UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science
Computer Science speciality

Martin Vels

Software Tool for Validation of Analytical HPLC Procedures

Master Thesis (30 EAP)

Supervisor: Prof. Marlon Dumas, PhD
Supervisor: Koit Herodes, PhD

TARTU 2013

Abstract

There is a steady demand, both from academia and industry, for efficient and reliable procedures to analyze various substances by means of High-Performance Liquid Chromatography (HPLC) equipment. To make sure these procedures are fit for the purpose they were designed for and also as reliable and widely usable as possible, they have to be validated against relevant validation guidelines. This validation process can be time consuming and tedious work, which contains many steps including reading lengthy and often general guidelines, deciding which parts of the guideline are relevant, measuring certain characteristics, performing certain statistical calculations on the gathered data and finally generating a validation report. As this work is done manually, it wastes a lot of valuable time and money which could be spent on improving the actual analytical procedure.

To alleviate the current situation a working prototype of a software tool was created during this Master's thesis which allows end-users to reduce time and effort needed for analytical procedure validation. The prototype implements one specific validation guideline (The International Conference of Harmonization Harmonized Tripartite Guideline), and allows users to create validations, enter correct values for the specific characteristics, perform statistical calculations on the entered data and generate report based on the previously entered data and calculations. The tool has been designed with extensibility in mind. Specifically, additional guidelines can be added via configuration files while additional input validation and report generation components can be plugged into the tool in order to cope with additional requirements.

Extensibility is to large extent achieved by borrowing ideas from dynamic forms specification models, which allow field visibility and form completion conditions to be defined by at the level of individual fields or groups of fields.

As there is an actual need for the software tool that was created during this thesis, it will be developed further by adding new validation guidelines, implementing additional functionality and improving the overall usability of the software.

Contents

Acknowledgements

First, I would like to thank my professor, Marlon Dumas, for suggesting the idea as well as giving me all the pointers on what direction to move. Next, I would like to thank Koit Herodes, Karin Kipper, Riin Rebane and Anneli Kruve from Institute of Chemistry for helping me to understand the problem domain and testing the software tool. And last, but definitely not the least, I would like to thank my employer, Siim Vips and my family for giving me the opportunity to work on this thesis by being flexible and understanding.

The supervisor and domain experts involved in this Masters project have been supported by the EU Regional Development Funds via the Estonian ICT Program.

Introduction

There a many different fields in industry and academia where there is a need to perform some kind of chemical analyses. It may be analysis that tries to determine if there are some substances present in the urine or blood of an athlete, or some analysis that tries to measure the active substance in a drug. To be able to successfully perform such analyses, there is a need for a certain protocol, that determines what conditions need to be met and what steps have to be followed. This protocol is called analytical procedure. To make such analytical procedure as widely usable and reliable as possible it has to be validated. The aim of validation is to ascertain that the procedure is fit for purpose, i.e. it will measure what it is meant to measure. To validate some analytical procedure there are special guidelines available.

Many of these analyses can be done using chromatography and more precisely high-performance liquid chromatographs (HPLC). Now, imagine a chemistry laboratory with very expensive and high-tech equipment. In particular one with the HPLC. As you would expect, these machines are controlled by the computers. Humans are interacting with the equipment via some specialized software. Now that all this has been taken care of, still a lot of manual and tedious work needs to be done by humans when it comes to validating some analytical procedure.

First, a lengthy validation guideline has to be read to get a general idea on how some analytical procedure has to be validated, next there is a need to measure and log certain characteristics during the validation procedure, then there is a need to perform certain calculations with the previously collected data to determine if some criteria are met. And finally there is a need to produce a report, that could be attached to a analytical procedure description, to show that it really is fit for a given purpose.

Unfortunately, all this work needs to be done manually, meaning that instead of spending their valuable time to improve the analytical procedure itself, chemists are doing a lot of manual labor to read the guidelines, finding what are the right characteristics that they need to measure, log that data, perform some calculations on that data and finally create a report by copying the previously gathered data, again manually, to some document. If something needs to be adjusted or changed, it means that a lot of repeated work occurs and possibility to make a mistake grows.

Solution to these problems is of course to implement a software tool that would help chemists to reduce the amount of such manual work. The idea of the software tool was initially posed by University of Tartu's Institute of Chemistry (Koit Herodes). Based on an initial vision and interviews with domain experts in this institute, an initial high-level architecture was defined, based on two main components: one for data entry management and one for calculations and report generation. This architecture was later refined by specifying concrete models for dynamic form specification (for data entry) and report generation, which are described in this thesis.

The objective for this thesis was to create a working prototype of a software tool that would make the validation process of analytical procedure less time consuming and help end-users, who might not be experts on the validation guidelines, still being able to successfully perform such a validation task by helping the software to guide them. This prototype is designed and built from scratch in the PHP programming language and Symfony web-framework. Initial prototype only supports The International Conference of Harmonization (ICH) Harmonized Tripartite Guideline [1], but is built so that adding new guidelines will be relatively easy and every part of the application would be extensible as easily as possible.

This thesis is divided into 5 chapters. In first chapter I give a short overview of the problem domain and background of the analytical procedure validation. In second chapter I will introduce an existing commercial software for analytical procedure validation and some of the existing ideas about what has been done in the field of dynamic form generation and dynamic questionnaires. Next, in chapter three I will describe my models for form definitions and report generation. In chapter four, the actual implementation of the software tool is described. Thesis is finished with the chapter five, where conclusive thoughts are presented together with the ideas for future work.

Chapter 1

Background and Problem Domain

In this chapter I will introduce the problem domain. I will give a short overview of liquid chromatography and validation of analytical procedures.

1.1 Liquid Chromatography

Chromatography is a process that makes it possible to separate components of mixtures from each other. This is based on the distribution of components under inspection between two immiscible phases. These phases are called stationary phase that lies in the column and mobile phase that is pushed through stationary phase. The components of the sample have different affinity for the phases, thus their differential migration causes the separation of components [5].

In liquid chromatography the mobile phase is liquid. Silica gel is used as stationary phase support. In reversed phase liquid chromatography stationary phase is non-polar and mobile phase is polar, usually mixture of organic solvent (e.g. methanol) with water [5].

Columns are usually 3-25 cm in length and with internal diameter of 4.6 mm or less. Column with small diameter reduces significantly the amount of mobile phase used for elution and also increases the resolution and sensitivity as the diffusion inside the column is as small as possible. Column thermostat improves the reproducibility and allows using temperature as additional parameter for resolution optimization. Smaller columns and smaller amount of mobile phase exiting the column allows combining liquid chromatography with mass spectrometer [5].

Nowadays the stationary phases consist of spherical shaped micro-particles or porous monolithic material to improve selectivity and resolution. The efficiency of chromatographic process is inversely-proportional to the size of the particle of stationary phase. The main obstacle for not using even smaller particles in the stationary phase is the pressure needed to achieve the constant flow of mobile phase through tightly packed column. The pressure in the system depends on the flow rate, viscosity of the mobile phase and the particle size of the stationary phase. HPLC (High Performance Liquid Chromatography) systems available are using maximum pressure of 400 bar, UHLPC (Ultra-HPLC) systems allow one order of magnitude higher pressure to be used [5].

In HPLC, the main principles and separation mechanisms remain the same, but the speed, sensitivity and resolutions are improved. The main advantage is significant reduction of time and amount of solvent used for analysis. Also the amount of time needed for method development experiments, balancing of the column, stabilization of the gradient elution of the column and method validation is reduced significantly [5].

A conceptual diagram of the liquid chromatography is shown in figure 1.1.

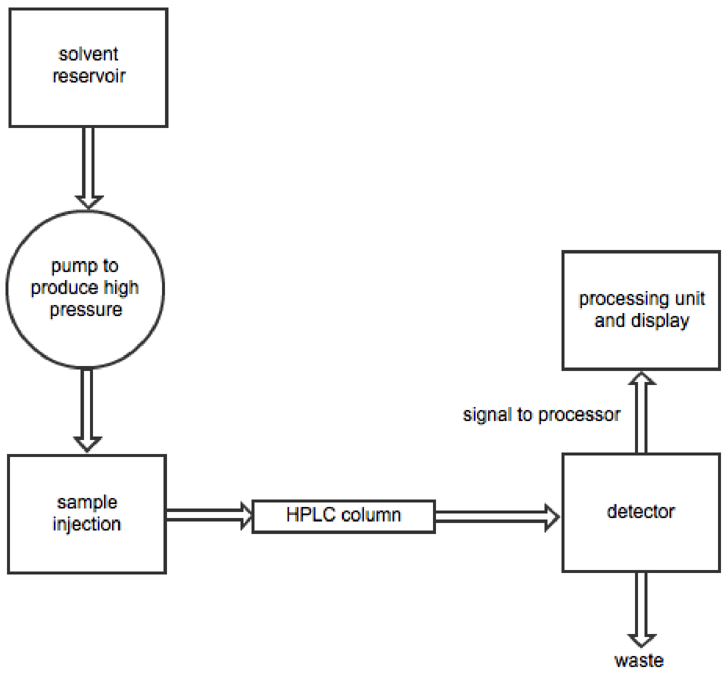


Figure 1.1: Conceptual diagram of liquid chromatography

1.2 Validation of Analytical Procedures

The term analytical procedure refers to the way of performing the analysis. It should describe in detail the steps necessary to perform each analytical test [1].

Validation is a process for determining if the given analytical procedure is fit for purpose, i.e. suitable for the analysis that it is intended for. The fitness for purpose is determined by the parameters characterizing the goodness of the analysis procedure [6].

There are four most common types of analytical procedures:

1. Identification tests;
2. Quantitative tests for impurities' content;
3. Limit tests for the control of impurities;
4. Quantitative tests for the active moiety in samples of drug substance or drug product or other selected component(s) in the drug product.

Identification tests are intended to ensure the identity of an analyte[1] in a sample. Testing for impurities can be either a quantitative test or a limit test for the impurity in a sample. Assay procedures are intended to measure the analyte present in a given sample [1].

[1]analyte - a substance or chemical constituent that is of interest in an analytical procedure

There are several characteristics (validation parameters) that characterize an analytical procedure:

1. Accuracy;
2. Precision;
   (a) Repeatability;
   (b) Intermediate Precision;
3. Specificity;
4. Detection Limit;
5. Quantitation Limit;
6. Linearity;
7. Range;
8. Robustness;

Often the validation only determines some of the aforementioned characteristics. The main purpose of the validation is not to determine all these characteristics but to get confirmation that the given analytical procedure is performing as expected. And if the procedure is not performing as expected, to get information about how the procedure needs to be modified to make it perform as expected [1, 6].

Next I will give a brief description of the characteristics to get better idea what kind of info needs to be determined.

1.2.1 Specificity

The specificity of an analytical procedure characterizes how well the procedure is capable of determining the analyte in the sample without other components interfering. Specificity is probably the most important characteristic that characterizes the analytical procedure. In chromatography the specificity is expressed via resolution criterion R_s[6]. It can be calculated from retention time and peak width at half-height, using following formula: $R_s = \frac{1.18(t_2-t_1)}{W_{0.5,1}+W_{0.5,2}}$ [5], where $t_1$is retention time of the first peak, t_2 is the retention time of the second peak and $W_{0.5,1}$ and $W_{0.5,2}$ are peak widths at half height of the first and second peak respectively. $R_s$is used to express how well is the analyte peak separated from a peak of (possibly) interfering compound.

1.2.2 Accuracy

The accuracy of an analytical procedure expresses the agreement between the value which is accepted either as a conventional true value or an accepted reference value and the value found [1].

1.2.3 Precision

The precision of an analytical procedure expresses the agreement between a series of measurements obtained from multiple sampling of the same homogeneous sample under the prescribed conditions. Precision is usually expressed as the variance, standard deviation or coefficient of variation of a series of measurements [1]. Precision may be considered in three levels:

1. Repeatability - expresses precision under the same operating conditions over a short interval of time;

2. Intermediate precision - expresses within-laboratories variations: different days, different analysts, different equipment, etc.;

3. Reproducibility - expresses the precision between laboratories;

1.2.4 Detection Limit

The detection limit of an analytical procedure expresses the minimum amount of analyte in the sample that can be reliably detected and identified by the given analytical procedure [6]. Detection limit is related to both the signal and the noise of the system and usually is defined as a peak whose signal-to-noise ratio is at least 3:1 [5].

1.2.5 Quantitation Limit

The quantitation limit of an analytical procedure expresses the lowest amount of analyte in the sample that can be quantitatively measured by the given analytical procedure [6]. Similarly to detection limit, the quantitation limit is also related to the signal and the noise of the system and is usually detected as a peak whose signal-to-noise ratio is at least 10:1 [5].

1.2.6 Linearity

The linearity of an analytical procedure is its ability to obtain test results which are directly proportional to the concentration of analyte in the sample [1]. The linearity of a method is a measure of how well a calibration plot of response vs. concentration approximates a straight line, or how well the data fits to the linear equation: $y = mx+b$ where y is the response, x the concentration, m the slope, and b the intercept of a line fit to the data. Ideally, a linear relationship ($b \approx 0$) is preferred because it is more precise, easier for calculations and can be defined with fewer standards [5].

1.2.7 Range

The range of an analytical procedure is the interval between the upper and lower concentration of analyte in the sample for which it has been demonstrated that the analytical procedure has a suitable level of precision, accuracy and linearity [1].

1.2.8 Robustness

The robustness of an analytical procedure is a measure of its capacity to remain unaffected by small, but deliberate variations in method parameters and provides an indication of its reliability during normal use [1].

1.3 Scope and Requirements

Validation of an analytical procedure is usually done manually where user picks up the guideline document which are usually written in quite general form and determines which characteristics need to be measured according to the type of the analytical procedure. Then he needs to plan and carry out experiments according to the validation guidelines, e.g. analyze material which contains analyte at foreseen concentration level. Next he needs to make some calculations with the obtained data according to the description in the guideline. Using these calculations, user can determine if some needed criteria are met and thus understand if the analytical procedure in question is fit for the purpose it was meant and whether the procedure is actually validated.

Finally based on the results user needs to manually create a validation report where all the needed characteristic data is shown together with the calculations and criteria. As can be seen this process is not very user-friendly and in case of some adjustments are needed in the analytical process, a lot of work related to calculations and report creations need to be redone. The main purpose of the software that was created, is to reduce the work the user has to do during analytical procedure validation and thus concentrate more on the actual analytical procedure itself. Here are the main goals that the software had to achieve:

1. Reduce the time user needs to spend to get familiar with the exact validation guideline and describe all the needed criteria in the software so that user only needs to choose what type of procedure he has and choose characteristics that needs to be measured.

2. Guide user in regard of what data needs to be entered exactly and in what form (text, numbers, diagrams etc.).

3. Make all the calculations needed for a given characteristic, based on the input data and give user idea whether needed criteria are met or not.

4. Create validation reports automatically based on the pre-defined templates and sections user can freely choose from.

5. Make it possible for several users to access the same procedure validation and thus divide workload between several users or make it possible for supervisors to access the work of the students/employees without waiting for the final report to be ready.

Also it was important to make the software as dynamic as possible from the point of view of a developer, so that it would be possible to easily add new validation guideline descriptions, create new report templates and expand the overall functionality if such need arises.

1.4 Roundup

In this chapter I gave an overview of the background of the field of chromatography and analytical procedure validation. Short description of the main characteristics used during validation were also presented. Finally, the scope of the software was described with general requirements.

Chapter 2

Related Work

In this chapter I will give an overview of the only software tool that I could find that is currently available for the exact task of analytical procedure validation. I will also discuss the solutions that are available for dynamic form generation and decision making.

2.1 Existing solutions

As the target audience who are dealing with the analytical procedure validation is not too wide and consists mainly of people who have different kind of expertise than creating software to solve their problems, it was not surprising that there were not a lot of such existing software available.

I was able to find one commercial software product, called VALIDAT[1]. This software product is developed by a company called ICD and they have several specialized software tools available for the different chemical laboratory related tasks. VALIDAT is for analytical method validation. It supports several guidelines, has many characteristics available, also contains diagram and report generation. It runs on the Microsoft Windows platform and on Microsoft SQL Server or Oracle SQL Server [3].

The main advantages of the new software that was created during this theses, compared to the existing commercial solution, were following: being able to run it on the web-browser thus making it available on virtually all operating systems, not only limiting to Microsoft Windows. Build it with open-source tools and to make it open-source, so everyone interested could extend it further. Design it the way it would be easily expandable, thus making it possible to add new validation guidelines and reports later by only describing new guidelines in special configuration files.

2.2 Dynamic form generation

As one of the main parts of the software tool that I was implementing consists of dynamic form generation for various characteristic fields, I was looking for a solutions that were already present. Usually forms are generated manually by the software developer using a web-framework. This makes it more convenient for the developer to both generate form elements as well as validate the data entered by the end-user and finally retrieve the data and persist it in database. However, this approach still means

[1]`https://www.icd.eu/produkte/methodenvalidierung-software.html`

that form generated are fairly static in the sense that developer needs to describe in the code what fields need to be present in the given form. Depending on the programming language and framework used, changes in such forms could take significant amount of developer's time as well as introduce a risk of creating an error in the program code while making a change. One idea on how to solve this issue would be using some domain specific language (DSL) that would describe the form elements and all the related meta-data about the form fields. One of the examples of such a solution is XForms that I will introduce in next section.

As the fields that should be shown on the dynamic forms may depend on the previous decisions that user makes during validation creation and setup phase, it is important to specify which exact fields are present in which dynamically generated forms. In [35] the idea of pre-conditions was introduced. The idea is to use preconditions which determine if some user interface element (widget) should be visible or not. Preconditions are presented as boolean expressions which can have true (widget visible) or false (widget not visible) values. There is a global Current State Blackboard (CSB) present in the system where predicate manager will be writing messages if any of the predicates will become true and remove these messages when predicate becomes false. Next, the widget manager will read the CSB during the view rendering and determine whether the fields that should be present in that view satisfy the boolean conditions of the predicates and show only these fields to end-user. I will be using similar idea in my application, where depending on the decisions user will make during validation configuration step, certain fields will be shown (or not shown) in later steps, where user starts to enter data for certain analytical procedure characteristic.

2.3 XForms

XForms is an XML application that was designed to supersede the existing HTML-forms. XForms is intended to be integrated into other markup languages as XHTML, ODF or SVG. The main idea behind XForms is to separate the concerns using widely-known design pattern called Model-View-Controller (MVC). There is a model, that contains formulas for data calculations and constraints, data type and property declarations and data submission parameters. Next, the view layer contains the actual form controls that end-user interacts with. And finally, the controller binds the view and model together, orchestrating all the interactions between view and model as well as data submissions [14].

Next I will give a simple example of a document containing XForms components and introduce some of the concepts of XForms, that were helpful when I started creating the software tool for analytical procedure validation.

2.3.1 Document containing XForms elements

A simple XHTML document containing XForms components would look as following:

```
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:ev="http://www.w3.org/2001/xml-events"
  xmlns:d="http://www.mydata.com/xmlns/data">
  <head>
    <title/>
```

```
 8      <xf:model>
 9        <xf:instance id="user_profiles">
10            <d:user>
11              <d:firstname>Homer</d:firstname>
12              <d:middleinitial>J</d:middleinitial>
13              <d:lastname>Simpson</d:lastname>
14          </d:user>
15        </xf:instance>
16        <xf:submission action="http://example.com/submit" method="post
             "/>
17      </xf:model>
18    </head>
19    <body>
20      <xf:input ref="d:firstname">
21        <xf:label>First Name: </xf:label>
22      </xf:input>
23      <xf:input ref="d:middleinitial">
24        <xf:label>Middle initial: </xf:label>
25      </xf:input>
26      <xf:input ref="d:lastname">
27        <xf:label>Last name: </xf:label>
28      </xf:input>
29      <xf:submit>
30    </body>
31  </html>
```

Model can be seen in the head-tag of the document (lines 8-17). It consists of an instance of user with it's own namespace "d" (lines 10-14). In the body of the document the actual form-controls are defined (lines 20-28) which are referring to model. So the ref="d:firstname" points to the firstname tag in the model user-instance. This is an abbreviated XPath reference, relative to the default context. It would be equivalent to following xml-snippet:

```
<xf:input ref="/d:user/d:firstname">
  <xf:label>First Name: </xf:label>
</xf:input>
```

If there would be multiple models present, it is possible to explicitly specify the model as well:

```
<xf:input ref="/d:user/d:firstname" model="user_profiles">
  <xf:label>First Name: </xf:label>
</xf:input>
```

Finally there is a submit-control (line 29) that performs the actual form submission to the url that was defined in the model (line 16) [16].

2.3.2 Output fields

In addition to input fields that user can interact with by entering some data into them or choosing some value from the given list of values, there is also a concept of output fields present in XForms. These fields are non-editable components that can reflect the value of a given item in the data-model (using the ref-attribute) or show a result of some calculation (using XPath notation) and using the value attribute. A simple example on how this would look like in XForms syntax is following [17, 18]:

```
<xf:model>
  <xf:instance>
```

```
    <d:data a="5" b="6"/>
  </xf:instance>
</xf:model>
<xf:input ref="@a"/> + <xf:input ref="@b"/> =
<xf:output value="@a + @b"/>
```

2.3.3 Required fields

XForms introduces a special mean to specify that some of the model properties are required before the instance data is submitted. This is achieved by associating a required property with a certain element ("lastname" in given example) [18]:

```
<xf:instance id="user_profiles">
  <d:user>
    <d:firstname>Homer</d:firstname>
    <d:middleinitial>J</d:middleinitial>
    <d:lastname>Simpson</d:lastname>
  </d:user>
</xf:instance>
<bind nodeset="/d:user/d:lastname" required="true()"/>
```

2.3.4 Repeated fields

Another useful concept that XForms provides is the possibility to define repeated structures. This means that it is possible to make a single field or a collection of fields to be easily repeated in the document. E.g. in case of shopping cart, this allows end-user to dynamically add new rows to his shopping cart or remove existing ones. An example, which is not too trivial and also contains the concept of nested repeats is following [17]:

```
<xf:repeat nodeset="instance('dataStore')/d:record">
  <fieldset>
    <legend><xf:output value="concat(d:identity/d:surname,
       ', ',d:identity/d:firstname)"/></legend>
    <xf:repeat nodeset="d:identity">
      <h3>Identity</h3>
      <xf:input ref="d:firstname">
        <xf:label>First Name: </xf:label>
      </xf:input>
      <xf:input ref="d:middleinitial">
        <xf:label>Middle Initial: </xf:label>
      </xf:input>
      <xf:input ref="d:surname">
        <xf:label>Surname: </xf:label>
      </xf:input>
    </xf:repeat>
    <xf:repeat nodeset="d:address">
      <h3>Address</h3>
      <xf:input ref="d:street">
        <xf:label>Street: </xf:label>
      </xf:input>
      <xf:input ref="d:city">
        <xf:label>City: </xf:label>
      </xf:input>
      <xf:input ref="d:region">
        <xf:label>Region: </xf:label>
      </xf:input>
```

```
28       <xf:input ref="d:country">
29         <xf:label>Country: </xf:label>
30       </xf:input>
31     </xf:repeat>
32   </fieldset>
33 </xf:repeat>
```

It can be seen that there are two repeats (line 5-16 and 17-31) nested inside another repeat (lines 1-33).

2.3.5 Rendering of XForms

Unfortunately none of the contemporary browsers like Firefox, Chrome, Internet Explorer or Safari support XForms directly, thus some special renderers are needed. There are in principle two kinds of XForms renderers:

- browser plugins that can parse and render the given document containing XForms elements;
- specific external libraries that transform given XForms elements into according HTML-form elements and Javascript that would provide needed functionality. These special libraries in turn can be divided into client-side and server-side libraries.

Regarding the browser plugins, there aren't any of them available any more for current browsers. E.g. last working plugin for Mozilla was for the version 3.6 [22]. Author of the Mozilla XForms extension comes to a conclusion that XForms as browser plugin is dead and predicts that the future of web forms is HTML5 together with its surrounding technologies [21].

An alternative solution to browser plugin would be some transforming library on client side or server side, that would transform the XForms into HTML and JavasScript to achieve the same functionality as browser plugin would. According to a survey done in [23] these libraries are not very actively developed or supported any more.

There is a commercial product available for server-side XForms handling, called Orbeon Forms[2] which is actively developed at the moment and has reached version 4. It has nice graphical form builder available as well. However, there seems to be one important shortcoming in latest Orbeon Forms version 4.2 regarding the repeated elements. In particular, it doesn't support nested repeats [25].

2.4 Dynamic Questionnaire Systems

In addition to dynamic form generation with some pre-defined fields there is also a need to specify if some of these fields should not be shown to end-users in case user made some preliminary selection between choices given. E.g. in case of analytical procedure validation, values that user needs to enter for a specific characteristic may vary depending on what kind of decision user made previously while specifying the existing data sources he has.

There has been a lot of research done in the field of variability management. Variability of an information system can be captured as a collection of parameters, features

[2] `http://www.orbeon.com/`

or choices. These choices determine the actions that should be performed to derive an individualized model or system from the generic one. In case of configuration of business process models, such actions may correspond to removing a fragment of a process model [24]. E.g. in case of payment system, when user has choices on how to pay for a purchase, after he makes a certain choice to pay only after the goods are delivered, he will not be presented with the payment details like credit-card choices or internet bank links.

In [24] the configuration model contains questions that capture the way how the variability of a generic system is resolved at configuration time. Each question contains facts that can be answered true or false. These facts encode the variability of the system. Based on the answers given to these facts, certain actions can be performed on the generic system to derive a specific system. An overview of such a framework can be see on figure 2.1.

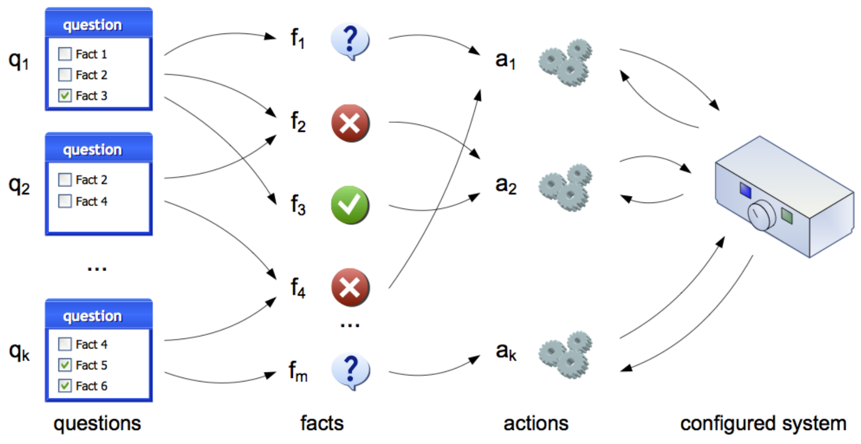


Figure 2.1: Configuration framework overview [24]

2.5 Roundup

In this chapter I gave an overview of existing software solution that is currently commercially available for procedure validation. Next, I introduced XForms as an example of how dynamic forms can be described. And finally I introduced a research done in the field of dynamic questionnaire systems.

Chapter 3

Forms and Report Definition

In this chapter I will introduce the model for defining dynamic forms for procedure validation and also the model for defining reports.

3.1 Model for defining the dynamic forms for procedure validation

To be able to perform analytical procedure validation, there is a need to gather certain characteristic parameters and perform certain calculations on them. This means, that there is a need for data entry form where end-user can enter needed data. As the data needed for every characteristic is different, this means that the forms containing the data fields are also different. It is not reasonable to describe these forms in the program code itself, but rather use an external definitions that are used by the software tool to render correct fields in correct characteristic forms. Based on the ideas from XForm I created my own model for defining these forms. The class diagram of the model can be seen on figure 3.1.

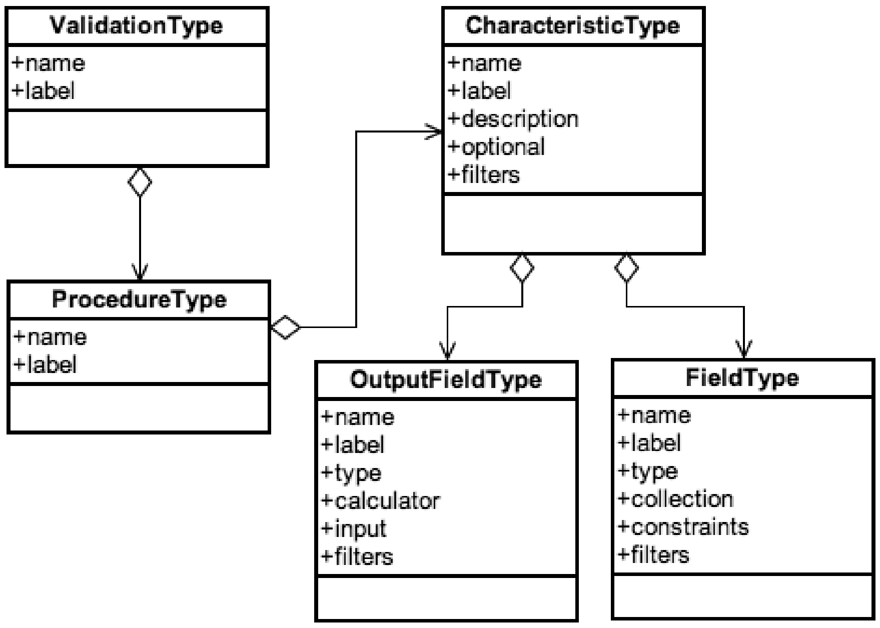


Figure 3.1: Procedure Validation Model Class Diagram

The model for defining dynamic forms for procedure validation consists of following parts:

1. Validation definition, which describes a particular validation guideline and contains arbitrary amount of analytical procedures;
2. Analytical procedure definitions, each of which describe characteristics relevant to that particular analytical procedure;
3. Characteristic definitions, each of which describe which particular data-fields are relevant to that certain characteristic;
4. Input and output data-field definitions, where the exact data fields are defined together with all the needed meta-data for validation rules, data types, etc.

All the definitions of the meta-data that is used to generate forms are stored into configuration files written in Javascript Object Notation (JSON) format. JSON is a lightweight data-interchange format. It is both, easy to read by humans as well as easy to parse and generate by computers [12]. Actual data that is entered by the end-user during data entry is stored in the database. Configuration files are organized into following structure:

1. Validation type configuration;
2. Procedure type configuration;
3. Characteristic type configuration;

This structure is hierarchical, meaning that the top level is Validation Type Configuration, which may contain many Procedure Type Configurations and any of which may contain many Characteristic Type Configurations. In following subsections I will give an overview of these configuration files.

3.1.1 Validation Type Configuration

Validation type configuration acts as the main entry point to the whole validation procedure configuration. A sample validation type configuration file looks as following:

```
{
  "name": "ich",
  "label": "ICH Validation",
  "procedure_types": [
    {
      "include": "/procedure_types/assay/procedure.json"
    },
    {
      "include": "/procedure_types/identification/procedure.json"
    },
    {
      "include": "/procedure_types/impurity_quantitation/procedure.
          json"
    },
    {
      "include": "/procedure_types/impurity_limit/procedure.json"
    }
  ]
}
```

There is a name of the current validation type (line 1), which needs to be unique. There is a label (line 3), that can be anything and is only used inside the application to show to end-users. And then there is an array of procedure types (lines 4-17). These procedure types are included as paths (lines 6, 9, 12 and 15) to specific procedure type configuration files.

3.1.2 Analytical Procedure Type Configuration

Analytical procedure type configuration contains the definition of a specific analytical procedure and acts as a container for specific characteristics relevant to that analytical procedure.

A fragment of a sample Procedure Type configuration file looks as following:

```
{
  "name": "assay",
  "label": "Assay",
  "description": "Longer description for the assay procedure",
  "characteristics": [
    {
      "filter": [
        {
          "name": "filter_1",
          "label": "Visual Evaluation"
        },
        {
          "name": "filter_2",
          "label": "Based on signal-to-noise"
        },
        {
          "name": "filter_3",
          "label": "Standard deviation of the response and the slope",
          "filter": [
            {
              "name": "filter_3_1",
              "label": "Using blank"
            },
            {
              "name": "filter_3_2",
              "label": "Based on the calibration curve",
              "filter": [
                {
                  "name": "filter_3_2_1",
                  "label": "Using residual standard deviation of a regression line"
                },
                {
                  "name": "filter_3_2_2",
                  "label": "Using y-intercepts of refression lines"
                }
              ]
            }
          ]
        }
      ],
      "include": "/procedure_types/assay/characteristic_types/detection_limit.json",
      "optional": true
43         }
44     ]
45 }
```

Similarly to Validation Type configuration the Procedure Type configuration also contains name (line 2) and label (line 3) elements. In addition there is also a description (line 4) element, that can contain arbitrary text that could be shown to end-user during the data entry.

Most important part of this configuration is of course the array of characteristic types (lines 5-44). There are include-elements, that contain path to specific characteristic configuration (line 41). There is also a boolean "optional" element (line 42) that is used to guide end-users during configuration part and indicates if this particular characteristic is usually needed for this particular procedure type or not.

Finally, there is the "filter" element (lines 7-40), which has recursive structure with "name", "label" and "filter" elements. The idea behind the filter is to enable end-user to make decisions about the validation procedure before he starts entering data. This acts as a decision tree, where user can choose one particular branch of the tree. As can be seen on the sample configuration, these are questions about a certain characteristic.

Based on selections that user makes with these filters he will only see a subset of all the possible data fields in data entry form that need to be filled. In other words, when a filter is applied, certain fields in the form are made invisible (because they are irrelevant given the data the user has entered so far). This is akin to XForms where a field or group of fields is visible (called "relevant" in XForms) if a certain condition is true.

Filter names need to be unique inside one analytical procedure configuration as these are used later to identify which exact fields should be shown to end-user. In next subsection I will describe field definitions in more detail, but the main concept of the set of fields and how filters are applied to them is shown in figure 3.2 . It can be seen, that there can be a lot of fields present in a certain characteristic, but only a subset of them is shown depending on what filter was selected by the end-user during the validation configuration process.

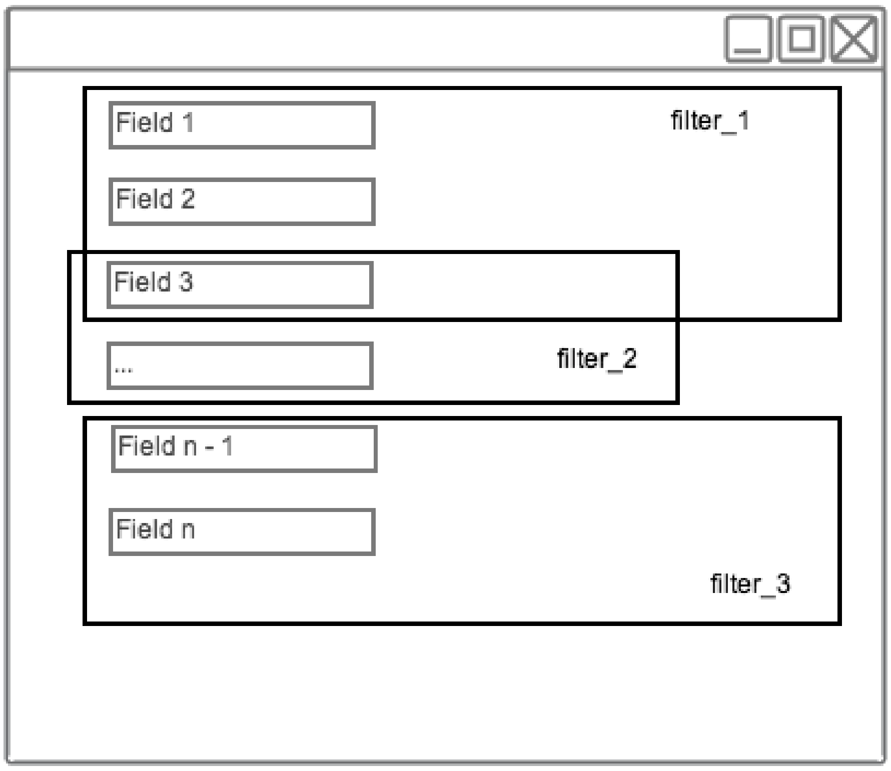


Figure 3.2: Concept of a set of fields and filters

3.1.3 Characteristic Type Configuration

Finally, there are characteristic type configuration files which contain actual data-field definitions that will be used to generate forms dynamically. A fragment of such a file is shown next:

```
 1 {
 2   "name": "linearity",
 3   "label": "Linearity",
 4   "description": "Analysis of substances with known concentrations",
 5   "fields" : [
 6     {
 7       ...
 8     }
 9   ],
10   "output_fields" : [
11     {
12         ...
13     }
14   ]
15 }
```

Similarly to validation type and procedure type configuration files, also this file contains "name" (line 2) and "label" (line 3) elements. Also there is a "description" element (line 4) for longer description that would be shown to end-user to guide him through data entry process.

Next, there is a "fields" element (lines 5-9) array which contains definitions of data fields that will be shown to end-user. And lastly there is an “output_fields” (lines 10-14) array, that contains fields that can not be entered by users, but can be shown to end-users and contain data that is calculated based on the data entered in the fields. I will give more detailed description of input and output fields in next two subsections.

3.1.4 Input Field Type Configuration

A sample input field definition is following:

```
 1 {
 2   "name": "signal",
 3   "label": "Signal",
 4   "type": "number",
 5   "required": true,
 6   "min_count": 1,
 7   "help": "Signal",
 8   "unit_type": "signal",
 9   "filters": ["filter_1"],
10   "options": [
11     {"name": "precision", "value": 4}
12   ],
13   "constraints": [
14     {
15       "class": "NotBlank"
16     },
17     {
18       "class": "Range",
19       "options": [
20         {"name": "min", "value": 0},
21         {"name": "max", "value": 999}
22       ]
23     }
24   ]
25 }
```

There are again “name” (line 2) and “label” (line 3) elements present, next the “type” (line 4) element determines what type of the field it is. There a many different types available, I will introduce these types that are relevant in the context of the software tool for validation:

- number - used for entering numerical data (e.g. retention time, signal strength, peak widths, etc.);
- text - used for textual data;
- textarea - used for longer textual data, possibly on multiple lines;
- file - used for uploading files from end-user computer, in the context of the software tool for validation, these will be mainly images of various diagrams;

Next there is an element “required” (line 5) that is used to inform end-user that this particular field is mandatory. Element “min_count” (line 6) is used for specifying what is the minimum amount of these fields that user needs to fill. In case this value is larger than one, end-user will be presented with additional buttons next to these fields that allow him to add more fields or remove existing ones dynamically. I will introduce the user interface more precisely in the next chapter.

This "min_count" constraint is akin to the "minoccurs" qualifier in XML Schema, which allows one to specify that a given element type should appear at least a certain amount of times in the form. During discussions with the domain experts, we did not find the need to define a constraint of type "max_count" (i.e. maximum possible number of elements).

Next, there is a “help” element (line 7) that is used to show a special tooltip next to a field. Following element “unit_type” (line 8) indicates what type of units should be entered to this particular field. This is important, as there a many different unit types available in different characteristics. End-user can specify the exact units for the particular validation during validation configuration.

Next, there is a “filters” element (line 9) that is an array of filters. The concept of filters was described in previous subsection and on figure 3.2. It is important to notice that one field can have many filters assigned to it, which means that depending on what choice end-user has made during configuration phase, this field will be shown in the particular characteristic form or not.

“Options” element (lines 10-12) can contain an array of objects which are used to specify certain properties of a field. E.g. in case of a number type-field, it is possible to specify what is the precision of that field i.e. how many digits after comma should be shown and stored in database.

Final element is “constraints” (lines 13-24) which can contain many different sub-constraints. Constraints are used to validate a field, this means that the data entered into field will be checked against the rules specified in the constraints array and only if the constraints are satisfied, the data will be stored in the database. This is akin to the "constraint" property in XForms, which allows one to specify a predicate that needs to be satisfied for the instance data associated with a field or group of fields in the form to be considered valid. Otherwise, an error will be shown to user next to the particular field and no data is stored. It is possible to create additional constraint classes if there will be need in the future, currently the existing constraint-classes that were provided by the web-framework Symfony that was used for implementing this software, were sufficient.

There is one more important concept present in input-fields, that is the concept of “multi_fields”, which is the equivalent of a "group" in XForms. A multi_field is a container of fields, this means that it is possible to create a block of fields that are grouped together and can be accessed as one in various calculators and report renderers. A sample of a multi_field configuration is following:

```
{
  "name": "chromatograms",
  "label": "Insert representative chromatograms",
  "help": "Insert representative chromatograms with appropriately labelled individual components",
  "type": "multi_field",
  "collection": true,
  "min_count": 1,
  "fields": [
    {
      "name": "chromatogram",
      "label": "Chromatogram",
      "type": "file",
      "help": "Chromatogram image",
      "constraints": [
        {
          "class": "Image"
        }
18          ]
19        },
20        {
21          "name": "description",
22          "label": "Description",
23          "type": "text"
24        }
25      ]
26  }
```

In the “type” element (line 5) there is a special field type called “multi_field”, also there is a “collection” (line 6) element present that indicates that this field should be repeated. This is a way to explicitly tell the form builder that this field needs to be repeated, this is useful when the minimum amount of such fields is 1 as in current example. If the “min_count”-element is larger than one, then the “collection” element is not necessary to be specified as the form builder assumes that itself.

Finally there is a “fields” array (lines 8-25) in this sample configuration that contains definitions of input fields. This kind of nesting of fields was important for the validation software tool, as there are many cases where user needs to enter certain data in blocks, e.g. chromatogram images together with the description of that image, or retention time together with the peak width at half-height. Now that these fields can be grouped together it will be both easier for the end-user to replicate these fields during data-entry as well as use the data from these fields later for calculations by the software itself.

3.1.5 Output Field Type Configuration

In addition to input fields there are also special output-fields. These fields are only there for displaying some info and can not be entered by the end-user (i.e. read-only). The values shown in these output-fields are calculated based on values that were entered in the other fields. This is akin to the "calculate" property in XForms, which allows one to specify that the value of a certain field is calculated based on that of other fields.

A sample output-field element is following:

```
1  {
2    "name": "residual_squared_sum",
3    "label": "Residual sum of squared",
4    "filters": ["filter_1"],
5    "calculator": "LinearityCalculator::getResidualSumSquared",
6    "input": [
7      "assay.linearity.substances_signal"
8    ]
9  }
```

First there is a “name” (line 2) and “label” (line 3) elements just like in input-field, next there is the “filters” (line 4) element, which acts similarly as in case of input fields. Next there is a “calculator” element (line 5), which points to a specific Class::method pair, that should be used to produce this particular output. And finally there is a “input” element (lines 6-8), which is an array of input and/or output fields. The name of the input-field has three parts: analytical procedure name, characteristic name and field name. As was mentioned earlier, all these names need to be unique in the context of a particular block. This way it is possible to uniquely identify any field present in the current validation.

3.2 Model for defining reports

In addition to dynamic form generation that was described in previous section, there is also a need for defining reports for a particular validation guideline. The idea is that each validation can have many reports bound to it. Each report entity has a specific report template assigned to it. This report template contains all the meta-data needed for generating a report. The overview of the report model is shown on figure 3.3.

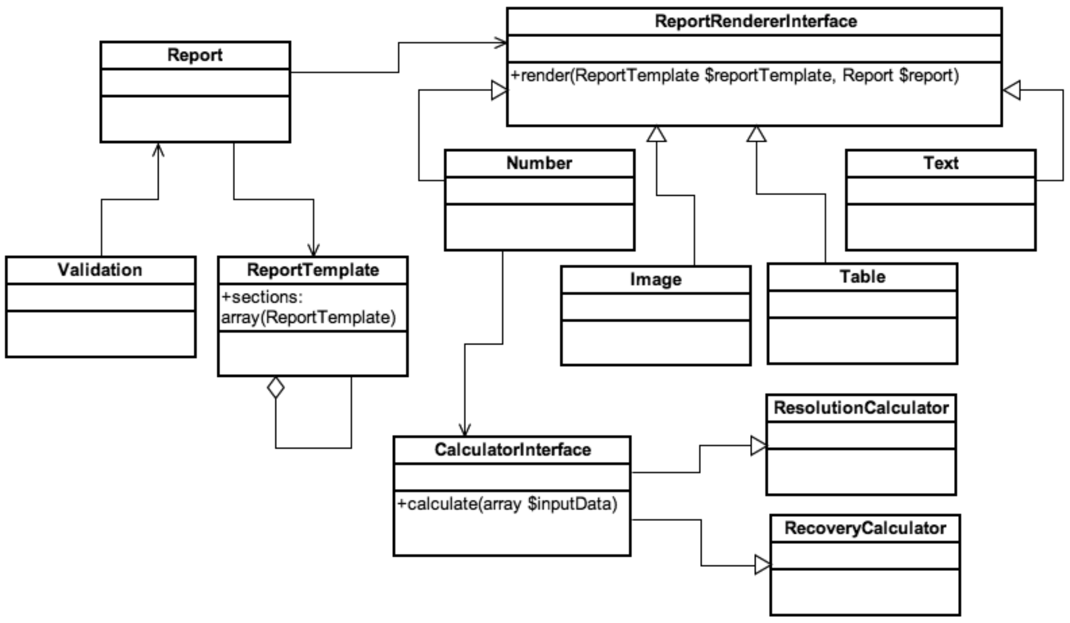


Figure 3.3: Report Model Class Diagram

A sample report template configuration file is following:

```
{
  "name": "unique_report_name",
  "label": "Report Name",
  "renderer": "Dummy",
  "input": [
  ],
  "sections": [
    {
      "name": "section_accuracy",
      "label": "Accuracy",
      "renderer": "Dummy",
      "input": "Accuracy for Assay",
      "sections": [
        {
          "name": "section_accuracy_results",
          "label": "Results",
          "renderer": "AccuracyTable",
          "input": {
            "head": {
               "concentration_level": "Concentration Level",
               "result": "Result",
               "average": "Average",
               "percent_recovery": "Percent Recovery"
            },
25            "body": {
26              "calculator": "AccuracyCalculator",
27              "input" : "assay.accuracy.current_results"
28            }
29          },
30          {
31            "name": "percent_recovery",
32            "label": "Average Percent Recovery",
33            "renderer": "Recovery",
34            "input": "assay.accuracy.current_results"
35          }
36       }
37    ]
38  }
39 }
```

First there is a “name” (line 2) and “label” (line 3) elements present in the configuration file. The name is there to uniquely identify a certain report and/or section, label is for showing to end-user. In addition to name and label elements there are some more important concepts present in this short fragment of a report template configuration file. First it can be seen that the configuration is recursive, meaning that there can be as many nested sections (lines 7-38) inside each sections as needed. Next thing to notice is the "renderer" element (lines 4, 11, 17 and 34), which holds the actual renderer class name that should be used to render this section of the report. Input key can hold a scalar as well as array with values and it will be used by the renderer. The way how input is handled is dependent on what exact renderer class is used. There are several renderers available like Dummy-renderer, which simply displays the value that was given in input element. There is a special renderer for showing images, renderer for showing data as table. There are also specific renderers that are used for specific characteristic to display the data for them.

Each report template can contain many sections, which are also instances of ReportTemplate class. Each of these sections can also contain many sections in a recursive manner. To be able to actually generate a report, each of these sections have a renderer class specified. Renderer classes are implementing ReportRendererInterface. There is one method, called "render" that takes in an instance of ReportTemplate and an instance of a Report. There can be arbitrary number of concrete renderer classes, in figure 3.3 there are only some of them shown like Number, Text, Image and Table renderers. The idea is that it will be very easy to add new renderer classes later when need arises. Each concrete renderer can in turn use as many external classes as needed to accomplish the result. E.g. number renderer uses CalculatorInterface, which can have as many concrete calculators as needed. In the diagram there are ResolutionCalculator and RecoveryCalculator classes present, which are doing some specific calculations needed to transform some input data into the form that is needed for some particular report section.

3.3 Roundup

In this chapter the conceptual overview of the the dynamic forms specification model and the report specification model. Both were described based on UML diagrams and sample instances.

Chapter 4

Implementation

In this chapter I will describe the architecture and implementation details of the application. First an overview of the used software stack is given followed by the general overview of the application architecture. Finally an overview of the main components of the user interface will be given.

4.1 Software Stack

In this section I will introduce the software that was used to develop the application for analytical procedure validation.

4.1.1 Framework for Back-End

As with any application there is a question, what tools to use to build it. As one of the requirements for application was that it has to be web-based it was natural choice to choose a framework for web projects. As I am familiar with the PHP-language it was natural choice to choose a framework written in PHP. I have been developing various web-applications for many years and have been using mainly commercial in-house built platforms which were not suitable for this project as one of the requirements was, that it needed to be built with open-source tools. After some comparison of numerous PHP frameworks I finally chose one called Symfony2[1].

Symfony is a PHP framework for web projects. It has been developed since year 2005 and reached version 2.2.1 as of this moment. It is a mature framework, meaning that it is well-written, well-tested and has an excellent documentation as well as large community behind it. All the code is open-source and freely available. It is very easy to install using dependency manager called Composer[2].

Symfony itself uses many other external software building blocks like the "Doctrine" Object-Relational Mapping (ORM) framework for persistence, Swiftmailer for sending out e-mails, Monolog for writing log into various targets, etc. Using the Composer, it is very easy to add new building blocks to your application and instantly gain new functionality. There is a huge database of available software packages hosted in Packagist[3] website. This is the central repository for all the various software packages that can be installed using Composer dependency manager.

[1]`http://symfony.com/`

[2]`http://getcomposer.org/`

[3]`https://packagist.org/`

4.1.2 Framework for Front-End

In addition to the PHP framework that was used for back-end side of the application, I also needed a way to manage the user interface. Again, there are many different solutions available, but after testing several of them I finally chose a front-end framework called Twitter Bootstrap[4] that is built by the developers behind one of the largest social networks Twitter. As the slogan of the Twitter Bootstrap says: "By nerds, for nerds" it is meant for developers who want to create solutions quickly and don't want to spend a lot of time tuning various HTML and CSS parameters. All of this has been done very convenient for the developers and all the components needed to create a modern and nice looking user interface are present. There is also very well written documentation that demonstrates possibilities of the framework together with examples how the end-result would look like [7].

Even though the Twitter Bootstrap contains many necessary building blocks for creating a user interface, there are some things that are not included or that are just not good enough. So I discovered an extended version of the Twitter Bootstrap framework developed by Jasny.net[5]. It has a lot of additional functionality added on top of Twitter Bootstrap, like possibility to make the whole table row to be a link, nicer looking file upload interface, more convenient way to display alerts, etc. [8].

4.1.3 Development Tools

In addition to software packages used to build the application itself, there are several other tools that were invaluable to help building software for the web. These tools include naturally an Integrated Development Environment (IDE). There are many different IDE-s available, both freeware and commercial. I was using a commercial IDE built by the company JetBrains and it is called PHPStorm[6]. It has Symfony framework support built in as well as the support for writing HTML, CSS and JavaScript.

In addition to IDE, I also used version control software (VCS) called git[7]. Git is free and open source distributed version control system, meaning that there really is no need for one central repository as other widely-used VCS-s like SVN. I used git both for version control as well as for simple deployment tool [28].

And finally, I was using a special tool for building a development environment where the application could be run during development. This tool is called Vagrant[8] and the idea behind this tool is that it helps a developer to create a virtual machine, that runs the operating system with all the needed additional software installed. It is easy to package such a virtual machine and to deliver it to other developers so they can simply download it, and run it in their own machine. This way there is no need for every developer to build their own development environment, but just use already existing virtual machine with all the needed packaged installed. Just run it and only concentrate on development [10]. Application can run on any platform that supports PHP, so it can run on Linux, Windows and Mac OS X. However, I have only really tested it on Linux Ubuntu 12.04 operating system with Apache 2 web server, PHP 5.3/5.4 and MySQL 5 database. In reality there should be no difference what database

[4]`http://twitter.github.io/bootstrap/`
[5]`http://jasny.github.io/bootstrap/`
[6]`http://www.jetbrains.com/phpstorm/`
[7]`http://git-scm.com/`
[8]`http://www.vagrantup.com/`

to use, as the Doctrine object relational mapper (ORM) supports many different SQL-servers in addition to MySQL. It is only a matter of changing the configuration of the application and install needed SQL-server to start using it.

4.2 Architecture

In this section I will introduce the architecture of the application.

4.2.1 MVC

As with many web-frameworks that are used today, Symfony also uses the model-view-controller (MVC) design pattern. The main idea behind MVC is the separation of the domain objects that model the particular domain and the presentation objects that are the GUI elements shown to end-user. Domain objects should be self contained and work without the reference to the presentation. In MVC these domain objects are called models. Models have no knowledge of the user interface. The presentation part in MVC contains two parts: view and controller. The controller acts as the middleman between view and model. Its task is to take user input from the view and decide what to do with it. View is simply the interface that end-user sees and interacts with [27]. The general model of the MVC-pattern can be see on figure 4.1.

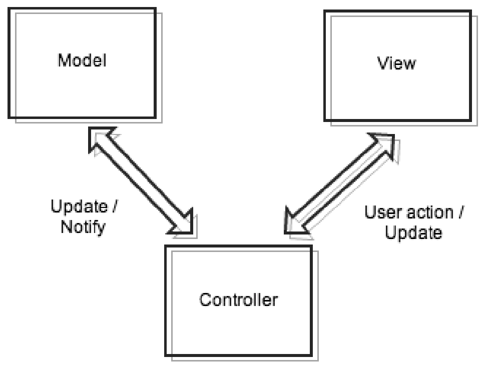


Figure 4.1: MVC design pattern

Here controller accesses the data from the model and updates view according to it. User can see the data presented in the view and interact with the application via view. Controller handles user interaction coming from the view and updates model accordingly.

Even though the author of the Symfony web-framework calls the framework a “HTTP framework” instead of MVC framework, it actually contains all the components that help developer to organize his code following the well-known Model-View-Controller (MVC) design pattern [26]. Models are called entities in the Doctrine object-relation-mapper (ORM) implementation and are responsible for data persistence. Controllers of the application are extending the Controller class of the Symfony Framework bundle and views are handled by the Twig[9] templating-engine that comes with the base Symfony package.

[9]`http://twig.sensiolabs.org/`

4.2.2 Bundles

Symfony2 web-framework organizes the code into bundles. Bundle is simply a directory that contains everything related to a specific feature, including PHP classes, configuration, also styleshseets, Javascript files and html templates. Bundle is similar to plugin in other software. The main difference is that everything in Symfony2 is a bundle, meaning that all the framework core functionality as well as the software written for an application are organized in bundles. Using bundles, makes it very easy to add new packages into the application. All that is needed, is to include new bundle location into Composer configuration, install it and make it available in the application by including new bundle name in the application configuration [11].

The main functionality of the application is located inside the ValidatorBundle. In addition there is also UserBundle, which expands a special bundle called FOSUserBundle[10] and contains all the functionality that is needed for registering, authenticating and authorizing users to access the application as well as user profile management.

ValidatorBundle contains three controllers:

- ValidationController
- ShareController
- ReportController

In following subsections I will give more detailed overview about each of these controllers.

4.2.3 ValidationController

The responsibility of the ValidationController is to control how end-user can create, read, update and delete his validation related data entities. The class diagram of the data entity classes can be seen on figure 4.2.

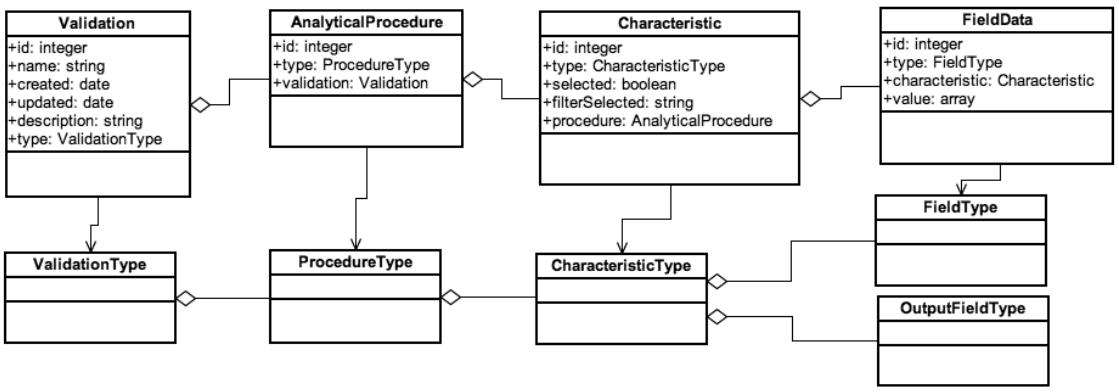


Figure 4.2: Validation class diagram

The main idea behind the data entities is that each Validation entity can contain many AnalyticalProcedure entities. AnalyticalProcedure entities can contain many Characteristic entities and Characteristic entities can contain many FieldData entities. FieldData entities are the actual data containers. These entities store the data that

[10]`https://github.com/FriendsOfSymfony/FOSUserBundle`

is entered by the end-user. All the calculations and report generation is done based on the data stored in FieldData entities. When end-user creates and configures a new validation process, a new Validation entity together with AnalyticalProcedure and Characteristic entities are created and stored in the database. FieldData entities are created and stored in the database only after end-user actually enters some data to any of the characteristic form.

Actual data persisting is handled by the Doctrine ORM, which has many convenience features like automatic cascade persist and remove as well as orphan removal. These features help developer to achieve consistency of the data model as it automatically persists and removes entities that are related to given instance of entity class. Also, when some of the related entities are removed, Doctrine ORM takes care of orphan entities removal without developer needing to create a lot of boilerplate code to handle such cases [13].

Each validation entity contains the configuration data that user specifies during the creation of the validation entity. In this configuration data, user can choose which analytical procedures he wants to work with, which characteristics should be present in each of the analytical procedures and finally choose specific filter values for each characteristic, which are shown to end-user as a decision tree, where he can choose only one certain branch that determines which fields will be present in the form that will be later generated during data entry.

4.2.4 ShareController

The responsibility of the ShareController is to control how end-user can manage sharing his validations with other end-users. The idea on how the entities are organized to achieved the sharing of a validation between users is shown on the figure 4.3.

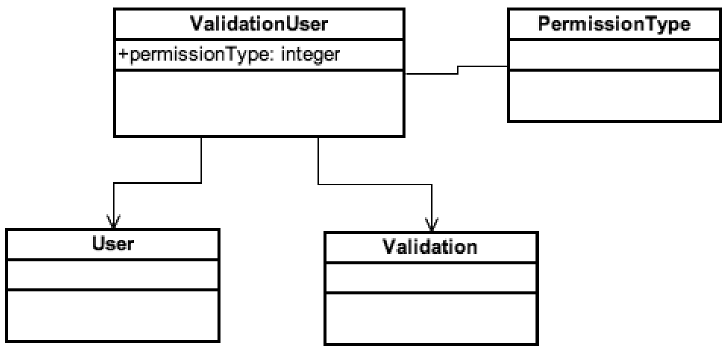


Figure 4.3: Validation Sharing Class Diagram

ShareController interacts with the system users that are managed by the UserBundle. There can be many ValidationUser class instances that each know about the User and Validation entities that should be bound together. In addition, there is a PermissionType entities, that determine what an user is allowed to do with the Validation. There are three permission types available:

1. PERMISSION_TYPE_ALL - this is only assigned to the user who created a particular validation;

2. PERMISSION_TYPE_READ - this allows user to only read the validation and not make any changes to it;
3. PERMISSION_TYPE_UPDATE - this allows user to both read and update a particular validation entity.

4.2.5 ReportController

The responsibility of the ReportController is to control how end-user can configure reports to some specific validation and also handles the coordination of how any of the reports gets rendered based on the report configuration and data entered by end-user during validation process. Each user can create many reports for any given validation. The overall class diagram for report generation was shown on figure 3.3 and described in more detail in section 3.2.

4.3 Form Generator

As I described in section 3.1 there was a need for a dynamic form generator in this software tool. My implementation is based on the form component[11] available in Symfony web-framework.

4.3.1 Form generation

In [29] the author of the form component describes following key aspects that was used while creating this component:

- Abstraction - ability to take any part of the form and put it into a reusable data structure;
- Extensibility - contains two main concepts: specialization, which means that it is possible to extend any given data structure and mixins, which allow attaching functionality to existing object without specializing them;
- Compositionality - by using Composite pattern [30], it makes it possible to nest data structures into themselves, as there is no difference between fields and forms in Form component, this makes it possible to build as complex forms as needed;
- Separation of Concerns - data transformation, HTML generation (the view), validation and data mapping are all decoupled components;
- Model Binding - the form component reuses existing metadata from the model and also reads default values from the domain object and writes values back to object;
- Dynamic Behavior - this allows adding and removing field to form dynamically, allowing to create repeated form elements

[11]`https://github.com/symfony/Form`

By default this form component is used to map existing data entity fields to a certain form elements and all this is done in the code. As this would have meant that all the characteristics have to be described in the program itself, making the expanding of the software extremely difficult. I came up with the idea, where I described the forms and fields present on them in external configuration files and built a special configuration parser. This parser creates a hierarchy of configuration objects and injects this hierarchy into special form builder class instance. Form builder dynamically creates a form for a certain characteristic based on the meta-data that comes from configuration and also the configuration data that user made previously while creating and configuring the actual validation record. This includes the filters specification which allows the form builder to create a form that only has the fields available that have certain filter assigned to them.

4.3.2 View rendering

Forms are rendered by the Twig templating engine. Together with the form component, the actual form rendering is made really simple. There is no need to specify the actual fields in the form but instead the compositionality aspect takes care of the automatic form rendering. As all the forms and form elements are nested in other forms and form elements, it is sufficient to only pass the top-level form element to the view and view can recursively render all the parts. So the actual code needed to render the whole form in the view is just one line as following:

```
{{ form_widget(form) }}
```

There is one important idea in the Twig form rendering, and that is the idea that it is possible to override every aspect how any of the form element types are visually rendered. Each form element is rendered by some widget. It can be number_widget, file_widget, text_widget, etc. By specifying how exactly we want to show any particular form element, we are describing that in the template file and next the rendering engine will use that info. If we are satisfied with the default looks of the elements, we really don't have to do anything, but in case we want to apply some styling or additional formatting to any of the elements, we are free to do that by modifying these widgets. E.g. if we want to apply certain CSS-class to form fields that will show numeric values, we can do following in our Twig-template:

```
{% block number_widget %}
    {{ form_widget(form, { 'attr': {'class': 'input-small'} }) }}
{% endblock %}
```

Here we redefine the block number_widget and assign class-attribute with value "input-small". Similarly we can redefine every aspect of how form elements should be rendered without the need to actually explicitly assign any of these customization parameters to specific form elements, but only apply them to the generic form field types instead.

4.3.3 Data Validation

In addition to form generation and rendering in views that end-user can see and interact with, there is also a need to actually validate the data that user enters to make sure that we are going to save only correct values. This is achieved by applying the validation constraints to form fields. These validation constrains are described in characteristic configuration files for each of the input fields as described in section 3.1.4. After these validation constrains are specified while forms are generated, they will be applied automatically after user submits the form. In case of any of the validations fail, data is not persisted into database and user will be notified about what fields need to be corrected.

4.3.4 Data Persisting

Final step in form data handling is the actual data persistence. This is achieved by Doctrine ORM component. Again, I did not want to create any characteristic-specific data entities, that would make the software very inflexible, instead there is a general FieldData entity that stores the actual data as serialized representation of the actual value, which can be any kind of data, like array, textual values, numeric values, path to certain uploaded file, etc.

4.4 Report Generator

After end-user has entered data into characteristic forms, it is important that the entered data and data calculated and generated based on the entered data could be shown in a specific report. For that purpose a report generator was implemented. After end-user has created a new report instance for a specific validation entity, he will be presented with the possibility to choose which sections will be present in the final report. These sections are recursive as was described in section 3.2 and allow end-user to choose which exact parts he want to show in final report. General class diagram of how report generation works was shown in 3.3. Reports are built based on the pre-defined templates. Each of the templates contains sections which can contain sub-sections recursively. Report generator parses the template configuration file and creates instances of particular renderer classes. These instances will use the given input data whether this is a data field value, or some calculated value. After the renderer has produced the output this output will be returned to report generator and finally passed to Twig-template engine which will produce the final report.

4.5 Unit Testing

There is a widely used unit testing framework available for PHP language, called PHPUnit[12]. PHPUnit belongs to a XUnit family together with the JUnit for Java, CppUnit for C++, NUnit for .NET and many other. The XUnit family started with the with the SUnit for Smaltalk. In [31] the idea of the unit testing framework was introduced. The philosophy of the unit testing is to create a special unit test class for each one of the classes you want to test. Framework gives developer convenient way of writing such test-classes and run these tests regularly. This helps developer to cover the important parts of the application with unit tests and thus make the changing of the application much more secure. If some change has made to a class that is covered by unit tests, developer can run the pre-defined test cases and see if there was a regression or not.

[12]http://www.phpunit.de/

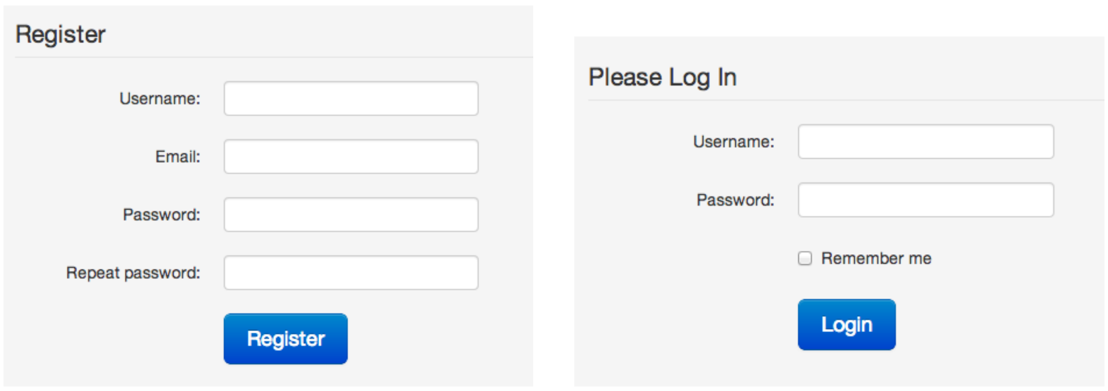
In current software tool, I covered all the calculator classes with unit tests, that perform calculations on user-entered data and produce a result of some statistical computation.

4.6 User Interface

In this section I will introduce the user interface of the software tool for analytical procedure validation. User interface is built using Twitter Bootstrap library that contains many ready-made components to conveniently and quickly produce nice-looking and functional user interfaces.

4.6.1 User registration and login

Before user can start using the application, he has to register as the user of the application. After registration user will be able to log in. Both registration and login views are shown on figure 4.4.

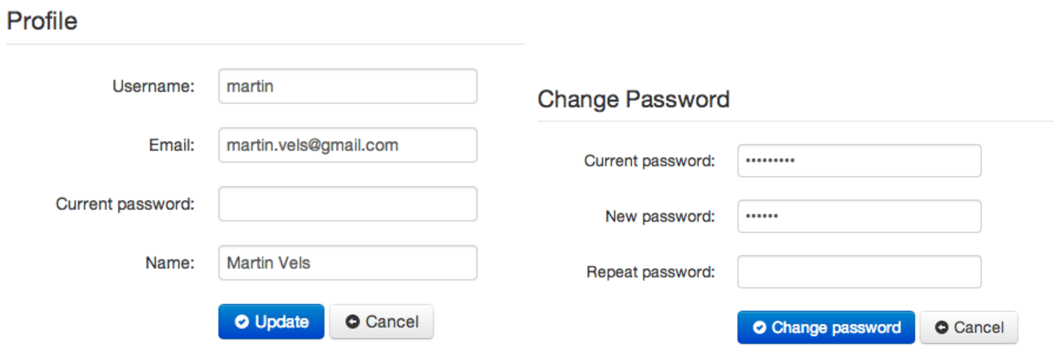


(a) User registration (b) Login

Figure 4.4: User registration and login views

4.6.2 Profile modification

After user successfully logs in, he can manage his profile. He can specify the full name and also change password of his account. Sample views can be seen in figure 4.5.


Profile

Username: martin

Email: martin.vels@gmail.com

Current password:

Name: Martin Vels

Change Password

Current password: •••••••••

New password: ••••••

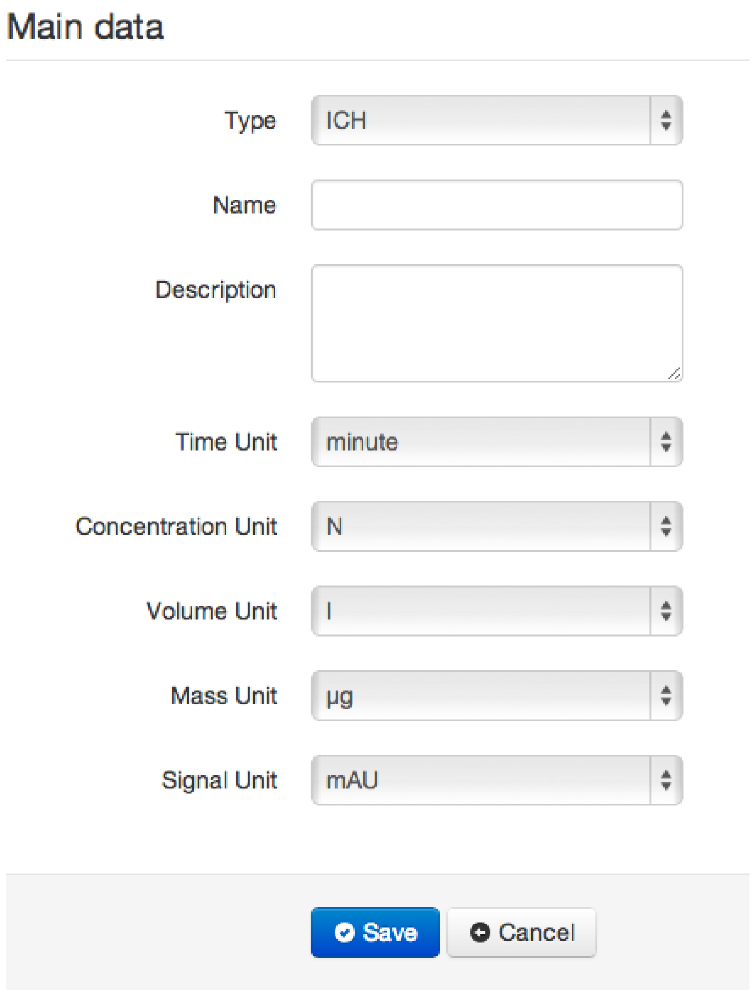
Repeat password:

(a) User profile update (b) Password change

Figure 4.5: User profile and password change views

4.6.3 Validation creation

First step in starting an actual analytical procedure validation is to create new validation entity as shown in figure 4.6.


Main data

Type: ICH

Name:

Description:

Time Unit: minute

Concentration Unit: N

Volume Unit: l

Mass Unit: μg

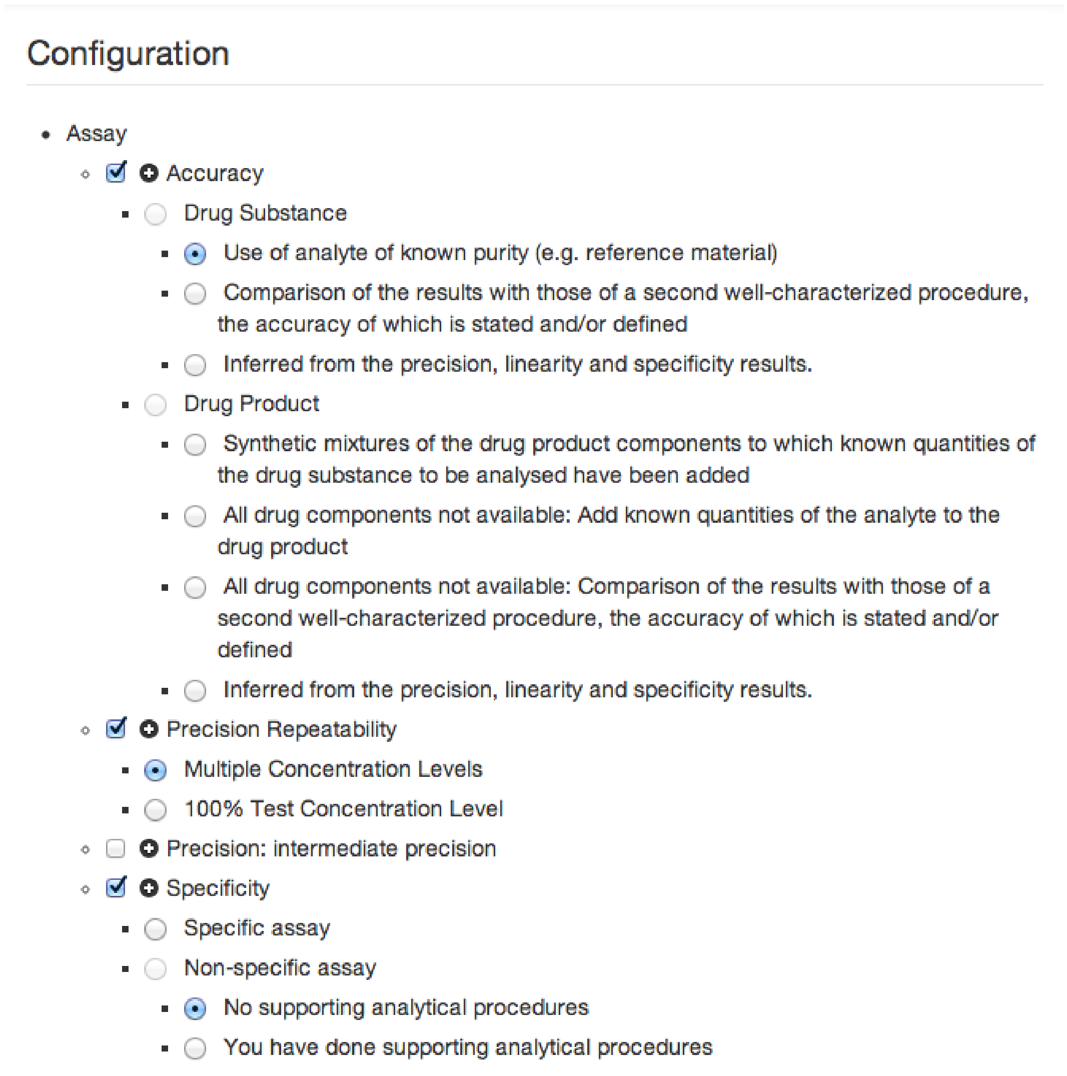
Signal Unit: mAU

Figure 4.6: Validation entity creation

There are several important parameters here that user needs to choose. First, the type of the validation, which indicates what validation guideline should be followed for current validation. Current prototype only has one type "ICH" available, but additional validation guidelines can be added later.

Next, the "name" and "description" parameters are there so user could identify his validations from the list of many validations. And finally there are several measurement unit selections that user can choose from. Each one of these selections represents one specific unit type. E.g. user can choose what should be the time units for all the fields storing time-related data throughout the current validation. These selected unit types will be shown next to fields during the data entry in characteristic forms and guide user so he knows which unit is expected at any particular field.

After validation entity is created, user will be redirected to configuration view, where he can start specifying which exact analytical procedures he wants to validate and which characteristics are needed there. A sample view of such configuration tree fragment is shown in figure 4.7.



Configuration

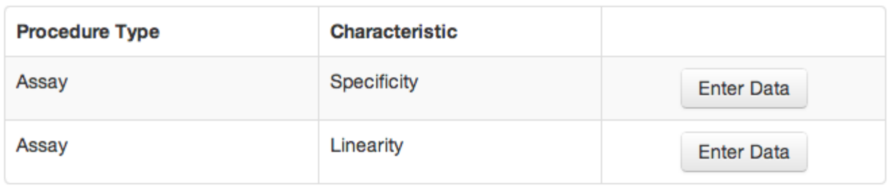
- Assay
  - [✓ Accuracy]
    - Drug Substance
      - [✓ Use of analyte of known purity (e.g. reference material)]
      - Comparison of the results with those of a second well-characterized procedure, the accuracy of which is stated and/or defined
      - Inferred from the precision, linearity and specificity results.
    - Drug Product
      - Synthetic mixtures of the drug product components to which known quantities of the drug substance to be analysed have been added
      - All drug components not available: Add known quantities of the analyte to the drug product
      - All drug components not available: Comparison of the results with those of a second well-characterized procedure, the accuracy of which is stated and/or defined
      - Inferred from the precision, linearity and specificity results.
  - [✓ Precision Repeatability]
    - [✓ Multiple Concentration Levels]
    - 100% Test Concentration Level
  - Precision: intermediate precision
  - [✓ Specificity]
    - Specific assay
    - Non-specific assay
      - [✓ No supporting analytical procedures]
      - You have done supporting analytical procedures

Figure 4.7: Validation configuration view

The configuration view is shown as one decision tree, where on top level there are validation procedures ("Assay"), next there are characteristics ("Accuracy", "Precision Repeatability", "Specificity", etc.) and for each characteristic there are configuration choices, which are actually filters that will help to show only subset of all possible fields in any particular characteristic data entry form. User can choose many characteristics using check-boxes, but only choose one of the branches for every characteristic. For this purpose radio-buttons are used. E.g. in current example shown in figure 4.7 user can only choose one of the values under "Precision Repeatability", it can be "Multiple Concentration Level" or "100% Test Concentration Level".

4.6.4 Characteristic data entry

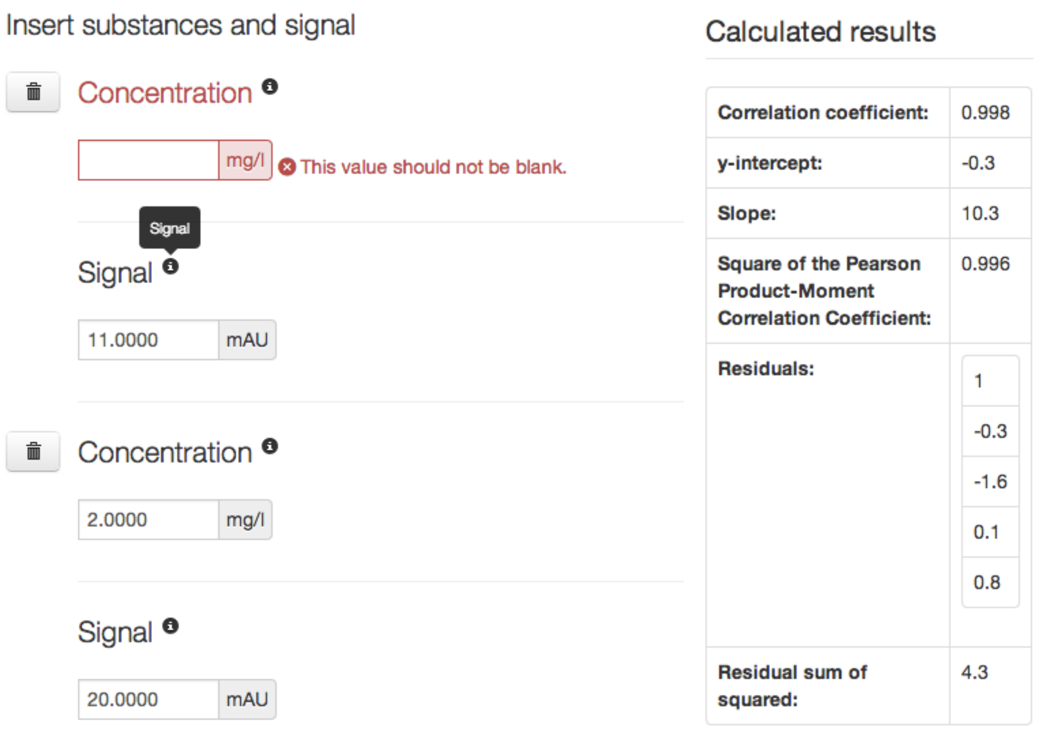
After user has configured the characteristics that he wants to enter data for, he can head to actual data entry part of the application. Based on the previously made choices he will be shown only relevant characteristics as shown on figure 4.8.



Procedure Type	Characteristic	
Assay	Specificity	Enter Data
Assay	Linearity	Enter Data

Figure 4.8: Characteristics available for data entry

After clicking on one of the "Enter Data" buttons, user will be headed to actual data entry form that is dynamically generated based on the configuration data as shown on figure 4.9.



Insert substances and signal

Concentration

mg/l
This value should not be blank.

Signal

11.0000 mAU

Concentration

2.0000 mg/l

Signal

20.0000 mAU

Calculated results

Correlation coefficient:	0.998
y-intercept:	-0.3
Slope:	10.3
Square of the Pearson Product-Moment Correlation Coefficient:	0.996
Residuals:	1 -0.3 -1.6 0.1 0.8
Residual sum of squared:	4.3

Figure 4.9: Data entry form for linearity

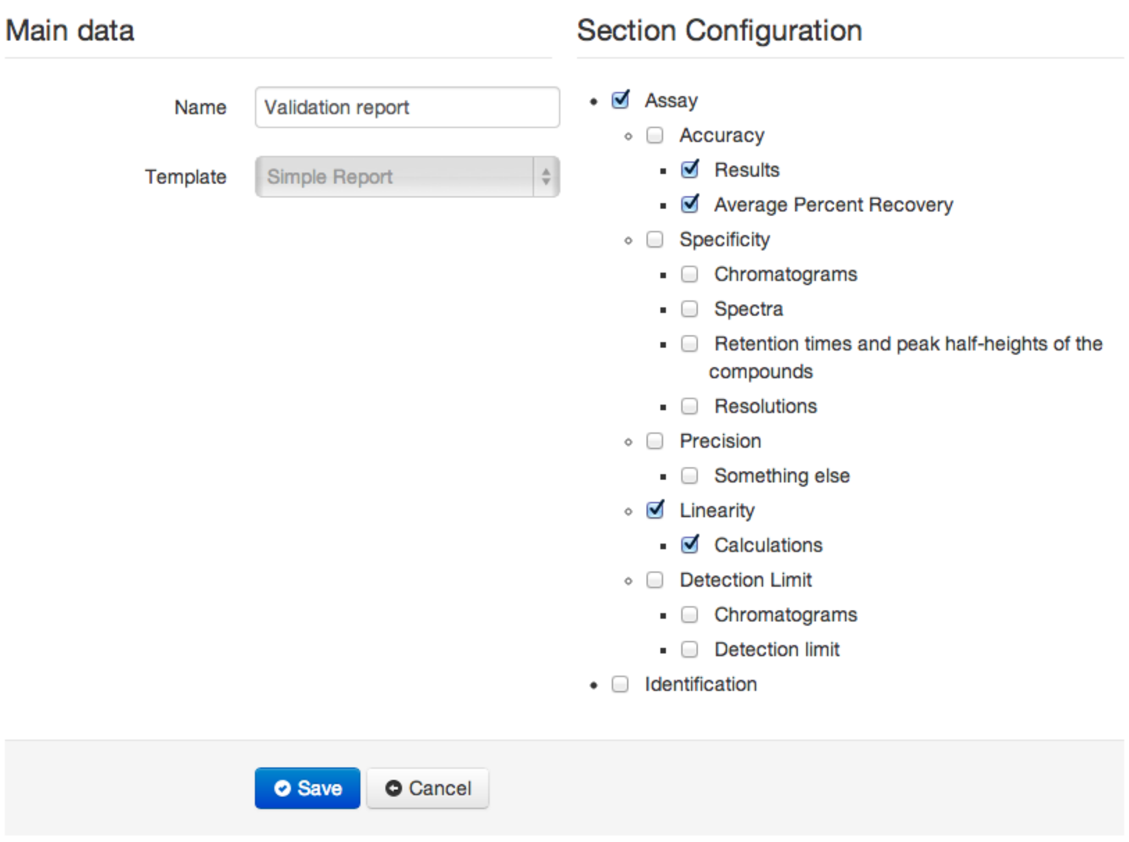
There are several important ideas present in this figure, first there is the concept of multi-field present in the form of two fields "Concentration" and "Signal" grouped together. Next, the repeated field idea is present, which can be seen from the fact that more than one block of same fields are present. The trashcan button next to field is for user to remove that particular field if needed. Also, it can be seen that units are shown next to field, these are the actual units that user selected during validation entity configuration. Small icons with the "i" on them are used to indicate that there is a tooltip present for that field, which can be seen after user moves his mouse cursor over the icon.

The concept of data field constraints in action can be seen from the first "Concentration" field, which is shown with red color and special error-message next to field, which indicates that given field should not be left blank.

And finally there is the concept of output fields present in figure 4.9. As described in section 3.1.5, these are special fields that use other fields, both input and output, to perform some calculations and return a value or array of result values. In current example it can be seen, that many statistical calculations were made on the concentration and signal values, like correlation coefficient, y-intercept and slope of the regression line, and many other. All these calculated results are important for the end-user to see right next to data entry form, as he will get the idea on how well his analytical procedure is suited for a particular need from the validation point of view.

4.6.5 Report creation

First step in creating a report for a validation is to add new report entity. As shown on 4.10 user needs to specify a name for a report together with the template. After creating a report entity, user will be presented with the available sections in that particular template. User can choose as many sections as needed. Sections are organized hierarchically in tree-like structure, which allows user to see which sub-sections belong to which parent section.



Main data

Name: Validation report

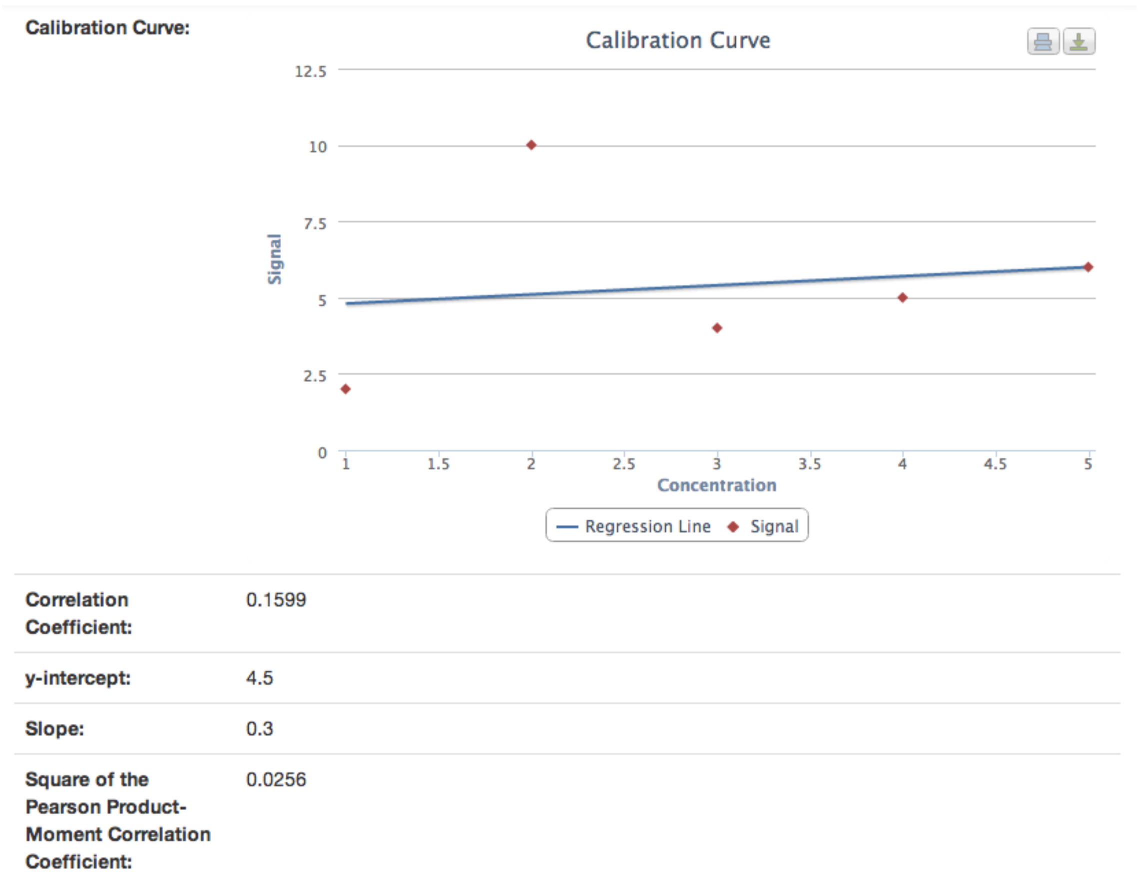
Template: Simple Report

Section Configuration

- [✓] Assay
  - [ ] Accuracy
    - [✓] Results
    - [✓] Average Percent Recovery
  - [ ] Specificity
    - [ ] Chromatograms
    - [ ] Spectra
    - [ ] Retention times and peak half-heights of the compounds
    - [ ] Resolutions
  - [ ] Precision
    - [ ] Something else
  - [✓] Linearity
    - [✓] Calculations
  - [ ] Detection Limit
    - [ ] Chromatograms
    - [ ] Detection limit
- [ ] Identification

Figure 4.10: Validation report creation

After the report entity is created and output sections are configured, an actual report can be generated. A fragment from a sample report could look as shown on 4.11.



Calibration Curve:	
Correlation Coefficient:	0.1599
y-intercept:	4.5
Slope:	0.3
Square of the Pearson Product-Moment Correlation Coefficient:	0.0256

Figure 4.11: A fragment of a sample report

There can be any kind of data present in a report, it may contain simple values, calculated values based on the data entered, tables, images, diagrams created based on the data entered, etc.

E.g. calibration curve diagram is shown on 4.11. For generating charts dynamically I am using the Highcharts JS library[13].

4.7 Roundup

In this chapter I gave an overview of the implementation of the software tool for analytical procedure validation. First an architecture of the software was overviewed, followed by the more detailed description of some of the important components like form generator and report builder. Final section of the chapter introduces the user interface of the software tool via sample screen shots.

[13]`http://www.highcharts.com/`

Chapter 5

Conclusion and Future Work

The aim of this thesis was to implement a working prototype of a software tool for analytical procedure validation. This prototype was built based on one certain validation guideline [1]. All the possible characteristic types that were present in mentioned validation guideline were described in special configuration files. It is now possible to add new configurations for similar validation guidelines, based on the given solution, to extend the application functionality. In addition to validation guideline description, also the needed calculated fields were described in configurations together with the calculator classes that are capable of producing the results based on the entered data.

In addition to describing the validation guideline related characteristics, also simple report template was described together with the classes capable of rendering these reports. Reports can contain the values entered by users, as well as computed results and charts generated based on the entered data. Different renderers allow showing data in various forms, like simple numbers and text, as tables, images with descriptions or diagrams.

The software tool built, contains two major parts: dynamic form builder and dynamic report builder. Both of these parts are built in modular fashion, meaning that they can be easily extended to add new functionality.

Current software tool, that was built, describes all the needed configuration data for specific validation guidelines in special JSON-format configuration files. It is not very easy for domain experts to describe such configurations themselves without learning JSON-format. This means that one of the most important feature needed in the future is a special configuration management tool, that would help domain experts to describe the validation guidelines themselves without the help of a developer. We foresee that the configuration tool could take the form of a simple visual forms builder allowing domain experts to create and drag-and-drop fields and groups of fields and to add filters, constraints and other properties as defined in the forms specification language. The report generation could similarly take the form of a visual tool allowing the domain experts to drag-and-drop fields defined in the forms corresponding to a given validation procedure. This configuration tool could use current configuration files as data storage, or save the configuration into database.

In addition to configuration management tool, there is a need to create libraries that would help importing data from the files exported from the software controlling the actual HPLC equipment. At the moment all the data is entered manually by the end-users, after such libraries are created, it would be possible to simply import data from data files.

It would be important to add context-sensitive help-system to the program, so that end-user would be able to get as much support from the program as possible. The idea would be generating an index of terminology used in procedure validation and make it possible for user to access that index from the program.

Finally, a conversion tool would be needed, that would help users to convert a validation of one specific guideline into validation of some other validation guideline. Such a conversion tool would help an analyst to validate his analytical procedure against many different validation guidelines without the need to enter the same data more than once. As there are overlaps of the data needed in many of the validation guidelines, it would be possible to create some conversion rules from one guideline to other.

Tarkvara analüütiliste HPLC protseduuride valideerimiseks

Paljude tööstus- ja teadusasutuste igapäevatöö hõlmab regulaarset keemiliste analüüside läbiviimist. Vajadus analüüside järele on erinev, see võib olla kas vajadus määrata sportlaste uriinis või veres keelatud ainete sisaldust või hoopis vajadus mõõta toimeainete sisaldust ravimis. Selleks, et sarnaseid analüüse usaldusväärselt läbi viia ning saavutada usaldusväärsed tulemused, tuleb jälgida kindlat reeglistikku, mis määrab, et kindlad tingimused on täidetud ja kindlad sammud analüüsi protsessis on läbitud. Sellist reeglistikku nimetatakse analüütiliseks protseduuriks ehk analüütiliseks meetodiks. Selleks, et analüütiline meetod oleks nii usaldusväärne ja laialdaselt kasutatav kui võimalik, tuleb see valideerida. Selleks, et mingit analüütilist meetodit valideerida, tuleb järgida valideerimise juhendit, et kindlaks teha, kas konkreetne metoodika on tegelikult sobiv selleks, mille jaoks ta kavandati.

Paljudel juhtudel saab keemilisi analüüse sooritada kromatograafia abil, täpsemalt kasutades kõrgsurve vedelikkromatograafi. Kujutades ette keemialaborit kalli ja tehnika viimase sõna järgi sisustatud seadmetega, siis on loomulik, et neid seadmeid juhitakse ja nendega suheldakse arvuti abil. Hoolimata selles, et seadmetega suhtlemine ja nende kontrollimine käib arvutite abil, tuleb analüütilise meetodi valideerimist endiselt sooritada käsitsi. Esmalt tuleb kasutajal ennast kurssi viia konkreetsete valideerimisjuhenditega, mis võivad olla ühtaegu mahukad kui ka kaunis üldised. Järgmisena tuleb valideerimise käigus mõõta ja üles märkida kindlad karakteristikute väärtused. Edasi tuleb nende talletatud mõõtmistulemuste väärtustega sooritada erinevaid arvutusi ja saadud tulemuste põhjal otsustada, kas kindlad kriteeriumid on täidetud või mitte. Viimaks tuleb saadud tulemuste põhjal luua detailne valideerimisraport, mida oleks võimalik analüütilise protseduuri juhendile kaasa panna, et näidata vastava metoodika sobivust ettenähtud eesmärgi täitmiseks.

Kõike seda vaevarikast tööd peab kasutaja sooritama käsitsi, kulutades suure hulga oma tööajast rutiinsele valideerimistegevusele, samas kui selle asemel võiks parendada analüüsi metoodikat ennast.

Loomuliku lahendusena eelkirjeldatud probleemile oli luua tarkvara, mis aitaks keemikuid vähendada sellise käsitsi tehtava töö hulka. Käesoleva magistritöö eesmärgiks oligi luua tarkvara, mille ülesandeks on aidata kasutajal valideerida kindlat analüütilist protseduuri, juhendades teda, milliseid konkreetseid karakteristikuid tuleb mingil juhul sisestada, arvutab vajalikud väärtused etteantud karakteristikute põhjal ning genereerib lõpuks raporti, mis sisaldab nii sisestatud andmeid kui arvutuste tulemusi.

Kuigi loodud tarkvara loomisel lähtuti ühest konkreetsed valideerimise juhendist, täpsemalt ICH Harmonized Tripartite Guideline-st, on tarkvara ehitatud viisil, mis võimaldab lisada lihtsalt uusi juhendeid ja laiendada vajadusel tarkvara funktsionaalsust.

Bibliography

[1] Text on Validation of Analytical Procedures. ICH Harmonised Tripartite Guideline. October 1994.

[2] Documentation for Symfony2 framework. Published online http://symfony.com/doc/current/index.html. Last visited April, 2013.

[3] VALIDAT - The Leading Solution for Method Validation. Published online https://www.icd.eu/produkte/methodenvalidierung.html. Last visited April, 2013.

[4] Kromatograafia loenguslaidid. Published online http://tera.chem.ut.ee/~ivo/Chrom/chrom.pdf. Last visited April, 2013.

[5] Lloyd R. Snyder, Joseph. J. Kirkland, Joseph L. Glajch. Practical HPLC Method Development. 1997.

[6] Ivo Leito. Analüüsimetoodikate valideerimine. Lecture slides. February 4, 2013.

[7] Documentation for Twitter Bootstrap framework. Published online http://twitter.github.io/bootstrap/getting-started.html. Last visited April, 2013.

[8] Documentations for Jasny Twitter Bootstrap extension package. Published online http://jasny.github.io/bootstrap/. Last visited April, 2013.

[9] Documentation for PHPStats Statistical Library. Published online http://mcordingley.github.io/PHPStats/index.html. Last visited April, 2013.

[10] Documentation for Vagrant software. Published online http://docs.vagrantup.com/v2/getting-started/index.html. Last visited April, 2013.

[11] The Bundle System. Symfony documentation. Published online http://symfony.com/doc/current/book/page_creation.html#page-creation-bundles. Last visited April, 2013.

[12] Introduction to JSON. Published online http://www.json.org/. Last visited April, 2013.

[13] Working with Associations. Doctrine ORM documentation. Published online http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/working-with-associations.html. Last visited April, 2013.

[14] XForms W3C recommendation. Published online http://www.w3.org/TR/xforms/ Last visited May, 2013.

[15] Kurt Cagle. Why XForms Matter, Revisited. Published online http://www.oreillynet.com/xml/blog/2006/03/why_xforms_matter_revisited.html. Last visited May, 2013.

[16] Kurt Cagle. Understanding XForms: The Model. Published online http://www.oreillynet.com/xml/blog/2006/03/understanding_xforms_the_model.html Last visited May, 2013.

[17] Kurt Cagle. Understanding XForms: Components. Published online http://www.oreillynet.com/xml/blog/2006/06/understanding_xforms_component.html Last visited May, 2013.

[18] XForms W3C Recommendation. 6.1.3. The required Property. Published online http://www.w3.org/TR/xforms/#model-prop-required Last visited May, 2013.

[19] XForms W3C Recommendation. 8.1.5. The output Element. Published online http://www.w3.org/TR/xforms/#ui-output Last visited May, 2013.

[20] XForms W3C Recommendation. 9.3. The XForms Repeat Module. Published online http://www.w3.org/TR/xforms/#ui-repeat-module Last visited May, 2013.

[21] Philipp Wagner. The Future of Mozilla XForms. Published online http://www.philipp-wagner.com/blog/2011/07/the-future-of-mozilla-xforms/ Last visited May, 2013.

[22] XForms in Mozilla developer network. Published online https://developer.mozilla.org/en-US/docs/XForms Last visited May, 2013.

[23] Rein Raudjärv. Dynamic Schema-Based Web Forms Generation in Java. Master Thesis, University of Tartu, 2010.

[24] Marcello La Rosa, Wil M.P. van der Aalst, Marlon Dumas, Arthur H.M. ter Hofstede. Questionnaire-based Variability Modeling for System Configuration. 2008.

[25] Orbeon Wiki. Repeated Content. Published online http://wiki.orbeon.com/forms/projects/form-runner-builder/repeated-content Last visited May, 2013.

[26] Fabien Potencier. What is Symfony2? Published online http://fabien.potencier.org/article/49/what-is-symfony2 Last visited May, 2013.

[27] Martin Fowler. GUI Architectures. Model View Controller. Published online http://martinfowler.com/eaaDev/uiArchs.html#ModelViewController Last visited May, 2013.

[28] Abhijit Menon-Sen. Using Git to manage a web site. Published online http://toroid.org/ams/git-website-howto Last visited May, 2013.

[29] Bernhard Schussek. Symfony2 Form Architecture. Published online http://webmozarts.com/2012/03/06/symfony2-form-architecture/ Last visited May, 2013.

[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. 1994.

[31] Kent Beck. Simple Smalltalk Testing: With Patterns. Published online http://www.xprogramming.com/testfram.htm Last visited May, 2013.

[32] Requirements for running Symfony2. Published online http://symfony.com/doc/current/reference/requirements.html Last visited May, 2013.

[33] Installing and Configuring Symfony. Published online http://symfony.com/doc/current/book/installation.html Last visited May, 2013.

[34] Databases and Doctrine. Symfony documentation. Published online http://symfony.com/doc/current/book/doctrine.html Last visited May, 2013.

[35] Daniel F. Gieskens, James D. Foley: Controlling User Interface Objects Through Pre- and Postconditions. In Proceedings of the International Conference on Human Factors in Computing Systems (CHI), Monterey, CA, USA, May 1992, pp. 189-194

Appendix A

Resources

The source code of the software tool for analytical procedure validation is available on the CD attached to this thesis.

Appendix B

Required Software

As the software tool is built on top of Symfony2 web-framework, it relies on the system requirements of Symfony2 [32]:

Required

- PHP needs to be a minimum version of PHP 5.3.3
- JSON needs to be enabled
- ctype needs to be enabled
- Your PHP.ini needs to have the date.timezone setting

Optional

- You need to have the PHP-XML module installed
- You need to have at least version 2.6.21 of libxml
- PHP tokenizer needs to be enabled
- mbstring functions need to be enabled
- iconv needs to be enabled
- POSIX needs to be enabled (only on *nix)
- Intl needs to be installed with ICU 4+
- APC 3.0.17+ (or another opcode cache needs to be installed)
- PHP.ini recommended settings
  - short_open_tag = Off
  - magic_quotes_gpc = Off
  - register_globals = Off
  - session.auto_start = Off

Doctrine

If you want to use Doctrine, you will need to have PDO installed. Additionally, you need to have the PDO driver installed for the database server you want to use.

Installation

Actual installation of the application together with the Symfony2 framework and other used bundles is simple when there is a Composer dependency manager installed:

```
curl -s https://getcomposer.org/installer | php
```

After composer is available on the system the source code needs to be copied to the folder where the software will be hosted in the server and next following command needs to be run to retrieve all the needed packages:

```
php composer.phar install
```

After all the software packages are downloaded, it is needed to set correct file permissions. There are several possibilities for setting these permissions depending on the filesystem and ACL support availability on particular system. In case of ACL is supported, permissions can be set using following commands:

```
$ rm -rf app/cache/*
$ rm -rf app/logs/*

$ sudo chmod +a "www-data allow delete,write,append,file_inherit,
   directory_inherit" app/cache app/logs
$ sudo chmod +a "‘whoami‘ allow delete,write,append,file_inherit,
   directory_inherit" app/cache app/logs
```

Other possibilities are listed in [33].

After the permissions are correctly set, there is only one more thing to do, it is configuring database and creating the schema. This is done by editing the app/config/parameters.yml file, where correct database accessing requisites have to be specified [34]:

```
# app/config/parameters.yml
parameters:
    database_driver:    pdo_mysql
    database_host:      localhost
    database_name:      test_project
    database_user:      root
    database_password:  password
```

Creating the database is done using following command:

```
$ php app/console doctrine:database:create
```

And finally to create the schema, following command has to be run:

```
$ php app/console doctrine:schema:update --force
```

After that application should be ready and can be accessed from the address where the application was installed in a particular server.

Non-exclusive license to reproduce thesis and make thesis public