course has a set of assignments the students need to complete before taking the final exam. These assignments are handed in as a program file containing the student's solution.

Today, the practice of handing in a program file is to send it by email to the teaching assistants in the programming courses. The students are assigned to groups with one teaching assistant responsible for each group. There is no practice of gathering these programs and comparing them to each other, or to compare them with older programs. This has lead to the need of a system that gathers all handed in assignments and processes them in order to find attempts to cheat, i.e. a pure copy or a slightly modified version of another student's assignment. This is the main goal of the Joly system.

All program files submitted to Joly are going to be processed in order to find any similarity with previously submitted programs stored in a database. A comparison algorithm should be applied to each program when submitted, giving it a similarity factor which works as an indicator for the teacher, telling whether the program could be a copy of another or not, thus recognizing a possible attempt to cheat. The comparison algorithm works with a template defining the elements in the program to match, and the degree of positive matches sets the similarity factor. The system is named Joly after the known corruption hunter Eva Joly.

4.1 System requirements

This chapter describes the functional and non-functional requirements of the Joly application.

4.1.1 Functional requirements

The application should allow for two different categories of users to interact with the system from different interfaces designed for their specific needs and privileges. The different user categories are the students and the employees. The employees are divided into three groups; (1)teaching assistants, (2)lecturers and head teaching assistants, and (3)the administrator. All users shall access the system through a web-interface and thus some users – the employees – need to be securely logged on to access the restricted functions of the system.

Students

The students are only allowed to submit their assignments. This has to be done through a web form where the student inputs data about him/her self, the course and group he/she is attending and which assignment is being submitted in addition to the program file containing the student's solution. The system has to check the correctness of the data given and respond to the student with a receipt indicating that the assignment has been handed in.

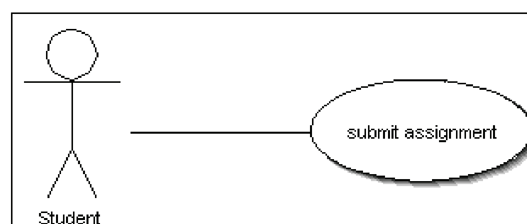


Figure 4.1.1 Use case "Submit assignment".

Each mandatory assignment must be registered with a deadline for submission. A student should still be able to submit an overdue assignment in case there is an agreement between the teaching assistant and the student to do so. The system must notify the teaching assistant in case the assignment has been submitted past due.

An assignment may comprise of several files other than the program file containing the student's solution. A student must therefore be able to upload more than one file through the students' user interface. It's only possible to submit one program file and this is the only file that is to be saved in the database and processed for similarity. All files must be sent to the teaching assistant by email.

Teaching assistants

The teaching assistants are responsible for correcting and registration of approved assignments submitted by the students assigned to their group. After a student has submitted his/her assignment to the system and the program file has been processed, the teaching assistant should have access to the program and be able to view, print and download the program as a file. In addition, the teaching assistant should be able to see each program's status after the system has marked it with a similarity factor and be able to view the resembling files for further examination. The teaching assistants use a separate system for registering approved student assignments, thus the application is not intended to be used for this.

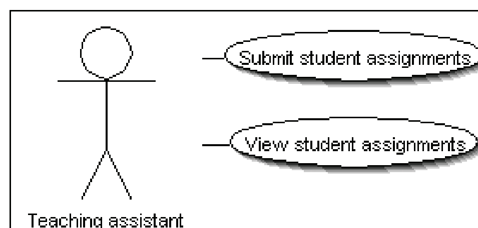


Figure 4.1.2 Use case "Submit student assignment" and "View student assignments".

Lecturers and head teaching assistants

Lecturers and head teaching assistants should have all the privileges of the teaching assistants, in addition to the ability to view all programs submitted by every student in every group at the course. The maintenance of the groups created under each course, as well as the meta-data to each assignment given in a course, should be accessible thru the lecturers and head teaching assistants' interface.

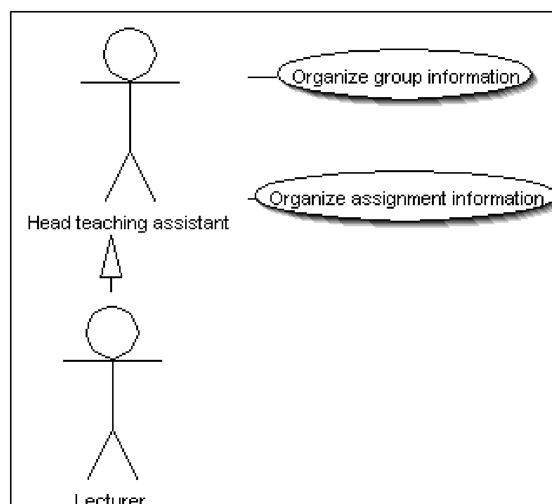


Figure 4.1.3 Use case "Organize group information" and "Organize assignment information".

Administrator

The administrator of the system should have all the privileges as the above mentioned user groups, as well as the ability to modify and maintain the data in the database. The administrator is responsible for creating new courses, assigning users and their privileges and the overall maintenance of the system. The template used for the similarity processing should be accessible and modifiable by the administrator, thus giving the administrator the responsibility for defining the degree of positive matches which indicates similarity.

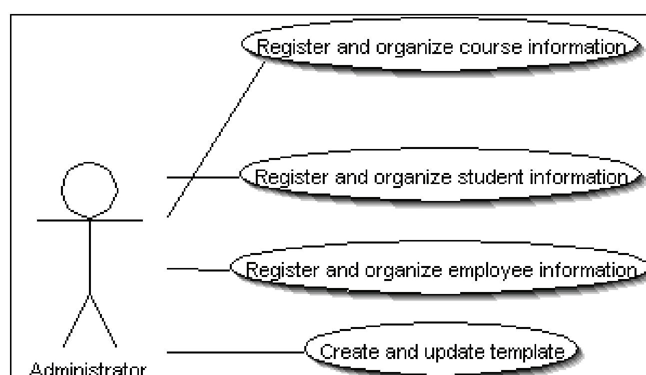


Figure 4.1.4 Administrator use cases.

4.1.2 Non-functional requirements

The non-functional requirements of the system are very similar to those of DHIS:

- Open source software development.

- Platform-independency. Meaning the application should be portable to other platforms and be able to adopt different DBMSs.
- Easily maintainable code.
- Modern technology.
- Web enabled user interface.
- Java development platform.
- Support incremental development process with several developers.
- Extendable and decoupled design. In general, the extensibility of software determines how easily changes can be accommodated.

4.1.3 Implemented functionality

This chapter describes the functions implemented in the Joly application. The application runs and functions properly with all the described functions. The description is divided into a description of the non-restricted function available to the students, and a description of the restricted functions available only to the employees. The restricted part of the Joly application is called JolyAdmin. The application's domain is described in the following domain model (figure 4.1.5).

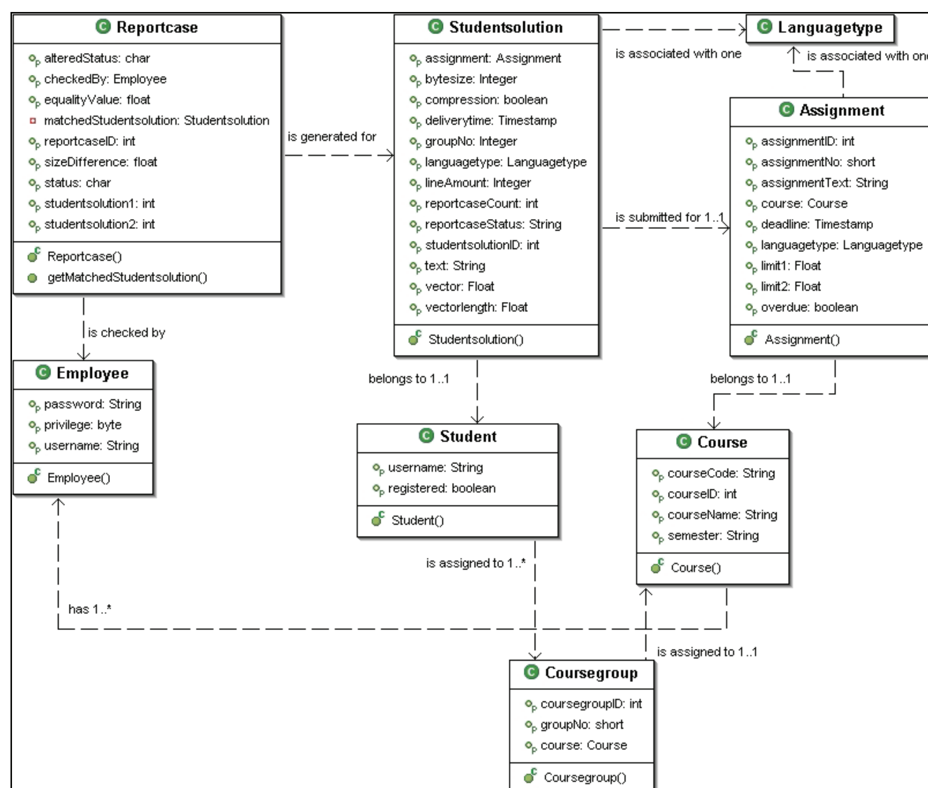


Figure 4.1.5 Domain model of Joly.

User interface

The web pages representing the user interfaces are aimed at being intuitive and user friendly for users of some knowledge of browsers and web page interaction. The following snapshot is an example of one of the web pages from the student's interface.

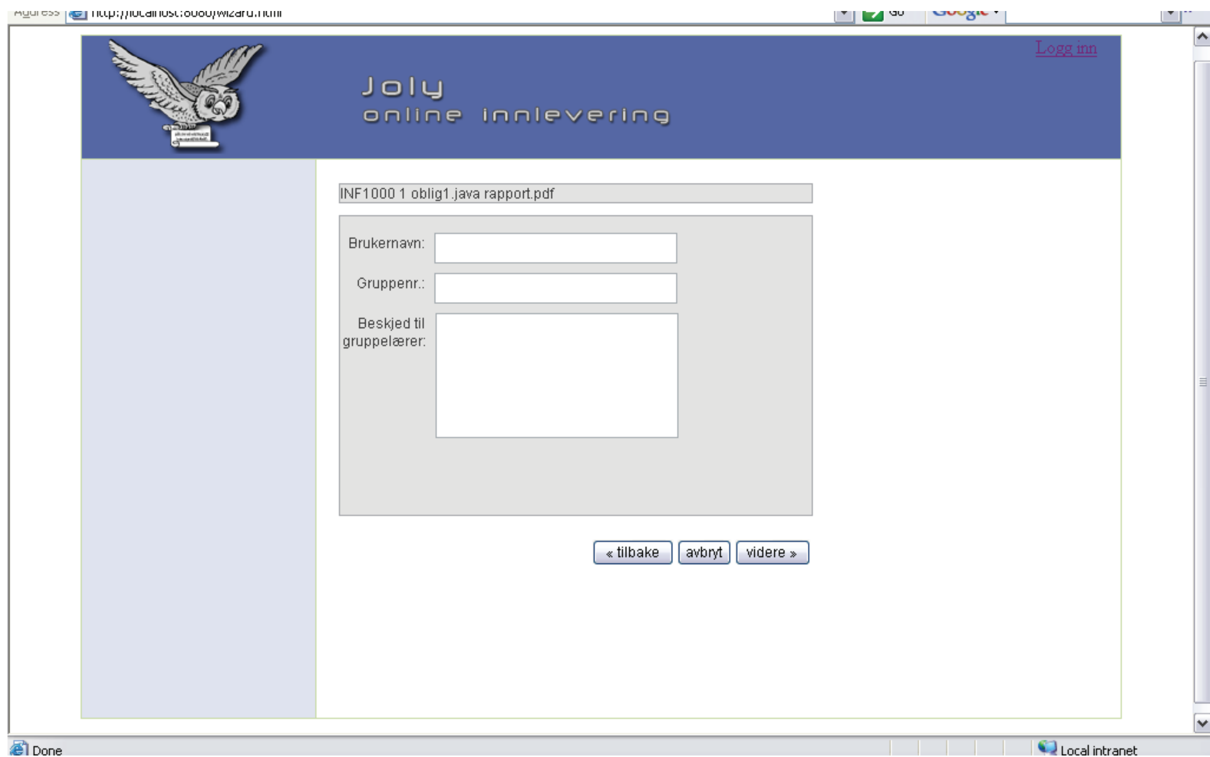


Figure 4.1.6 Snapshot of the user interface.

4.1.4 Non-restricted function

Use case: Submit student assignment

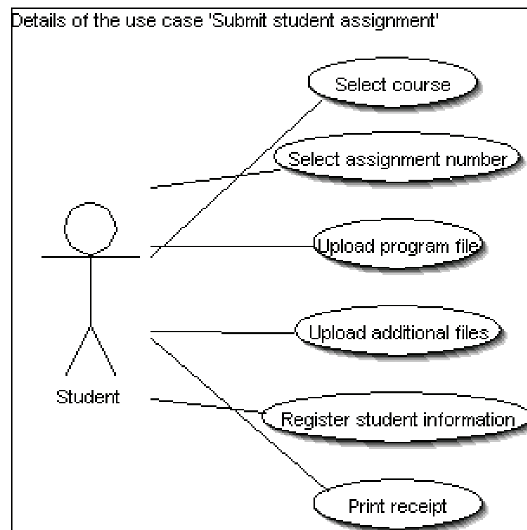


Figure 4.1.7 Detailed use cases of use case 'Submit student assignment'.

The user scenario of a student submitting his/her assignment is implemented as a wizard, controlling the student's navigation thru the different steps of submitting an assignment. Each step is an implementation of the detailed use cases described in figure 4.1.7. To go to the next step in the wizard, the proceeds button must be pushed. The student also has the choice of:

a) Pushing the cancel button to cancel the submission.

b) Pushing the back button to go back to the previous step.

The below figure describes the different elements found in the wizard pages.

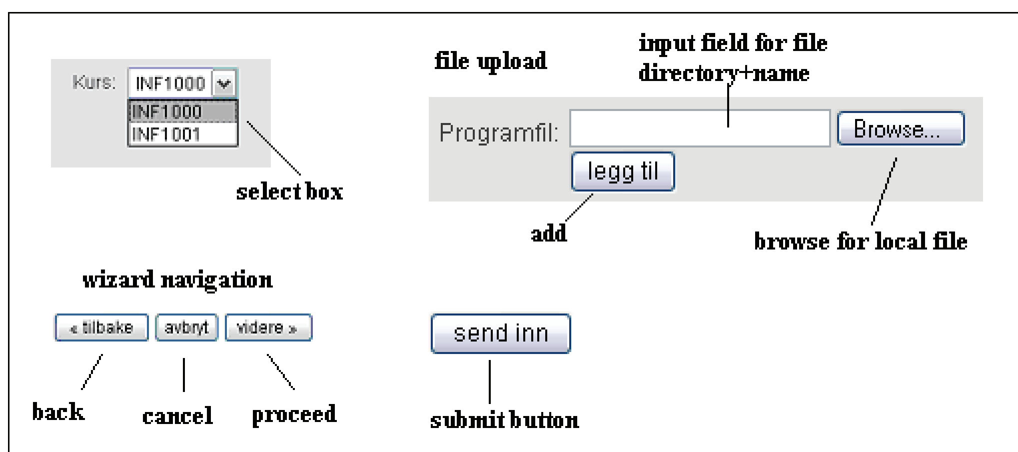


Figure 4.1.8 Wizard form elements.

User scenario:

1. The student selects a course from the available courses in a select box.

2. The student selects an assignment number from the select box of assignments registered at the previously selected course.

3. The student chooses the program file located on his/her local machine to be submitted via a file dialog. To avoid spamming and overload in the database, this function is restricted to files with a maximum size of 50 Kb.

4. This is an optional step. The student can upload additional files that are required by the assignment. If there is more than one additional file they must be zipped before upload because the function is restricted to only one additional file. This function is skipped by simply pushing the proceed button.

5. The student register his/her username, the group number of the group he/she is assigned to, and optionally a comment to the teaching assistant. This comment field is supplied so that students can notify the teaching assistant of specifics regarding the submitted assignment, e.g. the username(s) of students collaborating on the assignment. This

comment is only sent as part of the email to the assistant teacher and not saved in the database.

6. In this step the student is presented with the information he/she has registered to be able to correct entries if he/she wishes. This is the final step before the assignment is submitted. The assignment is submitted and sent to the server for processing when the student pushes the submit button.

Registrerte opplysninger

Brukernavn:	hannevi
Gruppenr.:	101
Kurs:	INF1000
Oblignr.:	
Programfil:	oblig1.java
Andre filer:	
Beskjed til gruppelærer:	

« tilbake | avbryt | send inn

When the request for submitting the assignment is received by the server, several operations are performed;

a) The system checks whether the student is registered at the course or not.
b) The system checks whether the assignment was submitted past due.
c) A comment is added to the program file containing the student's username, the course code, assignment number, group number and time and date of submission.
d) If the student is registered, the program file is processed by the comparison algorithm.
e) An email is sent to the teaching assistant of the group the student informed of, containing all information and files submitted, and messages stating the result of a), b) and d). There is also an option of sending a copy of the email to the lecturer at the course. This option can be turned off or configured by the administrator to be activated just in case the result of the comparison algorithm exceeded a pre-configured limit which indicates the similarity level.

7. After the student has submitted his/her assignment the wizard ends by presenting a receipt. This receipt can be printed by the student.

Kvittering for innlevert obligatorisk oppgave

INF1000

oblig1.java

hannevi

101

skriv ut

Error messages and the general error page

If the student tries to upload a program file exceeding the maximum file size of 50kb, the student will be prompted with an error message explaining the cause. The student can mark the file for removal, push the remove button and then will be navigated back to the program file upload page to choose another file.

Figure 4.1.9 Error message when trying to upload file exceeding maximum file size.

If the student tries to submit the assignment without registering the required information, or if there is a type mismatch, he will be prompted with error messages stating the cause.

Figure 4.1.10 Error message for type mismatch and missing data.

If an error occurs during the submission process which terminates the process, the student will be directed to a general error page. This error page contains a simple error message that is common to all non-specific and terminating errors of the system, e.g. when a database connection fails. If such an error occurs, the user must start the process from the beginning.

Figure 4.1.11 General error message.

4.1.5 Restricted functions - JolyAdmin

All employees must be logged in to access the restricted functions in JolyAdmin. This is done thru a login form. Once the employee is logged in, the application will make available the functions the employee has access to by his/her privilege level.

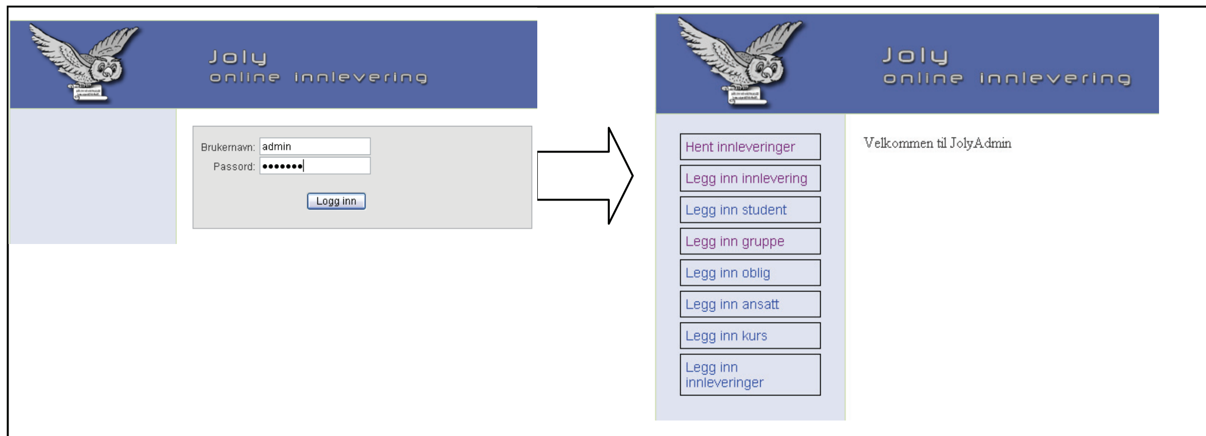


Figure 4.1.12 By logging in as an administrator all functions are made available.

Use case: View student assignments

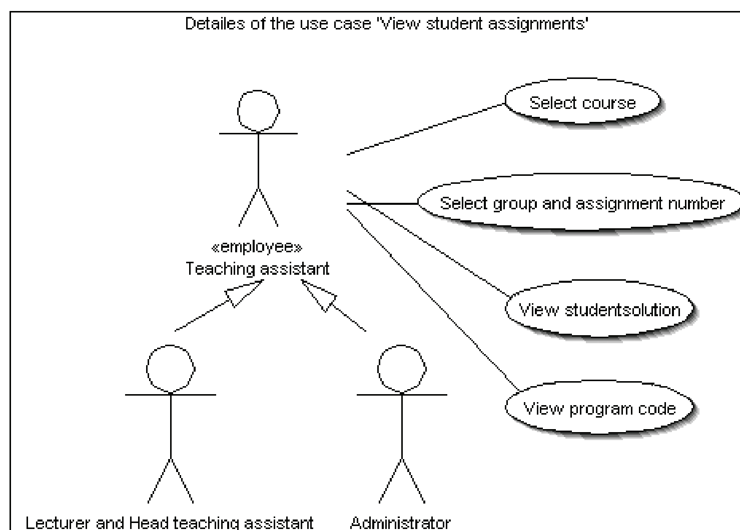


Figure 4.1.13 Detailed use cases of use case 'View student assignments'.

User scenario:

1. The employee selects a course from a select box with all available courses.

2. The employee selects an assignment number and group number registered under the selected course.

Innleveringer

INF1000

Gruppenr.: 101

Oblignr.: 1 Vis

Figure 4.1.14 Step 2 – select assignment number and group number.

3. All student solutions registered at the selected group and assignment number are displayed.

Innleveringer

Oblignr.: 1

Brukernavn	Innleveringstidspunkt	Likhetsstatus
hannevi	2006-04-21 17:54:44.0	Ulik
therst	2006-04-18 10:58:37.0	Ulik

Figure 4.1.15 Step 3 and 4.

4. The employee can view one specific student's solutions by clicking the link stating the student's username.
5. All the solutions belonging to the selected course, assignment and group registered with this student are displayed. The status of the comparison algorithm is shown and if there is a similarity with any other program(s) this is indicated in a status field.

Innleveringsnr.	Innleveringstidspunkt	Likhetsstatus
1	2006-04-18 10:58:37.0	Ulik
2	2006-04-17 12:56:49.0	Ulik
3	2006-04-17 12:52:23.0	Ulik
4	2006-04-17 12:51:42.0	Ulik
5	2006-04-17 12:38:54.0	Ulik
6	2006-04-16 16:35:30.0	Ulik

Innleveringsnr.	Innleveringstidspunkt	Likhetsstatus
1	2006-04-21 17:54:44.0	Ulik
2	2006-04-21 12:13:43.0	Ulik
3	2006-04-18 10:56:07.0	L 1 Sammenlign
4	2006-04-17 19:59:06.0	Ulik
5	2006-04-17 19:50:13.0	Ulik

Figure 4.1.16 Step 5.

6. From this page the employee has access to two functions. The first is to view the student's program code in a new window (figure 4.1.17). The other function is only available if there were any similarities with other submitted programs. Then the employee can view the similar programs alongside the current student's program to examine the similarities further.

Joly - Microsoft Internet Explorer

```
// hannevi 101 INF1000 1 Tue Apr 18 10:56:07 CEST 2006
Oppgave 1:

class Oppgave1{
    public static void main(String[] args){
                        System.out.println("Rachmaninovs 3. klaverkonsert har et vakkert åpningstema");
          } //slutt main
} //slutt class Oppgave1

----------

Oppgave 3:


class Utskrift {

    //I denne setningen er det tre feil: static er feilskrevet,
    //String[] mangler [] og den siste parantesen er feil - sto ( skulle vært {

    public static void main(String[] args) {

                        //Denne manglet et semikolon på slutten av setningen.
                        System.out.println("Beethoven skrev Skjebnesymfonien");

                        //Og denne manglet avsluttende " etter symfonier.
                        System.out.println("og åtte andre symfonier.");

    }

}

----------

Oppgave 4:

class Oppgave4 {
    public static void main(String[] args) {
                        System.out.println("Gruppe:        Bill Evans Trio");
                        System.out.println("Tittel:        Exlorations");
                        System.out.println("Platenummer:   RLP 351");
```

Done | Local intranet

Figure 4.1.17 View of one program.

Joly - Microsoft Internet Explorer

Velg sammenlignet innlevering: therst | ok

```
// hannevi 101 INF1000 1 Tue Apr 18 10:56:07 CEST 2006
Oppgave 1:

class Oppgave1{
    public static void main(String[] args){
                        System.out.println("Rachmaninovs 3. klaverkonsert har et vakkert åpningstema");
          } //slutt main
} //slutt class Oppgave1

----------

Oppgave 3:


class Utskrift {

    //I denne setningen er det tre feil: static er feilskrevet,
    //String[] mangler [] og den siste parantesen er feil - sto ( skulle vært {
```

```
// therst 101 INF1000 1 Tue Apr 18 10:58:37 CEST 2006
Oppgave 1:

class Oppgave1{
    public static void main(String[] args){
                        System.out.println("Rachmaninovs 3. klaverkonsert har et vakkert åpningstema");
          } //slutt main
} //slutt class Oppgave1

----------

Oppgave 3:


class Utskrift {

    //I denne setningen er det tre feil: static er feilskrevet,
    //String[] mangler [] og den siste parantesen er feil - sto ( skulle vært {
```

Done | Local intranet

View of two similar programs.

Use case: Organize group information

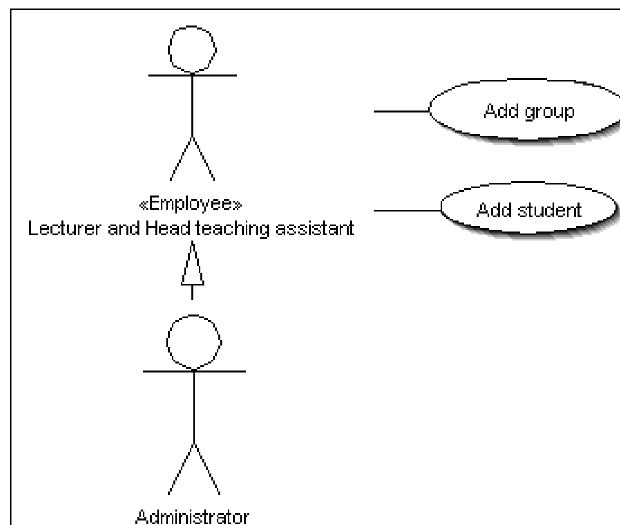


Figure 4.1.18 Detailed use case of use case 'Organize group information'.

The functionality implemented in the "Organize group information" use case is the 'Add group" and "Add student" functions. The "Add group" function is a simple function for

adding groups to an already registered course. The employee selects the course in which the groups are to be added and fills in the group numbers of the new groups. When the new groups are submitted the system displays the newly registered groups.

Legg inn gruppe(er) på kurs

Kurs: INF1001

Gruppenr.: 1

Gruppenr.: 2

Gruppenr.: 3

Gruppenr.: 100

Gruppenr.: 101

Gruppenr.: 102

Legg inn

Hvis flere grupper ønskes registrert på kurset, gjenta prosessen.

Innlagte opplysninger

Gruppenr.: 1

Gruppenr.: 2

Gruppenr.: 3

Gruppenr.: 100

Gruppenr.: 101

Gruppenr.: 102

Figure 4.1.19 Add group.

The ”Add student” function is an equally simple function for adding students to an already registered course and group. The employee selects the course and group in which to register new students before filling in the username of all the new students. This is a function for the prototype Joly application. A real life scenario would require a registration of approximately 1000 students, thus this functionality is not sufficient. The application must be extended with a function for registering student information by processing a text file.

Legg inn student(er) på kurs

Gruppenr.: 101

Brukernavn: hannevi

Brukernavn: therst

Brukernavn:

Brukernavn:

Brukernavn:

Brukernavn:

Brukernavn:

Brukernavn:

Brukernavn:

Brukernavn:

Legg inn

Hvis flere studenter ønskes registrert på gruppen, gjenta prosessen.

Innlagte opplysninger

Brukernavn: hannevi

Brukernavn: therst

Figure 4.1.20 Registering students at a selected group. The course has been selected at the prior web page.

Use case: Organize assignment information

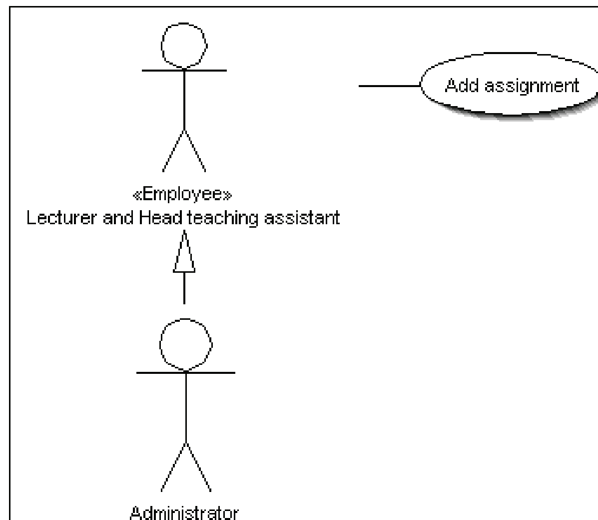


Figure 4.1.21 Detailed use case of use case "Organize assignment information".

The only function implemented in the "Organize assignment information" use case is the "Add assignment" function. This is a function for registering a new assignment in a course. The employee selects the course to add the new assignment, and fills in the fields for all necessary information regarding the assignment. When the assignment has been submitted, the system displays a page with the newly registered assignment.

Legg inn oblig for kurs

Kurs: INF1000

Oblignr.: 5

Innleveringsfrist: 20006-06-05 23:59:59

Format YYYY-MM-DD HH:MM:SS

Likhetsgrense: 0.8 Normalt 0.8

Obligtekst:
```
Du får gitt følgende problemstilling
som du skal lage et datasystem for.
Meteorologisk institutt har en rekke
værstasjoner rundt i Norge. For hver
slik værstasjon har vi
oppgitt et entydig stasjonsnummer,
```

Legg inn

Innlagte opplysninger

Kurs: INF1000

Oblignr.: 5

Innleveringsfrist: 20006-06-05 23:59:59.0

Likhetsgrense: 0.8

Figure 4.1.22 Adding a new assignment to a course.

Use case: Register and organize course information

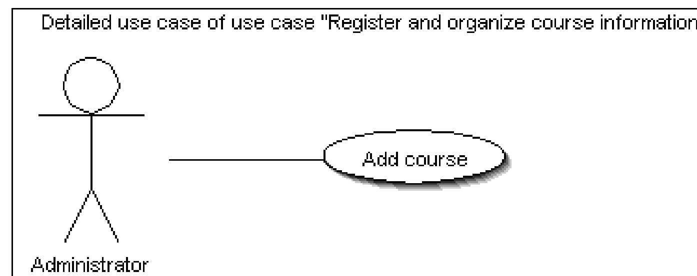


Figure 4.1.23 Detailed use case of use case "Register and organize course information".

The function implemented for this use case is the "Add course" function. This enables the administrator to add a new course to the system.

Legg inn kurs

Kurs: INF1010

Kursnavn: Objektorientert programmering

Semester: V2006

Format HYYYY / VYYYY

Legg inn

Figure 4.1.24 Add course.

Use case: Register and organize employee information

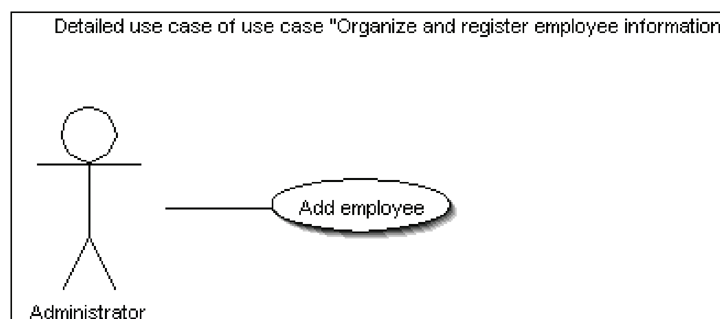


Figure 4.1.25 Detailed use case of use case "Register and organize employee information".

The function implemented for this use case is the “Add employee” function. This enables the administrator to add a new employee to the system, set the privilege level and assign a password to the employee.

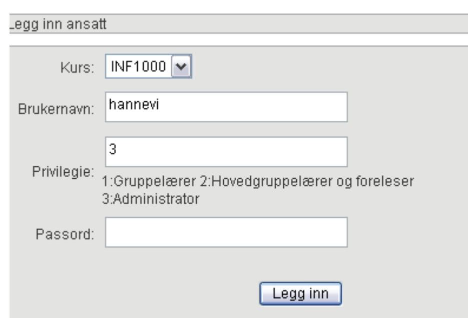


Figure 4.1.26 Add Employee.

4.2 Summary

The implemented functions are only a subset of the intended functionality of the system. The intended functionality can be found in the original design document in appendix B. The functionality implemented describes all the necessary aspects of Joly as a web application. We will use this prototype implementation to explore the different aspects of web application development and seek experience that can be useful for the further development of Joly, as well as the development of DHIS 2.0.

Chapter 5

Architecture, design and tools for Joly

This chapter contains an overview of the architecture and design of Joly, as well as an introduction to the tools used.

The aim of the development of Joly is to build a system that can be used as a prototype for the investigation of different aspects associated with the development of web applications. The Joly application will be the basis and starting-point for the final release of Joly, which will be a standalone application to be used by the Institute of Informatics at the University of Oslo. In addition, the Joly application will be built and designed with the DHIS 2 project in mind, meaning that we aim to fulfil the same non-functional requirements as the DHIS 2 application has, and to identify problems that arise when developing web applications subject to these requirements.

We will explore how to build an architecture that is the most suitable for a web application, as well as what development platform to use and what features it can bring to the development of a web application.

The design of the application is one of the most important aspects of our investigation process. The design of the application must result in an application that is highly flexible and can accommodate the varying requirements of a widely deployed system such as the DHIS system. We seek to minimize the efforts needed in adapting to various technologies, extending the application with new and/or different implementations of components and maintenance of the application. To achieve a flexible application, the design must support loosely coupled dependencies to and within the different components of the application. Components with larger responsibilities should not be too weld together, because if so a

minor change could result in a cascade effect of changes needed to be made in the dependent components. It should be made possible to replace both larger parts of the system, such as the database, and smaller parts such as a specific implementation of a function.

5.1 Architecture

The architectural foundation of the Joly system is a web-enabled architecture. This architecture forces a separation of the application's user interface from the application logic. Further there is a logical partitioning of the application logic into domain logic, the display logic that defines the user interface, and the persistent data of the application. This enables the system to be separated into three parts. The system architecture of Joly is described as a logical three-layer architecture comprising of a presentation layer, a domain layer and a persistence layer. Seeing that Joly is a web application, the user interface, which is a browser, is not seen as part of the presentation layer. The presentation layer only represents the server side code responsible for the user interface. The source of the data the application uses, the relational database, is in the same manner not referred to as the persistence layer, but as a resource from where the persistence layer receives its actual data.

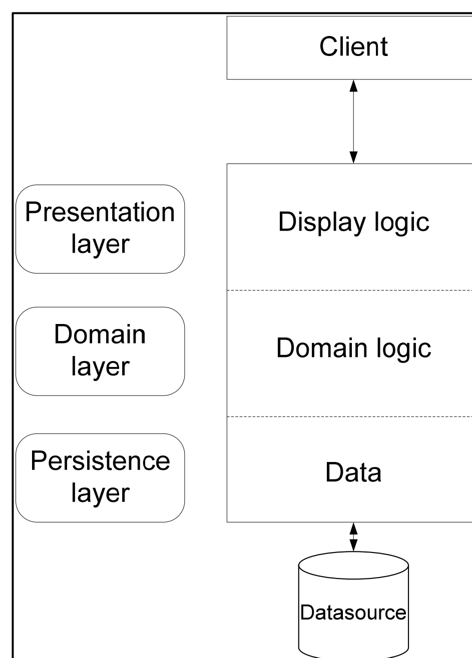


Figure 5.1.1 Three-layer architecture

The goal of this architectural approach is to decouple the technology used in each layer so that the technology used in one layer can be changed with little impact on the other layers. This makes the system more flexible and open to adopting new technologies in the future.

The alternative architecture would be a two-layer architecture where the presentation logic and domain logic are combined in one layer and the second layer represents the database. This approach offers good performance because it requires the least processing for one single request, but the trade off is good code factoring and the ability to access the application logic from multiple directions. The mix of display and application logic also inhibits code reuse

and easy switching to other view technologies to accommodate different types of clients. In these types of applications it is also harder to change the datasource used, because the data access logic is coded with the rest of the application logic.

5.2 Java 2 Enterprise Edition (J2EE) development platform

The J2EE development platform offers a wide range of services to the development of Java applications. It is designed to ease the development of multi-tiered applications with a thin client – like a thin-client web application. The J2EE comprises of many standardized, reusable and module based components [42]. We have used the J2EE 1.4 release, which includes the Java Development Kit (JDK) 5, in the development of Joly. The J2EE implementation can be downloaded for free from the Sun Microsystems web site and there are many free and open source projects related to Java/J2EE development available. This makes the J2EE platform suitable for the development of DHIS, seeing that they can take advantage of free and open source products in the development, and redistribute their system as an free and open source system with the integration of other free and open source products. We will use some of these J2EE open source products in the development of Joly to show how free and open source implementations can support the development of a J2EE application.

The J2EE APIs extend the base Java 2 Standard Edition (J2SE) APIs with additional functionality for large-scale Java applications. The Joly application is built using the following extensions:

- **Servlets 2.4** – package javax.servlet
- **Java ServerPages (JSP) 2.0** – package javax.servlet.jsp
- **Java Standard Tag Library (JSTL)** – a custom tag library specification
- **JSP 2.0 Expression Language API (EL)** – package javax.servlet.jsp.el
- **Java Database Connectivity (JDBC)** – package java.sql, javax.sql
- **Java Transaction API (JTA)** – package java.transaction
- **JavaMail API** – package javax.mail
- **JavaBeans Activation Framework (JAF)** – package javax.activation

With the J2EE as development platform we have a solid, standardized framework to build the Joly application on, but J2EE is also a complex standard that leaves us with many design decisions [43]. Application logic that incorporates J2EE features is subject to performance overhead [43], thus we need to carefully consider the features that we choose to incorporate in our application.

The Servlet API provides a service to handle HTTP request processing and adding dynamic content to web applications. This is an underlying service that Joly needs in order to support user interaction thru the web. JSP is an extension to the Servlet technology, and enables us to combine dynamic content with static content in web pages. In addition to plain JSP, we have included the JSTL and JSP Expression Language to enhance the JSP features. The JSPs have the responsibility of displaying the content, and the Servlets are responsible for generating the content. Servlets can use any J2EE feature offered by the application, including our own domain logic. How these three components (Servlets, JSPs and domain logic) of the application interact is a design decision left to the developers and thus needs to be addressed by us.

The JavaMail API abstracts the use of (amongst others) the Simple Mail Transport Protocol (SMTP) and Multipurpose Internet Mail Extension (MIME) that enables the sending of emails from our application. JavaMail is a low-level and complex service and it's dependant on JAF for handling the complex content of an email, such as the attachment feature required by Joly. JavaMail is a rich API for the development of large email systems and Joly only needs to use a subset of the classes available. The challenge here is to find a way to implement a simple mail sending logic that does not show the complexity of the underlying technology, enabling easily understandable and maintainable code.

The JDBC API provides a database-independent connectivity between Java applications and a wide range of SQL databases by abstracting the functionality common to relational databases. JDBC makes it possible to establish a connection to a persistent storage, perform queries on the persistent data, and process the results. A way of efficiently performing queries, and mapping of the results from the relational database to objects in Java in a simple manner needs to be addressed. JTA abstracts application code from the underlying transaction implementation, and allows the application to control the transaction boundaries via a

consistent API, independent of a transaction manager implementation. JTA contributes to keep applications portable between different kinds of execution environments and containers. When handling transactions, the implementation of a transaction manager and the desired transaction isolation levels, needs to be addressed by the developers, namely us.

5.3 Web servers

A J2EE web application is packaged as a WAR file (Web Application aRchive) so it can be deployed to a Java web container and accessed by http clients through the internet. The web container needs to support the JSP and Servlet API versions used in the application. The WAR file will contain all the Servlets and JSPs that make up the web application, as well as any supporting class files and static content such as html files. An XML deployment descriptor file is used to define how the web container should deploy the web application.

The web application acts as a delegate to the domain logic. The domain logic can either be classes in a specific *classes* directory of the web application, separate JAR files in a library of the web application, or standalone Enterprise Java Beans (EJBs) separated from the web application. If the domain logic is to be accessed by external components, such as EJBs, the domain logic cannot be a part of the WAR and deployed to a web container, because then the files will not be visible to these external components. For an application with such requirements, the domain logic needs to be packaged in an EAR file (Enterprise Application aRchive). To deploy an EAR file it is not sufficient with a web container. It requires an additional type of container – an EJB container. To have an application that has both a web module and an EJB module, it requires both containers and thus a full J2EE application server.

For the Joly application it is sufficient to put all the domain logic in the WAR, because the domain logic is not going to be accessed from other components than the Servlets within the web application. In addition, the domain logic does not depend on interaction with any external components such as EJBs, thus we only need a web container to which we can deploy our application.

The DHIS 2 will be a significantly larger system than the Joly application and this could lead to the need of a full J2EE application server to serve different types of clients, not just HTTP clients. In addition, it is likely that the system will be made up of more J2EE components that need another container than a web container to run, and components that need to be able to interact interchangeably with each other. To support such a large-scale J2EE application, an application server that supports the full J2EE stack is required.

For the Joly application we have considered different open source web containers that support our deployment requirements. The requirements are simple and few:

- Support for the Servlet 2.4 API.
- Support for the JSP 2.0 API.
- Support for the HTTP 1.1 protocol.

There are many open source implementations of the J2EE references for web containers. Sun Microsystems' own implementation, which is part of the J2EE development package, is not free and open source so we can not use this server in distribution of the Joly application. Because of this restriction we have chosen to omit it all together and use an open source implementation during development as well as for deployment of the application.

5.3.1 Jetty HTTP server and Servlet/JSP container

Jetty is an open source Java web server distributed under the Apache 2.0 License, free for commercial and non-commercial use and distribution. The user and developer community of Jetty is very active and the server has been used and included in many development projects. Because the server is small and easily embeddable into other projects and products, it is often used to provide http and servlet management for larger-scale products. For instance, the Jetty server is being used in open source application servers such as JBoss, Jonas and Apache Geronimo, and it can also be found in IBM and Cisco products.

We have not located any commercial documentation of the Jetty server, a reason for this could be that Jetty is a very simple web container, and there shouldn't be any need for a complete book to cover its functionality. The mail lists indicate that there is an ongoing effort to improve the web container, which is also confirmed by the fact that a new version, Jetty6

beta, is under development. Because of all this we consider Jetty to be a mature and stable web container that can be used for the development of Joly.

Jetty can be considered both as a HTTP server and as a JSP/Servlet container. As a core HTTP server (supporting HTTP 1.1) it can serve static content such as HTLM. As a JSP/Servlet container it can be used for deployment of standard J2EE web applications. Jetty uses Tomcat's JSP parsing module, the Jasper JSP engine, for compiling JSPs. This module is included in the Jetty release. Because Jetty supports both HTTP and JSP/Servlets it can be used as a standalone web server.

Jetty has been released with every new JSP and Servlet version, thus Jetty supports our requirement of JSP 2.0 and Servlet 2.4 support, and as mentioned, HTTP 1.1.

5.3.2 Apache Tomcat Servlet/JSP container

Tomcat is an open source Java web server developed by the Apache Software Foundation as one of their top level projects. It is distributed under the Apache 2.0 License, and like the Jetty server, available for both commercial and non-commercial use. Amongst the contributors to the development of Tomcat is Sun Microsystems. Sun has included the code base of Tomcat in the J2EE SDK application server. Other products embedding Tomcat includes JBoss, Jonas and Geronimo application servers.

Tomcat is often used in combination with the Apache HTTP server to provide static content, but Tomcat also includes an internal HTTP server and can therefore be configured to be used as a standalone web server.

To use Tomcat as a standalone web server one must configure a HTTP *Connector* component. Because Tomcat is mostly used with Apache HTTP, the documentation for configuring Tomcat most often refer to how to set up Apache as the HTTP server. Our experience with setting up Tomcat leaned us towards the opinion that Tomcat was unnecessary troublesome to configure. Because Tomcat requires more configuration than the Jetty server, we have chosen to use Jetty. Still we would like to point out that there is no significant difference between these two web servers. It is also unnecessary for both Joly and

DHIS to commit to only one web server. The objective of both developments is to make a portable application, thus being able to run on any Java web server that supports the technical requirements.

Product	License	Community	Activity	Documentation	Comments	Candidate
Jetty	Apache License 2.0	Active community.	Last stable version 5.1.10 released 5.01.05	Reference manuals Mail lists	Lightweight container	X
Tomcat	Apache License 2.0	Active and large community.	Last stable version 5.5.17 released 14.04.06	Reference manuals Mail lists Commercial	Lightweight container, some extra configuration required	X

Figure 5.3.1 Evaluation of web containers after the criteria for evaluating FOSS.

5.3.3 Open source J2EE application servers

Because of an incompatibility between free and open source licenses and Sun Microsystems' legal agreement that defines how Java standards are to be developed, there has not been an open source implementation of a full J2EE application server. In recent years however, there has been made changes to the Java Community Process so that it now allows free and open source implementations of J2EE specifications.

After the change in the Java Community Process, the Apache Software Foundation became the first open source J2EE TCK (Technical Compatibility Kit) licensee [44]. The ASF started a project called Geronimo for the development of a certified server implementing the full J2EE stack. Other J2EE certified implementations that followed were JOnAS and JBoss. These relatively new open source J2EE implementations can provide a way for DHIS to take advantage of the full J2EE stack, developing a large-scale J2EE application, and still be able to deploy the application on an open source server.

5.4 Design patterns

A design pattern is a description of a well-proven solution to a recurring design problem.

[45]

J2EE leaves us with a semantic gap between the abstractions and services it offers and our final application. This is not only a J2EE problem, but a recurring software design problem in any development platform one chooses. However, this gap has been subject to many applications before us, and developers have struggled with the same design problems that we now are to explore. A good software development process makes use of the experiences and successful solutions that developers have discovered on beforehand. These discoveries are described in *patterns* – a way of documenting software expertise so that it can be shared and reused in an application-independent fashion [46].

Design patterns solve many common problems in software development; communicating application design to other developers, designing for extensibility and maintainability, building an architecture that accommodates software change [47]. These issues are relevant to the development of Joly as well as DHIS. The non-functional requirements are very similar in both systems and reflect the common problems mentioned. Both systems are undergoing an incremental development process, though consisting of a development team of dissimilar size, both in need of a way to document and communicate application design.

Because the Joly application consists of data access logic, domain logic and presentation logic, the lack of a structured and clearly separated code could lead to a maintenance problem, as well as the inability to separate the application logic into layers of responsibility, as the tree-layered architecture requires. We have used design patterns to force a well-defined structure to the application, assigning responsibilities to the various parts of the application.

We are also applying the design patterns to resolve the non-functional requirements of the application. We are doing this so that we can accommodate the changes one may meet in the future or during the development process. There may be changes to the underlying technology the application is based on. Changes that we are unaware of at this point, and cannot predict will happen, but very possibly could occur. We do not know how such changes will affect the application or development of the application, most likely such changes will occur many years from now, when the current developers have left the project and the responsibility has been assigned to other developers. Design patterns describe the *hinges*; places in the design where the application is able to adapt to a different implementation, making it less expensive to accommodate future changes [47]. A hinge is what joins objects

together, and by clearly describing the hinges we can make it more evident where objects can be substituted with a different object.
Joly has a non-functional requirement of being platform-independent. We define the application's platform dependency by the specific platforms the application is bound to and requires in order to run properly. To achieve the development of a platform independent application, we saw the need to not only choose a platform independent language such as Java, but also to design the application so that is has the ability to adapt to different technologies. The choice of a layered architecture is one step of the way, seeing that we now have the ability to decouple the technology used in each layer. Exactly how to decouple the layers is an additional design process and requires more detailed and in-depth design decisions. This is where design patterns are used to design adaptability into the application at designated places; the hinges [47].

5.4.1 Inversion of Control and the Dependency Injection pattern

Inversion of Control (IoC) is the principal of shifting the control of an applications dependencies to an external source, e.g. event listeners in a user interface or, as it is in our case, the framework surrounding the application. In these days it's mostly referred to in the context of framework containers [48]. For these containers the inversion is about how they lookup a plugin implementation. Objects usually depend on other objects or components of an application to perform its tasks. These objects or components are termed dependencies. With IoC, the implementation of the dependencies are plugged in by the framework instead of being hard coded into the collaborating objects. The objects declare their dependencies and then it's the responsibility of the framework container to supply the required implementations. The use of Inversion of Control conceals complexity from the application code, hiding the specific implementations so that objects only knows of the provided Interfaces. By hiding the specific implementation of the dependencies, we create the hinges where the application can adapt to different implementations.

The figures described next show how the dependencies are controlled in a non-IoC approach and how the control shifts by applying IoC.

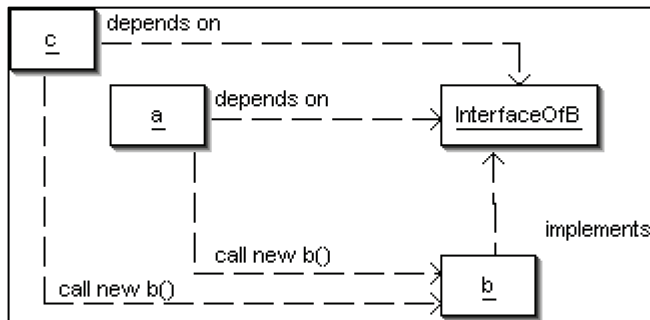


Figure 5.4.1 Object dependencies without IoC.

Both object a and c depends on InterfaceOfB. They both need to be aware of the specific implementation of InterfaceOfB, b, and create an instance each of b.

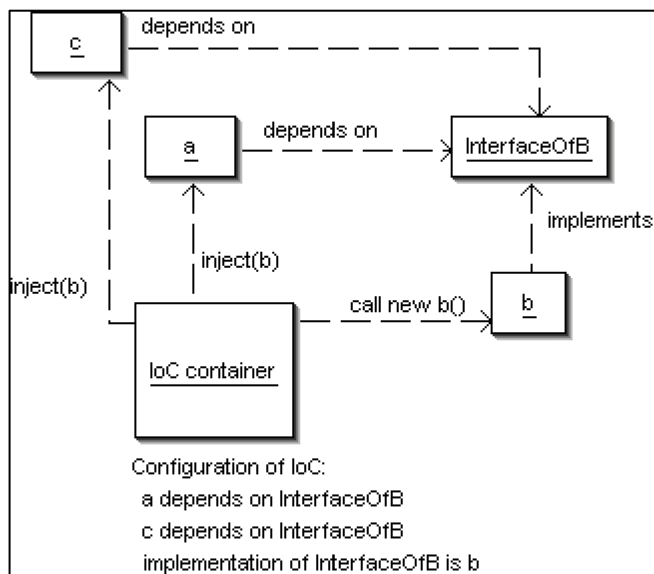


Figure 5.4.2 Applying IoC and configuring an IoC container.

By applying IoC, object a and c does not need to be aware of the specific implementation of InterfaceOfB. Neither do they have to create an instance each of b. The IoC container has a configuration that says that a and b need an implementation of InterfaceOfB, and that this implementation is b. The container creates one instance of b and injects this instance into both a and b.

Dependency Injection

The Dependency Injection pattern is a design pattern used to apply the Inversion of Control principal to objects. This pattern addresses how to lookup a plugin implementation and describes how to employ a separate object, which assembles the application's objects and populate these object's dependencies with appropriate implementations. In this way, the dependent objects never create an instance of the implementation themselves, but receive it from the container where the objects are configured. There are three types of dependency injection; Constructor injection, Setter injection and Interface injection.

By Constructor injection, an objects dependency, declared by a property in the object, is populated with a specific implementation by passing it in as an argument to the constructor. This requires that the object has declared a constructor that takes the appropriate argument, or arguments in the case of multiple dependencies. When using Setter injection, the implementation is passed in and assigned by the setter method of the property declared. In case of multiple dependencies, it requires one setter method for each property declaring a dependency. The last type of Dependency Injection is the Interface injection. With this technique an Interface is used for the injection, and the object with the dependency needs to implement this Interface to obtain the implementation.[49].

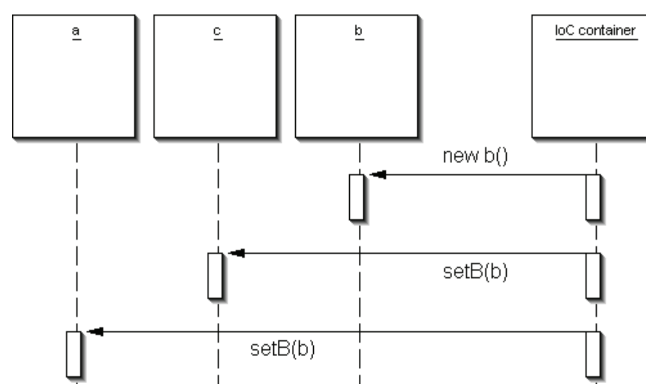


Figure 5.4.1 Dependency Injection by Setter injection.

An alternative to Dependency Injection is the Service Locater pattern [49]. By the Service Locator pattern one defines a class that is responsible for looking up services commonly used by many objects in the application. The main difference between the Dependency Injection pattern and the Service Locator is that with the first one you get a “push” effect of the dependencies from the IoC container, whereas with the Service Locator you “pull” the dependencies from the service locator object. In practice this means that the “pulling” of required services must be coded into the dependent objects, and calls to the service locator’s look up methods must be made when a service is required. We have chosen to apply the Dependency Injection pattern because the IoC container we use promotes this pattern, and it gives us a very clean and easily maintainable code. However, the Service Locator pattern can be used in conjunction with the Dependency Injection pattern where developers find it appropriate, so we have not tied the implementation to one particular design. The Service Locator is a core J2EE pattern [50], used for locating JNDI-administrated service objects, such as EJBs and JMS. It takes a JNDI directory name from the client and obtains the

required service for the client. In this way, the Joly application can be extended with other services, possibly remote services, without interfering with the current services applied by the Dependency Injection pattern.

Dependency Injection is applied to the whole application. We use the pattern on all layers, so the IoC container is responsible for configuring and managing all beans made available to it.

5.4.2 Design within the layers

The database is separated from the application to achieve the requirement of a DBMS independent application. How the system communicates with a DBMS needs to be abstracted in the application code, and made easily accessible and understandable to developers who wish to change the DBMS, thus the design of the persistence layer is a critical aspect of our application. In the next chapter we describe how we apply the Data Access Object pattern to the persistence layer to abstract and decouple the data access logic.

The domain specific operations are assigned to the domain layer. This is the heart of the application, encapsulating all domain logic and making the operations available to the presentation layer. To decouple this layer we need a single entry point to access the logic. The solution to this is to apply the Façade pattern, as described in the next chapter, to build a stable articulation point for the client to interact with. This is also a layer that should enable different implementations of the specific logics, such as the mail sending routine and the comparison algorithm. This would make the system easier to extend with additional functionality and changing the specific implementations. Decoupling within the layer is done by defining interfaces and letting the IoC container plugin the implementations.

For the application to be web-enabled it requires the ability to offer user interaction via a user interface accessible thru the web. As we mentioned in section 5.2, J2EE offers us Servlets and JSP to handle this interaction. The responsibility for handling user interaction is assigned to the presentation layer. In chapter 6 we describe how we have applied the Model-View-Controller pattern to this layer to handle the interaction between the collaborating objects.

5.4.3 Design maintenance

Designing and coding the hinges in an application is relatively time consuming, depending on the developers experience with using patterns and whether there already exists a pre-defined set of patterns that the developers are required to use, or if they need to assemble their own set of patterns. It takes time to find a set of patterns that apply well to the requirements of the application. Using design patterns usually includes creating additional classes, interfaces and articulation points, to tweak the logic into the desired pattern, and this is another factor that could increase the development time. This is the tradeoff for designing a highly flexible system that aims for a long operational time and continued development process. It will take longer before the implementation can start, and the design patterns must be reconsidered in every phase of an incremental development process.

Even if we have succeeded in creating an application that is as flexible as we aimed for, the application is vulnerable to developers fixing defects in the application's functionality without following the constraints of the pattern applied. If a developer breaks a pattern during a change, this could mean that the application no longer supports one or more non-functional requirements. But if the developers preserve the integrity of the patterns, the maintenance of the application is inexpensive in comparison to pattern-less applications. Hinges build without design patterns are even more vulnerable to careless functionality changes because it's harder to detect the damage the developers may have done. Ordinary methodologies rarely focus on the non-functional requirements, the hinges, and thus there is no way of testing whether they are preserved or not after a change [47].

5.5 Frameworks

Designing an application by applying patterns can be a very time consuming and cumbersome experience, as there are many patterns to choose from. When we first started designing the Joly application, we studied several patterns, many including other patterns and some were just renaming of old and maybe upgraded patterns. We saw that the detailed patterns, like the Front Controller pattern and the Model-View Separation principle, could be seen combined in more high-level design patterns as the architectural MVC pattern. Because of these somewhat overwhelming pattern choices, we decided to find a tool that could help and guide us in the

development process. One approach to ease these design decisions is to use one or more frameworks.

Frameworks provide a way to fill the semantic gap between the services offered by the J2EE platform and the application. They bring another abstraction level to the services, guiding the developers to the correct use and implementation, and disguise the complex, low-level APIs of the development platform. Frameworks use class libraries in conjunction with applying patterns to create the infrastructure for building applications [51]. We can customize a framework to our application by inheriting and instantiating selected framework classes [52]. This enables us to focus on the application logic, and let the framework handle the glue that pieces together the application. The framework is a semi-application surrounding our application and providing features so that we do not have to implement them from scratch by our selves. This enables code- and software expertise reuse in an even higher degree than design patterns, because we combine patterns with pre-defined classes from the framework, and these classes handles and interacts with the J2EE abstractions and services.

The next step is to find a set of frameworks that is suitable for the Joly application.

5.5.1 Hibernate

Hibernate is a persistence and query framework that provides a mapping between objects and relations, when Java is used as the programming language and data needs to be stored and retrieved from a relational database. The use of a framework to perform such a mapping can provide an additional abstraction layer on top of the JDBC API, and relieving the developer from implementation details of making objects persistent and performing queries on the data stored in a database.

There are many alternatives to using Hibernate, such as other persistence frameworks that performs object and relational mapping, abstraction tools, or just using JDBS. These alternatives will be explored in the next chapter, in addition to a explanation of why the Hibernate framework is the chosen method for persisting objects.

5.5.2 Spring

The Spring framework is an open source framework designed for J2EE development. It is targeted at the application as a whole, covering all layers of the application. The origin of the framework is the infrastructure code in the book *Expert One-on-One J2EE Design and Development* by Rod Johnson, published in 2002. In his book Johnson points out, by experience, that the use of Enterprise JavaBeans (EJBs) in J2EE applications is often overkill. He describes an alternate approach using a lightweight, JavaBeans-based framework. This framework became the Spring framework when it was published as an open source project on SourceForge in 2003.[53].

There are not many alternatives to the Spring framework seeing that the framework is an application framework that covers all layers of a Java application, and not just separate layers or web applications as many other frameworks do [53]. The alternative to developing an application using Spring would be to follow the J2EE recommendations, and possibly also choose separate frameworks for the individual layers and/or individual aspects of the application.

Spring versus plain J2EE

Though J2EE has been in widespread use since its arrival, its success has been limited in practice. This is mainly due to the complexity it brings to applications that they do not necessarily need [54].

Even though Spring is a J2EE-based framework, it differs from the J2EE recommendation in the use of EJBs as mentioned. The choice of following the J2EE recommendation would lead to the need of an EJB container to manage the EJBs. Because the Joly application is a relatively small application, we seek to minimize the technical requirements of running the application. Thus the Spring way seems more adequate, since then we would only need a lightweight Servlet container to run the application.

Spring components

The Spring framework has a module based, layered structure which makes it possible for developers to either apply the whole framework to every layer, or employ separate

components where needed. The framework contains the following components, each distributed in its own JAR file:

- **Spring Core**: Bean Factory and IoC container.
- **Spring Context**: Application context - information relating to the application.
- **Spring AOP**: Spring supports Aspect-Oriented Programming for adding functionality at runtime.
- **Spring ORM**: Object-Relational Mapping support.
- **Spring DAO**: Data Access Objects.
- **Spring Web MVC**: A module that provides the implementation of Model-View-Controller in web applications.
- **Spring Web**: provides the context of the web specifics.

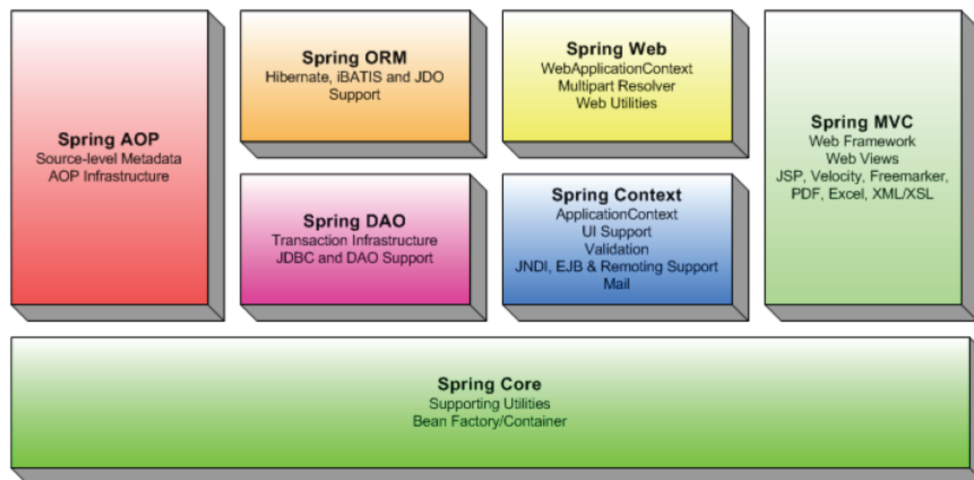


Figure 5.5.1 Overview of the Spring Framework [55].

Spring's objective is to be a "lightweight" framework with little impact on the overall application. Developers can pick and choose which part of the framework to use as they see fit. It is also possible to integrate other frameworks with Spring, enabling developers to use their preferred framework at specific layers. The listing on the next page shows the Spring modules that the Joly application uses.

Spring Core	Bean Factory / Container IO resource
Spring Context	Application Context Message source Mail Validation
Spring Web	Servlet Util Bind Multipart resolver WebAppliationContext
Spring MVC	Controllers, Views, Models View resolver JSP/JSTL renderer
Spring ORM	Hibernate3 support
Spring DAO	DAO support Transaction

Spring Core

The core of Spring is the Bean Factory – a container based on the Inversion of Control principal; externalizing the creation and management of dependencies between objects. The Bean Factory is not a web container such as Tomcat or Jetty, or heavy-weight EJB container, but a simple class that represents a "bean container" or "object library" and provides configured beans to the application [53]. The configuration of the beans in the container is done in a configuration file. This is where each bean is configured with its dependencies to be handled by Dependency Injection, and it is also possible to assign pre-defined properties to the beans. Spring's Bean Factory offers Dependency Injection by Setter injection as well as Constructor injection, but the most preferred technique of the Spring team is the Setter injection, thus we follow their recommendation and use the Setter injection method.

There are two kinds of Bean Factories available; the XmlBeanFactory, where the bean definitions are read from an XML file, and the ListableBeansFactory where the definitions

are read from a properties file. The most common way to configure J2EE applications is by XML files [53], so we have chosen to use an XmlBeanFactory.

Other open source IoC container implementations

There are several open source implementations of IoC containers for Java available; PicoContainer and HiveMind are two examples of frameworks other than Spring that offers an IoC container. The main difference of these frameworks is that PicoContainer and HiveMind provides only an IoC container, and Spring provides an IoC container in addition to all the previously mentioned components.

We have chosen to use the Spring container even though one could have chosen another container and integrated it with the different Spring components we are using. The main reason for choosing Spring's container, is to keep the number of independent tools used at a minimum, and we only wish to integrate a tool if it clearly gives us an advantage. Using an IoC container gives us an implementation of IoC and Dependency Injection, which is what we want to apply to our application to achieve decoupling of dependencies within our application's objects. We see it as an advantage to incorporate an open source container rather than designing our own.

Why we want to keep the number of independent tools at minimum can be explained in a scenario of what would have happened if our project had started a year earlier, and we had chosen the Apache Avalon container. At that time, the Apache Avalon container would have been a fairly good choice, because it was the result of a solid Apache Software Foundation project. Then, in late 2004 the project was closed down and dissolved into four other projects, some evolving the core IoC container of Avalon, and some evolving the extensions added in the original project [56]. If we had chosen the Avalon container, an upgrade could have lead to the need of an integration of several distinct tools to support the original requirement of our application. Even if all four were not required, it would have been an additional workload trying to figure out which projects to integrate.

This also shows that we must be aware of the possibility of our chosen tools being subject to the same discontinuance, or rather; separated continuance. The Apache Avalon project was dissolved because the developers could not agree on which direction the project should take.

Too many different development projects had emerged within the original project, pulling the project in separate ways [56]. The Spring framework is also a project containing many distinct components, but in contrast to Avalon, Spring is designed with this in mind from the beginning, thus the layered design and flexible component integration. This has not insured us that Spring will continue to develop each component of the framework we have used, but their components are independent of each other, and able to integrate with external components, making it easier to substitute a component with another from a different vendor. This is probably the best insurance we can get in regards to a free and open source project being able to evolve alongside our application, and still support our original requirements so that we can upgrade to newer versions of the framework.

5.6 Summary

Three-layer architecture is the standard architecture for web applications. By building the Joly application on this architecture, we have clearly defined where the application's logic associated with specific responsibilities should be placed. For small applications it could be sufficient to use a two-layer architecture, because it increases performance and development time by minimizing the design process that is required when building a decoupled application. However, the Joly application is designed with the large-scale DHIS system in mind, thus we have conformed to the standard tree-layer architecture of web applications.

The J2EE development is suitable for layered architecture development. It offers many services that support the requirements of Joly. These services are somewhat complex, and the use of frameworks on top of J2EE eases the development by providing another abstraction level to the J2EE APIs. Frameworks also offer implementations of design patterns that are well known to resolve the design issues of J2EE applications. We believe that incorporating frameworks in the development process could give DHIS the advantages of reusing software expertise. With our investigation we aimed to get a hands-on experience with different frameworks that seemed to be the new standard when developing a web application. With this experience we will seek to identify the problems associated with the use of frameworks.

The use of free and open source tools encourages reuse of program code by incorporating other free and open source tools. Most free and open source products are built on, or include,

one or many other free and open source packages, thus the use of free and open source tools introduces a large amount of dependencies to other packages. Spring provides a way to make a loosely coupling to other tools for an application. This can facilitate the use of many tools and make it easier to maintain the dependencies to these external tools.

Chapter 6

Detailed layer design of Joly

The objective of this chapter is to give a detailed description of the design decisions briefly laid out in chapter 5. We will present the design in each layer of our three-layered architecture, from bottom to top; first the database and the persistence layer, next the domain layer, and finally the presentation layer.

6.1 Persistence layer

The responsibility of the persistence layer is to encapsulate the behaviour needed to make objects persistent[1]. The behaviour includes reading, writing, updating and deletion of objects to/from a permanent storage. By isolating persistence logic into its own layer, the details of implementing persistence is hidden, and protects the application code in the other layers from changes in the persistence mechanism.[58] Before we describe the design of the persistence layer, we will present the choice of database management system for the application.

6.1.1 Database

There are many types of data management systems, but the three main types in use today are; relational database management systems (RDBMS), object-relational database management systems (ORDBMS) and object-oriented database management systems (OODBMS).

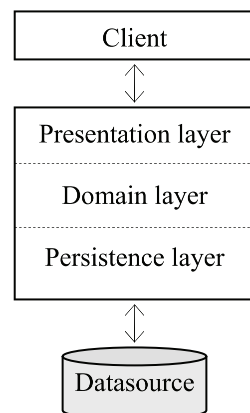


Figure 6.1 Three-layer architecture.

[1] A persistent object is "an object that exists after the thread that created it has ceased to exist" [57].

With OODBMS, the need for a tool to handle the mapping between objects and relations (described in next part of the chapter) isn't present, because the objects can be stored directly in the database. OODBMS stores two types of data; objects and values. An object has both state and behaviour, where its state is represented by values or previously defined objects, and its behaviour by operations on those values and objects. The ODMG 3.0 object data standard specifies how objects are to be stored in an OODBMS, and is based on OMG's[2] object model which includes concepts such as object, encapsulation, polymorphism and inheritance. The ODMG standard also includes ODL (Object Definition Language) and OQL (Object Query Language). No existing OODBMS supports complete object encapsulation, which limits the advantages of storing both an objects state and behaviour. [59]

An ORDBMS is a hybrid of object and relational databases, with an object front end layered over a relational back end. The main benefit of this type of database is the fact that it allows for storage of complex objects, making it unnecessary to use an ORM tool in the applications persistence layer. Most of the ORDBMS products adhere to the SQL:1999 standard. The standard has the same features as the object model used by OODBMS, but it doesn't match the object model used by object programming languages. This is because SQL:1999 was required to be backward compatible to the SQL:1992 standard for RDBMS. The object-relational capabilities are centred around user defined types (UTD's) that can have methods. UTD is roughly analogous to a class declaration in ODL.

RDBMS is based on the relational model introduced by Edgar F. Codd in 1970, where he proposed that database systems should present the user with a view of data organized as tables - called relations. Codd later defined a set of 12 rules that a database management system must implement to be qualified as a RDBMS, but the term has in the subsequent years came to describe a broader class of database systems. These database systems are defined as relational if they present the data to the user as relations, and if they provide relational operators to manipulate the data in tabular form [60]. If an object-oriented programming language, e.g. Java, is used to develop an application, and the underlying database is a RDBMS – there needs to be a mapping between the objects and the relations (as described in the next part of the chapter).

[2] OMG = Object Management Group. A distributed computing standards group.

OODBMSs aren't widespread in use today, and one of the reasons might be that even if it has advantages when dealing with complex objects, it doesn't otherwise show clear superiority over RDBMSs. RDBMSs are the most popular database system in use today, but ORDBMS are gaining a significant portion of the market share. This is mainly due to relational database users upgrading their databases to object-relational databases, to take advantage of using complex objects but still being able to use SQL queries. [59]

One of the requirements for DHIS is that it is platform-independent, something that also includes database-independency. Any of the three types of database management systems above should therefore be possible to use, but the implementation of the persistence layer is different for each type because of how it handles storage of an object. HISP has chosen to use a RDBMS, and uses Hibernate to perform the mapping of objects to relations.

Commercial RDBMS like the Oracle database, IBM's DB2 and Microsoft's SQL Server can be costly to purchase and often has an additional licensing fee for each user or device that will use the database. DHIS is a software product developed for use in developing countries, and the costs by purchasing a commercial database can thus be impossible to overcome. This is the reason why HISP requires that the database server, and other technologies used to develop DHIS, have to be open source. Open source databases are growing in popularity, and are according to a survey conducted in Europe by IDC[3] in 2005, used by 33 percent of the respondents [61].

We will now describe three of the major open source RDBMSs, and look at their licensing, which platforms they can be run on, SQL standard compliance, performance and scalability.

MySQL

The development of MySQL started because the developers of mSQL (MiniSQL) concluded that it lacked the speed and the features they wanted, and so the first version of MySQL was released in 1995. MySQL is developed, distributed and supported by the commercial company MySQL AB. MySQL database server is the most widely used free and open source relational database today, and MySQL AB claims that is has over eight million installations

[3] IDC = International Data Corporation

[62]. The database server is popular for web applications, and acts as the database component in the LAMP[4]/WAMP[5] platforms.

MySQL AB provides two different editions of the database; MySQL Certified Server and MySQL Community Edition. Of the two, only MySQL Community Edition is freely available. The database server is licensed under a “dual license”, meaning that the user can choose between the free and open source GPL, or a commercial license that can be bought if the database is to be used in a commercial application. MySQL runs on more than 20 platforms, including Linux, Windows, and a number of Unix-platforms.

MySQL (version 4.1) understands a subset of SQL92 and some subsets of the SQL99 syntax, along with its own extensions. Features like stored procedures, triggers, views and cursors, are not supported. After we started our development of Joly, MySQL has released a version 5.0. This version supports views, stored procedures and cursors, and version 5.1 will support triggers.

MySQL can handle large databases (maximum table size is 64 TB), and it handles connections very fast and thus making it very suitable for use for web applications [63] . MySQL supports both ODBC and JDBC for network connectivity, as well as native database access methods.

PostgreSQL

PostgreSQL is an ORDBMS based on the POSTGRES project at the University of California at Berkley Computer Science Department. POSTGRES originally used a query language called PostQUEL, but in 1994 the POSTGRES SQL interpreter was added (originally known as Postgres95) and was then re-licensed under the BSD license and renamed PostgreSQL. The ORDBMS isn’t controlled by any single company, but is directed by the community of developers and users. PostgreSQL claims to be the world’s most advanced open source database. [64]

[4] LAMP = Linux-Apache-MySQL-PHP/Perl/Python
[5] WAMP = Windows-Apache-MySQL-PHP/Perl/Python

PostgreSQL is released under the BSD license, and can therefore be used, modified and distributed without fee in both open and closed source (proprietary software). It runs on all major operating systems, including Linux, Windows (supported in version >= 8.0) and seven different Unix-platforms. PostgreSQL strongly conforms to the SQL:1992 and SQL:1999 standards, and supports features like triggers, views, sequences, stored procedures, cursers and UTD's. The procedures and triggers can also be written in other languages than its query language, such as PL/TCL, Pl/Perl and PL/Python. Like MySQL, PostgreSQL supports both ODBC and JDBC for network connectivity, as well as native database access methods. PostgreSQL is highly scalable (maximum table size is 32 TB), but it can be somewhat slow. PostgreSQL provides a lot of options for improving the performance.

Firebird

Firebird is a RDBMS that was forked from the open sources of the commercial database InterBase from Borland, which were released July 2000. The database was programmed, and is maintained, by the Firebird Foundation.

Firebird provides two server architectures; the classic and the super server. The classic allows for programs to directly open the database file, while the super server provides a server process and all SQL requests are done via the server using a socket. The super server is the main architecture for Windows, while UNIX style environments often have the choice of both the architectures. The super server implementation postdates the classic implementation, and is the future architecture of Firebird. [65]

The original modules that Firebird is build upon, are licensed under the InterBase Public License 1.0, and the new core modules added are licensed under the Initial Developer's Public License (IDPL) – both are modified versions of the Mozilla Public License 1.1. The licenses give the users the freedom to deploy the database for use with any third-party software, both open and closed source, without any registration, licensing or deployment fees. Firebird runs on Linux, Windows, Solaris, HP-UX, FreeBSD and MacOS X.

Firebird fully supports the SQL:1992 standard and most of the SQL:1999 standard, and provides the user with features like stored procedures, views, custom data types and triggers.

Firebird supports programming interfaces like ODBC, JDBC and some others. Firebird is like PostgreSQL highly scalable with a maximum table size of 32 TB, and claims to offer high performance and excellent concurrency [65].

6.1.2 Selection of database

The table in figure 6.2 shows how the RDBMS solutions considered is evaluated after the criterions we decided in one of the previous chapters, and they are all evaluated as the top candidates, though MySQL and PostgreSQL can be considered as more mature since they have been developed for a longer period of time. MySQL and PostgreSQL can therefore have the benefits of receiving support and feedback from an active and stable community, and more bugs can be discovered and fixed.

Product	License	Community	Activity	Documentation	Comments	Candidate
MySQL	Dual license: commercial and GPL	Active and large community.	Last stable version 5.0 released 24.10.06	Reference manuals Web forum Commercial	Runs on 20 platforms. Emphazises speed.	X
PostgreSQL	BSD license	Active and large community.	Last stable version 8.1 released 08.11.05	Reference manuals Web forum Commercial	Runs on all major operating systems. Emphazises functionality.	X
Firebird	InterBase Public License 1.0 & IDPL	Active and large community.	Last stable version 1.5.3 released 24.01.06	Reference manuals Web forum Commercial	Only runs on Linux, Windows, Solaris, HP-UX and MacOS X	X

Figure 6.2 Evaluation of RDBMS solutions after the criterions for evaluating FOSS.

DHIS has focused its development so far on MySQL and PostgreSQL, but they have in addition fully Java based alternatives such as Derby, and HSQLDB for use on stand alone clients where it can be run embedded in the client. [66] Firebird and other RDBMSs have also been evaluated for use, but have so far not been deployed.

Because we are going to relate our experiences with developing Joly to DHIS, we wanted to choose one of the databases that DHIS uses under its development, although Joly, like DHIS, is designed to be independent of the underlying RDBMS. Since Joly isn't going to be run on stand alone clients, the choice stood between MySQL and PostgreSQL. The two databases differ in their compliance to the SQL-standards and the features they offer, and while MySQL

is a RDBMS – PostgreSQL is an ORDBMS. But because Hibernate (that DHIS has chosen for handling of object and relational mapping) only supports RDBMS, PostgreSQL's object-relational features like UDT's cannot be deployed, and with that making the advantages of using PostreSQL over MySQL smaller. Thus we chose to use MySQL for the development of Joly, because it is somewhat faster and more scalable than PostgreSQL.

Since MySQL is licensed with the GPL, the resulting product will also have to be licensed with GPL if the RDBMS is distributed along with the rest of the application. Since we are using other licenses that conflict with the GPL (Apache licence version 2.0), it isn't possible to license the application under the GPL. MySQL can therefore not be distributed together with Joly, and has to be downloaded from MySQL's product web page by the user. Joly will only have one installation, because its users are going to access the system through a web-browser. We therefore don't see it as a problem that the user (in the case for Joly – the administrator) has to download the database management system himself/herself. DHIS on the other hand, for users that are going to use the health system as a standalone application because they don't have network access, MySQL cannot be used. This is the reason for why the DHIS team have developed other Java based alternative database implementations that can be distributed along with the rest of the application, in addition to the size being smaller than for MySQL or PostgreSQL.

6.1.3 Persistence layer design

One of the requirements for the application is that it is platform-independent - which also includes database-independency. It is therefore important that the persistence layer design, in addition to encapsulate the database access routines, also can adapt to different persistent stores.

The Data Access Object (DAO) pattern is one of the Core J2EE patterns. The DAO pattern is used to abstract and encapsulate all access to the data source, in our case – a relational database. The interface exposed by the DAO to the business layer does not change when the underlying data source implementation changes, with that the pattern allows the DAO to adapt to different storage schemes without affecting the components that relay on it [67].

Factory Method and the Abstract Factory patterns can be used in combination with the DAO pattern to make the design highly flexible. The DHIS design needs to be flexible to handle different persistence storages, and should therefore use the Abstract Factory pattern in combination with the Factory Method pattern. The Abstract Factory, implemented as an abstract DAO factory object, can be used to construct various types of concrete DAO factories, where each factory support a different type of persistent storage implementation.

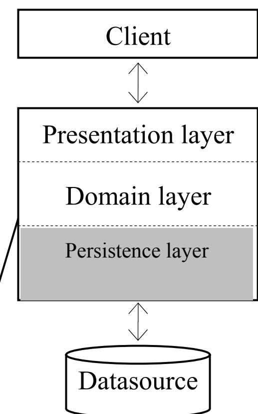


Figure 6.3 Three-layer carchitecture.

If the underlying persistence storage is not subject to change from one implementation to another, as is the case for Joly, the Factory Method pattern alone can be implemented to produce the number of DAOs needed by the application. Since this is the best suited design solution for Joly, and there should be possible to add the Abstract Factory pattern to the design at a later point if necessary, we choose to use the Factory Method pattern as the factory for the DAO strategy.

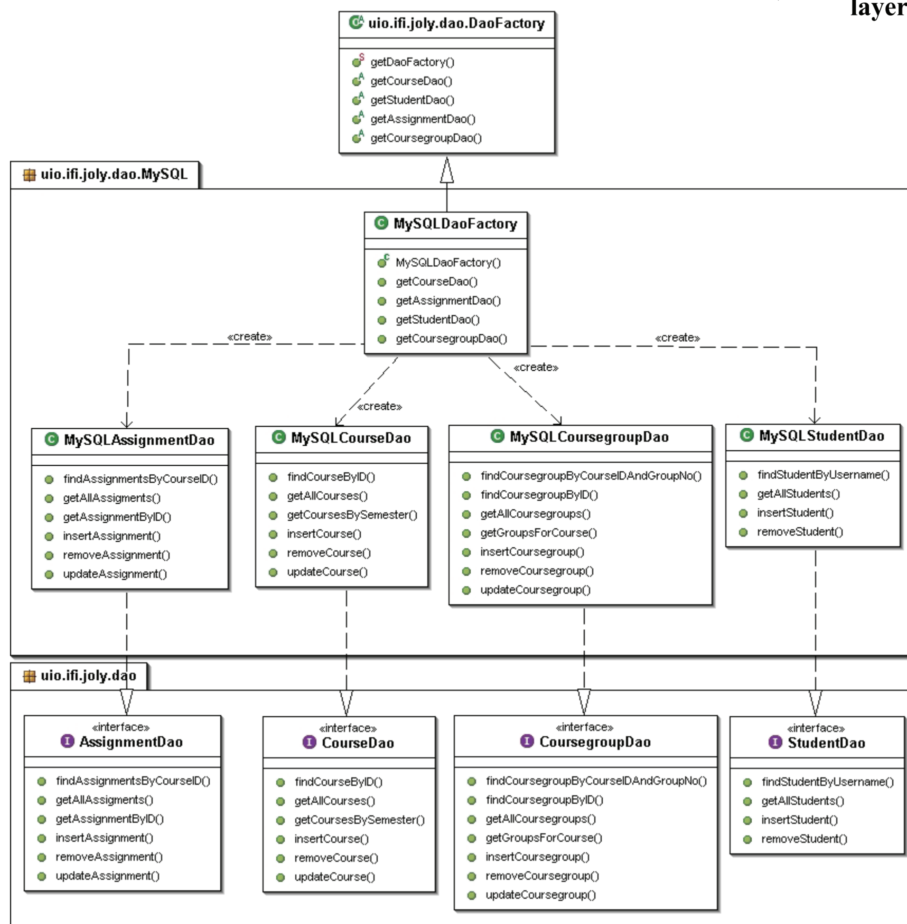


Figure 6.4 The design of the persistence layer (an excerpt of the classes that constitute the persistence layer).

Spring DAO

Spring provides a DAO implementation with abstract DAO base classes. The base classes provide easy access to common database resources. Spring supports following persistence technologies; JDBC, TopLink, iBATIS, JDO and Hibernate (all described below), and there are DAO implementations for each of these.[68]

6.1.4 Data access from Java

Since Java is the chosen programming language, we will in this section consider ways to implement the persistence layer in Java and with free and open source solutions.

Impedance mismatch

When the underlying permanent storage is a relational database management system (RDBMS) (the chosen database management system type for DHIS), there is an impedance mismatch between the object model of Java and the database's relational model. There are several differences between the two models that result in the mismatch [69]:

- Objects cannot be saved or retrieved directly from a relational database. An object has, in addition to data, also identity, state and behaviour, whereas a RDBMS only stores data.
- Objects are traversed using direct references, whereas RDBMSs consist of tables related through primary and foreign keys.
- Object-oriented design "models a business process by creating real-world objects with data and behaviour", whereas the relational modelling goal is to normalize data.
- Existing RDBMS don't have a parallel to Java's object inheritance for data and behaviour.

An object-relational integration therefore requires a strategy for mapping the objects to the relational model, in order for the Java objects to become persistent to the RDBMS.

JDBC API

The most common approach to Java persistence, is for application programmers to work directly with SQL and JDBC. The JDBC API defines a standard way for Java code to save and retrieve data to/from a relational database, and is the industry standard for database-

independent connectivity between the Java programming language and a wider range of databases [70].

Even though the JDBC API is simple to use and hides the complexity of many data access tasks, it can be too low level for application developers to use productively. The application developers still need to do a lot of low-level work, and the work involved in hand-coding persistence for each domain class can be considerable. Another drawback is that when requirements change, manually coded persistence always requires more attention and maintenance effort. [71] Because of this, it can be desirable to use a sophisticated Object/Relational Mapping tool to get a level of automation, and eliminate the need for hand-coding the persistence.

iBATIS

iBATIS SQL Maps is an example of an abstraction tool that work at a slightly higher level than JDBC frameworks, and that automates much of the JDBC-to-SQL mapping. [72] iBATIS is an open source object relational mapper that couples objects with stored procedures or SQL statements using a XML descriptor [73]. Simplicity is the biggest advantage of iBATIS, but it requires writing of RDBMS dependant SQL code. Changing the underlying persistent storage therefore requires rewriting of the SQL Maps, which can take a lot of time.

Because of our database-independency requirement, iBATIS will be insufficient because of the rewriting of mappings that is needed when the underlying persistent storage changes.

Object/Relational Mapping tools

Object/Relational Mapping (ORM) products integrate RDBMS with the object programming language capabilities, and bridges the object/relational impedance mismatch. ORM is performed by a persistence framework. The framework takes automatically care of querying the database to retrieve e.g. Java objects, and persisting of those objects back to their representation in tables and columns in the RDBMS. ORM is done by using metadata that describes the mapping between objects and the database. If ORM tools are used appropriately, it can reduce the development effort and improve maintainability. ORM tools

can abstract database-specific SQL dialects, and thereby improve the portability between databases. As described earlier, database-independency is an important requirement for our application, and because ORM tools can improve this we choose to use one of these tools to implement the persistence layer. [72]

An ORM solution consists of 1) “An API for performing basic CRUD (create, read, update and delete) operations on objects of persistent classes”, 2) “A language or API for specifying queries that refer to classes and properties of classes”, 3) “A facility for specifying mapping metadata” and 4) “A technique for the ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimisation functions” [71].

We will consider ORM tools that implement a full domain model mapping, which maps a whole layer of objects and behaviours to database tables.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) have in recent years been a recommended way of persisting data, but are rapidly declining in popularity. Bean-managed persistence entity beans are prevented from performing efficiently because of design flaws in the EJB 2.0 specification. Container-managed persistence is a marginally more acceptable solution, but it doesn’t represent a solution to the object/relational mismatch. [71]

The EJB 3.0 specification (requires JDK 5.0) is developed by Sun. It introduces a new lightweight Plain Old Java Object (POJO) persistence model for Java Enterprise Edition, and is modelled after successful approaches to ORM like Hibernate, JDO and TopLink (described next). In EJB 3.0, the ORM is done using Java metadata annotations and/or XML.

The EJB 3.0 specification has not been released yet, but since the EJB 3.0 public draft was released, it has gotten a lot of attention and a preliminary support in JBoss (and their persistence framework, Hibernate).

Oracle TopLink

TopLink is one of the most mature ORM products, and was first developed in 1994 for Smalltalk, but has since 1997 had its main focus on Java. Even though TopLink is now owned and supported by Oracle, it works with any major database. Because TopLink is a commercial product licensed under Oracle's OTN[6] licence, it will not meet our requirement for products to be free and open source and is therefore not of current interest.

Java Data Objects

Java Data Objects (JDO) API is a standard interface-based Java model abstraction of persistence. JDO is a specification developed by Sun, and is implemented by multiple vendors, both commercial and non-commercial. JDO is used to directly store Java domain model instances into a persistent store (database). [74]

The specification requires that vendors provide the JDO query language, named JDOQL. An application that uses JDO therefore becomes database independent, because it doesn't have to use the query language of a specific database. JDO also provides an application that uses it, with portability to run on multiple implementations, since it is a Java API.

Some of the open source implementations of JDO are JDOInstruments, Speedo, Jakarta OJB, Triactive JDO, XORM and JPOX. All these implementations support JDOQL and a variety of the major RDBMSs on the market today.

The JDO specification will not include the EJB 3.0 specification, and they will be two separate persistence APIs. Because of this, Sun expect that JDO developers and vendors over time will shift their focus to the new persistence API, EJB 3.0 [53].

Hibernate

Hibernate is a free and open source object/relational persistence and query framework, and provides an ODMG 3 [7] interface alongside a custom API. Hibernate is released under the Lesser GNU Public License, and can thus be used in both commercial as well as free and

[6] Oracle Tecnology Network

[7] ODMG 3 is a specification developed by the Object Data Management Group. ODMG 3.0 is a portability specification designed to allow for portable applications that can run on more than one product.

open source applications. Hibernate only provides mapping to relational databases, but supports all major RDBMS. The framework can use native SQL, but also provides a rich query language (Hibernate Query Language - HQL) that bridges the object and relational model. Hibernate is currently the most popular ORM solution for Java. [75]

Hibernate 3.0 is Hibernate's implementation of EJB 3.0. By combining Hibernate 3 with Hibernate Annotations, "developers can achieve an EJB 3.0 style of programming outside of the EJB 3.0 container and within standalone Java applications" [76].

Hibernate has a flexible mapping that can be declared with XML-mapping files, or with annotations. The framework provides transparency through reflection and runtime byte-code generation, and SQL generation occurs at system start-up time, something that ensures that it does not impact upon IDE debugging and incremental compile. In addition to ORM, the framework also includes other features such as transaction management and concurrency control.

JDO or Hibernate?

The table in figure 6.5 shows how the considered ORM solutions are evaluated after the criterions we decided in one of the previous chapters, and that JDO and Hibernate are evaluated as the top candidates.

Product	License	Community	Activity	Documentation	Comments	Candidate
Enterprise JavaBeans	Expected to be included in J2EE	-	-	-	Not released yet.	-
Oracle TopLink	Commercial – thus not evaluated	-	-	-	-	-
Java Data Objects	Multiple vendors	Vendors have active communities	Stable versions are released continuously	Reference manuals Web forum Commercial	Difficult to find an appropriate vendor implementation	**X**
Hibernate	LGPL	Active and large community	Last stable version 3.1.3 released 20.03.06	Reference manual Web forum Commercial	A very popular ORM framework	**X**

Figure 6.5 Evaluation of ORM solutions after the criterions for evaluating FOSS.

JDO and Hibernate both have a persistence framework with ORM that meets our requirements, that is, database-independency and they both have free and open source implementations.

One of the drawbacks by choosing to use JDO is that it is implemented by so many vendors, and it can take some time to evaluate the implementations that are most suited for an free and open source application. Hibernate only has one vendor. Another drawback with JDO is that it won't implement the new EJB 3.0 specification, and it's expected to be replaced with the new standard for persistence in Java over the next years. Hibernate is at this point the first persistence framework that implements EJB 3.0. Because of all this, but mainly because DHIS uses Hibernate, we chose Hibernate as the persistence framework for Joly.

6.2 Domain layer

The responsibility of the domain layer is to provide a structured model for application data, and to provide procedures for performing domain logic [50]. Even though Joly is a web application, it's important that the infrastructure is not web centric, so that we are not tied to one specific type of presentation implementation. All domain specific operations are encapsulated in the domain layer and made accessible to the presentation layer with a single entry point – the JolyFacade. The application data is represented in JavaBeans; domain objects to be used by all layers of the application.

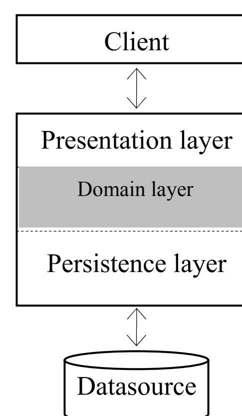


Figure 6.6 Three layer architecture.

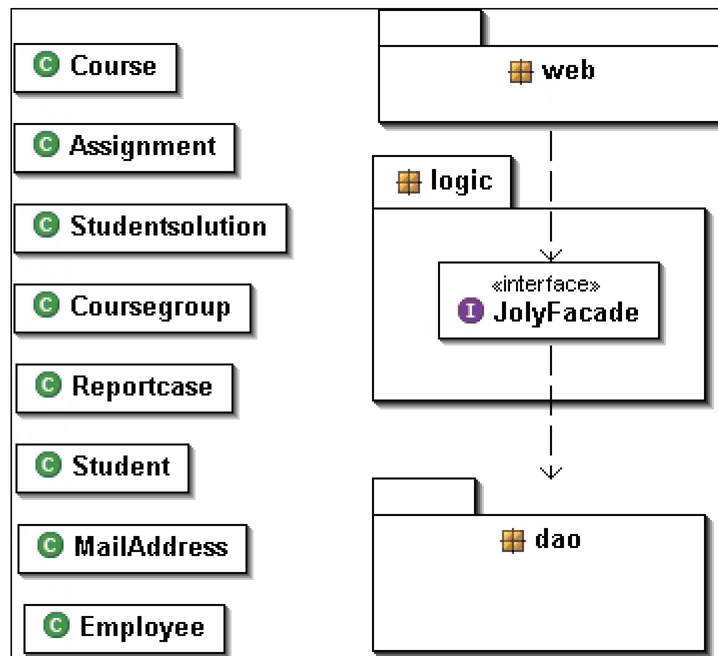


Figure 6.7 Domain objects to the left and layer packages to the right.

6.2.1 Decoupling domain logic

The algorithm for comparing similarity is written by another developer, thus the design of the system needs to support the implementation of the algorithm in a flexible manner. There should be a well-defined interface to the algorithm, and a separation of the logic so that the algorithm can be developed side by side with the system, and also enable the system to easily adopt new versions of the algorithm. It is likely that such an algorithm will be subject to frequent optimizations along side with the development of the rest of the system. We also wish to enable the system to easily adopt new versions of algorithms after the version 1.0 of Joly is put into production. To accomplish this, we have used the Façade pattern that supports good separation of logic and low coupling, and enables us to have pluggable business logic. The Façade pattern provides a single point of contact to a subsystem – a façade object that encapsulates the subsystem. This façade object is an interface that is responsible for collaboration with the subsystem. This is also useful if there are to be more than one implementation of the comparison algorithm. Then the façade would be a unified interface to a set of disparate implementations. [45].

By the rules of the Façade pattern we have defined an interface, ProgramProcessingFacade, which interacts with the subsystem encapsulating the comparison algorithm, and returns the result to the caller.

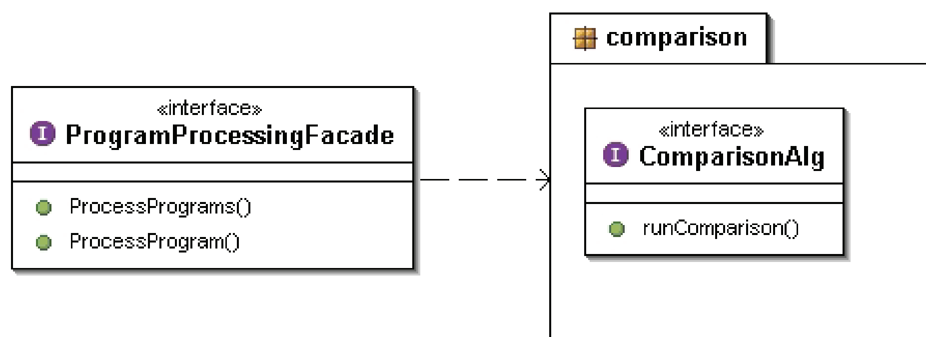


Figure 6.8 ProgramProcessingFacade encapsulates the comparison subsystem.

The Façade pattern has also been applied to the JolyFacade to achieve the single point of entry to the domain logic. The JolyFacade is responsible for all transactional operations, assembling all operations needed to perform one transaction for the client, so that the client only needs to call one method in the JolyFacade to perform a specific task. The implementation of JolyFacade is responsible for collaborating with the various components in the domain logic – the program processing and the send mail logic, as well as initiating methods for retrieving persistent objects for the client and making client instantiated domain objects persistent.

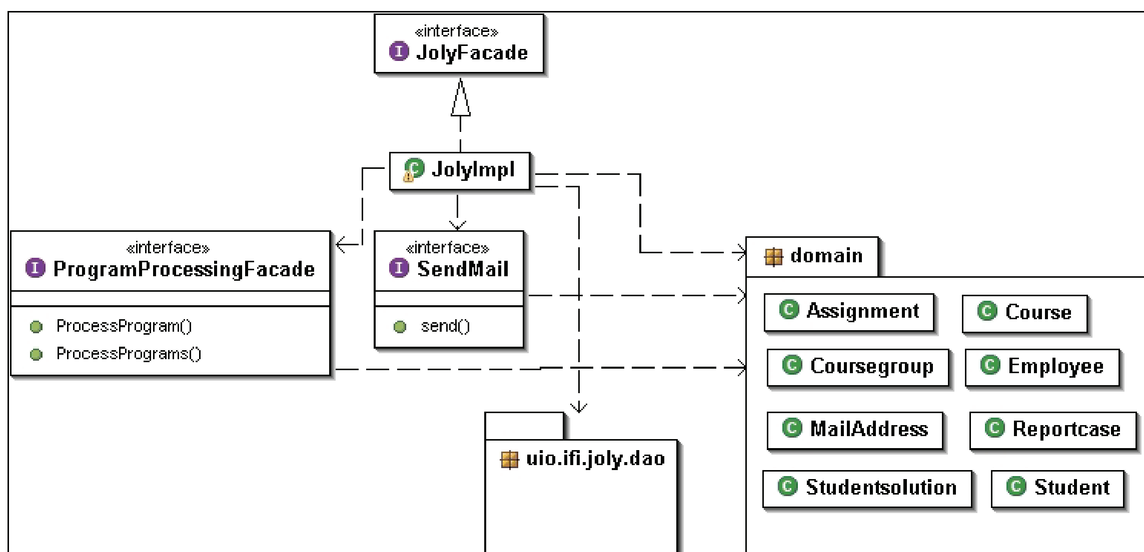


Figure 6.9 JolyFacade is the façade and single entry point to all domain logic.

The routine for sending mail is decoupled with the use of an interface named SendMail. The reason for decoupling this from the overall domain logic, is that we have implemented a very specific mail sending logic that uses the Spring Mail class library to abstract the complex JavaMail API. This strongly ties the implementation of SendMail to the Spring framework's

Mail library, and without a clearly separation thru the interface, the adaptation to other implementations would have been difficult.

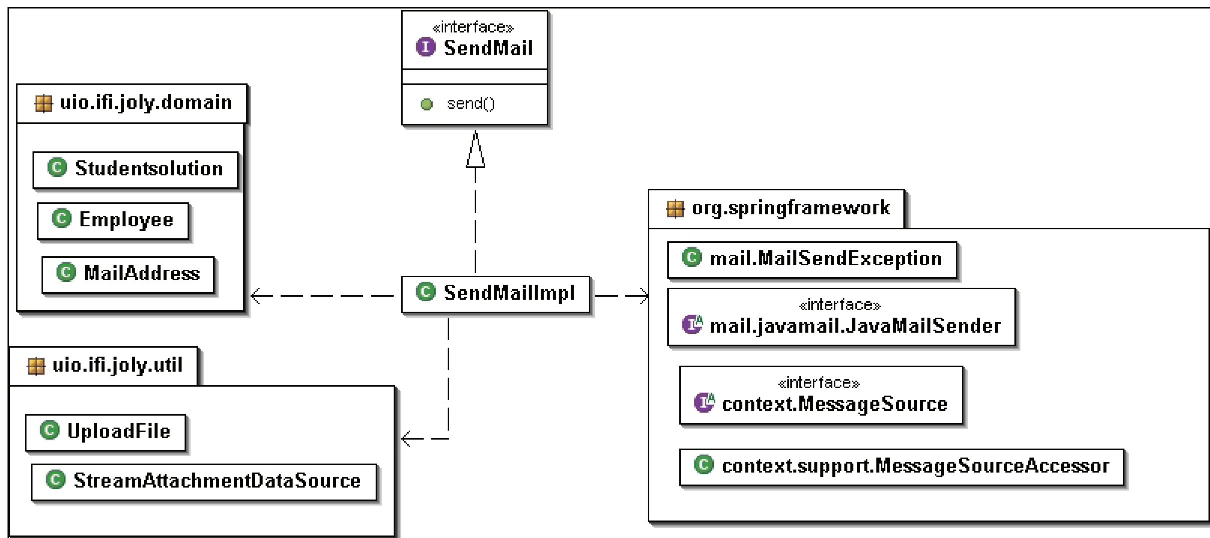


Figure 6.10 The send mail logic is decoupled by using interface – SendMail.

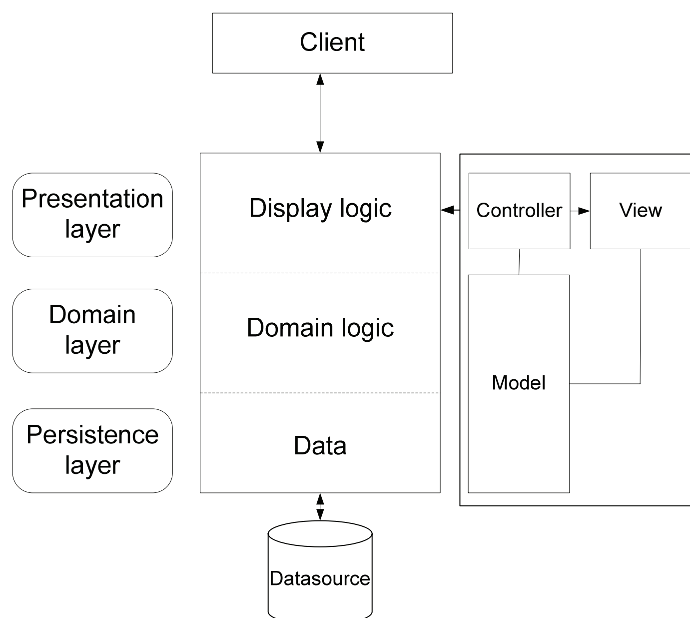
6.3 Presentation layer

This part of the chapter presents how we have designed the presentation layer. We will describe today's standard approach when designing presentation layers in web applications, and what tools can be used to ease the development and force a consistent design.

6.3.1 The Model-View-Controller pattern

The most popular design pattern applied to web applications is the Model-View-Controller (MVC) pattern. This is also the pattern recommended in the Java BluePrints Patterns Catalog[77] for interactive applications. The presentation layer in our web application requires more structuring to decouple the view technology and to handle the specific user interaction logic that a web application needs, namely the request/response interaction with the client. We apply the MVC pattern to the presentation layer to resolve the design problem of the system having a web interface.

The MVC pattern uses the object-oriented design principle to modularize the application into *view* components, *controller* components and *model* components. The result can easily be applied to the three-layer architecture of our application and creates an overall map of the application [50].



A model object contains the data to be displayed in a view and the methods that can be applied to the data. In other words, the model represents the applications data in the form of domain objects, and it represents the operations which apply to those objects. The view object uses the model object's methods to obtain data and then displays them in a specific way. The specification of how the data is represented is the responsibility of the presentation layer, thus the view is a part of the presentation layer and the model is its reference to the domain layer. The controller object holds the knowledge of how the user interacts with the view and manipulates data in the model. It examines the request from the user and translates it into the appropriate actions to be performed by the model [78]. The controller decides which view to display in accordance to a user's interactions, and is therefore a part of the presentation layer. Together these modules encompass the web functionality of the system, thus we continue with using the term *web layer* as the reference to our specific implementation of a presentation layer.

6.3.2 Web MVC frameworks

The traditional implementation of the MVC pattern was not targeted at web applications, but at general GUI implementation without http processing. When MVC was applied to web applications it was often referred to as *Model 2* pattern to set it apart from the traditional MVC implementation. Recently, as the MVC pattern has become the standard web application pattern to use and is the basis for many web frameworks, the term *web MVC* is used when applying the pattern to web applications [54].

An implementation of the web MVC must provide a *pull model*, in contrast to the traditional MVC pattern's *push model* where the Observer pattern is used to propagate changes in the model to the views. The limitations of http require the view to pull data from the model on every request. When a request has been issued, the server updates the model and view and sends it back as a response. The push model is the basic feature a web MVC framework provides, in addition to other features for abstracting the http nature.

A common problem in web applications is the mismatch between html tags and Java objects. This makes the handling of dynamic data and structure in web applications difficult. Since html is string oriented and not object oriented, we need to translate the strings passed by http into the appropriate objects the application uses at the server side. The core Spring framework (or other frameworks that cover the basis of an application) does not offer an abstraction of the request/response nature of http. But by using a web framework we can set the properties of an object to the values gathered using an html form. We can also populate a pure html form with the data contained by an object. The translation of the strings passed by http to objects is handled seamlessly by the framework, enabling us to focus on the functional aspects of the user interaction.

When considering web frameworks based on the MVC pattern, there is a lot to choose from. Most of them are also free and open source projects so this has left us with quite a few to consider. Our main criterions was that it could easily be integrated with the Spring framework we had already chosen to use. It was also important to us that the framework had an active community, indicating the continuance of the framework. Also important was the documentation available, including code examples and tutorials to avoid a steep learning curve. These criterions narrowed our focus down to the most popular web MVC frameworks available; Struts, Tapestry, WebWork, JSF(MyFaces) and Spring MVC. To fairly compare these frameworks we would have to implement them all, since the factors differentiating them are small and they all have their pros and cons. We have reviewed the pros and cons and read multiple reviews of each framework, articles and discussions by experienced developers comparing the frameworks to each other, but the discussion seems to go on and the developers don't even agree whether the cons really are cons or in fact pros. [79, 80]

Our solution was to use the Spring MVC, since it is already included in the Spring framework and the easiest to integrate, and compare it to the WebWork framework which is the choice of the development team of DHIS.

WebWork supports JSP, Velocity and FreeMarker as view technologies, whereas Spring MVC in addition supports XSLT, Jakarta Tiles, JasperReports, Excel spreadsheets and PDF. This is not necessarily a restriction of view choices in WebWork because Velocity and FreeMarker kan generate most types of files, but with Spring's *view resolvers* one can produce dynamically generated binary content within Spring. Although the Joly application currently only needs JSP support, we recognize the extended view possibilities as a benefit for DHIS, seeing that they need view technologies suitable for reporting, like Excel and PDF documents.

Both frameworks offer a validation feature – a way to check the correctness, or validity, of the domain objects and their properties. This can be used by the classes responsible for handling user input, to check that the entered data is correct and corresponds to the property types. The idea of the framework handling this, is to configure each class that needs validation externally, and let the framework handle the validation before the objects reach the methods which are responsible for performing operations with these objects. This assures that all objects are valid before being used for e.g. updating of a database table, and that domain logic is not mixed with validation logic. WebWork's validation is borrowed from another OpenSymphony project called XWork, which is distributed with WebWork's latest release bundle so there is no need for a separate download to use the validation feature. XWork validation can be used for advanced validation and offers a distinction of validation in the different contexts the application may be used, thus not tying the validation to http request processing. This advanced feature is not needed by the Joly application, but can be useful in a more extensive system as the DHIS application. WebWork also supports another simpler approach, where the developers can specify a validate method and programmatically specify the validation rules and have WebWork call this method. The Spring approach to validation is very similar to WebWork's. Spring offers a pluggable validation mechanism based on its own Validator interface and has in addition built in support for Commons Validator, which is an xml-based declarative approach to validate objects. Commons Validator is seen as the most mature validation framework in the open source community [54], and offers more advanced

validation features than the Joly application currently needs, thus the Validator interface is sufficient for our needs just as WebWork's validate() method would have been.

Both frameworks support the Inversion of Control principal. WebWork is originally built on top of XWork for handling IoC container management, but as of WebWork 2.2 the XWork container has been deprecated, and the WebWork development team recommends using Spring for IoC management [81]. This means that WebWork can integrate with Spring, and use Spring's Bean Factory for dependency injection and configuration of web layer beans. In the same way, Spring MVC integrates with the Spring Bean Factory, but since it's a module of the Spring framework it integrates more seamlessly than WebWork, and dependencies from the web layer to the domain layer are easier to configure. The domain and persistence layer beans are available in the ApplicationContext of the Spring Context module, and the web layer beans are available in the WebApplicationContext of the Spring Web module. The WebApplicationContext inherits the ApplicationContext, so it has direct access to the domain and persistence layer beans defined there.

Spring in general has proven to be a very flexible framework, allowing developers to use the framework as they see fit and there are many alternative solutions to a common problem. This flexibility is however not always desirable. For instance will the many alternatives lead to the need of a good understanding of the differences between the alternatives, to make the appropriate choice. WebWork is a counterpart to Spring in this area. WebWork's simplicity makes it easier to start using faster because there are fewer approaches to solve common problems, but then again, there might not be a satisfactory solution to a particular problem one may encounter.

The most significant difference between Spring MVC and WebWork is the way they handle http requests. Controllers in WebWork must supply the execute() method which the framework calls when it receives a http request. This method is called regardless of the type of http request sent, which means that there is no distinction between POST and GET methods. It is therefore up to the developer to figure out the type of request and call the necessary methods from the execute() method. POST and GET is what normally distinguishes a request for a simple web page, from a submission of an html form. Spring MVC has different controller types for processing of POST and GET. The base Controller interface can be implemented for GET processing to generate a web page, and the implementation of

SimpleFormController can be extended to handle submission of html forms. The Joly application is made up of several web page requests and html form submissions, so the Spring MVC framework is more suitable for us since it offers more support for these typical web application workflows than the WebWork framework does. However, a developer experienced in web application development could prefer the WebWork controller, because it does not force a specific implementation, but gives him all the control of how a request is to be processed.

Product	License	Community	Activity	Documentation	Comments	Candidate
Spring MVC	Apache License 2.0	Large and active community.	Last stable version 5.2.7 released 27.02.06	Reference manuals Web forums Mail lists Commercial	Integrated with Spring framework	X
WebWork	OpenSymph-ony Software License, Version 1.1	Active and large community.	Last stable version 2.2.2 released 23.03.06	Reference manuals Web forums Mail lists Commercial	Integratable with Spring. Version 2.3 will be the same as Struts Action 2.	X
Struts	Apache License 2.0	Active and large community	Stable version 1.3.2, released 22.03.06	Reference manuals Mail lists Web forums Commercial	Has forked into two frameworks; Action (to be WebWork 2.3) and Shale.	X
Tapestry	Apache License 2.0	Active community	Last stable version 3.02, released	Reference manuals Commercial Mail lists		X
MyFaces	Apache License 2.0	Active community	Version 1.1.1 released 27.10.06	Reference manuals Mail lists	the first OS JSF implementation.	X

Figure 6.11 Evaluation of web frameworks after the criteria for evaluating FOSS.

This discussions has lead us to the conclusion that it is more important to design the application so that the web layer can be implemented with other frameworks than the one we choose, than to try to find the perfect framework for our application. Developers most often choose their framework based on their past experience and preferences, but since we started this project without any relevant experience in web frameworks, the choice of Spring MVC, being a very up-and-coming and popular framework, seemed good.

6.4 Summary

In this chapter we have described the detailed design decisions of the various layers of the Joly application.

The main focus of the persistence layer design has been to encapsulate the creation of persistence objects. This is done in order to decouple the data access logic from the application logic, to enable easy change of persistent storage. Well-known design patterns, as well as free and open source tools, have been used to resolve this common task.

When designing the domain layer, we have focused on the functional requirements, as well as the application's ability to support a plugin architecture for component based domain logic. We have used patterns to solve design problems related to the encapsulation of subsystems, which in our case is needed to decouple the comparison algorithm written by another developer. We have also described how we have enabled the domain logic to be a separate part of operational units that can be accessed from all type of layers, not just web layers.

The presentation layer of a web application needs more logic than a traditional non-web application, because of the http nature of the user interaction. To solve this, we applied the Model-View-Controller pattern and employed an open source web framework for the implementation of typical web workflow scenarios.

Chapter 7

Implementation of Joly

We will in this chapter give a more in-depth description of the implementation of the three layers of the Joly architecture, in addition to the implementation of the database, to present the decisions that were made during the development of Joly to meet the functional, as well as non-functional, requirements for the web application.

7.1 Database

As described in the previous chapter, we chose to use MySQL to implement the database. MySQL supports several storage engines. The storage engines act as handlers for different table types, and MySQL provides both transaction-safe table handlers and non-transaction safe table handlers. Transaction safe means that the transactions are ACID[1] compliant. Transaction safe tables has several advantages over non-transaction safe tables; a rollback can be executed to ignore the changes made before a commit, if an update fails, the made changes are reverted, if the database crashes, the data can be recovered either by an automatic recovery or from a backup plus the transaction log, and they can provide better concurrency for tables that get many updates concurrently with reads [82]. Because we want transactions in the database to comply with the ACID model for transactions in a database, we chose to use a transaction safe table handler. The InnoDB and BDB (Berkeley DB) storage engines are both transaction safe table handlers for MySQL. The BDB storage engine aren't supported by as many operating systems as InnoDB, and for that reason we chose to use the InnoDB storage engine.

[1] ACID = Atomicity, Consistency, Isolation and Durability

The designed database schema is in BCNF[2] (there are no multi-value dependencies in the schema), i.e. that all determinants in a table is a candidate key. A description of the database is given in appendix A. Because it was easier to implement primary keys that consisted of one column in the primary key with Hibernate, we chose to add a single unique primary key to some of the tables that had composite primary keys. This had however no effect on the normal form of the affected tables.

Platform-independency is a requirement for our application, and this also includes database-independency. It must therefore be easy to switch the underlying database, without having to make a lot of changes to the persistence layer. Hibernate makes this change easy. The only changes needed to be done to the persistence layer to change a database, is to change the settings of the JDBC the connection provider uses, and the Hibernate dialect (all described in the following part of the chapter). The Hibernate dialect represents the dialect of SQL implemented by a particular RDBMS. For MySQL with an InnoDB storage engine, the dialect is set to *org.hibernate.dialect.MySQLInnoDBDialect.* To make it easy for an administrator of Joly to change the settings without having to make changes in the code, we have provided a *jdbc.properties* file that defines the JDBC-properties and the Hibernate dialect. A property resource configurer provided by Spring, *PropertyPlaceholderConfigurer*, is used in the application's context to pull the values from the properties file into it's appropriate bean definition (see section 7.3.2 in this chapter for a more in-depth description).

7.2 Implementation of the persistence layer

The DAO pattern is used to abstract and encapsulate all access to the data source in the persistence layer, and it can conceal the persistence technology specifics from the above layers. The DAO interfaces should be designed in a way that shields the calling code from changes, preventing lock-in to a particular ORM framework and making the process of adapting to changes in the framework, or new technology, easier.

We don't expect that the data access concept will change in the next few years, because it is a widespread pattern and the new persistence specification EJB 3.0 (described in the previous chapter) that is soon to be released supports it. But what we can expect, is that the

[2] BCNF = Boyce-Codd Normal Form

technologies that implement the DAO will change. One of the changes that we already can foresee, is that many of the ORM frameworks will implement the new specification. Hibernate has already implemented an early draft of EJB 3.0. Thus we don't anticipate that it would require a lot of changes to the existing code in the persistence layer of Joly, if we were to change the current used version of Hibernate to one that fully implements the EJB 3.0 specification.

In the years to come it might also be desirable to swap out the used ORM framework with a new one. This might occur because Hibernate is phased-out or for some reason is not suitable for the application anymore, or because it might be another ORM framework that would ease further development.

These are just two of the possible changes that can appear in the persistence layer and that can have an impact on the application's operating time. There are of course changes that we can't foresee or plan for, but having an isolated persistence layer that shields changes for the rest of the application would make the adaptation to these changes easier, because it won't affect the above layers in the architecture.

7.2.1 Hibernate architecture

Hibernate consists of several objects that form its architecture (figure 7.2.1) [83]:

- A SessionFactory is a thread-safe cache of the compiled mappings for a single database. It is a factory for Session and a client of ConnectionProvider, and might hold an optional second-level cache of data that is reusable between transactions.
- A Session is a single-threaded, short-lived object that represents a conversation between the application and the persistent store, and constitutes the first-level cache. It wraps a JDBC connection, and is a factory for Transaction.
- Persistent objects are ordinary POJOs that are short-lived, single-threaded objects containing persistent state and business function. They are associated with a Session, and are detached and free to use in any application layer after the Session is closed.
- Transient objects are instances of POJOs that are not currently associated with a *Session*, either because they have been instantiated by the application but not yet persisted, or they may have been instantiated by a closed Session.

- A Transaction is a single-threaded, short-lived object that is used by the application to specify atomic units of work. It is optional to use Transaction.
- A ConnectionProvider is a factory and a pool of JDBC connections for use by a *Session.*
- The TransactionFactory is a factory for Transaction instances. It is optional to use TransactionFactory.

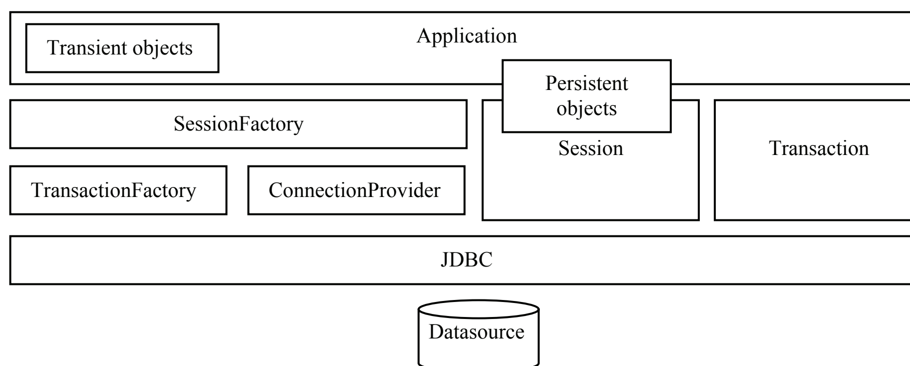


Figure 7.2.1 Hibernate achitecture.

Because we use Hibernate in combination with Spring, the TransactionFactory and Transaction parts of the architecture isn't used, because Spring handles the transaction management for Hibernate. The reason for why we chose to use the two frameworks in combination is described below.

7.2.2 Connection pool

The version of Jetty we have used under development provides a non-managed environment for Java web applications, i.e. that the application itself must manage database connections and demarcate transaction boundaries. When using Hibernate in a non-managed environment, it is responsible for handling the transactions and obtaining JDBC connections. It is advisable to use a JDBC connection pool for handling the connections, because it can be expensive to acquire a new connection and maintain many idle connections. A non-managed environment like Jetty doesn't provide its own connection pool, and must therefore rely on a third-party library like the free and open source C3P0 connection pool. Hibernate defines a plugin architecture that allows for easy integration with any JDBC connection pool through the ConnectionProvider. Hibernate acts as a client of the pool – relieving the application code of handling the database connections (see figure 7.2.2). [71]

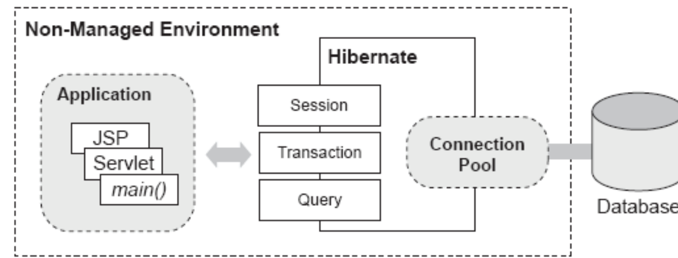


Figure 7.2.2 Hibernate with a connection pool in a non-managed environment [84].

A benefit of not using a third-party connection pool instead on one that is provided by a web container, or an application server, is that its configurations won't need changes if the applications web container or application server is changed.

7.2.3 Hibernate Query Language - HQL

Hibernate provides three ways of retrieving objects from the database;

1. Native SQL queries.
2. Query By Criteria (QBC) and Query By Example (QBE) using Query API.
3. Hibernate Query Language (HQL).

The Hibernate Query Language is a full object-oriented query language, much like the relational query language SQL. But unlike SQL, HQL isn't a data-manipulation language – it can only be used for retrieval of data from the database. All queries that can be expressed in SQL can be mapped to HQL, but HQL has in addition all the attributes of an object-oriented language which includes polymorphism, inheritance and association. HQL is the preferable way to retrieve objects when Hibernate is used in the persistence layer, because it facilitates writing of database-type independent queries. Hibernate will at run-time generate the appropriate SQL dialect of the underlying database, send it to the database and then populate the object with data. The usage of HQL contributes to making the persistence layer database independent, and for that reason we chose to use it for writing queries.

7.2.4 Required Hibernate libraries

Hibernate has a number of third-party libraries it depends on, which are all distributed together with Hibernate. We notice that Hiberante depends upon at least eight subsystems to function, and an additional two subsystems if the application doesn't employ an application

server, as is the case for Joly. We will in the next chapter consider the implications of being dependant on a number of third-party subsystems. The following are the libraries Hibernate requires, and that we have used under the development of Joly:

- **antlr-2.7.6rc1.jar**: ANother Tool for Language Recognition (patched with proper context classloading).
- **asm.jar**: ASM bytecode library.
- **c3p0-0.9.0.jar**: C3P0 JDBC connection pool.
- **cglib-2.1.3.jar**: CGLIB (Code Generation LIBrary) bytecode generator.
- **commons-collections-2.1.1.jar**: Commons Collections.
- **commons-logging-1.0.4.jar**: Commons Logging.
- **dom4j-1.6.1.jar**: XML configuration and mapping parser.
- **Ehcache-1.1.jar**: cache required if no other cache provider is set.
- **hibernate3.jar**: Hibernate core.
- **jdbc2_0-stdext.jar**: Standard Extension JDBC APIs (required for standalone operation, outside application server).
- **Jta.jar**: standard JTA (Java Transaction API) API (required for standalone operation, outside application server).
- **log4j-1.2.11.jar**: Log4j Library (not required) .
- **xml-apis.jar**: standard JAXP (Java API for XML Processing) API, some SAX (Simple API for XML) parser is required.

Because Hibernate uses JDBC to connect to the database, it is necessary with a JDBC driver. We chose to use the official JDBC driver for MySQL, MySQL Connector/J:

- **mysql-connector-java-3.1.11-bin.jar**

To use Hibernate Annotations 3.1 with Hibernate core, the following additional libraries are required (distributed along with Hibernate Annotations):

- **hibernate-annotations.jar:** Hibernate Annotations.
- **ejb3-persistence.jar:** EJB3 persistence.

7.2.5 Object/Relational Mapping

Hibernate uses EJB3 entity beans (they are Plain Old Java Objects (POJOs)) as the programming model for persistent classes. A POJO is a normal Java object that doesn't

implement any special interface or serve any other role, the only requirement is that it has a no-argument constructor. The properties of the POJO represent the fields in a table in the database. The persistent class shields the internal representation of its properties, but provides access through getter and setter methods. All Java JDK types and primitives, including classes from the Java collections framework, can be used as property types. The POJOs constitute the domain model of the application that needs persistence, and Hibernate uses them for persistence and as return objects in queries.

Hibernate (since version 3.0) support two ways for defining the mapping of objects to relations in a database; XML-mapping files and annotations. The persistence layer can consist of both XML-mappings and annotations, but it can't have two declarations for the same class. When using XML-mapping files, each persistent class needs to be mapped with its own Hibernate mapping file, which results in one mapping file for each class that needs to be persistent. With Hibernate Annotation,s the mappings are defined through JDK 5.0 annotations in the POJOs. Usage of annotations saves the developer from writing all the wiring code in the XML-files. It also makes the code cleaner and easier to understand, because the mapping is declared together with the properties of the POJO that it maps.

We will in the following describe how mapping of an object to a relation can be done with both Hibernate Annotations and XML-mapping files, and conclude which strategy is the best solution for Joly, in addition to our experiences with the chosen strategy.

Definition of a persistent class with annotations

Every bound persistent POJO class is an entity bean, and is declared by using the *@Entity* annotation at class level (line 6, figure 7.2.3). The class is then defined as a persistent class. There are two different ways the properties of the class can be accessed; through its getter methods, or directly through its fields. There is no difference between the two access-methods. We have chosen to let the properties be accessed through its getter methods, and defined this by setting the access type of the entity to PROPERTY. Mappings for the properties will because of this be defined at the getter-methods level.

To define which table in the database the persistent class corresponds to, we used the *@Table* annotation (line 7, figure 7.2.3). It is optional to use this annotation, if there isn't defined one

the default class name is used as the table name. We have named the persistent classes and its properties after their corresponding table names and column names. It's therefore not necessary to define them in the mapping, but we have chosen to do so because we feel that the mapping becomes a bit clearer.

```
1   package uio.ifi.joly.domain;
2
3   import java.sql.Timestamp;
4   import javax.persistence.*;
5
6   @Entity(access = AccessType.PROPERTY)
7   @Table(name="Assignment")
8   public class Assignment {
9
10      private int assignmentID;
11      private short assignmentNo;
12      private Course course;
13      private Languagetype languagetype;
14
15      public Assignment() {
16      }
17
18      public Assignment(short assignmentNo, int assignmentID){
19              this.assignmentNo = assignmentNo;
20              this.assignmentID = assignmentID;
21      }
22
23      @Id(generate = GeneratorType.AUTO)
24      @Column(name="assignmentID", nullable=false)
25      public int getAssignmentID() {
26              return assignmentID;
27      }
28
29      public void setAssignmentID(int assignmentID) {
30              this.assignmentID = assignmentID;
31      }
32
33      @Column(name="assignmentNo", nullable=false)
34      public short getAssignmentNo() {
35              return assignmentNo;
36      }
37
38      public void setAssignmentNo(short assignmentNo) {
39              this.assignmentNo = assignmentNo;
40      }
41
42      @ManyToOne(targetEntity=Course.class, optional=false)
43      @JoinColumn(name="courseID")
44      public Course getCourse() {
45              return course;
46      }
47
48      public void setCourse(Course course) {
49              this.course = course;
50      }
51
52      @ManyToOne(targetEntity=Languagetype.class, optional=false)
53      @JoinColumn(name="languageID")
54      public Languagetype getLanguagetype() {
55              return languagetype;
56      }
57
58      public void setLanguagetype(Languagetype languagetype) {
59              this.languagetype = languagetype;
60      }
61
62  }
```

Figure 7.2.3 Excerpt of some of the mapped properties in Assignment.java, mapped with annotations.

There's one other annotation that's required to map a POJO, the *@Id* annotation. This annotation defines which of the class's properties that represents the primary key field. The *@Id* annotation also allows for definition of the identifier generation strategy with the *generatorType* property. Hibernate provides more id generators than the basic EJB3 ones, there are five types [54];

- **AUTO** - either identity column or sequence depending on the underlying DB).
- **TABLE** - table holding the id.
- **IDENTITY** - identity column.
- **SEQUENCE** – sequence generates the id.
- **NONE** - the application has the responsibility to set the id.

The *AUTO* generator is the recommended type for portable applications; the database then has the responsibility to set the primary key. The primary key fields in Joly's database are automatically assigned by the database, and for that reason we used *AUTO* as the generator type for the primary key field (line 23, figure 7.2.3).

@Column is an optional annotation, like *@Table*, and defines which column the property maps to. Some of the properties in the persistent class maps to columns in the database that cannot be empty (NULL), and this can be defined in the mapping by setting the attribute *nullable* of *@Column* to false (line 24, figure 7.2.3).

The POJOs can be associated through relationships/associations. Hibernate Annotations provides three different annotations for mapping a relationship: *@OneToOne*, *@ManyToOne/@OneToMany* and *@ManyToMany*. The mappings for the relationships are declared at the property level. The relationships can be either one-directional or bi-directional, but we have only used one-directional relationships because it reflects the relationships between relations in the database.

In the excerpt of the *Assignment* POJO (figure 7.2.3) we have used *@ManyToOne* annotations to define the POJOs relationships with *Languagetype* and *Course*. The annotation contains information about the target entity it has a relationship with, and whether it can be empty or not. The default value of the target entity attribute is the type of the property that stores the relation, and because our property name is the same as the target entity it isn't necessary, but we declared it to make the mapping clearer. In addition to the defined

attributes for @ManyToOne, it is also possible to define a cascade type for the relationship. We don't want changes in *Assignment* to be cascaded to its related POJO's, and by not declearing a cascade type we achieve that. The *@JoinColumn* annotation is used together with *@ManyToOne* to define which of the properties in the *Assignment* POJO that constitutes the foreign key to the related POJO.

Definition of a persistent class with XML-mapping

When using XML-mapping, it is still necessary with a POJO, and it will look exactly like the POJO for *Assignment* in figure 7.2.3, but without the annotations. The mappings are defined in an XML file with the suffix hmb.xml. Mapping objects to relations with XML-mapping provides the same possibilities and properties as mapping with annotations, but they are defined with XML-tags.

The mapping file use the Hibernate DTD (Document Type Definition) http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd (line 2, figure 7.2.4). The DTD file is included in *hibernate3.jar* and Hibernate will look it up from the classpath. The declaration of the mapping information must be between the two *hibernate-mapping* tags. The *class* tag defines which POJO it uses for mapping, and which table in the database the mapping is provided for (corresponds to the *@Entity* and *@Table* annotations).

The *id* element declares the identifier property with name, column and datatype information. Hibernate will access the property through its getter and setter methods. The identifier generation strategy is specified with the nested *generator* element, and the *native* generation strategy corresponds to the *AUTO* generator used with Hibernate Annotations.

The persistent properties of the class are declared with the *property* tag, and the *name* attribute tells Hibernate which getter and setter methods to use. As with annotations, the *column* attribute to the *property* tag can be skipped if the name of the property and the column name are the same. The *not-null* attribute corresponds to *nullable* attribute of the annotation *@Column.*

Mapping of relationships with *Course* and *Languagetype* is done with the *many-to-one* tag. The definition of which of the properties in the *Assignment* POJO that is the foreign key to

the related POJO is defined by the *column* attribute (corresponds to the *@JoinColumn* annotation). When using annotations, leaving out the *cascadetype* attribute sets the cascading type to none, but with XML-mapping this must be decleared by setting the *cascade* attribute to *none.*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
3      "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
4 <hibernate-mapping>
5    <class name="uio.ifi.joly.domain.Assignment" table="assignment">
6        <id name="assignmentID" column="assignmentID" type="int">
7            <generator class="native" />
8        </id>
9
10       <property name="assignmentno" column="assignmentNo" type="int" not-null="true" />
11       <property name="deadline" column="deadline" type="timestamp" not-null="true" />
12
13      <many-to-one name="course" column="courseID" class="uio.ifi.joly.domain.Course"
14            cascade="none" />
15      <many-to-one name="languagetype" column="languageID"
16            class="uio.ifi.joly.domain.Languagetype" cascade="none" />
17   </class>
18 </hibernate-mapping>
```

Figure 7.2.4 XML-mapping file (Assignment.hbm.xml) with an excerpt of the properties in *Assignment.java*.

Annotations vs. XML-mapping

We are of the opinion that it is easier to maintain a persistence layer where the objects are mapped to relations with Hibernate Annotations. This is because the code is cleaner and the mapping is declared in the persistent classes, and it would therefore be quicker for new developers to understand the mapping between objects and relations. Additionally, we think the use of Hibernate Annotations can reduce development time and errors if the underlying database changes, because all modifications to the mapping of a relation can be done in one place. Annotations are therefore the selected solution for mapping objects to relations in Joly, and should be used in DHIS which has a database consisting of more tables than Joly.

Experience with Hibernate Annotations

Since Hibernate Annotations is a relatively new added functionality to the Hibernate persistence framework, we discovered that there was a lack of extensive documentation. Hibernate provides tools for automatic generation of XML-mapping files from a database scheme, or the other way around, but there aren't provided any such tools for automatic generation of POJOs with Hibernate Annotations mappings. We spent a considerable amount of time at the beginning of the development process to understand Hibernate Annotations and

define the mappings. When we look back we should have evaluated Hibernate more thorough before the development process. The evaluation only involved Hibernate Core, and not the additional Hibernate Annotations. While Hibernate Core can be considered as a mature and stable framework, the additional functionality in Hibernate Annotations still has a way to go, in our opinion, before it can be considered as mature because it e.g. lacks comprehensive documentation.

Even though the amount of time spent on defining the mapping with Hibernate Annotations, as opposed to using one of Hibernates tools for an automatic XML-mapping generation, slowed down the start of the development process, we feel that it has improved the maintainability of the persistence layer (as described above). We will recommend developers new to Hibernate Annotations to start the development process by using XML-mappings, to reduce the start up time and development of the persistence layer - so that the persistence of objects can be tested out as early as possible. For DHIS on the other hand, and other systems that has a database scheme with a large amount of tables to maintain, we recommend to swap out the XML-mappings once everything works. We feel that the amount of time spent on understanding and declaring mappings with annotations will be worth it in the long run, since the maintenance of such mapping seems easier and less time consuming.

7.2.6 Spring and Hibernate

We will in this part of the chapter show the benefits of using Hibernate together with Spring, in contrast to just using Hibernate in the persistence layer. Spring and Hibernate is a popular combination, because Spring provides the developer with an easy way of integrating ORM frameworks that reduces the amount of work needed to be done to implement the persistence layer. Spring offers resource and session management, exception wrapping and integrated transaction management that developers have to manage manually when using non-Spring Hibernate. Spring contributes to isolation of the persistence layer, and makes it easy to replace or combine several ORM frameworks in an application.

Spring’s common data access exception hierarchy can be used instead of implementation-specific exceptions, and is used by wrapping exceptions specific for an ORM framework to a set of unchecked runtime exceptions. The exception hierarchy contributes to isolation of the persistence layer, because the above layer doesn’t need to be concerned with which type of

ORM framework being used. Replacement of the persistence technology will, by using the common data access exceptions, therefore not affect the exception handling in the above layer, and contributes to making it easier to maintain the application in the future.

Spring provides, as described in the previous chapter, DAO implementation support for Hibernate. The support includes convenience classes (named templates) that simplify the implementation of the DAOs with the persistence technology, and can significantly reduce the amount of data access code. Spring's DAO template for working with Hibernate is named *HibernateTemplate*, and can be used standalone or obtained from the DaoSupport parent class – *HibernateDaoSupport*. When using a template, the developer is relieved from manually managing Hibernate *Sessions* because it will automatically be synchronized with Spring transactions.

We will now look at two possible implementations of the *AssignmentDAO* interface (figure 7.2.5) in Joly, one with non-Spring Hibernate and one with Hibernate and Spring to illustrate the benefits of using the two frameworks in combination.

```
1  package uio.ifi.joly.dao;
2
3  import org.springframework.dao.DataAccessException;
4  import uio.ifi.joly.domain.Assignment;
5  import java.util.List;
6
7  public interface AssignmentDao {
8
9      void insertAssignment(Assignment assignment);
10     void updateAssignment(Assignment assignment);
11     List getAllAssigments();
12     void removeAssignment(Assignment assigment);
13
14     List findAssignmentsByCourseID(int courseid);
15     Assignment getAssignmentByID(int assignmentID);
16  }
```

Figure 7.2.5 The DAO interface for AssignmentDao.

Non-Spring Hibernate

As we can see from the example in figure 7.2.6 line 15, an instance of *Configuration* is created. *Configuration* is used to specify properties and mapping documents that are going to be used when creating a *SessionFactory*. By calling *Configuration*'s method *addClass()*, the mapping file for the persistent class is defined. In the DAOs constructor, the *SessionFactory* is also instantiated (line 21, figure 7.2.6), with the properties and mappings in the specified

Configuration. These specifications, and the setup of the *SessionFactory,* could be done in a common superclass, and thus saving some redundant code in the DAOs when using non-Spring Hibernate. The *SessionFactory* is used to open a single-threaded recourse, a *Session*, in the example's method *getSession()* (line 27, figure 7.2.6).

To perform a transaction it is necessary with a *Session*, which offers create, read and delete operations for the instance of the mapped entity class. Each of the methods in the DAO therefore needs to get a *Session,* to be able to perform an operation against the database (as in line 38, figure 7.2.6). To save an instance of an *Assignment*, the method *save()* is used which saves the given instance of the entity class. After saving the new instance to the database, *Session'*s method *flush()* is called. *flush()* synchronizes the persistent store with the persistent state in memory, and must be called before committing the transaction and closing the session (line 41, figure 7.2.6). The last method that needs to be called to complete the transaction is *Session*'s *close()* which disconnects the session from the JDBC connection and cleans up.

To perform a query, the method *createQuery()* of the *Session* interface is used, which takes a HQL query string as input parameter (line 60, figure 7.2.6). Because we only fetch data in the query, the persistent store doesn't need to be synchronized with any persistent state in memory, and we don't have to perform a flush.

Implementation of DAOs with non-Spring Hibernate results in redundant opening/closing of sessions, in addition to a lot of exception handling for Hibernate's specific exceptions. In figure 7.2.6 we only show implementations of an excerpt of methods in the *AssignmentDao* interface, because the whole implementation is 149 lines long and would have covered many pages.

```
1  package uio.ifi.joly.dao.MySQL;;
2
3  import java.util.List;
4  import org.hibernate.*;
5  import org.hibernate.cfg.Configuration;
6  import uio.ifi.joly.dao.AssignmentDao;
7   import uio.ifi.joly.domain.Assignment;
8
9   public class MySQLAssignmentDao implements AssignmentDAO {
10       private static SessionFactory sessionFactory = null;
11
12       public MySQLAssignmentDao(){
13           Configuration cfg = null;
14           try {
15              cfg = new Configuration().addClass(Assignment.class);
16           } catch (MappingException e) {
17              e.printStackTrace();
18           }
19
20           try {
21              sessionFactory = cfg.buildSessionFactory();
22           } catch (HibernateException e) {
23              e.printStackTrace();
24           }
25      }
26
27      private Session getSession() {
28           Session session = null;
29           try {
30              session = sessionFactory.openSession();
31           } catch (HibernateException e) {
32              e.printStackTrace();
33           }
34           return session;
35      }
36
37      public void insertAssignment(Assignment assignment) {
38           Session session = this.getSession();
39           try {
40              session.save(assignment);
41              session.flush();
42           } catch (HibernateException e) {
43              e.printStackTrace();
44           } finally {
45               if (session != null) {
46                  try {
47                      session.close();
48                  } catch (HibernateException e) {
49                      e.printStackTrace();
50                  }
51              }
52          }
53      }
54
55      public List getAllAssignments() {
56           Session session = this.getSession();
57           List list = null;
58           try {
59               Query query =
60       session.createQuery("select Assignment from uio.ifi.joly.domain.Assignment");
61               list = query.list();
62           } catch (HibernateException e) {
63               e.printStackTrace();
64           } finally {
65              if (session != null) {
66                  try {
67                      session.close();
68                   } catch (HibernateException e) {
69                      e.printStackTrace();
70                  }
71               }
72           }
73        return list;
74    }
75 }
```

Figure 7.2.6 An excerpt of MySQLAssignmentDao *without* Spring's DAO support.

Hibernate with Spring

When Hibernate is used together with Spring it is usually set up via Spring's *LocalSessionFactoryBean* in the application context, as shown in figure 7.2.7. The bean refers to the JDBC *DataSource* it uses (also set up in the application context), and which Hibernate mapping files to load. The Hibernate mapping files are defined in *hibernate.cfg.xml* along with other Hibernate properties (this is the case for both non-Spring Hibernate, and Hibernate in combination with Spring), thus it needs to be referenced through the *configLocation* property. Because the mapping between objects and relations is done by Hibernate Annotations, the property *configurationClass* has to be set to Hibernate's *AnnotationConfiguration* (line 12, figure 7.2.7). When Hibernate is used in combination with Spring, the transaction management is delegated to Spring's generic transaction facilities. A *HibernateTransactionManager* is defined in the application's context (not shown in figure 7.2.7) that allows transaction execution on a single *SessionFactory*, and the Hibernate-based DAOs can then seamlessly participate in such transactions.

The DAO receives a reference to the Hibernate *SessionFactory* instance via a Spring bean reference (line 27, figure 7.2.7), and can then be used by the *HibernateDaoSupport* base class.

```
1   <bean id="sessionFactory"
2      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean" abstract="false"
3      singleton="true" lazy-init="default" autowire="default" dependency-check="default">
4
5      <property name="dataSource">
6             <ref bean="dataSource" />
7      </property>
8      <property name="configLocation">
9             <value>/WEB-INF/hibernate.cfg.xml</value>
10     </property>
11     <property name="configurationClass">
12            <value>org.hibernate.cfg.AnnotationConfiguration</value>
13     </property>
14     <property name="hibernateProperties">
15            <props>
16                   <prop key="hibernate.dialect">${hibernate.dialect}</prop>
17                   <prop key="hibernate.hbm2ddl.auto">validate</prop>
18            </props>
19     </property>
20   </bean>
21   ....
22   <bean id="assignmentDao" class="uio.ifi.joly.dao.MySQL.MySQLAssignmentDao"
23     abstract="false" singleton="true" lazy-init="default" autowire="default"
24     dependency-check="default">
25
26     <property name="sessionFactory">
27            <ref bean="sessionFactory" />
28     </property>
29   </bean>
```

Figure 7.2.7 Excerpt of setup of Hibernate in applicationContext.xml.

The example in figure 7.2.8 extends Spring's DaoSupport parent class, *HibernateDaoSupport* (line 11, figure 7.2.8). *HibernateDaoSupport* takes a *SessionFactory* instance as a bean property, and provides a *HibernateTemplate* based on the *SessionFactory* to its subclass. The template works with Hibernate's *Session* API underneath. It is therefore not necessary to manually handle transactions because the Hibernate *Session*'s will automatically be synchronized with Spring transactions, which takes care of opening and closing a session. *HibernateTemplate* has basically the same operations as *Session* (described above), but throw Spring's *DataAccessException* (in Spring's common data access exception hierarchy) instead of Hibernates checked *HibernateException*.

```
1  package uio.ifi.joly.dao.MySQL;
2
3  import java.util.List;
4
5  import org.springframework.dao.DataAccessException;
6  import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
7
8  import uio.ifi.joly.dao.AssignmentDao;
9  import uio.ifi.joly.domain.Assignment;
10
11 public class MySQLAssignmentDao extends HibernateDaoSupport implements AssignmentDao{
12
13     public void insertAssignment(Assignment assignment)throws DataAccessException {
14             getHibernateTemplate().save(assignment);
15     }
16
17     public void updateAssignment(Assignment assignment)throws DataAccessException {
18             getHibernateTemplate().update(assignment);
19     }
20
21     public List getAllAssigments()throws DataAccessException {
22             return getHibernateTemplate().loadAll(Assignment.class);
23     }
24
25     public void removeAssignment(Assignment assignment) throws DataAccessException {
26             getHibernateTemplate().delete(assignment);
27     }
28
29     public List findAssignmentsByCourseID(int courseid)throws DataAccessException {
30             return getHibernateTemplate().find("from Assignment where courseID=?",courseid);
31     }
32
33     public Assignment getAssignmentByID(int assignmentID)throws DataAccessException {
34             List list =
35          getHibernateTemplate().find("from Assignment where assignmentID=?", assignmentID);
36
37             if(list.isEmpty())
38                     return null;
39
40             return (Assignment)list.get(0);
41     }
42 }
```

Figure 7.2.8 MySQLAssignmentDao *with* Springs DAO support.

Non-Spring Hibernate vs. Hibernate used togheter with Spring

As we can see from figure 7.2.8, the implementation of the *AssignmentDao* interface covers 42 code lines; this is 107 less code lines than a full implementation of the DAO without using Spring's support. The code is also less messy because the template handles opening and closing of sessions. These two issues are the reason we chose to use Hibernate in combination with Spring to implement the persistence layer in Joly. It was easy and quick to get Hibernate to function when the Spring support was used, and it has saved development time. We feel that the use of Hibernate together with Spring also has improved the maintainability of the persistence layer, because when dealing with changes, less code needs change. Spring's inversion of control approach also makes it easy to swap the implementation and configuration of the Hibernate *SessionFactory* instances and the transaction manager.

7.3 Implementation of the domain layer

This section covers the implementation of the domain layer. We will go into the details of the specific implementations of the design decisions explained in chapter 6.

7.3.1 Domain objects

The domain objects are defined by classes at the domain layer, but they are implemented so that they can be used by all layers of the application. This is achieved by implementing them as JavaBeans (POJOs), meaning they have private properties which are made accessible by public setter and getter methods, and they have one or more constructors for initialization. The domain objects are not tied to Spring,or aware of what context they are being used in. In addition, they do not have access to the domain logic or any other classes defined at the various layers. The only methods provided by the domain objects, other than the public accessors, are methods that manipulate the data contained by the object it self. Meaning that the object can control how its own data can be manipulated, as long as it is outside the context of persistence and does not require a transaction. A small example illustrating this is the MailAddress class, which is used for holding data related to an email address. An email address for e.g. an employee, is generated by placing the employee's username in a specific string format that makes up an email address specified by our specific domain; the Institute of

Informatics. This method alters the username, but the result is never persisted, so the method can be applied by every other object without impact on the persistent data.

```
public class MailAddress {
        …
        private String postfix;
        …
        public String employeeAdress(String username){
                return username+"@"+this.postfix;
        }
        …
}
```

Figure 7.3.1 Excerption of the MailAddress class

This specific implementation of the domain objects enables us to use the domain objects across all layers without exposing domain logic to the holder of a domain object. There is a restriction on the domain objects so that they can only be used as domain data containers, but on the other side they can be used freely to enhance the flexibility of the implementation at the various layers. We will see an example of this in the web layer implementation, where domain objects are used in the MVC implementation.

7.3.2 Domain logic

In this section we will focus on three major elements in the domain logic; the implementation of the domain logic facade, the encapsulation of the comparison algorithm, and the decoupling and implementation of the mail sending logic.

SendMail implementation

In chapter 6 we described how we designed the mail sending logic to decouple it from the rest of the domain logic. We defined an interface, SendMail, which declares one method that must be provided by the implementation of the interface; send(). The reason for defining an interface for only one method is that this method would otherwise strongly tie the whole domain layer to Spring and the JavaMail API. The implementation has dependencies to external libraries that we wish to separate from the overall logic. The dependencies are:

- ***mail.jar*** (JavaMail API)
- ***activation.jar*** (JavaBeans Activation Framework)
- ***spring.jar*** - Sub packages used are ***context***, ***core*** and ***mail***.

Our main concern that lead us to defining a separate interface and class for this logic, is that the implementation class, SendMailImpl, implements one of Spring's context classes that makes it aware of the container (the Bean Factory) surrounding it. This means that the class can never be used outside a Spring container, thus limiting the flexibility of the system to adapt to different frameworks. The solution was to make the choice of implementation flexible so that other implementations can easily substitute our implementation. By defining an interface we can set the specific implementation we wish to use in the configuration file of the IoC container.

```
<bean id="sendMail" class="uio.ifi.joly.domain.logic.SendMailImpl">
      <property name="javaMailSender" ref="mailSender" />
</bean>

<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
      <property name="host" value="${mail.host}" />
</bean>
```

Figure 7.3.2 Configuration of the implementation of SendMail.

The mail component of the Spring framework gives us an implementation and abstraction of the lower-level API for sending mail from the application. Spring offers two implementations; one of them using JavaMail and the second using COS MailMessage – a simpler API than the JavaMail API. We have used the implementation based on JavaMail because COS does not support sending emails with attachments.

The Spring implementation did not give us a complete abstraction of the JavaMail API, because of what we consider to be a shortcoming in the creation of a MIME message in Spring. We needed to define a MIME message that can handle a multipart message for attachments, and in addition we wanted to set the encoding format of the attachments. This forced us to use classes of the JavaMail library directly and thus we could not rely on Spring for handling the JavaMail dependency. Our implementation is because of this, explicitly tied to both JavaMail and Spring's Mail library.

The functional requirements of the Joly application states that the implementation must provide a way for the administrator to set whether an email should be sent to the lecturer or not in case of a similarity between submitted student solutions. In addition, the administrator should be able to alter the limit that states the level of similarity for when to send a copy of the email to the lecturer. This can be achieved in two different ways; either by using

properties files for setting these values or by retrieving them from a database. We wanted to explore how we could exclude the database for these types of requirements in general. This was done with the DHIS system in mind to support requirements that should not involve the database scheme which is used for statistical data storing. Because DHIS is widely distributed, they have the need to adapt the system to local requirements. Thus an easy way to enable and disable different functionalities can be one way of achieving such a flexible system.

Properties files enable the administrator to easily alter values without having to change the Java code and then recompile it before deploying the whole application again. Spring ease the way of assigning values to an object's properties by providing a way to set properties when configuring beans in the container. The values can either be set directly in the configuration file or they can be read from properties files by using a Spring provided class – the *PropertyPlaceholderConfigurer* shown in the snippet of our configuration file below.

```
<!-- Configurer that replaces ${...} placeholders with values from properties
files -->
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
       <property name="locations">
              <list>
                     <value>WEB-INF/joly.properties</value>
                     <value>WEB-INF/mail.properties</value>
                     <value>WEB-INF/jdbc.properties</value>
              </list>
       </property>
</bean>
```

Figure 7.3.3 Configuration of the PropertyPlaceholderConfigurer class.

The properties of the beans to be assigned values from these properties files, must define placeholders of the form ${…} so that the container can configure the beans with values found in the properties files. The bean definition of the SendMail implementation uses this approach to set the values that can be altered by the administrator.

```
<bean id="sendMail" class="uio.ifi.joly.domain.logic.SendMailImpl">
       …
       <property name="sendToLecturer" value="${mail.sendToLecturer}" />
       …
       <property name="limit" value="${mail.limit}" />
</bean>
```

Figure 7.3.4 Bean definition of the SendMailImpl class.

The properties files must have a syntax of key-value pairs on a single line separated by an equal sign. These are files read by a java.util.Properties class, but this is a concern of the Spring framework and the specific implementation is hidden by the framework. The

excerption below is from the mail.properties file which can be used by the administrator to make the application send a copy of the email to the lecturer.

```
#send Cc mail to lecturer? true | false
mail.sendToLecturer=false
#send Cc mail to lecturer when the result
# of the comparison algorithm exceeds or equals this limit (1/2/3)
mail.limit=3
```

Figure 7.3.5 Properties file for properties relating to the mail sending logic.

The approach described so far, requires that the administrator restarts the web server so that the Spring container re-initializes the properties in the beans. This inhibited the dynamic altering of the values that we wished to achieve. Our initial aim for this functionality was to provide a way to alter the values thru a user interface available to the administrator. To achieve this, we had to implement the send mail logic so that it read the configuration files explicitly, not depending on the container to set the values. The easiest way to achieve this was to make the SendMail implementation aware of the resources available to the application context. It was this solution that made the SendMailImpl class tied to the Spring container by implementing the Spring context class *ResourceLoaderAware*. When the Spring container detects a bean implementing this interface, it notifies the bean of the available resources. More precisely, it injects a *ResourceLoader* object of the Spring core package that can be used to load any available resources. The SendMailImpl class uses this object to get values from a properties file to override the values that were initially set by the bean container at startup. This is the responsibility of the setProperties method of SendMailImpl described in the following code snippet.

```
private void setProperties() throws LoadPropertiesException{

        resource=resourceLoader.getResource(resourceLocation);

        Properties properties = new Properties();

        try{
                properties.load(resource.getInputStream());

                if(properties.containsKey("mail.limit"))
                    limit=Integer.getInteger(properties.getProperty("mail.limit"));
                if(properties.containsKey("mail.sendToLecturer"))
                    sendToLecturer=Boolean.getBoolean(properties.getProperty("mail.sendToLecturer"));
                …

        }catch(IOException e){
                throw new LoadPropertiesException(e.getMessage());
        }
  }
```

Figure 7.3.6 setProperties() method of the SendMailImpl class.

By reading property values from a properties file, we have shown how to set properties dynamically. For the functionality to be complete, it requires a function for writing the altered properties to the properties files. The Joly application currently does not offer such functionality, meaning that the administrator must alter the properties file directly, but there is no need for restarting the web server after having done so, making it easy to extend the application with a function for writing to the properties file, and providing a graphical user interface for this function.

ProgramProcessingFacade implementation

The ProgramProcessingFacade handles the subsystem where the implementation of the comparison algorithm exists. By designing a façade to the subsystem we managed to separate the processing from the classes of Joly. To simulate the algorithm, we have written a dummy class that implements an interface that the ProgramProcessingFacade has declared dependency to. Because this interface is an unstable interface were the method signature can change, we needed a decoupling from Joly, so we have left it to the framework to plugin the implementation via a configuration file. The comparison algorithm is plugged into the Joly application by using Spring’s Bean Factory.

```
<bean id="programProcessing"
      class="uio.ifi.joly.domain.logic.ProgramProcessingImpl">
      <property name="comparisonAlg">
            <ref bean="comparisonAlg" />
      </property>
</bean>
<bean id="comparisonAlg" class="comparison.ComparisonAlgImpl" />
```

Figure 7.3.7 Bean configuration.

The bean configuration in figure 7.3.7 shows that the “programProcessing” bean has declared a dependency to the ComparisonAlg interface. This is configured by setting the property called “comparisonAlg” to a reference of another bean found in the Bean Factory with the id “comparisonAlg”. The configuration of comparisonAlg tells the Bean Factory where to find the implementation of ComparisonAlg by setting the class attribute to the specific implementation class. If the Joly application is going to use another implementation of the comparison algorithm, the class attribute can simply be substituted with the new implementation class. This shows how to apply IoC and usie an IoC container to make the application flexible and easily maintainable.

JolyFacade implementation

Chapter 6 described how we designed the domain layer to have a single entry point into the domain logic by applying the Façade pattern. We defined an interface, JolyFacade, which declares all the methods that must be provided to the presentation layer.

```
public interface JolyFacade {
	List getCoursesThisSemester();
	List findAssignmentsByCourseID(int courseID);
	Course getCourseByID(int courseID);
	Assignment getAssignmentByID(int assignmentID);
	Coursegroup getCoursegroupByID(int coursegroupID);
	Studentsolution getStudentsolution(int studentsolutionID);
	boolean isStudentRegistered(String username, Course course, int groupNo);
	boolean isAssignmentOverdue(Assignment assignment);
	boolean submitAssignment(AssignmentSubmitForm as)throws Exception;
	List getGroupsForCourse(int courseID);
	Employee findEmployee(String username, String password);
	List findStudentsolutions(int assignmentID, int coursegroupID);
	List findStudentsolutions(String username, int assignmentID);
	List getReportcases(int studentsolutionID);
	void insertEmployee(Employee employee, int courseID);
	void insertCourse(Course course);
	void insertCoursegroup(Course course, short groupNo);
	void insertStudent(int coursegroupID, String username);
	void insertAssignment(Assignment assignment);
	void insertStudentsolutions(ZipFile studentsolutions, int usertype) throws Exception;
}
```

Figure 7.3.8 The JolyFacade interface declares all methods provided to the presentation layer.

Each method represents a single transaction in the system. This is a coarse-grained interface where the code for one logical unit of operations, a transaction, is collected in one method and not spread across many methods. A fine-grained interface, in contrast, would have lead to stronger coupling of the presentation layer and the domain layer because the classes of the presentation layer would have to know the sequence in which to call the methods that made up one transaction.

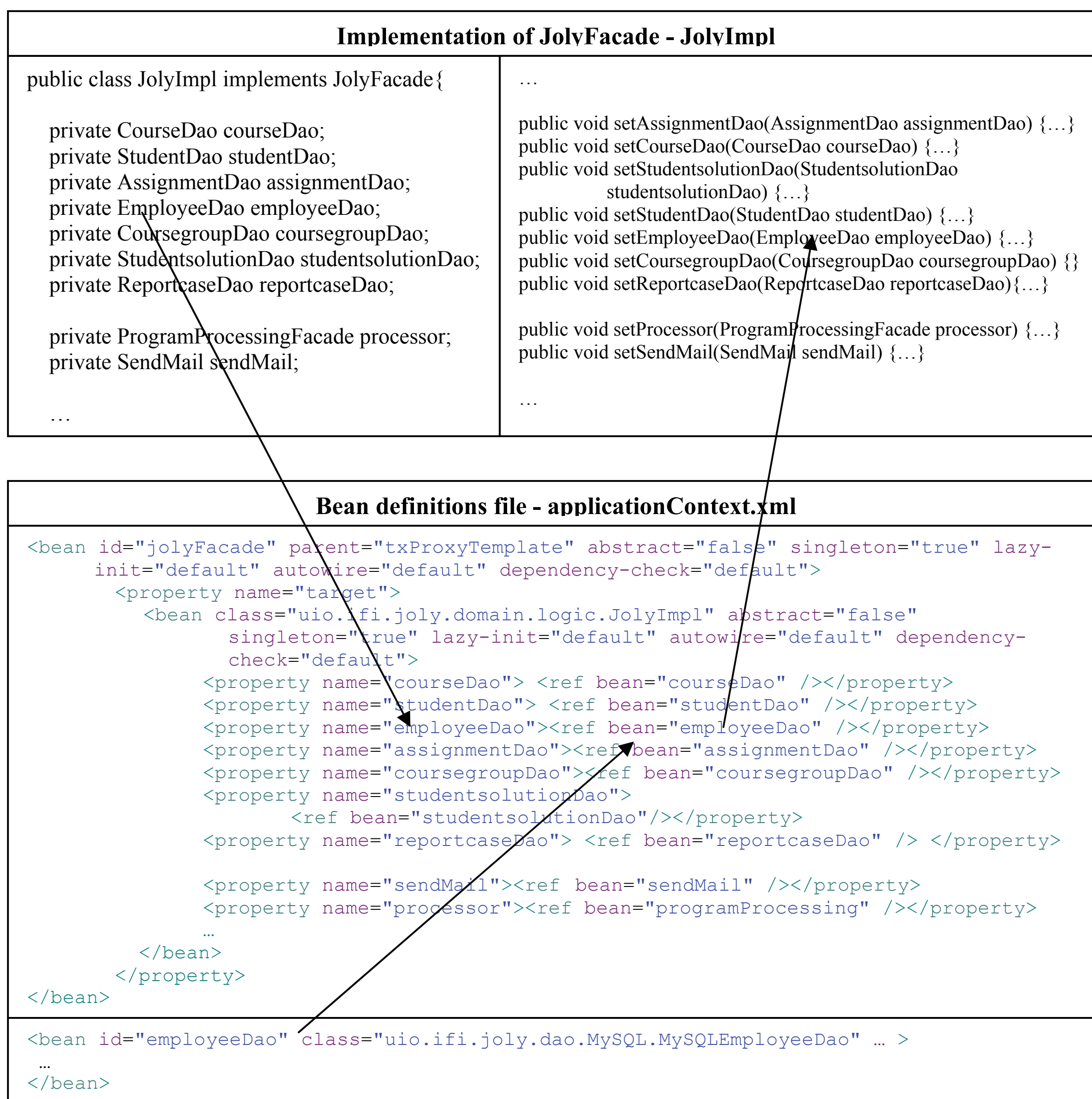
By putting all the domain logic in a non-web class, we have made it possible for web services and desktop application clients to use the same API as the web layer servlets. Even though the Joly application currently does not need such an extended feature, we recognize the ability to distribute the domain logic as web services to be used by desktop applications or local installations of the web application, as a useful feature for DHIS. For many users of the DHIS system, especially in the sub districts, they will have no or limited access to the internet and will therefore operate on a local installation of the system. For this scenario it would practical

to connect to a decentralized server when there was an internet connection available, and in the other case connect to the local server without the user having to interfere. By defining a common API for the domain logic, we have made the system flexible to be used by other types of presentation layers or rich client applications. There are no restrictions on were the domain logic can be accessed from, thus enabling access from other layers, possibly exposing the same logic as web services.

The implementation of the JolyFacade interface is the JolyImpl class. This class can be seen as a manager of all application logic. It implements all the required methods in a way that assembles all logic needed to perform one operation for the presentation layer.

The private properties of the JolyImpl are DAO interface implementations and domain logic interface implementations. Instances of these implementations are injected by the container and by this making the JolyImpl unaware of the specific type of implementation used. The JolyImpl has declared dependencies to interfaces, and provided setter methods for the container to use with the Setter injection technique.

When requests for operations on persistent data are made, the JolyImpl redirects the requests to the supplied methods of the DAO interfaces. This makes JolyImpl the entry point for the persistence layer, forcing the presentation layer to go thru this object to operate on persistent data. There are no getter methods supplied for the DAOs in JolyImpl, making the presentation layer unable to acquire a DAO for its own operations, thus all operations on persistent data are controlled by JolyImpl. This implemented restriction fulfills the demand of the three-layer architectural pattern, which states that to access the bottom layer one must go thru all the preceding layers. In addition, when maintaining the DAOs, there now only exists one class that needs to be updated if the DAO method signatures change.



Implementation of JolyFacade - JolyImpl

```
public class JolyImpl implements JolyFacade{

    private CourseDao courseDao;
    private StudentDao studentDao;
    private AssignmentDao assignmentDao;
    private EmployeeDao employeeDao;
    private CoursegroupDao coursegroupDao;
    private StudentsolutionDao studentsolutionDao;
    private ReportcaseDao reportcaseDao;

    private ProgramProcessingFacade processor;
    private SendMail sendMail;

    …
```

```
…
public void setAssignmentDao(AssignmentDao assignmentDao) {…}
public void setCourseDao(CourseDao courseDao) {…}
public void setStudentsolutionDao(StudentsolutionDao
            studentsolutionDao) {…}
public void setStudentDao(StudentDao studentDao) {…}
public void setEmployeeDao(EmployeeDao employeeDao) {…}
public void setCoursegroupDao(CoursegroupDao coursegroupDao) {}
public void setReportcaseDao(ReportcaseDao reportcaseDao){…}

public void setProcessor(ProgramProcessingFacade processor) {…}
public void setSendMail(SendMail sendMail) {…}

…
```

Bean definitions file - applicationContext.xml

```
<bean id="jolyFacade" parent="txProxyTemplate" abstract="false" singleton="true" lazy-
      init="default" autowire="default" dependency-check="default">
        <property name="target">
          <bean class="uio.ifi.joly.domain.logic.JolyImpl" abstract="false"
                singleton="true" lazy-init="default" autowire="default" dependency-
                check="default">
                <property name="courseDao"> <ref bean="courseDao" /></property>
                <property name="studentDao"> <ref bean="studentDao" /></property>
                <property name="employeeDao"><ref bean="employeeDao" /></property>
                <property name="assignmentDao"><ref bean="assignmentDao" /></property>
                <property name="coursegroupDao"><ref bean="coursegroupDao" /></property>
                <property name="studentsolutionDao">
                        <ref bean="studentsolutionDao"/></property>
                <property name="reportcaseDao"> <ref bean="reportcaseDao" /> </property>

                <property name="sendMail"><ref bean="sendMail" /></property>
                <property name="processor"><ref bean="programProcessing" /></property>
                …
          </bean>
        </property>
</bean>
```

```
<bean id="employeeDao" class="uio.ifi.joly.dao.MySQL.MySQLEmployeeDao" … >
 …
</bean>
```

Figure 7.3.9 Declaring and wiring dependencies to be injected by Setter injection.

The principal of restricting access to layers of the application is also used within the domain layer to restrict access to the other domain logic classes. These classes are the implementation of the send mail logic and the subsystem encapsulating the comparison algorithm. For the presentation layer components to use the logic provided by these objects, they have to go thru the JolyImpl object. The presentation layer is not even aware of the specific domain logic classes handling the different operations. It is the responsibility of JolyImpl to call the methods of the domain logic classes, while exposing only one simple method to the presentation layer that represents a single unit of logic. A visualization of this is provided in figure 7.3.10 which shows the sequence of all domain logic performed for submitting an assignment while exposing only one method to the presentation layer; submitAssignment().

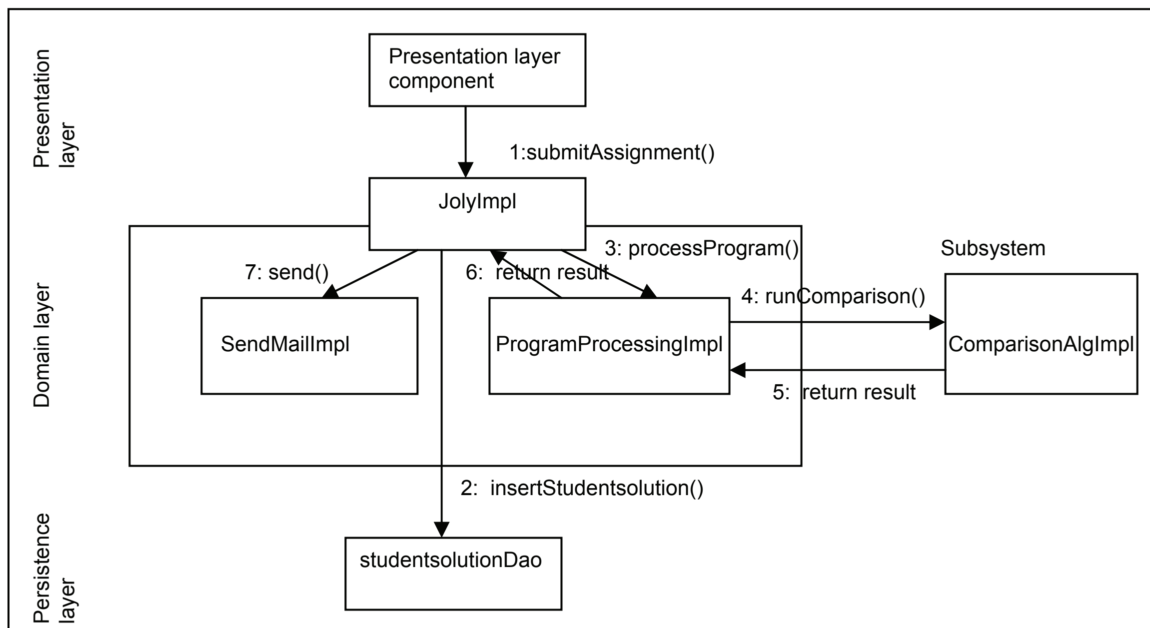


Figure 7.3.10 Method sequence of domain logic for submitting an assignment.

7.4 Implementation of the web layer

In this section we will explore the details of the web layer implementation with the Spring MVC framework. This layer introduces several technologies. We will describe the different technologies and show how we have used these technologies to implement the different components of the web layer.

7.4.1 Java Servlets

The servlet technology was introduced by Sun in 1996 as small Java-based applications for adding dynamic content to web applications. A servlet is a java class that can be loaded dynamically into, and run by a servlet container. It is loaded into the container when it is requested for the first time and then held in memory awaiting new requests. If there is a shortage of memory, the container will unload the servlet until a new request is issued. This is possible because the servlet is managed by the Java Virtual Machine, which takes care of memory leak and garbage collection. The container will also reload the servlet if it finds a more recent servlet, as in the case of a change in the servlet class.

Servlets work on top of http and interact with clients via the http request-response model. When a request is issued it is received by the servlet container which forwards it to the

servlet. The servlet processes the request and issues a response which is sent back to the client by the container. A servlet container will allow multiple requests for the same servlet by creating a different thread to service each request.

7.4.2 Java Server Pages

JSP is an extension to the Java Servlet technology and uses the same techniques for http request-response processing as servlets. JSPs are often used in conjunction with servlets to remove the responsibility of writing html strings from the servlets and place it into the JSPs. This decouples the request-processing logic from the presentation details that is made up of the html code.

Servlet

```
...
out.println("<html><head><title>Joly</title></head>");
out.println("<body>");
out.println("Velcome to Joly " + username);
out.println("</body>");
out.println("</html>");
out.close();
...
```

JSP

```
<html><head><title>Joly</title</head>
    <body>
        Velcome to Joly
        <% request.getParameter(”username”); %>
    </body>
</html>
```

Figure 7.4.1 Presentation logic in servlet versus presentation logic in JSP.

A JSP container is required for executing JSPs, but most container implementations can be used to run both servlets and JSPs, thus the term JSP/Servlet or web container is used to cover this type of container. The Jetty server that was used during the development and for the deployment of Joly, supports both servlets and JSPs.

There is no need for explicitly compiling a JSP, because when a request for a JSP reaches the web container for the first time, a servlet within the container, called *page compiler*, parses and compiles the JSP into a servlet class and loads it into memory. After that its lifecycle is similar to that of a servlet. The delay that is caused by every request for a new JSP, can be prohibited by pre-compiling all JSPs of the application and deploying them as part of a .war file in the web container. The first approach is convenient during development because there is no need for restarting the web container for every update in the code, whereas the second approach is the most appropriate when the application is to be deployed.

A standard JSP file contains both html tags and Java code in the form of scripting elements surrounded by specific JSP tags. These scripting elements are intertwined with the html tags, giving the disadvantage of lack of readability and not fully fulfilling our need of separation of logic and presentation. Also, if the presentation is to be altered by a graphical designer, it would require knowledge of Java as well as html. To resolve this problem we can separate the logic by using JavaBeans and calling its methods and properties from the JSP using *action elements*. The JSP API comes with a limited set of action elements (only three) for retrieving and setting bean properties, but there is a possibility of defining custom tags and there exists many custom tag libraries available from various vendors. Sun offers a specification of a tag library that can be used to extend, or rather replace, the JSP API tags.

7.4.3 Java Server Pages Standard Tag Library (JSTL)

The JSTL is Sun's custom tag library specification. By using these tags one can nearly exclude all Java code in the JSP file, and we also take advantage of the fact that JSTL is defined by a specification, meaning that it is likely it will be supported by most container vendors. We have used the open source Apache implementation of JSTL that is packaged with the Spring framework. By using Apache's implementation of the JSTL specification instead of writing our own, we limit the amount of code needed to be written.

Spring tag library

Spring offers some custom tags to facilitate the creation of web pages using the Spring MVC components. The Spring tags can be used to bind data to objects, handle error messages and internationalized text messages. All these three aspects are used throughout the JSPs of the Joly application, and the Spring tags has made this feasible in a very seamless way, with the exception of internationalized text messages. For the latter case we have used the Apache JSTL implementation of the i18n-capable formatting library "fmt". Because Spring and Apache offered the same functionality we decided to use Apache's implementation because this is an implementation of a standard, whereas the Spring tags are not.

7.4.4 The Expression Language (EL)

Since the JSP 2.0 release, the EL has been moved from JSTL to be incorporated in the JSP API. The EL facilitates access to JavaBeans properties. The value of an expression is

computed and inserted into the output. An example of using EL in JSP is shown in the example code below , which is an alteration of the JSP example in figure 7.4.1.

JSP

```
<html><head><title>Joly</title</head>
    <body>
         Velcome to Joly  ${username}
     </body>
</html>
```

Figure 7.4.2 JSP accessing bean property ‘username’ using EL.

7.4.5 JavaScript

The use of JavaScript in Joly is limited, and the use is restricted to the JolyAdmin interfaces. The interface for submitting assignments by students does not use JavaScript. The reason for this is that Joly needs to provide for those users who do not have JavaScript. There are several reasons why the users might not have JavaScript:

- They could be using a browser which does not support, or only partially supports scripting. (Opera 5, Netscape 4).
- They could be behind a proxy server or firewall that filters out JavaScript.
- They are using a browser where JavaScript has been deliberately switched off.

These possible constraints on browsers should not make the student unable to make an online delivery of his or her mandatory assignment. The use of JavaScript in the interfaces would have made the application faster by putting more logic on the client side and thus reducing the amount of server side processing, but we feel it is more critical to the students that they are able to submit their assignment in whichever browser they are using (due to the deadline of the assignment) than the submitting being fast and more user-friendly. The employees do not have the same critical time restriction on their work, and are more likely to use a JavaScript enabled browser.

7.4.6 Alternative technologies

PHP is an open source technology that started as a pure web technology and recently evolved into the field of multi-tier application development. The technology still bears signs of being a pure web technology, so it is more an alternative to JSP/Servlet than to Java in general. In the beginning of the Joly development process we were considering to use PHP for the front-end and Java for the back-end. It soon became apparent that this was a solution that lead to unnecessary complications due to the Java Virtual Machine and PHP module collaboration. We were determined to use Java as the back-end because PHP, in our opinion, has limitations with regards to its API. Java, with the J2EE development platform, is more suitable for handling domain logic. Because servlets have access to the J2EE API, using the servlet technology at the front-end speeds up the development process by eliminating the need for integrating PHP and Java. In addition, the web frameworks developed for PHP has not reached the quality of many of the Java web frameworks.

The main competitor to the JSP/Servlet technology is the Active Server Pages.NET (ASP.NET) technology of Microsoft. Just as servlets are handled by the JVM, the Common Language Runtime (CLR) of the .NET framework offers a vast class library to ASP.NET pages. Using ASP.NET requires using the .NET framework for development, and as we have pointed out earlier, this solution is inappropriate for us because of the costs related to the technology, and the immaturity of the only free and open source implementation of the framework - Mono.

7.4.7 Implementation of the web layer using Spring MVC

Our goal for applying the MVC pattern was to design a structure in the web layer that decoupled the view technology from the logic of user interaction thru the web – the web logic. By using the Spring MVC framework we can achieve this, as well as abstracting the http request processing and hiding the translation of html strings to domain objects.

The web MVC pattern is applied to the server side code, thus its processing logic starts when an http request from a user's browser reaches the server. The Spring MVC uses a *dispatcher* to control the processing of a request. It introduces the *DispatcherServlet* which is based on the Front Controller pattern. The Front Controller pattern forces a centralized access point for user interface request handling [85]. This makes the DispatcherServlet responsible for

receiving all requests, examining them and delegating them to the appropriate controller for the actual processing. The DispatcherServlet is implemented as a Java Servlet, while the controllers are ordinary Java classes. By using a dispatcher, we separate the processing of the http nature of the request from the controllers, making the controllers less closely coupled to http.

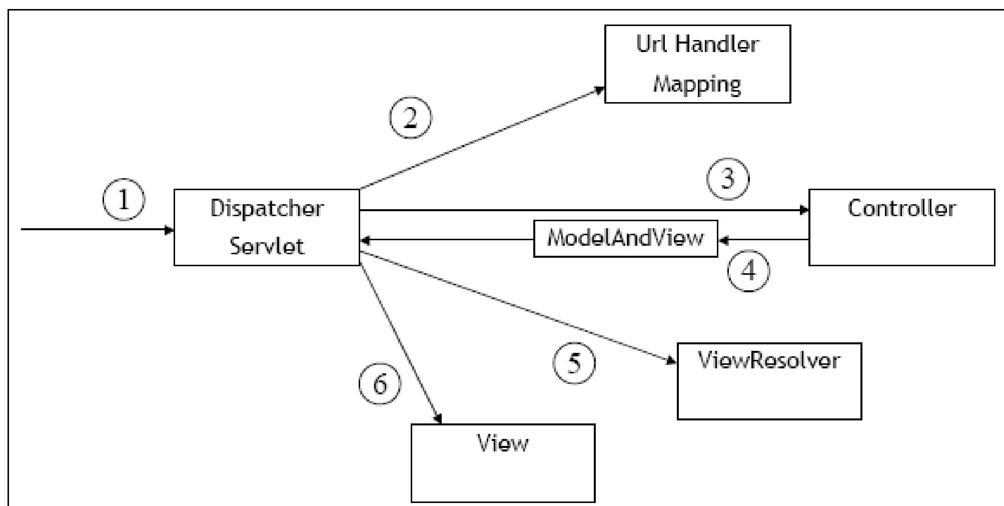


Figure 7.4.3 Spring MVC components [86].

We have defined one DispatcherServlet, Joly, which receives all requests coming from the web pages of the Joly application, dispatching them to an appropriate handler, which is an implementation of the Spring Controller interface. A handler can be any type of object, not restricting us to Spring classes, but we are using the Spring convention and implementing the handlers as Spring Controllers. This means that the one servlet implemented in Joly does not do any processing other than delegating work to ordinary Java classes, the controllers. This structure has given us the ability to code in a familiar environment and let the Spring framework handle the servlet implementation and thus the http specifics.

Controllers

Spring MVC has several controller interface types defined in an extensive hierarchy one can use in order to implement specific workflow scenarios (see appendix C for details). The framework also offers implementations of these interfaces, such as the *SimpleFormController*, to handle typical workflow scenarios found in many web applications. The SimpleFormController handles the workflow of generating an html form, and handles the post action when the user submits the form. The Joly application uses a couple of these available implementations as well as the base interface Controller.

All Spring controllers interact with the *HttpRequest* and *HttpResponse* classes of the JavaServlet API, enabling them to handle http requests and responses while hiding the details from its own processing logic. This means that the code in the controllers we implement will contain only the user interaction logic; how the controller reacts to a specific user interaction – which view to render and what domain logic to initiate. The next figure shows one of the simpler implemented controllers in Joly; the ViewStudentsolutionController class which implements the base Spring Controller interface.

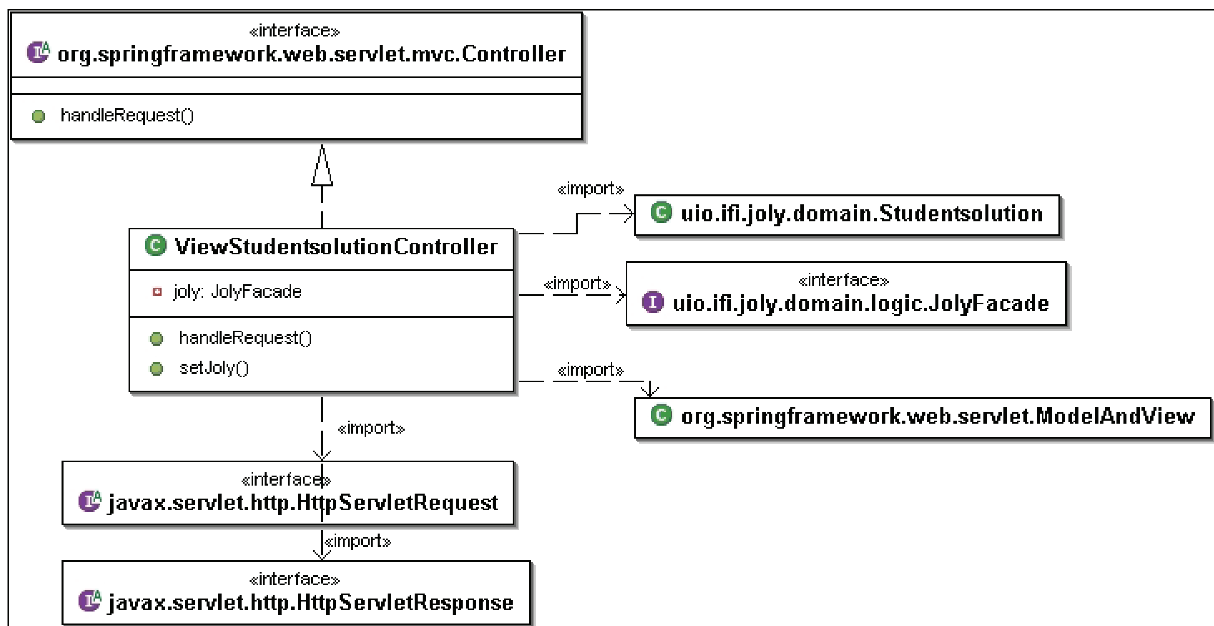


Figure 7.4.4 ViewStudentSolutionController and associated classes.

A more complex scenario of the Joly application is the wizard that guides the student's assignment submitting. This scenario requires a multi-page entry into one single form. This is more complex than a single-page form because navigating from one page to another means issuing a new request to the web server and the state of the previous page is lost. Spring has recognized that wizards are a very common workflow in many web applications and has thus created a suitable interface for this – the *AbstractWizardFormController*. This interface is found in the bottom of the Spring Controller hierarchy, meaning it inherits all the methods of the above interfaces. This interface provides methods that support the many aspects of a wizard style workflow, where the main aspect is holding the user input data throughout the whole session. A well-known drawback of HTTP is its inability to hold the state of the user session, in other words it is *stateless*. Spring provides us with a very simply way of disguising the statelessness by setting a property called *SessionForm* in the above inherited controller

interface. When using the Spring MVC Form Controllers, this is all a developer needs to know about the session scope of web applications. Behind lays a more complex logic, but this is all handled by the framework.

Model

The Spring MVC Model is comprised of a command object and/or reference data to be displayed in the view. In Spring MVC, the command object can be any JavaBean. This enables us to use our domain objects in the web layer so that we do not need to write separate classes for the command objects residing only in the web layer. The command object is an object associated with the data a user submits via the user interface. This is an advantage in Spring over other frameworks such as Struts, where every object associated with user input must inherit a specific interface. The Spring controllers using a command object are the controllers extending the Spring BaseCommandController, such as the SimpleFormController. This controller can map request parameters submitted from an html form to a command class, thus resolving the string to object mismatch problem.

Reference data can be used to pre-populate the html forms presented to the user, e.g. to fill a select box for the user to choose from. Reference data is gathered by the controller in a Java Map object and put in the model with the configured command object. Reference data is usually gathered by making one or more calls to the domain layer for retrieval of persistent data.

Views

Using Spring MVC in the web layer forces a separation of view implementation and the web logic. The view implementation can be of any view technology supported by Spring, and the model, view and controller components defined by Spring MVC are not aware of or tied to the specific implementation used. This decoupling is achieved by the *View* and *ViewResolver* interfaces available in the framework. As visualized in (figure 7.4.3), the controller returns an instance of a ModelAndView class to the dispatcher. This instance contains the model and either a view name or an implantation of the View interface. This flexibility gives us a choice of either implementing our own view with a method for rendering the content of a file such as a JSP or Velocity file, or to simply hand over a logical view name to the dispatcher so that it can resolve the view via a ViewResolver implementation. To use the available

implementations of the ViewResolver, it requires that Spring has an appropriate implementation that suits our needs. The ViewResolver implementation's responsibility is to map the logical view names to the actual implementation. Thus it needs to be able to recognize the type of view technology used to render it appropriately. Spring offers many ViewResolver implementations so it is very likely to find a suitable one.

7.4.8 View implementation

As explained in the previous section, the View element of the MVC is just an interface and not the actual implemented view. By separating the view technology from the MVC components in the web layer, we have introduced a thin layer on top of the web logic. This layer represents the view implementation. By providing a View interface, Spring has introduced another hinge for us to adapt the Spring MVC to our specific implementation. By using this interface and separating the view implementation, we have enabled other developers to change the views to their specific needs. For instance, if Joly needed to produce statistical reports, a view technology accommodating this could easily be integrated.

We have implemented the views using JSPs with JSTL and Spring offers JSP/JSTL rendering "out of the box", so we can use Spring MVC's *InternalResourceViewResolver* that supports the appropriate *JstlView* subclass.

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass">
         <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

Figure 7.4.5 Bean configuration of Spring MVC's InternalResourceViewResolver.

We have configured the ViewResolver to concatenate all view names with the prefix "/WEB-INF/jsp/", indicating the directory where the jsp-files are located on the server, and the suffix ".jsp", indicating that it is a jsp-file. By doing this, neither the placement of the view files, nor the actual type of view technology used, needs be hardcoded in our controllers. If the view technology is to be replaced, we can still use the same controllers and models, and simply alter the configuration of the ViewResolver.

In addition to configure the ViewResolver to recognize logical view names and associate them with the physical view file, we have configured the ViewResolver to render these files with a JstlView. This is the alternative to write our own implementation of the View interface. We chose to use an available implementation in the framework to render our jsp-files because Spring offered a very suitable implementation.

In the following sections we will describe how we have resolved a couple important aspects related to the views of Joly, and how the used features can also enhance the views of the DHIS application.

Validation of user input

Validation is an important aspect of web application development. Almost all web applications with a browser user interface will need to validate the user input. The support for validation is one of the features a good web frameworks should entail and can often be a deciding factor when choosing web framework. Validating the Spring way offers a fairly simple validation system, if the Commons Validator framework is not used, as is the case of the Joly implementation.

When data is bound to an object, it can be subject to domain object validation. Each domain object that needs validation, because it is used directly in the views, has a validation class associated with it. This class inherits the Spring Validate interface. Spring MVC takes care of calling the validation objects when a view that has data bound to an object, is submitted. To glue the domain object, its validation object, the view and the controller together, we use the Spring IoC container and extend this into the web layer. This is an example of how easily Spring MVC can collaborate with the domain layer using an IoC container. A visualization of how the components are configured is provided in figure 7.4.6.

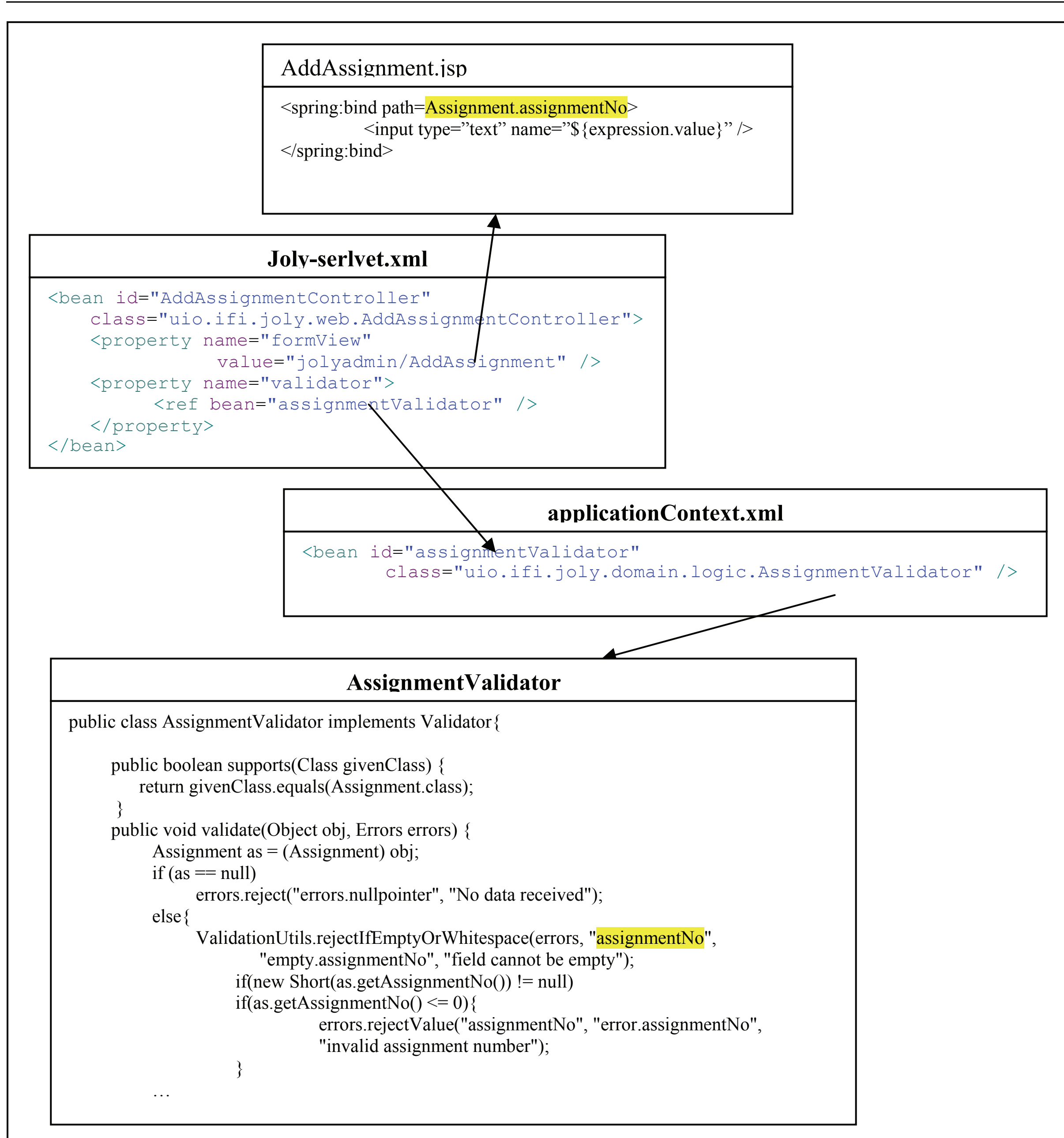


Figure 7.4.6 Binding and validating the Spring way. Joly-servlet.xml is the Web Application Context where all the beans of the web layer are configured. The web context can reference beans in the application context. The validators are configured in the application context because they are associated with domain objects.

An example scenario where validation needs to be performed, is when an employee wants to register a new assignment and supplies the required information in the fields of an html form. The data entered is going to be used to set the properties of an Assignment object, which will be persisted in the database. Some data need to be saved as numbers, such as the assignment number, so when the request is received by the server, the html string containing the data, needs to be checked to see if the employee has entered a valid number. If not, the employee will be prompted with an error message and be able to correct the wrongly entered data.

For advanced validation requirements, which is more likely to appear in the DHIS project than in Joly, the well-known Commons Validation framework can be integrated with Spring and handled from the container. In stead of incorporating the Commons Validation framework one selves, it is possible to let the Spring framework take care of the dependency. This is an example of how Spring has integrated another framework to reuse existing expertise. The Commons framework introduces a plugin architecture of validation rules that can be used to adapt the application to user-specific validation rules. DHIS, as described in chapter 2.5, requires a customization of validation rules for each user, and thus the Commons Validation can be used to support these requirements.

Internationalization

One of the aspects of the view implementation, has been to internationalize the views with text messages of the local language. This has been important mainly because DHIS has a requirement of being multilingual, but also because we see the need of the Joly application to support foreign students. The views contain a lot of text that needs to be translated into the different supported languages. Without the support of a tool handling the localization of the views, we would have to multiply each JSP file with the number of supported languages and translate the text to be displayed in each. By using Spring's support for message sources and combining it with the i18n-capable formatting library of JSTL, we have enabled the views to easily be internationalized.

Joly-servlet.xml

```
<bean id="messageSource"
       class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
               <list>
                      <value>etc/ViewMessages</value>
               </list>
        </property>
 </bean>
```

Figure 7.4.7 Configuration of a Spring implemented message source.

A Spring message source is configured with the localization of a properties file. This enables JSPs to access the properties by using JSTL tags.

ViewMessages.properties

```
label.assignment=Oblignr.

button.insert=Legg inn

error.assignmentNo=Feil i angitt oblignr.
typeMismatch.java.lang.Short=Feil format på inntasted data. Vennligst angi et
tall.
```

AddAssignment.jsp

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
…
<label for="assignmentNo"><fmt:message key="label.assignment" />: </label>

<input type="submit" value="<fmt:message key="button.insert" />" />
…
```

Figure 7.4.8 Excerpt of the ViewMessages and AddAssignment files showing the use of JSTL fmt tag library and Spring's message source.

7.5 Summary

When presenting the implementation of the Joly application, our focus has been to describe how we have implemented the design laid out in chapter 6 and how we have used the selected tools to provide both functional and non-functional support. Because of the focus on implementing the application to conform to the design patterns chosen, we have discussed the design decisions and revisited some of the previous discussions in this chapter as well. It may seem like we have spread the general design discussion to all chapters covering the Joly application, but we have felt the need to explain the general design when discussing how the detailed implementations conform to and support the overall design of the application.

The implementation of the specific functions of Joly shows how to use the frameworks and the concepts they are built on. This chapter has therefore described how to resolve the functional requirements as well as the non-functional requirements of the application. The level of detail has been high in some areas, but we felt that this was necessary to show how frameworks can be used to solve specific tasks and how to customize the general classes of a framework to ones own implementation.

We have described how we have implemented the persistence layer so that it is flexible towards database substitution. The integration of Hibernate is done in a manner that makes

Hibernate easily removable by using annotations as opposed to xml-mappings. The chosen release of Hibernate also enables the application to support implementation of EJB3 in the future.

The domain layer is implemented so that components can easily be substituted and added. In addition, the domain logic is accessible to all type of layers, such as web service layers, to support a potential requirement of web service implementation.

The web implementation conforms to today's standard when implementing web logic, by applying the web MVC pattern and implementing classes with the assigned responsibilities. The implementation and use of Spring's MVC framework has enabled the application to adapt to different types of view implementation, making the application flexible to developers' choice of technology in the presentation layer.

Chapter 8

Experiences

In this chapter we will present the experiences we have gained from the investigation process and development of the Joly application. We will point out the problems we have encountered when using free and open source tools and other relevant experience gained from the development process.

8.1 Experience gained by the analysis process of the DHIS software

The considerable amount of effort put in to the analysis of the DHIS software at the beginning of our research, has given us experience relevant for the DHIS project team. This is related to the documentation of the software, and the availability of system requirement documentation. For the latter there was none. For the former there was limited documentation, mostly found in user manuals.

When we started an evaluation of the DHIS software to find a new, best fit technological platform, it can broadly be seen as the beginning of a new maintenance process of the software. As the DHIS aims to be a long-lived system, they must anticipate a similar reevaluation of the software to take place at every major paradigm shift in the technological arena. For DHIS to be well equipped for these major maintenance processes, they must take experience from the ongoing major maintenance process.

Our experience in analyzing the software revealed four major concerns that should be addressed by the DHIS team with regards to future software maintenance:

Communication of application design: It is important to find a way of communicating application design. The lack of guidance for the existing application design was one of the factors that increased the effort needed to understand the software. This can be resolved by defining a set of patterns that explains the key design decisions of the application [47].

Introduction to existing software: A way of rapidly introducing new developers to the existing software should be explored. Frameworks can reduce the effort required to understand the software by encapsulating implementations, and localizing where changes can be made. For a larger system, constantly in an incremental development and maintenance process, a domain specific framework or analysis model can be developed to ease the software introduction to new developers. An analysis model can also make existing code reusable by describing the guarantees and assumptions they make, making it easier to incorporate it with new code [87].

Documentation of new requirements: New requirements to the DHIS system are received by the HISP team continuously. In this case, the requirements have lead to the need for a full reevaluation of the software. By incorporating an incremental requirements analysis and documentation process, a solid documentation of the problems needed to be solved, can be present when the need for a new reevaluation of the software emerges.

Discovery costs: A tool should be used to reduce discovery costs (or “getting started” costs)[88] related to the costs of understanding the problem to be solved, i.e. what is the actual problem to solve when an object oriented system needs to be RDBMS independent? The answer can often be found in frameworks, such as in this case, the Hibernate framework has already solved the problem of making an application RDBMS independent (by handling the mapping between objects and relations).

8.2 Free and Open Source Software evaluation process

Using free and open source software (FOSS) for development of a new application requires a comprehensive evaluation process before they are adapted, to ensure stability and maturity of the software in addition to investigate that they meet the necessary non-functional requirements. There exists a large amount of FOSS products, and it can be time consuming to

orientate oneself to find the most appropriate product solution. The definition of criterions for which the solutions are to be evaluated by is therefore an important grip in the evaluation process, which we found helpful in comparing the solutions up against each other.

There is a lack of standards for evaluation of FOSS solutions. Some evaluation models exists, but we found some of them too comprehensive and time consuming (more appropriate for large enterprises), and some of them to superficial. This led us to defining our own evaluation process, which included the basic steps that the evaluation models had in common. The evaluation process we carried out, combined with the evaluation of FOSS solutions' properties, provided us with a thorough analysis of the candidates that made us able to make a decision of the best fit solution for our development. In cases where the analysis resulted in several top candidates, DHIS' choice of solution was taken into consideration to choose which of the candidates to use for the development of Joly.

We recommend DHIS to define its own evaluation process and criterions, or use one of the existing, to provide a standardized method for evaluating new solutions, to ensure that all of the solutions have a certain level of stability and maturity before they are adopted. The definition of a standardized method could also ease the maintenance process when changes in the systems environments occur, to re-evaluate the selected solutions to investigate if they support the new functional or non-functional requirements introduced.

The evaluation process was time consuming, even though we had chosen only a few basic steps for the process, and a few important criterions for evaluation of the properties of FOSS. FOSS products are in great request at the moment, but the use could be even more widespread if it was less time consuming to evaluate the products, and thus minimizing the efforts needed at the beginning of a new development project. Because of this, we request the development of a standard for both the evaluation process and the criterions the products should be evaluated by, and we are looking forward to see the results of the development of the Business Readiness Rating model (to determine the maturity of a FOSS product), hoping that it will ease the evaluation process.

8.3 Using free and open source development tools

During the development process we have used the Eclipse Software Development Kit (SDK). Eclipse is an open source community which has developed a platform designed for building integrated development environments (IDEs), and arbitrary tools. Eclipse provides a plug-in based framework that makes it easier to integrate and utilize software tools from different vendors.

To support the use of the different free and open source tools we have used, Eclipse plugins for these tools have been integrated with the Eclipse IDE. These plugins include the Spring IDE plugin, the Jetty Launcher plugin, the Clay Database Modeling plugin, and the Omondo UML plugin.

Each of these tools must be located and downloaded by the developers and there can be several implementations of suitable plugins available, and many releases of each plugin. Because of this, the plugins have also been subject to an evaluation process when there has been more than one available, and when trying to find the most suitable version to use. The main problem when integrating plugins with the Eclipse IDE, is the compatibility between the plugin versions and the Eclipse IDE version. The latest plugin of a tool may not be compatible with the latest version of Eclipse and there may not be available a plugin for the latest release of a tool. Another problem that arose when integrating several plugins in one Eclipse IDE, was that the plugins each depended on the same Eclipse components, some on earlier versions of the components and some of the latest. The problems mentioned, and the process of locating and evaluating the plugins, resulted in a time consuming start up process that was more complicated than what we have experienced with commercial IDEs in the past.

8.4 Framework related experiences

The claimed advantages of using frameworks for application development are many. These include modularity, reusability, extensibility and Inversion of Control [52]. In this chapter we will discuss our experience using open source frameworks and the effect it has had on the development of the Joly application.

During the development process we have encountered some obstacles that have diminished the advantages that were intended by the framework.

8.4.1 Problems related to the development process

Although framework developers claim that it is feasible to develop complex applications with little development effort with the use of a framework, we have encountered some aspects that should be taken into consideration when deciding to use a framework in the development process.

When using a framework it is crucial to understand all the concepts of the framework and its architectural basis, to be able to develop an application that conforms to it. We are of the opinion that for a framework to bring benefits to the development process, it is necessary that the developers have a strong understanding of the basics of the framework and a good knowledge of the details. Further, to acquire this type of understanding, it is necessary with hands-on experience with the chosen framework. This is because the quality of the documentation of the various frameworks varies, and it's often necessary to read documentation from many different user groups to get a full description of the details of the framework. The documentation often shows signs of the writers preferences in which domain to apply the framework, and the experience the writer has with the framework in question can also be different from that of another user. For instance, when studying the Spring framework, a lot of the documentation assumed that the reader was familiar with another framework, most often the Struts framework because it was one of the first frameworks to strike root in the Java web application field. Since neither of us had past experience with frameworks, learning a framework's concepts by comparing them to other frameworks did not give a full understanding of what it was intended to do.

We have experienced that frameworks are hard to learn without trying them out in a real application. Thus the Joly application has been built with a "learn as you code" approach, and may be suffering from design flaws because of this.

Another problem associated with framework documentation is that open source frameworks are constantly evolving. The documentation is usually then the last artifact to be updated.

Thus the documentation of new features, and deprecated features, is poor. In these cases it is almost always necessary to refer to the API documentation, which gives little information of the use of the features.

Frameworks and patterns are claimed to be reusable software expertise. We do not question this claim because it is obvious that well established patterns, and frameworks based on those patterns, solve problems common to many application domains. However, how well the framework communicates this software expertise is more important, because the lack of such introduces side effects that can diminish the claimed advantages of the framework. For instance, if learning the framework demands an effort that exceeds the effort of developing an application without a framework, it clearly lacks good communication and the software expertise is hard to reuse.

When the goal of starting the development of a system is a rapid development process, to get a prototype up and running as soon as possible, we do not recommend a development team with no experience in frameworks to incorporate a framework in the development process. For the DHIS development project it is more important to achieve the non-functional requirements of the system, thus teaching the development team a set of useful frameworks can be very successful. The DHIS development team in Oslo has recognized the need to teach the developers in the relevant technologies and frameworks. In 2005 the University of Oslo started a course that covers open source development and Java frameworks where the students are introduced to the different technologies by solving relevant assignments. These developers will learn the frameworks by hands-on experience and thus will be fit to use the framework in an extensible development process. Therefore, we can recommend the use of frameworks in the development of DHIS, because the claimed advantages of frameworks will probably take effect.

The problems that arise early in the development process associated with frameworks are:

Determining the applicability of a framework: The process of deciding whether a certain framework is appropriate for the application in mind is comprehensive. In our case a full evaluation of the available frameworks would have reached beyond the scope of our study,

and thus the choice of framework had to be based on simple criteria. Consequently we can not be sure that the choice of frameworks for our application was the most suitable.

Estimating development progress: Because of the effort related to studying the frameworks as mentioned, it is hard to estimate the development progress of a project. For a small application, a framework-based approach will on average be slower than that of a traditional approach. We are of the opinion that this will be the case even with framework experienced developers, because of the effort needed to set up and integrate the various tools. For a larger and more complex application however, this time consuming start up effort will pay off and increase the overall efficiency. A second problem is that it is hard to foresee whether a framework will support all the requirements of the application. A shortcoming of the framework may be exposed at any time during the development and require an effort in trying to figure out a solution. By this time a considerable amount of time has probably been used to figure out the shortcoming, so in contrast to a traditional approach, a double amount of time may have been spent to resolve the problem.

Estimating application size: When developing applications using frameworks, dependent packages may not be introduced before specific requirements are implemented. Open source frameworks often use other open source tools to solve well-known problems. Some may be included in the framework release and some are not and thus needs to be added by the developers. Also, because a framework is designed to resolve general problems, many packages may need to be included to resolve a smaller specific problem.

8.4.2 Problems related to maintenance

In chapter 5 we stated that we could customize a framework to our application by inheriting and instantiating selected framework classes. The problem with Spring being such a flexible framework is that it has a considerable amount of available classes. To consider each class in order to select the most appropriate to use in a specific scenario, became a hard task and we have probably not succeeded in always using the correct Spring classes. This may have resulted in us assigning responsibility to wrong classes (not intended to be assigned this responsibility by the Spring developers), thus not conforming to the framework. For new developers of the application, it can be hard to understand the application design if they

expect to find implementations of classes they know have the responsibility of a specific problem.

Problems we see may occur after the release of a framework-based application are:

Maintaining framework conformity: By looking at the above mentioned scenario in reverse, an application developed by developers experienced in frameworks, may be subject to new developers not conforming to the frameworks intentions. This introduces a maintenance problem similar to that of design patterns. If developers break a pattern it is possible that they have removed the application's support for a non-functional requirement. The same applies to frameworks because they are based on patterns and offer implementations that conform to them, so misusing a framework's classes can introduce side-effects that have influence on the non-functional requirements they were intended to solve.

Maintaining framework dependencies: Using open source frameworks introduces a number of third-party dependencies. Most frameworks incorporate one or many other frameworks or libraries to accomplish a common task. If the framework developers are not directly tied to the developing of these third-party tools, the dependencies can be difficult to maintain. A good tool for incorporating different frameworks will lessen the complications, such as the Spring framework which is designed for framework integration.

Discontinuance of requirement support: A framework can evolve in a different direction than the application using it, possibly into a different domain, and thus not supporting the requirements of the application's domain. This means that the framework and the application does not evolve together, inhibiting the evolution of the application or forcing the developers to maintain the framework themselves.

8.5 Design process

The design process we went through when designing Joly, turned out to be more comprehensive than we originally planned. This was mainly due to the need for support of the non-functional requirements that originated from the DHIS requirements. For a small application as the Joly, the design process would normally not be this comprehensive, but

without the thorough design, we would probably not have achieved the flexibility that Joly now has. Designing for change by e.g. using patterns also prolonged the design process, but it is an important step in making the application able to easily adapt to changes in the future, as Wegner stated already in 1978 (quoted in [35]);

> *Construction of systems for evolution [so that modifications in response to changing requirements, improved methods, or the discovery of errors may be easily made,] may require greater overhead during development, but may result in enormous savings during operation and maintenance.*

We see it as probable that many developers will substitute the traditional design process of defining a set of patterns, with the use of frameworks. A problem we see relating to this, is that the application will be bound to the framework for achieving a flexible design, and thus will not be flexible in regards to changing the framework used. The ability to change the surrounding framework is necessary for an application which is aimed at a long operating time. One must take into consideration the possibility that the framework no longer exists at one point in the future, and the flexibility of the application can thus not rely on one specific framework.

The Spring framework's implementation of the DAO pattern is an example of a design that is bound to a specific framework. The DAO pattern used in combination with the Abstract Factory or Factory Method patterns (combined; the Factory pattern) makes the design flexible to adapting to different storage schemes, and de-couples the application from the implementing class. Spring provides this same kind of flexibility without the use of the Factory pattern, but through binding interfaces to the implementations in the application's context. It is necessary for the developers to be aware of these framework specific implementations of a pattern to design for flexibility not just within the framework chosen, but also for making the design flexible to be implemented in other frameworks. One way of doing this is to implement the design pattern as it originally is defined, but only use the part of it that is necessary for the chosen framework. This way the process of changing the framework would require less changes, as the flexibility the originally design pattern provides is already present.

The use of EJB in J2EE applications is often overkill, and design flaws in the EJB 2.0 specification has prevented bean-managed persistence to perform efficiently. Because of this we chose not to use EJB in the development of Joly, but we see the possibility that the new specification of EJB 3.0 could make it relevant for the further development of Joly. One of the reasons for choosing Hibernate and Hibernate Annotations for performing object to relational mappings in Joly was that it has already implemented an early draft of the EJB 3.0 specification. The process of implementing the final release of the new specification would therefore, after our opinion, require less change in the design. This is one of the changes in the design that we have been able to foresee and plan for, but it doesn't require that Joly fully implements the new specification but it is however likely if an upgrade of Hibernate is necessary.

In a system's operating time, new relevant specifications and technology will most certainly arise. It is important to evaluate the impacts these changes can have on the system's design and the costs of adapting it to fit the new requirements, before the changes are implemented. If the costs are too high, it can indicate that the system has reached the decline stage in the software maintenance life cycle (as described in chapter 3), and the new requirements should not be inducted. When a system reaches the decline stage in the maintenance process, as DHIS did with its version 1.4, a complete redesign and reimplementation is necessary to prolong the system's operating time. It is inevitable for a system to reach the decline stage as it ages, but the time it takes before it gets there can be prolonged by designing for change.

8.6 Incremental development process

The system development process for Joly has been done in Cathedral-style, which was the common development process before the entry of free and open source software, and the one used for developing DHIS1.3 and 1.4. The biggest change for the development of DHIS 2.0 is the change of the development process to the Bazaar-style. Investigating the impacts this kind of system development process can have on the resulting project, was outside the scope of our study. One experience we made however, with the Cathedral-model - which is even more relevant for the Bazaar-model, was the need for detailed and updated documentation during the development process. Even though we were just two developers working closely during the development of Joly, we sometimes lacked documentation about the parts of the

development the other person was working on. This lead to an extra iteration during the development process that was necessary to ensure that the design was consistent. For the DHIS developers, detailed and updated documentation is extremely important, as the development is performed by HISP teams in different countries. Inconsistency in the design could lead to a comprehensive iteration for the DHIS developers that could prolong the development process.

The design of an application should in any case be revaluated during the development to adapt to changes that can occur in the application's logic. We didn't experience any comprehensive changes during our development, but we saw the need for adding some new packages to the application as a result of many general classes (e.g. the Util-package in Joly) to improve the design.

A part of the incremental development process has been to run and test the Joly application continuously as new functions were added. This has forced us to fix bugs and improve the functionality during the development. The end result is that the application is able to run and function, with limitations in regards to the set of functions implemented and known errors described in appendix D. By integrating the Jetty Launcher plugin in the Eclipse IDE, we have also assured that the application is built and deployed correctly. At the finish of the development process, the application has been tested by packaging and deploying it to the Jetty server, making sure it ran and functioned properly.

8.7 Summary

This chapter has presented the experiences we have gained throughout our investigation, and we have emphasized considerations about design and maintenance that should be made when developing a free and open source web application with the use of frameworks. In the next chapter we present our conclusion, and our contributions to the DHIS 2.0 development.

Chapter 9

Conclusion

In this master thesis, we have investigated how to develop a web application by using free and open source tools, and how to design the application to easily adapt to change. During our investigation process we have evaluated FOSS products to determine their applicability to the Joly application, and whether they are suitable for the development of DHIS 2 as well. To investigate the tools' impact on the development and the final application, we have incorporated several free and open source frameworks that claim to provide benefits for the development of Java applications. Our focus has been to support the non-functional requirements through a thorough design process and by the reuse of software expertise found in design patterns and frameworks.

The functional requirements implemented has given us experience in the use of frameworks and the implementations they provide to solve common problems in Java applications. The implementation has resulted in the Joly prototype, which is a functioning web application that implements all main parts of a three-layered web application; a persistence logic, a domain logic and a presentation logic specific for a web interface.

We have experienced that employing frameworks in the development can result in a slow inception phase of the development process when the developers are unfamiliar with frameworks. Still, for the development of large-scale applications, we see that using such tools can be done with advantage because it can ease the remaining development, and increase the flexibility and quality of the resulting product. When using FOSS, it is important to define a set of criteria to use in an evaluation process both before the inception phase and

during the maintenance process. In the maintenance process the criterions can be used to insure that the employed solutions support new requirements imposed by changes in the application's environment. Designing for change can be a time-consuming part of the design process, but will result in a flexible application with the ability to adapt to future change, which also eases the maintenance process.

Criticism

Our main criticism when we look back on our investigation, is that because the design process turned out to be more comprehensive than we expected, we ended up with a lack of time to test the flexibility of the application. For instance, we have not experienced how well the application supports a substitution of the database management system. Testing the flexibility of the application is one of the next steps in the development process that we suggest for Joly.

Looking back, we see that by choosing to start the programming of Joly with the implementation of the function for submitting student assignments, we were introduced to the Spring framework by first learning one of its most advanced features. This has had an affect on the amount of time spent on learning the framework. We would probably not have used this much time if we had been aware of the extensiveness of the Spring wizard feature, and would instead have chosen to implement a simpler function to begin with.

In the same manner, we were introduced to the Hibernate framework by using their newest feature for mapping objects to relations, namely Hibernate Annotations, which is based on an early release of the new specification of EJB 3.0. Hibernate Annotations turned out to be an immature release, because it is based on an early and not final release of the specification, and the documentation was therefore poor. This resulted in another time-consuming process of learning a framework. These experiences show that the introduction of several frameworks in a development process, where the developers have no former knowledge of their use, was a poor choice.

Contributions to the development of DHIS 2

We have shown how HISP can design a flexible application that supports a requirement of platform-independency and makes it easier to adapt to software change. Our design has

enabled component-based logic for easy extension and change of implementations, and by encapsulating implementations and using framework-classes; the code is clear and easily maintainable.

Our experience has lead us to the opinion that free and open source products and tools can bring benefits to the development process, but impose some challenges that diminish their claimed advantages. We have experienced that the use of design patterns and frameworks increases the quality and flexibility of the application by reusing software expertise. However we have also seen challenges in applying these, because of the risk of not conforming to their intentions, and the process of determining their applicability to the application. In addition, the use of free and open source tools results in the application being dependent on a significant number of external tools and most likely will be dependant on other vendors upgrading these tools. Consequently, when employing several tools, the number of dependencies that needs to be maintained increases accordingly. This is one of the main drawbacks of using several external tools, and HISP should carefully consider the integration of many free and open source third-party tools.

Upgrades of external tools and packages used may no longer include features that the application is dependant on. Thus the DHIS developers should be prepared for the need to include the old class libraries in upgrades where the required features have been deprecated. Alternatively they can take part in the development community of the tool, and prolong the development of their required feature.

We have pointed out that dependencies to external tools will affect the maintenance process of the application, and all included tools should be reevaluated in each maintenance phase of the application. Still remaining is a study of how the dependencies affect the maintenance process and what impact the tools' own dependencies to other tools has. This is proposed for further work and should be explored by HISP.

Questions that remain to be answered

During our investigation we have come across some questions that are still to be answered. These should be considered before concluding whether the open source solutions approach is

the most appropriate for DHIS, and whether their end result will draw benefits from this approach.

- FOSS is often used to make free versions of tools and applications which have previously been developed by commercial vendors. It has proven to be effective this way, but the question that remains is whether an open source Bazaar-style development process is the best solution for developing new applications with functionality not already known from another application.

- The development of DHIS 2 has removed the local developer's ability to change the software in the same way as the Microsoft Access solution gave. The Access solution made it easier for less programming-skilled persons to change parts of the application. Access is a fourth generation environment where developers can use graphical elements to change the software, and it provides the fourth generation language Visual Basic for generating graphical user interface elements. The question that remains is whether a free and open source product will provide the same benefits to the local developers with less programming skills. The development process seems to be more professional and thus requires a higher level of computer and programming skills. Will this result in the system providing more for the administration in regards to costs, and for the current developers in regards to solving programming problems together, than providing for the end users and the health care data collection process in general?

- We have assumed that the costs related to the use of FOSS solutions are considerably lower than when using proprietary software. This assumption is based on the software being available free of charge. We request an analysis of the real total costs related to the use of proprietary software as opposed to free and open source software, throughout the lifetime of an application such as Joly or DHIS.

- The Joly application has been developed with the Cathedral model, whereas the DHIS 2 is being developed with the Bazaar model. Is this distinction a factor that inhibits the generalization of the problems we encountered, to the DHIS 2 system?

Proposals for further work

For the further work on Joly, we first recommend to implement all the intended functionality of the application, so that it can be deployed and used as a tool in the programming courses at the Institute of Informatics. A test phase, where the system's performance and scalability is measured, should be done after the application has been deployed in a real-life environment.

We also propose a study of the maintenance issues related to the system as an application that is made up of several external packages, and the dependencies this has imposed on the application. For such a study to be of best relevance, the application should have been operational for some time, so that new technology and significant software changes has emerged. The alternative would be to simulate a changing environment and expose the application to different types of change; different technology and software updates.

The Joly prototype is subject to be a starting point for various research, both related to DHIS and free and open source development in general. For DHIS it could be interesting to change the technology used and look for better technology choices. In particular, the development of a Mono application would be of interest for deciding whether Java or .NET is the most appropriate for a certain application. Whether this is related to DHIS or another system with different requirements, is up to the researcher as the Joly application can be seen as an implementation of a general web application. The Mono project was not mature enough for us to consider as an alternative, but the project should not be far from a mature release, and enable others to use Mono as a basis for comparison, without the trouble of using an immature platform and the start-up problems this involves.

As we pointed out earlier, we can not be certain that the generalization of our findings is correct, because DHIS is developed using the Bazaar-style of development. It could therefore be interesting to see if these two different approaches have impact on the resulting application, in terms of quality and the operating time of a system. Joly could be used for this kind of research, by running two development projects simultaneously with the two different system development models. Another study that also could be interesting is to explore what kind of criteria that needs to be present for users/developers other than the HISP teams to get involved in the development process, as the success of systems developed in Bazaar-style

requires continuous involvement from users that contribute to the development and maintenance of the system.

We also see it as interesting to study the flexibility of the application, and implement other layers to see how the domain logic can be accessed and used from other layers. What we see as most interesting, would be to apply a Service Oriented Architecture (SOA) and expose the domain logic as Web Services. SOA seems to be the new hype when designing web applications. Even though this is not needed by the current requirements of Joly, it is possible to explore what this type of architecture can bring to the DHIS system. We do not disregard the benefits web services could give the Joly application also, e.g. for integrating the application with other systems, such as the administrative systems of the University of Oslo.

Chapter **10**

Bibliography

[1] McIntyre, D. and L. Gilson, *Putting equity in health back onto the social policy agenda: experience from South Africa.* Social Science and Medicine, 2002. **54**(11): p. 1637-1656.

[2] Braa, J. and C. Hedberg, *The Struggle for District-Based Health Information Systems in South Africa.* The Information Society, 2002. **18**(2): p. 113-127.

[3] Congress, A.N., *White Paper on Reconstruction and Development*, A.N. Congress, Editor. 1994: Johannesburg.

[4] Congress, A.N., *A National Health Plan for South Africa*, P.b.t.A.w.t.t.s.o.W.a. UNICEF, Editor. 1994: Johannesburg.

[5] Titlestad, O. and J. Sæbø, *Evaluation of a bottom-up action research approach in a centralised setting: HISP in Cuba*. in *Proceedings of the 37th Annual Hawaii International Conference on System Science (HICSS'04)*. 2004. Hawaii.

[6] Braa, J., E. Monteiro, and S. Sundeep, *Networks of Action: Sustainable Health Information Systems Across Developing Countries.* MIS Quarterly, 2004. **28**(3): p. 337-362.

[7] Skobba, T.C., *Legacy systems and systems development in Mozambique : bridging the gap between the old and the new, showing the need for change*, in *Institute of Informatics, Faculty of Matematics and Natural Sciences*. 2003, University of Oslo.

[8] Wilson, R., et al., *South Africa's Health Information System: Case Study from Eastern Cape Province*. 2001. http://www.cpc.unc.edu/measure/rhino/rhino2001/theme2/safrica_paper.pdf (accessed 19.02.02).

[9] Abouzahr, C. and T. Boerma, *Health information systems: the foundations of public health*. 2005, Bull World Health Organ.

http://www.scielosp.org/scielo.php?pid=S0042-96862005000800010&script=sci_arttext (accessed 21.03.06).

[10] *Indicators to monitor maternal health goals*. WHO. http://www.who.int/reproductive-health/publications/MSM_94_14/MSM_94_14_table_of_contents.en.html (accessed 12.02.06).

[11] Wheeler, D.A., *Why Open Source Software / Free Software (OSS/FS, FLOSS, FOSS)? Look at the Numbers!* 2005. http://www.dwheeler.com/oss_fs_why.html (accessed 16.03.06).

[12] *Evaluate open source, or else*. 2004, SearchOpenSource.com. http://searchopensource.techtarget.com/qna/0,289202,sid39_gci999458,00.html (accessed 12.04.06).

[13] Stallman, R.M., *Free Software, Free Society: Selected Essays of Richard M. Stallman*, ed. J. Gray. 2002, Boston, USA: GNU Press, Free Software Foundation. ISBN 1-882114-98-1.

[14] Perens, B., *The Open Source Definition - Version 1.9*. 2006, Open Source Initiative. http://opensource.org/docs/definition.php (accessed 20.02.06).

[15] Holmes, C., A. Doyle, and M. Wilson, *Towards a Free and Open Source (FOSS) Spatial Data Infrastructure*. 2005. http://www.fig.net/pub/cairo/papers/ts_26/ts26_05_holmes_etal.pdf (accessed 11.04.06).

[16] *Memo: Comments on various issues raised on the HISP rollout & DHIS software*. 2002, HISP.

[17] Wilson, R., et al., *South Africa's Health Information System: Case Study from Eastern Cape Province*. 2001. (accessed 19.02.05)

[18] *DHIS documentation, Readme1st version 1.3*. 2003, DHIS. Distributed with DHIS version 1.3.0.x.

[19] Hedberg, C., *Hardware/Softeware for Primary Health Care Information Systems*. 2000. http://folk.uio.no/jumal/hardwarespecifications.htm (accessed 12.01.06).

[20] *DHIS documentation, Manual version 1.3*. 2003, HISP.

[21] Titlestad, O., *DHIS 1.4*. 2006. http://hips.ifi.uio.no:8080/confluence/display/DHIS1/DHIS+1.4 (accessed 14.03.06).

[22] Titlestad, O., *DHIS 2.0 decisions*. 2006, HISP. http://hips.ifi.uio.no:8080/confluence/display/DHIS2/DHIS+2.0+decisions (accessed 14.03.06).

[23] Titlestad, O., *Design and specification document - EU report (DHIS 2)*. 2005. http://hips.ifi.uio.no:8080/confluence/pages/viewpage.action?pageId=6090 (accessed 30.03.06).

[24] Tran, S.T.T., *DHIS 2 Overview*. 2005, DHIS. http://hips.ifi.uio.no:8080/confluence/download/attachments/3330/dhis+_2+overview.jpg (accessed 17.03.06).

[25. McCluskey, A., *Modularity: upgrading to the next generation design architecture*. 2000, e-news. http://www.connected.org/media/modular.html (accessed 30.03.06).

[26] Titlestad, O., *Basic overview of the design (DHIS 2.0)*. 2005, DHIS. http://hips.ifi.uio.no:8080/confluence/display/DHIS2/DHIS2+overview (accessed 20.03.06).

[27] Golden, B., *Succeeding with Open Source*. 2004, Boston, USA: Addison Wesley. ISBN 0-321-26853-9.

[28] Wheeler, D.A., *How to Evaluate Open Source Software / Free Software (OSS/FS) Programs*. 2006. http://www.dwheeler.com/oss_fs_eval.html (accessed 19.03.06).

[29] Wheeler, D.A., *Make Your Open Source Software GPL-Compatible. Or Else.* 2006. http://www.dwheeler.com/essays/gpl-compatible.html (accessed 23.04.06).

[30] *Various Licenses and Comments about Them*. 2006, Free Software Foundation. http://www.gnu.org/licenses/license-list.html (accessed 02.04.06).

[31] Raymond, E.S., *The Cathedral and the Bazaar*. 2000. http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ (accessed 08.11.05).

[32] St. Laurent, A.M., *Understanding Open Source and Free Software Licensing*. 2004: O'Reilly. 208. ISBN 0-596-00581-4.

[33] Blecherman, B., *The Cathedral versus the Bazaar (With apologies to Eric S. Raymond)*. 1999. http://www.ite.poly.edu/htmls/chapel_printable.htm (accessed 20.03.06).

[34] Metcalfe, R., *Top Tips for Selecting Open Source Software*. 2006, OSS Watch. http://www.oss-watch.ac.uk/resources/tips.xml (accessed 17.03.06).

[35] Greenfield, J. and K. Short, *Software Factories - Assembling Applications with Patterns, Models, Frameworks, and Tools*. 2004: Wiley Publishing, Inc. ISBN 666.0-471-20284-3.

[36] Tryggeseth, E., *Support for Understanding in Software Maintenance*, in *Department of Computer and Information Science, Faculty of Physics, Informatics and Mathematics*. 1997, Norwegian University of Science and Technology: Trondheim. p. 369.

[37] Bennett, K.H., et al., *Centres of Excellence: Research Institute in Software Evolution.* Computing and Control Engineering Journal, 2000. **11**(4): p. 179-186.

[38] Raja, U. and E. Barry, *Investigating Quality in Large-Scale Open Source Software*, in *Proceedings of the fifth workshop on Open source software engineering*. 2005, ACM Press: St. Louis, Missouri. p. 1-4.

[39] *IEEE Standard for Software Maintenance*. 1998, The Institute of Electrical and Electronics Engineers, Inc. http://ieeexplore.ieee.org/servlet/opac?punumber=5832 (accessed 07.03.06).

[40] Kung, H.-J., *Quantitative method to determine software maintenance life cycle.* IEEE International Conference on Software Maintenance, 2004: p. 232-241.

[41] Parnas, D.L., *Software Aging.* IEEE Proceedings of the 16th International Conference on Software Engineering, 1994: p. 279-287.

[42] *Java 2 Platform, Enterprise Edition (J2EE) Overview*. 2005, Sun Microsystems. http://java.sun.com/j2ee/overview.html (accessed 25.11.05).

[43] Broemmer, D., *J2EE Best Practices - Java Design Patterns, Automation and Performance*. 2003, Indianapolis: Wiley Publishing, Inc. ISBN 0-471-22885-0.

[44] Apache Software Foundation, A. http://geronimo.apache.org/ (accessed 08.11.05).

[45] Larman, C., *Applying UML and patterns - An introduction to object-oriented analysis and design and the unified process*. Second ed. 2001: Prentice Hall PTR. ISBN 0-13-092569-1.

[46] Berczuk, S., *Finding Solutions Through Pattern Languages.* Computer, 1994. **27**(12): p. 75-76.

[47] Cline, M.P., *The Pros and Cons of Adopting and Applying Design Patterns in the Real World.* Communications of the ACM, 1996. **39**(10): p. 47-49.

[48] Fowler, M., *Inversion of Control*. 2005. http://martinfowler.com/bliki/InversionOfControl.html (accessed 02.04.06).

[49] Fowler, M., *Inversion of Control Containers and the Dependency Injection pattern*. 2004. http://www.martinfowler.com/articles/injection.html (accessed 02.04.06).

[50] William Crawford, J.K., *J2EE Design Patterns*. 2003: O'Reilly. ISBN 0-596-00427-3.

[51] Ahamed, S.I.P., Ale. Pezewski, Al., *Towards framework selection criteria and suitability for an application framework*. in *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*. 2004: IEEE Xplore.

[52] Ridjanovic, D.O., V., *Using database framework in Web applications*. in *Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean*. 2004: IEEE Xplore.

[53] Raible, M., *Spring Live*. January/February ed. 2005: SourceBeat, LLC. ISBN 0-9748843-7-5.

[54] Hoeller, J., et al., *Professional Java Development with the Spring Framework*. 2005, Indianapolis: Wiley Publishing, Inc. ISBN 0764574833.

[55] Raible, M., *Overview of the Spring Framework*. 2005, SourceBean LLC.

[56] Krill, P., *Apache Avalon project closes down*. 2004, InfoWorld. http://www.infoworld.com/article/04/12/23/HNapacheavalon_1.html (accessed 19.04.06).

[57] Booch, G., I. Jacobsen, and J. Rumbaugh, *Unified Modeling Language Reference Manual*. Second Edition ed. 2004: Addison Wesley Professional. 752. ISBN 0-321-24562-8.

[58] Ambler, S.W., *The Design of a Robust Persistence Layer for Relational Databases*. 2005. http://www.ambysoft.com/downloads/persistenceLayer.pdf (accessed 30.04.2006).

[59] Bercich, N.H., *The Evolution of the Computerized Database*. 2003, Department of Computing Sciences. http://arxiv.org/ftp/cs/papers/0305/0305038.pdf (accessed 01.02.06).

[60] *Relational database management system*. 2006, Wikipedia. http://en.wikipedia.org/wiki/RDBMS (accessed 12.03.06).

[61] Broersma, M., *IDC: Quality Drives European Open-Source Adoption*. 2005, Linuxworld.com.au. http://www.linuxworld.com.au/index.php/id;1623210212;fp;16;fpid;0 (accessed 07.04.06).

[62] *Why MySQL?* 2006, MySQL AB. http://www.mysql.com/why-mysql/ (accessed 07.04.06).

[63] *Open Source Database Software Comparison*. 2005, Geocities. http://www.geocities.com/mailsoftware42/db/#speed (accessed 12.11.05).

[64] *PostgreSQL*. 2006, PostgreSQL. http://www.postgresql.org/ (accessed 11.02.06).

[65] *Firebird*. 2005, Firebird. http://firebird.sourceforge.net/index.php (accessed 09.12.05).

[66] *DHIS, Alternative Persitence Technologies*. 2005, HISP. http://hips.ifi.uio.no:8080/confluence/display/RandD/AP+Technologies (accessed 01.04.06).

[67] *Core J2EE Patterns - Data Access Object*. Sun Developer Network. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html (accessed 30.04.2006).

[68] Hoeller, J., et al., *Professional Java Development with the Spring Framework*. 2005, Indianapolis: Wiley Publishing, Inc. ISBN 0764574833.

[69] Objectmatter, *Visual BSF, Mapping Tool Guide*. 2003, Objectmatter. http://www.objectmatter.com/vbsf/docs/maptool/guide.html (accessed 30.04.2006).

[70] *JDBC Overview*. Sun Developer Network. http://java.sun.com/products/jdbc/overview.html (accessed 30.03.2006).

[71] Bauer, C. and G. King, *Hibernate in Action*. 2005: Manning Publications Co. ISBN 1932394-15-X.

[72] Clark, J. and R. Johnson, *Of Persistence and POJOs: Bridging the Object and Relational Worlds*. 2005, Oracle Technology Network. http://www.oracle.com/technology/pub/articles/masterj2ee/j2ee_wk9.html (accessed 30.03.2006).

[73] *WebSphere*. 2005, IBM. http://www-306.ibm.com/software/sw-bycategory/websphere/ (accessed 14.11.05).

[74] *Java Data Objects (DAO) Overview*. Sun Developer Network. http://java.sun.com/products/jdo/overview.html (accessed 31.03.2006).

[75] *Hibernate*. JBoss. http://www.hibernate.org/ (accessed 31.03.2006).

[76] *Hibernate 3.0*. 2005, JBoss. http://www.jboss.com/pdf/HibernateBrochure-Jun2005.pdf (accessed 01.04.2006).

[77] Alur, D., J. Crupi, and D. Malks, *Core J2EE Patterns*. 2001, Prentice Hall / Sun Microsystems Press. http://java.sun.com/blueprints/corej2eepatterns/index.html (accessed 15.11.05).

[78] Dass, K., *Model-View-Controller (MVC) Architecture*. 2006, indiawebdevelopers.com. http://www.indiawebdevelopers.com/technology/java/mvcarchitecture.asp (accessed 01.04.06).

[79] *Comparing web frameworks*. 2005, Theserverside. http://www.theserverside.com/news/thread.tss?thread_id=29817 (accessed 05.10.05).

[80] Raible, M., *Comparing web frameworks*. 2005, java.net. https://equinox.dev.java.net/framework-comparison/WebFrameworks.pdf (accessed 05.10.05).

[81] Lightbody, P., *WebWork - Inversion of Control*. 2006. http://wiki.opensymphony.com/display/WW/Inversion+of+Control (accessed 17.04.06).

[82] *MySQL Manual 4.1*. 2006, MySQL AB. http://dev.mysql.com/doc/refman/4.1/en/index.html (accessed 01.04.06).

[83] *Hibernate Reference Documentation*. 2005, Hibernate. http://www.hibernate.org/hib_docs/v3/reference/en/html/index.html (accessed 01.04.06).

[84] Bauer, C. and G. King, *Hibernate with a connection pool in a non-managed environment*. 2005, Manning Publications Co. p. From the book "Hibernate in Action", page 46.

[85] Alur, D., J. Crupi, and D. Malks, *Core J2EE Patterns - Best Practices and Design Strategies*. Second ed. 2003: Prentice Hall.0-13-142246-4.

[86] ChariotSolutions, *Request lifecycle*, http://springdeveloper.com/psug/Intro_to_Spring_MVC.pdf, Editor. 2005.

[87] Jackson, D. and M. Rinard, *Software analysis: a roadmap* *http://doi.acm.org/10.1145/336512.336545* in *Proceedings of the Conference on The Future of Software Engineering* 2000 ACM Press: Limerick, Ireland p. 133-145

[88] Fraser, S., et al., *Beyond the hype (panel): do patterns and frameworks reduce discovery costs? http://doi.acm.org/10.1145/263698.263757* in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems,*

languages, and applications 1997 ACM Press: Atlanta, Georgia, United States p. 342-344

Appendix A

The Joly database

The Joly database stores student-solutions delivered for an assignment in a course, and uses them to check for similarity when new student-solutions are handed in. In addition there are stored administrative information about students, employees, courses, groups and assignments.

Assumptions:

- Username is a unique identifier for students and employees.

A.1 EAR-diagram

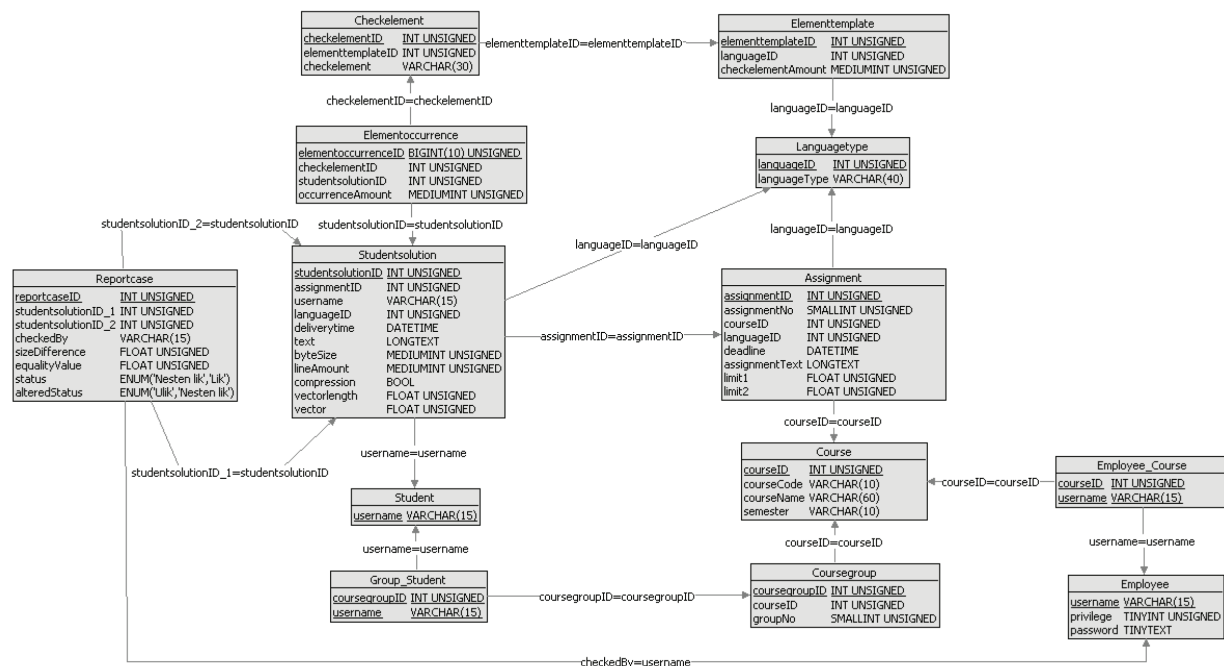


Figure A.1 EAR-diagram of the Joly database.

The EAR[1]-diagram shows a representation of the tables and columns in the database with entities and attributes. The relationships between the tables defined through primary- and foreign keys, are represented by a line between the entities with the attributes that couple the two entities together

A.2 Entities in the EAR-diagram

The following is a description of all the entities and their attributes in the EAR-diagram above. Relationships between two entities will be described together with the entity that is the owning side of the relationship, i.e. the entity that represents the "many" side in a many-to-one or a one-to-many relationship.

A.2.1 Elementtemplate

Elementtemplate holds information about a template that is used to check studentsolutions for similarity with other handed in studentsolutions.

Attribute	Description	Datatype	Key
elementtemplateID	Identifies a template. Is automatically incremented.	INT UNSIGNED	Primary
languageID	The languagetype the template is defined for.	INT UNSIGNED	Foreign
checkelementAmount	The number of elements a template uses to check for similarity.	MEDIUMINT UNSIGNED	

Relationship:

- *Elementtemplate –Languagetype* (many-to-one):
  An *Elementtemplate* is connected to one *Languagetype*, and a *Languagetype* can be connected to zero or many *Elementtemplate*'s.

[1] EAR = Entity And Relationship

A.2.2 Checkelement

Checkelement holds information about a checkelement used for an elementtemplate, e.g."if{", "for(" or "while{". It is normally 3-15 different checkelements per elementtemplate (but there are no restrictions).

Attribute	Description	Datatype	Key
checkelementID	Identifies a checkelement. Is automatically incremented.	INT UNSIGNED	Primary
elementtemplateID	Identifies the elementtemplate the checkelement belongs to.	INT UNSIGNED	Foreign
checkelement	A string that the elementtemplate uses to check studentsolutions for similarity (the string is pieces of code like e.g."for(" or "while{".	VARCHAR	

Relationship:

- *Checkelement –Elementtemplate (many-to-one):*
  A *Checkelement* is connected to one *Elementtemplate*, and an *Elementtemplate* can be connected to zero or many *Checkelement*'s.

A.2.3 Elementoccurrence

Elementoccurrence holds information about the number of occurrences of a specific checkelement in a studentsolution.

Attribute	Description	Datatype	Key
elementoccurrenceID	Identifies an elementoccurence. Is automatically incremented.	BIGINT UNSIGNED	Primary
checkelementID	Identifies the checkelement the elementoccurence belongs to.	INT UNSIGNED	Foreign
studentsolutionID	Identifies a student solution that the number of checkelement occurences is counted for.	INT UNSIGNED	Foreign
occurrenceAmount	The number of occurrences of a checkelement in a studentsolution.	MEDIUMINT UNSIGNED	

Relationships:

- *Elementoccurrence* - *Checkelement* (many-to-one):
  An *Elementoccurrence* is connected to one *Checkelement*, and one *Checkelement* can be connected to zero or many *Elementoccurrence*'s.
- *Elementoccurrence – Studentsolution* (many-to-one):
  An *Elementoccurrence* is connected to one *Studentsolution*, and one *Studentsolution* can be connected to zero or many *Elementoccurrence*'s.

A.2.4 Studentsolution

A studentsolution holds information about an assignment a student has handed in. Because a student has the possibility of handing in the same assignments several times, the entity needs a unique identifier, *studentsolutionID*, in addition to the foreign keys.

Attribute	Description	Datatype	Key
studentsolutionID	Identifies a studentsolution. Is automatically incremented.	INT UNSIGNED	Primary
assignmentID	The identifier of the assignment the studentsolution is handed in for.	INT UNSIGNED	Foreign
username	The students username.	VARCHAR	Foregin
languageID	The programming language the assignment is written in.	INT UNSIGNED	Foreign
deliveryTime	Time and date for the delivery of the studentsolution.	DATETIME	
text	The text of the studentsolution for an assignment.	LONGTEXT	
byteSize	The length of the text in number of bytes when it's stripped for comments and spaces (for comparison against other studentsolutions).	MEDIUMINT UNSIGNED	
lineAmount	The number of lines in the text, including comments.	MEDIUMINT UNSIGNED	
compression	Indicates whether the studentsolution is compressed or not (default: 0 - i.e. no compression)	BOOL	
vectorlength	The length of the vector used in the algorithm to check for similarity.	FLOAT UNSIGNED	
vector	The vector used in the algorithm to check for similarity.	FLOAT UNSIGNED	

Relationships:

- *Studentsolution – Assignment* (many-to-one):
  A *Studentsolution* is connected to one *Assignment*, and one *Assignment* can be connected to zero or many *Studentsolution*'s.
- *Studentsolution – Student* (many-to-one):
  A *Studentsolution* is connected to one *Student*, and a *Student* can be connected to zero or many *Studentsolution*'s.
- *Studentsolution – Languagetype* (many-to-one):
  A *Studentsolution* is connected to one *Languagetype*, and one *Languagetype* can be connected to zero or many *Studentsolution*'s.

A.2.5 Reportcase

A reportcase holds information about a check for similarity that ended with two studentsolution beeing suspicious.

Attribute	Description	Datatype	Key
reportcaseID	Identifies a reportcase. Is automatically incremented.	INT UNSIGNED	Primary
studentsolutionID_1	The identifier of a studentsolution.	INT UNSIGNED	Foreign
studentsolutionID_2	The identifier of a studentsolution.	INT UNSIGNED	Foreign
checkedBy	The username of an employee that has evaluated the two studentsolutions and sat the *alteredStatus*.	VARCHAR	Foreign
sizeDifference	The largest studentsolutions size divided by the smallest studentsolutions size. Calculated by the algorithm.	FLOAT UNSIGNED	
equalityValue	The angle betewwn the two studentsolutions vectors. Calculated by the algorithm.	FLOAT UNSIGNED	
status	The status of the similarity between the two studentsolutions, sat by the algorithm.	ENUM('Nesten lik', 'Lik')	
alteredStatus	The status of the similarity between the two studentsolutions, sat by an employee. Default is 'Usjekket'	ENUM('Usjekket' 'Ulik', 'Nesten lik')	

Relationships

- *Reportcase-Studentsolution (studentsolutionID_1 – studentsolutionID)* (one-to-one):
  A *Reportcase* is connected to one *Studentsolution* for the studentsolution identified by the attribute *studentsolutionID_1*.
- *Reportcase-Studentsolution (studentsolutionID_2 – studentsolutionID)* (one-to-one):
  A *Reportcase* is connected to one *Studentsolution* for the studentsolution identified by the attribute *studentsolutionID_2*.

A.2.6 Assignment

An assignment holds information about assignments in a course. Because the assignment-number can be similar for different courses, the identifier of the entity is *assignmentID* that uniquely identifies an assignment.

Attribute	Description	Datatype	Key
assignmentID	Identifies an assignment. Is automatically incremented.	INT UNSIGNED	Primary
assignmentNo	The number of the assignment (e.g. 1, 2, 3, ...).	SMALLLINT UNSIGNED	
courseID	Identifies the course the assignment belongs to.	INT UNSIGNED	Foreign
languageID	The identifier of the languagetype (programming language) the assignment is to be written in.	INT UNSIGNED	Foreign
deadline	The deadline (time and date) for delivery of an assignment.	DATETIME	
assignmentText	The text of the assignment.	LONGTEXT	
limit1	Limit 1 used by the algorithm to determine similarity.	FLOAT UNSIGNED	
limit2	Limit 2 used by the algorithm to determine similarity.	FLOAT UNSIGNED	

Relationships:

- *Assignment – Course* (many-to-one):
  An *Assignment* is connected to one *Course*, and a *Course* can be connected to zero or many *Assignment*'s.
- *Assignment – Languagetype* (many-to-one):
  An *Assignment* is connected to one *Languagetype*, and a Languagetype can be connected to zero or many *Assignment*'s.

A.2.7 Course

A *Course* holds information about a course. Because a course can be hold several times with the same course code and course name, there is a need for an identifier that uniquely identifies a course, *courseID*.

Attribute	Description	Datatype	Key
courseID	Identifies a course. Is automatically incremented.	INT UNSIGNED	Primary
courseCode	Identifier for a course used at the University of Oslo, e.g.”INF1010”.	VARCHAR	
courseName	The full name of the course, e.g. ”Objektoriented programming”.	VARCHAR	
semester	A course can be held in both spring and fall semester (e.g. V2006 or H2006).	VARCHAR	

A.2.8 Coursegroup

A *Coursegroup* holds information about a course that belongs to a course.

Attribute	Description	Datatype	Key
coursegroupID	Identifies a coursegroup. Is automatically incremented.	INT UNSIGNED	Primary
courseID	The identifier of the course the group belongs to.	INT UNSIGNED	Foreign
groupNo	The number of the group.	SMALLINT UNSIGNED	

Relationship:

- *Coursegroup – Course* (many-to-one):
  A *Coursegroup* is connected to one *Course*, and a *Course* can be connected to zero or many *Coursegroups.*

A.2.9 Student

Student holds information about a student that participates in a course. A students username is uniqe.

Attribute	Description	Datatype	Key
username	Identifies a student.	VARCHAR	Primary

A.2.10 Group_Student

A *Group_Student* is a coupling entity used to remove the many-to-many relationship between *Student* and Course*group*, and holds information about a student's participation on a group in a course. The primary key consists of the primary keys from the two entities; *coursegroupID* and *username*.

Attribute	Description	Datatype	Key
coursegroupID	Identifies a coursegroup.	INT UNSIGNED	Primary and foreign
username	Identifies a student.	VARCHAR	Primary and foreign

Relationships:

- *Group_Student – Student* (many-to-one):
  A *Group_Student* is connected to one *Student*, and a *Student* can be connected to zero or many *Group_Student*'s.
- *Group_Student – Coursegroup* (many-to-one):
  A *Group_Student* is connected to one *Coursegroup*, and a *Coursegroup* is connected to zero or many *Group_Student*'s.

A.2.11 Employee

An *Employee* holds information about a teaching assistant, a head teaching assistant, a lecturer or an administrator that is allowd to use JolyAdmin. An employees username is uniqe.

Attribute	Description	Datatype	Key
username	Identifies an employee.	VARCHAR	Primary
privilege	States which user privileges an employee has. Privilege-levels: Teaching assistant: 1 Head teaching assistant/lecturer: 2 Administrator: 3	TINYINT UNSIGNED	
password	The password the employee uses to login to JolyAdmin.	TINYTEXT	

A.2.12 Employee_Course

An *Employee*_Course is a coupling entity used to remove the many-to-many relationship between *Employee* and *Course*, and holds information about an employee's courses. The primary key consists of the primary keys from the two entities; *username* og *courseID*.

Attribute	Description	Datatype	Key
courseID	Identifies a course.	INT UNSIGNED	Primary and foreign
username	Identifies an employee.	VARCHAR	Primary and foreign

Relationships:

- *Employee_Course – Employee* (many-to-one):
  An *Employee_Course* is connected to one *Employee*, and an *Employee* can be connected to zero or many *Employee_Course*'s.
- *Employee_Course – Course* (many-to-one):
  An *Employee_Course* is connected to one *Course*, and a *Course* can be connected to zero or many *Employee_Course*'s.

Appendix **B**

Original design documentation for Joly

The following document is the original design document of the Joly application. It is written in Norwegian, and primarily appended with this thesis with thoughts for those specially interested in the future development of Joly. The documentation describes the original design and the functional requirements that the application should support.

JOLY

WEBLØSNING FOR INNLEVERING AV STUDENTOPPGAVER

Innhold

1. INNLEDNING ... 198

2. BRUKERGRENSESNITTENE ... 198

2.1 STUDENT-BRUKERGRENSESNITTET ... 198
2.1.1 Use case-diagram ... 199
2.1.2 Brukerscenario ... 199
2.1.3 Sitemap ... 202
2.1.4 Scenario for systemet ... 202
2.1.5 Aktivitetsdiagram ... 204
2.1.6 Mulige utvidelser av funksjonalitet ... 205
2.2 FAGANSATT-BRUKERGRENSESNITTET ... 205
2.2.1 Use-case diagram ... 205
2.2.2 Brukerscenario for gruppelærer ... 206
2.2.1.1 Innlogging ... 206
2.2.1.2 Legg inn innleveringer ... 206
2.2.1.3 Innleveringer ... 208
2.2.1.4 Studentopplysninger ... 211
2.2.1.5 Utlogging ... 212
2.2.3 Sitemap for gruppelærer-brukergrensesnittet ... 212
2.2.4 Aktivitetsdiagram for brukergrensenittet til gruppelærer ... 213
2.2.4.1 Aktivitetsdiagram for "Innlogging" ... 213
2.2.4.2 Aktivitetsdiagram for "Legg inn innleveringer: Legg inn innlevering til en bestemt student" ... 214
2.2.4.3 Aktivitetsdiagram for "Legg inn innleveringer:Legg inn fasiter og eierløse innleveringer" ... 215
2.2.4.4 Aktivitetsdiagram for "Innleveringer" ... 216
2.2.4.5 Aktivitetsdiagram for studentopplysninger ... 217
2.2.4.6 Aktivitetsdiagram for "Utlogging" ... 217
2.2.5 Brukerscenario for foreleser og hovedgruppelærer ... 217
2.2.5.1 Innlogging ... 217
2.2.5.2 Legg inn innleveringer ... 217
2.2.5.3 Innleveringer ... 218
2.2.5.4 Gruppeopplysninger ... 219
2.2.5.5 Obligopplysninger ... 221
2.2.5.6 Obligopplysninger ... 221
2.2.6 Sitemap for foreleser/hovedgruppelærer-brukergrensesnittet ... 222
2.2.7 Aktivitetsdiagram for brukergrensesnittet til foreleser og hovedgruppelærer ... 222
2.2.7.1 Aktivitetsdiagram for "Gruppeopplysninger" ... 223
2.2.7.2 Aktivitetsdiagram for "Obligopplysninger" ... 224
2.2.8 Mulige utvidelser av funksjonalitet ... 224
2.3 ADMINISTRATOR-BRUKERGRENSESNITTET ... 225
2.3.1 Use case-diagram ... 225
2.3.2 Brukerscenario for administrator ... 225
2.3.2.1 Innlogging ... 225
2.3.2.2 Legg inn innleveringer ... 226
2.3.2.3 Innleveringer ... 226
2.3.2.4 Studentopplysninger ... 226
2.3.2.5 Gruppeopplysninger ... 226
2.3.2.6 Obligopplysninger ... 226
2.3.2.7 Kursopplysninger ... 226
2.3.2.8 Fagansattopplysninger ... 226
2.3.2.9 Mal ... 226
2.3.2.10 Utlogging ... 226

1. Innledning

For å undersøke om studenter fusker på studentoppgaver, er det tenkt å utvikle en webløsning hvor studenter innleverer sine oppgaver og disse blir sjekket opp mot fasiter og andre studenters innleveringer. Hjelpelærere og forelesere kan så gå inn og hente ut innleveringene for retting og godkjenning, samtidig som de kan hente ut rapporter om eventuelle innleveringer som er mistenkt for likhet. Systemet har fått navnet Joly etter korrupsjonsjegeren Eva Joly.

2. Brukergrensesnittene

Webløsningen vil bestå av tre forskjellige brukergrensesnitt; én for studenter for innlevering av studentoppgaver, én for forelesere, hovedgruppelærere og gruppelærere (fagansatte) for uthenting av studentinnleveringer og rapporter om fusking, og én for administratoren av webløsningen for administrering av data om kurs, studenter og gruppelærere/forelesere, samt malene som blir benyttet for å sjekke innleveringene for fusk.

De følgende skjermbildene er laget for å illustrere hvordan brukergrensesnittet kan være.

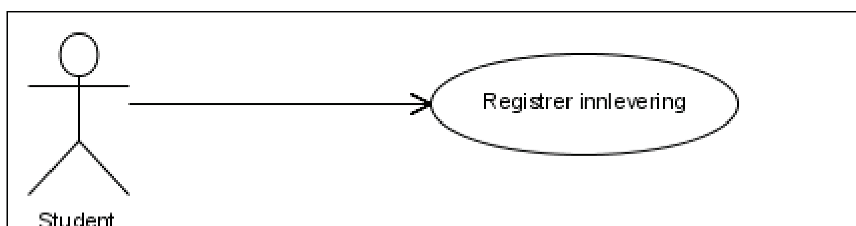
2.1 Student-brukergrensesnittet

Grensesnittet for studentene skal gi brukern mulighet til å laste opp oppgave-filer, registrere innleveringen under riktig kurs og oppgavenummer, og få en tilbakemelding på at oppgaven er registrert. Det vil alltid være den siste innleveringen en student gjør som er den gyldige, dvs at dersom en student velger å innlevere den samme oppgaven flere ganger, vil det være den siste innleveringen som gruppelæreren godkjenner.

Det vil være innleveringsfrister (dato og tidspunkt) for studentoppgavene. En student har likevel mulighet til å levere inn en oppgave også etter at tidsfristen har gått ut, da det kan være en avtale mellom gruppelærer og student om at oppgaven kan innleveres senere. Gruppelærer vil få beskjed om innleveringer som gjøres etter at fristen har gått ut.

En innlevering kan bestå av flere filer og ikke bare Java-filer (se for eksempel Oblig 1 i INF1010, http://www.ifi.uio.no/~inf1010/obliger/oblig1.html). En student må derfor ha mulighet til å laste opp mer enn en fil. Det kan kun være en programfil (Java-fil) og det vil kun være den som skal legges inn i databasen og sjekkes mot andre studentinnleveringer, men alle filene sendes til gruppelærer på mail.

2.1.1 Use case-diagram



Use case-diagram for student-brukergrensesnittet.

2.1.2 Brukerscenario

1. Studenten velger kurskode fra nedtrekksliste.
2. Studenten velger oppgavenummer fra nedtrekksliste for obligatoriske oppgaver som kurset er registrert med.
3. Studenten velger program-filen som skal innleveres fra en fildialog og velger å legge til filen til innleveringen ved å trykke på knappen ”Legg til”. For å unngå *spamming* begrenses filstørrelsen på programfilen til 50 KB.
4. Dersom filen som legges til ikke er av korrekt type (.java) får studenten opplyst dette, filen legges ikke til innleveringen, og studenten er tilbake ved steg 3. Hvis filen er av korrekt type får studenten oversikt over program-filen som er lagt til innleveringen med mulighet til å fjerne den. Studenten kan kun legge til én program-fil til en innlevering og vil få beskjed om dette dersom han prøver å legge til flere. Dersom studenten ønsker at en annen program-fil skal sendes sammen med innleveringen istedenfor den som allerede er lagt til, må filen fjernes ved å klikke ”Fjern” ved siden av filnavnet, før en ny program-fil kan legges til (tilbake til steg 3).
5. Studenten kan i tillegg legge til andre filer som er en del av den obligatoriske oppgaven, og som skal sendes til gruppelærer per e-post. Filer velges i en fildialog og legges til innleveringen ved å trykke på knappen ”Legg til”, og studenten får oversikt over filene som er lagt til innleveringen.
6. Dersom studenten ønsker å innlevere flere filer til den obligatoriske oppgaven, utføres steg 5 igjen. Studenten har også mulighet for å fjerne disse filene fra innleveringen ved å klikke ”Fjern” ved siden av filnavnet.
7. Studenten har mulighet for å skrive inn tilleggskommentarer (f.eks. at innleveringen er et samarbeid mellom to studenter og legge til informasjon om den andre studenten innleveringen gjelder for) i en tekstboks som sendes sammen med e-posten til gruppelæreren.
8. Studenten skriver inn brukernavnet sitt.
9. Studenten skriver inn gruppenummeret på gruppen han ønsker å gjøre innleveringen til.
10. Studenten trykker på knappen ”Videre” for å registrere studentinnleveringen.
11. Dersom ikke alle de nødvendige opplysningene er fylt inn (det behøver ikke være lagt til andre filer), blir studenten fremvist skjemaet igjen med informasjon om hvilke felter som må fylles inn.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Kurs: INF1000

Oppgave: oblig1

Last opp fil :

Programfil: Bla gjennom...

Legg til

oblig1.java fjern

Andre: Bla gjennom...

Legg til

oblig1.pdf fjern

utfil.txt fjern

Tilleggskommentar:

Brukernavn: hannevi

Gruppenr: 1

videre>>

Skjermbilde steg 1-10

12. Dersom program-filen som studenten har lagt til innleveringen overstiger den begrensede filstørrelsen, eller brukernavnet til studenten ikke eksisterer i databasen vil studenten sendes til en egen feilside hvor han har mulighet til å gjøre eventuelle endringer for at innleveringen skal kunne sendes inn.
    a. Dersom filstørrelsen overstiger begrensningen, må studenten fjerne program-filen og legge til en ny program-fil som ikke er større enn begrensningen og klikke ”Videre” for å registrere innleveringen, eller velge ”Avbryt” for å avbryte hele innleveringen.
    b. Dersom studenten får beskjed om at brukernavnet ikke eksisterer i databasen, gis det en feilmelding som forteller at studenten har skrevet brukernavnet feil eller at det ikke er registrert i databasen. Studenten får da to valg:
        i. Brukernavnet er skrevet feil: Studenten får da mulighet til å endre brukernavnet dersom han ser at det er skrevet feil og klikke ”Videre” for igjen å sjekke om brukernavnet nå eksisterer i databasen. Dersom brukernavnet eksisterer i databasen blir studenten da sendt til skjermbildet til steg 13, ellers tilbake til steg 12 igjen.
        ii. Brukernavnet er skrevet riktig: Studenten ser at brukernavnet er skrevet riktig - studenten er da ikke registrert i databasen. Studenten har mulighet til å avbrute innleveringen ved å klikke ”Avbryt”. Dersom studenten ønsker å innlevere oppgaven selv om han ikke er registrert er dette mulig og studenten vil bli sendt til skjermbildet til steg 13. Innleveringen vil da ikke lagres i databasen – kun sendes til gruppelærer som får beskjed om situasjonen. Studenten får også beskjed om at han bør kontakte administrasjonen for å sørge for at han er registrert i systemet.

Kurs: INF1000
Oppgave: oblig1

Last opp fil :
Programfil: Browse... Legg til
oblig1.java fjern
Andre: Browse... Legg til
oblig1.pdf fjern
utfil.txt fjern

Feil ved opplasting
Filstørrelsen på filen *oblig1.java* overskrider 50kb. Maks filstørrelse per fil er 50kb.

Tilleggskommentar:

Brukernavn: hannevi
Gruppenr: 1

Angitte brukernavn er ikke registrert i systemet. Dersom du allikevel ønsker å registrere innleveringen under dette brukernavnet, klikk "videre". Innleveringen din vil bli sendt til gruppelærer på angitte gruppe for godkjenning. Ta kontakt med administrasjonen for å sørge for at du blir registrert i systemet.

avbryt videre>>

Skjermbilde for steg 12.

13. Dersom opplysningene ikke inneholder noen feil, blir studenten fremvist opplysningene han/hun har registrert og forespurt om opplysningene er korrekte før oppgaven registreres. Dersom tidspunktet for innleveringen ikke er innenfor tidsfristen for innleveringen av den obligatoriske oppgaven, blir studenten opplyst om dette. Studenten får likevel innlevere oppgaven, men blir opplyst om at overskridelsen må klareres med gruppelærer dersom den skal vurderes for godkjenning.
14. Dersom studenten ønsker å korrigere opplysningene som er registrert, trykker studenten knappen "Korriger" og blir fremvist det første skjermbildet (steg 1) og har mulighet til å redigere opplysningene. Når studenten er ferdig med redigeringene, gjentas steg 10.
15. Dersom studenten ikke ønsker å korrigere opplysningene, trykkes knappen "Send inn".

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Registrerte opplysninger:

Brukernavn:	hannevi
Gruppenr:	1
Kurs:	INF1000
Oppgave:	oblig1
Fil:	oblig1.java, oblig1.pdf, utfil.txt

<< korrigér send inn >>

Skjermbilde steg 13-15.

16. Studenten får bekreftelse på at oppgaven er registrert, med mulighet for å skrive ut en kvittering på dette.

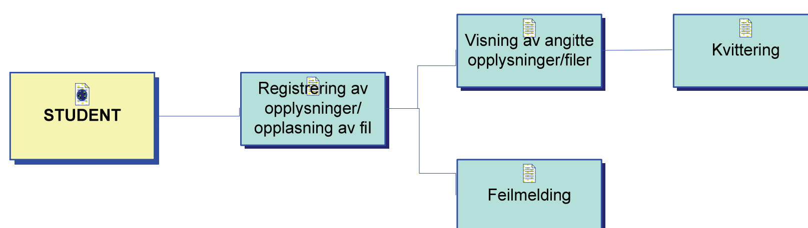
Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Kvittering for innlevert obligatorisk oppgave	
Dato:	16.07.2005, kl. 16.08
Brukernavn:	hannevi
Gruppenr:	1
Kurs:	INF1000
Oppgave:	oblig1
Fil:	oblig1.java, oblig1.pdf, utfil.txt

skriv ut

Skjermbilde steg 16.

2.1.3 Sitemap



Sitemap for student-brukergrensenittet.

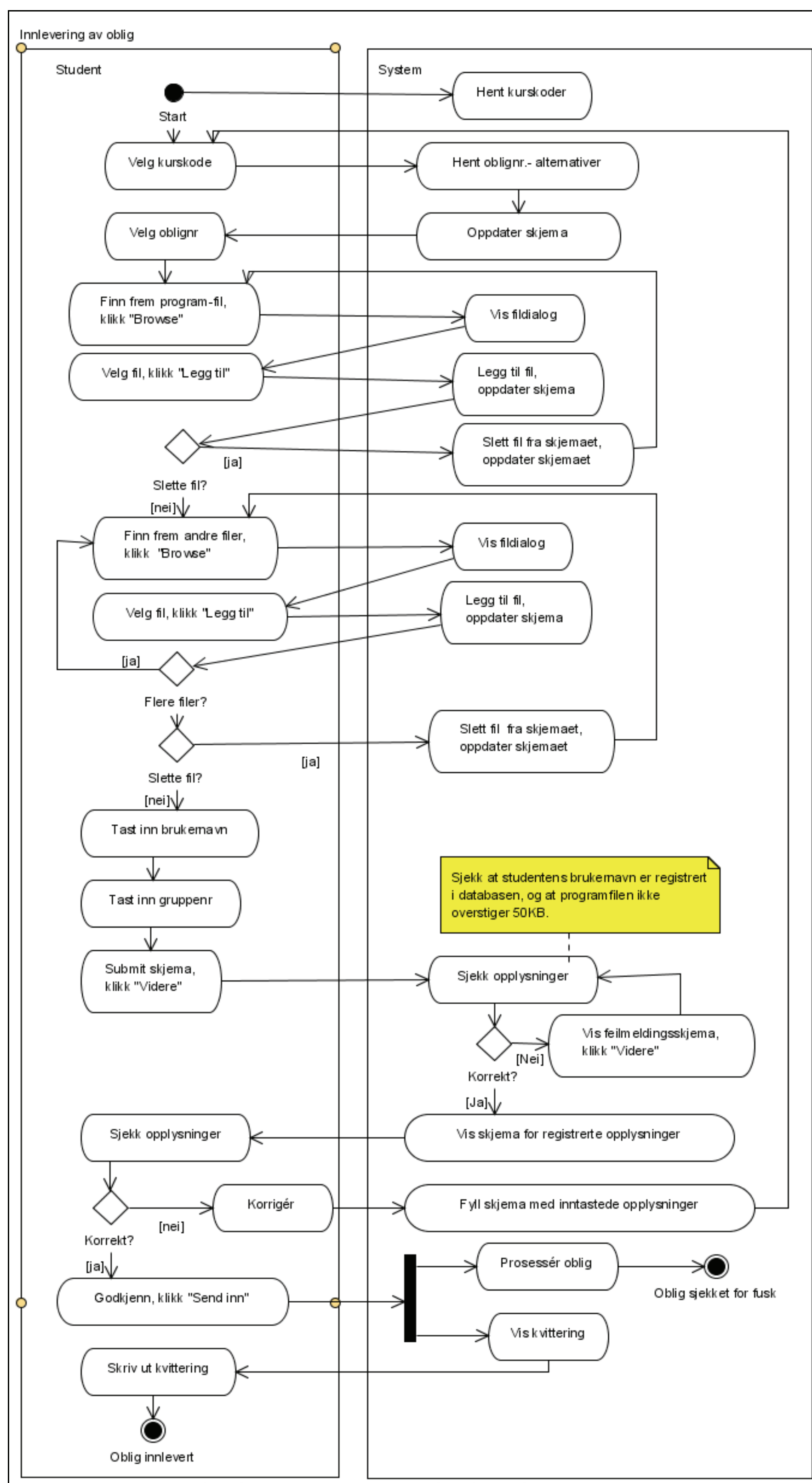
2.1.4 Scenario for systemet

1. Før studenten får opp det første skjemaet for innlevering av oppgave, hentes alle kurskodene som er lagret i databasen og nedtrekkslisten med kurskoder fylles opp.
2. Etter at studenten har valgt kurskode fra nedtrekkslisten gjøres et kall til databasen som finner alle oppgavenummer for innleveringer i det valgte kurset, og fyller opp nedtrekkslisten for oppgavenummer med disse.
3. Når studenten ønsker å velge program-fil som skal innleveres vises en fildialog over filer på disken som studenten kan browse igjennom.
4. Etter at studenten har valgt å legge til ønsket program-fil, sjekkes det at filen er av riktig type (.java), og får beskjed om det dersom den ikke er det og scenarioet er tilbake ved steg 3. Dersom riktig filtype er lagt til får studenten oversikt over filen, samt mulighet for å fjerne den. Dersom studenten trykker på ”Fjern” ved siden av filnavnet, fjernes filen fra oversikten og scenarioet går tilbake til steg 3. Dersom studenten prøver å legge til mer enn én program-fil, får studenten en feilmelding som opplyser om at dette ikke er mulig.

5. Når studenten ønsker å velge andre filer som skal legges til innleveringen, vises en fildialog over filer på disken som studenten kan browse igjennom.
6. Etter at studenten har valgt å legge til en ny fil, får studenten oversikt over filen sammen med andre filer i en liste, med lik mulighet for å fjerne en fil som for en program-fil.
7. Dersom studenten ønsker å legge til flere filer gjentas sceanrioet fra og med steg 5 inntil studenten har lagt til alle ønskelige filer til innleveringen.
8. Når studenten klikker ”Videre” etter at alle opplysningene er registrert, sjekkes følgende:
   a. At nødvendig informasjon er fylt inn, dvs at kurskode og oppgavenummer er valgt, en program-fil er lagt til og at brukernavn og gruppenummer er registrert.
   b. At filstørrelsen til program-filen ikke overstiger 50KB.
   c. At brukernavnet eksisterer i databasen.

   Dersom noen av de ovenstående tilfellene ikke er tilfredsstilt vil studenten bli opplyst om dette. Dersom punkt a ikke er oppfylt, vil studenten få opplyst dette i det første skjermbildet og få mulighet til å registrere de nødvendige opplysningene. Dersom punkt b eller c ikke er oppfylt vil studenten fremvises en egen feilside hvor opplysningene kan endres.
9. Dersom opplysningene ikke inneholder noen feil, fremvises studenten en oversikt over de registrerte opplysningene. Hvis studenten ønsker å korrigere de registrerte opplysningene taster studenten ”Korriger”-knappen, og de registrerte opplysningene fylles da inn igjen i det første skjermbildet og scenarioet starter igjen ved steg 8 når studenten har korrigert opplysningen.
10. Dersom studenten velger ”Send inn” starter prosessen med å behandle innleveringen:
    a. Dersom gruppenummeret studenten har innlevert oppgaven til ikke eksisterer i databasen, opprettes den.
    b. Java-filen åpnes og legges til informasjon om innleveringen (dato/brukernavn/kurs/oppgavenummer/filnummer/gruppenr) til første linje i filen som en kommentar.
    c. Komprimerings-metoden kalles og resultatet er en string som benyttes for å generere lengden på filen.
    d. Filen gjennomgår en sjekk for å finne elementene i filen, samt beregne vektorlengden.
    e. Filen sammenlignes mot andre innleveringer i databasen (snevrer inn søket ved bare å se på innleveringer av en viss lengde).
11. Når systemet er ferdig med å sjekke innleveringen for likheter mot andre innleveringer, sendes en mail til gruppelærer med informasjon og alle filene studenten la til innleveringen, samt informasjon om søket etter like innleveringer.
12. Dersom en innlevering er mistenkt for likhet med en annen innlevering sendes også en mail til foreleser(e) i kurset om dette. Siden denne funksjonen kan bidra til at foreleser(e) får veldig mange mail’er, skal administrator kunne ”skru av” denne funksjonaliteten.

2.1.5 Aktivitetsdiagram



Aktivitetsdiagram for student-brukergrensesnittet

2.1.6 Mulige utvidelser av funksjonalitet

I den første versjonen av systemet trenger ikke studenter passord for å innlevere en oppgave, men dette kan være mulig utvidelse av systemet. En student kan da logge seg på med brukernavn og passord, og deretter komme til en personlig side med oversikt over innleveringer som har blitt gjort i de forskjellige kursene en student deltar på, og muligheter for å innlevere nye oppgaver.

I den første versjonen av systemet har studenten mulighet til å laste opp flere filer som del av en obligatorisk oppgave, men det er vil kun være mulig å laste opp én fil eller én zip, så funksjonalitet for å laste opp flere filer kan legges til i fremtidige versjoner.

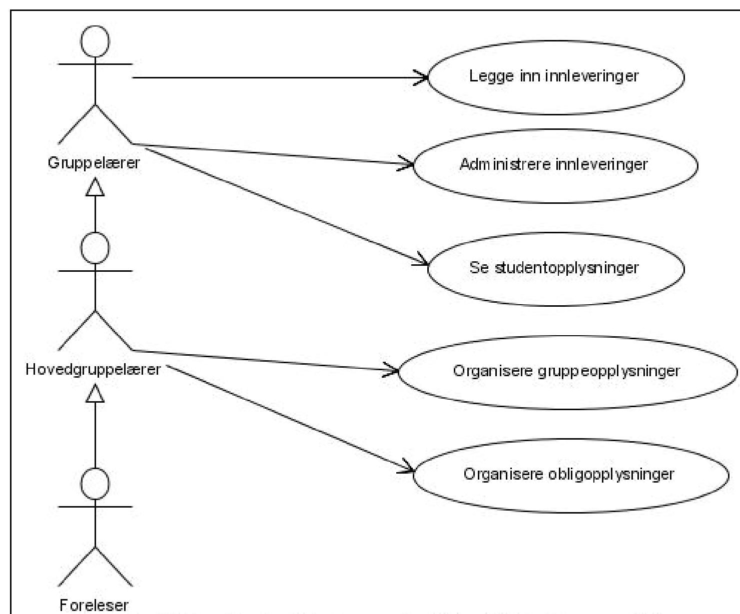
I den første versjonen av systemet vil brukergrensesnittet være på norsk, så en senere utvidelse kan være å utvide brukergrensesnittet til engelsk.

For å forhindre at student-brukergrensesnittet benyttes for å *spamme* systemet, kan en mulig utvidelse være å kreve at studentene må taste inn en kode/passord for hver oblig-innlevering. Denne koden kan kun fås på IFI via en Unix-kommando (tilsvarende adgangskoden til terminalstuen på IFI).

2.2 Fagansatt-brukergrensesnittet

Grensesnittet for de fagansatte skal gi gruppelærere, hovedgruppelærere og forelesere mulighet til å hente ut studentoppgaver som er innlevert for godkjenning, samt rapporter om mulighet for likhet i innleverte oppgaver mot andre innleverte oppgaver. Gruppelærere bruker et eget system for å registrere godkjente oppgaver, og systemet er derfor ikke tenkt til å benyttes til dette. Gruppelærer, hovedgruppelærer og foreleser har forskjellig privilegienivå og tilgang på funksjoner i systemet, brukergrensesnittet beskrives derfor for hver av disse.

2.2.1 Use-case diagram



Use case-diagram for fagansatt-brukergrensesnittet.

2.2.2 Brukerscenario for gruppelærer

Gruppelærer har tilgang til tre forskjellige funksjoner i systemet; innlegging av innleveringer, oversikt over innleveringer og studentopplysninger. De tre forskjellige funksjonene beskrives hver for seg.

2.2.1.1 Innlogging
Gruppelæreren taster inn brukernavn og passord, og blir logget inn og sendt til en ny side dersom brukernavn og passord er korrekt. Dersom brukernavn og passord ikke er korrekt får gruppelærer beskjed om dette og får mulighet til å prøve på nytt.

Skjermbilde for innlogging

2.2.1.2 Legg inn innleveringer
Denne funksjonen gir gruppelærer mulighet for å legge inn innleveringer i systemet. Eksempler på innleveringer som typisk vil legges inn er:
- Innleveringer fra studenter som ikke klarte å benytte student-brukergrensesnittet for å gjøre en innlevering og har sent oppgave-innleveringen på e-post til gruppelærer. Gruppelærer må da selv legge inn innleveringen i databasen for studenten for at den skal kunne sjekkes mot andre innleveringer for likhet.
- En student sitt brukernavn var ikke registrert i databasen da en innlevering ble gjort, og oppgave-innleveringen ble da sendt på mail til gruppelærer. Det er da gruppelærers ansvar å legge inn innleveringen etter at studenten er registrert i databasen.
- Gruppelærer, hovedgruppelærer eller foreleser har utviklet en fasit og ønsker at senere innleveringer skal sjekkes mot denne. Siden en fasit ikke tilhører en student, må det opprettes en fiktiv student med brukernavn som benyttes for fasiter for å kunne overholde bestemmelsene i databasen som sier at en innlevering tilhører en student.
- Foreleser har en samling av gamle innleveringer i kurset som det er ønskelig å legge inn i databasen slik at andre innleveringer kan sjekkes mot likhet mot disse. Disse innleveringene vil heller ikke ha noen eier (student), og det må derfor også opprettes en fiktiv student som gamle innleveringer kan registreres på.

Gruppelærer har her altså to valg; enten kan han velge å legge inn innleveringer til en bestemt student, eller han kan velge å legge inn innleveringer som er fasiter eller som er gamle innleveringer og som ikke tilhører en bestemt student.

Legge inn innlevering til en bestemt student

1. Dette skjermbildet er likt som for det første skjermbildet for studentinnlevering (steg 1 – 11 i brukerscenario for studenter), men gruppelærer må i tillegg registrere innleveringstidspunkt for innleveringen.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Legg inn innlevering

Kurs: Velg kurskode .

Oppgave: Velg oppgavenr

Last opp fil :

Bla gjennom...

Legg til

oblig1.java fjern

Innleveringstidspunkt: (tt mm) (dd mm yy)

Brukernavn:

Gruppenr:

legg inn

Skjermbilde for å legge inn en innlevering, steg 1.

2. Når gruppelærer har valgt ”Legg inn” vises ett nytt skjermbilde som er likt som for det tredje skjermbildet for studentinnlevering (steg 13 i brukerscenario for studenter), men uten ”Skriv ut”-knappen.

Legge inn fasiter og eierløse innleveringer

Dette scenarioet er likt som for å legge inn en innlevering til en bestemt student, men gruppelærer har kun mulighet til å registrere innleveringen(e) på en av de to definerte fiktive brukerne (studentene) og har i tillegg mulighet for å laste opp komprimerte mapper av tidligere innleveringer eller fasiter.

Gamle innleveringer blir lagt inn i databasen uten tilhørighet til kurs og oblig, uten dato og gis et fiktivt brukernavn som benyttes for å markere gamle innleveringer.

1. Gruppelærer velger en av de to fiktive brukerne (fasit eller eierløs innlevering) fra en nedtrekksliste.
2. Gruppelærer velger så de filene som skal legges inn i databasen ved å velge en fil i en fildiaglog og klikke ”Legg til”. Gruppelærer kan legge til en enkel fil av gangen, eller legge til en pakket mappe med filer. Innleveringene som ønskes innlagt vises i en oversikt, med mulighet for å fjerne innleveringen ved å klikke ”Fjern”.
3. Når gruppelærer har lagt til alle de ønskede filene, klikkes ”Legg inn” og en oversikt over de innlagte filene fremvises.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Legg inn innlevering

Brukernavn: Velg brukernavn .

Last opp fil :

Bla gjennom...

Legg til

oblig1.java fjern

Legg inn

Skjermbilde for steg 1-2.

2.2.1.3 Innleveringer

Denne funksjonen gir gruppelærer mulighet for å se innleveringer gjort til en oppgave i et kurs og sjekke de innleverte filene for likhet mot andre innleveringer.

1. Gruppelæreren ønsker å se oversikt over student-innleveringer for en oppgave i ett kurs og velger kurskode, oppgavenummer og gruppenummer fra nedtrekkslister og trykker deretter ”Vis innleveringer”.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Innleveringer

Kurs:

Gruppe:

Oppgave:

vis innleveringer

Skjermbilde for steg 1.

2. Gruppelæreren blir så fremvist en liste over studenter (sortert på brukernavnet til studentene) som er medlem av gruppen og deres siste innleveringer til en oppgave. Gruppelæreren kan også her se innleveringstidspunktet for studentenes innleveringer. Innleveringer som er registrert senere enn den registrerte innleveringsfristen i databasen er

registrert med rødt. Graden av likhet med andre innleveringer i databasen vises også for hver student (likhet er delt inn i tre forskjellige grader av likhet: ”Ulik”(ikke markert), ”Nesten lik”(markert med oransje) og ”Lik”(markert med rød)).

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Innleveringer

inf1000 - gr.1 - oblig1

Ny kontroll

brukernavn	Innleveringstidstidpunkt	likhetsgrad
hannevi	09.07.05 13:25:16	Lik
student1	07.07.05 22:15:07	Ulik
student2	05.07.05 14:32:45	Nesten lik
student3	10.07.05 16:21:02	Ulik

Skjermbilde for steg 2.

3. Gruppelærer har mulighet til å sjekke alle de siste innleveringene til gruppens studenter på nytt ved å klikke på knappen ”Ny kontroll”.
4. Gruppelæreren har så mulighet til å gå inn på en enkelt student ved å klikke på brukernavnet til studenten i oversikten. Gruppelæreren får da oversikt over alle innleveringer som studenten har gjort til en oppgave-innlevering. En student har som tidligere nevent mulighet til å gjøre flere innleveringer til en oppgave, og oversikten inneholder derfor informasjon om alle innleveringene studenten har gjort til den bestemte oppgaven (med den siste innleveringen først). I tillegg vises informasjon om innleveringstidspunktet til hver fil, likhetsgraden til hver fil, antall filer i databasen som er nesten lik eller lik med studentens fil, filen med høyest likhetsgrad og status (en hovedgruppelærer/foreleser kan endre på statusen til en innlevering).

Innleveringer

inf1000 - gr.1 - oblig1

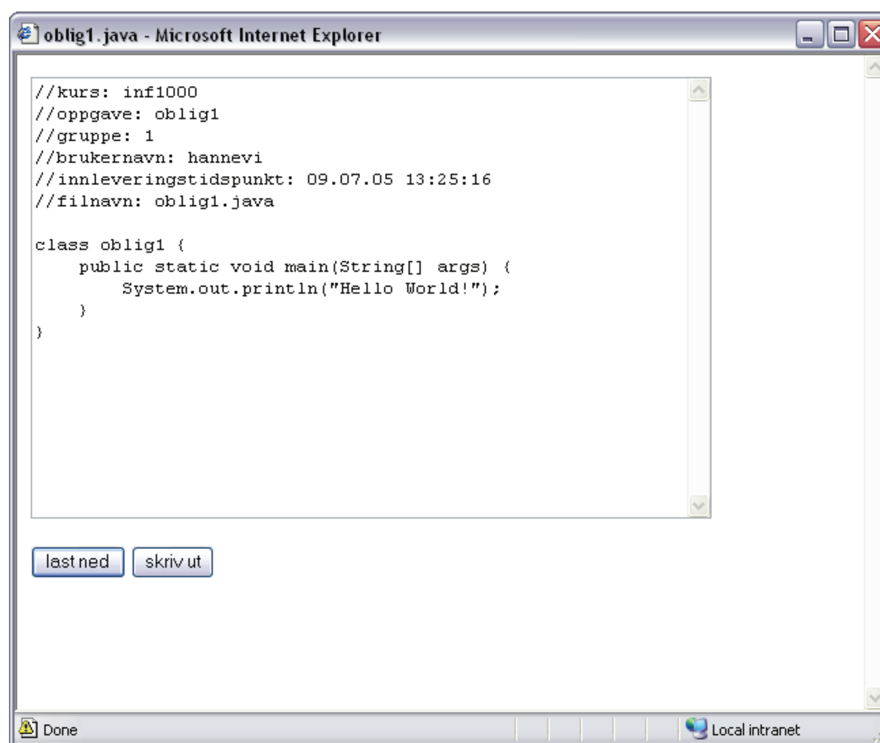
hannevi

Fil	Innleveringstidspunkt	Resultat av kontroll				
		Likhetsgrad	ny kontroll	Like filer		Status
oblig1.java	09.07.05 13:25:16	Lik	☐	3	oblig1.java	
oblig1.java	07.07.05 14:23:02	Nesten lik	☐	1	helloworld.java	
oblig1.java	05.07.05 23:15:33	Ulik	☐			

Skjermbilde steg 4.

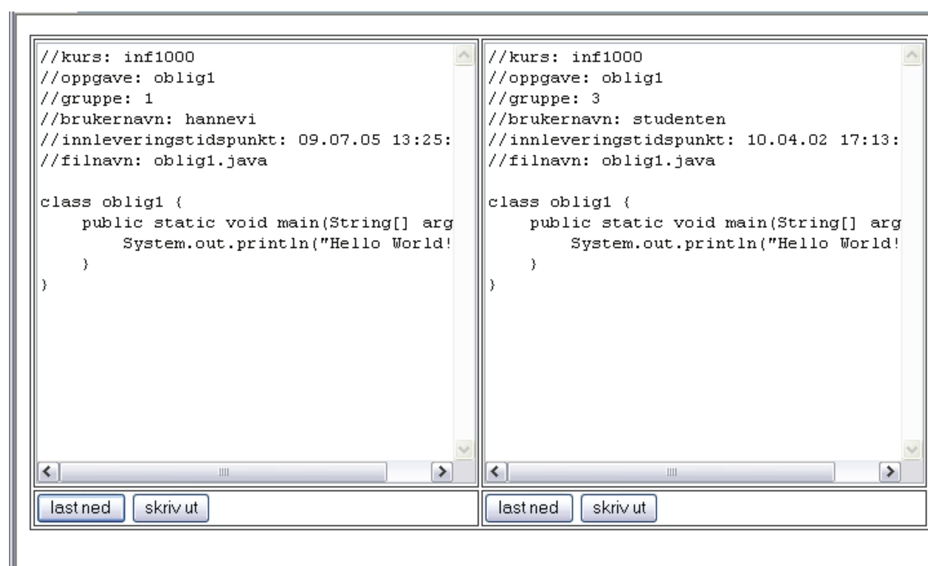
a. Gruppelærer har mulighet til å huke av avkrysningsbokser utenfor hver fil for å kunne gjøre en ny kontroll av de valgte filene.

b. Gruppelærer har mulighet til å gå inn og se på en enkelt fil ved å klikke på filnavnet i oversikten. Filen blir da fremvist i et nytt browser-vindu med mulighet for å laste ned eller skrive ut filen.



Skjermbilde for steg 4b.

c. Gruppelærer har mulighet for å se filen som er mistenkt for likhet med en annen fil, side om side med den like eller nesten like filen. Filene vises i et nytt browser-vindu, og gruppelærer har mulighet til å laste ned eller skrive ut en eller begge filene.



Skjermbilde for steg 4c.

d. Gruppelærer får opplysning om antall filer som kontrollen avdekket var nesten like eller like. Ved å klikke på antallet blir gruppelærer fremvist alle filene som er testet

for likhet med studentens innlevering sammen med likhetsgraden til hver fil. Ved å klikke på en av de like filene åpnes filen i et nytt browser-vindu sammen med studentens innlevering, slikt som i steg 4c.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Innleveringer

inf1000 - gr.1 - oblig1

hannevi | oblig1.java | levert 09.07.05 13:25:16

Resultat av kontroll	
Like filer	Likhetsgrad
oblig1.java	Lik
oblig.java	Lik
helloworld.java	Nesten lik

Skjermbilde for steg 4d.

e. Gruppelærer får i tillegg til antall filer som er nesten like eller like med studentens innlevering opplyst om den filen som har høyest likhetsgrad. Ved å klikke på filen åpnes filen i et nytt browser-vindu sammen med studentens innlevering, slik som i steg 4c.

2.2.1.4 Studentopplysninger

Denne funksjonene gir gruppelærer oversikt over alle innleveringer en student har gjort i et kurs.

1. Gruppelærer velger kursnummer og brukernavn fra nedtrekkslister, og trykker på knappen ”Vis innleveringer”.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Studentopplysninger

Kurs:

Brukernavn:

Vis innleveringer

Skjermbilde steg 1.

2. Gruppelærer blir så fremvis en oversikt over alle innleveringer en student har gjort i det kurset med den siste innleveringen øverst. Dersom det er gjort flere innleveringer til en oblig vises disse også. Gruppelærer får også informasjon om hvilken gruppe innleveringen har blitt gjort til, tidspunktet for innlevering (markert med rødt dersom den ikke har overhold fristen) og graden av likhet.
3. Gruppelærer har så mulighet til å benytte funksjonene som også er tilgjengelig for ”Innleveringer” (avsnitt 2.2.1.3) fra og med punkt 4.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Studentopplysninger

Innleveringer

hannevi

inf1000 - gr.1 - oblig1

Fil	Innleveringstidspunkt	Resultat av kontroll				
		Likhetsgrad	ny kontroll	Like filer		Status
oblig1.java	09.07.05 13:25:16	Lik	☐	3	oblig1.java	
oblig1.java	07.07.05 14:23:02	Nesten lik	☐	1	helloworld.java	
oblig1.java	05.07.05 23:15:33	Ulik	☐			

inf1000 - gr.1 - oblig2

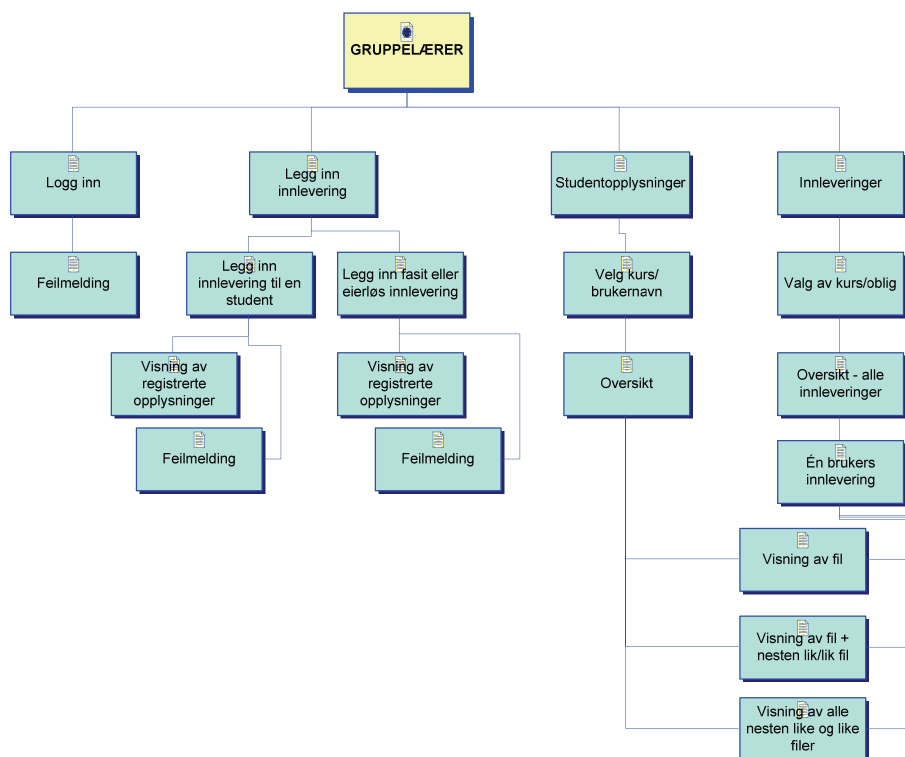
Fil	Innleveringstidspunkt	Resultat av kontroll				
		Likhetsgrad	ny kontroll	Like filer		Status
oblig2.java	15.07.05 20:35:15	Ulik	☐			

Skjermbilde for steg 2.

2.2.1.5 Utlogging

Gruppelærer logger seg ut ved å velge en funksjon som er tilgjengelig i skjermbildet til enhver tid.

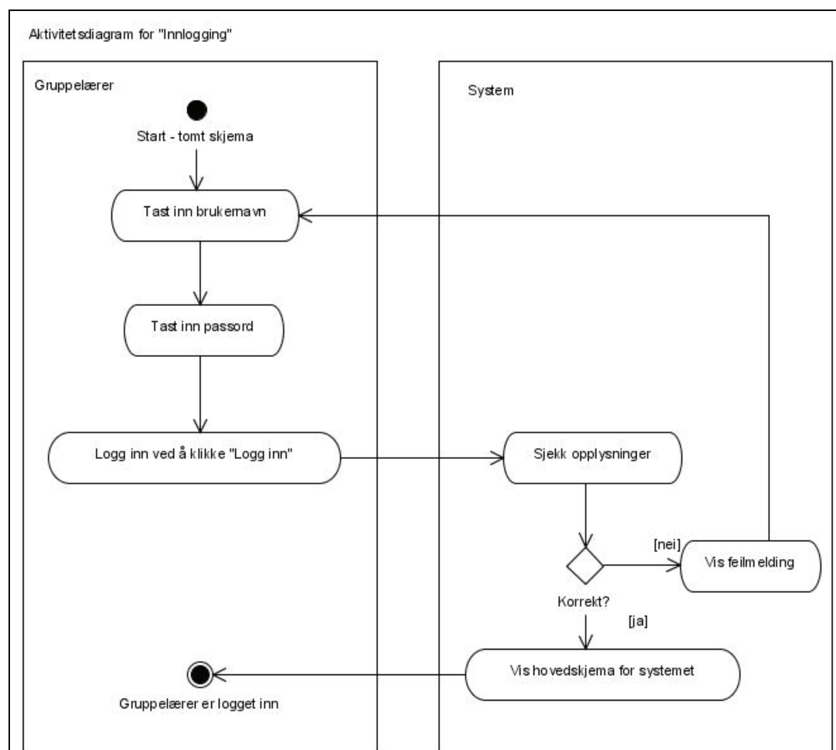
2.2.3 Sitemap for gruppelærer-brukergrensesnittet



Sitemap for gruppelærer-brukergrensesnittet.

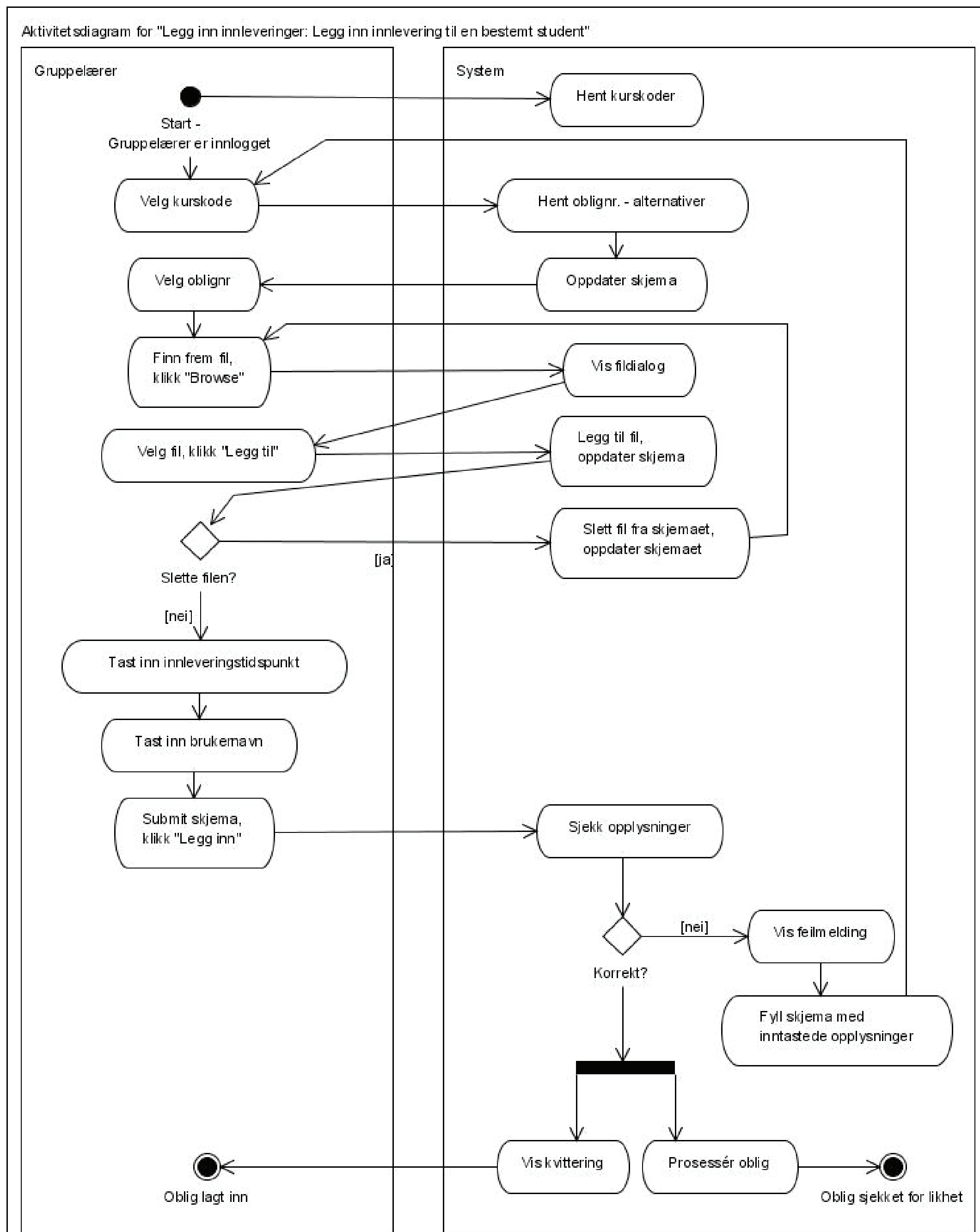
2.2.4 Aktivitetsdiagram for brukergrensenittet til gruppelærer

2.2.4.1 Aktivitetsdiagram for "Innlogging"



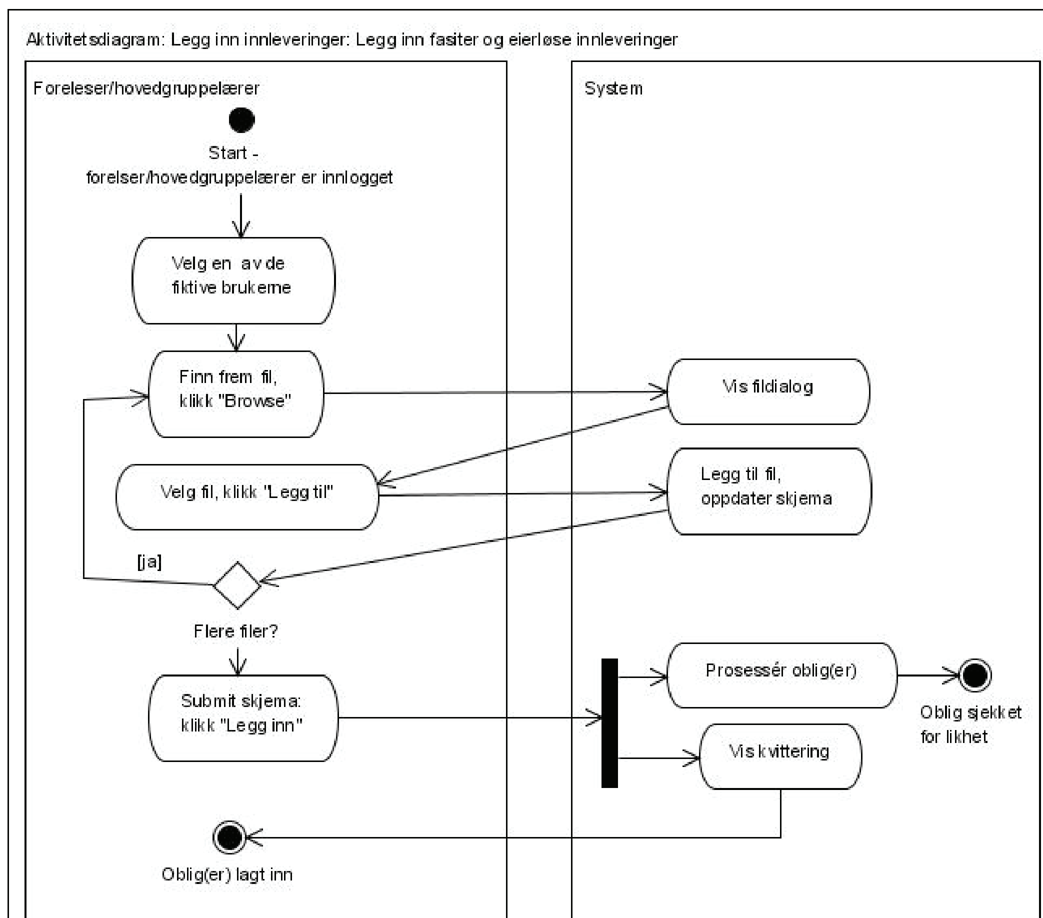
Aktivitetsdiagram for "Innlogging".

2.2.4.2 Aktivitetsdiagram for "Legg inn innleveringer: Legg inn innlevering til en bestemt student"



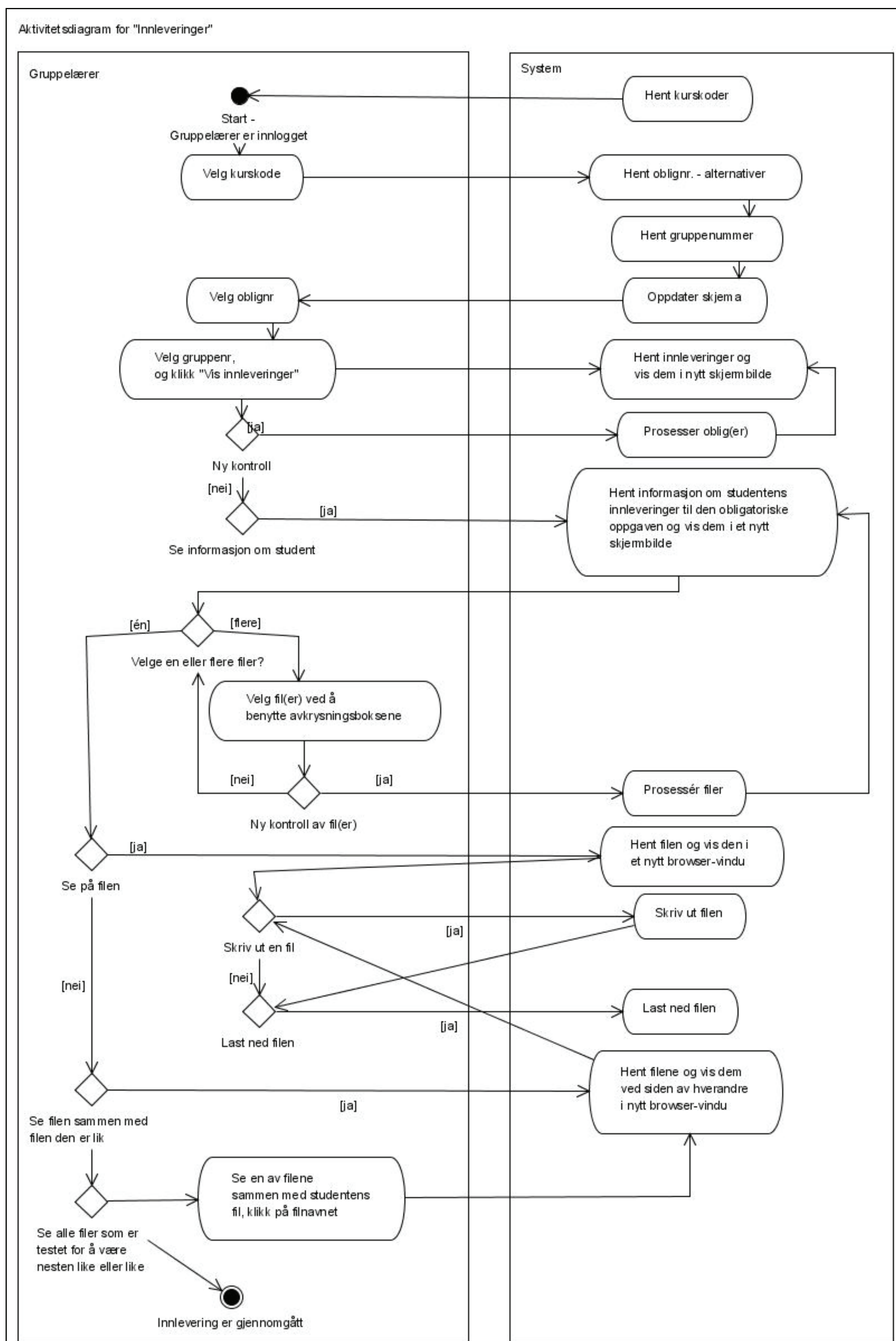
Aktivitetsdiagram for "Legg inn innleveringer: Legg inn innlevering til en bestemt student".

2.2.4.3 Aktivitetsdiagram for "Legg inn innleveringer:Legg inn fasiter og eierløse innleveringer"



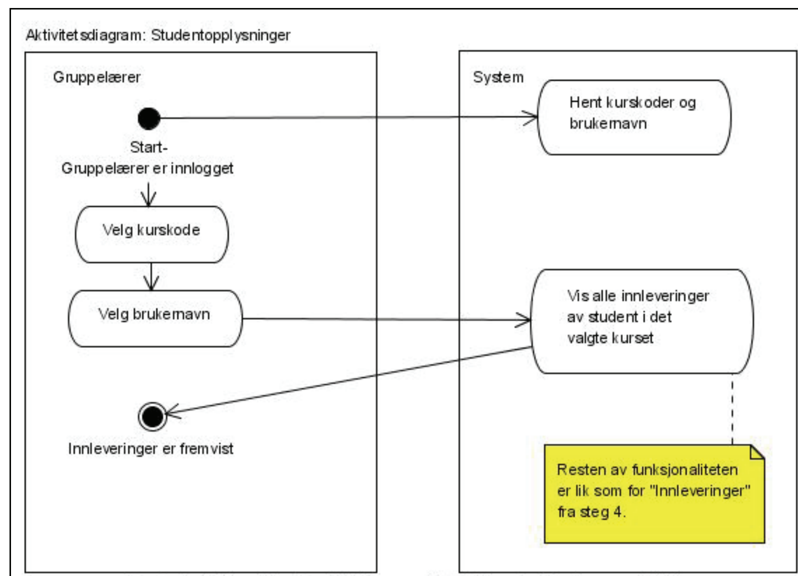
Aktivitetsdiagram for "Legg inn innleveringer: Legg inn fasiter og eierløse innleveringer".

2.2.4.4 Aktivitetsdiagram for "Innleveringer"



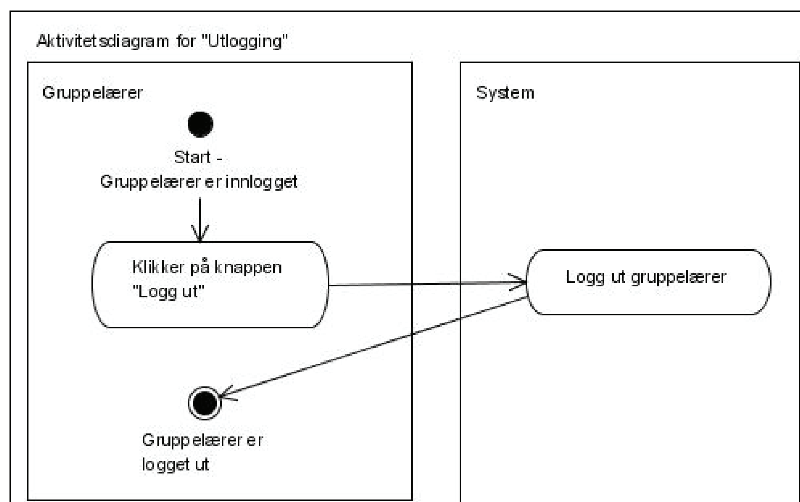
Aktivitetsdiagram for "Innleveringer".

2.2.4.5 Aktivitetsdiagram for studentopplysninger



Aktivitetsdiagram for "Studentopplysninger".

2.2.4.6 Aktivitetsdiagram for "Utlogging"



Aktivitetsdiagram for "Utlogging"

2.2.5 Brukerscenario for foreleser og hovedgruppelærer

Hovedgruppelærer og foreleser har tilgang til fire forskjellige funksjoner, mange av dem like som for gruppelærer bare med mer funksjonalitet. De forskjellige funksjonene beskrives under.

2.2.5.1 Innlogging
Funksjonaliteten er lik som for gruppelærer (se avsnitt 2.2.1.1).

2.2.5.2 Legg inn innleveringer
Funksjonaliteten er lik som for gruppelærer (se avsnitt 2.2.1.2).

2.2.5.3 Innleveringer

1. Foreleser/hovedgruppelærer ønsker å se oversikt over student-innleveringer for en oppgave i ett kurs og velger kurskode og oppgavenummer og gruppenummer og trykker deretter ”Vis innleveringer”. Det eneste som er forskjellig fra gruppelærer-brukergrensesnittet er at foreleser/hovedgruppelærere har mulighet til å velge ”Alle” grupper i nedtrekkslisten ”Gruppenr”.

Legg inn innlevering
Innleveringer
Gruppeopplysninger
Obligopplysninger
Studentopplysninger

Innleveringer

Kurs: INF1000

Gruppe: Alle

Oppgave: oblig1

vis innleveringer

Skjermbilde steg 1.

2. Dersom foreleser/hovedgruppelærer velger å se innleveringer for en spesifikk gruppe, vil brukerscenarioet være likt som for gruppelærer.
3. Foreleser/hovedgruppelærer velger å se innleveringer for alle gruppene, og blir fremvist statistikk for innleveringen samt informasjon om alle studenters innleveringer. Statistikken innebærer oversikt over antall studenter på kurset, antall studenter som har innlevert den spesifikke oppgaven, antall studenter som har innleveringer som er ”Nesten lik” og antall studenter som har innleveringer som er ”Lik”. Listen med informasjon om studenters innlevering er lik som for gruppelærer, bortsett fra at listen er sortert etter gruppenummer og deretter alfabetisk på studentenes brukernavn innenfor hver gruppe. Funksjonaliteten er videre lik som for gruppelærer.

Legg inn innlevering
Innleveringer
Gruppeopplysninger
Obligopplysninger
Studentopplysninger

Innleveringer

inf1000 - oblig1

ny kontroll

Antall studenter på kurset:	13
Antall innleveringer:	12
Antall "nesten lik":	3
Antall "lik":	3

gruppe 1

brukernavn	Innleveringstidstidpunkt	likhetsgrad
hannevi	09.07.05 13:25:16	Lik
student1	07.07.05 22:15:07	Ulik
student2	05.07.05 14:32:45	Nesten lik
student3	10.07.05 16:21:02	Ulik

gruppe 2

brukernavn	Innleveringstidstidpunkt	likhetsgrad
hannevi	09.07.05 13:25:16	Lik
student1	07.07.05 22:15:07	Ulik
student2	05.07.05 14:32:45	Nesten lik
student3	10.07.05 16:21:02	Ulik

gruppe 3

brukernavn	Innleveringstidstidpunkt	likhetsgrad
hannevi	09.07.05 13:25:16	Lik

Skjermbilde for steg 3.

4. Foreleser/hovedgruppelærer har også, som gruppelærer, mulighet til å gå inn på en enkelt student ved å klikke på brukernavnet til studenten i oversikten. Funksjonaliteten her er også tilsvarende som den for gruppelærer, men foreleser/hovedgruppelærer har i tillegg mulighet til å endre likhetsgraden på filen til en innlevering som tilhører en student.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Obligopplysninger
Gruppeopplysninger
Studentopplysninger

Innleveringer

hannevi

inf1000 - gr.1 - oblig1

Fil	Innleveringstidspunkt	Resultat av kontroll					Endre likhetsgrad
		Likhetsgrad	ny kontroll	Like filer		Status	
oblig1.java	09.07.05 13:25:16	Lik	☐	3	oblig1.java		Velg likhetsgrad
oblig1.java	07.07.05 14:23:02	Nesten lik	☐	1	helloworld.java		Velg likhetsgrad
oblig1.java	05.07.05 23:15:33	Ulik	☐			Endret	Velg likhetsgrad

Oppdater

Skjermbilde for steg 4.

5. Resten av funksjonaliteten er også lik som for gruppelærer.

2.2.5.4 Gruppeopplysninger

Siden databasen skal kreve så lite administrasjon som mulig er det studentene som skal registrere et kurs sine grupper gjennom sine innleveringer. Når en student registrerer et gruppenummer som ikke allerede er registrert for det valgte kurset, opprettes dette gruppenummeret som navnet på en gruppe tilhørende det valgte kurset. Det eksisterer altså ingen sjekk for om det studenten taster inn er korrekt, og det kan derfor etterhvert være registrert grupper i databasen som ikke egentlig eksisterer på kurset. Gruppenummeret vil opptre i alle nedtrekkslister for gruppenummer i brukergrensesnittet til en fagansatt, og dersom studenter legger inn mange ”ikke-eksisterende” grupper, kan listen bli lang. Det kan derfor være ønskelig med en funksjon som gir forelesere og hovedgruppelærere mulighet til å fjerne de ”ikke-eksisterende” gruppene fra databasen, og flytte studenters innleveringer til disse til andre grupper.

1. Foreleser/hovedgruppelærer velger kursnummer og får fremvist en liste med alle gruppene som er registrert under det valgte kurset samt antall innleveringer gjort til en gruppe.

Online innlevering av obligatoriske oppgaver ved Institutt for Informatikk

Legg inn innlevering
Innleveringer
Gruppeopplysninger
Obligopplysninger
Studentopplysninger

Gruppeopplysninger

Kurs: Velg kurs

vis grupper

Skjermbilde steg 1 – valg av kurskode.

Legg inn innlevering
Innleveringer
Gruppeopplysninger
Obligopplysninger
Studentopplysninger

Gruppeopplysninger

Kurs: Velg kurs

vis grupper

Grupper registrert i kurs: inf1000

Gruppenr	Antall innleveringer
1	3
2	3
3	3
9	1
111	1

Skjermbilde steg 1 – liste med alle grupper registrert under det valgte kurset.

2. Dersom foreleser/hovedgruppelærer ønsker å fjerne en gruppe fra kurset siden gruppen ikke eksisterer i virkeligheten, klikker foreleser/hovedgruppelærer på navnet til gruppen og blir fremvis en oversikt over innleveringer som har blitt gjort til den valgte gruppen med oblignavn og studentens brukernavn.

Legg inn innlevering
Innleveringer
Gruppeopplysninger
Obligopplysninger
Studentopplysninger

Gruppeopplysninger

Kurs: inf1000 - Gruppenr 111

slett gruppe

Innleveringer gjort til denne gruppen:

brukernavn	Oblignr	Flytt innlevering til annen gruppe
student7	1	3

flytt innleveringer

Skjermbilde steg 2.

3. Før foreleser/hovedgruppelærer kan slette en gruppe, må innleveringer som er gjort til den flyttes til andre grupper. I oversikten over innleveringer er det mulighet til å velge en ny gruppe for hver og en innlevering, ved å gjøre valg i en nedtrekksliste ved siden av hver

innlevering. Ved å klikke på ”Flytt innleveringer” flyttes alle innleveringene til de valgte gruppene.
4. Når alle innleveringer til gruppen er flyttet får foreleser/hovedgruppelærer mulighet til å slette gruppen ved å klikke på knappen ”Slett gruppe”.
5. Når gruppen er slettes vises foreleser/hovedgruppelærer skjermbildet fra steg 1 igjen, med oppdaterte opplysninger (dvs at den slettede gruppen er borte).

2.2.5.5 Obligopplysninger
Foreleser og hovedgruppelærer har mulighet for å legge inn og endre opplysninger om en oblig for et kurs (kurset må allerede være lagt inn i databasen, dette er administratorens ansvar).

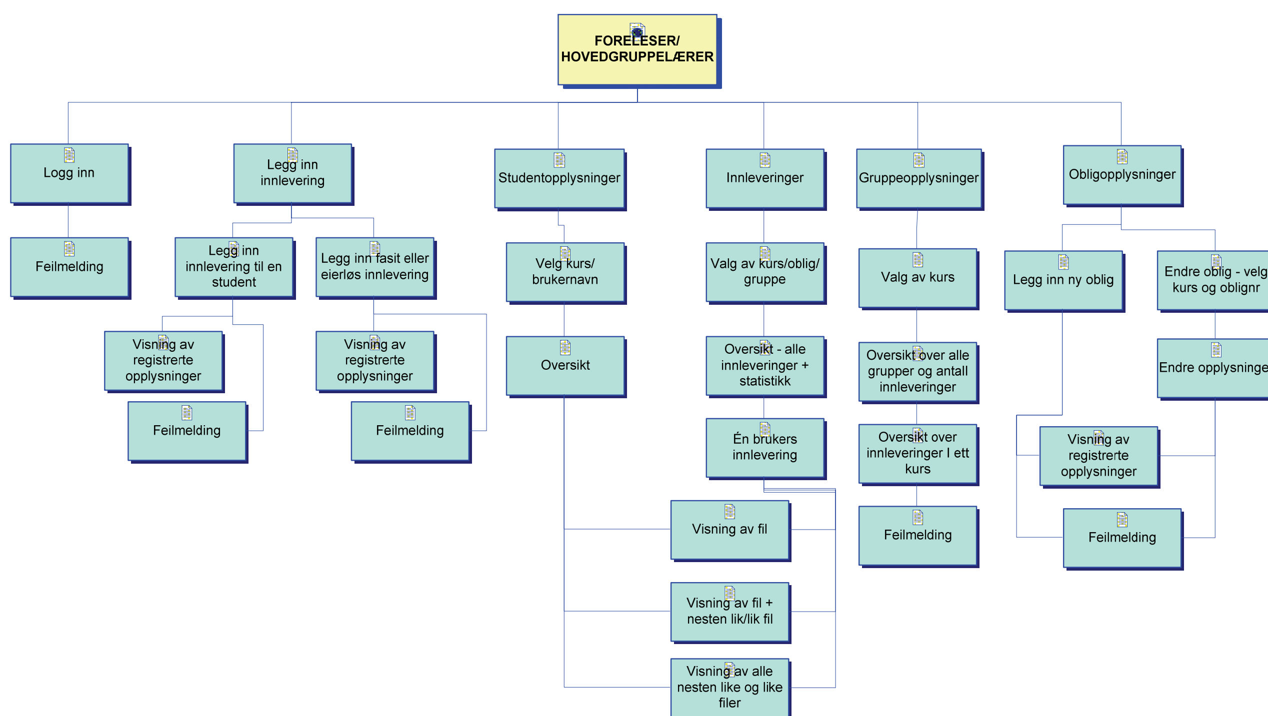
Innlegging av ny oblig
1. Foreleser/hovedgruppelærer velger kurset fra en nedtrekksliste.
2. Foreleser/hovedgruppelærer skriver inn oppgavenummer, innleveringsfristen, to grenseverdier (verdier som algoritmen for sjekk av likhet benytter for å fastslå graden av likhet mellom to innleveringer) og laster eventuelt opp oppgaveteksten i en fildialog dersom dette er ønskelig, og trykker til slutt på knappen ”Registrer oppgave”.
3. Foreleser blir fremvist en oversikt over de registrerte opplysningene.

Endring av opplysninger for en oblig
1. Foreleser velger kurs og oppgavenummer fra nedtrekkslister.
2. Foreleser får mulighet til å endre opplysninger om oppgavenummer, innleveringsfrist og de to grenseverdiene, og får i tillegg mulighet for å eventuelt slette oppgavetekst som er lagt inn eller laste opp en ny fil. Når foreleser er ferdig med å endre opplysningene, trykker han på knappen ”Registrer nye opplysninger”.
3. Foreleser bli fremvist en oversikt over de registrerte opplysningene.

Dersom foreleser/hovedgruppelærer ønsker å beregne likheten for alle innleveringer gjort til en oblig etter at grenseverdiene er endret, benyttes funksjonaliteten i avsnitt 2.2.6.3.

2.2.5.6 Obligopplysninger
Funksjonaliteten er lik som for gruppelærer (se avsnitt 2.2.1.5).

2.2.6 Sitemap for foreleser/hovedgruppelærer-brukergrensesnittet

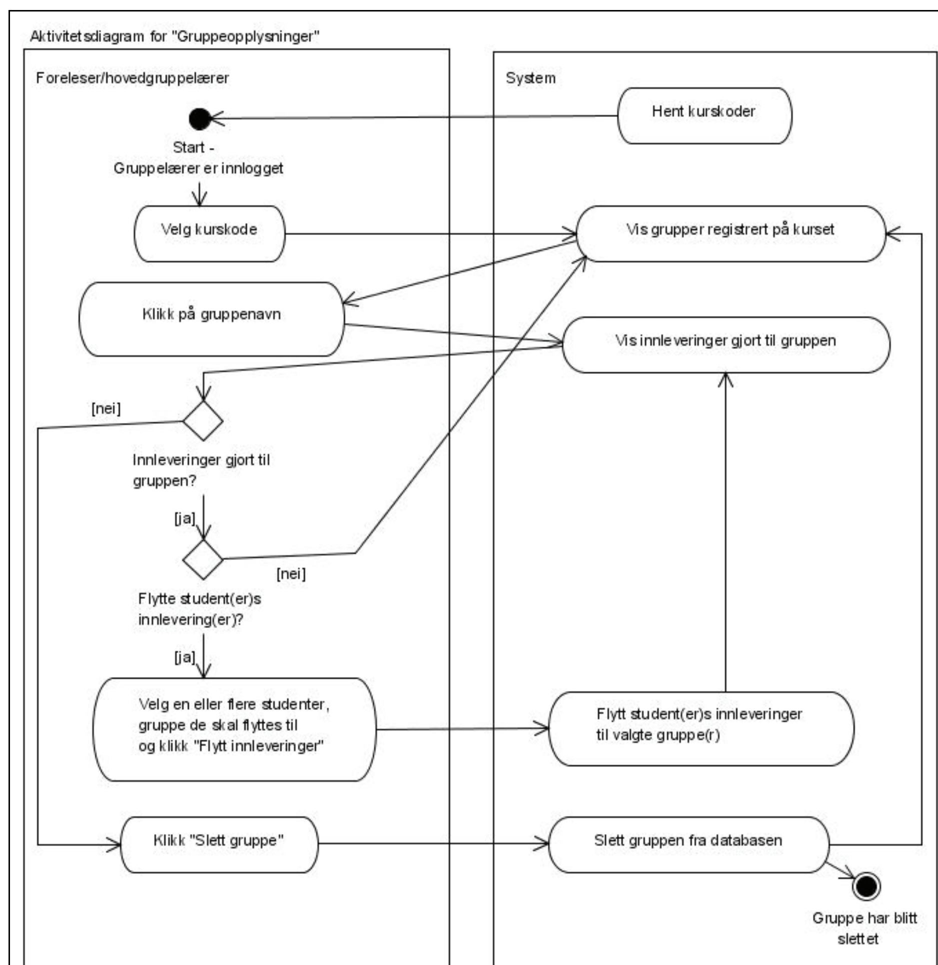


Sitemap for foreleser og hovedgruppelærer-brukergrensesnittet.

2.2.7 Aktivitetsdiagram for brukergrensesnittet til foreleser og hovedgruppelærer

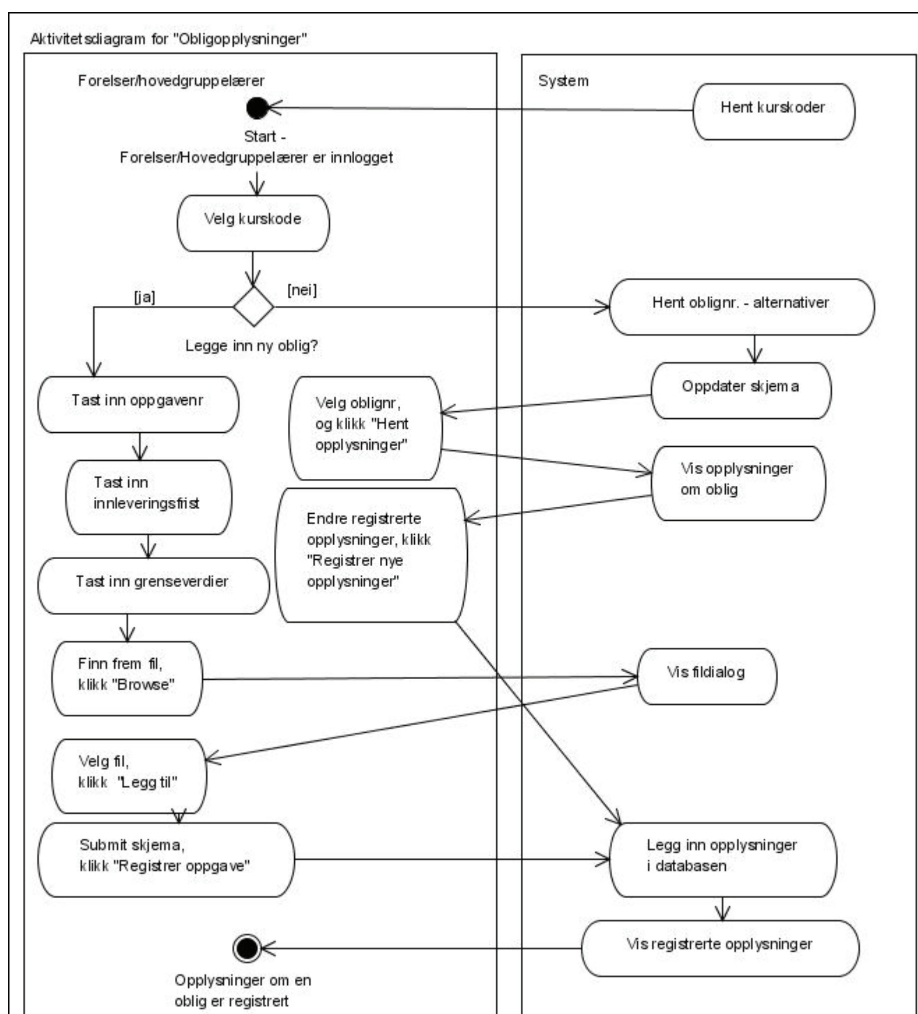
Siden det er mye av funksjonaliteten til foreleser og hovedgruppelærer som er helt lik som for gruppelærer, vil vi kun vise aktivitetsdiagrammer som beskriver funksjonalitet som har tilleggsfunksjoner for foreleser og hovedgruppelærer som er veldig forskjellig.

2.2.7.1 Aktivitetsdiagram for "Gruppeopplysninger"



Aktivitetsdiagram for "Gruppeopplysninger".

2.2.7.2 Aktivitetsdiagram for "Obligopplysninger"



Aktivitetsdiagram for "Obligopplysninger".

2.2.8 Mulige utvidelser av funksjonalitet

En foreleser eller hovedlærer kan være interessert i å se statistikk over antall innleveringer som er mistenkt for fusk i et bestemt kurs, over en valgt tidsperiode, dette kan være mulig å legge til systemet i fremtidige versjoner.

For å gjøre sammenligning av to filer som er mistenkt for likhet enklere, kan elementene som det er sjekket for likhet etter markeres med farge der de opptrer i filen.

En annen mulig utvidelse er å gi hjelpelærer mulighet til å se hvilke student-innleveringer som han har godkjent i oversikts-skjermbildet for "Innleveringer".

I en senere versjon av systemet bør en fagansatt kunne endre passordet han har blitt tildelt for å kunne logge seg på systemet.

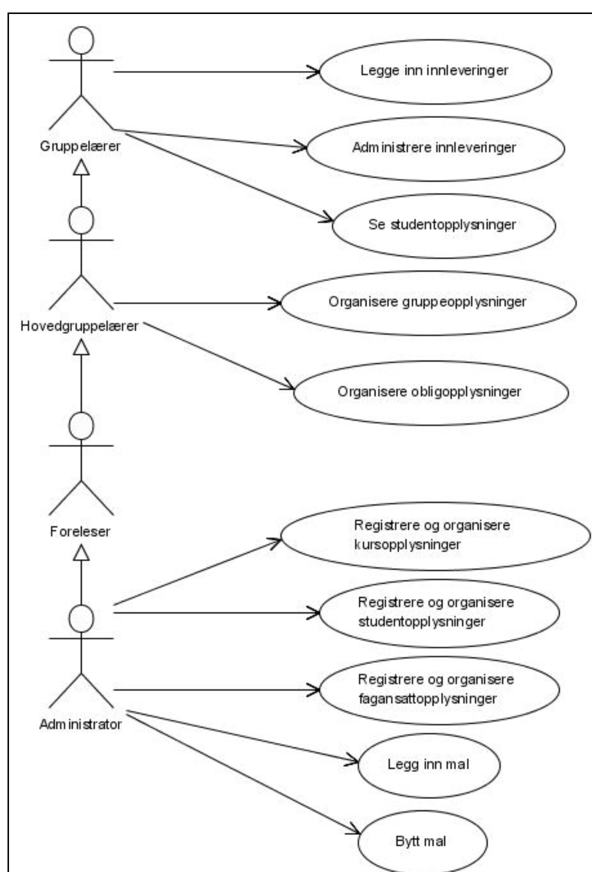
2.3 Administrator-brukergrensesnittet

Grensesnittet for administratoren gir mulighet for innlegging av informasjon om kurs, grupper, fagansatte, studenter og maler som benyttes for å undersøke innleveringer for fusk. Administratoren har i tillegg all funksjonalitet som foreleser har tilgjengelig.

Administratoren er også ansvarlig for å legge inn passord som de fagansatte benytter for å logge seg på systemet (denne må ligge kryptert i databasen), og har mulighet til å velge hvilken mal (dvs hvilke elementer) innleveringene skal sjekkes opp i mot og kunne generere nye lengder for innleveringer lagret i databasen.

Informasjon om grupper, kurs, studenter og fagansatte legges i første versjon inn i databasen ved hjelp av informasjon i flate filer.

2.3.1 Use case-diagram



Use case-diagram for administrator-brukergrensesnittet.

2.3.2 Brukerscenario for administrator

Administrator må kunne selv sette fristene for når et semester starter og slutter.

2.3.2.1 Innlogging
Funksjonaliteten er lik som for gruppelærer (se avsnitt 2.2.1.1).

2.3.2.2 Legg inn innleveringer
Funksjonaliteten er lik som for gruppelærer (se avsnitt 2.2.1.2).

2.3.2.3 Innleveringer
Funksjonaliteten er lik som for foreleser/hovedgruppelærer (se avsnitt 2.2.6.3), men administrator har i tillegg mulighet for å slette en innlevering til en student.

2.3.2.4 Studentopplysninger
Funksjonaliteten for å se en innlevering til en bestemt er lik som for foreleser/hovedgruppelærer (se avsnitt 2.2.1.4), men administrator har i tillegg mulighet til å legge inn, endre og slette studenter fra databasen.

2.3.2.5 Gruppeopplysninger
Funksjonaliteten er lik som for foreleser/hovedgruppelærer (se avnistt 2.2.6.4).

2.3.2.6 Obligopplysninger
Funksjonaliteten er lik som for gruppelærer (se avnistt 2.2.6.5), men administrator har i tillegg mulighet for å slette en oblig fra databasen.

2.3.2.7 Kursopplysninger
Denne funksjonen er kun tilgjengelig for administratoren og innehar funksjonalitet for å legge inn, endre og slette kurs fra databasen.

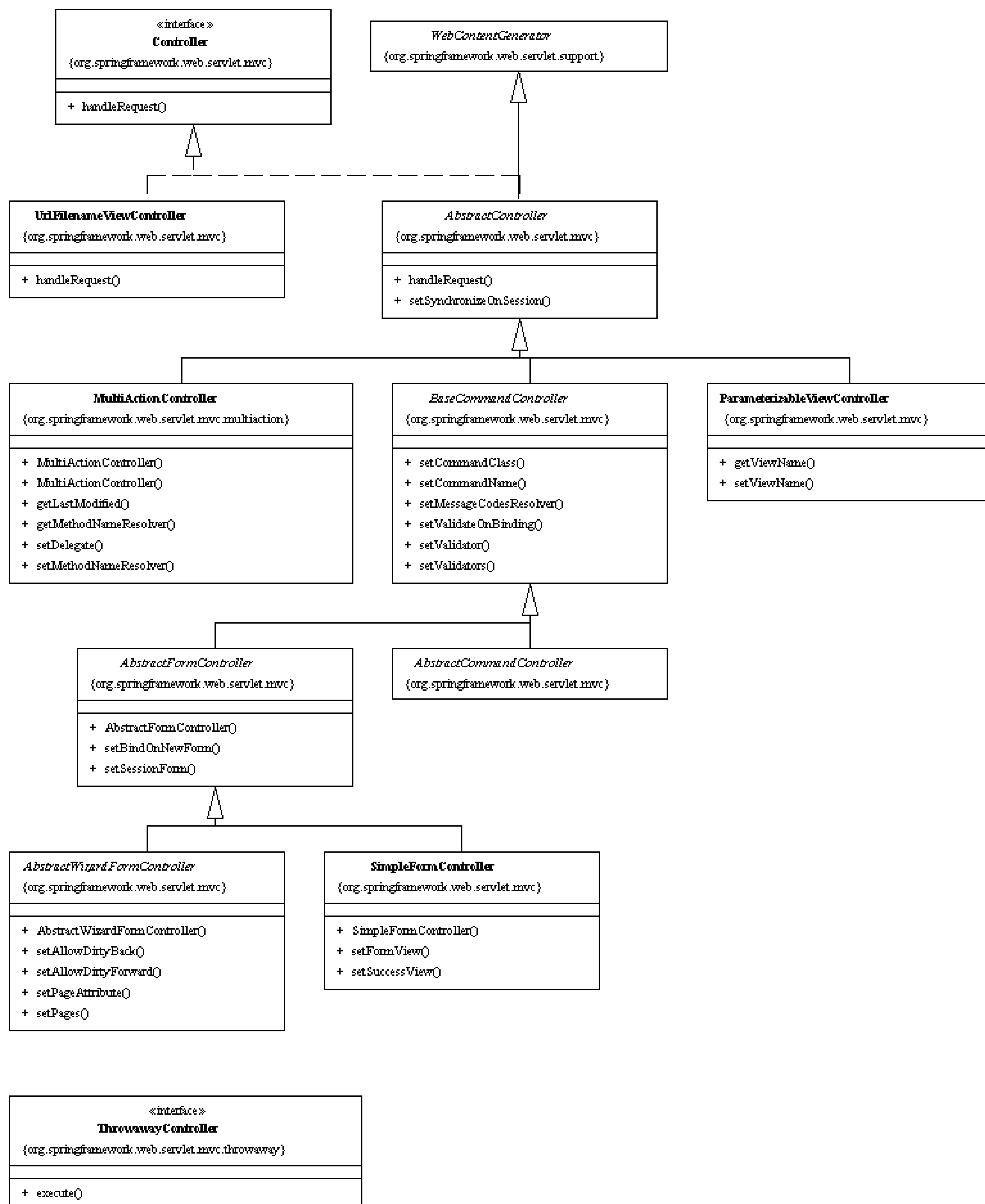
2.3.2.8 Fagansattopplysninger
Denne funksjonen er kun tilgjengelig for administratoren og innehar funksjonalitet for å legge inn, endre og slette fagansatte fra databasen. Opplysninger om en fagansatt innebærer brukernavn, passord, privilegienivå (gruppelærer, foreleser eller hovedgruppelærer) samt informasjon om de skal motta e-post for hver innlevering som er mistenkt for likhet med andre.

2.3.2.9 Mal
Denne funksjonen er kun tilgjengelig for administratoren og innehar funksjonalitet for å definere en mal som skal benyttes for å sjekke innleveringer for likhet, endre den gjeldende malen som benyttes og kjøre ny kontroll av samtlige innleveringer i databasen.

2.3.2.10 Utlogging
Funksjonaliteten er lik som for gruppelærer (se avsnitt 2.2.1.5).

Appendix C

Spring MVC Controller hierarchy



Appendix **D**

Known errors and shortcomings in Joly

Security

The Joly application currently has no advanced authentication mechanism. Even though it exist a login function which works fine, there is no encryption of the password. We recommend using Acegi Security, which is a security framework for Spring and is meant to be integrated with the application after development. Acegi offers user role based authentication which is suitable for the Joly application.

Session timeout and the login page

There is one problem when redirecting to the login page in the case of a session time out. When an employee is logged out because of a session timeout, he will be prompted with the login page if he tries to perform an action on a restricted page. If this restricted page happens to be the result of the “View program code” function, which opens a new browser window, the login page will appear in this new window and all subsequent pages are shown here. The problem is that this window is only meant for displaying the program code and has no visible browser elements. To resolve this problem the login page must be redirected and displayed in the browser window where the user triggered the action in stead of displaying it in the result window.

Internationalization

Joly uses the Spring application context to implement a message source; an interface to obtain localized messages. This enables Joly to be internationalized by offering language-specific views to the users. Currently the standard and only language available is Norwegian, so to extend the application with additional language support, separate properties files for each

language must be written. In addition, the joly-servlet.xml configuration file must be extended with a configuration of the Spring class *AcceptHeaderLocaleResolver*, to be able to recognize the client-side browser locale configuration.