UNIVERSITY OF OSLO
Department of Informatics

The Recovery of an Old Lost System

Master Thesis

Siamek Darisiro

May 2, 2012

Acknowledgements

I would like to thank my supervisor, Arne Maus, for his insight during discussion and for the valuable feedback he has given throughout the time it has taken to finish this thesis.

My family, for their support and encouragement.

I would also like thank my fellow students and friends for their opinions whenever I was in doubt, and for keeping up a great atmosphere in the computer lab on the 9th floor.

Abstract

Software maintenance is the act of keeping software operational and updated after its initial release. The software industry spends hundreds of millions of dollars yearly on software maintenance and it is clear that there is no blueprint for how it should be performed. In this thesis we will look at performing software maintenance on a system called Joly. The first version of Joly was released in 2006, and it has been used at the Department of Informatics at the University of Oslo since then. Joly is written in Java and makes use of several pieces of Open-Source software. The main theme of this thesis is about how partially existing source code can be reconstructed by reverse engineering it through the use of a decompiler. And while there are many Java decompilers out there, few of them actually work. We look at how Open-Source software, which has become increasingly popular over the years, can affect software maintenance and what to keep in mind when deciding to use it. Joly has had different developers working on it to improve it over the years and we will see what impact this has made, and finally we look at how we can correct various errors in the system, with one particular error that is causing it to crash.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this master thesis we look at the software maintenance of a system that has been abandoned by its developers. A system that has not recieved regular maintenance since its creation and which is now struggleling. The system we will be looking at is called Joly. It was designed in 2006/2007 by former Master students as a part of their Master's thesis and it is used at the Department of Informatics. Joly is written in Java and makes use of Open-Source Software, it is a system designed for handling mandatory assignments in the introductory Java course at the University of Oslo. Joly has since it was first created been in the hands of several developers who have each had their chance to both add to and maintain the system. We will try to get an overview over what sort of state the system is in, and depending on this, save it from a premature end or perfect it.

1.1 Motivation

The students who created Joly are no longer at the Department, they have graduated and moved on, and as such they can no longer maintain the System. This is of course a problem if one cannot find suitable candidates that can carry on doing the work that is needed in order for it to remain operational. Since this is the Department of Informatics there has never been a shortage of students willing to do this, but as with the creators of the system they have also moved on. This has been a pattern for all the students who have worked on Joly. They have come in, made an effort to improve and then moved on. As of the start of this thesis Joly is experiencing some problems. It is no longer operational and even when it was, it would crash for reasons no one can remember, or fully understood.

The motivation behind our work is therefore to uncover how, having many developers over a short amount of time has affected Joly and how their actions have made an impact. We will look

at whether or not outdated open source frameworks and tools have caused any problems and the influence Open-Source Software has had on the system. We also want to find out how Joly will handle a change of environment and how Java can be beneficial in such circumstances. We will look at correcting bugs that may be present in the source code and if it is possible to optimize the system. Another thing that will hopefully be somewhat clearer at the end of this thesis is why so much effort and money is spent on doing such maintenance work.

1.2 Note to the Reader

This thesis might at times seem a little fast paced and not as focused as it could have been. This is partly because the events were recorded as they occured, but also because it was at first thought that Joly was in a much better state than it actually was.

1.3 Summary of Chapters

Chapter 2: In this chapter we take a look at what software maintenance is, why it is necessary, and how it can be categorized. We then look at maintenance costs, a software maintenance process and what neglecting software maintenance can lead to.

Chapter 3: This chapter is about Joly, the software system we will perform our maintenance effort on. We look at how it is built, which open source frameworks it makes use of and discuss the different problem areas of the system.

Chapter 4: In this chapter we begin our maintenance effort by getting to grips with the state of the system. We also try to determine resource usage when Joly is in use. We are prevented from doing any further testing when we find out that the source code for the latest version has been lost by a former developer. This leads us to search for a way to retrieve the lost source code .

Chapter 5: In this chapter we look at our options for solving the problem of not having source code, whether to create a new version from scratch or decompile the class files we have available. We set up a plan for our continued effort and proceed by executing upon that plan. Towards the end we discover a more recent version of the source code.

Chapter 6: Before we proceed with wrapping up the first part of our maintenance effort we look at how our discovery in the previous chapter affects our plan. We also look at how successful the work we have done so far has been.

Chapter 7: This chapter details our options when it comes to finding the bugs that Joly has. We then choose the approach we believe will suit our situation the best.

Chapter 8: In this chapter we look at how we can perform our debugging effort so we can find the bugs that are causing Joly to crash. We also look at some other bugs that we proceed with correcting.

Chapter 9: In the last chapter we summarize the experience we have gathered through our maintenance work with Joly. We discuss the way we handled the challenges that we faced and take a look at what future work remains.

Chapter 2

Software Maintenance

Before we begin the software maintenance process we must first look at what it means to maintain software and the different ways of doing it. It may not always be viable to maintain old software because of the cost versus benefit ratio and we will therefore look at what can be done to reduce maintenance costs.

2.1 Maintenance

Maintenance is defined as "the process of preserving a condition or situation or the state of being preserved.."[36]. For material objects like cars, which are prone to corrosion, or clothing , which is affected by wear and tear, this would mean to keep them in such a shape that they can still serve their original purpose. Software is however not tangible, and as such physical factors have little to no impact on it and the concept of doing maintenance work on software may therefore seem odd. On the other hand there are other challenges that software faces, like bugs, performance issues, new functionality that needs to be implemented and environmental changes.

An automobile maker like Mercedes cannot release a car with faulty brakes but there may be other errors such as the car's chassis being exceedingly prone to corrosion. The car will work great and fulfill its purpose, but while the chassis is expected to last for 10 years before corrosion occurs, it may only take 3 years. This problem will be fixed for their next models but through the affected models lifetime they will have lost millions of dollars because of their obligation wrt. their corrosion warranty.

It is the same case with software, the first version of a software program often has bugs that affect the user experience. If these bugs are not fixed early on it will cause negative publicity for the company releasing it and revenues will go down. It may not even be the first version of the software. When Microsoft released Windows Millenium they experienced a huge amount of negative

feedback due to all the bugs it contained, forcing them to release their next iteration(Windows XP) a lot earlier then what was normal for Microsoft.

2.1.1 Definition

Software maintenance is defined by the the Institute of Electrical and Electronics Engineers(IEEE) as follows[46]: "Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.". By this definition software maintenance is something that is done after it is delivered to the customer to either improve it or make it run as it was originally intended to. This definition is disputed because many believe that software maintenance is something that should start at an early stage[3]. The reason for this is to identify problem areas or parts of the software that may need to be changed in the future, and so that design decisions and other issues that may affect the difficulty of making changes at later stages become easier. It is clear that software maintenance plays a huge role in the software industry because many products are rushed out to the market before sufficient testing can be done, while others need updates throughout their lifetime to remain operational.

2.2 Categories

It is possible to divide software maintenance into different categories or types based on what type of maintenance is done. Lientz and Swanson decided to categorize them into three types, corrective, adaptive and perfective[28]. IEEE implements the same categories but adds one more and so combined we have these four categories[45]:

***Corrective maintenance*:** targeting the removal of bugs and faults.

***Adaptive maintenance*:** performed to make software run in a different environment or on a different platform.

***Perfective maintenance*:** when implementing new functionality.

***Emergency maintenance*:** unscheduled corrective maintenance performed to keep a system operational

There are other ways to categorize them, but in effect they can be considered variations of the IEEE or Lientz and Swanson versions. The International Organization for Standardization(ISO) for example divides them into three categories. *Problem resolution* which corresponds to corrective maintenance, *Interface modifications* which corresponds to adaptive maintanance and *functional expansion or performance improvement* which is like perfective maintenance but with emphasis

also on improving performance. Corrective type maintenance will likely have the highest priority in the beginning of a products life cycle as it affects user experience the most, but over time perfective type will consume most resources. Developers will often seek to prolong software products lifetime by implementing new functionality that maybe competing products already have. Emergency type maintenance is also becoming increasingly important as software systems are targeted by DDOS attacks and hacking in recent times[50].

2.3 Maintenance Cost

With an increasing reliance on software in business environments, home appliances, cars, entertainment, social networking and so on, the number of software systems is on an ever increasing curve. Ever since the first systems were built there has been a need for maintenance in one form or the other, and if we consider the type of software systems that are popular today, many of them provide services that millions of people are dependent on. Faceboook, various Google applications such as their search engine and Microsoft's Windows operating system are all examples of software systems that need constant maintenance to provide a high level of quality, new functionality for their users and to stay competitive. These are systems that have a long lifespan and as such a majority of their cost goes into maintenance. Surveys done up through the years show that between 60-90% of the total cost of software systems is spent on maintenance[25]. In the United States alone it was estimated in 1995 that around 70 billion dollars were being spent on software maintenance. The amount of resources that is put into software maintenance today is huge and as Müller at al. showed in 1994, the number of code lines maintained doubles every seven years. A survey done in Norway in 2003 showed that the overall time software companies used for maintenance tasks to be around 40% of their total[26].With an increase in the code base that needs to be maintained there is an increasing focus on how to do it efficiently, software development processes have improved over the years and there are even processes specifically targeted at software maintenance.

2.4 Famous Cases

Therac-25 was a radiation therapy machine that was used to treat patients who had cancer, but because of a software malfunction it ended up killing 3 people due to extremely high amounts of radiation. The machine used some of the same software that its predecessor Therac-6 used, but because the producer wanted to cut costs they did not include the hardware level safety mechanisms that had previously been used. The machine was controlled by software with input from the users and because of bugs in the software it resulted in patients getting a higher dose or several doses because it would display to the user that no dose had been given. If the hardware level safty mech-

anisms had been present this would have been prevented but instead three lives were lost[27].

Ariane 5 was a European rocket designed to deliver payloads into earth orbit and was the successor of Ariane 4. Most of the software was reused from Ariane 4, but because of the physical difference between the two rockets, which was not taken into account, the rocket steered off course because the wrong signals were sent to the engines. This resulted in a wrong exit trajectory from the earths atmosphere and the rocket burned up. The software fault occurred when a 64-bit floating point conversion to a signed 16-bit integer caused an overflow which did not affect Ariane 4 because the variable could never reach a high enough value for that to happen[6].

During the cold war a Soviet satellite signaled that the United States had fired four ballistic missiles at Russia. The software in the satellite had falsely signaled the suns reflection in the clouds as missiles and had the operator of the warning system believed in it then Soviet would have launched a counter attack with their own missiles. However the alarm was luckily disregarded and a possible new world war was avoided[30].

2.5 Processes

Every organization that develops software has some sort of software development process. A software development process describes when the organization should be doing what, how and who should be doing it. The classic variant is the waterfall model where there are different phases like requirements specification, design, implementation, review etc. Each of the phases are performed in succession until the project is finished. The problem with this model is that for the product to be completed with as few resources as possible, each phase needs be completed as close to perfectly as possible since there is usually only one iteration through all the phases. A mistake in one phase may propagate through to the subsequent phases, costing more resources. In an effort to make software development more efficient there has been a change from rigid development processes like the waterfall model to agile processes like Scrum. The idea is to have shorter iterations but more of them, starting testing and improvement phases at an earlier stage and thereby discovering potential problems at earlier stages. Agile methods are therefore much better suited for maintenance work as they provide feedback at a much earlier time compared to non-agile processes.

Agile methods may still be too general, and although they may be adapted to work as maintenance processes there are also processes that are specifically targeted at maintenance, like the ISO/IEC defined software maintenance process in figure 2.1, with table 2.1 showing the tasks in each activity[44]. Such processes may be more efficient when it comes to maintaining software but they will require personnel training and effort to implement and may not be suited to every

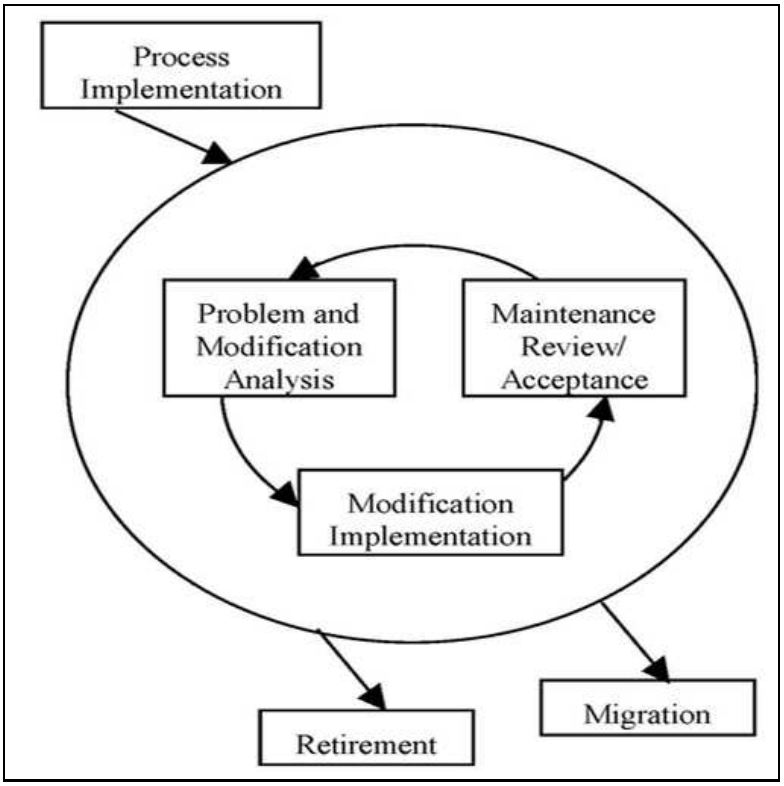

Figure 2.1: The ISO/IEC 14764-00 Software Maintenance Process

organization.

2.6 Summary

Software maintenance will be an issue in the forseeable future. Unless organizations actively prevent it, they will over time loose the required in-house competence to successfully maintain their software systems. Many old software systems may have been written in code languages that may be tens of years old and that are no longer learned by new developers, they may have designs that make them hard to maintain or just poor documentation and messy code that requires too much effort for developers to immerse themselves inn. The alternative to maintaining old software is to start from scratch, building new software systems. This will not remove the problem, as new systems will eventually end up in the same situation, that the systems they are replacing were in, if measures are not taken to prevent it. It will however give developers the opportunity to create systems that are designed to be more easily maintainable, leaving it up to the organizations to keep their personnel up to date with the systems.

Activity	Tasks
Process Implementation	Develop maintenance plans and procedures, establish procedures for Modification Requests, implement the Configuration Management Process
Problem and modification analysis	Perform initial analysis, verify problem, document the results, develop options for implementing the modification ,obtain approval for modification option
Modification implementation	Perform detailed analysis, develop, code, and test the modification
Maintenance review/acceptance	Conduct reviews, obtain approval for modification
Migration	Develop a migration plan, notify users of migration plans, conduct parallel operations, notify user that migration has started, conduct a post-operation review, ensure that old data is accessible
Retirement	Develop a retirement plan, notify users of retirement plans, conduct parallel operations, notify user that retirement has started, ensure that old data is accessible

Table 2.1: Activities and tasks of the ISO/IEC maintenance process

Chapter 3

Joly

Since most of the work in this thesis is based on Joly, having a good understanding of what it does and how it does it, is important. This chapter will therefore give an introduction to Joly, an application developed to handle many of the problematic areas surrounding the electronic delivery of mandatory assignments, and why Joly is now showing signs of decay. We will take a look at the system architecture, the framework stack and look at some of the issues with the application.

3.1 Background

In most of the programming courses that the Department of Informatics offers, the students have to pass one or more mandatory assignments in order to take the final exam. The normal handling procedure for these courses has been for the students to deliver their assignments as a compressed file by email to their teaching assistants who then have had to download each delivery and process them and give feedback. This opens up for many different problems such as students forgetting to attach their solutions, teaching assistants not getting deliveries because the mail client flags deliveries as spam mails and so forth. Most of these cases are trivial and are caused by human error and could be solved by having some form of a delivery system that provides an easy way to deliver and process the solutions. Mandatory assignments where students have to write code in various programming languages are perhaps not as easy to check for signs of cheating as it is to check an article for signs of plagiarism. Having a system where this is automated, and guaranteed to be done, would encourage students to write their own code instead of obtaining bits of code from the Internet and fellow student and then patching them together to make it work.

Joly is a web application that was developed in 2006 as a part of two masters thesis' by three former students and its purpose it to provide a solution for some of these problems[47],[22]. The application is written in Java and runs on a web server that powers a web page that can be accessed

from any machine with an active Internet connection. The web page provides an interface for the users and is designed to be easy to navigate and self explanatory to make the delivering and handling of assignments as efficient as possible. The application itself does not consist of code written only by the developers, it also makes use of open source frameworks and software, which will be explained in greater detail later on, that provide more complex functionality for the application to work. The main users of this application are of course the students, teaching assistants and lecturers, as well as the administrator, and what Joly does can best be described by looking at it from each of their angles.

3.2 The User Perspective

When students access the Joly web page they are met with a page where they are asked to select the course for which they want to make a delivery. This is followed by a series of pages where they are asked to select which assignment they are delivering, upload their assignment and any extra files they want to deliver and finally their user name and an optional message to the teaching assistant. They are then presented with a receipt for their delivery which can be printed out. This process does not take more then a couple of minutes to complete and guarantees the student that the delivery was made successfully. From this perspective, the application is presented as a step by step form which is filled and confirmed.

3.2.1 Teaching assistants

Teaching assistants get access to a different part of Joly, after logging in with a user name and password, where they are provided with four possible actions. They can change their password, add students to their group, if this was not done by the lecturer, and deliver an assignment for a student if the student failed to do so on his or her own for some reason. The last and most important functionality that Joly provides to the teaching assistants, is the possibility to select a given assignment and get an overview of which students that have delivered a solution for it and the option to view their solutions. Joly has a built in algorithm for detecting plagiarism and when a student delivers a solution it gets checked. If Joly suspects it of cheating, it flags the delivery as suspicious and displays this on screen. If the teaching assistant views a flagged assignment, Joly displays a side-by-side view of the suspected and reference solutions. Joly saves a lot of time for teaching assistants, as they no longer need to cross check deliveries for plagiarism manually. They have all the deliveries in one easy to manage place, making it easier for them to handle their group.

3.2.2 Lecturers and Administrators

Lecturers are given the same options as teaching assistants and are in addition granted the possibility of creating groups for their courses, assigning teaching assistants, adding the students who are attending the course and giving Joly old reference solutions that it can use when checking for plagiarism. The last option lecturers have is to create assignments and specifying the details of it, such as the deadline date. For lecturers, in essence Joly provides the tools and facilities to organize their courses. The administrator has all of the these options and they can also add lecturers. It is also the administrators responsibility to make sure that Joly is working properly at all times and to handle any problems that may occur and this means that he or she needs to have a deeper understanding of the system.

3.3 The Application Perspective

Although Joly can be considered a single coherent system from the user perspective, it is possible to split it into different parts depending on the type of tasks its needs to complete. Joly was developed with a three layered architecture in mind, which consists of a persistence layer, a business logic layer and a presentation layer. Each of these layers have separate tasks and are meant to be isolated from changes made in any of the other layers. The persistence layer handles all the tasks concerning the storage and retrieval of data, the presentation layer is responsible for the interaction between the user and the system and finally, it is the business logic layer that makes sure that communication between these two layers works seamlessly.

The application code is made up of combination of self produced and open source code. Most of the Open-Source Software that can be integrated into an application usually provide services that can be related to one or more of the three mentioned layers, requiring that additional code is written so they can be integrated properly. Much of the code that is self written is therefore code that enables these frameworks, such as data access objects that are needed on the persistence layer. Although Joly might not be considered a big system in a commercial setting it has a considerable scope being an academic project.

3.3.1 Information Flow Between Layers

Now that we know what the user can expect from Joly and how the application architecture is designed, we can look at an example of how the information flow between the different layers works. Figure 3.1 shows the information flow that occurs when teaching assistants, lecturers and the administrator log in to the system. When the user has typed in his username and password the first step is for the presentation layer to send this information to the business logic layer, which

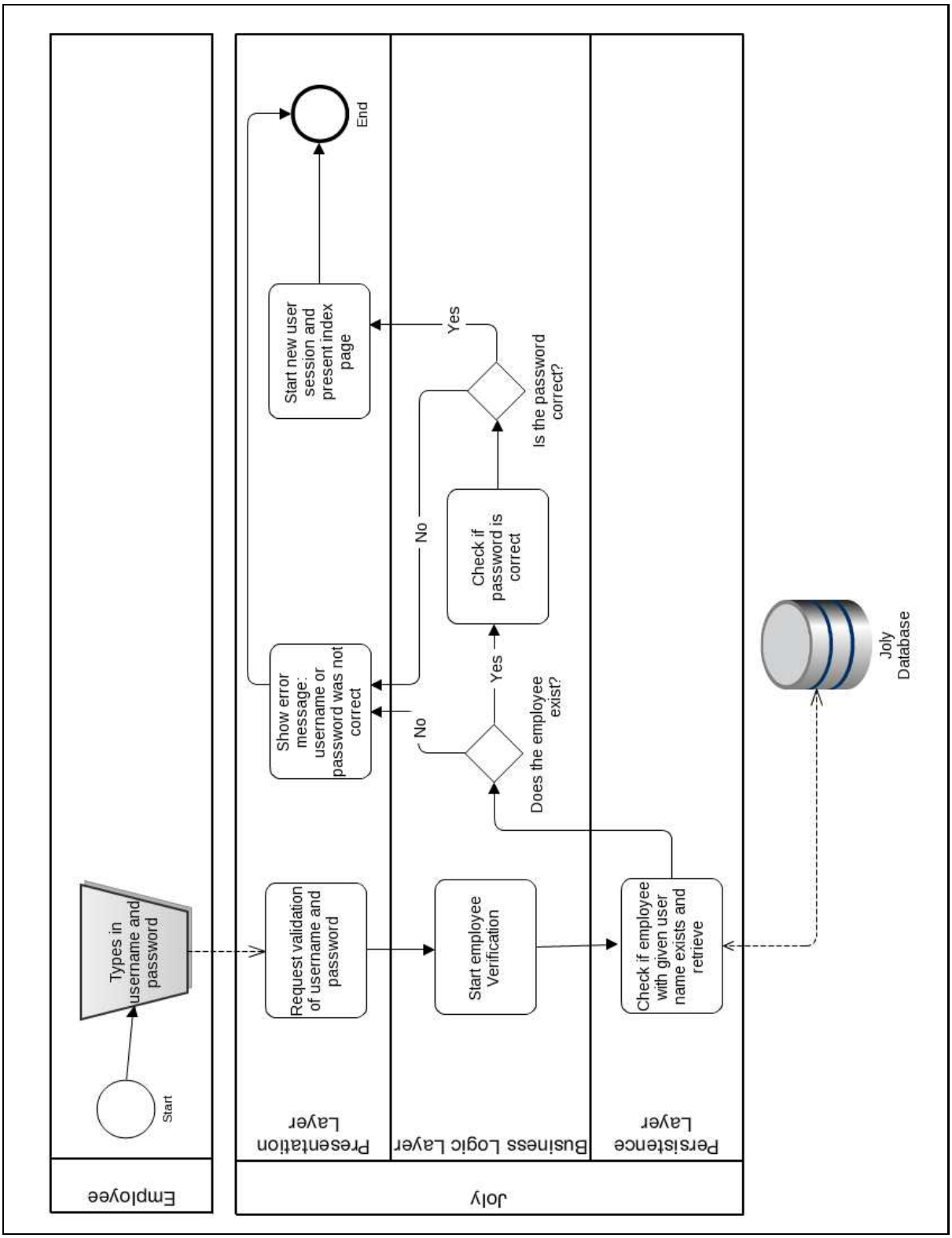

Figure 3.1: Information flow between the layers

knows that it must now check if this is an authentic user. The second step is therefore to ask the persistence layer if there is a user with the given username and password. The persistence layer will then query the database and give either a positive reply, denoting that the user exists by returning the user object, or a negative answer, saying that this user does not exist. The business logic layer then checks if the password that was given matches the one that is stored. The final step is for the business logic layer to tell this to the presentation layer so that it can either notify the user that submitted the information, that he or she was not accepted or show the user the index page. This example illustrates only one of many possible data flows in Joly between the different layers.

3.4 Software and Framework Stack

Open source software has become more and more popular over the years and as a result, it is very likely that you will find an open source framework or tool that can provide some or all the functionality you may need. A framework is in many ways just a library, a collection of resources such as classes and other code resources, that offers an application programming interface(API) that can be used to achieve certain functionality. The difference is usually that frameworks have a default behaviour which in most cases is useful in some way. Most frameworks do not allow the programmer to alter the framework code, but instead allows them to extend it by writing their own specialized code [56]. Many systems have multi layered designs like Joly and frameworks can also be categorized according to which layer they provide functionality on. Some of them provide functionality on only one layer while others may handle aspects on many layers. Joly has a three layered design and before we go into greater detail on what functionality the frameworks Joly uses provide on the different layers and how Joly makes use of them, it might be useful to get an overview of the software and framework stack.

As we can see from figure 3.2, at the bottom we have the virtual machine, which Jetty runs on. Jetty is the webserver that Joly is being depoyed on. Joly uses the Spring application framework which works on all three layers, and uses the Hibernate framework to map data between Java objects and relational database tables in a MySQL database. At the top we have JavaServer Pages(JSP) and Java Servlets that provide the dynamic content for the users. Now lets take a closer look at the software and frameworks, and try to understand them better.

3.4.1 The Virtual Machine

A virtual machine can best be described as the software implementation of a physical machine that shares resources with the physical machine that it is hosted on. Virtual Machines can be divided into two types, those that emulate a complete machine and those that only emulate the necessary parts needed to run a certain application or process. There are many benefits of using virtual machines

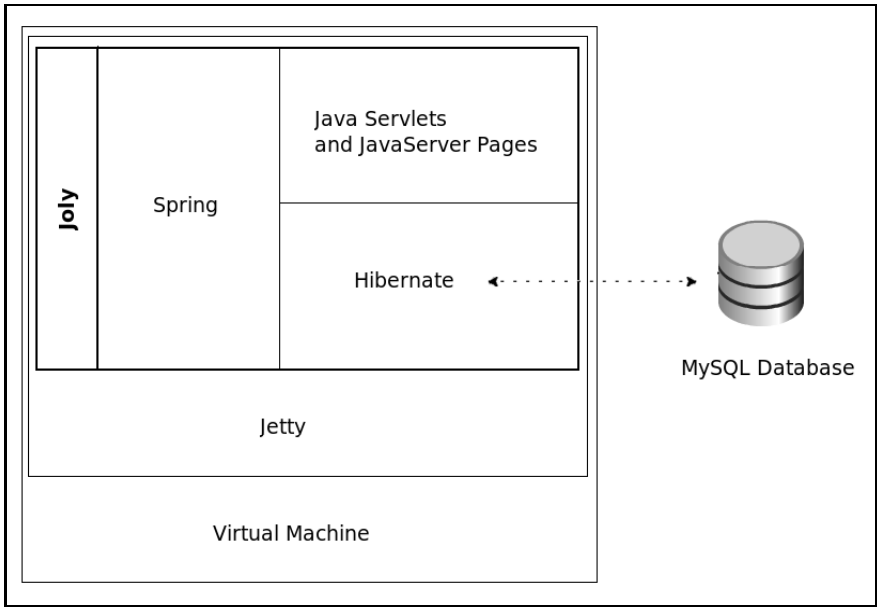

Figure 3.2: Joly's software and framework stack

as they can run their own operating systems and handle their own processes and applications. This allows the use of different operating systems and architectural implementations at the same time on one physical machine and provides an isolated instance of a machine that can be used, for example, for testing new software. In contrast to physical machines, they provide much better recovery and backup options since it is all software based. In addition to the practical benefits there is also the matter of saving resources. A virtual machine machine does not use any physical space, which makes them excellent for saving space, and are also a lot more cost effective than their physical hosts. The virtual machine that Joly was currently deployed on runs a 32 bit Redhat Linux operating system and this provides a very stable environment as well as allowing for easy connection and handling of the application[20].

3.4.2 Jetty

The first version of jetty was under development in 1995 and it was initially only intended to be a java HTTP server component, but since then it has grown into a free open source project. It is currently being developed as part the Eclipse Foundation and is released under the Eclipse Foundation Software User Agreement[57]. It has grown, from being a single component, into a suite of components containing, among others, an HTTP server and a servlet container and it is today either used by or part of many applications and frameworks[58]. Because its fast and small, jetty is well

Type	Role
Coordination Handlers	Routing requests to other handlers
Filtering Handlers	Augmenting requests and passing them on to other handlers
Generation Hanlders	Producing content

Table 3.1: The different type of handlers in Jetty

suited for providing web services in embedded Java applications.

Jetty is used for deploying Joly as a web application and this is done by packaging Joly as a web application archive(WAR), containing necessary files such as servlets, JSPs, class and image files, and placing the WAR file in the webapps folder in the Jetty directory. Jetty will scan the webapps and context folder at startup to check for any WAR files and corresponding XML context files, which describe the WAR file and also the setting in which the web application should be deployed[16]. An important thing to note is that the JSP files must use the same API version that is supported by Jetty. This is the default way of using Jetty unless it is used as a plugin for other frameworks such as Maven. Requests to an application deployed by jetty can be handled in two ways, either by using Jetty handlers or through use of servlets. A handler is a Jetty component and there are several built in handlers that can be used with some modification. The handler API provides the necessary methods for creating the three types of handlers seen in table 3.1 [17]:

This provides an easy way to identify requests, adding optional information to them and then providing the correct content for the request. If you want to do handle requests in another way, it is also possible to write your own handlers in accordance with the API. The servlet approach is done in a similar way but instead of having handlers that route requests and display content, the servlets do this themselves by using the Jetty Servlet API, which is how Joly does it.

Jetty versions and Alternatives

At the time Joly was first being developed, the original developers had narrowed down their choice of web servers down to two choices, Jetty and Tomcat. The third possible choice was Sun Microsystems implementation, but it was discarded due to being a commercial product and they wanted to go with an open source approach. The alternatives today are somewhat the the same, Apache Tomcat is probably still the most viable option because it can provide all the necessary features required and it has a proven development cycle[13]. Most of the other candidates are either developed as commercial products or do not have frequent updates, and as a result are probably not options we would want to pursue.

When Joly was being developed the latest Jetty version was 5.1 and version 6 was in the beta stage. There have been some new releases since then and the latest stable release is version 7 while

version 8 is currently under development. The biggest improvement seems to have been made going from version 5.1 to 6.1 when it was improved and made more lightweight and faster. Besides supporting newer versions of the Java Servlets and JSP API's and adding some new features that are really not relevant for Joly, the main thing is that it is that the latest version provides more stability and maturity.

3.4.3 JavaServer Pages and Servlets

Joly's user interface consists of a series of web pages that can be accessed by any machine connected to the university network. The user simply opens up a web browser, types in the URL and is presented with the first web page. A web page can be created in many ways and by using different languages of which the most simple and common one is HTML. HTML code is structured by using start and end tags, with code between them, to declare the different parts of a web page. HTML is used to create static web pages. Static means that they are not subject to change at any time, which means that a purely HTML written web page will always remain the same no matter what user input is made or when it is accessed, unless changes are made to the code. As with the development in other parts of information technology, we have seen a shift from static to dynamic web pages. Web pages that can update themselves, displaying fresh content as time progresses or as user input is made. This is in many cases impossible or hard to do with HTML and should therefore be done using other languages such as PHP or in Joly's case, JavaServer Pages and Servlets.

Listing 3.1: JavaServer Pages allow Java code to be written inside the scope of tags that begin and end with the % character

```
1 <%! private int accessCount = 0; %>
2 Access to page since server reboot:
3 <%= ++accessCount %>
```

JavaServer Pages allows the developer to mix Java code with HTML code, making it possible to have dynamic pages. Writing JSPs is done in the same way as with HTML pages, the structure is the same and Java code is embedded using special tags as seen in listing 3.1. By adding Java code it is possible to have some , or all, of the content be different each time the web page is accessed or certain user input is made. There are too many possibilities to name all of them but it can be a small detail such as displaying the current time or changing the background color or it can be the more obvious things such as providing new content for whole page and thereby in essence having one page serve as many different ones. Servlets provide many of the same things that ServerPages provide but they are structured in a different way. As explained, with ServerPages the Java code is embedded in the HTML code but with Servlets this is done the other way around. Servlets are written as normal Java classes where Java is used to either create or embed HTML code. Because

Table Name	Rows
Assignment	111
Comparison	3184
Course	13
Coursegroup	115
Elementoccurrence	0
Elementtemplate	1
Employee	64
EmployeeCourse	103
Group_Student	3466
Languagetype	1
Preprocessed	7711
Reportcase	0
Student	2397
Studentsolution	18785

Table 3.2: This table shows how many tables the Joly database has and the number of rows each table contains

of this, servlets are more suited when creating pages with a large amount of logic. Servlets can respond to HTTP requests and by utilizing the Jetty Servlet API, the Servlets in Joly are tasked with handling requests by routing them and displaying the desired content[19].

3.4.4 MySQL

Joly needs to store a lot of different types of data like user account information, course and assignment descriptions and solutions that are submitted. There is also the need to be able to relate the different data to each other, for example if an solution is delivered, it needs to relate the person delivering it with the assignment and also in which course the assignment should be registered. To do this, it needs to persist a lot of data. This can be done in a couple of ways depending on the size of the system and the different varieties and amounts of data. If the system is relatively small, it might be enough to just store the data in one or more files. However, once we reach a certain amount of data, the speed at which we can retrieve it is going to decrease and it will keep on decreasing if there are many types of data. Obviously this means that such a solution is not viable for Joly and that is why it needs to have a database in which to store the data.

A database is a structured collection of data and it needs to able to store and retrieve large amounts of data in an efficient way. Joly uses a relational database, which is based on storing data in tables containing rows and columns, where columns are usually the different attributes of an entity and the rows are instances of entities. Tables can then be related to each other by using

attributes as keys[53]. There are two other types of databases that were considered for the first version of Joly, object-oriented and object-relational databases. Both of these store data as objects and are therefore well suited to object oriented languages. They were however discarded by the original developers due to them having experience using a relational database in a previous course and this also influenced their decision when choosing which database management system to use.

MySQL is a relational database management system developed by MySQL AB and the first open source version was released in 2000 under the GNU General Public license[2]. Since then MySQL AB has been bought up by Sun Microsystems, who were later acquired by Oracle, and over the years MySQL has become more and more popular and is today one of the most used open source database management systems(DBMS) in the world. A DBMS provides you with an easy way of handling transactions between your system and the database by providing you with an API that conforms with the database you are using.

Alternatives

Although MySQL was picked as the DBMS, there were two other candidates that were considered, Firebird and PostgreSQL. Both of them were viable alternatives, PostgreSQL more so because it was considered to have a more stable community. There are many other relational DBMS' but the majority of them are proprietary. Among the possible alternatives today we still have both PostgreSQL and Firebird, as well as SQLite which is among the most widely used database engines[43],[52].

3.4.5 Spring

Spring is a module based application framework developed by SpringSource and is released under the Apache License 2.0. The first release was in 2003 and was written by Rod Johnson, but many things have happened since then. Today, Spring has a large community and a large team of contributors[21]. Spring does not force unneeded features on to the developers, instead the module based approach provides developers with the choice of picking only those things they need, and as a result it is entirely up to the developer how lightweight they want Spring to be. There are a dozen modules in total and they provide a wide variety of functionality. Looking at the documentation we have for Joly, we can find the following list of the modules and features that are used by Joly:

As we can see from the amount of services Joly makes use of, Spring is an important framework for Joly and there are not many complete application frameworks in the style of Spring out there that can provide all the features in one package. Many of them can provide the same functionality of one or two modules, but a solution consisting of using Spring and some other frameworks instead of the provided modules would probably lead to more complex solution while being harder to integrate.

Module	Features
Spring Core	Bean Factory/Container, IO resource
Spring Context	Application Context, Message Source, Mail, Validation
Spring Web	Servlet, Util, Bind, Multipart resolver, WebApplicationContext
Spring MVC	Controllers, Views, Models, View resolver, JSP/JSTL renderer
Spring ORM	Hibernate3 support
Spring DAO	Data Access Object(DAO) support, Transaction

Table 3.3: This table shows which Spring modules and features Joly uses

Alternative

A possible alternative to Spring is Tapestry. Although It does not seem to sport all of the same features as Spring, it is based on the Java Servlet API and claims to help developers structure their web applications as well as providing good scaling[14].

3.4.6 Hibernate

Since Joly is based on object oriented programming, all the data in the system is initially represented as objects. And since we are using a relational database we need a to transform these objects in a meaningful way to be able to store them in the database. This is where Hibernate comes in. Hibernate is an open source object-relational mapping framework for Java, providing mapping from Java objects to database tables, in addition to data query and retrieval facilities. Following many successful releases since the first version was started on in 2001, the development of new versions is currently being done by JBoss, a division of Red Hat, Inc. and is released under the GNU Lesser Public License[54].

In order for Hibernate to know how it should map the Java objects, some setup is needed. The hibernate configuration file, hibernate.cfg.xml, must be configured, it contains information such as connection details and which SQL dialect to use. The option to choose which SQL dialect to use is significant as it allows configuring it to work with any SQL variant DBMS, providing you with the option of changing it without having to rewrite any hibernate code. The Java classes that you want to persist can be mapped to the database in two ways, either by Java annotations in the objects or through the use of the Hibernate mapping file, which then states how Hibernate should store and load the objects you want to persist. Finally the classes working towards the hibernate API must be set up correctly[23]. Figure ref3.3 shows the hibernate architecture.

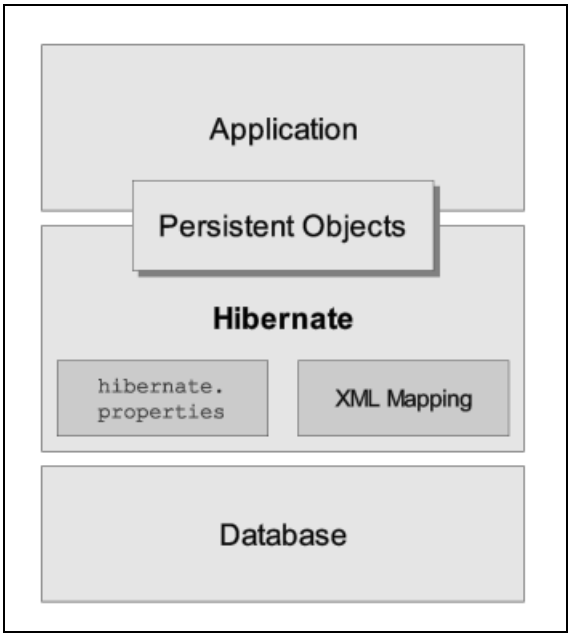

Figure 3.3: A high level view the Hibernate architecture[24]

Alternatives

There are a handful of other ORM tools out there that can provide most, if not all, of the functionality provided by Hibernate. Most of them can however be excluded before delving deeper into what they offer due to the simple fact that they are either developed for other platforms or programming languages or are commercial products[51]. In fact, the only tool that seems to be a fair open source alternative is MyBatis. It is being developed by the MyBatis Team and is released under the Apache License 2.0 and is nearing a decade of continued development since the first release in 2001[48].

3.5 Open Source Software

When deciding to use Open-Source Software it is important to consider a few issues. Firstly, to develop any piece of Open-Source Software, there is no need for the developer to have some form of higher education within informatics or work for an IT company. Anyone who develops an application, framework, software or writes a library for a programming language, and who wishes to share it with others is free to do so through the Internet, at for example open source distribution portals such as SourceForge[42]. What this means is that a lot of the time there is no guarantee that what is being offered actually does everything it says it will and that there are not any major bugs. The developers may not have the skill and knowledge to implement what they do properly and in a

standardized fashion, this is especially true for projects where there are only a few developers who implement all the functionality. It is also likely that they do not test their solutions properly before releasing them and all of these things put together mean that although it may seem like a framework or application is doing what it should, there may be cases where this is not true. Of course this is not to say that all Open-Source Software is stippled with bugs, in fact many of the authors go way and beyond to ensure that what they provide to the community is in perfect order and well documented.

3.5.1 Considerations

Doing a background check on the developers is often a good way to find out whether what they produce is worth giving a try or not[32]. The open source model provides a good way for developers to get feedback and from the community and there are today many developers, such as the SpringSource community and Apache Software Foundation, who are known to produce high quality open source code. If, for example, you are developing an application that deals with sensitive data such as personal information about users or passwords and usernames, and are considering using an open source solution for a part of that application, then it is likely that there are at least a couple of such solutions out there and that at least one or two of them that will work well enough. In later years it has also become more and more popular for commercial companies who need to develop non critical modules for their own products to make use of the open source community to help achieve this. Companies can in a sense outsource their work and get a cheap solution that they themselves and others can use[1].

Perhaps the most important thing to consider is whether or not the software you want to use has a regular updating schedule. Knowing that the developer simply will not abandon a project after its first release is important because it provides you with the assurance you need about whether or not it will provide you with same quality of service for the near future. That it will be kept up to date when newer features become available as well as fixing any bugs that may be discovered. This is perhaps the main reason why many of the leading open source products out there are developed by large communities or established companies who have the resources and know how required to do this. The reason Joly uses Spring, Hibernate, MySQL and Jetty is because all of them have this attribute.

3.6 Licensing

The Open-Source Software and frameworks Joly makes use of are all distributed under some sort of license, describing the terms and conditions of their use. In addition to this, they also give the licensee a copyright and to understand what a copyright is, we also need to know what an intellectual property is. An intellectual property is a product of the mind, for example music,

literacy, art, software or things that are not tangible in other ways, basically things that are not physical. An intellectual property right is usually granted to the author or creator of the work. There are many forms of intellectual property rights and among them are patents, trademarks and copyright. Copyright is a set of property rights granted by authorities to someone who has created something original or new. Creating something new and original is usually expensive and time consuming so the purpose of granting someone a copyright is to reward the author of the work since only the owner of the copyright has the legal right to copy, distribute and make alterations to the copyrighted work [49],[35].

3.6.1 Types

Although there are many different types of Software licenses, they can be categorized into two main types, those that are copyleft and those that are commercial. Copyleft licenses aim to make use of the copyright laws in such a way that the software they protect is kept free to the public.[10] This is done by stating in the license that people are allowed to use, modify and redistribute the software as they like while also keeping it free by stating that any derived work that is distributed must have an open source code. This does not mean that you are not allowed to sell software that is distributed with a copyleft license but it would make no sense to do so, which is why some companies offer Open-Source Software and then charge money for support services for the products. On the other side are the commercial licences, which are the exact opposite. These licenses restrict the licensees in such a way that it is impossible for them to share or alter the software. Software that comes with this type of licence usually costs money to acquire and since you are not allowed to redistribute it, this is the only way of legally acquiring it. Commercial companies, such as Microsoft and Apple, use these type of licenses to ensure that their products can generate income to cover their research and development fees.

3.6.2 Licenses to consider for Joly

Since we are talking about Open-Source Software, the licences to consider for Joly are all close to the copyleft side and to illustrate how many different types of licenses there are we can look at the four open source tools that Joly uses, MySQL, Hibernate, Spring and Jetty, to find that there are four different licenses. But what is the difference between them and what do they entail? All of the four licenses have things in common and to get an overview of what they impose on the licensee, it could be handy to represent their core parts in tabular form as seen in table 3.4[9],[40],[15],[8],[12].

As we can see from table , all four of the licenses grant us the right to redistribute and modify the software as well as stating that a copy of the licence must be included when redistributing. The big difference is when it comes to whether or not distributed copies must include a copy of the

	Allows re-distribution of source code	Allows mod-ification of source code	Must in-clude a copy of lisence when redis-tributing	Must keep source code open	Commercially viable
GNU General Public Lisence (MySQL)	Yes, only in source code form	Yes	Yes	Yes, but only if resulting software is distributed	No
GNU Lesser Gen-eral Public Lisence (Hibernate)	Yes	Yes	Yes	No	Yes, derivative work may be licenses under a different license
Apache License 2.0 (Spring)	Yes, in both source code and object form	Yes	Yes	No	Yes, allows integra-tion into propria-tory products and thereby broadening terms
Eclipse Public Lisence 1.0 (Jetty)	Yes, in both source code and object form	Yes	Yes, if the software is distributed in source code form	No	Yes, if the licensed software is seperate from the rest of the product

Table 3.4: An overview of the licenses in the open source tools used by Joly

original license. The GNU General Public License aims to keep protected software free by making sure that people can not block access to source code and that any program that uses code protected by it must also keep their source code open. This means that GPL is not commercially viable. However this only applies if your are actually distributing your program and this is an important point. The question then is whether or not the facilitation of a web page that can be accessed by anyone can be considered distribution. Joly is essentially made up of a server side web application that can be accessed by multiple users at once through web browsers. It is not distributed to any users as an application for download or through a physical disc of any sort and is only meant to be used "in-house" by students attending select courses at the university. It is therefore within reason to say that the source code is not required to be made available to everyone[11].

3.7 Joly as open source

If we consider that Joly was made open source, would it have a huge impact and make the system more vulnerable? Joly does not only serve as portal where students can deliver their assignments, it also has algorithms for detecting plagiarism and thereby finding out if anyone is trying to cheat. If these algorithms were made available to students it is not unlikely that some students will find weaknesses and try to exploit them, which in a worst case situation would make the algorithms useless. However, unless the algorithms are riddled with weaknesses, allowing for many different types of exploits, it would be possible to detect if students are trying to cheat this way by perhaps manually sampling some of the deliveries. It would also be possible for potential attackers to review the code and find weaknesses that could be used to bring down the server Joly is using, or obtain information such as passwords and other private data. This would mean putting an extra effort into making the system as secure as possible. The question of whether or not students would try to exploit a system that is hosted by the university could be raised. Getting caught is likely to lead to trouble so the fear factor here might ward off at least some students.

3.7.1 Benefits

Making it open source could also have a positive effect as it would allow other people to contribute by finding bugs, improving the application and keeping it updated. It could also serve as a good way of spotting weaknesses in the security aspect of the program. This would however require some organization, since there would have to be someone who could review and make sure that code that is submitted actually does what it intends and that the quality is good enough to be used. Overall, these are the two sides of the coin and both need to be weighed up properly before such a decision can be made of whether it is worth it or not.

3.8 Possibilities

Five years is a considerable time for any software to work flawlessly without updates, and especially programs that make extensive use of Open-Source Software as essential components. Most of the Open-Source Software frameworks and tools have an aggressive update and release schedule due to the need for stability and new and improved functionality. With the introduction of mainstream 64 bit processors and operating systems there has also been a transition from 32 bit to 64 bit software that benefits from this technology. What this means is during the last couple of years almost all software has been released for both versions, but this is something that does not look likely to continue for long. Developers and people who maintain systems are likely to slowly stop supporting 32 bit software and focus solely on 64 bit.

3.8.1 32bit to 64bit

An example of this is the operating system running on the virtual machine that Joly is currently being deployed on. It is running a 32 bit version of Red Hat Linux and due to circumstances where support for 32 bit versions of this operating system is being discontinued, the next version of Joly will have to be deployed on a 64 bit OS. This does not mean that Joly will stop functioning. One of the strong points of Java is that it is platform independent, which means that any java bytecode will run on any operating system as long as the appropriate Java Virtual Machine(JVM) is present on that system[29]. So this means that neither the source code nor the frameworks should have any problems being run on a 64 bit platform, which we will find out whether or not is true when we make the transition. In fact this transition can be seen as a purely beneficial thing as what it does is provide us with more memory and since the new servers have more powerful CPU's, we also have more power on that front. To utilize the increased memory we could look into perhaps loading parts of the database into memory so that we can access it more quickly and have write throughs to the database in regular intervals. This would likely reduce the time Joly spends on retrieving and inserting data into and from the database and make it faster, but this will have to be tested to find out if its worth the effort.

3.8.2 Multithreading

During the last couple of years the number CPU cores have increased and CPUs with anything from 2-6 cores have become mainstream. A Steam survey for the month of April 2011 shows that over 90% of the users either have 2 or 4 CPU cores at their disposal[4]. With an increase in the number of cores, the CPU can process more than one thread or task at a time and to make use of these extra resources, software needs to be multi threaded. Multi threading tasks that are known to be CPU intensive means that they are divided into smaller parts which can each be run separately on their own core, speeding up the process. It could therefore be worth checking if this can be done with any parts of Joly, such as the algorithm that compares solutions, which could allow for handling heavier loads more efficiently.

3.9 Versions

There is some confusion around which versions of Joly that are currently available to us and based on the documentation and source code we have available we can say the following. We have one on the server Joly is hosted on, one in a subversion repository and one which has been supplied by one of the former developers. The 1.0 version was developed by three people. Hanne Stensvik and Therese Vibekk, who designed the applications architecture and wrote most of the Java code, and

Version	Can be deployed	Algorithm Type	Developers
1.0	Yes	Vector Based	Hanne Vibekk, Therese Steensen and Christian Kielland
1.5	Yes	Diff	Cato Morholt and Jose Rojas
2.0	No	Diff	Arne Skjærholdt

Table 3.5: A first look at the different versions and authors of Joly. A revised version of this table can be se later in in figure 8.1.

Christian Kielland who designed and implemented a vector based plagiarism detection algorithm. This was done in 2006 and since then there have been three other developers who have been involved with trying to improve Joly. Cato Morholt implemented some new functionality and also made adjustments to the interfaces so that a new algorithm could be incorporated. This was a diff based algorithm that was implemented by Jose Rojas and so these two developers produced what we can call version 1.5. This version worked without problems and received several tweaks up until the autumn semester of 2008 when it started crashing because of what is thought to be bugs in the diff algorithm. Since then, any courses that have used Joly as a delivery system have not used the cheat detection algorithms in the system, but have instead relied on a modified version of Jose's algorithm that can be run offline. The last developer to have worked on Joly was Arne Skjærholt who intended to update the whole system and create version 2.0. Unfortunately he did not finish this work and also managed to loose the source code that he was working on because of a failure to do proper backup. So there are three versions in total that we are aware of at this time, as can be seen in table 3.5, but it is hard to determine the exact capability of the versions so that remains to be tested further.

3.9.1 Documentation

The only reliable documentation we have of the current system, and which most of what we have looked at up until now is based on, is written by the two original developers. They outline their architectural decisions and provide reasoning for their choices and this helps provide a certain understanding of Joly, but it gives little information about how things have actually been implemented. Explaining every line of code is of course not the way to go but it would useful to create a design document explaining the data flow between different parts and classes of the system as well as explaining how things are actually implemented. There are of course many more things that can be done to improve Joly, but before those things can be implemented, it would be best to make sure

Issue Relating	Description
Documentation	Little to no documentation on the changes that have been made to Joly since version 1.0
Server	Moving Joly from a 32bit server to a 64bit server could cause a problem
Deprecation	Method deprecation between the Java version Joly was last compiled with and todays version
Source Code	Many versions of the source code spread between different sources
Open Source Software	The Open-Source Software and frameworks are likely outdated
Bugs	There are likely to be bugs in the system that will need correction. Bugs in the diff algorithm cause crashes sometimes when students submit assignments

Table 3.6: List of issues that we know of so far

that the current version can work sufficiently as soon as possible. Joly would be needed in some courses already in the autumn semester of 2011, and having a working version would provide some much needed time to both create an improved version, as well as testing it properly to find out how much of the current code that works and that can be used.

3.10 Issues

Up until now we have already touched upon some of the potential issues we face. Little documentation on what the developers have done to the system since version 1.0, possible problem of changing servers, different source code versions and finally the matter of outdated open source frameworks and software. A complete list of issues can be seen in table 3.6.

3.11 Competing Systems

When Joly was first released, it was the only system of its kind at the Department of Informatics. This has now changed as a new delivery system, Devilry, has been designed by a group of students to be used in courses at the department. Devilry is written in Python and, like Joly, uses an open source approach to deliver essentially much of the same functionality, but without the cheat detection algorithms.[5] The difference is that Devilry takes the open source approach one step further by having all of the code as open source, inviting anyone who wants to contribute. It can be argued that there is no need for two applications of this kind at the department and it is likely that Devilry

will suffer the same problems that Joly is having, once the students who developed it graduate.

3.12 Planning

Although there have been attempts at updating Joly during the previous years, it is hard to find out exactly what their goals were and if anything was completed and this is partly down to a lack of documentation, logging and version control. Before you start to implement anything, it is best to have a clear plan on what you want to do, how you want to do it and what you do if things do not work out how you think they will. This can be the difference between success and failure and could be one of the reasons why previous attempts have failed. We will therefore try to set up a plan for our maintenance work when we have a complete overview of what we have at our disposal.

Documentation plays an important role, not only for those who develop the application, for which it serves as an overview of what they have accomplished, but also for anyone who will later continue their work and will need reference literature. Logging and version control is equally important and this can be done by using a tool like Subversion or Bazaar, which provide the opportunity to log each change as they are committed.

3.13 Summary

We have now reviewed Joly based on what little we have of documentation and looked at why it exists and how it is designed to work. There are several issues with the system as it is today and these need to be addressed but there are also possibilities for improving it. We must also ask ourselves if we want to create a brand new system or update the one we have. Starting from a clean slate would perhaps give more freedom in terms of architectural choices and make it possible to implement things more to our liking but it would require a greater effort to do this. However, with all the work that has gone into Joly so far from other developers it would be a waste not to use the foundation we have. In the short term, the current version needs to get up and running at a sufficient level again, so it can be used while we update it. After this it would probably be a good idea to make a viable plan on how to proceed so the next version can be made. Since the newest version we have of Joly is version 1.5, we will use it as the starting point for our work and see if we can solve at least the bugs that are causing it to crash.

Chapter 4

State of the System

Before proceeding with the creation of a new version, we would need to review the source code for version 1.5 to find out if there was anything we could learn from it and also test its performance. As was mentioned in the previous chapter, documentation on the implementation level is scarce so we will have to learn as we go. Arne Skjærholt, the last developer to work on Joly mentioned that the cause for the crashes could be a memory leak or the diff algorithm using too much memory. We will therefore be taking a look at how Joly performs with regards to memory and cpu usage, to see if the problems really are related to this area, and to see if it is possible to get version 1.5 up and running again.

4.1 Goal

Joly was planned to be used in the INF1000 course for the Autumn 2011 semester and as such, either the new version or the current version had be ready before the course started on August 20. Obviously this left us with too little time to create a new version, at least one that would be satisfactory, which meant that the current version would have to be made ready again for that semester. To be able to get it up and running we need to take a thorough look into version 1.5. Among other things, to see how things are implemented on the code level, while looking for any obvious flaws that could cause a performance hit and also checking for any bugs that could be causing the system to crash.

4.2 Problems

Ideally we would be able to perform a variety of different tests on Joly but this was where we met our first problem. After we had a look at what we thought to be version 1.5 of the source code we

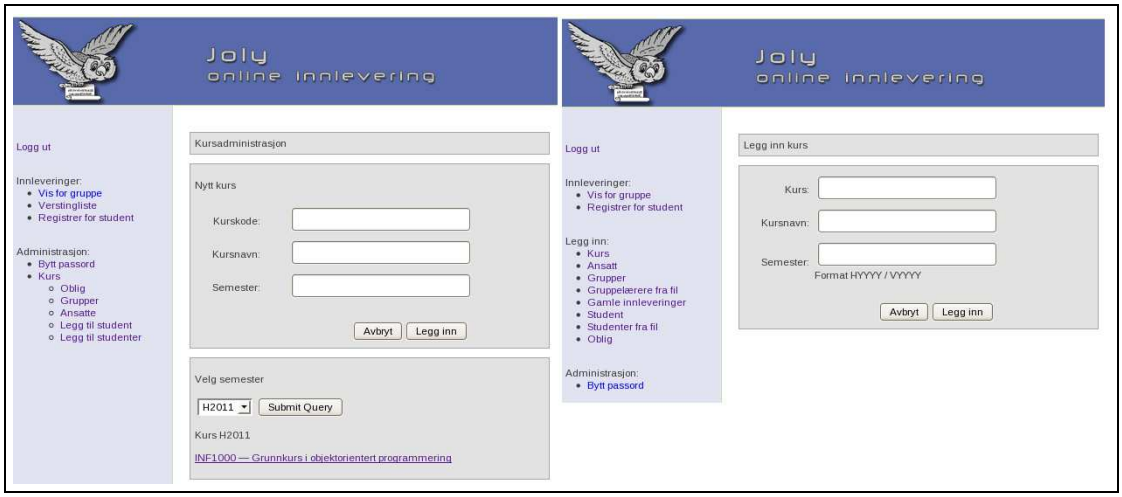

Figure 4.1: A side by side look at the user interface of the two different versions. Version 1.5 on the left and 1.0 on the right

go on to find out during an event where we deploy the system on two different machines, that there were certain differences between the pages on the two machines. The reason for this was that on one machine we had deployed version 1.5 and on the other machine we had deployed what could only be, due to noticeable changes in the user interface, which can be seen in figure 4.1, and functionality available between them, an older version which is likely version 1.0. Now this would have been trivial if not for the discovery that we did in fact not have the source code for latest version but only the compiled class files. This presented us with a problem since we could see that some of the functionality that was present in the latest version was quite valuable, such as being able to query the database for information. Since the JSPs are compiled at run-time we have the source code for the UI changes and it is likely that a good part of the source code between the two versions is the same. This could be explored by trying to decompile the class files in the the latest version and comparing them to the source code we do have. Depending on how well the decompiling works we could then find out if it is possible to recreate the source code and how much time this would take.

The previous virtual machine that hosted the application has been removed, by the technical staff at the depertmant, as a part of the transition from 32bit to 64bit software and has been replaced with a new server that runs a 64 bit operating system. This means that Joly must be deployed and tested on this server to see if there are any incompatibilities. This also means that we have an increase in memory to 8 gigabytes which, if there is a memory leak, will mean that the it will take more time for the application to run out of memory, prolonging its up-time. This of course does not solve the issue of a memory leak but it would mean more up time for Joly if we can not find it as a result of not having the source code available. Before we started testing the functionality part,

we thought it would be smart to do this on a separate database. This would give us the possibility to have one instance of Joly that would be available for use at all times, using one database, while we could use the other database for testing and debugging once we got started. Unfortunately we did not receive the new database before we were well into the testing part and this caused another delay.

4.2.1 Delete

When testing functionality we discovered that deleting a course would lead to an error when trying to deliver an assignment in any new courses that were created after a delete operation had been performed. I was later told by my supervisor that this functionality had not been implemented properly in version 1.0 because of time constraints. In any case it would not have been a big problem if we had only been able to restore the database to how it was prior to the testing. We had a database dump as a backup for this particular reason but because the dump had not been done in the correct way, some options enabled should have been enabled that we were not aware of, this was not possible. This meant that we had to wait for the support person responsible for the databases at the support department to come back from vacation before we could move on with further functionality testing and also some of the algorithm testing.

4.3 Testing the Algorithm

The diff algorithm removes all comments, line breaks and other things that have little to no impact on how the problem is solved such as variable names , from the solutions, creating files containing the "skeleton" of the solutions. Each skeleton is then compared to other skeletons to calculate how similar the solutions are. It was thought that the amount of operations that this required was using too much memory, because all the skeleton files are kept in memory, and thereby causing the whole application to crash. To find out if this was indeed the problem, we will test the algorithm first by itself and then through Joly, by making deliveries, while monitoring the memory and CPU usage. We will use three different amounts of solutions for the standalone tests, 1248, 623 and 311 solutions, to see how this impacts memory usage.

As we can see from figure 4.2 the algorithm starts by slowly consuming more and more memory and it peaks at around 136 MB of memory before it drops down a bit and starts to fluctuate between 107 and 84 until it finishes. On a computer with 8 GB of memory this will barely have any impact if only a couple of instances of the algorithm is running at a time. The old Joly server, which was running a 32bit OS, probably only had around 3.3 GB of memory available but it would still be able to handle many simultaneous instances of the algorithm. The CPU usage varied between 7 and 12% during the whole test. When running the algorithm with 623 and 311 solutions the picture was still

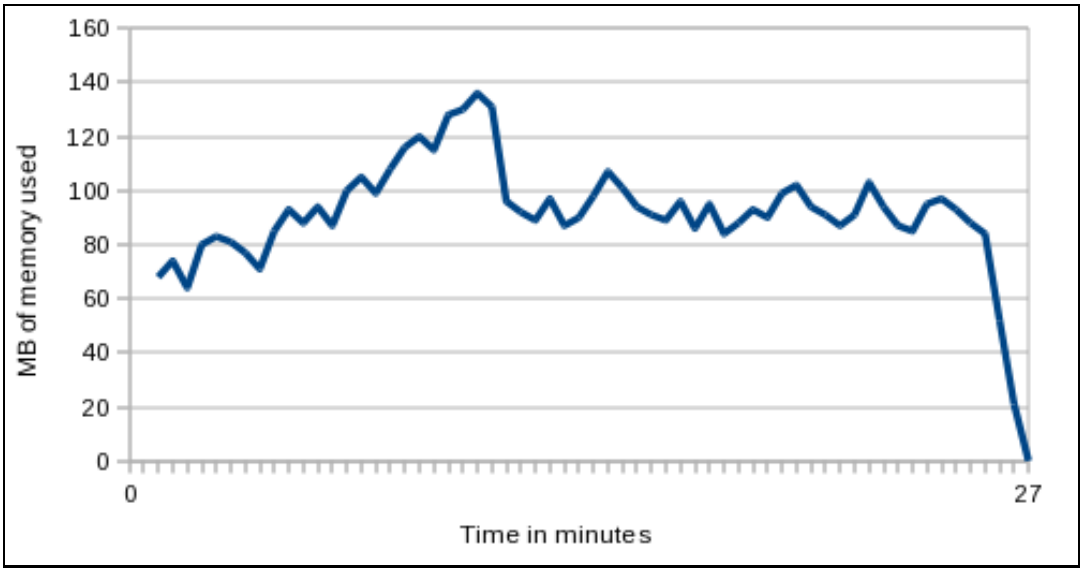

Figure 4.2: A line chart depicting the memory usage of the diff algorithm when compairing 1248 solutions

pretty much the same although the memory usage peaked at a lower point. This leads us to believe that the only way the algorithm could be causing an issue is if Joly runs many instances of the algorithm at the same time, something that would be hard to confirm without a working database.

4.4 Testing the Application

With the algorithm unlikely to be the culprit for hogging or leaking memory, the next thing we will test is the memory usage of Joly. For this purpose Joly was deployed and the application was tested by performing tasks a student or an administrator would normally do. The memory usage was monitored with the Linux System Monitor tool and we also used Eclipse Memory Analyzer, a plug-in to the Eclipse IDE, which analyzes Java heap dumps made using the jmap command, to check for memory leaks.

Joly was using 512 MB of memory when it had been deployed and this increased to 541MB after logging in as an administrator which is perhaps due to the fact that the application has to compile new JSP's when a new page is requested. With this in mind we then proceed with using the menu to navigate the different pages to get all the JSP's compiled and at this point the memory usage had increased to 605 MB. We then continued navigating around, creating courses and groups which seemed to have little impact, only increasing the memory usage slightly. However, if we keep clicking on any of the menu buttons repeatedly the memory usage will increase first by a few

kilobytes and then jump up a couple of megabytes. Doing this for a few minutes, we are actually able to increase the memory usage with over 50 MB, bringing the total memory usage up to 671 MB. We then let Joly idle for a while to see if it would start releasing any of the memory but after two and a half hours it still had not done this, but instead increased a little.

This was a bit strange and it could be due to a memory leak, so we will make a heap dump and use the Eclipse Memory Analyzer to analyze it. The analyzer revealed three possible memory leaks. Of the 3 possible suspects listed by the Memory Analyzer, two of them were objects generated by Jetty while the third was a java class generated by Spring. This points to the frameworks being responsible for the memory leak as it looks like they may not be releasing old objects for the garbage cleaner to dispose of. It is possible that updating the two frameworks could provide a quick fix for this but this will have to be looked further into.

4.5 Redeploying the Systems on the New Servers

Since we were running out of time at this point, because the new semester was about start, we had to deploy Joly on the new servers. Although we did not have the source code for version 1.5 we still had the compiled binary files which allowed us to do this. Since Joly has its own user at the department, and every user has a personal space, we did not need to copy anything over to the new servers. The only thing we had to do was change which database it should use so that it would now use the new one. The transition from 32 to 64 bit OS seemed to work without any problems at all and we can see the benefit of having a platform independent language like Java. We encountered a minor problem because the terminal machines at the university are configured to kill user tasks after logout to prevent them from hogging resources. Which means that after deploying Joly and loging out from the server Jetty would die and thereby taking Joly down with it. We will work around this by using the linux screen tool which allows a user to start an independent session on the same machine in which tasks that are started are non user specific, which hopefully will cause the task cleaner to ignore it.

4.6 Retrieving Source Code

Java programs are compiled using the Java compiler which transforms the .java files from simple Java code to platform neutral Java bytecode which is what the .class files are. Decompiling would then be the reverse process, going from Java byte code to Java code. Since we have the .class files for version 1.5 we can try to decompile these and see if we can get either working. Java files back or at least get somewhat complete files so we can try to reconstruct the missing source code if possible. There are several decompiling tools available so it will be interesting to see if any of them work

since there is no official Java decompiler. Although a decompiler would be a valuable tool to have in projects such as this, where there is missing source code, there are of course reasons for why there is not an official one and one being that such a decompiler could be used to retrieve the source code for commercial applications. This could potentially discourage developers from using the Java platform for their applications and this would not be good for Oracle business wise. If we have no luck using a decompiler we could try using a disassembler, which can transform java bytecode to an assembly language of sorts. Though this may at first glance not look to be very helpful, it could give us a clue as to how big the difference is between the two versions if we can get information like method and variable names. With this information we could then compare the .class files in version 1.5 with the equivalent 1.0 version files. Fortunately there is an official disassembler available for Java, called javap, and so before moving on with the decompilers We will try this one out to see how it works and if any valuable information can be learned.

4.7 Disassembling

Javap can be used directly through a command line interface such as the linux terminal. What you basically do is navigate to the same folder as the class file you want to use it on and then proceed by typing "javap <class name>" and hitting enter. The default behaviour of javap is that it prints out the package, protected, and public fields and also the methods of the class but it also offers several options such as displaying the disassembled code and variable names. We only tested with a few classes to see how it works, it would have been too time consuming to check every class. For each one, we test once using javap with the '-private' option enabled to get the local variables and all methods. The output was then copied into separate files before 'diff' was used to see the differences.

4.7.1 First Case

The first class we are testing it on is jolyImpl, this class handles much of the logic between the presentation and persistence layers and would give a good indication as to how different the two versions are. As we can see from figure 4.3, there are large differences between the two classes. In total there are two variables and two methods from version 1.0 that are not apparent in version 1.5 while the latter has five new variables and 25 new methods. This is a pretty big difference but if we look past the numbers and at which new methods were added we can determine that the actual changes to the data flow between the layers remain pretty much as it was in version 1.0. However, as expected, what it does mean is that a significant amount of new functionality was added in version 1.5 and that some of the method signatures have been changed. Based on the names of the new

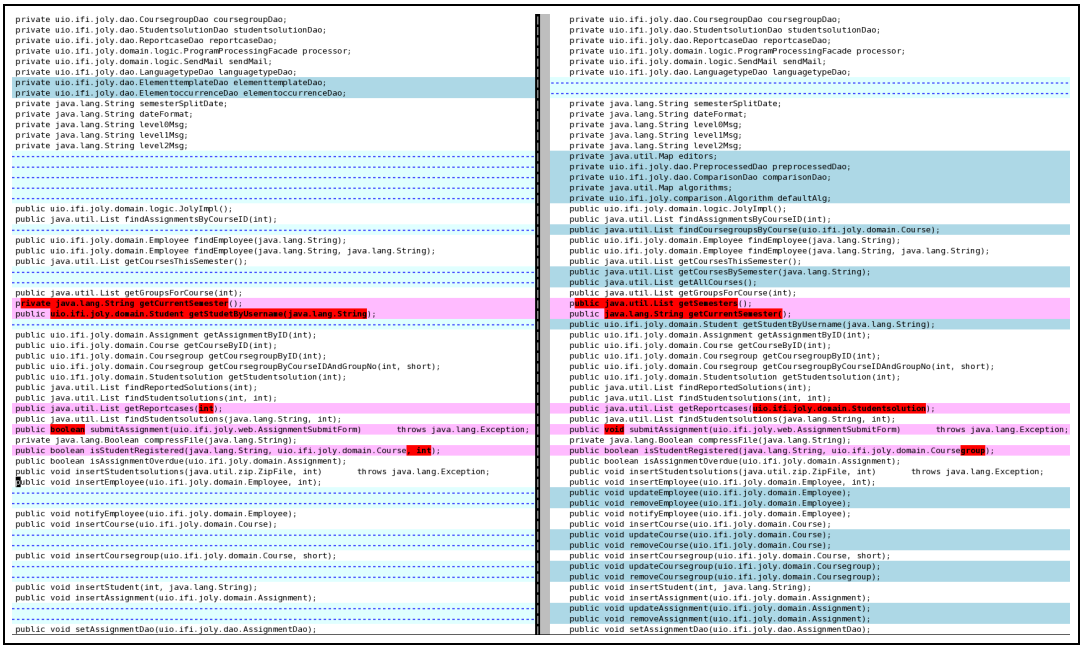

Figure 4.3: Version 1.0 to the left and 1.5 to the right. A stippled light blue line in one file means that it is missing compared to the same line in the other file, which is turqoise. A pink line with red marking means that the lines are not the same in both files with the red part showing the difference.

methods it is fairly easy to determine what most of them do since they are mostly getter and setter methods.

Running javap with the'-c' option enabled on this class we can see what the result would look like and as we can see from figure 4.4 the disassembled code looks very similar to assembly code.

4.7.2 Second and Third Cases

The second and third classes we tested were Assignment and Studentsolution which are both key objects in the system. Keeping in mind the changes in the JolyImpl class, it is reasonable think that these two had at most maybe two or three new methods that corresponded with the new functionality in version 1.5. This was the case with Assignment, two new methods and two new variables that reflected the changes but Studentsolution had some additional changes. As we can see from 4.1 there is no longer support for different languages and some of the methods that are used by the vector algorithm have also been removed. Although certain data structures were in place to accommodate different languages in version 1.0 it never actually implemented or supported it fully, so those two functions must have been deemed redundant. This must also be why the Vector and Line methods have been removed, since the new diff algorithm was implemented in version 1.5

```
public void notifyEmployee(uio.ifi.joly.domain.Employee);
  Code:
   0:   aload_0
   1:   getfield        #455; //Field sendMail:Luio/ifi/joly/domain/logic/SendMail;
   4:   aload_1
   5:   invokevirtual   #670; //Method uio/ifi/joly/domain/Employee.getUsername:()Ljava/lang/String;
   8:   aload_1
   9:   invokevirtual   #704; //Method uio/ifi/joly/domain/Employee.getPassword:()Ljava/lang/String;
   12:  invokeinterface #707,  3; //InterfaceMethod uio/ifi/joly/domain/logic/SendMail.sendPassword:
(Ljava/lang/String;Ljava/lang/String;)V
   17:  goto    25
   20:  astore_2
   21:  aload_2
   22:  invokevirtual   #503; //Method java/lang/Exception.printStackTrace:()V
   25:  return
  Exception table:
   from   to  target type
     0    17    20   Class java/lang/Exception
```

Figure 4.4: A small portion of the disassembled code for the JolyImpl class

Method	v1.0	v1.5
setLanguagetype	Yes	No
getLanguagetype	Yes	No
getLineAmount	Yes	No
setLineAmount	Yes	No
getVector	Yes	No
setVector	Yes	No
setReferenceSolution	No	Yes
isReferenceSolution	No	Yes
equals	No	Yes

Table 4.1: The methods that differ in the Studentsolution class between version 1.0 and 1.5

these functions were perhaps not needed anymore or changes have been made to how the algorithm works. Two new methods are in place for setting and checking if a solution is a reference solution which means that version 1.0 did either not differentiate between the two, which would be a bit strange, or handled this differently. If not It would mean that if version 1.0 detected a potential plagiarism attempt it would have no way of notifying whether this it was towards a reference solution or not. Finally, there is a new method called equals which we can assume is being used by the diff algorithm.

When you create a new assignment through the UI in version 1.5 you have the option of selecting either the diff algorithm, the vector algorithm or no algorithm. Having the option to select the vector algorithm makes it fair to assume that it should work since it would make no sense otherwise. This coupled with the fact that we now know that changes have been made in the Studentsolution class that will affect the vector algorithm, means we will check the class representing the algorithm

to see if any changes had been made. In version 1.0 this class is named ComparisonAlgImpl but in version 1.5 this has been changed, the vector algorithm appears to have been split into 4 classes while diff algorithm is in a separate folder. This means that a direct comparison is not possible, but looking at the methods that were available we see that they are completely different from the version 1.0.

4.7.3 What Have We Learned?

Given the amount of changes made in the few classes that we have looked at, it is possible to conclude that changes have been made to the majority of the classes. Most of the new methods and variables that have been added are there to provide the option of retrieving information from the database and as such the new methods are mostly getter and setter methods that are probably easy to rewrite. This will take some time but it is fully possible. Those methods that are not of this type, like the "equals" method in Studentsolution, can probably also be rewritten but may require some more time. By combining this with the disassembled code for each class it will also be possible to see which method calls are being made in each method, giving us further information about them. All in all the option to retrieve method and variable names from the classes has given us an idea as to how much effort has to be made to recreate the source code for version 1.5.

4.8 Finding a Decompiler

Taking into account the information provided by the disassembler, having a decompiler would obviously be an enormous time saver. Finding a decompiler that actually works however proved to be a tiresome task. Working our way through the different Java decompilers available on Internet, we soon find out that most of them are either outdated, and thereby only support earlier versions of Java, or no longer exist. The same goes for Eclipse plug-in decompilers as they either did nothing or only worked as disassemblers. After looking for a while longer we discovered has a standalone application for the Jad decompiler, which had not worked on Linux or as an Eclipse plug-in, called DJ Java Decompiler[34]. Since the decompiler is a standalone application it is fairly simple to use. Once it is installed on Windows, .class files become associated with the application and can be read and edited in its editor.

4.9 Examples

Comparing the decompiled classes with classes from version 1.0 we can see that there are some differences. For example, The decompiler places all the global variables at the end of the file and although all the methods are there, the order in which they are placed in the file is a bit different.

Further testing also shows that the decompiler can not, in some cases, find the name of local variables but instead seems to reference them from the local variable table that the Java Virtual Machine uses. This can be seen in listings 4.1 and 4.2, where the case names have been replaced with numbers with the name commented out behind it. For convenience all decompiled code snippets have a double line at the bottom.

Listing 4.1: A small section of a switch statement

```
1 switch (option) {
2         case 'm': case 'i': makeSkels_ifNotExist = true; break;
```

Listing 4.2: The decompiled version of listing 4.1

```
1 switch(option)
2                 {
3                 case 105: // 'i'
4                 case 109: // 'm'
5                     makePrepds_ifNotExist = true; //Skels
6                     break;
```

Initially the decompiler would not do anything when it came to annotations which seemed kind of odd, since they are a part of Java. This was however because we were not using the latest version and after updating it, the annotations were being decompiled as well. Listings 4.3 and 4.4 show the differences.

Listing 4.3: An example of an annotated method

```
1   @Id(generate = GeneratorType.AUTO)
2   @Column(name="assignmentID", nullable=false)
3   public int getAssignmentID() {
4     return assignmentID;
5   }
```

Listing 4.4: The decompiled version of listing 4.3

```
1     @Id(generate=javax.persistence.GeneratorType.AUTO)
2     @Column(name="assignmentID", nullable=0x00000000)
3     public int getAssignmentID()
4     {
5         return assignmentID;
6     }
```

For some reason the decompiler likes to decompile while loops into for loops. So the result of decompiling the code in listing 4.5 results in the while loop in listing 4.6.

Listing 4.5: A while loop

```
1       while (args.length > argNr
2             && args[argNr].substring(0, 1).equals("-")) {
3
4   ....
5
6           argNr++;
7       }
```

Listing 4.6: The decompiled version of the while loop in listing 4.5

```
1   for(; args.length > argNr && args[argNr].substring(0, 1).equals("-");
        argNr++)
2           {
3
4   ...
5
6       }
```

Indentation looks mostly the same except in cases where there is more than one statement on a line of in which case the decompiler splits it into several lines. Having global variables at the end of the files does not make a huge difference, in fact the only case where this will present a problem is if there are non static variables that are being accessed by a static method. Different indentation does not lead to any errors so unless it looks really bad it won't be necessary to make any changes to it and the for loops it generates instead of while loops are working in the cases we have seen.

4.10 Deploying with decompiled classes

As a test of how well the decompiler actually works we should see if it is possible to deploy joly when replacing some of the classes with their equivalent decompiled versions. Since annotations are not decompiled properly we know that classes that make us of hibernate and spring functionality most likely won't work. We decompile three classes, JolyImpl, Course(which contains annotations) and AddCourseController which is a Servlet and then recompile and add them to the application war file. This resulted in Joly failing to deploy correctly, only displaying the typical error page. This was also the case when testing the classes one by one. A possible reason for this that the classes are compiled with a newer version of the compiler than the original classes.

4.11 Summary

In chapter 3.9 we took a look at which developers were involved with the different versions of Joly and that one of the developers, Skjærholt, had lost some source code. Before we started testing the application we were not sure which version of the source code he had lost but as we found out it was for version 1.5. Throughout this chapter we have found out that Joly is not in a good state. We dont have the latest source code, there is functionality that is not fully implemented and there could possibly be a memory leak caused by the open source software.

Luckily, there are options when faced with such a problem. The javap disassembler, although not giving us any actual source code, can be used to learn how much source code we are missing. Unless you have way of turning the JVM instructions into Java code however, you cannot be certain of whether or not any changes have been made inside the classes. It was surprising to discover that among all the decompilers available, so many of them were outdated or refused to work on both Windows and Linux. Maybe it is because there is little use for them as a result of version control systems playing a bigger role in software projects. A search on google to look for other developers with similar problems generated a lot of hits though so maybe there is another reason behind this. Newer software might have a more rigid source code backup plan but for legacy software, software that is no longer supported but is still used, this may not be true and tools such as decompilers are handy to have in such cases. Despite the few shortcomings have seen so far the decompiler has now given us access to the newest source code version. How much we actually need it depends entirely on whether or not we decide to use it as a base for a new version but it may in any case be useful to have.

Chapter 5

What's next

A successful software project is often thought of as a well planned project. Every aspect of the project process is analyzed and potential problems are identified before any coding is actually done. This can help organizations save a lot of resources, which is why it is a part of every software development process. Since there are a couple of ways we can proceed with Joly we will now decide what to do next by looking at the pros and cons for each and finally decide on one of them. We will then start working towards the option we have chosen.

5.1 Options

The different ways we can proceed from here on can be split into two categories, those that are based on the current version of Joly and those that are not. While both have their advantages and disadvantages it essentially comes down to how much effort and time we can put into it. This, combined with how events have transpired so far, will affect our choice when we decide. Table 5.1 shows a list of our options as well some pros and cons for each. We first take a look at the options we ended up choosing before briefly discussing the ones we discarded.

5.1.1 Our Choice

Since we now have a way of getting the source code and because the decompiler is as impressive and well functioning as we have seen, we have decided that our first action should be to get a working source code for version 1.5. This will involve decompiling all of the class files and modifying them if necessary to make them work. The reason we want to do this is that it will provide us with a wider array of options later. Our Initial testing has revealed that it is not as straight forward as it seems as there is definitely some tweaking that needs to be done to get it running. When this is done we will

Option	Benefits	Negatives
New Java version based with different design	No need to decompile and get the current version working, Java is a familiar and well suited language	Need to design large parts of the system, requires a lot of effort
Html, PhP and Javascript	Lightweight, very well suited for web applications	Some learning involved, hard to get an anti-plagiarism algorithm to work
Python or other suitable language	Maybe more efficient languages and framework for this type of application	Requires a lot of effort to acquire needed skills and has already been done with Devilry
Update current version	familiarity with the application architecture and proven architecture that works	Considerable effort required to understand and get an overview of the source code

Table 5.1: Our options and some pros and cons for each one

hopefully have a near faultless version 1.5. For simplicities sake, we will from now on be calling this decompiled version of Joly version 1.5D.

After updating the open source tools we can then look at implementing new functionality, the amount of which will depend on what is required by the users and administrators. This whole operation would mainly be corrective type maintenance, correcting faults as the source code is decompiled, and also perfective maintenance. This would maybe not be a version 2.0 but rather a 1.9 or such, as it would be more of an update. It could be argued that a 2.0 version will need to be implemented in a way that makes Joly more maintainable. It may need a newer interface than the current one and it needs to have features that make Joly worthwhile to use ahead of competing systems.

5.1.2 The other options

Our other options include creating a completely new version that is still based around the same tools and frameworks as before but with a new architecture that is more maintainable. We would still use Java and as such we would likely be able to make use of some of the code from version 1.5. This will probably require more effort as we will need to design much of the system from scratch including the user interface. It would however provide us with the opportunity to make Joly more future-proof. Alternatively we could design a new system that is based on other tools

1	Decompile all the classes
2	Fix any errors in the decompiled class and make sure that they can be compiled again
3	Try to deploy with a mix of the old classes and some newly compiled classes
4	Compile the whole application as one
5	Successfully Deploy version 1.5D

Table 5.2: Part 1 of our maintenance plan

6	Add new Functionality
5	Do fault correction and optimization
7	Update the frameworks and tools
8	Update the UI

Table 5.3: Part 2 of our maintenance plan

and written in another programming language. Devilry uses Python with the Django framework to provide essentially the same functionality as Joly 1.5, so this is possible[7]. It might however be a waisted effort considering that this has already been done and could be seen as merely a clone. One possibility could be to create a more lightweight application using html, php and javascript. This would remove the need for tools such as spring and hibernate but any form of plagiarism detection could be harder to implement and might not be as efficient as doing it in Java.

5.2 Plan

Our highest priority is of course to have a working system at the end. That means decompiling all the class files and correcting any faults that are introduced by the decompiler. Then we can recompile the classes and try to deploy with some of them to see if that works. If that works then we will have some confirmation that our classes are good and we can proceed to make sure the whole application can be compiled together. A modified version of the ant build script from version 1.0 can perhaps be used for this purpose. So the first part and steps of our plan can be found in table 5.2.

Depending how how well this goes we can then proceed with the perfective part of the maintenance work which can be found in table 5.3.

The order in which the steps in the second part are done depends on what we want to prioritize as well as how much time we have left. While doing this we will be measuring how much time we spend on each task. I have estimated that we have so far spent around a 120 (more of a moderate

guess than an estimation) hours working on Joly in some form.

5.3 Decompiling and Correcting

Our first task is a fairly simple one, we decompile the classes from a fresh copy of the .war file file from the webapps folder on our server. We create a new project using eclipse and import the java files while maintaining their original package structure. The reason we use an integrated development environment(IDE) like Eclispe is to make use of the benefits it provides, such as error and warning highlighting and on-the-fly compiling each time we make changes to and save a Java file. After decompiling we have a total of 187 java files containing aproximatly 15118 lines of code. This is with the default formating of the decompiler. What we will be doing next is to review all the classes to see what kind of correction that needs to be done.

5.3.1 Nested classes

The number of class files and the amount of code seems like a lot for a small application(at least in a commercial setting) like Joly and one of the first things we will notice is that there are a lot of classes that have a '$' character in their names, like utilities$mapper or SSdump$1 for example. The reason that we have these classes is that if a class contains another class is compiled, depending on how the local class is declared or if the class is initialized in its parent class, we sometimes get so called anonymous inner classes[38]. So for example in Joly we have a class called utilities containing another class called mapper. If we compile the utilities class we get the two .class files, utilities and utilities$mapper. This is of course not the decompilers fault since it has nothing to do with how the Java compiler handles such classes. Since they are essentially duplicates of classes already defined in the their outer class we will simply remove these files from our project, leaving us with just 143 Java files.

5.3.2 Shortcomings

In section 4.9 we found out that the decompiler seemed to be able to handle most of our basic tests. Our tests were not very complex and so, as we shall now find out, it will be interesting to see what the limitations and shortcomings of our decompiler are and just how much effort this task will require.

5.4 Warnings

Warnings are the least serious faults a java classes can contain, and despite containing warnings it is fully possible to compile them and run them. If a line of code is highlighted with a warning it merely states that an error or undesired behaviour can occur during run time as a result of it. Our decompiled files contain two types of warnings, one because data structures, like Lists or Maps, would be unchecked and one because imported packages or variables would never be used or read locally. The latter ones could arise simply because the original classes did not use them and the author had just left them there or forgotten to remove them so those can probably be discarded and the packages and variables removed. While it is unlikely, the same could also be said of the unchecked Lists and Maps as the files would still compile and run, this is however unlikely as it is not good code practice to do so and if possible we will solve these warnings by type setting the data structures when they are initialized.

5.5 Errors

With the warnings handled we can now concentrate on the actual errors. There are many types of errors in Java and they can be categorized in several ways. In order for us to be able to compile our classes we will have to remove all the compile-time errors. As mentioned these are highlighted by our IDE and so we save time by not having to try to compile them in order to see them.

5.5.1 Imports and Annotations

We know that the decompiler can handle annotations, or at least the few we have tested so far, however it seems like it can not handle the imports required for some of the annotations to work. This means that code snippets like the one in listing 5.1 would be highlighted as errors.

Listing 5.1: The decompiler doesnt import the packages needed for some annotations. In this case javax.persistence.

```
1 @Entity
2 @Table(name="Comparison", uniqueConstraints={@UniqueConstraint(columnNames
      ={"extant", "submitted"})})
3 @Check(constraints="extant != submitted")
```

These are easily corrected however as the only thing that is needed is to add an import statement, in this case for "javax.persistence.*". We can now also see that the decompiler cannot handle some of the more complex annotations. For example, in the annotation in listing 5.2 we can see that there is some code missing between "table=" and the comma.

Listing 5.2: Example of an annotation that is missing some code.

```
1  @JoinTable(table=, joinColumns={@JoinColumn(name="coursegroupID")},
       inverseJoinColumns={@JoinColumn(name="username")})
```

When encountering problems like this we have two options. One is to look at which class we were dealing with and make an educated guess as to which table is in question, while the other is to look at the same class file from version 1.0 and if the class looks about the same then it is fair to assume that the annotation might be the same. In this specific case they looked very much alike and so we will simply insert the same code that is in 1.0. While looking at the classes in version 1.0 we also notice something that would explain the warnings we were getting in our decompiled classes. The two annotations in listing 5.3, one which is an instruction that tells the compiler to ignore warnings of the type "unchecked" and "unused", are not being decompiled. Which explains why we are getting the warnings.

Listing 5.3: Annotations that the decompiler does not decompile

```
1 @SuppressWarnings({"unchecked", "unused"})
2
3 @override
```

The second annotation is a notification to the compiler that tells it that the method that directly follows the annotation is overriding the method with the same name in the current class' super class. The benefit of doing this is to that if the super class does not contain such a method then the compiler will give a compile time warning. The other is that it helps other programmers in better understanding your code.

5.5.2 Illegal Modifiers

A lot of our decompiled classes contain "Illegal Modifier" errors. This error is shown when a variable is given modifiers, like private or static, which are not allowed. For example giving a variable in a method a modifier of the type private would cause this error since the variable would in any case only be local. In our case we were getting this mostly because the return values of some methods would be given a volatile modifier as can be seen in listing 5.4.

Listing 5.4: In some cases the decompiler adds the volatile modifier to the variables that methods return

```
1  public volatile int compare(Object obj, Object obj1)
2      {
3          return compare((Course)obj, (Course)obj1);
4      }
```

The volatile modifier in Java is used in multi threaded applications to either provide thread safety, be used instead of locks or both[18]. However, since Joly is not a multi threaded we will simply remove this keyword wherever we find it since it is of no use to us.

5.5.3 Constructors

A Constructor is a method bearing the same name as the class it is in. It is used when there is a need to initialize some variables in a class at the same time that an instance of the class is created. Our decompiler seems to be creating constructors for classes that originally do not have one.

Listing 5.5: The decompiler struggles with constructors in some cases.

```
1       MySQLReportcaseDao$1()
2      {
3          this$0 = MySQLReportcaseDao.this;
4          super();
5      }
```

The code snippet in listing 5.5 shows a constructor for the class MySQLReportcaseDao. Besides the fact that the class originally has no constructor there are a couple of things wrong. Firstly ,the name is appended with the characters "$1" which means that it will not be read by the decompiler as a constructor. Secondly, the super call is not the first statement in the method so this will show up as an error as it needs to be the first statement in the method. Lastly, the assignment statement, which does not make any sense, will give a compile-time error saying "No enclosing instance of the type MySQLReportcaseDao is accessible in scope". Since constructors of this type are not needed we will solve these errors by removing them from the classes.

5.5.4 Duplicate methods

A common error that we seem to be getting in some of the classes is duplicate methods. What this means is that we have two methods with the same name in the same class. The decompiler seems to be doing this for classes that are inheriting methods from their super class. So if the class CoursegroupAdminController extends JolyAdminController it must implement a method called createAddCommand. As seen in listing 5.6, not only does the decompiler add a duplicate of this method but for some reason it also adds the volatile modifier to its return value. Again, we will remove these duplicate methods from the classes to solve this error.

Listing 5.6: The decompiler creates a duplicate method for methods that are inherited.

```
1       protected volatile Object createAddCommand()
2      {
```

```
3           return createAddCommand();
4       }
5
6       protected Coursegroup createAddCommand()
7       {
8           Coursegroup group = new Coursegroup();
9           group.setCourse(new Course());
10          return group;
11      }
```

5.5.5 Missing external classes

In one of the classes we have an error stating that a data type cannot be resolved. This error is occurring because an import for an external library could not be resolved. The decompiler is however not at fault for this specific error, because after some searching on the Joly server i discovered that the is actually located in Jetty's "lib" folder. And because of this it was not automatically added to our project when we imported the folder containing our classes. So we will simply add the library to the project to resolve this error.

5.5.6 Assert

An Assert statement in Java can be used to check that a certain condition in a program is met, for example that variable a is greater than variable b, without writing any conditional statements[39]. If a is greater the program continues, if not an assert exception is thrown. Joly uses assert statements in a couple of classes in this way. What we can see however is that, after doing some testing, the decompiler cant properly handle assert statements.

Listing 5.7: An assert statement

```
1  assert xor(isEdit(), isDelete());
```

For example, decompiling the assert statement in listing 5.7 results in the following if statement and additional global variable seen in listing 5.8.

Listing 5.8: Example of an annotation that is missing some code.

```
1  if(!$assertionsDisabled && !xor(isEdit(), isDelete()))
2            throw new AssertionError();
3
4  static final boolean $assertionsDisabled = !JolyEditController.
       desiredAssertionStatus();
```

The method 'xor' just checks that only one of the variables isEdit and isDelete are true, since only one of the operations can be done at a time. The problem is that while the decompiler has created a variable '$assertionsDisable', it has not created the method desiredAssertionStatus which it assigns it to. This will obviously not compile and we will therefore replace the decompiled version with the regular assert statement and remove the extra variable.

5.5.7 Enum types

Enum is a data type in Java that only contains fixed constants and can for example be used to specify the days in a week or the directions on a compass[37]. In Joly it is used to indicate the different actions that can be performed by its users. The decompiled enum type ActionType from the class JolyAdminController which we will be discussing can be found in appendix A, section 1. We know from our experience with assert statements that the decompiler is capable of adding additional code when decompiling. While the decompiler was only adding an extra conditional statement and declearing a new flag variable for assert statements it is doing much more for enum types. It is in fact trying to create a new class that extends Enum, complete with its own methods and constructor. The decompiler is of course trying to decompile bytecode and this may then be how enum types are actually compiled. While it is interesting to see how enum types are compiled by the Java compiler, the decompiled code cannot be compiled again. It is for example not possible for a class to subclass Enum explicitly. Most of its methods are final and its constructor is protected. Secondly, it would be tiresome to read and unnecessary to have 35 lines of code instead of 4. By creating a small test files with enum types in them and decompiling them again with our decompiler we can see that it does so for all enum types and we can also conclude that the original code should be similar to the code snippet in listing 5.9. We will in cases like this therefore replace the decompiled code with similar code.

Listing 5.9: Using the decompiler to decompile this enum results in the code found in listing A.1 in appendix A

```
1  private enum  ActionType {
2    ADD,
3    EDIT
4  }
```

5.5.8 Labels, goto and continue

There are a few classes in which we have several errors relating to label, goto and continue statements. Label statements are used to alter the flow of a program and can be used much the same way as with loops. Continue statements are useful as they allow you to skip the current iteration

of a loop and check if conditions are met to do a new iteration. While the Java language does not support goto statements, other than it being a keyword, the Java Virtual Machine does. This is because Java is not the only language that uses the JVM and there are other languages that do support goto[55]. What goto does is allowing jumps to be made in the code to code blocks that are labeled. We have 4 classes where these types of statements are used and one of them is the JolyImpl class. The method we are looking at is insertStudentSolution, which can be found in appendix A, listing A.2. In addition to the syntax errors that these statements are causing we also have another error caused by the compiler trying to throw an exception. We will have to figure out what to replace the goto statements and block labels with so that they can be compiled.

Listing 5.10: A small portion of the insertStudentSolution method in the class JolyImpl

```
1            goto _L1
2 _L3:
3   ....(big code block)
4 _L1:
5         if(entries.hasMoreElements()) goto _L3; else goto _L2
6 _L2:
7     }
```

We will try to correct these issues by looking at the same method from version 1.0, found in appendix A, listing A.3, starting with the goto and label statements in listing 5.10. We can see that we here have a while loop enclosing the remaining code in the method, shown in listing 5.11. Notice that the condition for the while loop here is the same as for the last conditional goto statements in the decompiled version.

Listing 5.11: The while loop that the code in listing 5.10 is trying to replicate

```
1 while(entries.hasMoreElements()) {
2   ....(big code block)
3 }
```

What the decompiled version is trying to do is essentially mimmick the while loop in version 1.0. It first tries to jump straight to L1, check if the condition is met for L3 and jump there if so, or finish the method if not. So we will just replace it with the while loop to remove any errors related to these statements. The next oddity we have are the break statements, in listing 5.12, that are not present in version 1.0.

Listing 5.12: Break statements generated by the decompiler

```
1 break MISSING_BLOCK_LABEL_253;
2 and
```

```
3  break MISSING_BLOCK_LABEL_263;
```

Looking this up on the web tells us that these statements might be there because the decompiler cannot recognize or understand the bytecode it is reading. If we look at the 1.0 version of the method however we can see that there is no code there in that version either. It could be that these statements are generated simply because of the way the decompiler handles labels and for now we will just remove them. The last error we have is caused by an exception statement, and we can also note that there are some extra exception handling going on in the decompiled version that is not present in version 1.0. It looks like these are generated by the decompiler because it cannot handle the 'finally' block which is present in version 1.0 and shown in listing 5.13.

Listing 5.13: The 'finally' code block in the insertStudentSolution method in the JolyImpl class in version 1.0 of the source code

```
1  finally{
2   try{solution.close();}
3   catch(Exception ex){}
4  }
```

Finally is used in java to make a block of code execute despite of any exceptions that might occur. In this case Joly is closing the solution buffer if an error causes the program to terminate. In the decompiled version we only have the code snippet in listing 5.14 followed by a lot of extra exception handling. We will therefore replace this code with the code from the original version and remove the redundant exception handling.

Listing 5.14: A small portion of the insertStudentSolution method in the class JolyImpl

```
1  try
2  {
3   solution.close();
4  }
5  ...(lots of exception handeling)
```

5.5.9 Other minor errors

Besides the errors we have looked at so far we also have some other minor errors such as when a class is referenced. In these cases the decompiler lists the file path from the root folder to the class. As can be seen listing 5.15, for the Assignment class it lists the whole path instead of it being just Assignment.class.

Listing 5.15: The decompiler struggles with some class referencing

```
1
2 return getHibernateTemplate().loadAll(uio/ifi/joly/domain/Assignment);
```

As we saw in listing 4.4 in chapter 4.9, we had annotations which had hexa decimal numbers being assigned to boolean variables. This is of course not possible as Java does not permit anything but boolean values to be assigned to boolean variables. However it is also obvious that the hexa decimal values 0x00000 and 0x00001 refer to the boolean values false and true respectively. So we will just replace them with that. The last unique error we have was only found in one class and can be seen in listing 5.16, where we have a call to a method with two paramaters.

Listing 5.16: A parameter that the decompiler fails to decompile properly

```
1         binder.registerCustomEditor([B, new ByteArrayMultipartFileEditor());
```

As we can see there seems to be something missing where we only have a left square bracket and a B forming '[B' as the first parameter. By looking at the same class from version 1.0 it seems to be that it should read 'byte[].class' instead of just '[B', which makes sense, especially if we also consider what the second parameter is.

5.6 Missing Source Code Found!

On January the 31st of 2012, almost a year after we started working on this project, a copy of what is believed to be the source code for version 1.5 has been found. Jose Rojas, one of the developers who had previously created the diff based anti-plagiarism algorithm, discovered that he had the source code. Early on we had asked previous developers if they had a copy of the source code, which is how we got the source code for version 1.0 and the diff based algorithm, but had no luck getting code for version 1.5. Despite now having access to it we will ignore it for the time being, continue with correcting the files that we have decompiled, and then decide what we we will do with it in chapter 6.

5.7 Wrapping up: JVM instructions

When a Java file is compiled the output is written to a class file which can then be read by the Java Virtual Machine. For it to be able to read an execute the class file the code is compiled into bytecode. Bytecode is essentially instructions that the JVM reads and performs, and is similar to what assembly languages look like[33]. In the class PreprocessedBuilder the decompiler seems to have stumbled upon some code that it has flat out failed to decompile, leaving us with a combination of Java code and JVM instructions, seen in listing 5.17, that throw a bunch of compile time and

syntax errors. This class is a part of the diff algorithm and by combining what we know about the algorithm with how the rest of the class and the scrambled code look, we can say that there is at least some sort of String manipulation going on. We do have the source code for the diff algorithm, so it is not a big problem as such, but whatever code was originally there was sufficiently complicated for the decompiler to fail.

Listing 5.17: A small code snippet from the class PreprocessedBuilder. The full version can be seen in listing A.4 in appendix A

```
1          metadata[kl + 1];
2          1;
3          JVM INSTR dup2 ;
4          JVM INSTR aaload ;
5          JVM INSTR new #93  <Class StringBuilder>;
6          JVM INSTR dup_x1 ;
7          JVM INSTR swap ;
8          String.valueOf();
9          StringBuilder();
10         met_p.name;
11         append();
12         " ";
```

5.8 Summary

At the start of this chapter we decided that we wanted to use version 1.5 as the basis of our update. This meant that first we would need to decompile the class files using the decompiler we found in the previous chapter in order to recover the source code. As we have seen, most of the errors the decompiler has produced have had an easy solution but coupled with the amount of errors it has taken some time to correct them all. So far it looks like decompiling the class files is a viable option but we have yet to test our new classes. We have of course also had the benefit of having an older source code version available which has helped us a great deal. It is fully possible that some of them still contain run time errors or bugs that are not highlighted by our IDE, but version 1.5D has the potential to work. If we had recieved the source code for version 1.5 earlier, when we asked the previous developers, we could have saved a lot of time. As it is we now have to decide on which source code version we will proceed with.

Chapter 6

Reconstructed Code vs Original

During a project the circumstances on which the current effort is made may change. New information may be revealed that can make some of the work no longer useful or change the direction in which the project is going. Adaptability is important in software engineering and can often be decisive when it comes to how successful a project is. In our case we have found the source code, for version 1.5, that we thought had been lost and which our current course of action was based upon. We will now take a look at our plan and decide on whether or not to make any changes. Before acting further on our plan however, to finish things off with our decompiling effort, it would be interesting to see just how successful we were and to do that we will compare the 1.5D classes to the original 1.5 classes.

6.1 Revising our Plan

With the source code finally in our hands we have a new decision to make, we can either continue with our original plan or adjust it to reflect the new conditions. What we originally intended to do was to proceed with compiling and deployment testing of the classes, which are steps 3 through 5 in section 4.2. The compiling step is a somewhat trivial step as it would only involve an alteration of the ant build script, or maybe not even that since Eclipse compiles all our classes for us we could simply copy these into our war file, replacing the old ones. The deployment step would be a bit more interesting as it would show us if we were successful in reconstructing the source code but with the amount of classes that we have it would be time consuming to find out which classes would need more work.

6.1.1 Adjustment

Since we now have the original source code and to keep things interesting we will however skip these steps and jump straight to the second part of our plan, table 5.3. Perhaps the most pressing issue here is making sure that Joly is stable and to solve this we will have to look for and correct any code that might be causing it to crash, which means that we will be doing step 5 first. We have already tried to analyze the memory and cpu usage and did not find any serious problems there, so the code it self will need to be reviewed and the application tested.

6.2 Compairing

To get a good comparison we will look at classes that vary in complexity and how difficult it was to correct them. The classes we will look at are JolyImpl and PreprocessedBuilder, as well as JolyAdminController and JolyEditController.

6.2.1 Admin and Edit Controllers

We managed to correct these two classes by both testing how the decompiler handles assert statements and enum data types, and by looking at the version 1.0 files. Comparing the 1.5 and 1.5D version. we find that we are pretty much spot on in both cases. It is important to remember that the decompiler is reading JVM instructions, so it is of course generating Java code that it believes will generate the those instructions. This is a good example of how complicated it is for the compiler to generate instructions for something as simple looking as an Enum data type.

6.2.2 JolyImpl

It seems that we were correct in assuming that the block labels were while loops. What we didn't address in the section 5.5.8 was the continue statement, and it is also present in the original version of the class. What the continue statement does in this method is to save the programmer from writing an 'else' statement. The other thing that showed up in our 1.5D version class were the two break statements that had 'MISSING_BLOCK_LABEL' at the end. We thought that these might be indicative of code that the decompiler failed to decompile, but comparing the class with the original we can see that all the code is there. It is therefore unclear at this point why the decompiler writes these lines.

6.2.3 PreProcessedBuilder

The only thing we really failed to correct were the JVM instructions in this class but now that we have the 1.5 class we can check what these instructions are supposed to be. Based on the code lines generated by the decompiler for the 1.5D version, we can deduce that it at least appending some strings together and looking at the 1.5 code we can see in listing 6.1 that this is true.

Listing 6.1: The assignment statement that the decompiler only produced JVM instructions for

```
1     metadata[kl + 1][1] += met_p.name + " "
2         + met_p.begLineNr2
3         + "," + (prepdLine - met_p.begLineNr2 + 1) + " "
4         + met_p.nOfChars
5         + " (" + met_p.begLineNrOrg
6         + "-" + met_p.endLineNrOrg + ")  ";
```

What is interesting but perhaps not so surprising is the amount of code lines it generates for what is essentially a fairly straight forward assignment statement. A few lines below the code snippet in listing 6.1 we have another similar statement, which can be seen in listing 6.2. Interestingly this time it actually manages to decompile it properly as seen in listing 6.3.

Listing 6.2: An assignment statement located a few lines below the one in listing 6.1

```
1       metadata[kl + 1][0] = " " + class_p.name + " "
2     + class_p.begLineNr2 + ","
3     + (prepdLine - class_p.begLineNr2 + 1) + " " + class_p.nOfChars
4     + " (" + class_p.begLineNrOrg
5     + "-" + class_p.endLineNrOrg + ") " + met + "m:  ";
```

Listing 6.3: The decompiled version of the code in listing 6.2

```
1     metadata[kl + 1][0] = (new StringBuilder(" ")).append(class_p.name).
          append(" ").append(class_p.begLineNr2).append(",").append((prepdLine
          - class_p.begLineNr2) +
2      1).append(" ").append(class_p.nOfChars).append(" (").append(class_p.
           begLineNrOrg).append("-").append(class_p.endLineNrOrg).append(") ").
           append(met).append("m:  ").toString();
```

The only difference between these two assignment statements is in fact their locations. The latter statement is located in the 'for' loop outside the first one, and as we have seen before, the decompiler struggles with some of the more deeply nested code. Another interesting thing that can be seen when comparing the files is how the code snippet in listing 6.4 from the 1.5 file is decompiled:

Listing 6.4: An if and an else statement from version 1.5

```
1         if (linjen.equals("\t\tIn " + ID_ABRV + "1;") ||
2       linjen.equals("\t\tOut " + ID_ABRV + "1;") ||
3       linjen.equals("\t\t" + ID_ABRV + "0 = new In();") ||
4       linjen.equals("\t\t" + ID_ABRV + "0 = new Out();") ||
5       linjen.equals("\t\tIn " + ID_ABRV + "1 = new In();") ||
6       linjen.equals("\t\tOut " + ID_ABRV + "1 = new Out();")) {
7
8       //Removed commented out code
9
10        } else {
11      prepdIndex[prepdLine++] = i - 1;
12      nOfChars += linjen.trim().length();
13      class_p.nOfChars += len;
14      met_p.nOfChars += len;
15        }
```

All the code lines in the code block for the if statement are commented out and as a result there are no code lines that are executed in cases where one of the conditions are met. Listing 6.5 shows the corresponding code snippet from version 1.5D.

Listing 6.5: The decompiled version of the code in listing 6.1

```
1     if(!linjen.equals("\t\tIn i1;") &&
2        !linjen.equals("\t\tOut i1;") &&
3        !linjen.equals("\t\ti0 = new In();") &&
4        !linjen.equals("\t\ti0 = new Out();") &&
5        !linjen.equals("\t\tIn i1 = new In();") &&
6        !linjen.equals("\t\tOut i1 = new Out();"))
7               {
8                    prepdIndex[prepdLine++] = i - 1;
9                    nOfChars += linjen.trim().length();
10                   class_p.nOfChars += len;
11                   met_p.nOfChars += len;
12                }
```

As we can see the else statement has been removed and the code inside its code block has been moved into the code block for the if statement. In addition the conditions for the if statement have been changed to reflect that in cases where 'linjen' equals one of the conditions in the original if statement then it does nothing. Despite knowing that the decompiler most likely uses a set of rules to interpret the bytecode it is interesting to watch it create equivalent code.

6.3 Did We Miss Anything?

As mentioned in section 5.5.1 we are missing the suppress and override annotations but these are of course not that vital. We can also see that it will give method parameter variables in interfaces generic names if they are of a primitve data type like int or float. There are also some other small things like this that the decompiler cant decompile, but none of them should have any impact on whether or not the classes will function properly.

6.3.1 Debug Mode

In an attempt to see if it was in any way possible to get the variable names for the parameter variables from the decompiler we will run the decompiler in debug mode when decompiling. This resulted in the comments seen in listing 6.6, being added to the files when decompiling the interface MySQLCourseDao:

Listing 6.6: A snippet of the code from the class MySQLCourseDao when decompiling with the debug option enabled

```
1
2     // Parameter  0 added: Name this Type Luio/ifi/joly/dao/CourseDao; At 0
          0 Range 0 -1 Init 0 fixed
3     // Parameter  1 added: Name i Type I At 0 0 Range 0 -1 Init 0
4     // RetValue   2 added: Name <returnValue> Type Luio/ifi/joly/domain/
          Course; At 0 0 Range 0 -1 Init 0 fixed
5     public abstract Course findCourseByID(int i)
```

To have something to compare with we can look at the code in listing 6.7 which is from the CourseDao class which implements this interface.

Listing 6.7: A snippet of the code from the class CourseDao when decompiling with the debug option enabled

```
1     // Max stack: 3, #locals: 3, #params: 2
2     // Code length: 38 bytes, Code offset: 2412
3     // Local Variable Table found: 3 entries
4     // Line Number Table found: 2 entries
5     // Parameter  0 added: Name this Type Luio/ifi/joly/dao/MySQL/
          MySQLCourseDao; At 0 38 Range 0 37 Init 0 fixed
6     // Parameter  1 added: Name id Type I At 0 38 Range 0 37 Init 0 fixed
7     // RetValue   3 added: Name <returnValue> Type Luio/ifi/joly/domain/
          Course; At 0 38 Range 0 37 Init 0 fixed
8     // LocalVar   2 added: Name list Type Ljava/util/List; At 13 11 Range 13
           23 Init 13 fixed
9     public Course findCourseByID(int id)
```

Characteristic	1.5D	1.5
Readability	Low to Medium	Medium
Extendability	Low to Medium	Medium to High
Excessive code	Some	Little
Correctness	Medium to High	High
Ease of Understanding	Low	Medium

Table 6.1: An estimation of the quality level of the different characteristics of the codebase for version 1.5D after corrections compared to 1.5

As we can see there is a lot more going on for the class compared to the interface. This is of course because the class contains more data that needs to be stored, like local variables. The noteworthy line is the one where it loads Parameter 1, where it seems to have a value to load for the class method while for the interface it does not. Exactly why it does this is hard to say, but in any case it really isn't that important. Whether the name of the interface method parameters are correct or not does not make that much of a difference and if necessary it is possible to retrieve them by simply looking at the classes that implement them.

6.4 Maintainability

Decompiling the classes and getting them to work is one thing but that does not mean that they are in any way ideal for a new version. If a new developer were to work on these classes it would require a huge amount of effort to understand them as they are. There are close to no comments, the ordering of variables and methods is at times pretty confusing and some of the decompiled code is just cumbersome to read. These issues would need to be addressed and to do that would also require a lot of effort. It is therefore safe to say that the decompiled classes, as they are, are not easily maintainable. It would be hard to build further upon them without making the code even harder to understand or making numerous changes. A summary of the different characteristics of our decompiled version can be seen in table 6.1.

6.5 Correcting Without 1.0 Files

Throughout the decompilation effort we have of course had the java files from version 1.0 to look to when we were correcting the errors. But what if we didn't have them, would we still have been able to correct them? The answer is that we probably would have managed. That being said, it would probably have taken some more time to do so. Even though the they provided us with clues

and in some cases a simple copying of the code would work, we still did research that helped us understand why the decompiler was doing as it did and to make sure we were not missing anything.

6.6 Test

To wrap things up we will try to deploy Joly with one of our decompiled classes. We tried doing this once in chapter 4.7 but had no luck that time so the question is if our correction has had any effect. The class we will test this with is JolyImpl, the reason being that this class is at the center of the application and handles a lot of the interaction between the presentation and persistence layers. To do the test we simply import our decompiled class into the Joly war archive, Eclipse has already compiled the class so we do not need to do this separately. When we then deploy Jetty we can see that there are no launch errors and we are met with the login page when trying to access it. We are able to login and navigate the page and create courses so it seems like the application is working.

6.7 Summary

What is clear is that although the decompiler we have been using has been excellent when decompiling most of our classes, it is not without its faults. However we have been able to overcome most of these faults and, if we ignore the PreProcessedBuilder class, it is likely that we would be able to use our decompiled classes as the basis for a new version. As a free piece of software the decompiler has certainly done its job, and in cases like ours where only class files are available, it can be of enormous help. We know that we would be able to reconstruct Joly using version 1.5D but since we now have the souce code for version 1.5 it would make little sense not to use it. If the source code is fully functioning it will save us a great amount of time.

Chapter 7

Testing and Debugging

With the decompilation effort done and behind us, we can move on and start looking at how to improve Joly. Because Joly is a delivery system one of its key attributes is stability. Students should be able to rely on the system to be available at any time, but especially when it gets close to deadline time for deliveries. In the present state, that can only be guaranteed when Joly is used without any cheat detection algorithm. Before we proceed with looking at the source code we will now look at some possible ways of testing our application so we can discover the bugs and correct them.

7.1 Why

Testing an application can be both complex and tiresome depending on how comprehensively it needs to be tested. While critical systems require a high percentage of test coverage to uncover as many bugs as possible, others do not. So far we have tested the diff algorithm by it self without finding any serious issues in its memory and cpu usage. We have also tested the stability of the application without using cheat detection and can confirm that it does not crash without outside influence(ie. when system maintenance is performed). What we have not done is test the two together. The fault could after all occur because there is a problem with the interaction between the two.

7.2 Testing Options

In chapter 4 we tested Joly to find out its resource usage and we also tested its functionality to see if we could provoke any errors. The type of testing we did is called black box testing. Black box testing is when you test an application without any knowledge of the source code. At the time, that

was our only option but now that we have the source code we can perform white box testing, which is the opposite. White box testing is testing an application through the use of its source code, and while there are many ways to do white box testing, unit testing is perhaps the most common ways of doing it.

7.2.1 Unit Testing

A unit is generally the smallest part of a program that can be tested, like a function or a method. Joly consists of many classes, with the smallest having around 4 or 5 methods and the largest classes having over 10 methods. Already we can see that there would need to be a lot of tests, but we would also need to consider the contents of the methods. If there is for example a conditional part of code in a method we would need to test both cases where the condition is true, and the code is executed, and where it is not. If there is a loop, we would need to test it to confirm that it functions properly. This increases the amount of tests needed substantially. Unit testing can take a large amount of effort but, if the tests are well written, it is possible to achieve a high rate of coverage and discover many bugs.

7.2.2 Junit

Junit is an open source testing framework that is designed specifically for the purpose of doing unit testing and to promote test driven development[31]. One of the strong characteristics of Junit is that it is easy to use. All that is needed is to create a package structure within the project, that has a similar structure to the projects packages, that contain one test class for each class you want to test. The test class' then make use of the Junit framework to create test cases for each unit in the project class' that needs to be tested. This simple mechanic has made Junit very popular and it it even used in some courses at UiO.

+ Can get a good coverage of the whole code base

+ Easy to setup

- Time consuming

7.2.3 Test Script

Unit testing would work fine for a normal Java application, but with the number of classes we have and the limited effort available, something that requires less effort is preferable. The most likely case in which Joly crashes is when solutions are submitted. So if we can somehow simulate that lots of deliveries are being made we could perhaps provoke a crash. A possible way of doing this

could be to write a script that loops through a folder containing solutions and delivers them to the application.

+ Easy to perform

+ Requires little effort

- Not viable for testing more than specific cases

7.2.4 Manual testing

We could also try to test it manually by making deliveries through the Joly web page. This would take more time and it would also be hard to simulate cases where several deliveries are being made at the same time.

- Time consuming

- Hard to perform simultaneous deliveries

7.2.5 Adding Testing Functionality

Another possible option is to add testing functionality to Joly. This could be done by creating a separate page containing possible tests that can be run and that is available to administrators. When the application is deployed we could run tests at start up and output the results to this page, but also allow tests to be run at any other time. This would provide an easy way to test the application in the future and could be expanded to include tests for new functionality that is added.

+ Allows testing of Joly through the UI

+ Can be expanded with different tests

- Time consuming

7.2.6 Automated Web Testing

There are also some tools that allow for automation of web page testing, such as Selenium[41]. Selenium is a tool for automating web browsers using scripts, which can be written in any of the popular coding languages, or by recording user behaviour and then repeating it.

+ Can automate user behaviour

- Not possible to replicate certain behaviour

- Not compatible with all browsers

7.3 Debugging

Testing can be considered as something that is performed to uncover bugs that you may not be aware of, it is important in order to ensure that the software is behaving as intended. However if there is a specific fault that needs to be corrected that you are aware of you would more likely perform debugging. Debugging can be seen as the act of correcting bugs that you know are there and that you must correct. Many of the techniques used for testing can also be applied for debugging, such as unit testing, in which case you would only apply them to the areas of the software that you know might cause a certain problem.

7.3.1 Process

A typical way of debugging is to first reproduce the problem that is occurring. This can be done by giving the software certain input that can trigger it. When the problem can be reproduced consistently, the input might be simplified to narrow down the possible code area where the bug might be. Hopefully, when this is done, the code sections where the bug might be are small enough that one can find the bug. The bug(s) can then be found by for example checking the variable states via prints, checking stack traces after the program has crashed or by analyzing memory dumps.

+ Can target specific problems and correct them

- The effort required depends on the success of reproducing the bug

7.4 Summary

If our main goal is to find as many bugs as possible it is clear that we should use unit testing, as it will provide the highest code coverage. Manual testing is too cumbersome and time consuming so it is not viable. Adding testing functionality withing the application is time consuming and there is the question of how it would transfer to newer versions. Using a seperate test script could be an option but then it might be better to just insert the code directly into Joly for the duration of the tests. Automated web testing tools seem interesting but we would need to do more research and testing to find out if they can be applied in our case. We have a limited time to do our fault correction and so instead of testing parts of the system that might not be necessary we will adapt to the divide and conquer approach of debugging. In our case we have a specific problem that we want to target and if we can replicate the crash consistently it could give us a better idea of where it is occurring and how we can correct it. It is not certain that we will find the bugs that cause the crashing, in which case there are other things we can focus on. Such as updating the frameworks, solving the problem with the delete functionality or adding new functionality.

Chapter 8

Correcting

As we have discussed, many of the testing approaches require a substantial amount of time and effort to set up and perform. Which is why we have decided on correcting the crashing problem by debugging Joly. We will now look at the how we will accomplish this by first determining which parts of Joly we will need to include in our debugging, how we will try to replicate the crashing and finally what results we achieved.

8.1 Data Entry Location

Based on what we know about the crashing, and the need to test the data flow from top to bottom, we can narrow down where we should insert our debugging input to somewhere in the presentation layer. Meaning either in one of the JSP's or Servlets and since Servlets are more appropriate for handling logic code than JSP's we will need to find the appropriate Servlet to do it in.

8.2 Setup

What we need to accomplish this is, a collection of Java solutions, which we will place inside a folder so that they can be browsed by Joly. We have 1248 solutions at hand which we can use and so we will be making that many deliveries. A basic understanding of the data flow when an assignment is handed in, so we know where to make the changes we want, and some monitoring tool that we can use to view the resource usage during execution. The monitoring tools we will use are the same which we used in chapter four for our initial testing, Linux System Monitor, Eclipse Memory Analyzer and the bash program top.

8.3 Data Flow

A flow chart can be used for describing the order in which things happen within a program. Since we are focusing on deliveries it might be a good idea to have a look at the events that occur when a delivery is made in Joly. Figure 8.1 describes the process that happens within Joly from when the user presses the deliver button and until he either gets a success or failure message.

8.4 Input

What we want to do is simulate that a constant flow of assignments are submitted to Joly. So we will attempt to make a modification so that this happens every few seconds. In addition to this we also want to simulate cases where more than one assignment is submitted at the same time, so we must create something that can replicate this. While our main objective is to find the cause of the problem, it will also be interesting to see how Joly handles such a load.

8.4.1 Modification

Our modification will take place in the processFinish method in the AssignmentSubmitFormController servlet. The reason for this is that at this part in the program the solution that is currently being handed in is pretty much ready to continue through to the underlying business-logic layer. The solution is at this point stored in a AssignmentSubmitForm object called 'as' along with information about the student, assignment and course. So the only thing we need to do here is to start looping through our solution folder, select a file and create a byte stream that is then stored in a AssignmentsubmitForm along with a new Studentsolution object. The reason we want to create a new Studentsolution object for each solution is because otherwise hibernate will think it is persisting the same object twice and throw an error. The complete modification can be seen in listing 8.1.

Listing 8.1: The code we are adding the AssignmentSubmitFormController class in Joly

```
1 File dir = new File("/uio/arkimedes/s70/siamekd/solutions");
2
3 String internalNames[] = dir.list();
4
5 int i = 0;
6 //loop through the studentsolutions we have in our folder
7 while(i < internalNames.length){
8
9   AssignmentSubmitForm newAS = as;
10
11   //creat a bytestream from the studentsolution
```

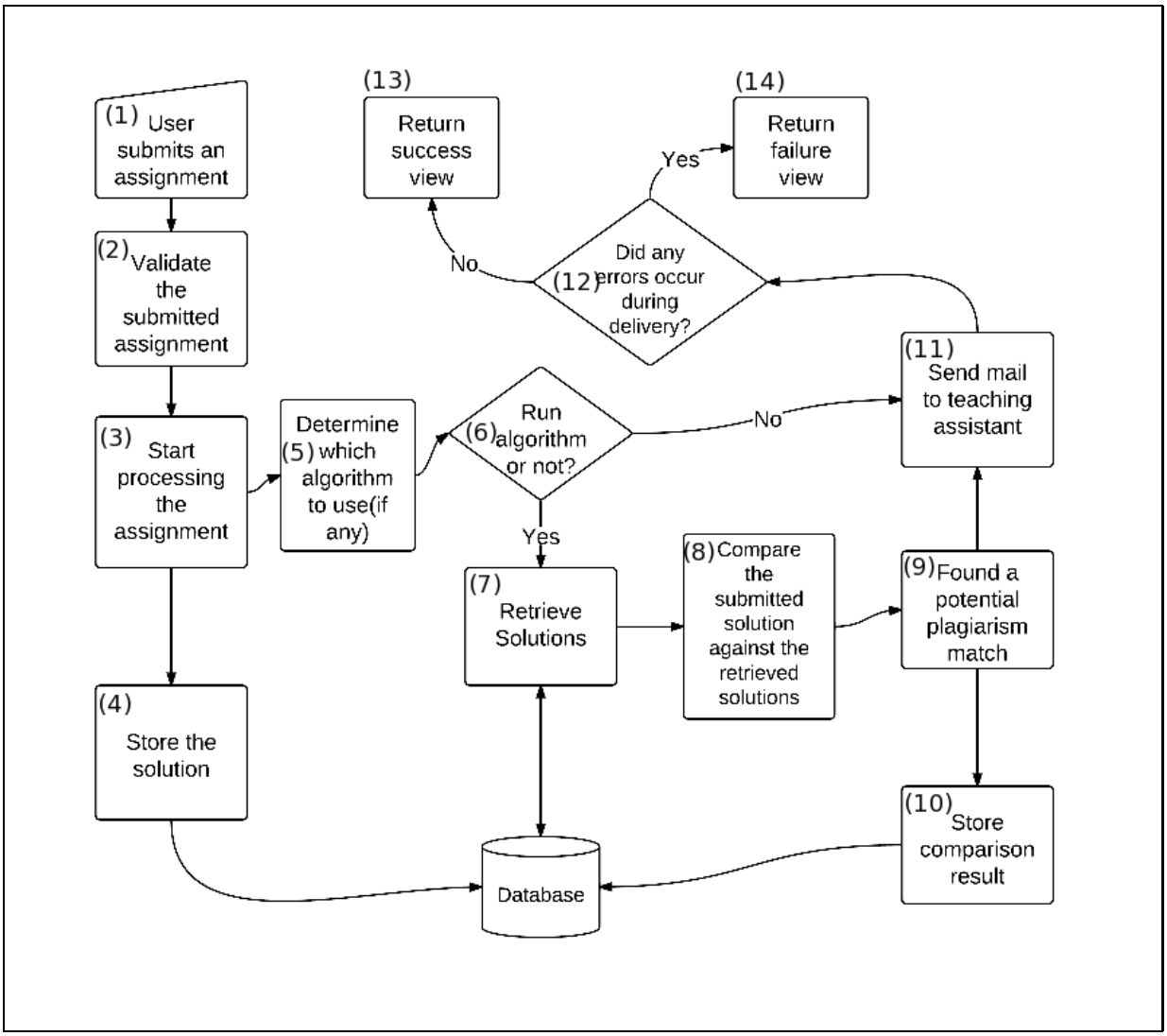

Figure 8.1: A flow chart describing the event sequence when a delivery is made by a user

```
12  UploadFile newUP = new UploadFile();
13  newUP.setFile(getBytesFromFile(new File(dir.getCanonicalPath()+"/"+
        internalNames[i])));
14  newUP.setFilename(internalNames[i]);
15  newAS.setProgramFile(newUP);
16
17  //create a new Studentsolution object for each solution
18  //if we dont do this then hibernate will think its persisting the same
        object twice and throw an error
19  Studentsolution SS = new Studentsolution();
20  SS.setStudent(as.getStudentsolution().getStudent());
21  SS.setAssignment(as.getStudentsolution().getAssignment());
22  SS.setText(as.getStudentsolution().getText());
23  SS.setDeliverytime(as.getStudentsolution().getDeliverytime());
24  newAS.setStudentsolution(SS);
25
26  //deliver the AssignmentSubmitForm
27  joly.submitAssignment(newAS);
28
29  //Pause for 3 seconds before next delivery
30  Thread.sleep(3000);
31
32  }
```

With this code it will be easy to simulate that deliveries are being at the same time. All that is needed is that joly is logged on to in for example different browser tabs, so that new connections are created, and the assignment delivery form is filled out and delivered.

8.5 1.5 Source Code?

Up until now we have assumed that the source code that we found was indeed for the same version that was running on our application server, version 1.5. This is however not the case. The source code we have is from an earlier stage which we find out when we try to run Joly and make deliveries. The reason we can draw this conclusion is because of the error messages we get, which we will look at soon, as well as how the code in the persistence layer looks. Now this obviously means that our debugging effort will increase a bit in scope, as we will have to sort out these problems first.

8.5.1 The Database

Between the version we have the source code for and version 1.5 some significant adjustments were made to the database. The developers working on Joly at the time decided to remove cer-

Version	Deployable?	Note
1.0	Yes	No significant bugs but only vector based algorithm is implemented
1.4	Yes	Some bugs that cause it to crash and persistence code that does not correspond with the database
1.5	Yes	Bug causes it to crash when running diff algorithm, no source code
1.5D	No*	Same as 1.5 and the decompiled source code has low maintainability. (*)Not tested if it can be deployed
1.6	Yes	Not finished, but will hopefully have no major errors after debugging. Based on version 1.4
2.0	No	Not finished and therefore does not work

Table 8.1: A revised list of the different Joly versions and their status

tain collumns in some of the tables, because they were no longer necessary, while some collumns were renamed to better reflect the data that was being stored. One example is the Vector column in the Studentsolution table which was removed because they were no longer using the vectore algorithm. Another was the two collumns in the Comparison table, solutionA and solutionB, which were renamed to submitted and extant.

8.6 Versions

Before we proceeed with looking at and correcting the errors we should get an overview of all our different versions. For simplicities sake we will call the version we are working on now for 1.4, since it it is older than 1.5. As we can see from table 8.1 we now have 5 different versions that we know of and while most of them can be deployed, they each have their faults. We will also add one more to this table, version 1.6, which will be the version that we have when we are done with the debugging.

8.7 Bugs and Corrections

Because of the changes made to the database what we need to do is essentially update version 1.4 to 1.5. To uncover the bugs we will simply deploy Joly, make a delivery and see where it crashes and what the error message it. We will continue doing this until we can successfully make a delivery using the diff algorithm.

8.7.1 Error Messages

The first error message we get, seen in listing 8.2, is NoSuchMethodError because there is a call in the JolyImpl class for a method called setCompressions in the Studentsolution class. The compression column is no longer in the Studentsolution table and has also been commented out from the Studentsolution class.

Listing 8.2: The error message thrown when attempting to call a method that no longer exists

```
1 Handler processing failed; nested exception is java.lang.NoSuchMethodError:
      uio.ifi.joly.domain.Studentsolution.setCompression(Ljava/lang/Boolean;)V
```

It was intended to be used to denote whether or not a solution should be compressed or not, in order to decrease the amount of storage space it would need in the database. The other errors are all directly tied to the database changes, they are 'SQL Error: 1054' errors which occur when an SQL statement is referencing an unknown column. The first unknown column we get is ReferenceSolution, which can be seen in listing 8.3, while the second one was Vector. The ReferenceSolution columns was removed because it was no longer deemed necessary to know whether or not a solution was a reference solution when comparing while the Vector column was removed because the vector algorithm is no longer used.

Listing 8.3: This is the kind of error we were getting as a result of the changes to the database tables

```
1 WARNING: SQL Error: 1054, SQLState: 42S22
2 Mar 12, 2012 3:59:21 PM org.hibernate.util.JDBCExceptionReporter
      logExceptions
3 SEVERE: Unknown column 'ReferenceSolution' in 'field list'
```

The insert statements are generated by hibernate and we can see that it is failing because it is generating a statement that contains a column that does not exist. In the Comparison table we know that the StudentsolutionA and StudentsolutionB columsn have been renamed to Submitted and Extant. What this means is that, in addition to the insert statements that must be modified, some of the query statements are also affected in the same way since they also reference columns that are no longer there, so we must deal with them as well.

8.7.2 Resolving the Problems

The first error we got was for the Studentsolution class so what we will do here is simply remove the isReference, compression and vector variables along with their mappings and the setter/getter methods. This is so that Hibernate doesn't include them in the insert statements it generates. In the MySQLPreprocessed and we have query statement that references columns that are no longer

there, so we need to modify this query to reflect that this. We also need to make a similar change in MySQLReportCase.

8.7.3 Making Use of Version 1.5D

Instead of just modifying the queries in MySQLReportCase we will seize this opportunity to make use of version 1.5D. Comparing the 1.4 class with the same class from version 1.5D we can see that the class is almost identical, if we disregard the layout of the class. So what we will do here is replace the SQL queries in the 1.4 class with the ones from version 1.5D. In the Comparison class we need to rename the variables and mappings to correspond with the name changes of the columns in the database. If we compare the 1.4 class with the 1.5D class we see that the 1.5D class actually has less redundant code while being correct with regards to the database. The biggest difference is that the 1.5D class doesn't have a constructor, while it is not needed in 1.5D it is in version 1.4 so what we will do is use the 1.5D class and modify it so that it has a similar constructor.

8.8 Will it still crash?

Now that we have solved these database inconsistencies we can move on with the debugging, there is however one thing we should consider first. The bug or bugs that crash Joly might not be there. The crashes were reported to have happened after some changes were made to Joly in preparation for the fall 2008 semester. Since the source code we have clearly is not for version 1.5, it might be that what we call version 1.5 is Joly after the changes while the version we have the source code for is from prior to the changes. We haven't looked for other differences yet because we are focusing on a smaller part of the application but it might be worth doing.

8.8.1 The Bug

Now that we are able to deliver solutions without the system crashing we can finally insert our code into Joly and check if we can get it to crash. Our loop goes to work and we can see that three deliveries are made, through debug prints, before it crashes on the fourth file with the error message and stack trace in listing 8.4:

Listing 8.4: The error message and stack trace the bug generates

```
1 java.lang.NullPointerException
2         at java.util.Hashtable.get(Hashtable.java:334)
3         at bmsi.util.Diff$file_data.<init>(Diff.java:772)
4         at bmsi.util.Diff.<init>(Diff.java:70)
```

```
5         at uio.ifi.joly.comparison.diff.ComparePreprocessedObjects.
              compareFixWith(ComparePreprocessedObjects.java:137)
6         at uio.ifi.joly.comparison.diff.ComparePreprocessedObjects.<init>(
              ComparePreprocessedObjects.java:102)
7         at uio.ifi.joly.comparison.diff.DiffAlgorithm.compare(DiffAlgorithm.
              java:124)
8         at uio.ifi.joly.domain.logic.ProgramProcessingImpl.ProcessProgram(
              ProgramProcessingImpl.java:51)
9         at uio.ifi.joly.domain.logic.JolyImpl.submitAssignment(JolyImpl.java
              :309)
10    ....
```

As we can see, there is a NullPointerException being thrown in the file_data class which is an inner class in the Diff class. Before we proceed with looking at what is causing the error we need to confirm that this is not happening because we are trying to deliver a bad solution file. So we will remove solution number four from our set and try again. The same error message apears once more at files 16,17,18 and 19, so it is apparent that this is not the case.

8.8.2 Explaining the Bug

NullPointerExceptions occur when a data structure or variable that has not been initialised is accessed so if this is happening we need to look closer at the code to understand why.

Listing 8.5: The for loop in which the error is thrown

```
1 for (int i = 0; i < data.length; ++i) {
2     Integer ir;
3     ir = (Integer)h.get(data[i]);
4     if (ir == null){
5       h.put(data[i],new Integer(equivs[i] = equiv_max++));
6     }else{
7       equivs[i] = ir.intValue();
8     }
9 }
```

The exception is thrown on line 3 in listing 8.5, which is an assignment statement where 'i' is set to whatever value is extracted from 'h', which is a Hashtable, with the key 'data[i]'. A Hashtable is essentially a lot like a Hashmap, the two main differences are that access to the hashtable is synchronized and that it does not permit null as keys or values. What this means is that we can rule out the Hashmap as being the perpetrator. This leaves us with whatever 'data[i]' is at the time the get() operation is performed. When we add a debug print, that shows us the value of 'data[i]' each time the get() operation is performed, we can see that the last entry in the 'data' array is null in

the cases where it crashes. To fix this problem we will add a null check to stop the get() and put() operations from happening if 'data[i]' equals null. This is more of a temporary fix than a permanent one. To have a real fix we need to find the root of the cause, which should be somwhere within the algorithm where it reads the solution objects into the array the first time.

8.8.3 No more crashing?

After inserting the null check we deployed joly again and ran our delivery loop once more. This time we can see that it delivers the solutions that used to fail without any problems and continues delivering until it either finishes or we stop it. To test how it handles simultanious deliveries we open two more tabs in our browser, log on to joly, and start the delivery loop on both of them. Experiencing no problems in this case either and seeing that both memory usage(which tops out at around 2,1gb) and cpu usage(which never goes above 31%) are reasonable, we will assume that the problem has been fixed.

8.8.4 Confirming the Bug

The reason why we can not confirm that the problem has been fixed with absolute certainty is that no one knows exactly which type of error message was thrown when the problem first started occuring. To reasure our selves that the bug we just fixed was not only a problem in version 1.4 we also try to deliver one of the solutions, that used to failed with 1.4, with version 1.5. We see then that it crashes with the same error message that 1.4 used to get so hopefully this means that this was the bug that caused the crashes.

8.9 Changes to the Interface

There are two problems related to the user interface that we are aware of. The first is that some of the text in the forms that users need to fill is not aligned with the correct text area, these are trivial to fix as we merely need to look at the correct JSP file and add line break tags to correct them. The second problem is related to the delete buttons that course administrators can use if they want to remove a course, assignment or course group. The reason they cause a problem is that they were not fully implemented the first time around. What should happen when you remove a course is that the assignments, course groups, and solutions in that course are also removed. As it is, if you remove a course, only the course is removed from the database. The same applies for coursegroups and assignments. When the diff algorithm then retrieves solutions and the course, assignment, or coursegroup it was related to no longer exist it throws an error, saying that the given column in the

database does not exist. For now we will simply remove the delete functionality, if it is needed, a cascading delete can be implemented at a later point.

8.10 Cleaning up the Source Code

The source code we now call version 1.6 is stable as far as our limited testing goes and we have corrected any inconsistencies that occurred with the database through the user interface. What we need to do now is clean up the source code so that it is less cumbersome to read. There is a lot of code in the various files that has been commented out. Much of it can be found in the files that the diff algorithm uses so we asked the developer if we could remove it or if it was serving some purpose but had been commented out for debugging reasons. It turned out that it was surplus code that is no longer needed so we can safely remove that as well along with our own debugging code.

8.11 Summary

When we started debugging we assumed that we would only be dealing with problems related to the diff algorithm. We then found out that we did not have the newest source code and had to make the necessary changes to the source code so that it would conform with the database so we could run our debugging code. We did not know exactly what was causing the crashes but we have managed to correct what looks to be the bug that caused Joly to crash and we now have a version, 1.6, for which we have a complete source code. Although we do not know whether or not the bug we corrected was the one that originally occurred we can say that the diff algorithm is now working and does not crash. As our final act, we used Subversion to ensure that future developers have easy access to the source code and so that that future changes are all recorded with the revision history log.

Chapter 9

Conclusion

In this masters thesis we have looked at several aspects of software engineering with our main focus being on software maintenance. Over the course of the project it changed from looking at how perfective and corrective maintenance could be performed on Joly, to how a system's source code could be recovered when only the binary files of that system are available. Before we move on to discuss the outcome of our investigations let us first summarize the key events in our project.

9.1 Events

Our aim was at first to investigate why Joly was not online anymore and get it up and running again on new servers, and then to correct any bugs we might discover and implement new functionality. This prompted us to review Joly from a design perspective, what the application is composed of and why the components were chosen so that we could get a better understanding of the system. Once we had Joly up and running on the new servers we started looking at the source code and how we could test it in order to find bugs. This proved to be harder than we expected as there was not only one version of the source code but several, and they were scattered between different developers and backup systems. While we had the class files for version 1.5 which was the latest version, and the only source code we had available at first was for version 1.0. Which is why we decided to make an effort at recovering the source code for version 1.5.

Armed with a decompiler that we found on the internet we started decompiler our class files and recovered the source code which we then looked through. We found a number of errors in the decompiled files that we proceeded to correct and as we were finishing this up we received news that the source code for version 1.5 had been found by a former developer. We used this opportunity to compare some of our decompiled files, which we finished correcting, with the ones from 1.5 to

check if they were concurrent with the original files. Confident that we now had the latest source code we proceeded with debugging them only to find out that that they were in fact not of the latest version. This meant that we first had to make the necessary changes to the source code in order for it to be consistent with the database before we could do any more debugging.

In the end we found a single bug in the plagiarism detection algorithm that was causing it to crash. We corrected this along with some other problems we knew Joly had before we finally cleaned up the code. Joly can now be used without any crashes occurring during deliveries which was the biggest problem we wanted to fix. We went from having no new source code, to having a decompiled source code version, to having source code that was inconsistent with our database to finally having a version that we can safely say has no major problems.

9.2 Questions

As we progressed through the project we had the opportunity to make some important decisions that affected the future of Joly. Would we choose differently now, considering how things progressed and how it turned out? Would the success we achieved with decompiling the class files have been possible if they did not run in to begin with?

At first we were contemplating whether or not we should start from scratch and build a completely new system. We decided against it because it was believed at the time that we already had a system that provided a good platform we could build upon. I believe this was is more or less a good decision, Joly is a well thought out system and expanding upon it is not a hard thing to do. The problem was that we did not have the newest source code which we did not know that at the time. From a pure programming point of view, Joly would perhaps be in a better state today, with more functionality as we would have spent a much bigger amount of time on it, but on the other hand we actually learned about how to recover source code from binary files.

Finding out that we did not have the newest source code brought up a new question. Should we build upon version 1.0 or find a way to retrieve the source code for version 1.5? We chose to try to retrieve the source code for 1.5. This choice was made based on curiosity as much as anything else, would it be possible? The answer to that is yes, as long as you can find a decompiler for the language it is written in there is a chance. But even then it depends on how good the decompiler is when deciding to whether or not it is a viable option. The decompiler we used was surprisingly good and produced understandable code in almost all cases. If we look away from the one case where it produced a block of JVM instructions the errors it produced were all correctable. Some of

the errors were more obvious than others and we corrected the ones we found but we can not be a hundred percent sure that we managed to correct all of them. Building upon version 1.0 would have been the safe choice but looking back, trying to decompile the class files was worth it. It was interesting to compare the decompiled versions of the class' with the ones from version 1.4 as it served to highlight the areas where the decompiler did things differently and the areas where it was spot on.

During our decompiling effort we got some good news, source code had been found which was thought to be version 1.5. Looking at it more carefully however we discovered that this was not the case. The source code was clearly newer than version 1.0 but due to incompatibility with the database it was clear that it was not version 1.5. The reason we discovered this a bit late was due to the fact that we wanted to finish decompiling Joly before looking at any original files so that we could properly assess whether or not decompiling would work. At this point we still could have chosen to continue with our decompiled version and tried to see if we could get it all working and not just a couple of the classes. The fact was though that version 1.5D was not as maintable as we would want it to be which would mean more work for future developers if we had chosen that route. It was nice to see that we were still able to make use of some code from version 1.5D in making version 1.6. Overall, considering the outcome we can say that things went well the way they happened.

One of the main reasons the decompilation of the class files went as well as it did was due to the fact that we knew version 1.5 could be run, and so if the decompiler produced good source code then version 1.5D would have good shot at working. Had this not been the case we would most likely have had to spend a considerable amount of time trying to find any errors that could cause it to not run. This coupled with correcting the other errors the decompiler produced and that the code generated was not the easiest to maintain would have been hard. We would however still have been able to salvage the code but it would have been a project on its own to get it working.

9.3 On Open-Source Software

Joly is a software system that relies heavily on Open-Source Software without being open source itself. From seamlessly handling the transferring of objects to and from the database to processing web forms, the frameworks are utilized to ease the programmer's job. Open source software has a lot going for it. The source code is freely available so anyone can download it, make adjustments to it and use it if they want to. There is a huge community that can help you with any problem you might have and if you want to you can contribute to most of the open source projects you want as long as you have sufficient skill. Open source software is however not without its drawbacks.

A software system that relies on open source frameworks adds an extra level of complexity and will increase the time it takes for a programmer to understand the ongoings in the system unless they are already familiar with the frameworks being used. Before you use a particular open source framework you have to find out a couple of things. The first is whether it does what you need it to do and if not then can you modify it to do so. Then you have to find out whether or not it has a proven update cycle so that that bugs are fixed and new features are added. If the framework is popular you can usually be confident in that it will still be supported at least a few years down the road.

The most important thing is perhaps knowing whether or not you are allowed to use the framework the way you want to. There are a wide variety of licenses that Open-Source Software is released under and depending on which license they are protected by, it can restrict their usage. This means that for example even though an Open-Source Software framework is perfect for solving a particular problem, you cannot make use of it because you want to sell the software it will be incorporated in while the license states that if you incorporate it in your own source code it must be open source as well.

Joly avoids many of the potential licensing problems because it is not a commercial software, it is used in-house. The only other minor issue with Joly's use of open source frameworks is that because the frameworks it uses are popular and receive frequent updates and because Joly does not receive frequent updates itself, it misses out on the benefits it would get by having the updated versions of those frameworks. This is however not a big problem as long as the releases it uses are stable as we have not seen any problems related to the frameworks so far except some possible memory leaks, which updating them may or may not correct.

9.4 Lessons Learned

During the course of this master thesis some valuable lessons were learned when it comes to software maintenance. First of all is that it is not an easy thing to do when you as a developer are new the software system. The most important thing is perhaps to ensure that you have a good understanding of what you have at your disposal. Does the system have any documentation that can read? Do you have the source code? Are there any particular bugs you are looking for? Can you understand the source code? Can you run it? Does it have a revision history or use any version control system? These are just some of the questions that you should ask yourself and be able to answer before you start to dvelve deeper into the issues.

The second thing is the effort required to perform maintenance. Getting to grips with the system, learning about the frameworks it might use, understanding the source code and looking through documentation, all take time. Skipping any of them will mean spending more time doing them later. Once you have a firm grasp on them it will be much easier to make a plan for what you are going to do and predicting where you might face problems. If you have a good overview of the problem it is important that you have clearly defined goals and objectives that are reachable for your maintenance work as these will help motivate you.

Joly is no more than five to six years old and yet we had some considerable problems of which most are consequence of neglect. The importance of backup and version control should not be underestimated. It is possible to say that if Joly had been under any form of version control at an early stage our work would be of a totally different kind. The fact that the various source code versions for Joly were spread between the different developers, subversion repositories and on the server on which it was deployed is a typical case of what can happen when software projects are not handled in a well planned manner. This is of course easy to say in hindsight but never the less the point stands and one of the most important contributions our work has made for Joly is that there is now a working version, that has no significant problems, under version control. This goes a long way in stating how important it is to manage your source code properly.

Although Joly may not be considered a big software system in a commercial setting, we have spent a fair amount of time on it. In a commercial setting time is money and it is easy to see why so much of it is spent on maintaining software each year.

9.5 Future Work On Joly

Although Joly is now up and running and can once more be used with the diff based algorithm, there are still a fair amount of improvements that can be made. Joly still further testing to ensure that it is completely stable and that there are no more serious bugs. Another point is that there is no security functionality implemented, passwords and other information is transferred in plain text which means that Joly is particularly vulnerable to 'man in the middle' type of attacks. Information transferred between the client side and the server should be encrypted to avoid this. There are also other minor improvements that can be made, like those that we found in comments in the source code, most of which address the correctness of specific methods.

One possible improvement is to implement a different delivery solution for the students. In its present state Joly does not feel like a complete system from the student point of view, they only

have one option and that is to go through a delivery sequence. There is no login required and they do not have the possibility to see if they have passed a given assignment through Joly, two functionalities that Devilry has. Implementing them would essentially bring Joly on par with Devilry as far as the students are concerned. The login part of the system should be based on the same type that other university systems use. This means that users log on with the same password they use to logon to the terminals at the university or their university email accounts. Joly will then no longer need to generate passwords for users and it means that users no longer need to keep track of an extra password specifically for Joly.

For teaching assistants one of the main features of Devilry is that they can download all the student deliveries they need to correct as a single packaged file while with Joly they have to retrieve the deliveries one by one. This is another feature that should be looked at and a similar thing should be implemented. Another feature that would provide ease of use is to have the possibility to get a list of all the solutions that are suspected of plagiarism for a given assignment. It would also be interesting if certain statistics could be provided to both students, teaching assistants and course administrators, like how many students that delivered and passed or failed a certain assignment or how many solutions that were suspected of plagiarism.

Despite its advantages, Devilry is lacking an important feature compared to Joly, and that is the presence of a plagiarism detection algorithm. If the above mentioned improvements are made to Joly it is likely that it would offer a better solution for assignment deliveries, at least based on features. What is important then is the usability of the features and this means streamlining the user interface. A possible way of doing this is to make the interface more engaging by using icons or tabs instead of pure text when navigating the system.

9.6 Final Words

This thesis has had a few significant changes of direction through its course. When first started, it was intended that we would be making greater improvements to Joly that what we had time to do. This can to a large degree be attributed to poor project management. Some of it is down to the previous developers, who have not not used proper backup or revision history, but it has always been my supervisor who has the overall responsibility for the management of Joly. If we to begin with, knew whether or not we had the newest source code or knew aproximatly what was causing the crashes, we could have been where we are now a lot quicker. On the other hand, the way ut turned out was perhaps more interesting and caused us to go beyond the standard "create a new version" task and has instead given us a different but rich experience. There is now a new Master

student working on Joly who will hopefully pick up from where we left, so Joly's future is looking a lot brighter than when we started.

Bibliography

[1] Matt Asay. *Why choose proprietary software over open source? Survey says!* Online. 2011. URL: http://news.cnet.com/8301-13505_3-9789275-16.html.

[2] Dries Buytaert. *The history of MySQL AB*. Online. 2011. URL: http://buytaert.net/the-history-of-mysql-ab.

[3] Gerardo Canfora and Aniello Cimitile. "Software Maintenance". In: (2000). URL: ftp://cs.pitt.edu/chang/handbook/02.pdf.

[4] Valve Corporation. *Steam Hardware And Software Survey: October 2011*. Online. 2011. URL: http://store.steampowered.com/hwsurvey/cpus/.

[5] *Devilry Home*. Online. 2011. URL: http://devilry.github.com/.

[6] Alan Dix. *The Ariane 5 Launcher Failure*. URL: www.comp.lancs.ac.uk/~dixa/../ariane.pdfLignende.

[7] Django Software Foundation. *Django | The Web framework for perfectionists with deadlines*. URL: https://www.djangoproject.com/.

[8] Free Software Foundation. *GNU General Public License*. Online. 2011. URL: http://www.gnu.org/licenses/gpl.html.

[9] Free Software Foundation. *GNU Lesser General Public License*. Online. 2011. URL: http://www.gnu.org/licenses/lgpl.html.

[10] Free Software Foundation. *What is Copyleft?* Online. 2011. URL: http://www.gnu.org/copyleft/.

[11] Free Software Foundation. *Why should I use the GNU GPL rather than other free software licenses?* Online. 2011. URL: http://www.gnu.org/licenses/gpl-faq.html#WhyUseGPL.

[12] The Apache Software Foundation. *Apache License*. Online. 2011. URL: http://www.apache.org/licenses/LICENSE-2.0.html.

[13] The Apache Software Foundation. *Apache Tomcat*. Online. 2011. URL: http://tomcat.apache.org/whichversion.html.

[14] The Apache Software Foundation. *Tapestry 5 Introduction*. Online. 2011. URL: http://tapestry.apache.org/introduction.html.

[15] The Eclipse Foundation. *Eclipse Public License - v 1.0*. Online. 2011. URL: http://www.eclipse.org/legal/epl-v10.html.

[16] The Eclipse Foundation. *Jetty/Howto/Deploy Web Applications*. Online. 2011. URL: http://wiki.eclipse.org/Jetty/Howto/Deploy_Web_Applications#Webapps_Deployment.

[17] The Eclipse Foundation. *Jetty/Howto/Write Jetty Handler*. Online. 2011. URL: http://wiki.eclipse.org/Jetty/Howto/Write_Jetty_Handler.

[18] Brian Goetz. *Java theory and practice: Managing volatility*. URL: %http://www.ibm.com/developerworks/java/library/j-jtp06197/index.html.

[19] Mart Hall and Larry Brown. *Core Servlets and JavaServer Pages*. Prentice Hall, 2004.

[20] VMware Inc. *Virtualization Basics*. Online. 2011. URL: http://www.vmware.com/virtualization/virtual-machine.html.

[21] Rod Johnson et al. *Introduction to Spring Framework*. Online. 2011. URL: http://static.springsource.org/spring/docs/3.1.0.M1/spring-framework-reference/html/overview.html.

[22] Christian K Kielland. "Metoder for likhetsvurdering av innleverte obligatoriske oppgaver i Java". MA thesis. University of Oslo, 2006.

[23] Gavin King et al. *Hibernate Getting Started Guide*. Online. 2011. URL: http://docs.jboss.org/hibernate/core/3.6/quickstart/en-US/html/.

[24] Gavin King et al. *Hibernate Reference Documentation*. Online. 2004. URL: http://docs.jboss.org/hibernate/stable/core/manual/en-US/html/architecture.html.

[25] Jussi Koskinen. *Software Maintenance Costs*. 2010. URL: http://users.jyu.fi/~koskinen/smcosts.htm.

[26] John Krogstie, Arthur Jahr, and Dag I.K. Sjøberg. "A longitudinal study of development and maintenance in Norway:Report from the 2003 investigation". In: *Elsevier* (2005).

[27] Nancy Leveson and Clark S. Turner. *An Investigation of the Therac-25 Accidents*. URL: http://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html.

[28] B. P. Lientz and B. E. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[29] Tim Lindholm et al. *The Java Virtual Machine*. Online. 2012. URL: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2.

[30] Tony Long. *Sept. 26, 1983: The Man Who Saved the World by Doing ... Nothing*. URL: http://www.wired.com/science/discoveries/news/2007/09/dayintech_0926.

[31] Object Mentor. *About Junit*. URL: http://www.junit.org/about.

[32] Randy Metcalfe and OSS Watch. *Top tips for selecting open source software*. Online. 2011. URL: http://www.oss-watch.ac.uk/resources/tips.xml.

[33] Sun Microsystems. *The Java Virtual Machine Instruction Set*. URL: http://java.sun.com/docs/books/jvms/second_edition/html/Instructions.doc.html.

[34] Atanas Neshkov. *DJ Java Decompiler*. URL: http://dj.navexpress.com/.

[35] United States Copyright Office. *Copyright Basics*. Online. 2011. URL: http://www.copyright.gov/circs/circ01.pdf.

[36] Oxford Dictionaries Online. *Definition of maintenance*. Online. 1990. URL: http://oxforddictionaries.com/definition/maintenance.

[37] Oracle. *Enums*. URL: http://docs.oracle.com/javase/1.5.0/docs/guide/language/enums.html.

[38] Oracle. *Inner Class Example*. URL: http://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html.

[39] Oracle. *Programming With Assertions*. URL: http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html.

[40] Ryan Paul. *Why Google chose the Apache Software License over GPLv2 for Android*. Online. 2011. URL: http://arstechnica.com/old/content/2007/11/why-google-chose-the-apache-software-license-over-gplv2.ars.

[41] SeleniumHQ. *What is Selenium?* URL: http://seleniumhq.org/.

[42] SourceForge. *Download, Develop and Public Free Open Source Software*. Online. 2011. URL: http://sourceforge.net/.

[43] SQLite. *Most Widely Deployed SQL Database*. Online. 2011. URL: http://sqlite.org/mostdeployed.html.

[44] International Organization for Standardization. *Guide to the Software Engineering Body of Knowledge*. 2001.

[45] International Organization for Standardization. *"Standard for Software Maintenance", IEEE Std. 1219-1998*. 1998.

[46] International Organization for Standardization. *"Standard Glossary of Software Engineering Terminology", IEEE Std. 610.12*. 1990.

[47] Hanne Steensen and Therese Vibekk. "DHIS and Joly: two distributed systems under development: design and technology". MA thesis. University of Oslo, 2006.

[48] MyBatis Team. *What is MyBatis*. Online. 2011. URL: http://www.mybatis.org/index.html.

[49] Prodromos Tsiavos. *Open Source Licensing*. Online. 2011. URL: http://www.uio.no/studier/emner/matnat/ifi/INF5750/h10/undervisningsmateriale/free_open_licensing.pdf.

[50] Jaikumar Vijayan. *"MasterCard, Visa others hit by DDoS attacks over WikiLeaks"*. Online. 2010. URL: http://www.computerworld.com/s/article/9200521/Update_MasterCard_Visa_others_hit_by_DDoS_attacks_over_WikiLeaks.

[51] Wikipedia. *Comparison of object-relational mapping software*. Online. 2011. URL: http://en.wikipedia.org/wiki/Comparison_of_object-relational_mapping_software.

[52] Wikipedia. *Comparison of relational database management systems*. Online. 2011. URL: http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems.

[53] Wikipedia. *Database Model*. Online. 2011. URL: http://en.wikipedia.org/wiki/Database_model#Relational_model.

[54] Wikipedia. *Hibernate (Java)*. Online. 2011. URL: http://en.wikipedia.org/wiki/Hibernate_(Java).

[55] Wikipedia. *List of JVM Languages*. URL: http://en.wikipedia.org/wiki/List_of_JVM_languages.

[56] Wikipedia. *Software framework*. Online. 2011. URL: http://en.wikipedia.org/wiki/Software_framework.

[57] Greg Wilkins. *About Jetty*. Online. 2011. URL: http://docs.codehaus.org/display/JETTY/About+Jetty.

[58] Greg Wilkins. *Jetty Powered*. Online. 2011. URL: http://docs.codehaus.org/display/JETTY/Jetty+Powered.

Appendix A

Code

A few of the code snippets were too big to include in the report in their entirety, and so they are referenced and found in this appendix.

A.1 Enum

Listing A.1: The decompiled enum type from the class JolyAdminController.

```
1
2     private static final class ActionType extends Enum
3     {
4
5         public static ActionType[] values()
6         {
7             ActionType aactiontype[];
8             int i;
9             ActionType aactiontype1[];
10            System.arraycopy(aactiontype = ENUM$VALUES, 0, aactiontype1 =
                  new ActionType[i = aactiontype.length], 0, i);
11            return aactiontype1;
12        }
13
14        public static ActionType valueOf(String s)
15        {
16            return (ActionType)Enum.valueOf(uio/ifi/joly/web/
                  JolyAdminController$ActionType, s);
17        }
18
19        public static final ActionType ADD;
```

```
20          public static final ActionType EDIT;
21          private static final ActionType ENUM$VALUES[];
22
23          static
24          {
25              ADD = new ActionType("ADD", 0);
26              EDIT = new ActionType("EDIT", 1);
27              ENUM$VALUES = (new ActionType[] {
28                  ADD, EDIT
29              });
30          }
31
32          private ActionType(String s, int i)
33          {
34              super(s, i);
35          }
36      }
```

A.2 Labels, goto and continue

Listing A.2: The decompiled insertStudentsolutions method from the JolyImpl class

```
1       public void insertStudentsolutions(ZipFile studentsolutions, int
            usertype)
2           throws Exception
3       {
4           Enumeration entries;
5           Student student;
6           Assignment assignment;
7           entries = studentsolutions.entries();
8           student = new Student();
9           assignment = null;
10          Course referenceCourse = courseDao.findCourseByCodeAndSemester("
                gammel", "H0001");
11          assignment = assignmentDao.getAssignmentByCourseAndNo(
                referenceCourse, (short)1);
12          if(usertype == 0)
13              student.setUsername("gammelinnlev");
14          else
15          if(usertype == 1)
16              student.setUsername("fasitinnlev");
17            goto _L1
18  _L3:
19          InputStream solution;
```

```
20          BufferedReader br;
21          StringBuffer sb;
22          ZipEntry entry = (ZipEntry)entries.nextElement();
23          if(entry.isDirectory() || !entry.getName().endsWith(".java"))
24              continue; /* Loop/switch isn't completed */
25          try
26          {
27              solution = studentsolutions.getInputStream(entry);
28          }
29          catch(IOException e)
30          {
31              e.printStackTrace();
32              throw new Exception(e);
33          }
34          br = new BufferedReader(new InputStreamReader(solution));
35          sb = new StringBuffer();
36          try
37          {
38              for(String line = null; (line = br.readLine()) != null;)
39                  sb.append((new StringBuilder(String.valueOf(line))).append
                        ("\n").toString());
40
41              break MISSING_BLOCK_LABEL_253;
42          }
43          catch(Exception ex)
44          {
45              ex.getMessage();
46          }
47          try
48          {
49              solution.close();
50          }
51          catch(Exception exception1) { }
52          break MISSING_BLOCK_LABEL_263;
53          Exception exception;
54          exception;
55          try
56          {
57              solution.close();
58          }
59          catch(Exception exception2) { }
60          throw exception;
61          try
62          {
```

```
63              solution.close();
64          }
65          catch(Exception exception3) { }
66          Studentsolution studentSolution = new Studentsolution(assignment);
67          studentSolution.setStudent(student);
68          studentSolution.setAssignment(assignment);
69          studentSolution.setDeliverytime(new Timestamp((new Date()).getTime()
                ));
70          studentSolution.setText(new String(sb.toString()));
71          studentsolutionDao.insertStudentsolution(studentSolution);
72          try
73          {
74              processor.ProcessProgram(studentSolution, true);
75          }
76          catch(Exception e)
77          {
78              e.printStackTrace();
79              throw new Exception(e);
80          }
81  _L1:
82          if(entries.hasMoreElements()) goto _L3; else goto _L2
83  _L2:
84      }
```

Listing A.3: The same method as listing A.2 from version 1.5

```
1   public void insertStudentsolutions(ZipFile studentsolutions, int usertype)
        throws Exception{
2
3    Enumeration entries;
4    entries = studentsolutions.entries();
5
6    Student student = new Student();
7    Assignment assignment = null;
8
9    if(usertype == 0){
10     student.setUsername("gammelinnlev");
11     assignment = this.getAssignmentByID(1);
12   }
13   else if(usertype == 1){
14     student.setUsername("fasitinnlev");
15     assignment = this.getAssignmentByID(1);
16   }
17
```

```
18  while(entries.hasMoreElements()) {
19    ZipEntry entry = (ZipEntry)entries.nextElement();
20    if(entry.isDirectory() || !entry.getName().endsWith(".java"))
21      continue;
22      InputStream solution;
23    try {
24      solution = studentsolutions.getInputStream(entry);
25      } catch (IOException e) {
26      e.printStackTrace();
27      throw new Exception(e);
28    }
29
30    BufferedReader br = new BufferedReader(new InputStreamReader(solution));
31      StringBuffer sb = new StringBuffer();
32    try{
33      String line = null;
34      while((line=br.readLine()) != null){
35        sb.append(line+"\n");
36      }
37    }
38    catch(Exception ex){
39      ex.getMessage();
40    }
41    finally{
42      try{solution.close();}
43      catch(Exception ex){}
44    }
45
46      Studentsolution studentSolution = new Studentsolution(assignment);
47    studentSolution.setStudent(student);
48    studentSolution.setAssignment(assignment);
49    studentSolution.setLanguagetype(assignment.getLanguagetype());
50    studentSolution.setDeliverytime(new Timestamp(new java.util.Date().
          getTime()));
51    studentSolution.setText(new String(sb.toString()));
52    studentSolution.setCompression(compressFile(studentSolution.getText()));
53
54    //Save to db
55    this.studentsolutionDao.insertStudentsolution(studentSolution);
56
57    try{
58      //CALL ALGORITHM
59      processor.ProcessProgram(studentSolution.getStudentsolutionID(), true)
          ;
```

```
60      }catch(Exception e){
61        e.printStackTrace();
62        throw new Exception(e);
63      }
64    }
65  }
```

A.3 JVM instructions

Listing A.4: The full version of the code snippet in listing 5.17

```
1           metadata[kl + 1];
2           1;
3           JVM INSTR dup2 ;
4           JVM INSTR aaload ;
5           JVM INSTR new #93  <Class StringBuilder>;
6           JVM INSTR dup_x1 ;
7           JVM INSTR swap ;
8           String.valueOf();
9           StringBuilder();
10          met_p.name;
11          append();
12          " ";
13          append();
14          met_p.begLineNr2;
15          append();
16          ",";
17          append();
18          (prepdLine - met_p.begLineNr2) + 1;
19          append();
20          " ";
21          append();
22          met_p.nOfChars;
23          append();
24          " (";
25          append();
26          met_p.begLineNrOrg;
27          append();
28          "-";
29          append();
30          met_p.endLineNrOrg;
31          append();
32          ")  ";
33          append();
```

```
34          toString();
35          JVM INSTR aastore ;
```