University of Applied Sciences and Arts
Northwestern Switzerland

Project Report

Automatic Detection and Analysis of Tumor Tissue in Prostate Punch Biopsies

Implementation of an Inventory System to Acquire Digital Image Data

Master of Science in Engineering

P7, Autumn Semester 2014

Dario Vischi

Advisor:
Prof. Dr. Christoph Stamm, FHNW

Customer:
Prof. Dr. Peter Wild, USZ
Dr. Qing Zhong, USZ
Norbert Wey, USZ

Institute: Institute of Mobile and Distributed Systems (IMVS)

I like to thank Prof. Dr. Christoph Stamm, Prof. Dr. Peter Wild, Dr. Qing Zhong and Mr. Norber Wey who made the project possible and provided valuable advisory support whenever needed. Also, I like to thank Mr. Roman Bolzern who supported me with new ideas and concepts in terms of web engineering during many discussions.

Abstract

The prostate cancer is the most commontype of cancer we have in Switzerland. Every year about 5300 people develop cancer and around 1300 men die from it. The research for unambiguous indicators for an early detection of the cancer is nowadays an active field in the area of medication. In this context, the aim of a current project at the university hospital of Zurich is the automatic detection of tumor tissues in prostate punch biopsies. We would like to perform the detection on a cohort from Aarau with samples from about 9900 men to build up a model to describe the cancer's progress.

The current documentation at hand describes the first out of three sub-projects for the automatic detection of the tumor tissues. We start with the acquisition of the image data by scanning prostate punch biopsies from the cohort from Aarau. Therefore we implement a semi-automatic process using a PHP web front-end to inventorize and manage the patient's data available from the given cohort.
The data is currently expected to be in Microsoft Excel's XLSX format whereas more formats might be supported in the future.
In a next step we have to manually scan the biopsies which are then automatic registered to the previously inventorized patient data. Up to now we could scan 261 slides of the cohort, from which 52 slides are manually annotated with tumor tissues. These annotations will be used later on to evaluate an algorithm for the automatic detection of cancer.

In the next sub-project the current process will be extended with the automatic detection of cell nuclei based on the 261 slides already scanned and further slides to scan. Next to the image processing part the process should also evaluate the quality of the algorithm, which is especially important to optimize the tumor detection algorithm from the third and last sub-project.

Table of Content

Abstract

Abbreviations 1

1 About the Document 2

2 Project Definition 3
2.1 Background . . . 3
2.2 Identification of requirements . . . 5

3 Acquisition Process 7
3.1 Storing diagnosis data form research studies . . . 7
3.2 The underlying PHP framework . . . 11
3.3 Import diagnosis data from research studies . . . 11
3.4 Print bar codes . . . 12
3.5 Register scanned images to the database . . . 14
3.6 Register image annotations to the database . . . 15
3.7 Export data from the database . . . 15
3.8 Bring it all together . . . 15

4 Software Architecture 18
4.1 The "Patho - Study and Research" system . . . 18
4.2 The File Organizer process . . . 20
4.3 Physical representation . . . 20
4.4 The 4+1 Architectural View Model . . . 23
4.4.1 Logical view . . . 23
4.4.2 Development view . . . 36
4.4.3 Physical view . . . 47
4.4.4 Process view . . . 48
4.4.5 Use case view . . . 51

5 Software Testing 56
5.1 Function tests . . . 56
5.2 System test . . . 56
5.3 Security test . . . 57

6 Results 58

7 Reflection 62
7.1 General recommendations . . . 63

8 Bibliography 64

9 Declaration of Originality 65

10 Appendix **66**
10.1 Software construction tools . . . 66
10.1.1 PHP_UML . . . 66
10.1.2 PHPDoc . . . 67
10.2 Selenium test cases . . . 68
10.3 Attached materials . . . 72

Abbreviations

API	Application Prog. Interface	Interface for interacting with a system
C#	C Sharp	Programming language for the CLI
CLI	Common Language Infrastruct.	System spec. for platform independency
CSV	Comma-Separated Values	Data representation and file extension
DAO	Data Access Object	Abstract interface for db. tables
DB	Database	Data collection
DBMS	Database Management Systems	System for managing databases
FHNW	Fachhochschule Nordwestschweiz	Univ. of Appl. Sc. and Arts NW Switzerland
IDE	Integrated Dev. Environment	Collection of app. supporting software eng.
IT	Information Technology	Technologies related to data processing
MSSQL	Microsoft SQL	SQL for Microsoft's SQL server
ODBC	Open DataBase Connectivity	Programming interface for accessing DBMS
ORM	Object-Relational Mapping	Concept of mapping objects into rel. db.
PHP	PHP: Hypertext Preprocessor	Server-side scripting language for web dev.
PSR	"Pathology-Study & Research"	Inventory system for study data
SQL	Structured Query Language	Query language for databases
TIF	Tagged Image File	Image format for raster graphics
TMA	Tissue Microarray	Collection of up to 1000 tissue cores.
UML	Unified Modeling Language	Modeling language used in software eng.
URL	Uniform Resource Locator	Reference to a resource
USZ	Universitaetsspital Zuerich	University hospital of Zurich
XLSX	Office Open XML	Microsoft Excel's XML-based file format
XML	Extensible Markup Language	Data representation of hierarchical data
ZF	Zend Framework 2	Enterprise web app. framework for PHP

1 About the Document

The documentation at hand describes the inventory process of the cohort from Aarau and the obtained results. We start with the project definition, its background and the origin problem to solve in chapter 2. Chapter 3 discusses possible solutions to the presented problem and describes the acquisition process build up. More technical details about the software architecture involved into the acquisition process can be found in chapter 4. As before, we present the underlying problems and the chosen solutions by looking at the architecture from five different aspects. Chapters 5 and 6 complete the acquisition process by presenting and verifying its results whereas chapter 7 gives a final overall reflection about the whole project.

2 Project Definition

In the first part of the project called "Automatic Detection and Analysis of Tumor Tissue in Prostate Punch Biopsies" we build up a process to inventorize patient data and there related punch biopsies from medical studies. In our concrete case we are working with a cohort from Aarau with samples of about 9900 men where cancer could be detected by 475 patients [1]. With the PSR system we migrate the anonymized diagnosis data of the 475 patients from the study and scan all related tumor tissues with the Ventana iScan HT[1]. Afterward, the images are examined by Prof. Dr. Peter Wild from the university hospital of Zurich and Prof. Dr. Grobholz from the canton hospital of Aarau who annotate the tumor tissues. The image's metadata and annotations are imported into the PSR system to get a database containing all relevant information we need for implementing and testing an algorithm which do the same but automated examination of tumor tissues.

In the second part of the project we then plan to integrate a second process to apply and evaluate image processing algorithms on the high-quality image data obtained so far. Finally, the third part deals with the implementation of a concrete algorithm for detecting tumor tissues inside the given images and evaluation of the algorithm.

Further details can be found in the attached document "Project definition" which is also listed in the appendix 10.3.

2.1 Background

The prostate cancer is the most common type of cancer we have in Switzerland. Every year about 5300 people develop cancer and around 1300 men die from it. A diseased person can be actively medicated, but only with the risk of complications and adverse reactions. The medication is not only restrictively recommended because of the possible risks but also because only 3 out of 40 people die as can be proved by prostate cancer. A major problem presents the early detection of the cancer. All known methods such as the digital rectal examination, the PSA-Test[2] or the biopsy of the prostate do not present unambiguous indicators. It is part of the nowadays ongoing research to find better indicators, e.g. the European Randomized Study of Screening for Prostate Cancer (ERSPC) [2, p.17] [3, p.2]. A current study at the university hospital of Zurich researches a regression analysis of historical data from patients and extracted features

[1] `http://www.ventana.com/product/page?view=iscanht`

[2] Test method for measure the amount of prostate-specific antigen (PSA) inside the blood

from DNA, RNA and Protein analyses. One sub-project of this regression analysis deals with the extraction of features from prostate images which will be combined with the features from the other areas. For this purpose an inventory system is required firstly which allows to manage the data of prostate cancer diagnoses. Secondly, in the follow-up projects we attempt to implement an algorithm for detecting cancer cells on which the features extraction will be based on. Figure 2.1 shows an overview of the whole project stack.

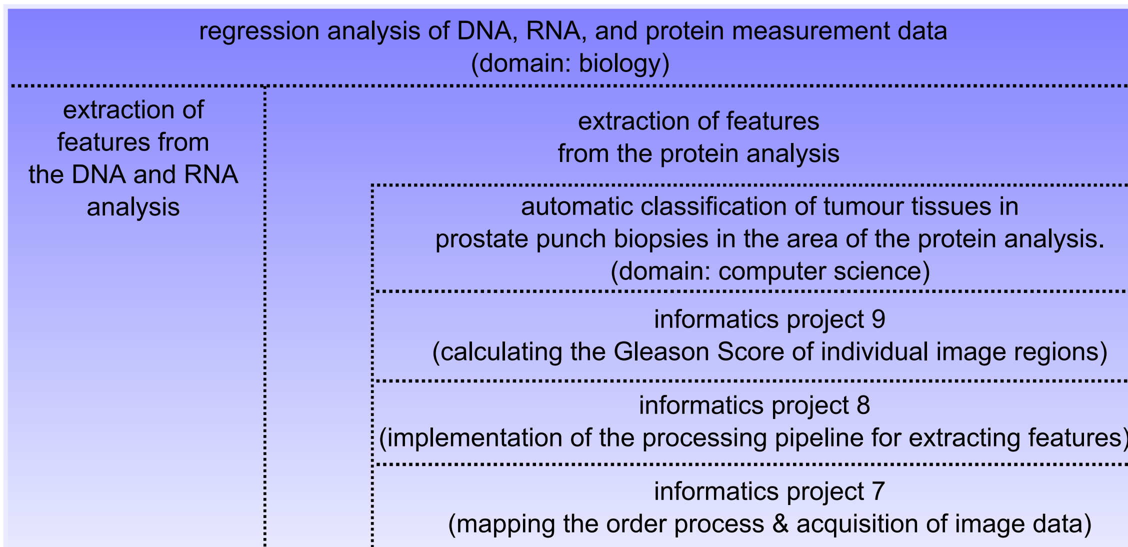


Figure 2.1: Overview of the project stack

The main objectives of the roadmap concerning the informatics project (IP) 7 to 9, which are part of the Master education, can be summarized as follows:

Project	Short description
IP7	Acquisition of digital image data with the Ventana iScan HT and semi-automation of the inventory process. Additionally, annotation of the image data with areas of normal and cancerous tissue.
IP8	Segmentation of the cell cores inside the acquired image data and calculation of the density distribution of cores within normal and cancerous tissues. Furthermore, implementation of a process to gain a precision & recall graph of an algorithm, e.g. for core segmentation, to measure its performance.
IP9	Image analysis of the acquired image data for areas with normal and cancerous tissue. The used algorithms will be validated with the annotations from IP7 as a test set and the process from IP8 to measure its performance.

Table 2.1: Overview of the informatics projects

2.2 Identification of requirements

The university hospital of Zurich has a developed IT infrastructure where we would like to implement the inventory system. The infrastructure already holds several provisions we need to consider. Firstly, the photographic laboratory where we scan the tissues later on already implemented several applications written in PHP, C# and C++ using a Microsoft SQL database as persistency layer. Based on this knowledge and the good experience from earlier projects using those technologies the inventory system should also be built up with PHP and C# as far as possible and using the Microsoft SQL database as its data source. Furthermore, the interface to interact with the database has to be ODBC. Secondly, we decided to use the new Ventana iScan HT from Roche, the self-proclaimed most powerful scanner in anatomic pathology[3], to scan the tissues and store them as image data. With the implementation of the Ventana scanner into the acquisition process it is possible to test its performance and its field of applications for the future simultaneously. The philosophy of the already implemented processes at the photographic laboratory is to use small reusable software tools which can be combined together in various ways. This approach makes it easier to change individual parts of a process in the future or to search for bugs by analyzing each tool separately instead of one huge application.
Based on these provisions the main task is to build up an acquisition process to scan the physical tissues (prepared on so called "glass slides"). A previous meeting was held to gather and specify the requirements from all stakeholders which should be met by the process later on. The requirements are listed as follows:

- Creating and setting up a database and its schema for persistently storing diagnosis data form research studies.
- Evaluating a PHP framework the inventory system will be based on and which provides the following features:
    1. Multilingualism
    2. Extensibility
    3. Database accessibility
    4. Performance oriented
    5. Security
- Import function to read diagnosis data from studies into the database. The data is given as XLSX-Excel files.
- Function to print bar codes for identifying tissue slides based on its study data.
- Background process for handling scanned images and registering them to the database.

[3] Product website: http://www.ventana.com/product/page?view=iscanht

- Background process for handling image annotations and registering them to the database.
- Export function to write data from the database to an XML file.

These requirements slightly differ in some points from the original project definition document as listed in the appendix 10.3. Firstly, we chose not to implement a data entry routine where external users could ask for scanning orders. During the project a concrete use case was not given and we decided not to produce functions for an uncertain future. However, instead of the order form we implemented a background process which can automatically recognize newly scanned slide images and annotations and register them to the inventory database. As shown during the scan process this feature is not only practical but also reduces the overall effort significantly. The second unimplemented requirement is the Kaplan-Meier curve of a study. During the project it appeared that the curve could not be calculated without the feature data from the image analysis, which is part of the follow up project.
Because of this reason we put back the requirement but implemented a simple correlation of patient data as a proof of concept for chart visualizations.

3 Acquisition Process

In the second chapter we talked about why we need an acquisition process and which requirements we need to met. The current chapter discusses about how these requirements could be implemented and how they could be combined together to a concrete process. Chapter 4 later on goes into details about the individual implementations discussed in this chapter whereas chapter 6 presents the results obtained. Finally, the reflection of the process and the modeling aspects can be found in chapter 7.

3.1 Storing diagnosis data form research studies

The database management system is already set by provision to be the Microsoft SQL Server 2012 whereas the schema to design has no restrictions. We firstly have a look at the patient data from the given Excel file to import:

```
ID                        4901
Round                     1
DoB                       26-Mrz-44
Age_at_study_entrance     54.63
Diagdat                   15-Jan-99
Quelle                    KSA
Bx_Nr_Aarau               B99.973-78
cTstage                   T1c
cNstage                   Nx
cMstage                   Mx
Biopsy_Gleason1           2
Biopsy_Gleason2           2
...
```

Listing 3.1: An example of a data record.
The data representation is transposed for better reading.

Each data record represents an anonymous patient. The ID is externally set and might be occur several times in our data set - which means it is not unique. Furthermore, it is possible as several records are connected to the same patient. However, this relation can not be derived as we can not clearly identify the records. Based on this knowledge we have to treat each record itself and import it as a new patient.

A simple database schema may put all data into a single table which attributes hold the record titles given in listing 3.1. However, this solution is not extensible and only supports Excel files with the exact formation as given above. If we would like to be more generally we can store the record titles and values in separate tables. With this approach we can handle all Excel files which holds titles in a first row and its values in the following rows. Our second version contains three tables for the patient's attributes, its values and a table relating the attributes and its values.
But is this already enough concerning the entities of the real world? In figure 3.1 we can see the physical representation of a tissue slice and its origin.

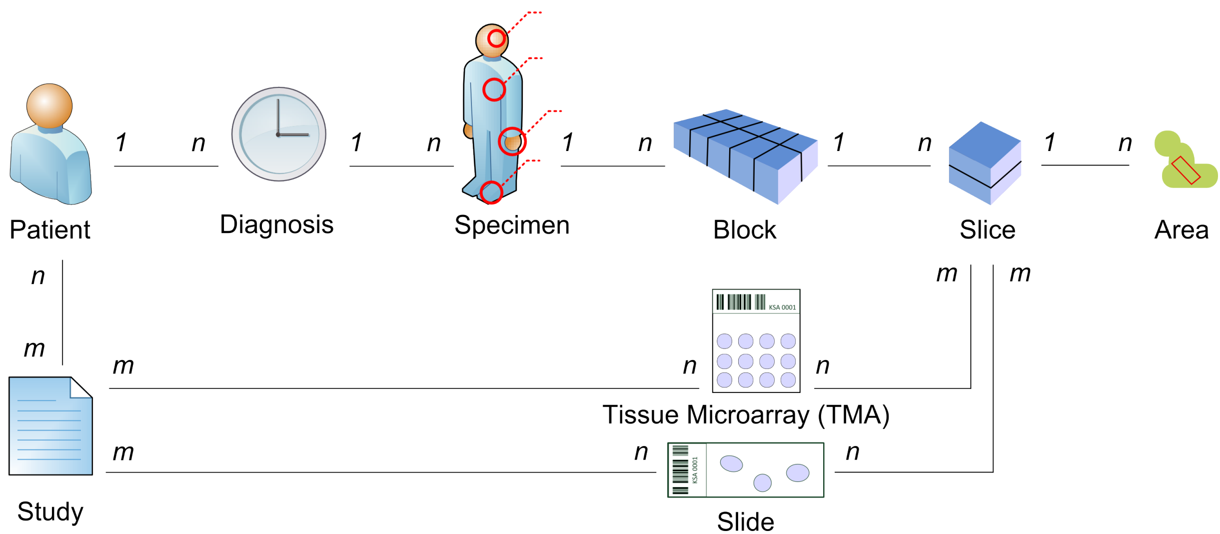


Figure 3.1: Origin of a tissue slice

In the real world we can find glass slides containing either single tissue slices, as given in our study from Aarau, or an array of tissues, a so called TMA. The tissue originally results from a specimen extracted on a specific diagnosis date. During the specimen we may have obtained several blocks of the tissue which later on are cut in small slices and prepared on glass slides. This slides finally can be scanned by a slide scanner. Again, a slide may contains several tissues with different areas of interest.

With the knowledge given we can take the next step and analyze the attribute "Bx_Nr_Aarau" from the Excel sheet given in listing 3.1. It contains a so called b-number which represents the year of a diagnosis and a range of IDs which identify the related specimens itself. Now we may ask what happens with our slide image we would like to map to its patient? As we can see a data record may contains more than one specimen and so may be related to more than on slide image. We have to find a third version of our data schema which also holds a table for the specimen, its blocks, its slices and its areas. Additionally, the slices have to be related with an image table which again is related to tumor annotations we would like to draw inside the images later on. The relation between the tables could be implemented in several ways. One way would be

to give each block an unique identifier and save its relations to the next upper entity, here the specimen, which again is related to the patient's diagnosis. We decided to use all this information directly as part of the unique identifier. So the patient's diagnosis, the specimen ID and the block ID are necessarily to identify a single block record. For this approach we need more attributes and the unique identifier e.g. for an area is much larger. However, it makes search operations e.g. for slices of a specific specimen much easier. Additionally, we have the possibility to set constants on the identifier so e.g. only slices with an already existing specimen could be stored to the database which guaranties consistent data. One last thing to mention: The attributes of the Excel's data records could be separated into two entities. On one hand we have the patient's attributes, e.g. the date of birth and sex. On the other hand we have the attributes of a diagnosis: e.g. the round, the diagnosis date or the Gleason score of the biopsy. The diagnosis date in our case is the date of the prostate biopsy. During a diagnosis we theoretically could extract more than one specimen from the patient which makes the diagnosis a superior entity. The final database schema can be found in figure 3.2 which also includes a table for bar code printers which we will discuss in chapter 3.4.

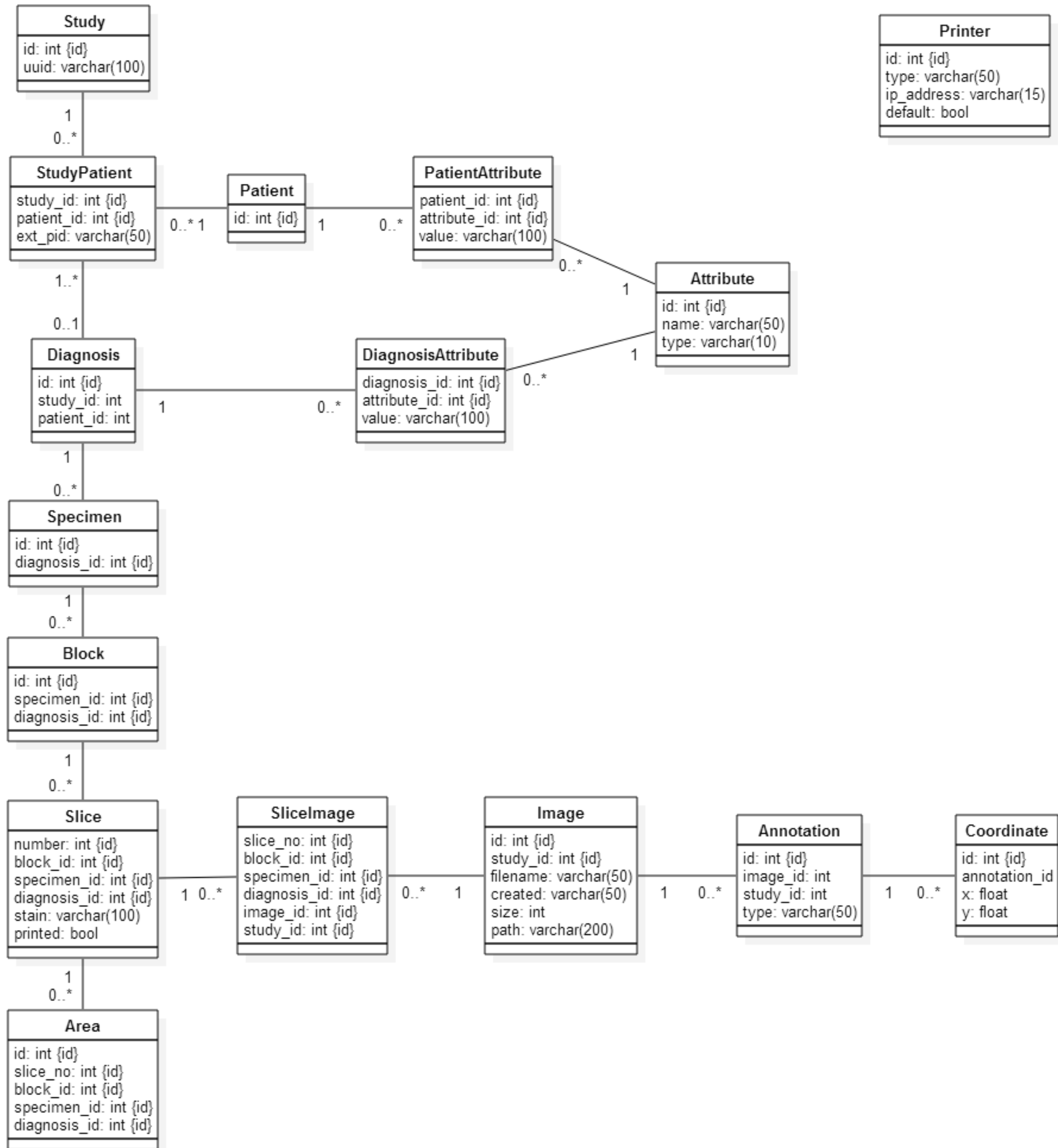


Figure 3.2: The database schema

3.2 The underlying PHP framework

To implement a web application we basically have two possibility. Either we implement all and everything by ourselves or we use an already existing framework which supports us with basic functions. As the implementation of a web application from the scratch needs a lot of time we decided to use a framework. We started with a very simple example[1] implemented by hand to get in touch with the basic structure of a PHP application which is described more in details in chapter 4.3. In a next step the right framework for the given problem has to be found. Roman Bolzern, which works in the field of web engineering since many years, recommended the Zend framework 2 which is already used by many famous web applications[2]. For a better overview we compared more frameworks such as Symfony2 (`http://symfony.com/`) and Laravel (`http://laravel.com/`). In the internet hundreds of articles can be found, each preferring another framework for good reasons. At the end we followed a statement from stackoverflow: “The answer as how to decide which is best is subjective. Pick the framework you best feel will meet your project needs.”[3]. As all three frameworks from above holds the requirements described in chapter 2.2 we decided to use Zend which was most familiar. Another important point was the efforts needed to make small modifications in the future. Here we see an advantage in Zend as we set all major settings in separate configuration files which can also be modified without the understanding of the whole source code behind. Some reviews call the Zend framework difficult to configure the experience however shows as it is relatively easy with an example given. Moreover, Zend offers a huge range of settings to choose which makes most wanted configurations possible.

A more detailed description about the use of the framework could be found in chapter 4.3.

3.3 Import diagnosis data from research studies

From the previous sections we know how our data to import looks like (see listing 3.1) and how the database schema looks like we would like to store our data in (see figure 3.2). A simple approach to import this data would be to step over each data record inside the Excel file and store it separately inside the database. As there is no standard for saving patient data in the field of pathology it is most probably as future data to import are not given as Excel files. In this situation we need to write a new importer for each new file format. But what happens if anything changes inside the import logic? In this case we have to modify each and all importer by hand. Furthermore, the code containing the commands to store the values extracted from the file to import is also redundant in each importer which is also not desired in software engineering. Therefore,

[1] `http://anantgarg.com/2009/03/13/write-your-own-php-mvc-framework-part-1/`
[2] E.g. Cisco’s WebEx or Centroy’s online collaboration tools
[3] `http://stackoverflow.com/questions/22675277/how-to-best-select-a-php-framework-laravel-symfony-zend-etc`

we firstly decided to write a parser which translates a data record from a specific file to import into an intermediate XML structure. In a second step we have a single importer for this XML structure. This approach also contains two more advantages. Firstly, to have both routines split up we can debug each routine individually and check its result for bugs which makes the error handling easier. Secondly, in chapter 2.2 we mentioned as small combinable software tools are preferred.

Another approach would be to use manual pre-processing. In this case we could only provide a single importer and only expect an Excel file to import. This might be easier for the software implementation but not practical in the future if we have a lot of studies to parse. Moreover, in chapter 2.2 we discussed about as we would like to automate each step of the overall process whenever possible which is not given here.

3.4 Print bar codes

The task of a slide scanner is to scan glass slides and save these images on the file system. This is also the only talk it does. We can not do any other automation here. If we use the standard settings e.g. of the Ventana iSacn HT it will save each image file with an incremental identifier starting from "1". As we do not have any information about the scanned image we have to open each file individually read its label information and assign it manually to a record inside the database. Obviously, this process is not acceptable if we have more than a dozen slides. We could automate this process by writing an image processing routine to analyze the slide label of the image and extract its written text. However, it will takes a lot of time to implement this routine whereas the Ventana iScan HT already hold such a build-in routing for bar codes.

To identify a scanned image we need not only an identifier which is set during the scan process but an own unique identifier which makes it possible to assign the image automatically to the database. In case of the Ventana iScan HT we have (only) one solution. We can use the bar code of each glass slide as a later file name which then can be proceeded by a background process which reads the file name and register it to the database. In this case we only need to write an additional routine which prints a bar code clearly assignable to a database record.

In this case a next point to discuss is the representation of the bar code. As discussed with Norbert Wey we have several constraints to meet:

- The bar code should be as short as possible to reduce errors while reading its code. The code should contains a maximum of nine characters.
- The bar code has to contains human readable information rather than only an unique identifier. This allows the easy identification of slides during daily business.
- By technical restrictions we can only print 1-D bar codes for now.

To identify a slide we need the following information: The b-number, the specimen ID and the numbers of the slices available on the slide. Unfortunately, if we put all those information together as a bar code it exceeds our limit of nine characters. So we decided to print this information as an additional label next to the bar code so the slide still could be identified. The bar code itself could now contains any unique identifier which could be related to a slice inside the database. We finally decided to weight the human readable information a little more high and define the bar code as following: <*patient id*>**S**<*image id*>. Now we can also read the patient's identifier out of the bar code. Only the image identifier is a number generated by the system which is only needed for the relation between the image and its entry in the database. To keep the image identifier as short as possible each study contains its own image identifiers starting with the number "1".

Finally, we need to discuss if such a bar code is clearly identifiable. Given, we have a patient ID which can be biunique related to a study we can unambiguously identify the image ID which again has to be unique for the related study. In this context each study can hold its own image IDs starting with the number "1". One problem occurs as shown in figure 3.2 where a patient theoretically could be participate to several studies. However, as we currently only have anonymous patient data it is not possible to have two studies with the same patient - otherwise the patient will not be anonymous anymore but identifiable. So to say, currently each imported patient gets a new unique identifier which makes the bar code clearly identifiable. This circumstance may have to be changed if we find patients participating more than one study in the future. However, we may also use scanners such as the Ventana iScan HT which can read 2-D bar codes then so restrictions such as the code length could be omitted. The final result of a bar code may looks like shown in figure 3.3.

Figure 3.3: A bar code printed from the PSR system

3.5 Register scanned images to the database

After scanning the images we have to register them to the database. The most trivial approach is to do so by hand which again violates our idea of making the process as much automated as possible. Another approach is to write a program/script which runs as soon as we would like to register new images to the database. Again, this needs a manual intervention by starting and stopping the application. Therefore, we decided to use a Windows service, a program that operates in the background like a Unix cron job, which scans for new images to register. But what are we doing if we find a new image to register? We could directly access the database and insert a new data record. However, with this solution we face two problems: Firstly, we need to have access to the database. Secondly, we need to handle the data access and possible errors inside the background process. If we use the already existing routine for importing the Excel data we would solve both problems at once as we could reuse the database connection as well as the data access routine.

The presented solution already fits our problem. However, we decided to separate the registration routine and the data access routine from the image file handling. Means, we implement a background program which is searching for new images scanned and transfer this information in an XML structure to another routine which finally register the data to the database. This approach again supports the idea of small software tools which are easy to debug, to test and to combine with each other.

When we scan image files we will face a folder full of images named by there bar code identification. If we have more than a few hundred files we need a structure to retain the clarity of our files. We decided to keep a folder for each patient named by its identification which contains all related image files. This approach also have another big advantage. The machine scanning the slides or even the user doing so may only have access to restricted or public network paths. However, maybe the organized files would like to be stored onto a more private file server which is not directly accessible. Therefore, the File Organizer could be installed on a machine with enough rights to access both paths, the one containing the new scanned image files and the path to organize the files in.

A last question might be why we do not save the image files directly inside the database but register them instead? On one hand, each file may fill more than 1 Gigabyte on the hard disk. Such images are not intended to be saved in relational databases. On the other hand, we would like to exam and annotate the images in the future which is easiest if we have them directly accessible.

3.6 Register image annotations to the database

The routine is quite similar to the one described in the last section. We also could present the same approaches and solutions. Our implementation is also straight forward. We again use the background process extended for searching not only new scanned images but also new annotations. In the same way we send the annotations in an XML structure to a separate routine which has access to the database and register them.

3.7 Export data from the database

The data export is the last routine to discuss about. Is should be possible to export data from the database so we can analyze them later on in an external tool. This routine is especially interesting after adding additionally information from the image analysis to the scanned images and export them again. We could analyze this information later on in an easy way by using corresponding tools such as Matlab or Excel. A first question to answer is the format we would like to export the data into. Most common formats are e.g. CSV or XML. As the data stored inside the database represents a connected and cycle-free graph we would have many redundancy saving the data into a 2 dimensional format. We take the specimens of a study as example. The first two attributes of a data record would be the study ID and the patient ID which are connected to a specific specimen. Obviously all data records will have the same study ID and many records will have the same patient ID as one patient is most probably connected to several specimen IDs. For this reason we decided to use an XML structure which can easily map such data structures. Moreover, the XML file is easy to read from most common software. Microsoft's Excel for example has a build-in assistant which allows to import and analysis the XML structure using only a few actions.

A more detailed description about the export routine could be found in chapter 4.4.1.

3.8 Bring it all together

We already discussed the different routines of the overall process but not yet the connections in-between. We could write each routine as a separate and standalone application. The disadvantage on this approach is e.g. as we need to write the database connection and the data access logic for each routine which means a lot of redundant code. Furthermore, each time we start the whole process we need to verify as each routine can communicate with its related routines. Another decision to make is the programming language to use as all routines mentioned could be implemented in both PHP as well as C#. We decided to implement only the background process for finding new scanned images and there annotations in C# as the .NET environment perfectly fits for the requested tasks. In C# it is very easy to get access to data and read its values such as the modification date etc. or to create an XML structure. Furthermore, the IT infrastruc-

ture already holds a tool to manage C# background processes where we could register our new background process to. Later on we will call this tool the "File Organizer". All other routines will be implemented in PHP and combined to the web-application called "Patho - Study and Research". This solution allows us to implement the database connection and the data access logic as a single module which could be reused later on by all others. Additionally, an end-user would like to interact or at least get informed about the process's progress which means we need a user-interface. Here we finally can bring in our first discussion where we decided to use the Zend framework as the basic for our web application. Zend is strongly known for such tasks and brings already a huge set of functions for creating user-interfaces, events, etc. However, the represented arguments could still be covered by C#. The last and most important reason for our decision, using PHP, is the aspect of accessibility. Inside the university hospital of Zurich we can find Windows, Mac as well as Unix systems. If we would like to implement these routines in C# we have to take care as we can install the applications on all systems. Furthermore, the installation of the routines are mandatory to run the process. However, if we offer our routines as a web-service everybody in the hospital could get access to it independent from its operating system and without worrying about permission restrictions. Figure 3.4 shows the elaborated overall process and its connections between the individual routines.

We start with the export of the anonymous study data from an external data source. After translating the exported data into a for our system readable format we can import the data into our own database. Those data may also define the slides related to the study and there specific patients. Based on that information we can print bar codes related to physical slides and sticking them on. Afterward, the Ventana iScan HT can read the bar codes and saves the data with an unique identifier each related with a database entry. As soon as the scanned image is available a background process copy the file into a structured file directory and register the new file to the database. If later on, a person annotates a scanned image the background process also hand in the annotation data to the database.

The current process only implements the Ventana iScan HT for acquiring image data. As shown in figure 3.4 the process can easily be extended so image data from other slide scanners, e.g. Hamamatsu NanoZoomer or Zeiss Axio Scan Z1, can be involved. We only need the information, whenever a new image is scanned. This information may be available as a text file and so, just needs a translation into an XML representation we can send to the web-application. Still, the remaining process do not change.

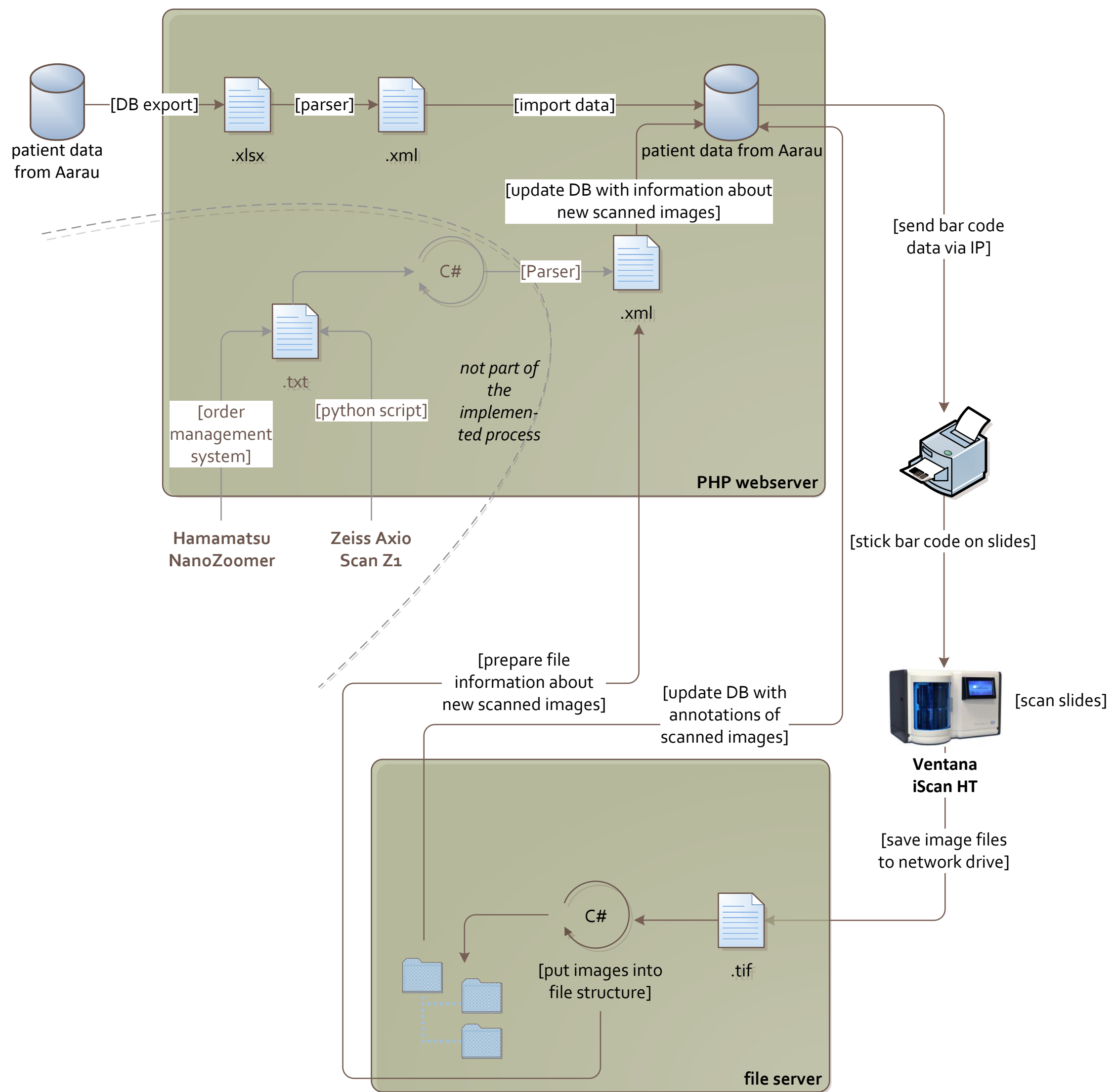


Figure 3.4: Process overview

4 Software Architecture

Chapter 3 described the overall process which is now presented more in detail in the following chapter. The focus for this chapter lays on the software engineering of the routines described earlier on. We first give slight introduction on the applications participating the process and later on describe its architecture. The source code itself is listed in the appendix 10.3 and available as attached material, documented by appropriate comments.

4.1 The "Patho - Study and Research" system

We start with the PSR system which represents a web application running on an Apache HTTP server and the Zend framework 2 (ZF2). Its setup can be summarized as follows:

<table>
<tr><td>Apache HTTP server</td><td>Version 2.2.25
Activated modules:
- php5_module
- rewrite_module
Mapping configuration for mime_module:
- AddType application/x-compress .Z
- AddType application/x-gzip .gz .tgz
- AddType application/x-httpd-php .php
PHP configuration:
- PHPIniDir "<PHP path>"
http://httpd.apache.org/</td></tr>
<tr><td>PHP</td><td>Version: 5.4.29
Activated extensions:
- php_sockets.dll [Socket extension, needed by ZF2]
- php_pdo_sqlsrv_54_ts.dll [MS SQL driver]
- php_sqlsrv_54_ts.dll [MS SQL driver]
- php_gd2.dll [Image extension, needed by PHPlot]
http://php.net/</td></tr>
<tr><td>Zend framework 2</td><td>Version 2.3.0
http://framework.zend.com/</td></tr>
<tr><td>PHPExcel</td><td>Version 1.8.0
https://phpexcel.codeplex.com/</td></tr>
<tr><td>PHPlot</td><td>Version 6.1.0
http://www.phplot.com/</td></tr>
</table>

Not to blow up the documentation we will not going into details about how to setup an Apache HTTP server with PHP nor how to configure it. This subject is very well known and described by many brilliant documentations. The official ones can be found as following:
`http://httpd.apache.org/docs/`
`http://php.net/manual/en/`

We already discussed why we choose PHP and why we choose the Zend framework in chapter 3. As a short secularization we would like to list the advantages using those technologies a little more in detail:

- Easy access to the application from everywhere inside the hospital without installation troubles
- Platform independent access
- Well-established technology, widely used in enterprises
- Easy and adaptable access to databases
- The Zend framework supports multiligualism
- The Zend framework is build up and supports a module oriented approach
- The Zend framework supports secure web application[1] as well as performance optimization
- The Zend framework supports innately the MVC pattern
- The Zend framework supports hierarchical organized configurations and routings
- The Zend framework integrates Bootstrap[2] which provides a consistent look and behavior for latest desktop browsers
- Last but not least, the Zend framework is ideally integrated in the Eclipse PHP Development Tools (PDT) which contains the most commonly used integrated development environment (IDE)

[1] For further details please visit `https://www.owasp.org`
Zend was also mentioned in `https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet`
[2] For more details please visit `http://getbootstrap.com/`

4.2 The File Organizer process

The second part of our overall system was implemented using C# for a background process, under Microsoft Windows also known as Windows service[3]. For the implementation we use Microsoft's Visual Studio 2012 with the .NET framework 4.5.

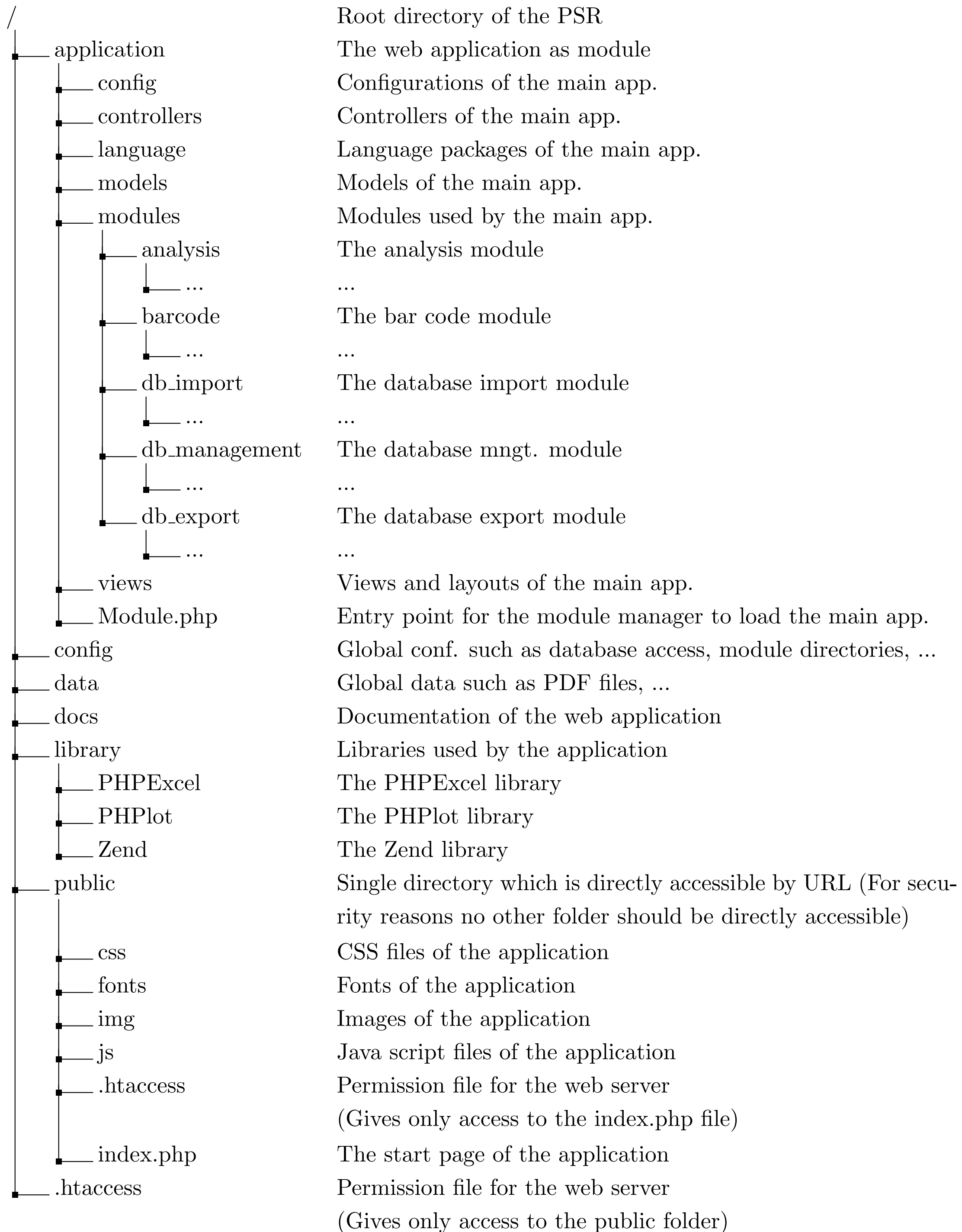
The discussion why we choose C# could be found in chapter 3.8.

4.3 Physical representation

The file structure of a general Zend application Before we go into any implementation details we will have a brief look about the file structure of the PSR and File Organizer. First, let us have a look about the structure of a general Zend web application. This also underlays several basic concepts we will use later on. As the framework does not expect a strict structure we could organize the application at free will. However, Zend offers already a so called skeleton application[4] which includes a modular MVC structure and many pre-configurations out of the box. The skeleton application is meant to be used as a basic template we can build our application with. We decided to structure our PSR system using the skeleton application as it saves a lot of time, it is easy to use and already combines many best practices. Additionally, many articles from the Zend documentation are based on the skeleton application.

The recommended project structure using the Zend framework is described and well-founded in the Zend documentation at `http://framework.zend.com/manual/2.0/en/ref/project.structure.html`. We slightly adapt this structure to fit our requirements as shown below:

[3] `http://msdn.microsoft.com/en-us/library/d56de412(v=vs.110).aspx`
[4] `https://github.com/zendframework/ZendSkeletonApplication`

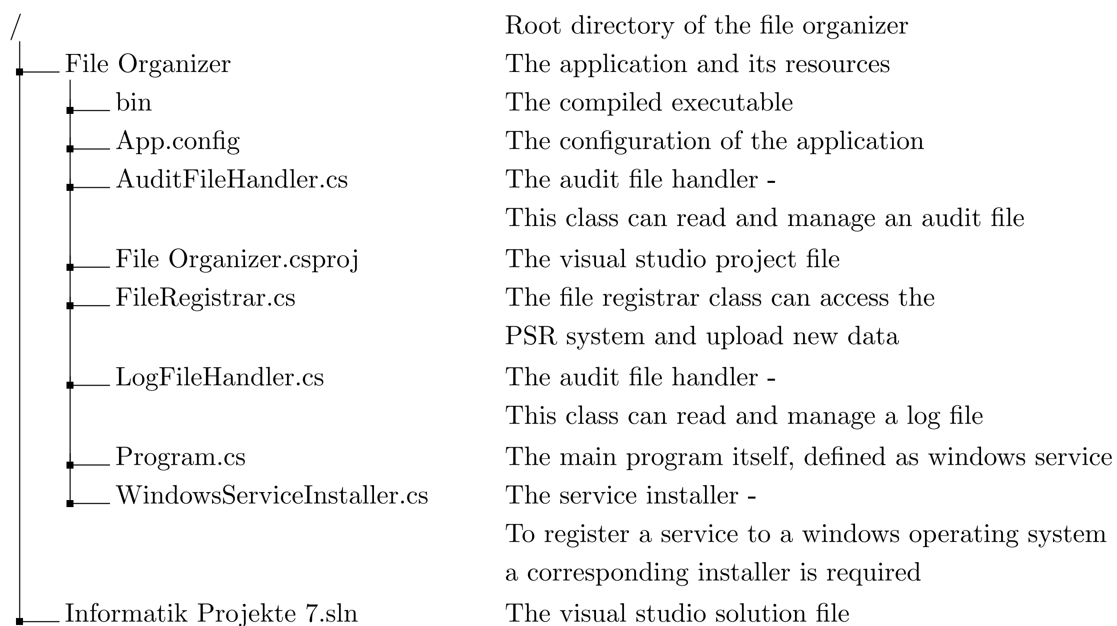


We see as the main application itself represents a module which again contains submodules, described later on. A valid Zend module only needs a “Module” class stored inside the “Module.php” file (see /application/Module.php from the above file structure). When the new module is loaded into the framework of Zend several actions take

place, depending on the module's setup[5]. It may automatically loads the corresponding classes with Zend's autoloader or defines URL routings and translations. Another note to remark, the only file directly accessible is the /public/index.php file, defined by the .htaccess configuration. The index.php also loads the ZF which again loads all registered modules and routing definitions. Only by the routing definitions a user is allowed to get access to other resources which increases the security of the application tremendously.

> The ZF itself is very huge and contains many more things to talk about such as the service manager, the translation suite or the autoloader[6]. For more information we refer to the official Zend manual at `http://framework.zend.com/manual/2.3/en/index.html`

The file structure of a general windows service Next, we have a look at the general file structure of a windows service. Again, this also underlays several basic concepts we will use later on. The structure was taken from the TechPro article "Creating a Simple Windows Service in C#"[7] which is based on the official documentation of Microsoft[8]. Because of its simplicity and clear description we did not search for other alternatives but implemented the solution straight forward. Its project structure, as for most Windows services, is given below:



[5] More information can be found at:
`http://framework.zend.com/manual/2.3/en/user-guide/modules.html`

[6] `http://framework.zend.com/manual/2.3/en/modules/zend.service-manager.intro.html`
`http://framework.zend.com/manual/2.0/en/modules/zend.i18n.translating.html`
`http://framework.zend.com/manual/2.3/en/modules/zend.loader.standard-autoloader.html`

[7] `http://tech.pro/tutorial/895/creating-a-simple-windows-service-in-csharp`

[8] `http://msdn.microsoft.com/en-us/library/d56de412(VS.80).aspx`

Each windows service consist of two components, the service and the corresponding installer. To register a service to a windows operation system the installer is needed to create new registry entries, to define the account under which the service will run and the display name of the service. The service itself will then be embed in the windows service environment and can be started by the windows service manager as needed.

A more detailed description about the installation process can be found inside the user manual of the PSR system. For further technical readings about windows services we refer to the corresponding literature:
`http://msdn.microsoft.com/en-us/library/d56de412(v=vs.110).aspx`
`http://tech.pro/tutorial/895/creating-a-simple-windows-service-in-csharp`

4.4 The 4+1 Architectural View Model

The 4+1 view model was designed by Philippe Kruchten 1995 and describes the software architecture of a system from different point of views [4]. As his model on one side is very easy to understand and on the other side provides a clear structure we will adopt it for the upcoming subsections. The following table gives a short overview of the views involved.

View	Description
Logical view	Describes the functionality and services for the end-user.
Development view	Describes the system's architectural approach and modules.
Physical view	Describes the distributed components and there connection.
Process view	Describes the communication between the components.
Use case view	Describes scenarios based on the given architecture and its design. This view is also called the "plus one" view.

We start with the logical view which takes a look into the functional requirements and there implementation and go on with the development view which gives a complete overview of the software. This order was chosen as the first part is more close to the user's view and gives an easier access to the architecture. Later on, we talk about the cooperation of the processes and there communication. Finally, we will illustrate some use cases bringing all previous sections together.

4.4.1 Logical view

A functional introduction to the PSR system After getting an overview of the given applications we will go on with a top-down approach starting with the available functions already discussed in chapter 3 - best seen in the menu structure as given in figure 4.1. The menu structure was designed trying to be user friendly and intuitive. To make the menu easy understandable the individual steps of the process were visualized with simple icons which finally holds the concrete functions a user could run. While presenting the

application the menu seems always easy understandable. However, no usability tests were made explicitly.

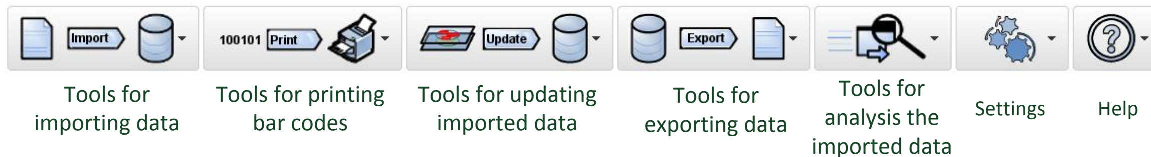


Figure 4.1: The menu structure of PSR

All the functions are directly accessible by the corresponding menu entries or related URLs as shown bellow:

Menu entry	URL (histodb2.usz.ch/patho_study_research/...)
Import data	/dbimport/main
▷ Parse XLSX to XML file	/dbimport/parse/upload
▷ Import master data	/dbimport/import/masterdata
Print bar codes	/barcode/main
▷ Show all bar code printers	/barcode/printer
▷ Create new print job	/barcode/job/print
Update database	/dbupdate/main
▷ Update metainfos for scanned images	/dbimport/import/scandata
▷ Update annotations for scanned images	/dbimport/import/annotationdata
Export data	/dbexport/main
▷ Export study data	/dbexport/xml
Analysis data	/analysis/main
▷ Correlate data from study	/analysis/diagram/correlation
Management	/application/main
▷ Settings	/application/settings
▷ Remove study	/dbmngt/study/remove
Help	/application/help/main
▷ User Manual	/application/help/manual
▷ About Patho Study Research	/application/help/about

We go on with the implementation details of each function starting with the data import workflow which is visualized in figure 4.5a. As discussed in chapter 3.3 we firstly parse the Excel file into an XML file which later on is imported to the database. In a first step the user has to upload its Excel file. In a very simple solution we could process the file without any user interaction. However, this solution is not flexible and would only works with the data from Aarau. But, if we present a user interface which settings

we should offer? In chapter 3.1 we talked about to distinguish between patient data and diagnosis data which has to be a mandatory setting. We realized as well as the attribute “Bx_Nr_Aarau” is not normalized and contains the b-number as well as the range of specimen IDs in one field. As a solution the user could simply choose a field which splits the b-number from the specimen ID by a fix defined pattern. However, we expect to find more denormalized data in future studies. Therefore we choose to offer a generic way to split a field in a customized way. But how could we cover as many splitting criteria as possible? We could use the same methods as the Excel assistant for importing text data whereas a specific character or a specific number of characters are chosen as splitting criteria. However, this criteria are very limited which brings us to regular expressions (RegEx) [5, p.15–23]. This method is not as easy to understand as the one from Excel but much more powerful. For simplicity we add a check button which visualizes the splitting criteria so its effect could be directly evaluated. A second setting to choose is the field containing the range of the specimen IDs. This range is split up and for each ID given we generate an individual specimen. Finally, the external patient ID emerge to be an important field which should be able to specify as well. This ID will be handled separately and be saved in an individual field inside the database for further use. We could think about many more settings such as data records we would like to skip or additional fields to insert. However, this actions could be done more easily as a pre-processing step in Excel so we do not cover this settings in the PSR system. After parsing the Excel file into an XML file an import function could store the data to the database. For simplicity and debugging reasons we offer the download of the intermediate XML file. This step is not essential but practical as we can verify the parsing process before we start the import routine. After importing the XML file we visualize a short summary of processed data and errors or warnings if available. This supports the user friendly aspects of the application and informs the user what happened during the import process.

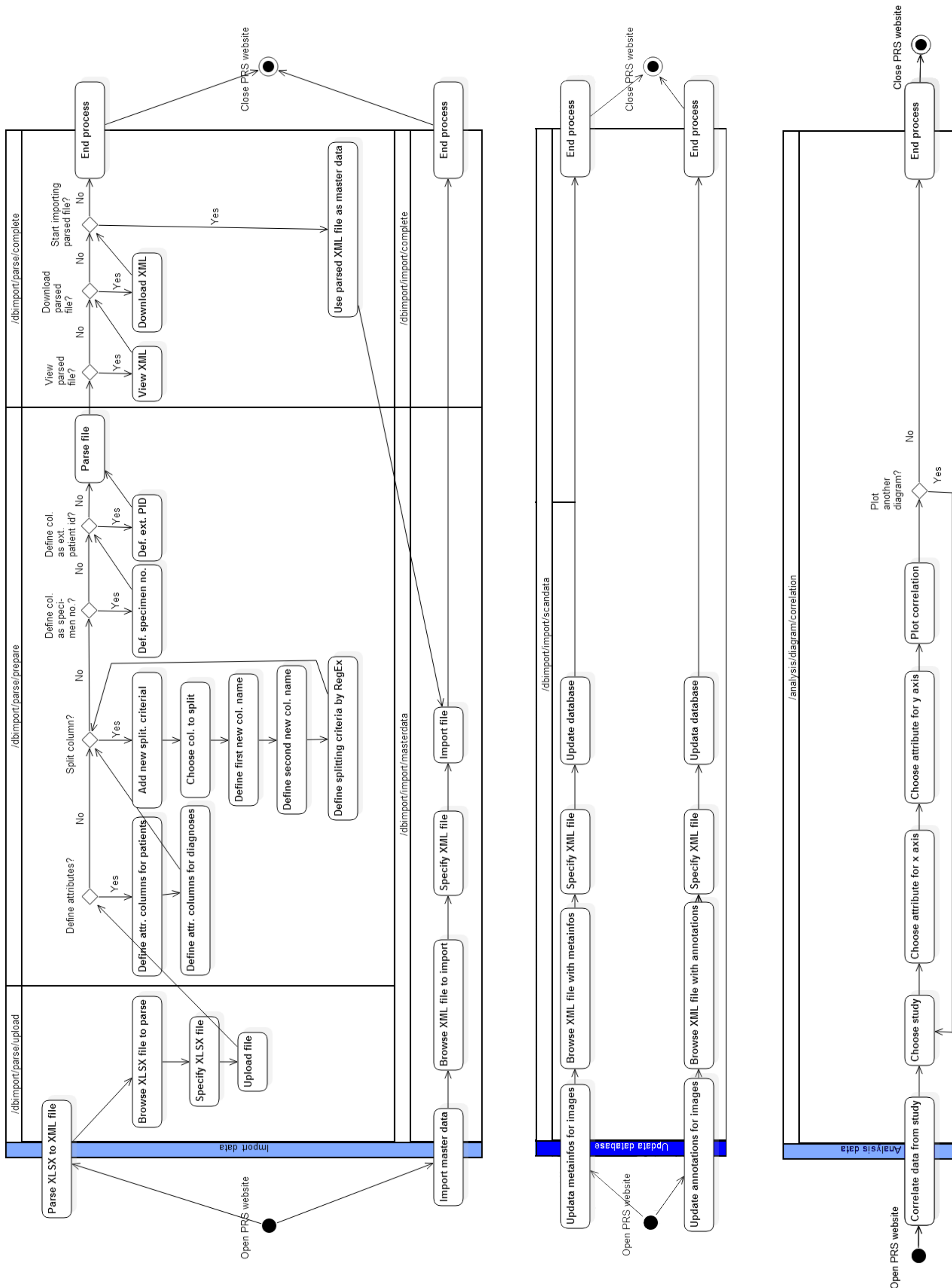
The next workflow covers the bar code printing routine as visualized in figure 4.2d. The representation of the bar code was already discusses in chapter 3.4. Now we would like to discuss the implementation aspects of the routine. In the most simple form we need an IP address of the bar code printer to use and the data to print. As we do not want to specify the IP address of the printer each time an easy management interface was implemented. This interface allows the user to add, delete and edit bar code printers. As during the scanning process we only used a single printer this IP adress could also be hard coded into the source code. However, this solution is hard to understand if we would like to extend the application later on and is not practical if we once use others than the currently bar code printer. After choosing the bar code printer we have to specify the label and bar code to print as discussed in chapter 3.4. Hereby, we need somehow to define the slices on the slide to scan. To clearly identify a slice we need its study, its patient, its specimen and its block as shown in figure 3.1. Another solution would be to use the b-number and the specimen number as this information also clearly identifies the slide. To print the bar code we have now two possibilities. Either we print a bar code for all imported specimen numbers or we print the bar code for each given

slide manually. As we have much more generated specimens than slides we decided not to print out all bar codes automatically but by hand. Furthermore, we do not know how many and which slices are available on each slide so we anyway need to add this values to the database manually.

After printing out the bar codes and scanning the slides the next step would be to register the new images and its annotations on the PSR system as visualized in figure 4.2b. As discussed in chapter 3.5 the data to register will be send by the File Organizer which we will look into details later on. Again, we have several options for implementing the registration service. One solution would be using a REST interface for handling the uploaded data [6, p.5–6]. The REST technology would fit the requirements. However, on one hand the implementation would need a lot of effort and on the other hand we actually do not need a REST interface for any other workflow inside the process. Another solution would be to offer a web-form where we could register the new images and annotations to. With this solution even an end-user could easily register images and annotations manually to the PSR system. The disadvantage is as the File Organizer has to open an individual HTTP connection for each image and each annotation to register. Because of this reason we decided to offer a web-form to upload an XML file containing all relevant data such as a list of new scanned images. This XML file however does not necessarily need to exists on the hard disk. The File Organizer for example creates the XML structure on the fly and directly send it to the web-form never holding the data persistent.

The final step of the overall process is the export routine as visualized in figure 4.2e. As discussed in chapter 3.7 we will export the data as an XML file. But which data should be exported? As we would like to analyze image features in the future those information would be most important to us. In a first step the user should be able to chose a study to export and all the wanted attributes from the data originally imported. Afterward he should choose how many details he would like to see in the export such as the specimen, the block and the slices an image is based on. For the sake of completeness the user can even choose to export the registered images with there data. A more simple routine could export all this data without offering any interface to the user. This solution is also possible but we have to delete all unwanted attributes in a post-processing step manually from the XML file later on.

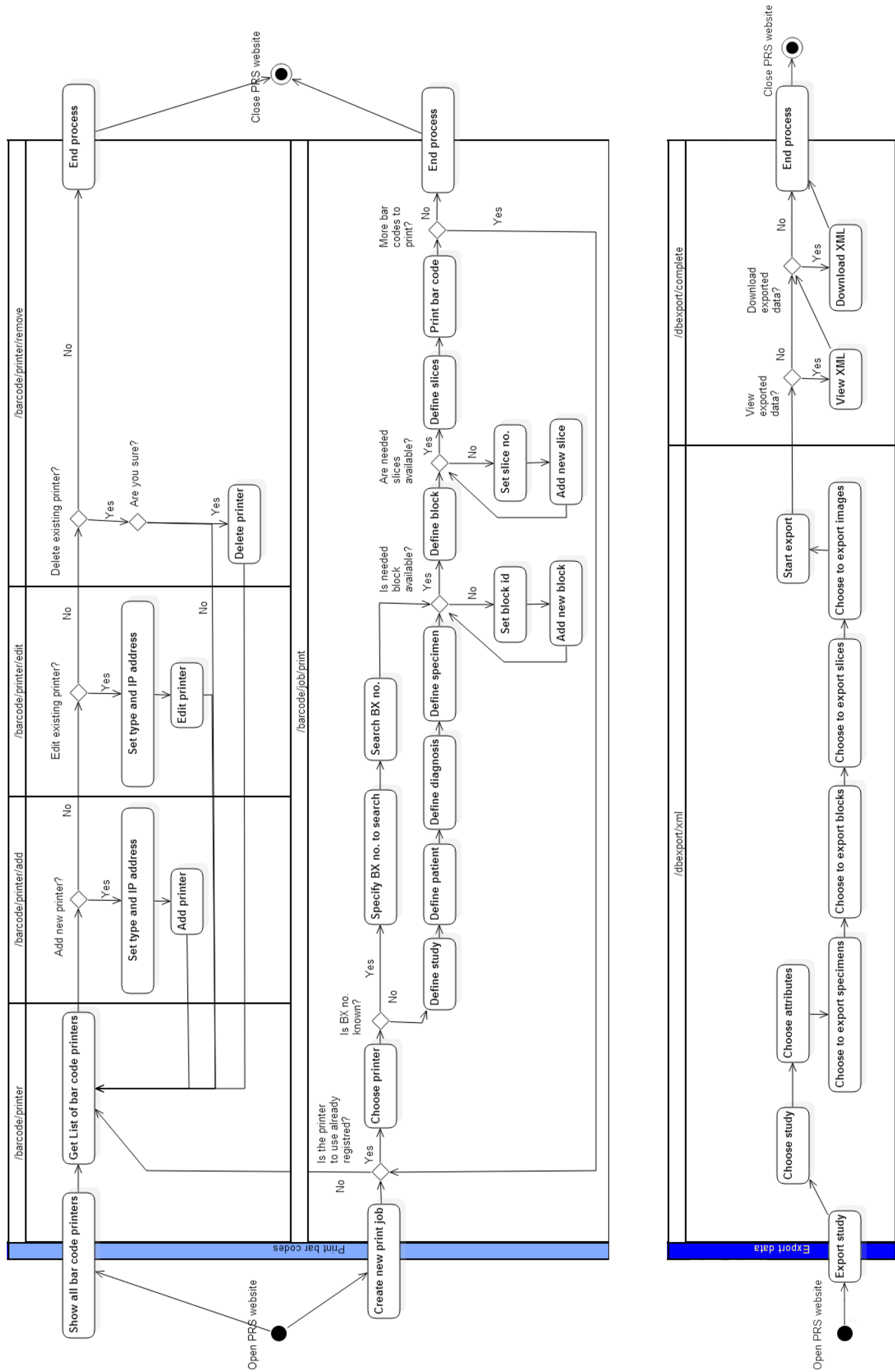
A last workflow to mention is the data analysis. As discussed in chapter 2.2 this workflow is implemented as a proof of concept for displaying the Kaplan-Meier curve and is visualized in figure 4.5b. We currently only support a scatter plot where a study could be chosen as well as two imported numeric attributes which are correlated later on. As this is only a proof of concept and not mandatory to the overall process a simple routine was implemented which could be easily extended. However, for time reason no alternatives are evaluated here.



(a) Import data workflow

(b) Database update workflow

(c) Data analysis workflow



(d) Bar code printing workflow (e) Export data workflow

Figure 4.2: Activity diagrams of the functional workflows of the PSR

The activity diagrams are generally easy to understand but may not give all information needed. Two things we have to discuss here: Firstly, if we parse an XLSX file as seen in the import data workflow we have to generate a new XML file on the server side. But this means as the server will store more and more files after a certain amount of time. To avoid this unwanted effect the PSR system checks each time the user calls the "/dbimport/parse/complete" view if XML files older than two hours exist and if so delete them automatically. This behavior also have the advantage as if we would like to go back in the browser history shortly after parsing an XLSX file we can still access the XML file during two hours. The same approach was also implemented for the XML file in the export data workflow. Another point to talk about is the print bar code workflow. As shown we can define new blocks and slices as needed in a simple way. But after defining e.g. 6 slices we would like to partitioning the slices in a unrestricted way, e.g. [(1), (2,3), (4,5,6)]. To meet this requirement we generate a new image entry related to the chosen slices inside the database each time a bar code is printed. With this approach a person can enter all available slices into the system without printing any bar code and define its assignments to the physical slides as needed in the future. Moreover, if we repartition the slices in the future we could keep the database model and data as given and directly print a new bar code which only creates a new image entry related to the already entered slices.

After talking about the functions given by the system we would like to go a little deeper and present its layer-architecture. At this point, we do not present a whole class diagram as it will contains to many information and do not support the understanding of the architecture. Instead, we will focus on the representative function "Management/Remove study" and the involved classes as present in the following diagram 4.3.

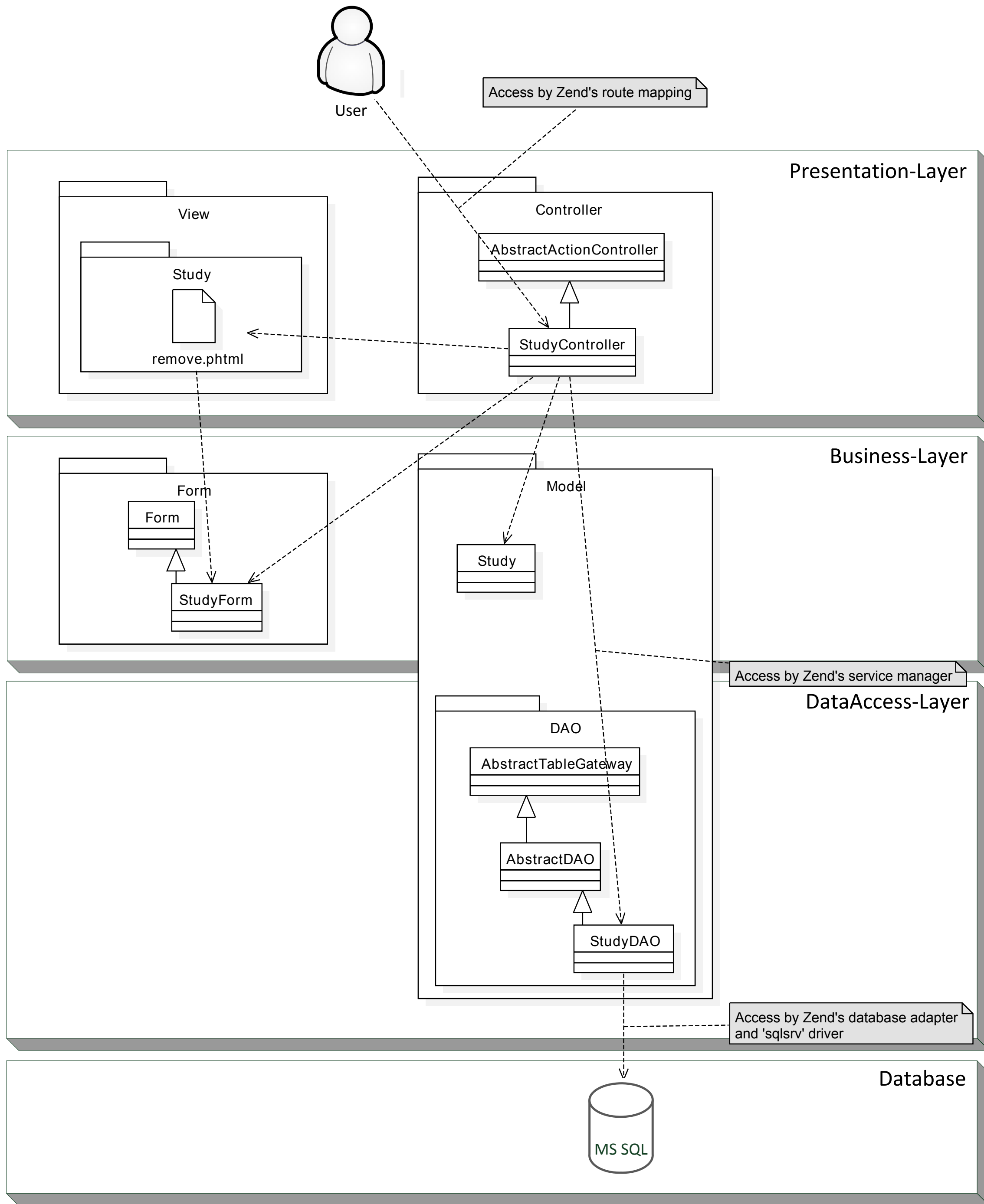


Figure 4.3: Representative extract from the class diagram

When an end user calls an URL of the PSR system, directly or by using a link, he will be redirected to the corresponding controller, e.g. “StudyController”, based on the route mapping definitions stored inside the Zend framework. In a first instance, the controller will instantiate a new from, e.g. “StudyForm”, and return the corresponding view, e.g.

"remove.phtml", and its bounded form as a response to the end user. Based on the form the user can specify his demand, e.g. specifying the study to remove. After submitting the form it will be send again to the origin controller which executes the demand and inform the user about the result. If the demand affects the database the controller will instantiate a new business model class containing the data to manipulate and hand it over to the corresponding data access object (DAO) which will finally forward the command to the database.

In the diagram shown above there are some points to discuss concerning the layer design of the architecture before we have a look about more details. Firstly, the model-view-control pattern seems to be just partly adapted to the architecture. This is not just given by the Zend framework but also in the logical distribution of a web application. Let us take Java as an example where user A manipulates data in the business model. By an observer approach changes could be notified to the controller which finally updates the Swing interface of a user B which is currently working with the same data. Looking at a web application we cannot update a view as the view will be send once and forever as a response to a user and we do not have any handle to the view anymore. However, it is possible using Javascript to dynamically load or update data in the business model. In this case we will send an asynchronous Ajax request to the web application which again will be handled by a controller. So to say, it is not possible to get direct access from a view to a business model by the logical restriction of the technology. A second aspect to mention are the data access objects. In the standard approach of Zend they are part of the model package and so called "Tables" whereas in Java and .NET we talk about data access objects. As the name "Tables" is confusing and not wildly used we followed the standard approach of Java and .NET. We can find two implementation of the DAO where they are separated in an own data layer, e.g. by a dedicated module or we use data transfer objects which are accessible by all layers inside the architecture. As a trade off, the data objects in the following are stored inside the model package which is a logical place as the model classes and DAO classes are strongly connected to each other. But, we do not separate the data access objects to a separate module as this will make the whole system more complicated to understand. If the application grows strongly in the future it might be an idea to encapsulate the data access layer to a dedicated module.

A functional introduction to the File Organizer In the same way as for the PSR system we would like to give a short overview of the available functions from the File Organizer. This time we do not have any menu structure but only a single service entry point. The user can define the behavior of the service by parameters listed later on. The parameters can be set directly as start parameter by the Microsoft management console and the service snap-in. This way is recommended for debug or test cases.

<table>
<tr><th>#</th><th>Parameter name</th><th>Description</th></tr>
<tr><td>1</td><td>Source Path</td><td>The path to search for new scanned images in.</td></tr>
<tr><td>2</td><td>Destination Path</td><td>The path to copy new scanned images to.</td></tr>
<tr><td>3</td><td>File Register URI</td><td>After copy new scanned images they will be registered to the PSR system by using the hereby given URI.</td></tr>
<tr><td>4</td><td>Annotation Register URI</td><td>The file organizer not only checks for new images but also for new or edited image annotations. Those annotations will also be registered to the PSR system by using the hereby given URI.</td></tr>
<tr><td>5</td><td>Network User Name</td><td rowspan="2">If we use UNC paths we may need additional rights to get file access than the account which currently executes the service. This can be achieved by setting a specific user name and password here. ATTENTION: This parameters are currently in experimental status and only for test cases. For security reasons do not store login data in plain text.</td></tr>
<tr><td>6</td><td>Network Password</td></tr>
</table>

If you do not provide any parameters the service will get access to the windows registry and read the corresponding values there. The registry path is listed below. Also there are many ways to pass over the parameters like using a configuration file we adapted the intended approach from Microsoft using the Window's registry[9]. A nice but not implemented tool would be a graphical interface to edit the parameters from the registry. Moreover, the tool could also provide additional information such as the log and audit file discussed later on.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\
    ↪Patho.Study.Reserach - File Organizer\
```

Given the above parameters the File Organizer copies new images, expected as TIF files, to an organized structure inside the destination folder. Hereby, a file's name has to hold the following pattern: $<$ *patient id* $>$**S**$<$ *image id* $>$, e.g. 1234**S**56. During the copy process the File Organizer creates a patient-folder named by its ID inside the destination folder and copies the image file renamed with only its image ID into it. Figure 4.4 visualize the copy process.

[9] `http://technet.microsoft.com/en-us/library/cc959506.aspx`

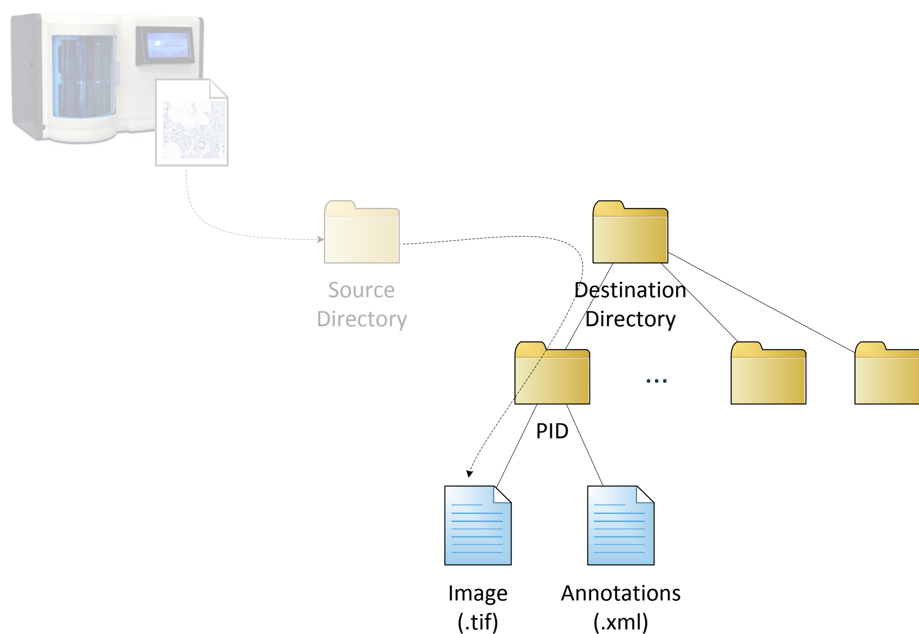
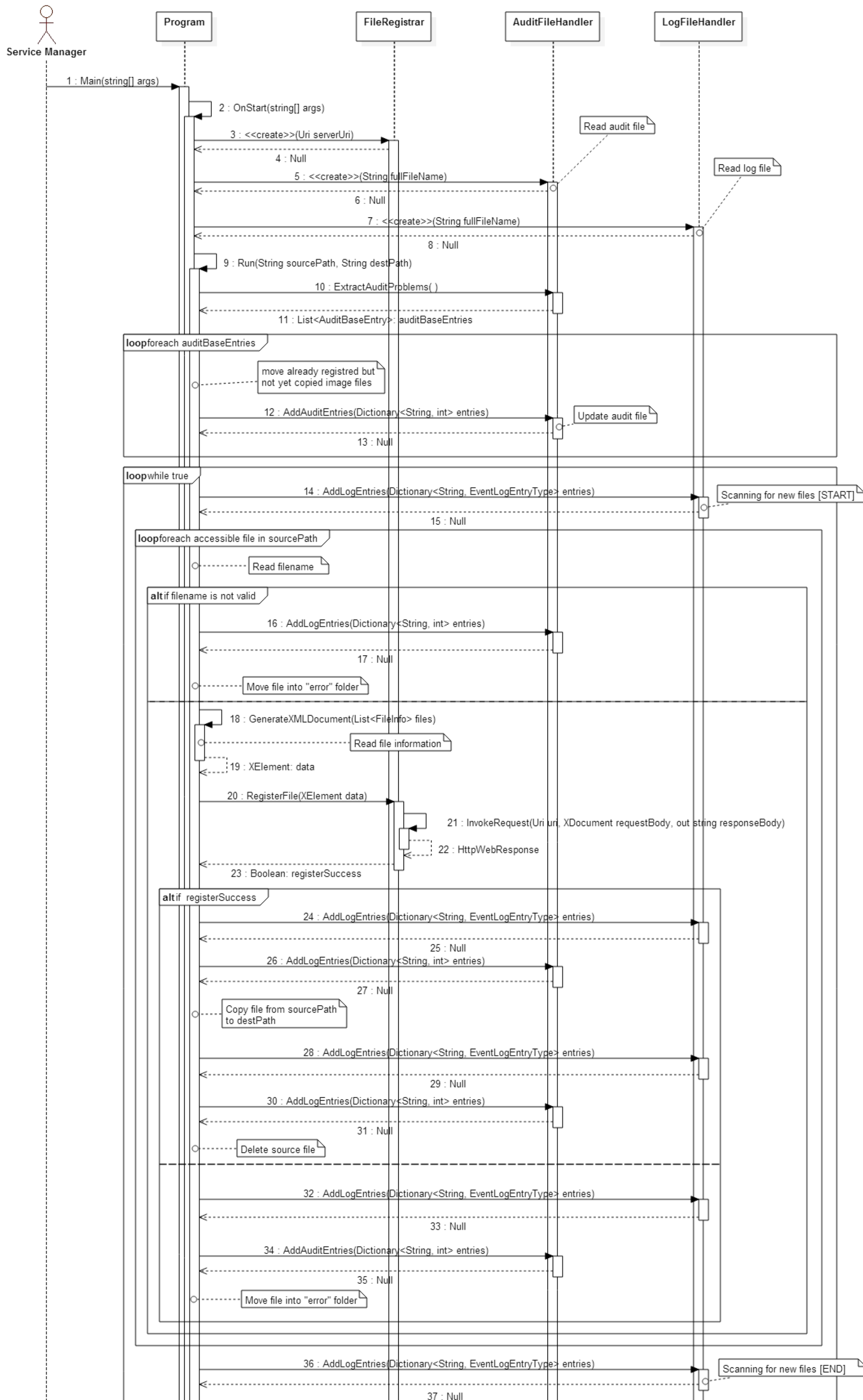


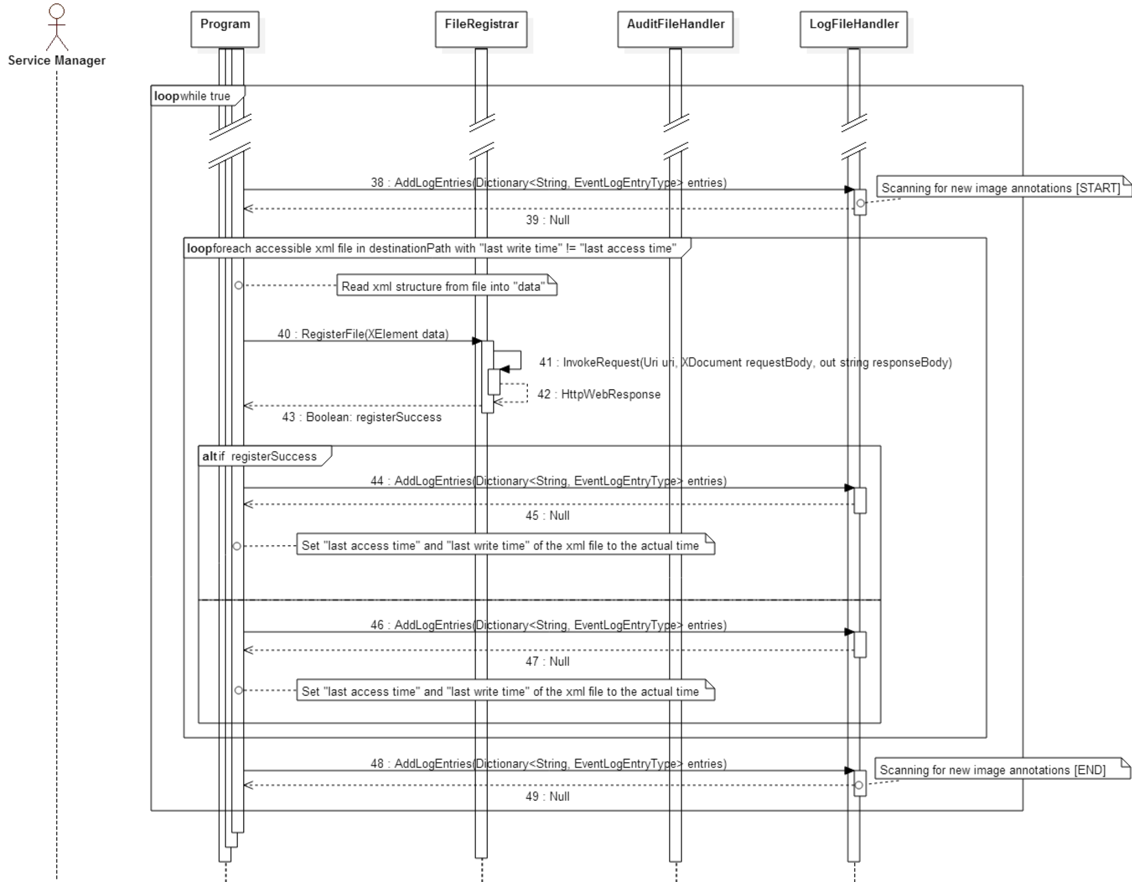
Figure 4.4: The file structure provided by the File Organizer

Files which do not represent a TIF container or do not follow the above pattern will be copied into an error folder so they will not be proceeded each time the File Organizer search for new files. Files copied into the error folder have to be handled manually.

For a better understanding of the File Organizer's process the visualization of its sequence diagram is given in figure 4.4.



(a) Service initialization and main loop for register new scanned images



(b) The loop for register new image annotations

Figure 4.4: Sequence diagrams of the file organizer

4.4.2 Development view

In the previous section we had a detailed look into functions of the PSR system and the File Organizer which included the collaboration between several subsystems. Now, we would like to take a step back and have a look about the architecture of the whole system as such.

The modular architecture of the PSR system The modular structure of the PSR system is based on the general approach of a Zend application and is also implemented in the basic skeleton application (see chapter 4.3). However, we do not need to follow this approach but implement either a structural approach or using only one single module containing all available functions. Both approaches are widely used for small and uncomplex web-applications. However, using modules makes it possible to encapsulate complex software components in well-arranged units we can combine later on which supports exactly our approach as discussed in chapter 2.2. Also in the future it is easier to debug or extend modular applications as we can focus on a single module which is easier than to deal with the whole application at once[10].

As we already know about the overall process and its needed functions we now have to encapsulate them into modules. We could use a separate module for each function which seems to be a little odd as the registration routine for new images or the registration routine for image annotations are very similar to each other which favor to put them together into one module. We orient ourselves on the requirements from chapter 2.2 which already provides a structure for the functions. Figure 4.5 shows the layer build-up of the system containing five modules where two modules contains third party libraries, the PHPExcel as an interface for Excel files and PHPlot as a plotting library for diagrams. PHPExcel was chosen because of the big amount of resources available on the web and the features provided which allows easy handling of both XLS and XLSX files. Alternative libraries seems to be faster but limited in its functions[11]. Additionally, we see as there is only one single module which has access to the database. This reduces repetitive code on one side and improves readability on the other side. Furthermore, if we would like to adapt the data access layer in the future we only need to modify a single module.

[10] `https://www.cs.princeton.edu/courses/archive/spring03/cs217/lectures/Modules.pdf`
[11] A short discussion can be found on stackoverflow as follows:
`http://stackoverflow.com/questions/3930975/alternative-for-php-excel`

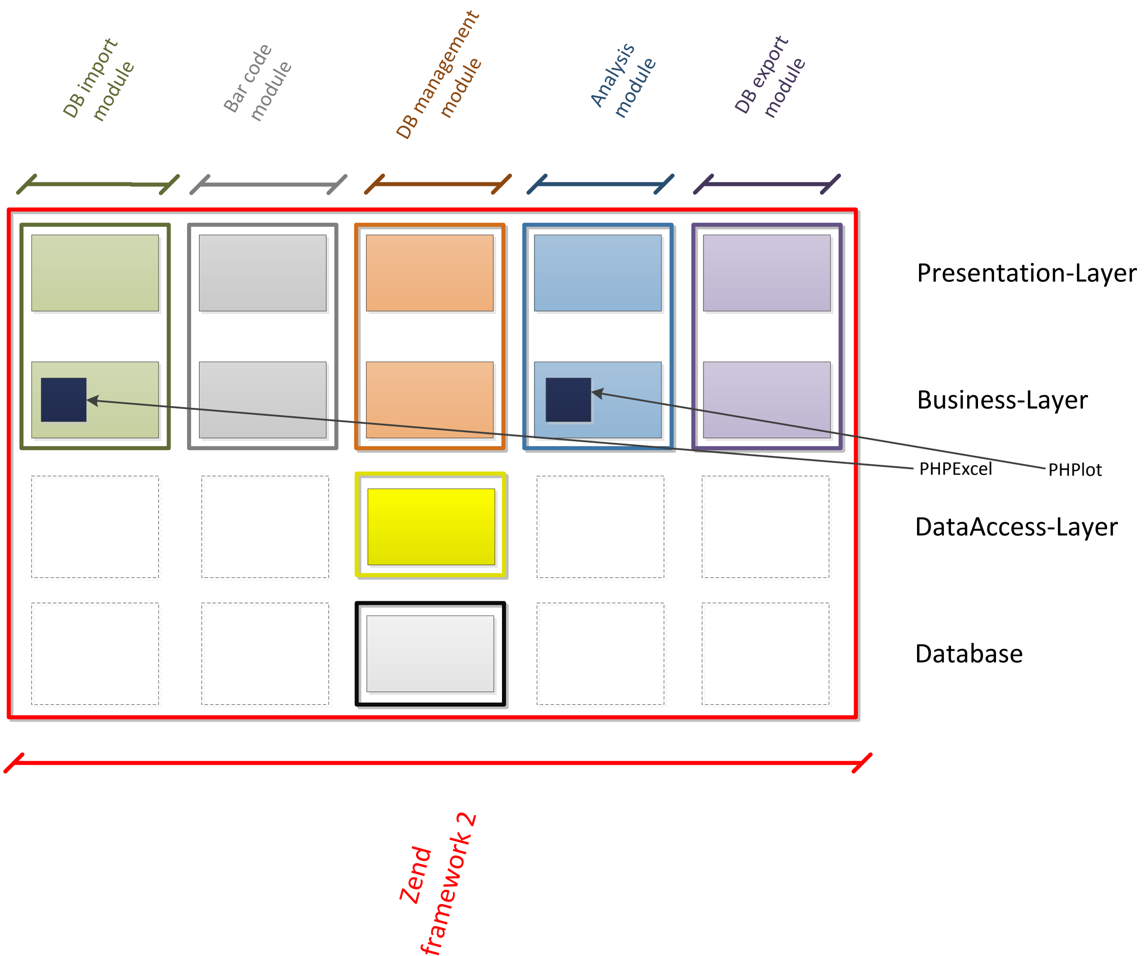


Figure 4.5: Layer architecture of the PSR

The modules of a Zend application can be loaded and managed individually by the Zend framework. First, we would like to give a rough description about the available modules:

Database import	Module to import or update data into the DB. This also includes the parsing of data into an import-friendly representation. For the PSR system we use XML definitions to import.
Database export	Module to export data from the DB. This also includes the parsing into an export- friendly representation such as XML.
Database management	Module to get direct access to DB tables by using a DB driver. This is the only module with a data access layer.
Bar code module	Module for maintaining and printing bar codes.
Analysis module	Module to calculate statistics based on the data stored inside the DB. Currently, it is possible to plot the correlation between attribute data.

To visualize the dependencies between the modules we use the component diagram 4.6. This also shows how the provided functions from the web interface are mapped to the modules. The modules itself are more generic than the process chain provided by the web interface. As an example we have the "DB import module" which do not only import new data but also update existing data into the database - still, importing data. This makes the system flexible for further extensions where we have a few basic modules which we can map to any kind of process visualization by the web interface.

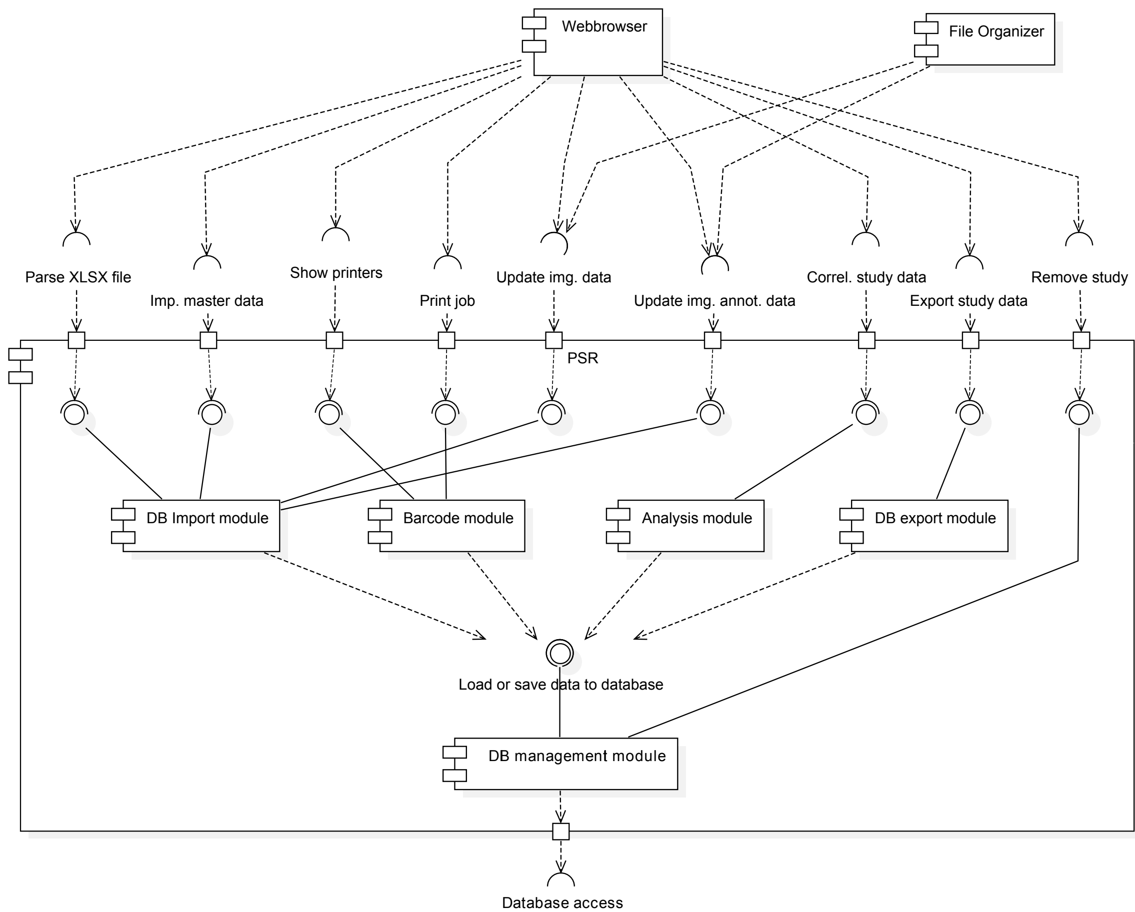


Figure 4.6: Component diagram of the PSR modules

The parsing process of the PSR system As discussed in chapter 3.3 our system do not import a source file containing patient data directly but parse it firstly into a generic XML structure. Currently we only support Excel's XLSX file format which has the advantage as the given data already holds its datatypes comparing to data from a CSV file. This makes it easy to determine if e.g. a date or a string is given. Following we give the XML structure which all parsers have to meet. Also, many structures would be possible we choose one which is self-consistent and adaptable for further patient data. In the following listing the defined XML standard is given where node's attributes are written in brackets.

- study — The new study to import
  - patient_attributes — All attributes of a patient
    - attribute — A specific attribute
      - [id] — ID of the attribute
      - [annotation] — Annotation of the attribute
      - name — Name of the attribute
      - type — Type of the attribute
  - diagnosis_attributes — All attributes of a diagnosis
    - attribute — A specific attribute
      - [id] — ID of the attribute
      - [annotation] — Annotation of the attribute
      - name — Name of the attribute
      - type — Type of the attribute
  - unknown_attributes — All unknown attributes
    - attribute — A specific attribute
      - [id] — ID of the attribute
      - [annotation] — Annotation of the attribute
      - name — Name of the attribute
      - type — Type of the attribute
  - patients — All patient entities
    - patient — A specific patient
      - [id] — ID of the patient
      - attributes — All attributes of the patient
        - attribute — A specific attribute of the patient
          - [ref] — Reference to the attribute
          - value — Value of the attribute
  - diagnoses — All diagnoses entities
    - diagnosis — A specific diagnosis
      - [id] — ID of the diagnosis
      - patient — Related patient to the diagnosis
        - [ref] — Reference to the related patient
      - attributes — All attributes of the diagnosis
        - attribute — A specific attribute of the diagnosis
          - [ref] — Reference to the attribute
          - value — Value of the attribute
      - specimens — All specimens of the diagnosis
        - specimen — A specific specimen of the diagnosis
          - [id] — ID of the specimen
  - unknowns — All unknown entities
    - unknown — A specific unknown entity
      - [id] — ID of the unknown entity
      - attributes — All attributes of the unknown entry
        - attribute — A specific attribute of the unknown entry
          - [ref] — Reference to the attribute
          - value — Value of the attribute

For a better understanding we give an example:

```xml
<?xml version="1.0"?>
<study>
  <patient_attributes>
    <attribute id="1" annotation="ext_pid">
      <id>1</id>
      <name>ID</name>
      <type>int</type>
    </attribute>
    <attribute id="5">
      <id>5</id>
      <name>DoB</name>
      <type>date</type>
    </attribute>
  </patient_attributes>
  <diagnosis_attributes>
    <attribute id="2">
      <id>2</id>
      <name>Round</name>
      <type>int</type>
    </attribute>
    <attribute id="3">
      <id>3</id>
      <name>Methpres</name>
      <type>string</type>
    </attribute>
  </diagnosis_attributes>
  <unknown_attributes/>
  <patients>
    <patient id="2">
      <id>2</id>
      <attributes>
        <attribute ref="1">
          <value>49936</value>
        </attribute>
        <attribute ref="5">
          <value>01-02-1947</value>
        </attribute>
        <attribute ref="6">
          <value>55.515400410678</value>
        </attribute>
      </attributes>
    </patient>
  </patients>
```

```
  <diagnoses>
    <diagnosis id="2">
      <id>2</id>
      <patient ref="2"/>
      <attributes>
        <attribute ref="2">
          <value>3</value>
        </attribute>
        <attribute ref="3">
          <value>1</value>
        </attribute>
      </attributes>
      <specimens>
        <specimen id="17197"/>
        <specimen id="17198"/>
      </specimens>
    </diagnosis>
  </diagnoses>
  <unknowns/>
</study>
```

The given structure is easy to adapt and extensible. Furthermore, the “unknown” nodes allow us to handle entities not related to a patient nor a diagnosis. After parsing an Excel file the resulting XML file could be downloaded and the unknown nodes proceed individually. However, while importing an XML file to the PSR system all unknown nodes are ignored.

The object-relational mapping of the PSR system To get access to the underlying database an object-relational mapping (ORM) layer is preferred. This approach makes it possible to use objects inside the business logic instead of array structures which has an enormous advantage. Using objects makes the code easier to implement, to structure and to maintain later on[12]. The PSR system represents an easy ORM layer which holds its business classes inside the “model” folder of the “Database Management” module. Inside the “dao” sub-folder we finally have the data access objects which maps the database results into business objects. To see the difference of the relational database schema as given in figure 3.2 and the business classes after the mapping we present the class diagram of the business classes in figure 4.7.

[12]More information can be found at:
`http://www.oracle.com/technetwork/java/dataaccessobject-138824.html`

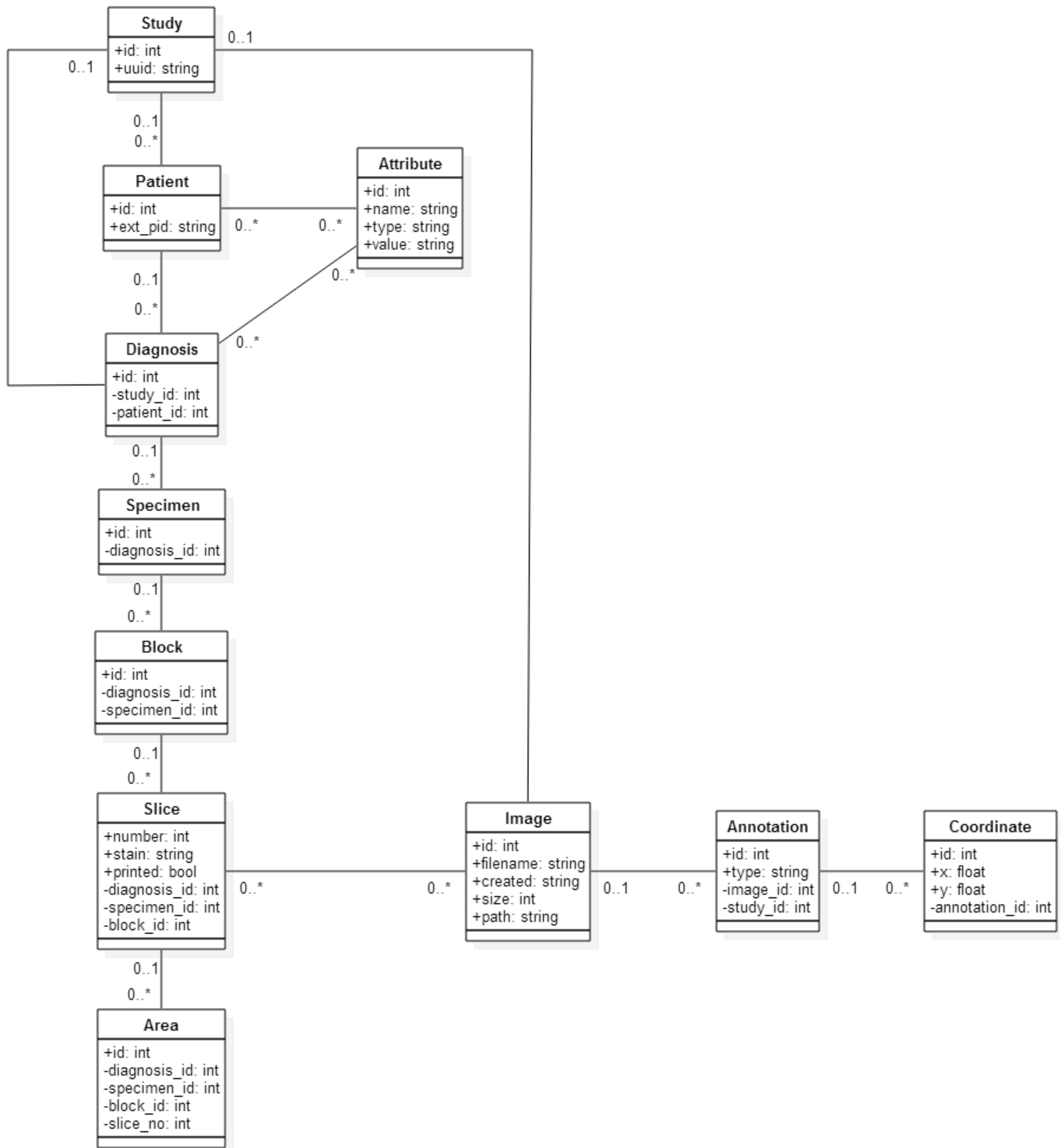


Figure 4.7: The class diagram of the business objects

The data access objects are inspired of modern object databases such as DB4O [7]. To load or save objects you have to give an example of the instance you are searching for. Following an example:

```
$study = new Study();
$study->id = 1;

$study = $studyDAO()->getStudyByExample($study);

if ($study != null)
    $studyDAO()->deleteStudyByExample($study);
```

Firstly, we create a new instance of a study business class with the ID 1. Afterward, we are searching for all studies with the ID equal to 1. Note as all fields of the business class containing there default values will be ignored as filter criteria. In a second step we delete the study found in the previous step. Again, the default values are ignored as filter criteria. This is an example for how easy it is to use objects instead of arrays inside the business layer.

We saw a first search request which gave us a study object with all its attributes, e.g. its ID and its uuid. However, the related patients are not loaded. To do so you have to set the deep load flag as following.

```
$study = new Study();
$study->id = 1;

$deepLoad = true;

$study = $studyDAO()->getStudyByExample($study,
  ↪$deepLoad);

foreach ($study->patients as $patient) {
    echo $patient->id;
}
```

The above code loads all direct related objects of a study. But, the related objects of a related object are ignored. The reason is simple as without a limitation the ORM layer will load all data from the database whenever we search for a single study. Even more worst, if we have a cycle relation the search request will never ends. This problem is addressed by nowadays object databases with the so called activation depth [7, p.142-148; 179-187]. This depth defines how many related objects are loaded from the database - in our case the depth is equal to 1. This allow us to load exactly those data we are really interested in. Furthermore, we can keep the code simple and well-understandable. If you would like to search for objects deeper in the relational chain the code could looks

similar to the following example:

```
$study = new Study();
$study->id = 1;

$deepLoad = true;

$study = $studyDAO()->getStudyByExample($study,
  ↪$deepLoad);

foreach ($study->patients as $patient) {
    $patient =
      ↪$patientDAO()->getPatientByExample($patient,
      ↪$deepLoad);

    foreach ($patient->attributes as $attribute) {
        echo $attribute->name;
    }
}
```

In the previous examples we used the ominous object $studyDAO and $patientDAO. But how are they initialized and how is the database connection established? Let us start with the class "/application/modules/db_management/Module.php". Here, having a look into the method "getServiceConfig()", we see the initialization of all data access objects which get the database connection information from the configuration file "/config/autoload/db.odbc.local.php".

```
$dbSettings = include
  ↪'../config/autoload/db.odbc.local.php';
$table = new StudyDAO_ODBC($dbSettings);
return $table;
```

Alternatively it is also possible to connect a database by Zend's database adapter as following.

```
$dbAdapter = $sm->get('Zend\Db\Adapter\Adapter');
$table = new StudyDAO($dbAdapter);
return $table;
```

The Zend's database adapter gets the database configuration previously from the file "/config/autoload/db.local.php". Hereby, you have a variety of drivers supported such as SQLSRV[13] which is already set up.

[13] http://php.net/manual/de/book.sqlsrv.php

Inside the corresponding data access class the constructor initialize a new database connection using the configuration handed over. If we use the database adapter of Zend we call the constructor of the super class for initialization whereas the ODBC classes directly open a connection using the standard PHP method "odbc_connect"[14]. All further calls are now using the database connection to communicate with the underlaying database. The solution with the Zend adapter is therefore more flexible as no concrete SQL syntax is needed whereas ODBC requires the query in a concrete string representation. However, all the ODBC queries are written in standard SQL-92 which is widely supported from relational database management systems.

But why do we write a new ORM layer anyway if we could use an already existing one? There are two reasons for this design decision. Firstly, the PSR system is neither a system we update every day with transactions nor a system we only use for analysis purposes. One one hand, we extend our master data with new image and annotation information which represents a so called on-line transaction process (OLTP) whereas on the other hand we correlate data which is a so called on-line analytical process (OLAP). The current ORM frameworks all cover OLTP requirements such as SELECT, UPDATE, INSERT statements. However, analytical functions are very rarely or not supported. Secondly, one requirement for the PSR system was to access the database only using an ODBC interface. While having a look on well known PHP frameworks pure ODBC is rarely supported. Zend or the maybe best known PHP ORM framework Doctrine2[15] as examples only support PHP data objects (PDO)[16]. With implementing an own ORM layer it was easy to solve both problems at once.

The architecture of the File Organizer As the File Organizer in the perspective of the development view is less complex than the PSR system we present the whole class diagram in figure 4.8. Each Windows service needs a concrete implementation of the "ServiceBase" class containing the main application and its service installer based on the class "Installer". Next to this two mandatory classes we subdivide the main application into logical entities to structure the code. Therefore we have two classes which are responsible for accessing and handling the application's log and audit files as well as one class to communicate with the PSR system, or any other HTTP service, for register new image files or annotations. We could think about to break the code into another helper class containing methods like "GenerateXMLDocument" or "WriteEventLog". However, we decided to keep the application as simple as possible to reduce its complexity. In the future a redesign might be reasonable containing the additional helper class and bringing the log and audit file handler together by using a common super class. This super class is especially recommended if further file handlers e.g. for configuration files in XML representation would be implemented.

[14] http://php.net/manual/de/function.odbc-connect.php
[15] http://www.doctrine-project.org/projects/orm.html
[16] http://php.net/manual/de/book.pdo.php

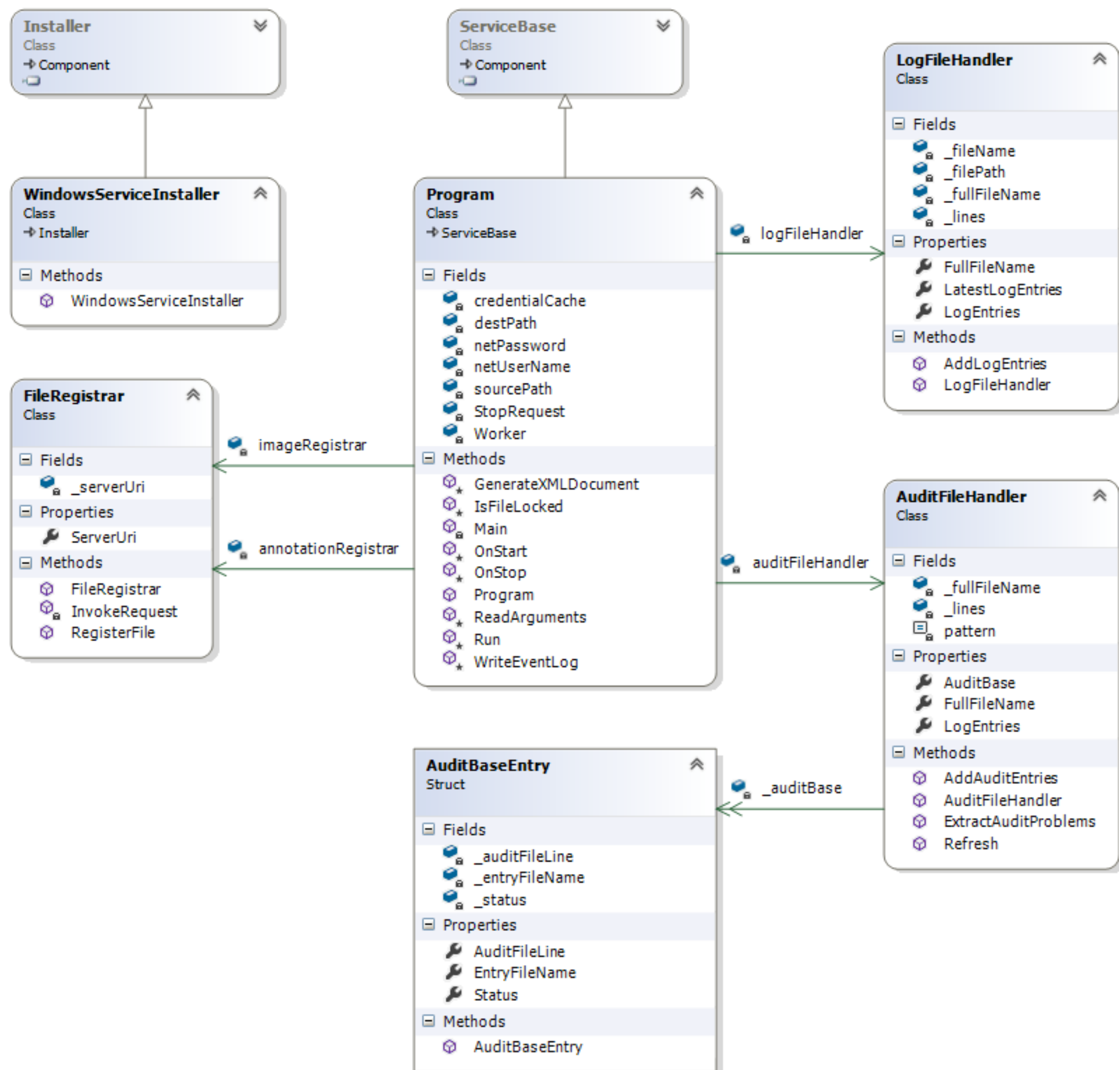


Figure 4.8: Class diagram of the file organizer

The class diagram is easy to understand and complement the sequence diagrams in figure 4.4. The "Program" class holds the never ending main loop which checks for new image data and its annotations. Whenever an event occurs it uses the log and audit file handler to write its actions. While the log file contains information about all actions the File Organizer did the audit file contains only the most recent actions made for a specific image file, e.g. gather information about it. If the File Organizer unexpectedly terminates it reads the audit file the next time it starts. Each audit entry will be analyzed for an inconsistent state and if so, tried to be fixed. The approach was adapted from the database domain where the audit file is also known as transaction log[17].

[17] http://msdn.microsoft.com/de-ch/library/ms190925.aspx

The error handling of the File Organizer The File Organizer analysis and handles a file within six steps which are based on the overall process described in chapter 3.8.

1. Find a new scanned image file
    2. Gather its information
    3. Register file to the PSR system
    4. Copy image file from the source folder into its destination folder
5. Find new annotations for an already copied image file
    6. Register new or edited annotations to the PSR system

If we analyze the upper actions it appears as there is only one single action which could cause inconsistency in our sub-process. This is, if we register a new scanned image to the PSR system (3) but do not copy the image file into its destination folder (4). In all other cases we may lose the information as a new image file or annotation is available. However, as soon as we restart the File Organizer we will find the same files and its information again and process them as expected. To make the system more flexible also for future changes a whole class was implemented which handles actions made to files. Each time an action appears the status of a specific file is updated inside the audit file. Later on, whenever the File Organizer is starting it checks the audit file for the status of each proceeded file. If we have the situation as an image file is registered to the PSR system (3) but not yet copied (4) we can react and copy it belated to gain a consistent status again. This behavior can be extended so other inconsistency can be located and fixed. Further more, the audit file handler may be used in any other C# application as desired.

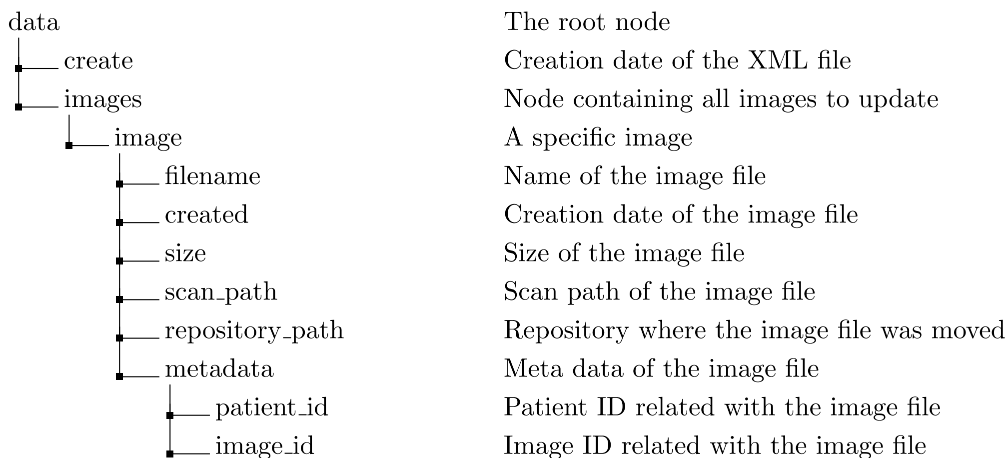
4.4.3 Physical view

We discussed a lot about the PSR system which manages and holds the inventory data. Next to the PSR system we have two more processes working hand in hand together to accomplish the overall process: the scanning process which creates new image data and the File Organizer which moves the new scanned image files into a logical structure and register them as well as new annotation files to the PSR system. With respect to all these processes we have an overview of all connected units and there communication. See also figure 3.4 for the process overview. While the units as such are already covered previously the communication in-between is now part of the next section.

4.4.4 Process view

As described we have three sub-processes working hand in hand with each other. Firstly, the manual process for scanning new image files. This process generates new images on the network file directory, in our case `\\fs-group\ds_00524_daten\PathoStudyResearch\`. The next sub-process, the File Organizer, takes these files and moves them to its destination network file directory, in our case `\\fs-group\ds_00524_daten\PathoStudyResearch\repository`, and organize the files there. If we challenge the current approach we may think about an external FTP server or even a cloud solution. Both are valid solutions but are much more complex to implement. Also, we do not have any advantage comparing to the first solution in respect to the given requirements.

Next, we would like to discuss about the registration operation of the File Organizer which sends a defined XML structure to the PSR system, in our case to `histodb2.usz.ch/patho_study_research/dbimport/import/scandata`.
Alternatively, the registration can also be performed manually by uploading an XML file using the web form at the URL given above.
The data represented in the XML file are based on the given requirements and needs from the overall process as discusses in chapter 2.2. Many other structures are possible whereas all of them needs to hold the same information. In the following definition each sub-node is intended to its parent.

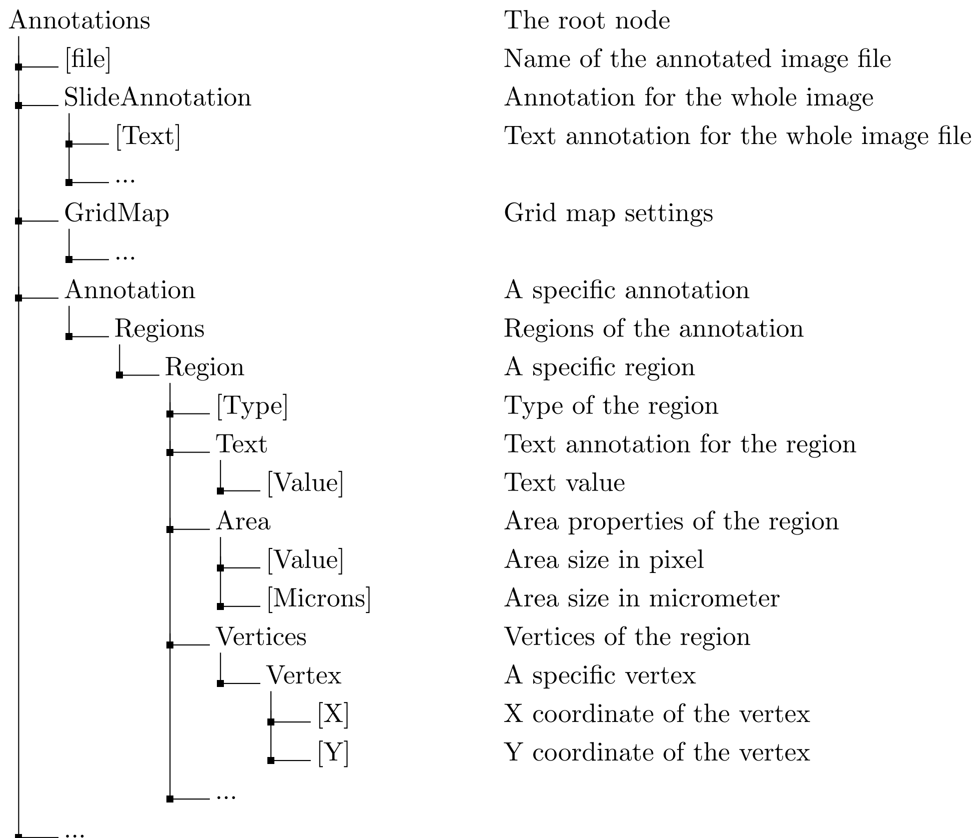


For a better understanding we give an example:

```xml
<data>
  <created>2014-10-03T17:37:22.9825922+02:00</created>
  <images>
    <image>
      <filename>169S001.tif</filename>
      <created>2014-10-03T17:37:07.5371309+02:00</created>
      <size>51</size>
      <scan_path>C:\tmp\in\169S001.tif</scan_path>
      <repository_path>C:\tmp\out\169\001.tif</repository_path>
```

```
      <metadata>
        <patient_id>169</patient_id>
        <image_id>001</image_id>
      </metadata>
      </image>
      <image>
      ...
    </image>
  </images>
</data>
```

Equally to the file registration on the PSR system new image annotations are handled. Therefore, the File Organizer scans the destination directory continuously for new annotations. The annotations have to be made in the Ventana image viewer to be recognized correctly. This decision is based on the fact as it is the only image viewer currently available at the university hospital which can read the Ventana image files and make annotations the same time. The implementation of an own image viewer would need a lot of effort and therefore is not practicable. Furthermore, the image viewer already creates a desired XML file for marked annotations. Its structure is given as follows where sub-nodes are intended and attributes are written in brackets.



For a better understanding we give another example:

```
<?xml version="1.0"?>
<Annotations file="C:\tmp\169\001.tif">
  <SlideAnnotation Text="Test" Voice=""/>
  <GridMap>
    ...
  </GridMap>
  <Annotation LineColor="65280">
    <Regions>
      <Region Type="rtRectangle" regSelected="FALSE">
        <Text Value="test"/>
        <Voice Path=""/>
        <Area Value="42920"/>
        <Area Microns="9280.38"/>
        <Vertices>
          <Vertex X="15447" Y="9752"/>
          <Vertex X="15679" Y="9937"/>
        </Vertices>
      </Region>
    </Regions>
  </Annotation>
   ...
</Annotations>
```

To distinguish if an annotation is new or modified we adapt the approach of modern backup tools. Each file on a modern Windows file system has a field called "attribute". Whenever the file is modified this attribute is set to "archive" which is an indicator for backup softwares to handle the file during the next backup process. We decided to use the same approach and scan for all annotation files containing the "archive" attribute. If given, we register the annotation to the PSR system, in our case to `histodb2.usz.ch/patho_study_research/dbimport/import/annotationdata`, and remove the attribute again. As an alternative solution we could use fields others than the "attribute" or save the registered files and there modification date in an external and maybe hidden file or database. As the given solution is robust and much more easy to implement we did not concern the other solutions. However, if once we would face a file system not providing the "attribute" field we could use one of the alternatives given.

4.4.5 Use case view

The following use cases cover all the functional requirements of the image acquirement process as described in chapter 2.2. The use cases related to the PSR system are also implemented as Selenium test cases which are described in chapter 5. This use cases are later on used in the acceptance test which is also listed in the appendix 10.3.

Use Case	**[UC001] Import an excel file with study data**
Description	The user uploads an Excel file to the PSR system, sets the attributes to import, the attribute containing the external PID and the specimen numbers. Afterward, he starts importing the file.
Precondition	- The user already face the start page of the PSR system - The user has an Excel file containing all study data to import
Postcondition	A new study is created inside the database and all data are imported.
Process	- Go to: Import/Translate XLSX to XML file - Choose the Excel file to upload and process - Split attributes which includes more than one value - Choose the attribute representing the external patient ID - Choose the attribute representing the specimen IDs - Translate the XLSX file into an XML file - Verify the translation report - Start importing the translated data - Verify the import report
Pass	- The translated XML file contains all data chosen - Proceeded attributes, rows and cells are reported - Empty cells are reported - All data are imported correctly into the database
Fail	- The translated XML file do not contains all data chosen - Proceeded attributes, rows or cells are not reported correctly - Empty cells are not reported correctly - Not all data are imported correctly into the database

Use Case	**[UC002] Define a new bar code printer**
Description	The user define a new bar code printer inside the PSR system.
Precondition	- The user already faces the start page of the PSR system
Postcondition	A new bar code printer is created inside the database.
Process	- Go to: Print/Show bar code printers - Verify as the new bar code printer is not already present - Add a new bar code printer - Specify the type and IP address of the printer - Save the new printer - Verify as the new bar code printer is now present
Pass	- The bar code printer is stored correctly to the database
Fail	- The bar code printer is not stored correctly to the database

Use Case	**[UC003] Print bar codes for slides to scan**
Description	The user prints new bar codes for physical slides which he would like to scan.
Precondition	- The user already faces the start page of the PSR system - Study data are already imported [UC001] - The bar code printer to use is already defined [UC002]
Postcondition	New blocks, slices and images are stored inside the database and the corresponding bar codes are printed.
Process	- Go to: Print/Create new print job - Choose a bar code printer - Search for slide values or specify them manually - Choose a block or create it if not available - Choose slices or create them if not available - Print the bar code - Verify the printed bar code - Print additionally bar codes if slides are sill available
Pass	- The new blocks are stored correctly to the database - The new slices are stored correctly to the database - The new images are stored correctly to the database - The bar codes are printed as expected (see also p. 13)
Fail	- The new blocks are stored incorrectly to the database - The new slices are stored incorrectly to the database - The new images are stored incorrectly to the database - The bar codes are not printed as expected

Use Case	**[UC004] Scan slides with the Ventana iScan HT**
Description	The user scans slides prepared with a bar code.
Precondition	- The user already prepared the slides with bar codes [UC003] - The slides are put inside the Ventana iScan HT - The Ventana iScan HT and the corresponding PC is started
Postcondition	Image data of the scanned slides are created and stored by the Ventan iScan HT.
Process	- Start the Ventana iScan HT software - Select a slide holder containing the slides to scan - Specify the slides to scan, its destination and quality - Add the slide holder for scanning thumbnails - Do the same procedure for all slide holders to scan - Start scan process for thumbnails - Select a slide holder containing the slides to scan - Verify each scanned slide and correct the ROI and focus points - Add the slide holder for scanning its slides in high quality - Do the same procedure for all slide holders to scan - Start scan process - Verify all stored image data by the Ventana iScan HT
Pass	- All images are scanned correctly and without errors
Fail	- Not all images are proceeded - Proceeded images thrown errors

Use Case	**[UC005] Update new scanned images to the PSR system**
Description	The file organizer moves new scanned images into a logical file structure and register them to the PSR system.
Precondition	- The user already scanned slides with the Ventana iScan HT [UC004] - The file organizer is installed and works correctly
Postcondition	The image data are moved in a logical file structure and registered to the PSR system.
Process	- Start the procedure if not done already
Pass	- The image files are moved into the correct destination folder - The file organizer do not thrown any error - The image data from the database are updated correctly
Fail	- An image file is moved into the error folder - The file organizer stopped unexpectedly with errors - The image data from the database are not or incorrectly updated

Use Case	**[UC006] Update new image annotations to the PSR system**
Description	The file organizer find new image annotations and register them to the PSR system.
Precondition	- The file organizer is installed and works correctly - The image data are already moved correctly to its destination folder and registered to the PSR system [UC005]
Postcondition	The image annotations are registered to the PSR system.
Process	- Start the procedure if not done already - Create a new image annotation with Ventana's Image Viewer
Pass	- The image annotations from the database are updated correctly - The file organizer do not thrown any error
Fail	- The image annotations from the database are updated incorrectly - The file organizer stopped unexpectedly with errors

Use Case	**[UC007] Analyse imported data of a study**
Description	The user analysis the imported study data by correlating its attributes.
Precondition	- The user already faces the start page of the PSR system - Study data are already imported [UC001]
Postcondition	A correlation diagram between two attributes is drawn.
Process	- Go to: Analysis/Correlate data from study - Choose a study to analyse - Choose an attribute for the x axis - Choose an attribute for the y axis - Correlate data and calculate the corresponding diagram - Verify the diagram
Pass	- The diagram is plotted correctly
Fail	- The diagram is plotted incorrectly or not at all

Use Case	**[UC008] Remove imported study**
Description	The user realize as the imported study contains incorrect data and delete the study as well as all corresponding data from the database.
Precondition	- The user already faces the start page of the PSR system - Study data are already imported [UC001]
Postcondition	The deleted study and all its related data are removed from the database.
Process	- Go to: Management/Remove study from database - Choose a study to remove - Confirm to remove the study - Verify as the study was removed
Pass	- The study and all related data are removed from the database
Fail	- The study or any of its related data are not removed correctly from the database

Use Case	**[UC009] Export a study and all its data**
Description	The user export all data of a specific study for further analysis from the database.
Precondition	- The user already faces the start page of the PSR system - Study data are already imported [UC001]
Postcondition	The study and all its related data are exported into an XML representation.
Process	- Go to: Export/Export study from database - Choose a study to export - Choose attributes to export - Specify if the export also should contains all related specimens - Specify if the export also should contains all related blocks - Specify if the export also should contains all related slices - Specify if the export also should contains all related images - Start the export procedure - Verify the export report and the created XML file
Pass	- The exported XML file contains all data chosen
Fail	- The exported XML file do not contains all data chosen

5 Software Testing

To ensure as the PSR system works correctly the software was tested continuously by function tests, system tests and security tests. During the function tests the individual workflows of the PSR system as well as the one of the File Organizer were evaluated while the system test combines all workflows together. Finally, the security was tested with the community edition of Netsparker[1].

5.1 Function tests

The function tests were made manually. Each new function of the PSR system or the File Organizer was tested individually to ensure as the standard workflow works correctly and as expected. This tests did not involve the system as such but only enclosed functions. We do not provide any results here as the system test covers all function tests.

5.2 System test

The system test is a combination of all function tests which involves the interaction between the subsystems. The system test for the PSR system is made with the Selenium IDE and covers the following use cases:

- Import Test
- Print Test
- Update Test
- Analysis Test
- Remove Test
- Export Test

The test descriptions can be found in the appendix 10.2. All tests were successfully and ensured the correct functionality during the development process while changing depended resources.

[1] https://www.netsparker.com/

5.3 Security test

With Netsparker the web-application was analyzed for security issues, namely SQL-Injections and Cross-Site-Scripting. For testing SQL-Injections we added a new printer of the following type:

```
DELETE FROM [PathoStudyResearch].[dbo].[Attribute]
```

Additionally, we created a new Excel file with the same content and uploaded it to the PSR system as well as exported the same data again. In all cases we performed database queries with the sensitive string given above but never deleted any record of the "Attribute" table due secure string escaping. Another Excel file was uploaded with a java script command. While previewing the file in the PSR system it appears as the script was executed - a typical Cross-Site-Script. The application was investigated and the described page fixed - no other pages with a potential risk were found.

As the PRS system is used internally it is unlikely as any of those attacks happen in the near future. Anyway, we never know if the application or any part may be used outside the secure intranet as for example in a demilitarized zone. In this case a secure and robust software fragment is very valuable to omit future security problems.

6 Results

After the implementation of the PSR system and the File Organizer we would like to summarize and reflect about its most important results.

Expandability A main aspect behind the design of the PSR system was the expandability for further requirements. We tried to meet this aspect as much as possible by several implementation decisions. With the URL mapping we mostly use parameters instead of submitting POST requests. For example plotting a correlation diagram from the study with the ID 1 where the 3rd attribute is used as x-axis and the 15th attribute as y-axis can be performed calling the following URL:

```
histodb2.usz.ch/patho_study_research/analysis/diagram/
  ↪plot/study=1&xaxis=3&yaxis=15
```

This approach additionally enables external software, e.g. shell scripts, to interact with the PSR system. Most responses from the PSR system contain data in the common "JSON" format which is easy to parse in most environments.
The only exception represents the URL to remove a whole study from the database. In this case a form request is expected containing the ID of the study.
This does not provide more security but prevent scripts from accidentally removing studies, e.g. by calling the URL with an incorrect ID.

Another important part, as previously mentioned, is the modular structure of the PSR system which easily allows us to add new modules and functions by only registering them in the application's configuration file. Also the already given modules are programmed to be reusable. An example is the XML importer or serializer which represents super classes for importing or exporting data from the database. If we would like to handle not only XML files but also text files we can simply implement a corresponding importer and serializer using the corresponding super classes and integrate them into the web-frontend of PSR.

Another best-practice from the Zend framework is to provide fragments used across the whole application by the service manager. This approach can be found for the data access objects which comes along with a meaningfully example. Originally, the database access was implemented using a PDO_ODBC driver[1] as the requirement was to use ODBC. At the end the requirement became more specific and the installation of

[1] http://php.net/manual/de/ref.pdo-odbc.php

a PDO driver was undesired. Without using the service manager it would be necessary not only to change the data access objects but also each code fragment using such an object. While using the service manager it was easy to adapt the requirement by only providing new ODBC data access objects. The other PSR modules wont realize any change as they only require any kind of data access object with some defined interface methods. So to say, the PSR system can also easily be adapted if a new DBMS is used.

The approach above even brings us to the next keyword: coupling. By using modules which interact with each other by calling URLs, instead of sending data such as HTML forms, we can individually change the process and use given functions by new modules. This flexibility does not only hold for the PSR system but also for the File Organizer which is only coupled to the PSR system by URL calls. Therefore, other scanners than the Ventana iScan HT may be supported in the future.

The XLSX-parsing-process currently only expects a table with titles in its first row and data in all rows below. With this data we can already fill our database. To improve the flexibility even more we use regular expressions to split attribute values whenever needed. This already covers a lot of future possibilities for given data, e.g. splitting a value after the first occurrence of a big letter. However, we can never cover all possibilities and future pre-processing adaptions that may be needed.
In this case an extension of the current XLSX-parser is required whereas the rest of the process remains.

Robustness It is always possible that a user sends invalid data to the PSR system. In all known cases the system reacts and gives a user friendly error message. The system therefore has four tiers to handle exceptions:

1. Using Javascript on a view to immediately react on wrong input data or to prohibit them. E.g. the bar code print button is only available after all necessary information are given. Please keep in mind as those checks can easily be avoided by manipulating the Javascript code and do not provide any security!

2. After data are sent to the PSR system the controller involved can be used to verify the given information. E.g. the XML file is verified for valid data before being imported into the database.

3. Instead of validating all forms or data access objects individually we can also use an input filter which can easily be configured inside the corresponding form or DAO classes. The validation can also be performed inside the controller class or any other business class by calling the $\$form- > isValid()$ method. Examples can be found all over PSR, e.g. while importing data into the database.

4. While interacting with the database the last validation is performed by the constrains from the database schema. An exception this late will throw a technical exception most likely not understandable by an end-user and should always be avoided.

Because the PSR system does not hold hard restrictions but tries to be flexible incorrect inputs are still possible. Another problem are the redundant checks on the client side and server side. While Javascript can immediately react on wrong user input it supports the usability aspect so the users do not need to send a web-form each time before they get informed about faulty inputs. However, Javascript is easy to omit and therefore does not provide a secure error handling. This means, we need to check for faulty inputs again on the server side. It would be nice to bring the error handling together so we need to implement it only once.

Miscellaneous A nice feature would be to manage the File Organizer and its configuration with a graphical user interface. With the new front-end one could start, stop or restart the File Organizer, change its configurations defined inside the Windows registry and visualize its status. The last aspect is rather easy to realize as we only need to present the log and audit file. Furthermore, the log file handler of the File Organizer holds the property "LatestLogEntries()" which allows to read the latest 50 log entries without reading the whole file itself. Because of the advanced time and to ensure a highest possible quality for the PSR system this feature was not implemented yet.

The scanning process After implementing the previously described process the anonymized data of the active surveillance cohort from Aarau[1] could be imported without any errors into the PSR system. In a next step we scanned all available slides with a HE stain and related them to the imported cohort data. The table 6.1 summarize the results. However, not all slides are available at this moment expecting around 350 more slides in the future.

As shown in table 6.1 the image acquisition needs a lot of time. Although the scanning itself could be established during the night handling the preview image, setting up the focus points and handling errors still has to be made manually. Unfortunately the focus points automatically set by the Ventana iScan HT are most of the time not suitable. If we could skip this part we may also skip the preview process which would improve the process enormous.

One last thing to mention: The Ventan iScan HT has a capacity of 360 slides which could be scanned over night. Therefore the scanner provides a good performance if we can use it for batch jobs. However, if we use the scanner during daily business to scan single slides we have to expect an average working time per slide of about 9 minutes and 50 seconds.

This does not affect the process itself which could also be performed by any other scanner but may impact the scanning strategy in the future.

Slides scanned	261	
Incomplete slides[1]	44	
Slides with technical problems[2]	4	
Slides with unrecognized bar codes	4	
Time elapsed for scanning slides w/o errors		
Scanning previews	178 min.	(6.95 %)
Setting focus points	88 min.	(3.44 %)
Scanning	1917.03 min.	(74.86 %)
Total time elapsed	2183.03 min.	(85.25 %)
Time elapsed for re-scanning slides w/ errors		
Scanning previews	26 min.	(1.02 %)
Setting focus points	18 min.	(0.70 %)
Scanning	310.68 min.	(12.13 %)
Handling fallen slides	23 min.	(0.90 %)
Handling unrecognized bar codes	4 min.	(0.16 %)
Total time elapsed	377.68 min.	(14.75 %)
Total time of scanning slides (w/ and w/o errors)	2560.72 min.	(100.00 %)
∅ time per slide without error handling	8.3641 min.	
∅ time per slide including error handling	9.8112 min.	

Table 6.1: Results of the scanning process

[1]Slides which were not accepted by the Ventana iScan HT and needed a re-scan.
[2]Slides which felt off the robotic arm and needed to be fixed manually.

7 Reflection

After talking about the process implementation in detail we would like to take a step back and reflect the overall process again. What could we improve the next time? Firstly, the productive environment and the development environment were not running under the exact same Apache web server but had one minor version difference. Because the Apache website did not offer the same version as used in the productive environment anymore we took the most equal version with the same major release. We did not expect big troubles in migrating the development environment into the productive environment which was not the exact case. Especially the routing and file access restrictions needed some adaptations. At the end, the solutions found were quite simple and only affected one or two lines of code. However, this scenario shows as a test environment as equal as possible to the productive environment is preferable to save time in the migration phase at the end of a project.

Another issue was the unfamiliar environment. On one hand PHP itself and on the other hand the IT infrastructure were not well known. If time would be available it could be very interesting to implement the PSR system with different frameworks than Zend to give a detailed statement about which framework would be best. Currently, we could only do so for small demo applications and by reading articles. Also, PHP was not familiar at the beginning the language and coding style was easy to learn and a large refactoring of the PSR system is not expected. Next to PHP also the infrastructure was unknown where Norbert Wey gave a very good insight. This was important to minimize the risk as a sub-process or even the whole process itself do not run as expected. As example we take the File Organizer which needs access to the network path containing the new scanned images and the network path containing the organized file structure of the reposited images. Depending on the infrastructure we may not have enough rights to read or copy the data.

Related to this issues it is also important to have a clear understanding of the overall process and keep in regular touch with the stakeholders. To start with the implementation of an unclear process most probably lead to bad results. A well known example is the multilingualism which has to be defined at the beginning of a project. The implementation at the end of a project is most probably not possible due the enormous time effort needed for restructuring the application. Another issue is the changing interest of the stakeholders during a project. If we stay in regular contact we can react on such situations. One example was the ODBC interface of the Microsoft database. Also this requirement was defined the connection did not work with the chosen solution in the beginning.

The problem occurred as the needed PDO_ODBC[1] driver was missing and for security reason its installation was not desired. As the problem was realized and discussed early the ORM layer could be adapted to a pure ODBC connection.

7.1 General recommendations

During the modeling of the process several best-practices have been emerged which can be adapted not only for the current project. Based on the reflection from above we would like to give a summary about this practices:

- A pivotal question in the beginning of a software project is the programming language and frameworks to use. It is wisely to use languages or frameworks which are already known by the programming team and the stakeholder. This is not only important for the understanding but also for the future maintenance which is another important point to think about. As in our case the stakeholder will maintain, use and maybe extend the software by himself the decision was clear to use a programming language he is familiar with.
- Before we start programming we should understand and modeling the overall process as detailed as possible so unexpected problems can be reduced to a minimum. During the modeling phase the stakeholder have to be involved so a useful solution can be realized. Also, not only the easiest case should be mentioned but also future extensions and how to handle errors.
- The stakeholders should be involved as much as possible. This makes it possible to react fast on misunderstandings and changing requirements.
- Trade-offs should be clear defined and discussed with the stakeholder. An example was the implementation of the ORM layer. The layer implementation on one hand needs effort but provides on the other hand not only programmatically advantages but is also very flexible for further changes of the application.
- Widely used functions have to be implemented as early as possible and whenever needed. Implementing e.g. the multilingualism or the ORM layer in a later stage of the project needs an enormous amount of time and might not be possible anymore due the complexity.
- The productive and development environment should be as close as even possible to avoid unexpected problems during the migration phase.

[1] A driver implementation of the PHP Data Objects (PDO) interface which could access databases through the ODBC interface. See also `http://php.net/manual/en/ref.pdo-odbc.php`

8 Bibliography

[1] F. H. Schröder et al., "Prostate-cancer mortality at 11 years of follow-up," The New England Journal of Medicine, vol. 366, no. 11, pp. 981–990, 2012. [Online]. Available: http://www.nejm.org/doi/full/10.1056/NEJMoa1113135

[2] E. Ebnöther and J. Hablützel, "Früherkennung von prostatakrebs," http://www.krebsliga.ch/de/shop_/prostatakrebs_d2.cfm, Krebsliga Schweiz, Effingerstrasse 40, Postfach 8219, 3001 Bern, 2010, p. 7.

[3] D. Ilic, M. M. Neuberger, M. Djulbegovic, and P. Dahm, "Screening for prostata cancer," The Cochrane Collaboration, Review CD004720, 2013, cochrane Database of Systematic Reviews 2013, Issue 1, p. 2.

[4] P. Kruchten, "The 4+1 view model of architecture," in Software, IEEE, vol. 12. IEEE, 1995, pp. 42–50, issue 6. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=469759

[5] J. Goyvaerts and S. Levithan, Regular Expressions Cookbook. O'Reilly Media, 2012, p. 15-23, ISBN: 9781449327484. [Online]. Available: https://books.google.ch/books?id=0Msuh5Vq-uYC

[6] M. Masse, REST API Design Rulebook, ser. Oreilly and Associate Series. O'Reilly Media, 2011, p. 5-6, ISBN: 9781449310509. [Online]. Available: https://books.google.ch/books?id=4lZcsRwXo6MC

[7] J. Paterson and S. Edlich, The Definitive Guide to db4o, ser. Books for professionals by professionals. Apress, 2006, p. 142-148; 179-187, ISBN: 9781430201762.

9 Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Date Signature

10 Appendix

10.1 Software construction tools

In a nowadays programming environment we expect to create a source code documentation or the generation of UML diagrams within a few clicks. PHP does support such software construction tools which are but often only available for a fee. For an open source solution we recommend the following tools.

10.1.1 PHP_UML

PHP_UML is a tool for creating UML diagrams. However, the flexibility of the diagram visualization is quite poor. To solve this circumstance we can also create an XMI[1] file which contains the available classes, its attributes, etc. This approach is similar to the source code documentation under Microsoft Visual Studio which generates an XML file with similar information. The XMI standard is defined by the Object Management Group and is supported by many UML modeling tools, e.g. ArgoUML[2].

To use PHP_UML a PHP interpreter is required as well as the following packages:

- PHP_UML 1.6.1
  (`http://pear.php.net/package/PHP_UML`)
- Console_CommandLine 1.2.0
  (`http://pear.php.net/package/Console_CommandLine`)
- PEAR_Exception 1.0.0beta1
  (`http://pear.php.net/package/PEAR_Exception`)

The packages can be either installed by Pear itself or manually downloaded from the website. If you download the content manually you can extract all the compressed archives into a folder of your choice. Please verify as PHP_UML can find the files of the depended packages.

One thing to mention: In PHP_UML 1.6.1 there is a bug which generates XMI files not readable by most UML modeling tools. This is, because the elements inside the XMI file hold an unique identifier which is calculated wrongly so the same identifier may occur more than once. The bug is known under the ID #20153 (Increase ID entropy) which also holds a solution for fixing the "SimpleUID.php" file[3].

[1] `http://www.omg.org/spec/XMI/`
[2] `http://argouml.tigris.org/`
[3] `http://grokbase.com/t/php/pear-bugs/13cmpnprnf/`

10.1.2 PHPDoc

A well known and good source code documentation tool for PHP is the so called phpDocumentor which is now available in the major version 2[4]. As PHP_UML you can install the tool manually or by Pear. The tool is easy to use and you can directly start with generating your documentation without the need of dependent packages. If you would like to generate diagrams you have to install Graphviz which is available as open source project under `http://www.graphviz.org` and register the folder "<graphviz dir>/bin" to your "Path" environment variable. While using phpDocumentor 2.8.0 you may face the following problems:

- The source code visualization of the generated documentation may not be loaded.
  1. problem:
  The phpDocumentor uses XMLHttpRequests for loading the source code dynamically. To load local resources by Javascript however is not safe and may be blocked by your browser. E.g. Firefox v.33 can open the source code directly while Chrome v.38 has to be started with the option "–allow-file-access-from-files "
  2. problem:
  All source code files are saved in a single folder. To simulate the path structure the file names containing backslashes (using the ASCII code "%5C") which may not be resolved correctly by the browser. If so, the files have to be moved to the corresponding folder structure manually or, preferable, by script.

- Parameters may be described correctly in the API documentation but generate an error inside the API report anyway. This issue depends on the PHP version you are using as uninitialized variables are interpreted in different ways. If you face the problem described you have to change the file
  "<phpDocumentor dir>/Plugin/Core/Descriptor/Validator/Constraints/Functions /IsArgumentInDocBlockValidator.php". On line 37 the value "$value→index" is not initialized but used as an array index. If the variable is interpreted as an integer, the index 0 is used and the application works as expected. Otherwise it is interpreted as a string and the index " " is used (which is a valid index under PHP). However, with the second interpretation the validation always throws an error. You can fix this issue easily by casting the value: "(int) $value→index".

- Several features are already documented inside the official manual[5] but not yet implemented. An example is the inline hyperlink which is only displayed as normal text. The documentation says: "The effects of the inline version of this tag are not yet fully implemented in PhpDocumentor2"[6]. Please verify tags first before using them or if needed implement the functionality yourself.

The phpDocumentor is an active open source project so the above issues may (hopefully) be fixed in the near future.

[4] `http://www.phpdoc.org/`
[5] `http://www.phpdoc.org/docs/latest/index.html`
[6] `http://phpdoc.org/docs/latest/references/phpdoc/tags/link.html`

10.2 Selenium test cases

The following tables describe the individual Selenium test cases which were successfully performed. For further information about the syntax please visit `http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#script-syntax`.

Import_Test		
open	/dbimport/main	
click	xpath=(//button[@type='button'])[2]	
clickAndWait	link=Translate XLSX to XML file	
type	name=input_file	C:\tmp\qry_Screening_Uebersicht_alle_3_Runden.xlsx
clickAndWait	id=submitbutton	
click	document.prepare.elements['patient_attribute_fieldsets[0]'][1]	
click	document.prepare.elements['patient_attribute_fieldsets[4]'][1]	
click	document.prepare.elements['patient_attribute_fieldsets[5]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[1]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[2]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[3]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[6]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[7]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[8]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[9]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[10]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[11]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[12]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[13]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[14]'][1]	
click	document.prepare.elements['diagnosis_attribute_fieldsets[15]'][1]	
click	name=add_splitting_criteria	
select	name=splitting_criteria_fieldsets[0][original_title]	label=Bx_Nr_Aarau
type	name=splitting_criteria_fieldsets[0][first_title]	Box No.
type	name=splitting_criteria_fieldsets[0][second_title]	Specimen No.
type	name=splitting_criteria_fieldsets[0][regex]	[0-9]+-[0-9]+
click	name=add_ext_pid	
click	name=add_specimen_no	
select	name=specimen_no_fieldsets[__index__][column]	label=Specimen No.
clickAndWait	id=submitbutton	
assertText	css=div.container > div > ul > li	18 attributes were processed
assertText	css=div.container > div > ul > li:nth-child(2)	637 rows were processed
assertText	css=div.container > div > ul > li:nth-child(3)	11448 cells were processed
clickAndWait	name=submit	
click	id=submitbutton	
waitForText	css=span.progress-value	regexp:importing patients.*1[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing patients.*2[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing patients.*3[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing patients.*4[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing diagnoses.*6[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing diagnoses.*7[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing diagnoses.*8[0-9]%.*$
waitForText	css=span.progress-value	regexp:importing diagnoses.*9[0-9]%.*$
waitForText	css=span.progress-value	Import complete!
assertText	css=div.container > div > ul > li	18 attributes were imported
assertText	css=div.container > div > ul > li:nth-child(2)	636 patients were imported
assertText	css=div.container > div > ul > li:nth-child(3)	636 diagnosis were imported
assertText	css=div.container > div > ul > li:nth-child(4)	2697 specimens were imported
assertText	css=div.container > div:nth-child(8) > ul > li	None
assertText	css=div.container > div:nth-child(11) > ul > li	None

Table 10.1: Import test case

Print_Test		
open	/barcode/main	
click	xpath=(//button[@type='button'])[3]	
clickAndWait	link=Show all barcode printers	
clickAndWait	link=Add new printer	
type	name=type	Zebra - USZ
type	name=ip_address	144.200.147.32
clickAndWait	id=submitbutton	
click	xpath=(//button[@type='button'])[3]	
clickAndWait	link=Create new print job	
select	name=search_study	label=1
type	name=search_specimen_id	2349
type	name=search_bx_no	B2002
click	name=search	
type	name=block_id	1
click	name=add_block	
select	name=block	label=1
type	name=slice_no	1-4
click	name=add_slice	
select	name=from_slice	label=1
select	name=to_slice	label=3
click	name=print_barcode	
assertNotVisible	id=error-report	
type	name=search_specimen_id	15831
click	name=search	
click	name=add_block	
select	name=block	label=1
type	name=slice_no	1-3
click	name=add_slice	
select	name=from_slice	label=1
select	name=to_slice	label=3
click	name=print_barcode	
assertNotVisible	id=error-report	
type	name=search_specimen_id	12563
click	name=search	
click	name=add_block	
select	name=block	label=1
click	name=add_slice	
select	name=from_slice	label=1
select	name=to_slice	label=3
click	name=print_barcode	
assertNotVisible	id=error-report	

Table 10.2: Print test case

Update_Test		
open	/dbimport/main	
click	xpath=(//button[@type='button'])[4]	
clickAndWait	link=Update metainfos of scanned images	
type	name=input_file	C:\tmp\ScanData_Update.xml
click	id=submitbutton	
waitForText	class=information	3 image(s) updated.
verifyText	class=information	3 image(s) updated.
click	css=div.modal-footer > button.btn.btn-default	
click	xpath=(//button[@type='button'])[4]	
clickAndWait	link=Update annotations of scanned images	
type	name=input_file	C:\tmp\Annotation_169_Update.xml
click	id=submitbutton	
waitForText	class=information	6 annotation(s) updated.
verifyText	class=information	6 annotation(s) updated.
click	css=div.modal-footer > button.btn.btn-default	
click	xpath=(//button[@type='button'])[4]	
clickAndWait	link=Update annotations of scanned images	
type	name=input_file	C:\tmp\Annotation_182_Update.xml
click	id=submitbutton	
waitForText	class=information	6 annotation(s) updated.
verifyText	class=information	6 annotation(s) updated.
click	css=div.modal-footer > button.btn.btn-default	
click	xpath=(//button[@type='button'])[4]	
clickAndWait	link=Update annotations of scanned images	
type	name=input_file	C:\temp\Annotation_193_Update.xml
click	id=submitbutton	
waitForText	class=information	6 annotation(s) updated.
verifyText	class=information	6 annotation(s) updated.
click	css=div.modal-footer > button.btn.btn-default	

Table 10.3: Update test case

Analysis_Test		
open	/analysis/main	
click	xpath=(//button[@type='button'])[6]	
clickAndWait	link=Correlate data from study	
select	name=study	label=1
select	name=x_axis	label=Age_at_study_entrance
select	name=y_axis	label=Biopsy_Gleason1
click	name=correlate	
waitForVisible	id=plot	
assertVisible	id=plot	

Table 10.4: Analysis test case

Export_Test		
open	/dbexport/main	
click	xpath=(//button[@type='button'])[5]	
clickAndWait	link=Export study from database	
select	name=study	label=1
click	document.export.elements['attribute_fieldsets[0]'][1]	
click	document.export.elements['attribute_fieldsets[2]'][1]	
click	document.export.elements['attribute_fieldsets[3]'][1]	
click	document.export.elements['attribute_fieldsets[4]'][1]	
click	document.export.elements['attribute_fieldsets[5]'][1]	
click	document.export.elements['attribute_fieldsets[6]'][1]	
click	document.export.elements['attribute_fieldsets[7]'][1]	
click	document.export.elements['attribute_fieldsets[8]'][1]	
click	document.export.elements['attribute_fieldsets[9]'][1]	
click	document.export.elements['attribute_fieldsets[10]'][1]	
click	document.export.elements['attribute_fieldsets[11]'][1]	
click	document.export.elements['attribute_fieldsets[12]'][1]	
click	document.export.elements['attribute_fieldsets[13]'][1]	
click	document.export.elements['attribute_fieldsets[14]'][1]	
click	document.export.elements['attribute_fieldsets[15]'][1]	
click	document.export.elements['attribute_fieldsets[16]'][1]	
click	document.export.elements['attribute_fieldsets[17]'][1]	
click	document.export.elements['attribute_fieldsets[18]'][1]	
click	name=exp_specimes	
click	name=exp_blocks	
click	name=exp_slices	
click	name=exp_images	
click	id=submitbutton	
waitForText	css=span.progress-value	regexp:êxporting study.*5[0-9]%.*$
waitForText	css=span.progress-value	regexp:Êxport complete!$
waitForText	css=h1	Export completed without errors.
assertText	css=h1	Export completed without errors.

Table 10.5: Export test case

Remove_Test		
open	/dbmngt/main	
click	xpath=(//button[@type='button'])[7]	
clickAndWait	link=Remove study from database	
select	css=select[name="study"]	label=1
click	name=delete_study	
click	//input[@value='Delete']	
waitForText	css=div.errors	None
assertText	css=div.errors	None
selectWindow	null	
clickAndWait	css=#remove-report > div.modal-dialog > div.modal-content > div.modal-footer > button.btn.btn-default	

Table 10.6: Remove test case

10.3 Attached materials

This report is related to the following materials which are deployed together:

- Project definition (in German)
- User-manual for the PSR system
- Acceptance test
- Source code of the PSR system
- Source code of the File Organizer
- SQL script for creating the inventory database
  (includes data of all currently scanned images)
- Source code documentation of the PSR system
- Source code documentation of the File Organizer
- Selenium test cases
- Excel report about the scanned tissue slides