Matthias Huber

Master Thesis

Signing, validation and comparison of experimental data within iLAP

Institute for Genomics and Bioinformatics,
Graz University of Technology
Petersgasse 14, 8010 Graz, Austria
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski

Supervisor:
Dipl.-Ing. Dr.techn. Gernot Stocker

Evaluator:
Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski

Graz, April 2010

Acknowledgments

First i would like to express my gratitude to my family, especially to my mom. Without your support and love my studies and this thesis would have never been possible to realize.

Many thanks to my girlfriend Karin for giving me motivation and words of encouragement to finish the work.
Also to my friends for the great time during my studies.

During the entire work Dipl.-Ing. Dr.techn. Gernot Stocker gave me continuous and professional support. I want to thank you for your patience and motivation.

Last but not least special thanks to Univ.-Prof. Dipl.-Ing. Dr.techn. Zlatko Trajanoski for giving me the opportunity to write this thesis on the Institute of Genomics and Bioinformatics.

Thanks a lot!

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ...

(date) (signature)

Abstract

English

iLAP (Laboratory data management, Analysis and Protocol development) is a web-based application which covers the data management of microscopy experiments. Hereby iLAP takes care of experiment protocols and offers basic functionality of a digital labbook.
Without using a system like iLAP protocols of experiments are written by hand and have to be signed by the experimenter to guarantee its authenticity. In analogy to conventional manual signatures the concept of digital signatures was adopted for iLAP. Therefore adequate cryptographic methods for signing and validating binary data were used. With the help of these methods signed experimental data can be checked for unauthorized manipulation and consistency. Additionally a keystore concept was implemented, that enables the secure persistence of asymmetric key pairs and provides the unambiguously determination of the signers identity.
In order to detect differences between protocols and to optimize results of experiments it is essential to compare them. Therefore iLAP was extended to visualize protocol comparison based on an already existing algorithm. This comparison allows the experimenter on the one hand to detect illegal changes of protocols, on the other hand the optimization of experimental results.
Further on the usability of iLAP was improved and essential extensions were integrated in the area of file preview and experimental rating of data.

Keywords: Microscopy, iLAP, Digital signature, Cryptographic methods

German

iLAP (Laboratory data management, Analysis and Protocol development) ist eine webbasierte Applikation zur Verwaltung von Mikroskopie Daten. Dabei kümmert sich die Applikation um die digitale Erfassung von Protokollen und bietet die Funktionalität eines digitalen Laborjournals.
Bei Experimenten, die nicht auf digitale Laborjournale zurückgreifen, werden diese Protokolle manuell verfasst. Um die Echtheit der Dokumente zu garantieren, müssen abgeschlossene Protokolle zusätzlich vom Experimentator unterzeichnet werden. Analog zur herkömmlichen Unterschrift wird dieses Konzept in iLAP in Form einer digitalen Unterschrift übernommen. Um dies zu erreichen, wurden geeignete kryptografische Methoden zum Signieren von Dateien verwendet. Dadurch können nun alle aus Experimenten gewonnenen Daten signiert und im Nachhinein deren Integrität und Konsistenz überprüft werden. Zusätzlich wurde eine geeignete Datenbankstruktur implementiert, die es erlaubt asymmetrische Schlüsselpaare für die Erzeugung der Signatur sicher abzulegen und dabei auch noch die eindeutige Identität des Unterzeichners zu garantieren.
Um Unterschiede in ähnlichen Protokollen ermitteln zu können ist das Vergleichen von Protokollen unabdingbar. Daher wurde iLAP um diese Funktionalität erweitert. Sie erlaubt dem Experimentator zum einen das Optimieren von Protokollen und zum anderen die Visualisierung von unerlaubten Manipulationen bei signierten Protokollen. Die daraus resultierenden Unterschiede werden über das Webinterface interaktiv und anschaulich dargestellt.
Zuletzt wurde auch noch die Benutzerfreundlichkeit des bestehenden Userinterfaces verbessert und dafür notwendige Erweiterungen durchgeführt. Diese betreffen sowohl unterschiedliche Varianten der Bildvorschau von Dateien, als auch die visuelle Bewertung von Dateien und Experimenten.

Stichwörter: Mikroskopie, iLAP, Digitale Unterschrift, Kryptografische Methoden

Contents

1 Introduction **7**
1.1 iLAP . . . 7
1.2 Objectives . . . 9
1.2.1 Usability improvements . . . 9
1.2.2 Protocol comparison . . . 10
1.2.3 Cryptographic infrastructure . . . 10

2 Methods **11**
2.1 JEE - Java Enterprise Edition . . . 11
2.2 Technologies . . . 13
2.2.1 Spring framework . . . 13
2.2.2 Hibernate . . . 14
2.2.3 Cryptography . . . 15
JCA/JCE framework . . . 17
2.2.4 Tapestry . . . 19
2.2.5 UML . . . 20
2.3 Tools . . . 21
2.3.1 Maven . . . 21
2.3.2 Eclipse . . . 22
2.3.3 MagicDraw . . . 22
2.3.4 AndroMDA . . . 22
Model Driven Architecture . . . 22
Cartridges . . . 23
Hibernate cartridge . . . 23
Spring cartridge . . . 24
2.3.5 pgAdmin . . . 25
2.4 Design and implementation . . . 25
2.4.1 Usability improvements . . . 25
Image preview . . . 25
Edit box for description . . . 28
Deletion . . . 29

RatingBox . . . 30
2.4.2 Protocol comparison . . . 32
2.4.3 Cryptographic infrastructure . . . 34
Data structures and services . . . 35
Experiment signature . . . 35
Signer key . . . 35
SignKey management service . . . 36
Experiment signature management service . . . 37
Crypto management service . . . 37
Graphical user interface . . . 43

3 Results **45**
3.1 Cryptographic infrastructure . . . 45
3.2 Protocol comparison . . . 47
3.3 Usability improvements . . . 48

4 Discussion and Outlook **52**
4.1 Discussion . . . 52
4.2 Outlook . . . 53
4.2.1 Usability improvements . . . 53
4.2.2 Protocol comparison . . . 53
4.2.3 Cryptographic infrastructure . . . 53

Glossary **55**

List Of Figures **57**

Chapter 1

Introduction

Microscopy experiments are usually conducted by following well established protocols and are manually documented in paper-based notebooks. Sometimes these experiments follow so called *SOPs (Standard Operating Procedure)* which describe frequently performed experiments as sequences of protocol steps.
Starting from these observations the need for a digital labbook was recognized to manage and organize the accumulated data in a workflow driven manner conserving its experimental context. iLAP [36] (Laboratory data management, Analysis and Protocol development) attempts to combine experimental protocol development, wizard-based data-acquisition and its analysis into one web-based software system. The intention behind the development of iLAP is to replace the handwritten notebook with its digital counterpart.

1.1 iLAP

iLAP is a data management system formed by a three tier architecture featuring a web frontend and a backend constructed by JEE (Java Enterprise Edition) components. The webinterface offers users platform independent access to their data. To perform a workflow driven microscopy experiment with iLAP the following data structures are introduced:

- **Projects:** The user begins with the creation and definition of a project. This project is defined by a textual description, including notes and files from literature research, microscopes, etc.
- **Subprojects:** Since subprojects represent child instances of the type *Project*, a complete data hierarchy can be established.

- **Experiments:** An experiment consists of a textual description of file attachments and of an underlying protocol. The protocol itself represents a sorted collection of so called *CWP steps (Current Working Protocol Steps)*. Each step describes a single procedure of the experimental workflow and can hold notes and attached files.

- **Analysis:** All accumulated data from a project can be subject of further analysis. Such an analysis has the aim to extract meaningful information out of the accumulated raw data. To store such analytical processes in a data structure the *Analysis* was introduced. iLAP offers several server-side analysis modules, but also supports external modules by exporting data sets and importing already processed ones.

Figure 1.1 demonstrates the hierarchical relations of a well defined project in iLAP.

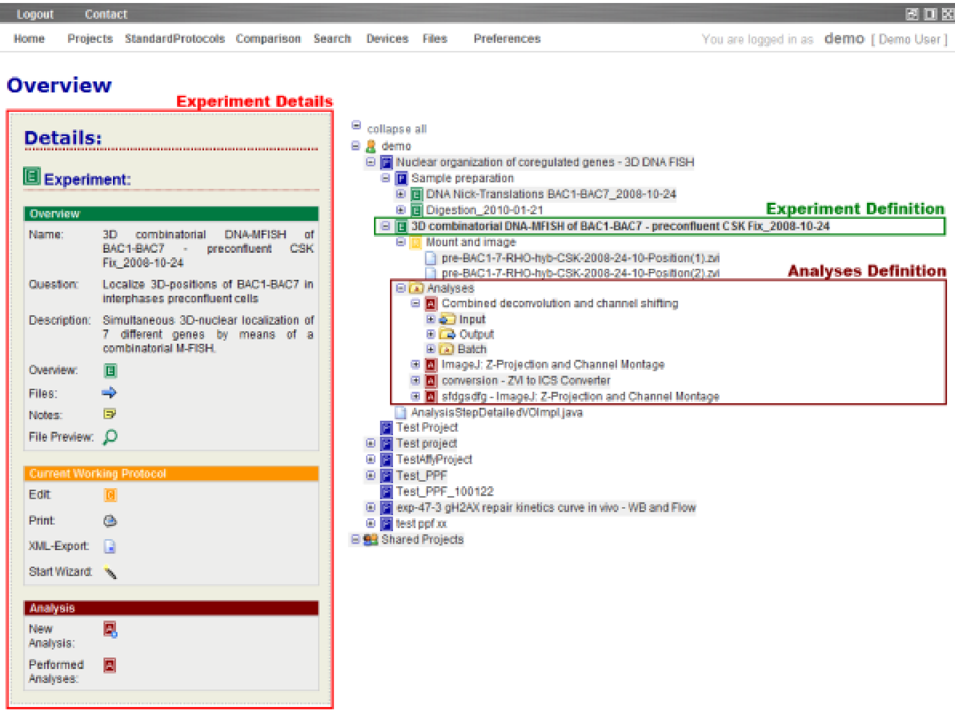


Figure 1.1: Overview of the intended data hierarchy in iLAP.

As mentioned before the iLAP application was implemented in a fashion to cover the complete laboratory workflow. This workflow is composed of several phases and is described in more detail under [36]:

Project definition phase: A laboratory project is based upon specific biological questions. In this phase of the workflow this question and a hypothesis that addresses it has to be defined. During the scientific research several documents are read to find methods for solving the problem. iLAP encapsulates this phase in its *Project* structure as described before.

Experimental design and data acquisition: Based upon the hypothesis of the project definition and several SOPs from previously performed experiments a specific experimental proceeding is declared. This proceeding consists of smaller steps represented by iLAPs CWP steps. By executing the established workflow several observations are accumulated to enhance the better understanding of the results.

Data analysis and processing: To retrieve meaningful information from the accumulated raw data analysis and post-processing steps are required. In order to conclude this phase it is important to relate generated files to the according raw data. Therefore iLAP offers several methods to analyze the accumulated raw data and to store it in a project oriented way.

Data retrieval: This experimental workflow leads to a chronological organized data accumulation consisting of e.g. protocols, notes and file attachments. By reflecting on the recorded experiment data scientists can obtain different viewpoints which lead to new interesting findings. This is supported by a search module which abbreviates the data retrieval for the researchers.

1.2 Objectives

This thesis can be divided into three tasks. The aim of the first task was to get a better understanding about the application structure and the used technologies. In a second phase of this work a method to compare experiment protocols was implemented. Finally the main focus of this thesis was to introduce a new service that offers the functionality of signing and validating of experimental data.

1.2.1 Usability improvements

Preview Overview Because users in a microscopy lab have to handle a big amount of raw files, the need for a preview overview of attached

files arises. For this purpose a tabular representation of all attached data should be implemented and for further inspection a bigger preview should be introduced. All file attachments in iLAP are stored as so called *dataobjects*. From here on such attachments are defined as dataobjects. The color design of the overview has to consider the type of parental container to which the objects are attached to.

Deletion of dataobjects Because of special user requirements at the beginning it was not possible to delete dataobjects and their parental containers like projects, experiments, etc. The intention behind this design decision was to avoid unwanted deletion of precious data. But the praxis shows that it is absolutely necessary to have this functionality in iLAP.

Rating Also a rating for dataobjects should be provided, which supports the user to highlight relevant data. This feature was additionally extended for experiments.

1.2.2 Protocol comparison

The second goal of the work was to introduce a method that allows the comparison of protocols and the visualization of the resulting differences in an intuitive manner. Hereby a protocol comparison algorithm described in [32] should be used to detect differences between two protocols.

1.2.3 Cryptographic infrastructure

Sometimes microscopy experiments lead to new discoveries and experimenters have to guarantee that their protocols and data was generated at a certain time point and not altered afterwards. For such a case it should be possible to digitally sign an experiment in order to conserve the status of an experiment. This must be realized in a manner that later on third parties can verify the signed status of the researchers experiment.
Additionally it should be possible that a user with higher responsibilities can countersign the signed experiment. Hereby this privileged user confirms the consistency of the underlying experimental data. The signing and countersigning process must be realized using asymmetric encryption methods, so that the signer and countersigner can be uniquely identified.

Chapter 2

Methods

2.1 JEE - Java Enterprise Edition

Stark describes in his book [35] JEE as a set of specifications and technologies, which enhances the development of secure, stable and portable business applications. These specifications can be realized by implementations of different providers. Besides a multitude of commercial projects also open source projects implement these specifications. The main goal of this collection of specifications is the separation of the application in different components. Each layer fulfills different needs, has different and responsibilities and communicates with other layers. This architecture brings several benefits like better maintainability, testability, stability or security. By extending or changing functionality of components, layers or tiers, only few changes have to be applied to the other parts because of the separation. This implies also that these tiers can be executed on different machines or JVMs.
Johnson introduces in his book [28] the separation of web applications in a three layer architecture. This architecture is illustrated in the figure 2.1.

- **User Interface (UI) Tier:** This tier is also called the *Presentation Layer* and can be divided into a client and a web tier. In such cases the client tier consists of the users browser and the web tier of the user interface. In iLAP the presentation layer is created by Tapestry, discussed later in this work. The main access point to the services is a servlet, which allows the user interacting with the application. For this purpose it handles user requests and returns results as HTML pages.

- **Middle tier:** The Middle Tier or business layer in case of the iLAP application is based on the Spring framework. It contains some business logic which is responsible for the processing of requests initiated by the client. This is achieved by so called Spring Beans.

- **Persistence Tier:** This tier was introduced to hide the persisted data from the business logic. It considers transaction management and data access security. Behind this layer a relational database management system takes care of the permanent storage of data. In iLAP it is a PostgreSQL [16] database.

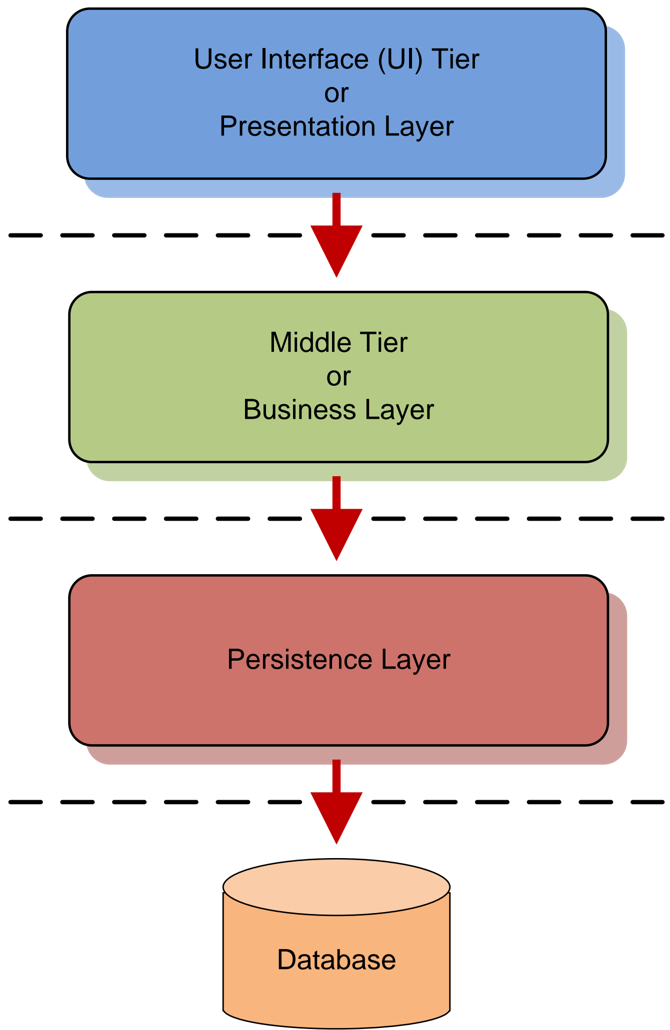


Figure 2.1: Representation of three tier architecture.

2.2 Technologies

2.2.1 Spring framework

The Spring framework is an open source application framework, that addresses major aspects of Java EE application development. Its architecture is predominantly defined by interfaces similar as the Java EE, but unlike the Java EE it has a POJO (Plain Old Java Object) based programming model. One of the main features of this framework is the application of the *Inversion of Control (IoC)* pattern. The integration of the Spring framework in high level applications brings the following benefits [17]:

- sparse dependencies to the container API (non-invasive)
- easy testability by using simple POJO-Beans for JUnit testing
- high scalability
- enables distributed applications
- distributed transaction management
- can be combined with different technologies as for example EJB and Hibernate
- can be used in combination with different application servers (Tomcat, Jetty, JBoss, ...)

Spring consists of different components and modules. Each of these offers a scope of specific functionality. The architectural design of Spring and its modules is illustrated in figure 2.2. These modules are working independently from each other. The developer can select the module that can be considered as best choice for a problem. In the case a developer wants to integrate just the Spring DAO module in its application there is no need to deploy the entire Spring framework. The only requirement for this scenario is the inclusion of the DAO module along with the Spring IoC Container, which represents the heart of the application framework.
The Spring modules have the following responsibilities and functionality:

Spring Core represents the core of the entire Spring framework. All other components of the framework are dependent on the core module. The core provides the support for dependency injection (DI). The central class of the core module is the *BeanFactory* a basic implementation of the IoC container. It consists of a central XML configuration file for

instantiating, configuring and managing beans. By using the BeanFactory it is possible to separate the configuration of dependencies emerged by beans from the application code.

Spring AOP In general the main goal of *AOP (Aspect Oriented Programming)* systems is to separate components and concerns. In AOP a component represents a system property encapsulated in a procedure and an aspect or concern represents a system property that can not be encapsulated in a single procedure. Examples for aspects are transaction safety of persistence, security or logging procedures. The Spring AOP brings Aspect Oriented Programming into the application framework with featured transaction management, logging, etc.

Spring DAO brings its own Data Access Object layer. This layer offers the developer a standardized communication to different access technologies and databases. Typically for connecting and performing operations on relational databases, the programmer has to write repeatedly glue code for retrieving a connection, binding SQL parameters, releasing the database resources and exception handling. The Spring DAO handles all these tasks and hides them from the developer.

Spring ORM builds the object/relational mapping layer. This module has not its own implementation of the object/relational mapping but instead offers support for multiple existing persistence solutions like Hibernate, iBatis, etc.

JEE offers a basis for simplifying the interaction between different JEE technologies like EJB.

Web MVC contains the scope of functionality to build flexible web applications. This module is based upon the *Model View Control (MVC)* pattern and supports the integration of many known frameworks like Velocity, JSP, Adobe Flex, etc.

The Spring framework can be downloaded at `www.springsource.org`. In this work only a short introduction to the Spring framework is given. For further reading the books [26] [31] [22] are a good choice.

2.2.2 Hibernate

Hibernate is an open source framework for *Object Relational Mapping (ORM)*. The Hibernate framework takes over the responsibility to persist objects into database tables. In the persisting procedure also inheritance of objects or

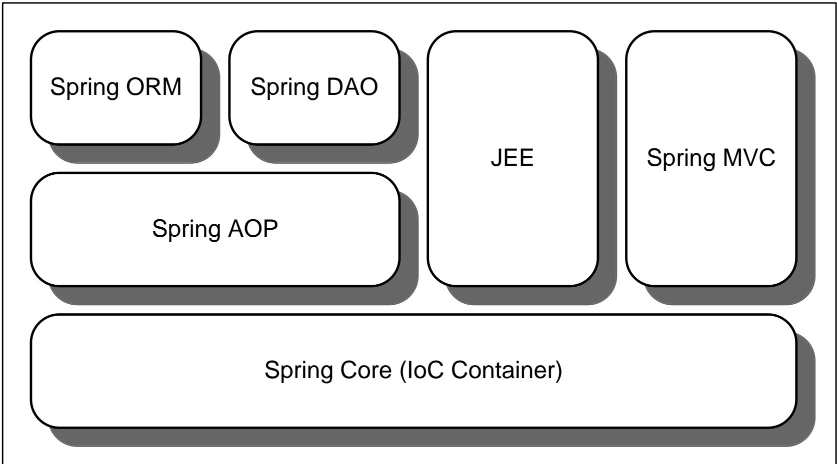


Figure 2.2: Representation of the modular architecture within Spring framework.

relations between objects are taken into account.
Hibernate provides a mechanism to transparently persist Java objects also called Plain Old Java Objects (POJOs) in an automatic fashion. To access a database with Hibernate there is no need to specify explicitly the SQL statements. This does the framework. To provide a broad usage of the framework Hibernate supports many common database implementations by generating the statements depending on the used SQL-dialect of the corresponding database. This makes it also easy to exchange the database if necessary.
The mapping of the POJOs to the database tables is defined in a XML file. Hibernate supports references as *one to one*, *one to many* and *many to many*, but also inheritance and reflective relations. By accessing objects it is possible to load related objects immediately or to load them at the time they were needed. The former procedure is referred as *eager loading* and the ladder as *lazy loading*. Another advantage of the framework is that it can be configured in such a way that operations like saving and deletion can be cascaded to preserve the integrity.

2.2.3 Cryptography

During the signing process of experiments iLAP uses different cryptographic approaches to encrypt, decrypt and sign sensitive data. In this section main topics of cryptography are discussed.
A *symmetric algorithm* describes a method to encrypt and decrypt data.

Hereby two parties share one common secret to perform both, encryption and decryption. The shortcoming of this algorithm is that the key has to be shared for both parties.
The principle behind an *asymmetric algorithm* is totally different from the latter one. This algorithm specifies a user specific key pair. One is the *private key* and the other is the *public key*. The private key is secret and therefore only known by the holder. The public key can be distributed to other users. The combination of public and private key allows to perform cryptographic operations.
In most cases the message or data that has to be encrypted or signed is too long. Several functions exist that represents the data in a shorter and unique manner. These functions computed on the data are called *hash functions*. The hash value has two important aspects. On the one hand it represents the message uniquely, which means for a certain message there exists only one deterministic hash. On the other hand the hash generation is a one way function, because by knowing the hash value the corresponding message can not be reconstructed.
An additional essential therm in this context is the *digital or electronical signature*. A user decrypts the hash value of some data with the private key and sends the unchanged data together with the decrypted hash value to another user. The receiver now can check the integrity of this data by recalculating the hash value of the original input data and comparing it with the public-key-encrypted hash value. It is important that each signature can be assigned reliably to its signer. In public services the identification of the signer is made by a trustcenter. The trustcenter issues a certificate for this user and ensures its identity. If it is an accredited trustcenter the signature is legally binding and substitutes the physical signature.
In the case of iLAP the new cryptography module fulfills the needs of the following scenario. A user signs the current status of an experiment. This means the service must sign each attachment, the protocol, experiment notes, etc. For the signing procedure the hash value of each part of the experiment is acquired, then the hash value is decrypted with the private key of the signer and stored in a file. In a second step a user wants to check the integrity of the signature and must therefore have the possibility to verify it and to detect possible manipulations. In a third usecase a countersigner user, which possesses a higher authority role, can confirm respectively countersign the signature of the experiment signer.
Therefore each user who wants to perform an action as signing or countersigning needs an asymmetric key pair. Another constraint must be that the private key for each user have to be secured, so that only the owner can access it during the signature process.

To realize the cryptographic functionality within iLAP the JCA/JCE framework is used. It offers the necessary algorithms to perform the signing process.

JCA/JCE framework

In Java two class libraries are provided which offer cryptographic operations and methods. The two libraries are the *Java Cryptography Architecture (JCA)* and the *Java Cryptography Extension (JCE)*. The separation is based upon political restrictions.
The JCA can be found in the `java.security.*` package and a reference guide can be found here [8]. The JCA contains the following scope of operations as proposed in this document [7]:

- **MessageDigest**: is used to calculate hash values from binary data.
- **KeyPair generators**: generates a key pair.
- **SecureRandomGenerator**: originates random numbers.
- **Signature algorithms**: these algorithms create signatures.
- **CertificateFactory**: offers methods to read in and process certificates.
- **KeyStore**: for the secure storage of keys in different formats and certificates.
- **KeyFactory**: for decomposing a key pair in private and public key.
- **CertStore**: to manage certificates.

The JCE is resident in the `javax.crypto.*` package, the reference guide can be found on this web page [9]. The JCE framework provides the following functions:

- **Cipher**: offers cryptographic algorithms for encryption.
- **KeyGenerator**: for generation of keys for the aim of encryption.
- **SecretKeyFactory**: is similar to the KeyFactory in the JCA and is used to decompose a key pair.
- **KeyAgreement**: for the secure exchange of keys.

To enable a simple exchange of the standard JCA/JCE with other implementations a Service Provider Interface (SPI) is provided. By integrating such a service provider the functionality of this provider can be immediately used as can be seen in figure 2.3. A new provider can be installed in two ways. One can include a new provider in the source code by simply adding the line illustrated in the listing 2.1 or by defining it in the java.security file statically.

```
Security.addProvider(new CryptoProvider());
```

Listing 2.1: Registering a new cryptography provider.

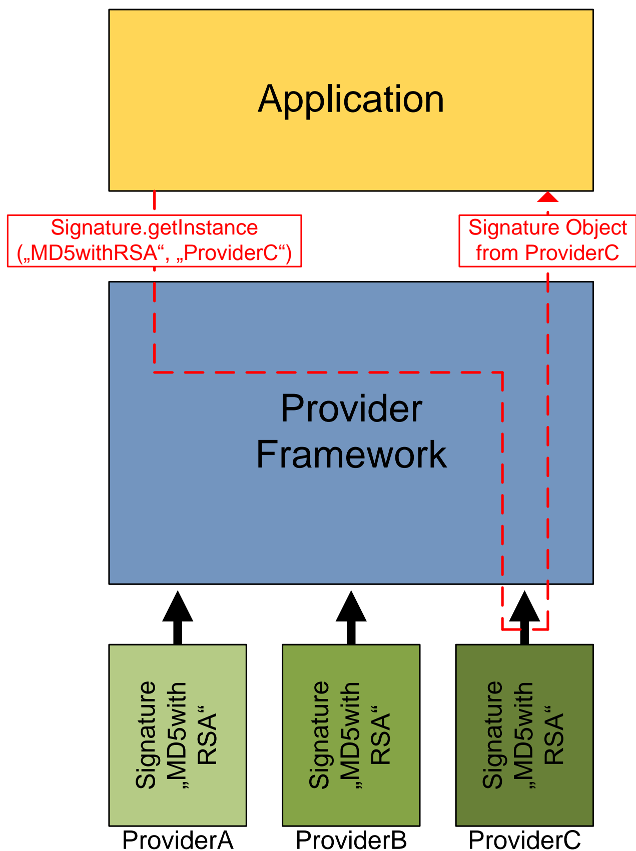


Figure 2.3: Representation of the provider framework of the JCA.

From the components discussed above in the cryptography service implemented in iLAP the KeyFactory of the JCA is used to generate a private/public key pair for each signer. Also a Cipher and the SecretKeyFactory

is used for the encryption and decryption of private keys. The last infrastructure object furnished by the JCA package is the Signature to sign each part of an experiment and in a further step to verify such a signature. The details of this implementation are discussed later on, also the provider implementations used for the realization of the cryptography service are specified. Further details about cryptography can be found in the book of Rosen [33]. It gives an overview of mathematically and theoretically background, besides the book of Hook [27] provides good practice for developing cryptographic functionality based on the JCA/JCE architecture.

2.2.4 Tapestry

Tapestry is an open source framework of the Apache Software Foundation. The framework offers the developer a tool to create dynamic and scalable web applications in Java. Tapestry is build upon the Java Servlet API and therefore applications developed with it are runnable in a multiple application servers or servlet containers. In the case of iLAP it runs on an Apache Tomcat server.
Tapestry is a component based framework, each page represents a collection of such components. Contrary to request based frameworks Tapestry focuses on placing components on pages and to react on events, which are triggered by components. For this purpose Tapestry includes several standard components for development. For a detailed list consult this web page [19]. Additionally Tapestry offers the possibility to create new components and to reuse them in different scenarios. This feature prevents the accumulation of duplicated code by reusing such generated components.
Unlike other web application developing frameworks Tapestry introduces its own perception of a web page. This page consists of three different parts a template, a Java class and a page specification file. This enables the separation of the view and the code needed for accessing, managing and modifying objects. It also offers the possibility that designers and programmers can work concurrently on the same pages by separating the view and the functionality. The page's template represents a HTML file with common HTML tags augmented by additional tags that are representing Tapestry components. These tags are marked with the keyword *jwcid (Java Web Component ID)* and are placeholders for the corresponding Tapestry component. The page specification file is a XML file which defines the depending Java class and containing component specifications. In the according Java class the developer can implement the functionality of the page.
Third party frameworks widen the possible field of Tapestry components. Such a framework called *Tacos* is also used in iLAP. It provides additional

components and applies *AJAX* behavior to Tapestry components. To get details on the functional range see here [18].
The inventor Howard M. Lewis Ship addresses in his book [34] the four main goals for the implementation and design of the Tapestry framework:

Simplicity Applications developed with Tapestry contain a small amount of code compared to other frameworks. This is because the developer has to write only code which is application specific. Glue code like extracting and interpreting parameters from the request is covered by the framework. Tapestry components are offering listener methods like in rich client applications.

Consistency Different Tapestry pages are working in the same fashion due to the fact that they are built up by the same reusable components. This consistent behavior avoids duplication of code.

Efficiency An important aim is the scalability of an application for high traffic use cases. For this reason Tapestry reads each HTML template and XML specification file once and stores it in component pools and caches. This minimizes the amount of processing and speeds up the application performance.

Feedback Many web application frameworks are reporting errors and exceptions by displaying the whole stack trace. This confronts the developer with the problem of error searching and debugging. Tapestry's architecture provides multiple layers for exception catching and reporting and ensures in that manner that as much information as possible is extracted. The error reporting in Tapestry will be shown either in the console of the server or in the web browser.

In the course of this thesis new Tapestry components have been developed. They are designed to be reusable. For the development of iLAP Tapestry version 4.1 is used. A great introduction and practical examples provides this book [37].

2.2.5 UML

The UML (Unified Modeling Language) [21] is a modeling language that provides the creation of visual representations of software systems. It is a specification of the OMG group [14]. Models assist the developer in understanding complex software applications by introducing different levels of abstraction. By using UML tools the developer can design the structure and

the behavior of a system. Also relations between introduced components can be modelled. A big advantage is that the design is decoupled from the underlying software platform.
The OMG group defines thirteen types of diagrams for the UML 2.0 specification, divided into three categories [6]:

- **Structure Diagrams** specifies static application structure by including the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram and Deployment Diagram.
- **Behavior Diagrams** include the Use Case Diagram, Activity Diagram and State Machine Diagram. These three types of diagrams describe general types of behavior.
- **Interaction Diagrams** include the Sequence Diagram, Communication Diagram, Timing Diagram and Interaction Overview Diagram. These diagrams include different aspects of interaction.

The book [25] gives a good overview and usage scenarios of UML.

2.3 Tools

2.3.1 Maven

Maven is a tool for managing the build process, reporting and documentation of a Java software project. By defining a local repository of dependent libraries needed by a software project, Maven tries to download missing dependencies from such remote repositories to enable a successful build process. It also gives the opportunity for different developers working on the same project to have the correct versions of libraries resident on the local repository.
The main goal of Maven is to support the developers comprehension of the complete state of development effort in a short period of time, as proposed here [11]. Therefore it tries to fulfill the following requirements for software projects:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best development practices
- Allowing transparent migration to new features

2.3.2 Eclipse

Eclipse is an open source *Integrated Development Environment (IDE)* for software development and it can be downloaded from [4]. It is conceived for Java software development. Due to its plugin based architecture it can be extended for additional software languages such as C, C++, etc. Beside syntax highlighting it provides the developer with code completion and source code analysis.
Additionally the commercial plugin *MyEclipse Enterprise Workbench* [13] expansion was used, which offers many useful functions for the development of JEE applications like deployment of the web applications, database access and UML/XML handling.

2.3.3 MagicDraw

MagicDraw [10] is a visual modeling tool for designing project specific data structures and relations between them. The tool is programmed in Java and therefore platform independent. MagicDraw generates code from existing UML diagrams for different programming languages as Java, C#, C++, etc. but also does the reverse process of generating visual diagrams from existing code.
Throughout this thesis it was used to extend existing data models and create new models for the iLAP application. Together with AndroMDA it forms a strong instrument for Model Driven Development.

2.3.4 AndroMDA

AndroMDA represents a tool for code generation based on the *Model Driven Architecture (MDA)* paradigm. AndroMDA transforms annotated UML model diagrams into platform specific code. Such UML models can be designed with modeling tools like MagicDraw. The models are specified in XML Metadata Interchange (XMI) format. In spite of the separation of application models and generated source code consistency is achieved. The AndroMDA framework consists of *Cartridges* and is extensible in the way that the code can be generated for several platforms. The models designed are not bound to a specific programming language.

Model Driven Architecture

As described in [12] the Model Driven Architecture introduced by the OMG [14] group is a progressive transformation of abstract software models to its implementation. It aims the separation of the application and the business

logic from the underlying platform. This is realized by splitting a software project in different abstraction levels. The behavior of an application is integrated in higher abstraction levels and details for the technical implementation are hidden in lower levels. For this reason the Model Driven Architecture introduces the following levels of abstraction described in [24]:

Computation Independent Model (CIM) This level of abstraction focuses on the perception of the user. It consists of the business model representing the application and fulfills the needs of concrete usage scenarios.

Platform Independent Model (PIM) represents the functionality and business logic by avoiding the platform specific focus. This separation aims for a design that can later be applied to different platforms.

Platform Specific Model (PSM) In contrast to the PIM layer it addresses the requirements for the platform specific code generation.

Code Model represents the platform specific code generated by some code generator as AndroMDA.

Cartridges

In AndroMDA cartridges [24] are rules also called templates for the code generation. Due to the fact that AndroMDA offers the possibility to remodel and regenerate code the cartridges are generating structures by separating manually editable code files from pure generated. The former files are stored in so called *target* folders and the ladder ones in a *source* location. After the remodeling the target folder will be overwritten.
The AndroMDA framework offers different cartridges, which can produce code for JEE platforms like EJB or Spring. For further reading about AndroMDA and its cartridges the following web resources [1] [2] can be consulted.

Hibernate cartridge As described in [24] this cartridge handles the persistence specific code. The persistence layer of the application uses Hibernate as its Object Relational Mapping framework. The cartridge processes the mapping schema and creates required Java classes and is also responsible to introduce the SQL statements for the creation and deletion of the database schema. Due to the fact that the Hibernate cartridge comes with the support of different SQL dialects the exchange of the used vendor specific database can be realized with minimal effort.

The generated persistence tier of the application can be hidden behind a service layer. This service is realized with EJBs or Spring POJOs.
The AndroMDA team [3] proposes the following stereotypes:

Entity represents a Hibernate POJO. The UML class must be annotated with this stereotype and the cartridge generates the according Java class and the mapping of the relations to dependent objects. For this purpose the multiplicities must be set on the begin and the end of the relation. This procedure ensures the consistent mapping of the generated database tables and the corresponding Java classes.

Enumeration by applying this stereotype a normal class is generated with the difference that each attribute specifies a default value. This enumeration stereotypes can be referenced also within a entity definition in its attributes.

Service can be seen as a service facade responsible for a set of entities. This stereotype represents a stateless Spring bean.

Not only the above described stereotypes can be used, but also some tagged values for primary key, foreign key, sequence or SQL query statements definition are offered. The ladder ones should be avoided to conserve an *Platform Independent Model (PIM).*

Spring cartridge Kainz [29] describes the Spring Cartridge as the generator for the business and persistence layer. As described above the persistence tier is dependent on the Hibernate cartridge so the Spring cartridge implies the usage of it. AndroMDA generates the persistence layer without any additional developer assistance and according *Data Access Objects* so called DAOs. This DAOs are based on the Spring framework and provide the access to the corresponding entities. The Spring cartridge offers the following stereotypes for definition:

Service works as the according stereotype in the Hibernate Cartridge and implements some specific business logic for depending entities.

Value Object To avoid the exposition of the plain persistence data to the presentation layer valueobjects were introduced. They represent an inherited object of the corresponding entity which gives the opportunity to hide some attributes or relations.

2.3.5 pgAdmin

pgAdmin [15] is an open source tool for administration and development for PostgreSQL databases. This application was ported to many platforms as Linux, Mac OSX or Windows. pgAdmin addresses many developer requirements from writing single SQL queries to create complex database schemas. The graphical interface makes administration easy. The application contains a SQL editor with syntax highlighting, a server side code editor, a job scheduling agent, etc.

2.4 Design and implementation

2.4.1 Usability improvements

During the experimental workflow iLAP users have to handle multiple of raw files like microscopy images, literature documents, etc., in iLAP such file attachments are called *dataobjects*. To improve the handling functionalities of iLAP some new features were introduced. The main aim of the design was the reusability of these features.

Image preview

All units that can hold and are bound to dataobjects like projects, experiments, analyses in iLAP are implementing the interface `DataObjectReceiver`. Since there was no preview modality to display all dataobjects attached to a dataobject receiver on one page, the first feature implemented takes over this functionality. Throughout the work of Kainz [29] a preview generator for dataobjects with known formats like "PDF", "PNG", "TXT" was implemented. This preview generator creates thumbnails for such known formats in an asynchronous way.
Each instance of a dataobject receiver can hold a collection of several dataobjects and so the choice for a tabular representation view was taken. Since a dataobject receiver can be a project, experiment, analysis or CWP step the user must recognize the type of receiver visually. Each view should represent the dataobjects of the corresponding receiver in a tabular view and different color and it should be given the possibility to inspect each dataobject separately by offering the user a *big preview*.
To realize these specifications Tapestry offers the class `AbstractComponent`, which generates dynamic HTML content without a defined component template and so it can render itself in any desired way. So the implemented class

in iLAP which renders the tabular view is derived from Tapestry's *Abstract-Component* and was called `DataObjectImageGrid`. The *AbstractComponent* offers also the method `renderComponent` and the `IMarkupWriter` and with these two tools it is possible to write out directly HTML tags at the time the component renders itself. There were added the following input arguments for the new component, which some of them are required and one is optional:

- `dataObjectReceiver (required))`: defines the given receiver, which can be a project, subproject, experiment, CWP step or analysis.
- `treeType (required)`: defines what type of receiver the current is. It is an integer which specifies the type. It has to be a type of `DataObjectReceiverInterface`.
- `numCols (required)`: defines how many number of columns should be used to represent the dataobjects in the tabular view. The data type is `INT`.
- `value (required)`: defines the current dataobject examined in the iteration. It is an instance of `DataObjectDetailedVO`.
- `numRows (optional)`: defines how many rows are used for the presentation in the table. The data type is `INT`.

To get the collection of dataobjects of the specified receiver the component needs a dependent method of the `DataObjectManagement` which resides in the business layer. The Tapestry framework has the ability to explicitly inject the Spring Service *DataObjectManagement* in the *DataObjectImageGrid* component with all its dependencies. After the component has fetched the collection of dataobjects it constructs the image grid depending on the number of columns specified. To differentiate in which hierarchical level - project, experiment or analysis - the user resides a color scheme was selected. The different color styles for each receiver type and the representation of one dataobject in the image grid should look like in the figure 2.4. The new introduced component `DataObjectImageGrid` can be instantiated in any Tapestry HTML template by typing a single tag, this is shown in the listing 2.2. What can be seen is that the body of the tag can be filled with other HTML tags that are rendered within the body of the component. This gives the possibility to define the look of the component in any desired way. Through the `value` argument the current examined dataobject in the rendering loop can be accessed. The `jwcid` expression in the component instantiation defines the *Java Web Component id*. It means that the enclosed specification defines a Tapestry component named `DataObjectImageGrid`. The naming

ognl stands for *Object Graph Navigation Language* and is a special encapsulation of code and template variables. To resolve the specified method call after the ognl expression, Tapestry looks in the object derived from the page class specification. This mechanism of resolving a method call is known as *reflection*.

```
<span jwcid="@DataObjectImageGrid"
        dataObjectReceiver="ognl:getCurrentReceiver()"
        treeType="ognl:getTreeType()"
        numCols="ognl:getNumCols()"
        value="ognl:getCurrentDataObject()">
                // within the body other functionality can be
                // inserted for the current dataobject
</span>
```

Listing 2.2: Shows the tag to instantiate the ***DataObjectImageGrid*** *in the HTML template of a Tapestry page.*

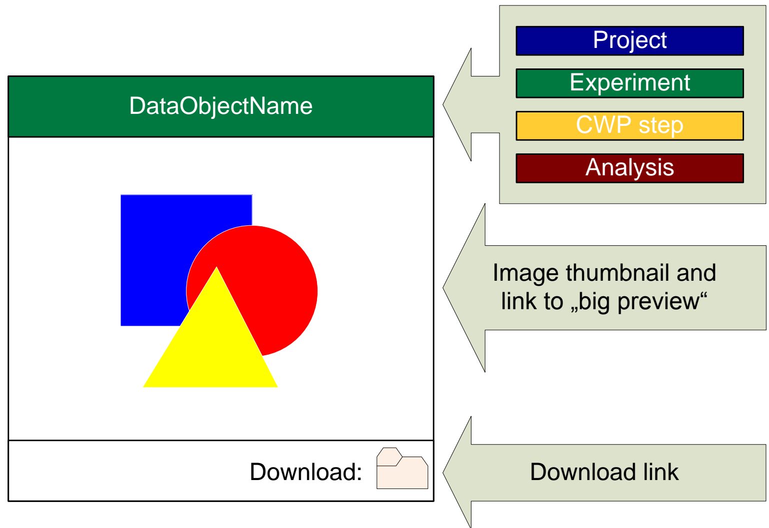


Figure 2.4: Dataobject representation.

Another requirement the image grid component should fulfill was a link to a bigger preview for further inspection. This link will only generated if for the current dataobject a preview exists, otherwise a placeholder picture should be displayed with the information that there is no preview. This preview link is encapsulated in an additional component which provides the bigger preview in a new browser window. Also the content of the opened window is styled

in a manner to differentiate the dataobject receiver which the dataobject is attached to. The big preview reveals the picture of the dataobject in a higher resolution. Also included in the new view are a download link and a close button. The component is named `StartPreviewLink` and like the `DataObjectImageGrid` it has the following input arguments to define its behavior:

- `dataObject (required)`: specifies the dataobject for the preview and is an instance of `DataObjectDetailedVO`.
- `treeType (required)`: specifies the type of receiver the `dataobject` is attached to. It is an integer which specifies the type.
- `spanImage (optional)`: the argument is boolean and has default value `TRUE`. If set to false the preview link is inactive.

By inserting the following tag 2.3, the preview link component can be inserted in any desired Tapestry HTML template . The only difference is that here it is not possible to define the body of the tag.

```
<span jwcid="@StartPreviewLink"
        dataObject="ognl:getCurrentdataObject()"
        treeType="ognl:getTreeType()"/>
```

Listing 2.3: Shows the tag to import the `StartPreviewLink` component in the HTML template of a Tapestry page.

Edit box for description

The `DataObjectDetailView` is an almost existing Tapestry component for iLAP. It shows some general information about the dataobject, that the user is inspecting at the moment. This component should be extended by a component that enables the changing of the dataobject description, which is a member variable of the `DataObject` entity. The new Tapestry component with the name `DataObjectEditBox` was conceived, that in a further stage other member variables of the entity `DataObject` could be changed too, for this purpose the following input arguments were defined:

- `dataObjectId (required)`: specifies the id of the dataObject which is unique. The input must be type of `Long`.
- `type (required)`: represents a `String` that defines which property of the entity would be changed.

The specified component is constructed via a `AjaxDirectLink` provided by the Tacos framework. This ensures that after submitting the form only the region of the edit box will be refreshed, not the complete page. By clicking on the edit link the state of the component gets into editing mode. The component changes its behavior and an input text field appears, in which the user can type the desired description for the current dataobject. In the `DataObjectManagement` a new method was implemented which changes the description of the dataobject. After the user submits the edit form, this update method will be called with the new description.
The `DataObjectEditBox` could be placed on any desired location of a Tapestry page by inserting the code tag listed in here 2.4.

```
<span jwcid="@DataObjectEditBox"
        dataObjectId="ognl:getDataObjectId()"
        type="literal:description"/>
```

Listing 2.4: Tag for importing the `DataObjectEditBox`.

Deletion

As described in the introduction initially it was not possible to delete dataobjects. But the praxis proved that such a feature is essential, but should be only accessible by privileged users.
Each project, subproject, experiment, CWP step or analysis is marked as not deletable if a dataobject is attached. This implies also that - if a dataobject is attached - for example to an experiment, also the parent projects are marked as "not deletable". This means after every deletion of a dataobject all parents and all first level children of the current `DataObjectReceiver` must be checked recursively for possible attachments. If there is no dataobject attached any more the dataobject receiver must be set as deletable. Additionally after a dataobject is attached to a receiver, it can be linked by other receivers in the hierarchy. So each dataobject has to be checked for relations to other receivers before it can be removed physically from the server storage. These considerations are taken into account during the implementation.
For the implementation in a first stage a new user role `ROLE_FILE_MANAGER` is defined. This role is stored in the user management and prevents that unauthorized users delete data. In iLAP for each type of receiver (like project, experiment, ...) a type-specific implementation of dataobject management methods exists. The methods to detach the relation from the underlying `DataObjectReceiver` is individual for each type, but the task to delete the

dataobject from storage is critical for all possible `DataObjectReceiverInterface` implementing classes. The `DataObjectManagement` service class is a superclass of all classes implementing the `DataObjectReceiverInterface`. Therefore all methods that are non-specific are implemented in the superclass and the type specific code goes in the corresponding service routines. This separation prevents the accumulation of glue code. The inspection for the deletion status of each receiver, which is computed recursively throughout the hierarchy and updated after every deletion action was added in all service classes of the receivers. The physical deletion depending on the state is realized in the superclass.
Another aim was to check for generated preview thumbnails and existing postprocessor states. These are generated from the postprocessing services implemented by Kainz and described in [29]. In the case of physical cleanup of the dataobject the thumbnails and postprocessor states must be deleted as well. This method is identical for all possible receivers and therefore added in the superclass.
The deletion process of the dataobject of a `DataObjectReceiver` - for example an experiment - follows the current workflow. The user clicks on the delete button and after the confirmation field is checked the deletion process is started centrally in the `DataObjectManagement`. There the process is assigned to the corresponding dataobject management service depending on the receiver type, for example in the case of an experiment the delete method in the `ExperimentDataObjectManagement` is called. Afterwards the relation from the receiver entity to the underlying dataobject is removed and the check for beeing deletable is performed. This is done by checking if all the underlying childs as CWP steps and analyses are also deletable. If it is so the experiment is marked as deletable and also recursively all parents are set to this state. Further on it will be checked if the dataobject has relations to other entities. If there are none left, all post processor states attached to this dataobjects are removed and in a last step all files are removed on the storage. These two methods are implemented in the superclass, the `DataObjectManagement`.

RatingBox

Since it was not possible to evaluate dataobjects by rating them, the need for such a modality arised. To give the iLAP user a possibility to mark dataobjects as relevant or not a rating box is introduced. During the implementation the rating functionality has proven useful and so it was applied also to experiments.
Therefore the dataobject entity had to be extended by adding an integer

value named objectRating. This variable holds values between "1" and "5". A vote of one means that the rated dataobject is *poor*, where the highest vote of five marks it as excellent. A modular Tapestry component was created named obviously RatingBox which contains five star images. By moving the mouse over such a star the corresponding item is highlighted to give the user a special grade of interactivity. By clicking on one of them the corresponding dataobject is rated and also here only the area where the rating box resides is updated. The idea was to inject the JavaScript code on render time of the component, but it seems that the Tapestry 4 framework had a problem with this operation. For this reason the JavaScripts are now included in the layout template. After one has rated a dataobject one can revert the action by cancel the current rating vote and start the process again.
This Tapestry component has the following set of input arguments:

- dataObjectId (required): specifies the id of the dataobject. It has to be type of Long.

- updateComponents (optional): holds a list of strings, which specifies the page parts which are refreshed after the rating action.

- starSize (optional): if the star size is specified "small" the stars are represented in a smaller resolution (15x15 pixels), otherwise they are (30x30 pixels).

The same rating functionality was introduced for the experiments. There it was necessary to construct a rating component for experiments called ExperimentRatingBox and to give the experiment entity a new member variable named experimentRating of type integer. Also the ExperimentManagement is extended with the corresponding method to rate the current experiment. The input arguments of the experiment rating box are the same as for the dataobject, except the dataObjectId which was substituted with the argument experimentId.
The following listing shows how to add the described rating box on a Tapestry HTML template. By not specifying the size of stars a resolution of 15 per 15 pixels will be taken as default.

```
<span jwcid="@RatingBox" updateComponents="areaToUpdate"/>
```

Listing 2.5: Tag for importing the RatingBox.

2.4.2 Protocol comparison

During an experiment scientists follow protocols which can be digitally stored in iLAP. In order to gain better results these protocols must be frequently adapted and optimized. If results are improving differences between protocols must be discovered and therefore protocols must be compared.
The first method to compare different iLAP protocols was developed by Oberstolz [32] as a standalone application. This method had to be ported to the web layer in order to offer the iLAP user the comparison functionality. Oberstolz implemented a function that takes two XML protocol files as input arguments and returns a list of so called `AlignmentObjects`. This item holds the left object which is a protocol step out of the first protocol, a right object which is the corresponding step from protocol two and an additional object called `EditOperations`. The design of the protocol comparison is illustrated in figure 2.5.

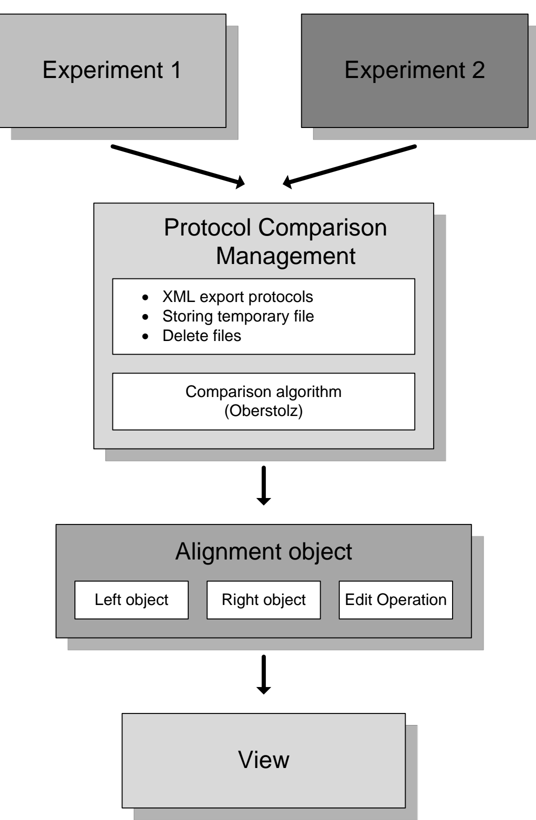


Figure 2.5: Process of protocol comparison.

The edit operation item is represented by an enumeration element, which

can have the following values:

- `UPDATE`
- `DELETE`
- `INSERT`
- `NONE`

An *update* operation means that the corresponding step exists in the two protocols, but was changed slightly in one of them. A *delete* specifies that this specific step exists only in the left protocol, but not in the right one. An *insert* operation is the counterpart of the *delete*. The marked step exists only in the right protocol but not in the left. And last the *none* enumeration points out, that the step in the left and right protocol are equivalent.
During the implementation of the resulting comparison view it must be considered that any protocol step could contain an infinite hierarchy of child steps. This hierarchy must be pointed out in the visualization.
The first step of implementation was to create a new service class in the business layer which includes the comparison calculation. The new service is called `ProtocolComparisonManagement` and is injected in the corresponding Tapestry pages. This service exports the protocols as XML files, performs the comparison and returns the above mentioned list of `AlignmentObjects`. For the export the existing `ArchiveExportService` is used, which returns an XML file of the protocol. Steps within a protocol can also contain notes or parameters. The former can be a notice made by the laborant. The latter can be for example a used liquid and the amount of it in e.g. *milliliter (ml)*. Parameters and notes have to be displayed in a different way as protocol steps, so the user can differentiate them visually. These considerations must be accounted for the visualization. A Tapestry page was designed called `ComparisonPage` consisting of two select boxes. In a first stage the user can choose two protocols out of the set of existing protocols. After confirming the choice the user will be redirected to the `ComparisonResultPage` to get the protocol comparison visualization. Because the comparison can take longer, two pages were implemented; after the submit of the protocol choice a wait box is opened and is replaced by a second page.
For the processing and visualization of the comparison results a new Tapestry component called `LoadComparisonResults` was introduced and derived from the `AbstractComponent`. This component iterates through the list of `AlignementObjects` and visualizes them in a tabular view. To point out the edit operation, steps with differences are highlighted with different colors. The

component also displays the steps in it's hierarchical order. To instantiate this component with the name `LoadComparisonResults` the following input parameters must be specified:

- `firstProtocol (required)`: specifies the first protocol selected. This will then be displayed on the left side of the tabular view. It must be type of `ExperimentBasicVO`, which defines an experiment valueobject.
- `secondProtocol (required)`: the same as the firstProtcol argument with the difference that it will be displayed on the right side of the tabular view.
- `diffList (required)`: holds the `AlignmentObjects` and will be provided from the protocol comparison management service. Its type is a list of these objects.
- `leftProtocolMaxDepth (required)`: holds the depth of childs of the left protocol. It has the type integer.
- `rightProtocolMaxDepth (required)`: holds the depth of childs of the right protocol and its type is also integer.
- `pageName (optional)`: this optional parameter influences the title of the HTML page and is set by default to the name of the comparison results page.
- `updateComponents (optional)`: specifies the components which are updated and can consist of a list of elements.

As described above protocol steps with differences are highlighted with different colors. By moving the mouse over over a highlighted step the specific edit operation is shown above the element. This was again realized with a JavaScript. There is a column on both sides of the comparison view called *type* that specifies if the current line represents a note or parameter. The comparison feature in action is shown in the results section.

2.4.3 Cryptographic infrastructure

An additional step to get away from the paper-based note books is to bring the concept of a physical signature into iLAP. In microscopy experiments the scientist guarantees with the signature for the integrity of the documented experimental data. One of the main aims for the cryptographic functionality implementation is to assure the identity related to an experiment signature

in a reliable and secure fashion. Additionally the introduced service has to have a modality to validate the signature and detect possible manipulation of experimental data. To fulfill these requirements two new entities were introduced.

Data structures and services

Experiment signature A fundamental part of the cryptographic feature is the signature itself. To realize it a new data structure the `ExperimentSignature` was introduced. It consists of the following fields:

- `id`: specifies the internal id of the signature. It is of type *Long* and must be unique.
- `creationTime`: is from type *String*, which defines the date and time when the signature was generated.
- `location`: points to the relative path, where the signature file was stored. It is also of type *String*.

In order to relate a experiment signature to a specific experiment a mapping between the id of the experiment signature and the experiment is enough. The relation between the experiment and the experiment signature is a one to many association. Each signature must belong to at least one experiment, but the experiment can own many signatures. Because there was not introduced a separate data structure for the countersignatures additional self-association was introduced to cover this relationship. Each signature can hold a collection of signatures, which contains the countersignatures. Each countersignature must have exactly one relation to a signature object. The last relation exists between the `SignKey` which will be introduced later. Each signature pertains precisely to one signers key. Also here it is sufficient to map the id of the experiment signature to the id of the signer key. This mapping is realized by Hibernate. The model of associations can be seen in figure 2.6

Signer key Another necessary item for the cryptographic service is the signer key. For this purpose the entity `SignKey` was implemented. This data type consists of the following fields:

- `signKeyId`: holds the id the of the signer key.
- `signKeyPublic`: contains the public key of a user. The type is *String*.

- `signKeyPrivate`: holds the private key of the signer. In contrast to the public key this one is encrypted via a symmetric algorithm and a password. This aspect will be discussed later on.
- `signKeyStatus`: specifies the active status of the corresponding signer key. The type of this field is *boolean*. Per user only one signer key can be active.*true*.

The SignKey object is related in a one to many association with the experiment signature. The relation from the application user to the signer key is a one to many, because each user can have multiple keys. For example if one forgets the password for the private key a new pair must be generated and the old one is stored to verify signatures made until this point of time. All associations between the introduced data structures can be seen in figure 2.6.

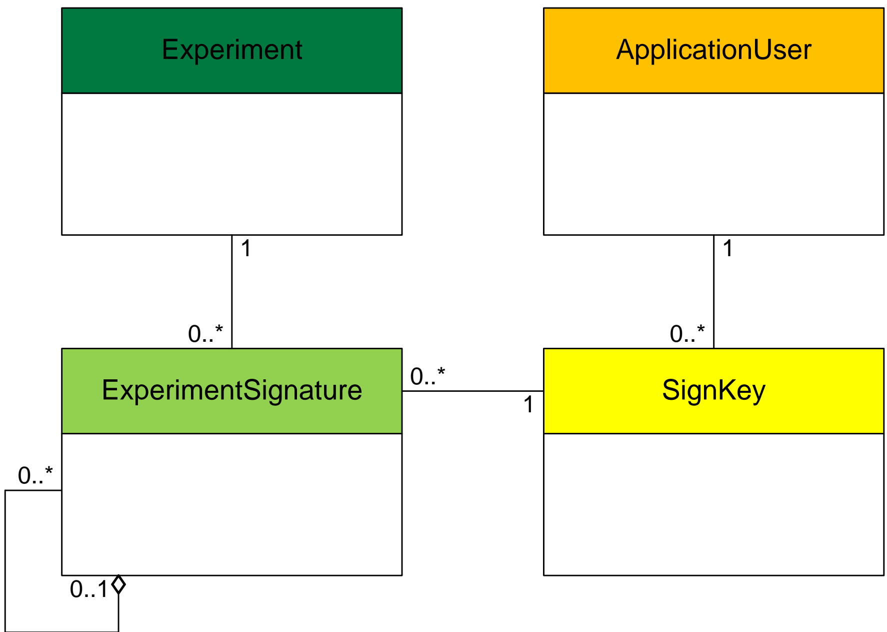


Figure 2.6: The cryptographic entities and their relations.

SignKey management service In order to administer the signer keys the new service `SignKeyManagement` was created. This service performs all necessary operations to store, create and retrieve the signer keys. In detail these are the following methods with their corresponding inputs:

- `createNewSignKey`: this method takes as input parameter the user valueobject and the password. This password will never be stored and only the user knows it. With this password the private key will be encrypted.

- `getSignKeyForUser`: takes the user as input and returns the corresponding signer key. This is the key with the status set to the value true.

- `getAllSignKeysForUser`: does the same as the method above, but returns the complete list of signer keys generated by the current user.

- `getDecryptedSignKey`: takes as input a signer key and the password. The method returns the `SignKey` structure with the decrypted private key.

- `getSignKeyOwner`: gets as input the signer key and returns the owner. The return value is of type `SigningUserVO` which is a derivate of the `ApplicationUserVO` object.

Experiment signature management service This management service is responsible for the experiment signatures and therefore named `ExperimentSignatureManagement`. It provides different methods for persisting the signatures:

- `createSignature`: takes the signer key, the experiment and the relative path into which the signature file has to be stored.

- `createCounterSignature`: has the same inputs as the method for creating a signature but with the additional experiment signature argument, which specifies the signature that must be counter signed.

- `deleteSignature`: simply deletes the current signature by taking the user value object and the experiment signature as inputs. This operation can onyl be performed, if the given user has the special role of `ROLE_SIGNATURE_CLEANER`. This role was first introduced here.

Crypto management service The Cryptographic service builds the core of this new feature. It is designed in a fashion that later on, the algorithm of cryptographic operations on the experiments can be extended easily. This is realized through the implementation of a factory pattern and considering that old signatures can be still verified if new signature methods are introduced.

Therefore a new implementation version must be specified in the `CryptoInfrastruchtureFactory`. Each new implementation must implement the interface `CryptoAgent`. To integrate rapidly cryptographic functionality into iLAP, the java-based JCA/JCE framework is used. This provides standard implementations of cryptographic methods, like signing, verifying or en- and decryption.
The service called `Crypto` consists of the following four principal methods:

- `signExperiment`: specifies the process of signing an experiment and takes the valueobjects for the user, the experiment, the signer key and the password to decrypt the private key as arguments.
- `verifyExperimentSign`: verifies an existing experiment signature and takes therefore as argument the valueobject for the user and the experiment signature itself.
- `counterSignExperiment`: countersigns an experiment signature. It has to be provided with the same input parameters as the `signExperiment` method and additionally the experiment signature that has to be counter signed. Only users with the special role `ROLE_COUNTERSIGNER` are allowed to perform this action.
- `verifyExperimentCounterSign`: verifies a countersignature by verifying also the parent experiment signature. This function gets the same inputs as the `verifyExperimentSign` operation and additionally the experiment signature.

The **signExperiment** method consists of several procedures. In a first step the private key of the signer is decrypted and the experiment specific data is accumulated for the signing process. This includes experiment notes, attached dataobjects and a XML export of the current working protocol. Also a file is created, that holds the signature of each part of the experiment. It has to be noted that each export of a protocol contains a tag of the export date and time. This tag has to be removed, because otherwise the verification process will never be successful. For the purpose of writing all relevant data for the signing process to a file a helper class was added to the framework. The `SignatureWriter` does the work for constructing the signature file. An example file for a typical experiment signature holds the information shown in the listing 2.6. Additionally the listing shows that the file contains the signer information. It is important that this information must be always checked by querying it in the database to be sure that the public key is consistent. The example signature file contains no dataobjects,

but in general this tag consists of the dataobjects id, name and the encrypted hashvalue. All XML tag specifications are defined in the `SignatureXMLSpec` interface. The file name of an experiment signature is composed of the leading string *SignatureFile*, the signers id and the creation timestamp in this way ”*SignatureFile_<SIGNERID>_<TIMESTAMP>.xml*”. This file will be archived in the data folder of the corresponding experiment. Another helper class is the `ExperimentSigner` which digitally signs each part of the protocol separately. Figure 2.7 reveals how the process of signing works in iLAP. For each part of an experiment like notes, dataobjects the hash value is calculated and afterwards it is decrypted with the private key of the signer.
As discussed earlier the cryptographic service was implemented in a manner to support different versions of the underlying cryptographic agent. The agent realized through this work with the name `CryptoAgent_V_1_0` contains the specifications for the algorithms used for key generation, private key encryption, hash calculation and encryption. The JCA/JCE framework supports algorithm implementations of different providers. The algorithms used here are *RSA* from the *SunRsaSign* provider for the key pair generation. The key size can be chosen by varying the corresponding value in the `CryptoAgent_V_1_0` class. The default length is 2048. To store the private key of each signer by encryption in the database the `PBEWithMD5andDES` encryption algorithm from the `SunJCE` provider was applied. In this context the leading *PBE* expression stands for ”Password Based Encryption” and is a symmetric algorithm as discussed in the beginning of this chapter. Normally also an incorrect password input returns a decrypted private key. For this reason a placeholder string was attached to the user specific password, so that a fault entry can be recognized. The last algorithm included for signing data is the `MD5withRSA` provided from `SunRsaSign`.

To validate such a signature the cryptographic service offers the **verifyExperimentSign**. The procedural workflow of this method is similar to the precedent operation of signing with an additional verification step. For this purpose a helper class the `ExperimentVerifier` was introduced. It performs a specific verification operation on each part of the experiment, like the experiment notes, dataobjects and protocol. The `SignatureReader` performs the reading of the decrypted hash out of the signature file. This procedure is illustrated in the figure 2.8. The algorithms for computing the hash value on the data are the same as discussed in the previous section. If these are different a signature can never be verified successfully. Each part of an experiment is signed separately. In that way it can be determined at which parts the verification has failed. For this reason the verification method returns a `VerificationResult` object, that contains collections of verified and not

```
<experiment-signature>
 <!--iLAP Experiment Signature-->
 <cryptoagent-version>0</cryptoagent-version>
 <protocol>
  <protocol-path>u_1011\p_10000\e_10000\signatures\
     ProtocolExport_1_20100224161407.xml
  </protocol-path>
  <protocol-signature>hVNBhdtiizccmHbLx+5bf/
     PdVzAHhyOsDRqKgKv95+A/LaWg7tN0SgqG8nlNhzTM5eUj2/yhIgvq
     cABRsPEJ+RPESzFk3Gh/CW/bUclkyOZ/
     wYAxBwGz5xZseWcG3LRqEsQSsCnvsJYwk698pFGZVYT5
     njWTKKByU9nUIyXamfyWSoazGzp50Q6SX7Tqf2apEdjipZY+ZSReE/
     h9l/ec7nYbRUMonDD5Y67c vG9DpCv3eOLaj4HS4zAPM+
     OkTbhryX4wvHP+ibByQ5T5+W37fAR+CyFvB6GIuhEv0fmqqPbtuRoE
     jgmc9TTlVFJqj10E32oxiBaHDHZ1wKFitDcqfw==
  </protocol-signature>
 </protocol>
 <notes>
  <notes-signature>p40tydMoNiVeMV8ZxusQdZ4+B3uwh/
     WvM00eUu3bIcJK/FhZCa0E1529rOEHbnS0yWsJJdlrhpPD
     qhj3npzXRWx4mYG0i3vN6MmDktG8NrhK6wckGhw4deOKJvQz6eW8W
     cbmcHKBPBVIrnY6K/qY3VwNcSgFMXYIR1uBDEYbYDLGDBSt2OTmM 6
     G8ZAcE3fBDtVDRXIk50HsweCTBi+SPDezYOY7HQYukqLyV 1
     Tpc1FbGTCaIBKeC18VxQxz8K2cyAKe0ZfxAHpOr7v191qveTmAo7dq 7
     Y7KvtqHtSVC0Y9i4CIy9
     BjJaO8hkZu1zfQpagAxu256LkkMF7x8m2wXH+A==
  </notes-signature>
 </notes>
 <dataobjects/>
 <signer>
  <signer-id>1</signer-id>
  <signer-name>huber</signer-name>
  <signer-publickey>
     MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAl35HRlY23M
     jdvC5T7ThUDst8GT4l4Jk0 7FQtfupRsOE+a+
     AuOLXiK919aBHshahq3FQ8kEucYjynxSyWMIGfmMLG+
     uYpUOKSfD5VHnfM08/P UUb8pPWSffRm+7LfLv9S6f7v/65
     oKsJgILOu7dKw3daYGLvikYVgynj9g4EyZrGi6H4vFF2M6Dsz
     VHOcyInzoodrKeyu7yM181ceZD4LQaeYDdugsRrKulC+
     Imc30E9T2xLpcFRRUOLXclK9DPvqYw60
     Syaddnz1Ls5IiOnXg4YCRm7Pi+
     I714K36jQZZctzZIWNtUo0XrBWj8tl4TKsPKjYg8M1XIhSY24Z
     jLDsuwIDAQAB
  </signer-publickey>
 </signer>
</experiment-signature>
```

Listing 2.6: The experiment signature file.

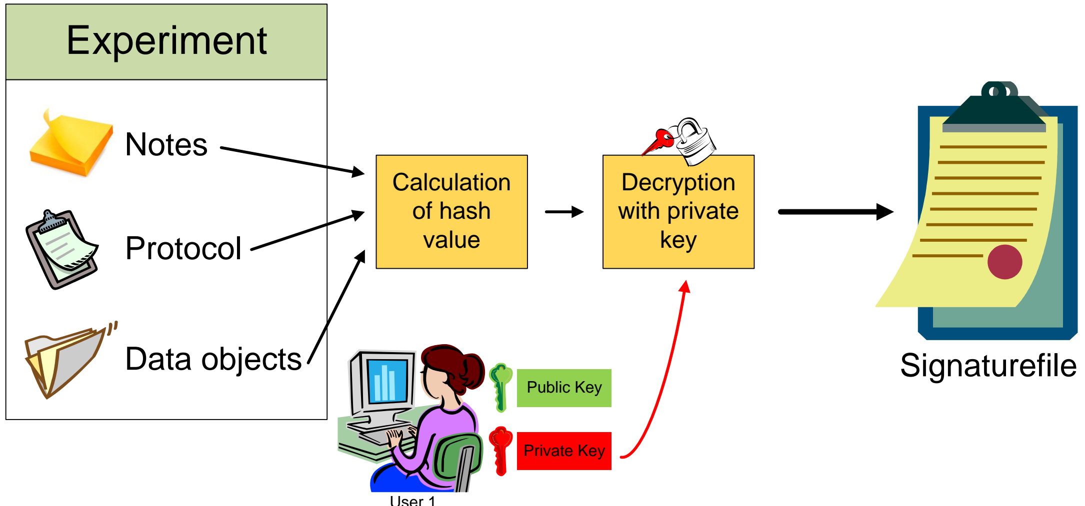


Figure 2.7: Overview of the signing process.

verified dataobjects and booleans for the protocol and notes. This makes it possible to offer the user a verification report by pointing out manipulated parts of the experiment. Corresponding to this verification object a Tapestry component for illustrating this verification report was implemented and will be described in the results part of this work. To get the corresponding verification agent, the version of the used agent is specified in the signature file.

Since one can share experiments and declare access rights to other users also signatures are shared. If a user has the special role for countersigning, the user can create a countersignature as well. This procedure consists in a first step of a verification of the signers signature. If this operation was successful the signature will be countersigned and a countersignaturefile is created. The new generated file contains the same experiment specific information as the signature file, but the signer tag will be substituted by the countersigner tag shown in the listing 2.7. In contrast to the signers tag in the signature file this one holds an additional one the `<countersigner-signature>`. This field holds the decrypted hash value of the signature file of the signer. Also the relative destination of the signature file is attached to the file.

The last method offered by the cryptographic management service is the `verifyExperimentCounterSign`. It performs a verification of the signaturefile and afterwards of the countersignaturefile. The result of the verification

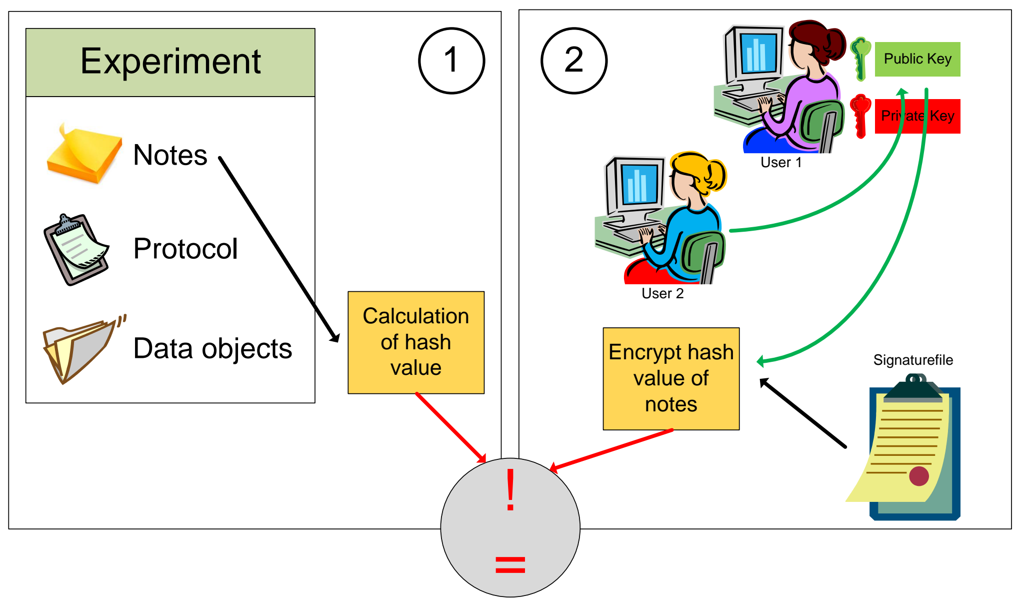


Figure 2.8: Overview of the signature validation.

```
<countersigner>
 <countersigner-id>2</countersigner-id>
 <countersigner-name>stocker</countersigner-name>
 <countersigner-publickey>
    MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqg+ ...
 </countersigner-publickey>
 <countersigner-signature>
  <countersigned-filepath>u_1011\p_10000\e_10000\signatures\
     SignatureFile_1_20100224162333.xml
  </countersigned-filepath>
  <countersigned-signature>p40tydMoNiVeMV8ZxusQdZ4+B3uwh/
     WvM00eUu3bIcJK/ ...
  </countersigned-signature>
 </countersigner-signature>
</countersigner>
```

Listing 2.7: A pullout of the countersignature file.

will also be returned in an instance of the `VerificationResult` as the signature verification operation does.

Graphical user interface

To give the user a graphical interface for signing its experiments some Tapestry components were introduced. These components are integrated on the `ExperimentSignaturesPage` page.
The first component to discuss is the `NewSignKeyForm`. If one has not generated yet a key pair for signing, this component will appear. It consists of two password input fields. The password filled in is the one the user needs to sign an experiment or if this user has the countersigner role to countersign experiment signatures. Only the user knows this password and will never be stored in the database. In the case the password is forgotten, the administrator has only to set the status of this signer key to *false*. The next time the user gets on the signatures page, the indicated component will be shown again. The `NewSignKeyForm` has no input arguments apart from a *updateComponent* parameter which optionally specifies the area that will be refreshed and it is not required. The component consists of a client side password validation. This validation checks the password length and the confirmation field. The password has to be typed two times, if the two input fields are holding the same string and the two input fields are not null the key pair is generated. The signer password has to have a minimal length of "8" and maximal length of "16" digits. These values can be changed in the interface `PasswordSpecs`.
The next component implemented for the purpose of signing is the `NewExperimentSignatureForm`. This component offers the principal operation of signing an experiment as described before. This component has the following three input arguments:

- `signer (required)`: is of type `GlobalUserVO` and specifies the signing user.
- `experiment (required)`: defines the experiment that has to be signed and is an instance of the `ExperimentBasicVO` valueobject class.
- `updateComponent (optional)`: specifies the area that has to be updated after signing.

Another component is the `VerificationDetails` component, that generates a visual representation of the verification results in the case the verification process fails. It writes a list of the experiment parts like the protocol,

notes, dataobjects and countersignatures and shows the corresponding verification status. As the component described before it holds an input parameter *updateComponents* and additional it gets an object of the type `VerificationResult` as described in the section before.
For countersigning a specific experiment signature the `NewCounterSignatureForm` was designed and is composed by a single input field for the password and two buttons for the *submit* and *cancel* operation. This field will only appear if the current user holds the countersigner authorization. It will be displayed beside the corresponding signature. It has to be provided by the following inputs:

- `countersigner (required)`: holds an object of the `GlobalUserVO` class and defines the countersigning user.
- `signature (required)`: specifies the signature that will be countersigned.
- `updateComponent (optional)`: defines the area that will be refreshed after the action.

Screenshots of the `ExperimentSignaturesPage` composed by these components and a tabular view of existing signatures will be shown in the next chapter.

Chapter 3

Results

The main achievement of this work is the introduction of the digital signature process. iLAP was extended to offer its users several cryptographic methods like signing, verifying and countersigning of experiments. To guarantee the identity of the signer and countersigner a special key store was created. Additionally a web based protocol comparison for experiments in iLAP was generated. This assists the user in detecting relevant differences derived from protocol optimization or unauthorized changes. To enhance the usability of data handling in iLAP several visual components were implemented.

3.1 Cryptographic infrastructure

Before a user gets the possibility to sign experiments one has to generate a user specific key pair. The component shown in figure 3.1 provides this operation of key pair generation. The user has to enter a password, that is required for the encryption of the private key as discussed earlier. The password has to be typed in two times and will never be stored. After a key pair is generated the view of the page will change. From then on one can sign experiments as demonstrated in figure 3.2.

Figure 3.2 shows a selected experiment where a set of existing signatures and corresponding countersignatures is displayed. This view is an example for a user who has the authorization of deleting signatures and creating countersignatures. By clicking on the *Verify* link in the right column an experiment signature or countersignature can be validated. If this process fails one can further inspect the reasons for the unsuccessful verification procedure. An example for this is shown in figure 3.3.

The validated signature has two dataobjects attached, which are verified successfully. In the screenshot the part of the experiment which has caused

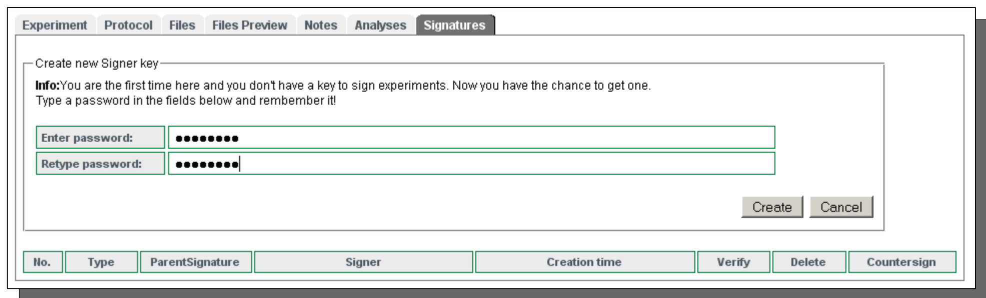


Figure 3.1: Component for creating a new key pair.

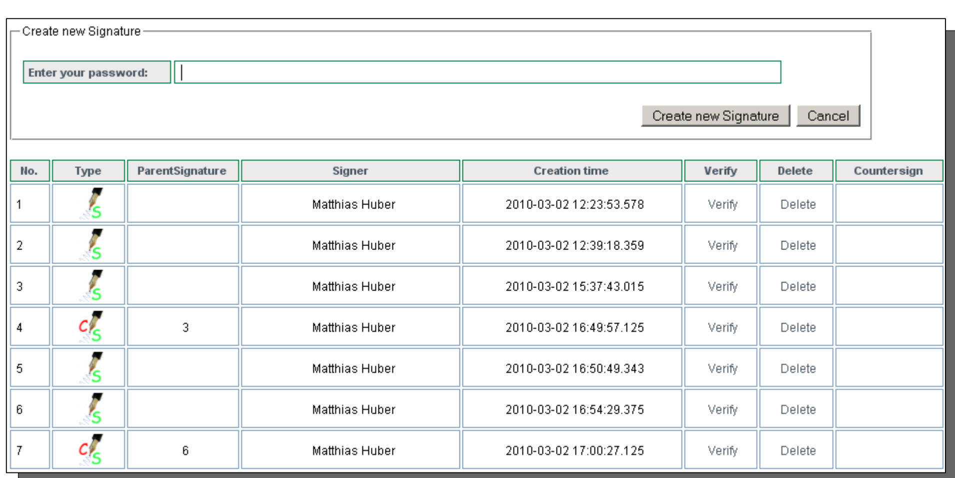


Figure 3.2: Tabular view of signatures and countersignatures in an experiment.

the validation failure is located in the protocol. In this case it is possible to compare the protocol attached to the signature with the current one. The protocol comparison presented in the next section will be a good choice for this.

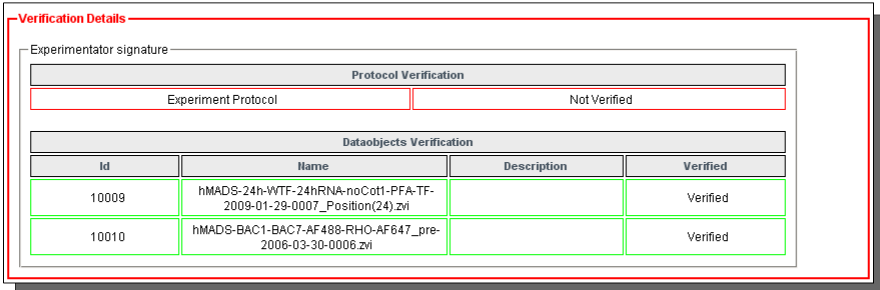


Figure 3.3: Feedback mask for a failed signature verification.

3.2 Protocol comparison

In order to compare two protocols the iLAP user has to go to the comparison page. In a first phase this page consists of two property selection boxes illustrated in figure 3.4 where also shared projects are provided. After one has chosen two protocols, the comparison can be started. For extensive protocols this can take some time. Therefore an intermediate wait box is displayed.

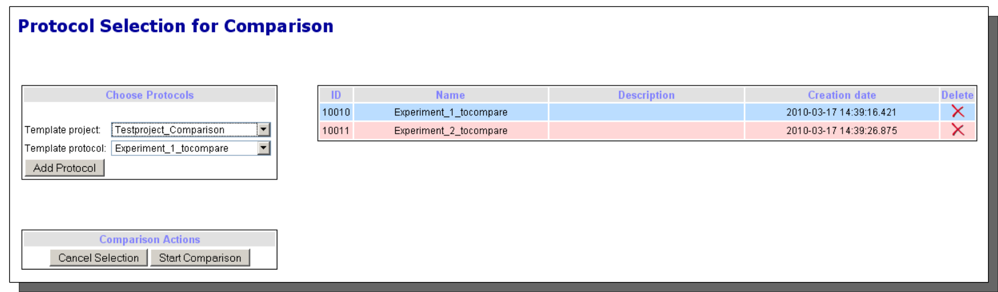


Figure 3.4: Page for selecting protocols before comparison.

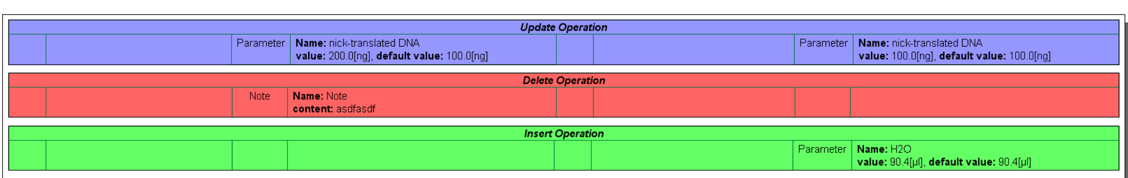
After the comparison process is completed, the user reaches the comparison view (see figure 3.5). The left side of the view represents the first protocol selected and the right the second one. Each protocol step is consecutively

numbered and parameters as well as notes are identified in the *Type* column. All lines that are rendered in black occur unchanged in both protocols. Colored sections mark the detected differences, which are illustrated in figure 3.6.

Experiment_1_tocompare			Experiment_3_tocompare		
Step Name	Type	Content	Step Name	Type	Content
1 Cell preparation / Probe preparation		**description:** cells and FISH probes are prepared for hybridization in parallel	1 Cell preparation / Probe preparation		**description:** cells and FISH probes are prepared for hybridization in parallel
1.1 DNA FISH probe preparation		**description:** FISH probe preparation: precipitate probe DNA together with human Cot1 and salmon sperm DNA and dissolve in hybridization buffer	1.1 DNA FISH probe preparation		**description:** FISH probe preparation: precipitate probe DNA together with human Cot1 and salmon sperm DNA and dissolve in hybridization buffer
1.1.1 prepare DNA mix for EtOH precipitation		**description:** Mix the following: - ~100 ng nick-translated BAC-probe-DNA for each target gene - 0.7 x 2/3 x n µl salmon sperm DNA 10 mg/ml (n=number of total probes used for combinatorial labeling. e.g. 7 targets n = 12) - 5 x 2/3 x n µl cot1 DNA 1 mg/ml (n=number of total probes used for combinatorial labeling. e.g. 7 targets n = 12) - 11.3 x 2/3 x n µl ddH20 (n=number of total probes used for combinatorial labeling. e.g. 7 targets n = 12) - 2 x Vol. abs. EtOH - 1/10 x Vol. 3M sodium acetate pH 5.2	1.1.1 prepare DNA mix for EtOH precipitation changed for testing		**description:** Mix the following: - ~100 ng nick-translated BAC-probe-DNA for each target gene - 0.7 x 2/3 x n µl salmon sperm DNA 10 mg/ml (n=number of total probes used for combinatorial labeling. e.g. 7 targets n = 12) - 5 x 2/3 x n µl cot1 DNA 1 mg/ml (n=number of total probes used for combinatorial labeling. e.g. 7 targets n = 12) - Same here 11.3 x 2/3 x n µl ddH20 (n=number of total probes used for combinatorial labeling. e.g. 7 targets n = 12) - 2 x Vol. abs. EtOH - 1/10 x Vol. 3M sodium acetate pH 5.2
	Parameter	**Name:** nick-translated DNA **value:** 200.0[ng], **default value:** 100.0[ng]		Parameter	**Name:** nick-translated DNA **value:** 100.0[ng], **default value:** 100.0[ng]
	Parameter	**Name:** salmon sperm DNA **value:** 5.6[µl], **default value:** 5.6[µl]		Parameter	**Name:** salmon sperm DNA **value:** 5.6[µl], **default value:** 5.6[µl]
	Parameter	**Name:** human cot1 DNA **value:** 40.0[µl], **default value:** 40.0[µl]		Parameter	**Name:** human cot1 DNA **value:** 40.0[µl], **default value:** 40.0[µl]
				Parameter	**Name:** H2O **value:** 90.4[µl], **default value:** 90.4[µl]
	Parameter	**Name:** sodium acteate **value:** 17.0[µl], **default value:** 17.0[µl]		Parameter	**Name:** sodium acteate **value:** 17.0[µl], **default value:** 17.0[µl]
	Parameter	**Name:** abs. EtOH **value:** 340.0[µl], **default value:** 340.0[µl]		Parameter	**Name:** abs. EtOH **value:** 340.0[µl], **default value:** 340.0[µl]
1.1.2 put the DNA mixture at -70°C		**description:** put the DNA mixture for 30 min. at -70°C	1.1.2 put the DNA mixture at -70°C		**description:** put the DNA mixture for 30 min. at -70°C
	Note	**Name:** Note **content:** asdfasdf			
	Parameter	**Name:** temperature **value:** -70.0[°C], **default value:** -70.0[°C]		Parameter	**Name:** temperature **value:** -70.0[°C], **default value:** -70.0[°C]
	Parameter	**Name:** time **value:** 30.0[minutes], **default value:** 30.0[minutes]		Parameter	**Name:** time **value:** 30.0[minutes], **default value:** 30.0[minutes]
1.1.3 spin down in a centrifuge		**description:** spin the precipitated DNA down in a centrifuge	1.1.3 spin down in a centrifuge		**description:** spin the precipitated DNA down in a centrifuge

Figure 3.5: Comparison view of two different protocols.

By moving the mouse over a colored region an additional line appears which explains the corresponding change. Blue marked steps indicate changes in the content. Red lines stand for a deletion, which means that this specific step exists only in the left protocol. Steps that are marked in green are inserts and exist only in the right protocol.



Update Operation					
	Parameter	**Name:** nick-translated DNA **value:** 200.0[ng], **default value:** 100.0[ng]		Parameter	**Name:** nick-translated DNA **value:** 100.0[ng], **default value:** 100.0[ng]
Delete Operation					
	Note	**Name:** Note **content:** asdfasdf			
Insert Operation					
				Parameter	**Name:** H2O **value:** 90.4[µl], **default value:** 90.4[µl]

Figure 3.6: Possible differences between two protocols.

3.3 Usability improvements

The figure 3.7 shows the `DataObjectDetailView` component which was extended by the following features. The first one is the edit box for changing the

description of a dataobject. By clicking the *Edit* link an input field appears in which one can type in a new description. The changing of the behavior of this component is done in a way that only the part marked with "1" in figure 3.7 is refreshed.
The area marked with "2" opens the big preview of the current dataobject which is shown in figure 3.8.
Also for the experiment exists a component that shows the experimental details. There a similar preview link can be found. By accessing this link one gets to the preview page where all dataobjects attached to the corresponding experiment are displayed.
The third marked area is the RatingBox. In the depicted case 3.7 the dataobject has the highest rate and the rating can be changed at any time. The rating function was also ported to experiments. In the tree view of the overview page the ratings are also displayed. The last improvement marked in the figure is the delete operation. This link appears only if the current user holds the role of *ROLE_FILE_MANAGER*. The operation is not processed immediately because a confirmation dialog is opened to avoid unwanted deletion.

As mentioned before the big preview looks like shown in the figure 3.8. On the left it has similar fields as in the `DataObjectDetailView` and shows dataobject specific information like file name, description, file size and so on. The second table at the left consists of operations like download, rating and a close button. The table headers are colored according the common iLAP color code of the parent like green for experiments, blue for projects, red for analyses, etc.

Figure 3.9 demonstrates a typical view of all dataobjects that are attached to an experiment. In this example the `DataObjectImageGrid` component is configured to show only three columns. This can be changed by modifying the corresponding parameter of the component. For images or dataobjects that possess no preview a placeholder picture is shown that indicates the absence of it. On this page also the big preview can be started for further inspection and rating can be performed.

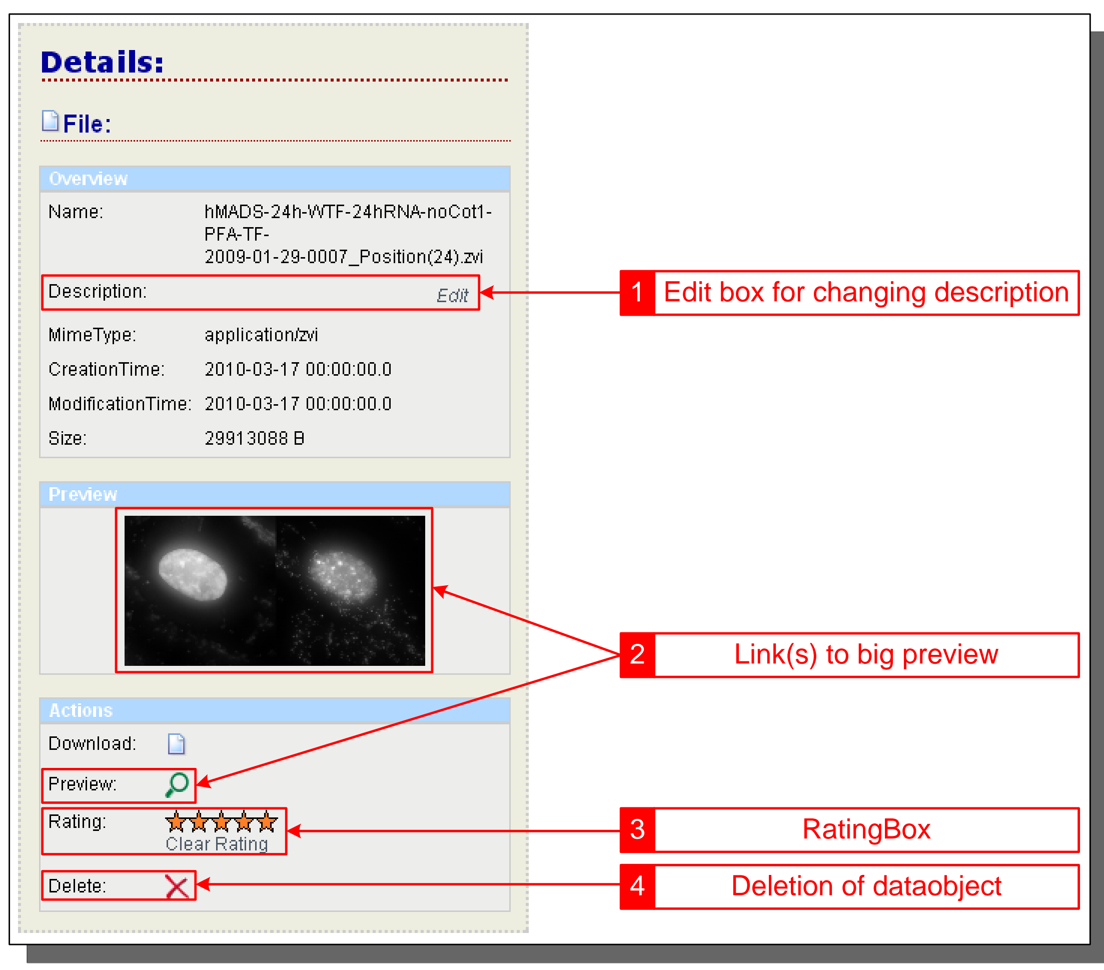


Figure 3.7: Usability enhancements for the dataobject details.

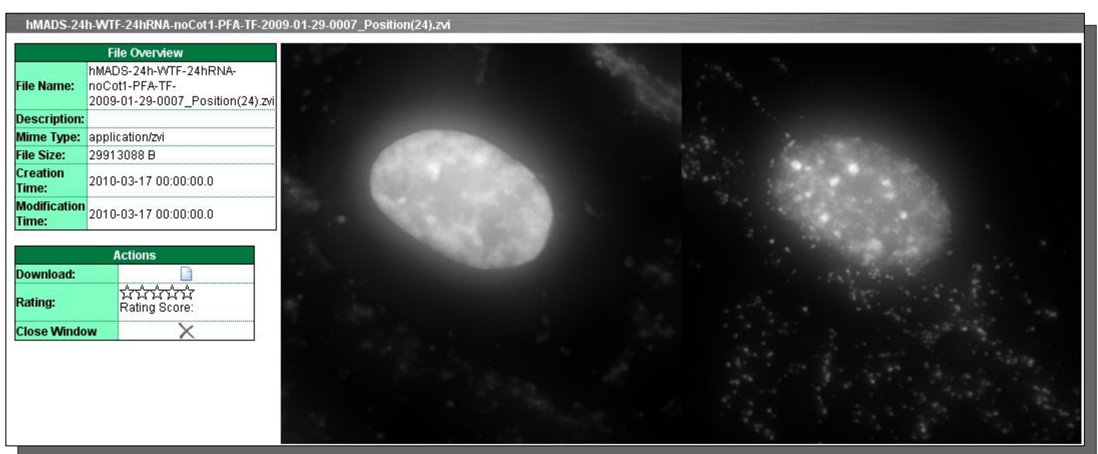


Figure 3.8: Big preview of a dataobject attached to an experiment.

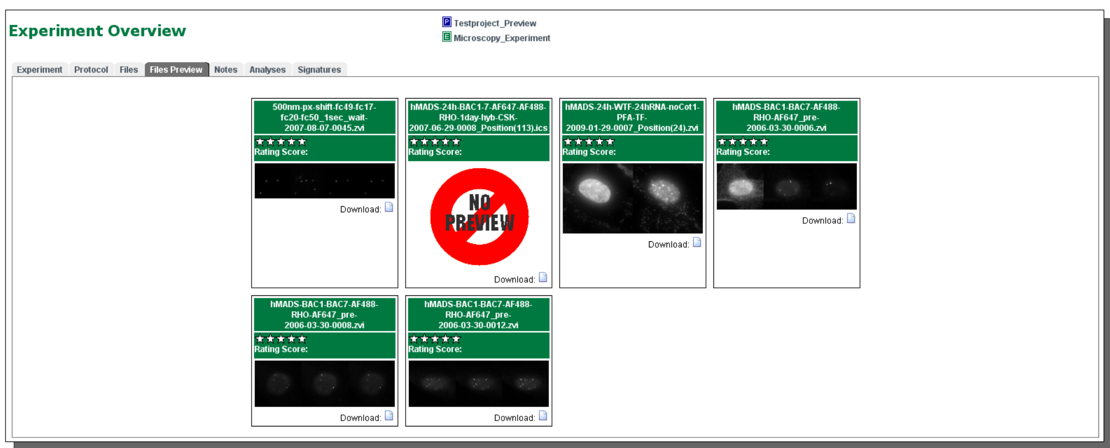


Figure 3.9: Example for an overview page for experimental data.

Chapter 4

Discussion and Outlook

4.1 Discussion

One major aim of iLAP consists in the substitution of the handwritten notebooks in the laboratory. An important part of such notebooks is the physical signature of the experimenter to guarantee for the authenticity of the described data. In analogy to this physical signature the concept of digital signature was integrated in iLAP, which gives the user the opportunity to sign, validate and countersign experiments. This functionality allows to validate the consistency of the experiment as well as the authenticity of the signer. In a next step a key store was introduced to persist asymmetric keys in a reliable manner. A user with a higher authorization such as a head of laboratory can countersign the signed experiment, which represents the last instance of this process. This functionality improves the quality management of worflows and brings iLAP closer to a full featured digital labbook.

In order to detect differences between protocols and to optimize results of experiments it is essential to compare them. Therefore iLAP was additionally extended to visualize protocol comparison based on an already existing algorithm. This comparison allows the experimenter on the one hand to detect illegal changes of protocols and on the other hand the optimization of experimental data. This feature had to be integrated in the web interface of iLAP. An additional user interface was created to select two protocols out of a working set. This selection is used in a component that produces an intuitive representation of the resulting differences.
Further on the usability of iLAP was improved and essential extensions were integrated in the area of file preview and experimental rating of data.

4.2 Outlook

4.2.1 Usability improvements

iLAP represents a solid platform to describe laboratory workflows and to store related data. Automatic postprocessing is performed for example multichannel images are processed to generate ordinary preview images with color channels. A possible enhancement is the implementation of an image viewer for high level image processing. Such interesting processing might be image segmentation, where the user marks manually a set of seed points to extract a region of interest.
Since the web layer is realized with the Tapestry 4 framework, another enhancement could be the porting of the web tier to the new Tapestry 5 web framework. Tapestry 5 introduces some new interactive components and pages are based on POJOs. Therefore it is not necessary anymore to inherit them from framework specific classes. Also the testing procedures of web components are easier to perform. However, the realization would be a huge effort because Tapestry 5 is not backwards compatible. For further readings on this topic consult [30] or [23].

4.2.2 Protocol comparison

The protocol comparison tool was newly introduced in iLAP during the implementation of this thesis. Nevertheless one further possible enhancement could consist in the comparison of multiple protocols. Hereby it would be interesting to find hotspots in a complete series of protocols or experiments. Another drawback is the minimal interaction of the user with the comparison view. A comparison view with editable protocol steps, parameters and notes would be useful as well.

4.2.3 Cryptographic infrastructure

The main goal of this work was to create a service for signing and validating experimental data. This feature could be extended to sign also a complete project hierarchy in iLAP.
In addition to this there exist several software projects that are implementing new algorithms in their own provider implementations for the JCA framework. The integration of such a implementation in the existing cryptographic infrastructure of iLAP could provide additional state of the art algorithms for encryption, signing, etc. Such implementations are commercially available like the *CRYPTO Toolkit* from the IAIK group [5]. Another non-commercial

implementation for Java represents the *Bouncy Castle Crypto API* [20].
A further enhancement could be the usage of a certified signature. This would imply the requirement for a certificate provider. If this provider is an accredited trustcenter, the signature would be legally binding as well. The last possible improvement could be the integration of smartcards in the cryptographic process of iLAP.

Glossary

AJAX Asynchronous JavaScript and XML

AOP Aspect Oriented Programming

API Application

CWP Current Working Protocol

DAO Data Access Object

EIS Enterprise Information System

EJB Enterprise Java Beans

HTML Hypertext Markup Language

IDE Integrated Development Environment

iLAP Laboratory data management, Analysis and Protocol development

IoC Inversion of Control

JEE Java Platform, Enterprise Edition

JPG Joint Photographics expert Group

JSP Java Server Pages

MDA Model Driven Architecture

PDF Portable Document Format

PNG Portable Network Graphics

POJO Plain Old Java Object

SOP Standard Operating Procedure

SQL Structured Query Language

UML Unified Modeling Language

VO Value Object

XMI XML Metadata Interchange

XML Extensible Markup Language

List of Figures

1.1 Overview of the intended data hierarchy in iLAP. 8

2.1 Representation of three tier architecture. 12
2.2 Representation of the modular architecture within Spring framework. 15
2.3 Representation of the provider framework of the JCA. 18
2.4 Dataobject representation. 27
2.5 Process of protocol comparison. 32
2.6 The cryptographic entities and their relations. 36
2.7 Overview of the signing process. 41
2.8 Overview of the signature validation. 42

3.1 Component for creating a new key pair. 46
3.2 Tabular view of signatures and countersignatures in an experiment. 46
3.3 Feedback mask for a failed signature verification. 47
3.4 Page for selecting protocols before comparison. 47
3.5 Comparison view of two different protocols. 48
3.6 Possible differences between two protocols. 48
3.7 Usability enhancements for the dataobject details. 50
3.8 Big preview of a dataobject attached to an experiment. 51
3.9 Example for an overview page for experimental data. 51

Bibliography

[1] AndroMDA.
http://www.andromda.org/index.php
April 8th, 2010

[2] AndroMDA Cartridges.
http://www.andromda.org/docs/andromda-cartridges/index.html
April 8th, 2010

[3] AndroMDA Hibernate Cartridge.
http://team.andromda.org/docs-3.3/andromda-cartridges/andromda-hibernate-cartridge/profile.html
April 8th, 2010

[4] Eclipse.
http://www.eclipse.org
April 8th, 2010

[5] IAIK CRYPTO Toolkit.
http://jce.iaik.tugraz.at
April 8th, 2010

[6] Introduction To OMG's Unified Modeling Language.
http://www.omg.org/gettingstarted/what_is_uml.htm
April 8th, 2010

[7] Java Cryptography Architecture (JCA), Java Cryptography Extension (JCE).
http://www.docstoc.com/docs/20887185/Java-Cryptography-Architecture-%28JCA%29-Java-Cryptography-Extention-%28JCE%29
April 8th, 2010

[8] Java(TM) Cryptography Architecture (JCA) Reference Guide.
http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
April 8th, 2010

[9] Java(TM) Cryptography Extension (JCE) Reference Guide.
http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html
April 8th, 2010

[10] MagicDraw UML.
http://www.magicdraw.com
April 8th, 2010

[11] Maven.
http://maven.apache.org/what-is-maven.html
April 8th, 2010

[12] MDA-Frameworks: AndroMDA.
http://www.wi.uni-muenster.de/pi/lehre/ws0506/seminar/02_andromda.pdf
April 8th, 2010

[13] MyEclipse.
http://www.myeclipseide.com
April 8th, 2010

[14] Object Management Group.
http://www.omg.org
April 8th, 2010

[15] pgAdmin.
http://www.pgadmin.org
April 8th, 2010

[16] PostgreSQL.
http://www.postgresql.org
April 8th, 2010

[17] Spring.
http://www.torsten-horn.de/techdocs/jee-spring.htm
April 8th, 2010

[18] Tacos Library.
http://tacos.sourceforge.net/tacos4.1/tacos-core/tapdocs/index.html
April 8th, 2010

[19] Tapestry, Framework Component Reference.
http://tapestry.apache.org/tapestry4.1/components/index.html
April 8th, 2010

[20] The Legion of the Bouncy Castle.
http://www.bouncycastle.org
April 8th, 2010

[21] Unified Modeling Language.
http://www.uml.org
April 8th, 2010

[22] Dhrubojyoti, K. *Pro Java(TM) EE Spring Patterns.* Apress, 2008.

[23] Drobiazko, I. *Tapestry 5, Die Entwicklung von Webanwendungen mit Leichtigkeit.* Addison-Wesley, 2010.

[24] Fischer, M. Digital Lab Book, a web-based module for experiment management within the Scientific Microscopy Lab Environment project. Master's thesis, Graz University of Technology, 2006.

[25] Hamilton, K., and Miles, R. *Learning UML 2.0.* O'Reilly, 2006.

[26] Harrop, R., and Machacek, J. *Pro Spring.* Apress, 2005.

[27] Hook, D. *Beginning Cryptography with Java.* Wiley Publishing Inc., 2005.

[28] Johnson, R. *Expert One-on-One J2EE Design and Development.* Wiley and Sons, 2002.

[29] Kainz, S. "Data retrieval" and "automatic data post-processing" within iLAP (Laboratory datamanagement, Analysis and Protocol development). Master's thesis, Graz University of Technology, 2008.

[30] Kolesnikov, A. *Tapestry 5, Building Web Applications.* Packt Publishing, 2007.

[31] Mak, G. *Spring Recipes: A Problem-Solution Approach.* Apress, 2008.

[32] Oberstolz, M. Archiving and comparison of experimental data within iLAP (Laboratory data management, Analysis and Protocol development). Master's thesis, Graz University of Technology, 2009.

[33] Rosen, K. H. *An Introduction to Cryptography.* Taylor & Francis Group, LLC, 2007.

[34] Ship, H. M. L. *Tapestry in Action.* Manning, 2004.

[35] Stark, T. *J2EE - Einstieg für Anspruchsvolle.* Addison-Wesley, 2005.

[36] Stocker, G., Fischer, M., Rieder, D., Bindea, G., Kainz, S., Oberstolz, M., McNally, J. G., and Trajanoski, Z. iLAP: a workflow-driven software for experimental protocol development, data acquisition and analysis. *BMC bioinformatics 10* (2009).

[37] Tong, K. *Enjoying Web Development with Tapestry.* Tip Tec Development, 2005.