TECHNISCHE
UNIVERSITÄT
DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

Concepts of Quality Assurance for the Constraint-based E-Assessment of Models

Paul Erlenwein

paul.erlenwein@tu-dresden.de
Born on: 31st December 1996 in Ludwigshafen
Matriculation number: 4609464
Matriculation year: 2016

Bachelor Thesis

to achieve the academic degree

Bachelor of Science (B.Sc.)

Supervisor
Dr.-Ing. Birgit Demuth
Supervising professor
Prof. Dr. rer. nat habil. Uwe Aßmann

Submitted on: 5th February 2021

Contents

1. Introduction 4
   - 1.1. Motivation . . . 4
   - 1.2. Research Questions . . . 5

2. Related Work 6
   - 2.1. INLOOM . . . 6
   - 2.2. Quality-Testing of existing Model Evaluation Systems . . . 7
     - 2.2.1. Grade Quotient Strategy . . . 9
     - 2.2.2. Grace Points and Clean Scores . . . 9
     - 2.2.3. Inter-Rater-Reliability . . . 10
     - 2.2.4. Informedness, Markedness . . . 10

3. QIT: Quality Inspection Tool 11
   - 3.1. Software Requirements . . . 11
   - 3.2. Available Data . . . 11
     - 3.2.1. INLOOM Result Files . . . 12
     - 3.2.2. Manually Evaluated Student Solutions . . . 14
     - 3.2.3. Data Driven Data Structure . . . 15
     - 3.2.4. Digitizing Manual Evaluations . . . 16
   - 3.3. KPI for Testing . . . 17
     - 3.3.1. Comparison Detail Levels . . . 18
     - 3.3.2. Categories of Comparison . . . 19
     - 3.3.3. Visualizing Validation Results . . . 20
   - 3.4. Summary . . . 21

4. Implementation 22
   - 4.1. Implementation as a Web-Application . . . 22
   - 4.2. Backend Implementation . . . 24
     - 4.2.1. Data Handling . . . 25
     - 4.2.2. XML Adapter . . . 25
     - 4.2.3. Meta-Evaluation . . . 27
   - 4.3. Frontend Implementation . . . 31
     - 4.3.1. Services . . . 31
     - 4.3.2. Page Tree . . . 31
     - 4.3.3. Evaluation-Digitization-Wizard . . . 33
     - 4.3.4. TestDataSet Details View . . . 33
     - 4.3.5. Exercise Details View . . . 35

4.3.6. Evaluator Details View 35

5. Conclusion and Future Work 36
5.1. Research Questions 36
5.2. Future Work 36
5.3. Conclusion 36

Bibliography 39

A. Weitere Latex-Dokumentation 42

1. Introduction

1.1. Motivation

Due to rising student numbers and the availability of modern interfacing technologies, the interest, in the automatic evaluation of student modelling work, has increased in recent years. Functioning evaluation tools and methods remain scarce [14] however.

2020 was jinxed. The Covid-19 pandemic changed how our life's work. It has presented the globalized world with little anticipated challenges and we can feel its influence on almost every aspect of our everyday life. To reduce the number of human contacts as much as possible, every aspect of human interaction was evaluated for its digitizeability. However: We were all forced to witness, that our digital infrastructure is most clearly not yet up to the task of enabling us to *live the remote life*.

Most of the fundamental problems were trivialities, like missing webcams or a too slow internet connection. Where such were taken care of, the harder-to-fix problems came to light [29]. Problems like inadequately educated overtaxed personnel and missing software solutions, that comply with European digital privacy regulation. At first glance, the personnel problems don't seem to matter much to software developers, it still are problems, which, I firmly believe, can and will, at least in part, be resolved by them.

I don't want to claim, that the digitization of everyday life was a complete failure though. Like me, most office workers were able to migrate to home office without much fuss. Still: Living the life of a remote student for half a year, definitely motivates me, to spend some thought on how to make e-teaching a little better.

Even though it proved hazardous sometimes, working with existing e-teaching tools made me realize, what huge potential lies within a properly digitized higher education. Such would not only help temper the effects emergencies, like the Covid pandemic, have on university life, but will also be a powerful tool in futureproofing universities [12] for the challenges of rising student numbers [33] in the years to come.

Intelligent-Tutoring-Systems like INLOOM, an ITS under active development at TU Dresden will make the increased workload manageable for university personnel. Developers are required to produce software that is as intuitive as possible, provides a decent grade of digital security, complies with privacy regulations, handles high traffic without complain and all that, while providing an unquestionably accurate and fair environment for everyone involved.

Integrating digital resources into their workflow seamlessly, will enable teaching personnel, to still be able to focus on the individual student, when the student groups they teach become way bigger, than they are today.

1.2. Research Questions

The goal of this thesis is finding or developing a concept for validating the quality of automatically generated evaluations of student created models. The result should be the functioning prototype of an application for quality testing INLOOM [14]. Quality testing is necessary, to ensure the correctness and fairness of the evaluations, INLOOM generates for student-created solution models.

For the purpose of validating INLOOMs evaluations I will aim to answer the following research questions:

RQ1 Which values can be extracted from the manual and automatic evaluations?
RQ2 What scale is qualified to provide a conclusive impression on the quality of the evaluations INLOOM generates?
RQ3 What methodologies are employed by existing ITS to validate their results?
RQ4 How can the developer/tutor/instructor best be assisted in collecting and pre-processing the evaluation data required for the quality validation?
RQ5 How can test results be presented to greatest effect?

The answer to RQ1 will determine which values are available to facilitate quality testing of the produced evaluations. The question is approached with an analysis of the available data sets. The second research question focuses on what to do with the data, once it is collected. To achieve validation of automatically generated evaluations their quality will have to be rated on a conclusive scale. RQ2 aims to identify such a scale. Answering the question will be approached by performing a literature survey concerning the quality validation methodologies of existing software solutions and automatic grading methods. Under any envisionable circumstances, it will be necessary to digitize manual evaluations created by a tutor. These are as of yet only available in an analog format. A digitization entails a high amount of overhead for the creation of test cases that should be reduced as much as possible. RQ3 aims to resolve this problem and will be tackled by an analysis of the test creation process. The analysis will determine the most workload intensive steps of the process. These steps can then be considered in the design of the software solution proposed in this thesis. Lastly, it remains to be determined how to best present or visualize the results of this *meta-evaluation* (the evaluation of the evaluations) to the developer/tutor/instructor (RQ5).This is the reason why RQ4 is listed among the research questions. Answering it will depend greatly on the answer to RQ2.

2. Related Work

2.1. INLOOM

This work aims to validate the quality of the results the INLOOM software [14] produces. For that reason, it is inevitable to take a look at what the software does and how it works. INLOOM is an acronym for *INnteractive Learning center for Object-Oriented Modelling*. The software, as the name suggests, is intended to be employed in a learning environment. It is intended to be used to evaluate student solutions to modelling tasks, the students have to work on, as part of the mandatory beginner software engineering course at TU Dresden. It is specifically designed to aid in teaching *Object-oriented analysis* and *Object-oriented design*. INLOOM was originally developed as an extension to the existing INLOOP[19] software, which allows students to solve programming tasks online and to evaluate their results using supplied JUnit test cases.

Evaluating student-created software models is a complex task. For a given modelling exercise, there may exist multiple correct solutions, which makes defining *one* optimal solution that can be used to compare student solutions against, very hard, if not impossible. Evaluating student-created models is an *ill defined problem*. Additionally, the evaluation process itself, once one or more optimal solutions to evaluate a student solution against are found, can prove challenging. This is again due to the fact that whoever creates a model enjoys a wide range of freedom in solving the modelling task. The student solution can, for example use a different naming scheme than the expert solution it is checked against, or model a property using aggregation or composition, where the expert solution does not, and still be correct.

INLOOM evaluates supplied student models by performing a number of *constraint tests*. In the first step, a *Constraint-based Test Generator* generates a number of constraint files from an expert solution. One constraint file is generated per element found in the expert solution. Each file holds constraints INLOOM later applies to the elements of a submitted solution. A constraint, in this context, means a feature requirement applied to the student model. If, for example, the expert solution to the given modelling task contains a class called "Student", the student solution is expected to contain an equivalent element. This expectation is expressed by the existence of a constraint, that checks for the presence of the class "Student" in the student solution. By extracting the constraints from an existing model automatically, the instructor, who wants to create a new task, is not required to have any deeper understanding of how the constraints work or how they are implemented. The instructors work is reduced to supplying an expert solution for the modelling task he creates.

The constraint generation is made possible by the existence of a *Master Constraint-Set*, a collection of constraint templates. The *Master Constraint Set* is compiled per diagram type, since required features inevitably vary from one model type to another.

For each constraint, INLOOM generates an output which identifies the employed constraint and specifies information on the result of the constraints application. The result stores, how many points were awarded for the feature checked by the constraint. It also identifies which element of the solution was checked and assigns a category flag to the result. Each such category flag represents a degree to which a constraint was met. All the results of the constraints, thus created for the student solution, are collected in a common output XML file. In addition to the constraint results, this XML also contains some meta data, like the identification of the student who submitted the evaluated solution, which expert solution it was compared to, and the exercise the student tried to solve with the supplied diagram. The total points achieved, as well as the maximum points one can achieve, are also stored in the file.

2.2. Quality-Testing of existing Model Evaluation Systems

In this section, the results of a literature research, into concepts for validating the quality of evaluations, automatic grading systems produce are presented. Starting point for the research, was a collection of such systems, referenced in [14]. Since the design of the listed systems influenced decisions made during the design of INLOOM, it is only natural to also focus on them in a pre-study, that aims to identify possibilities to validate an alike system. The collection is also rather recent and quite complete, since an accompanying research into automatic assessment systems, turned up only a single tool, that was not listed. The literature research was performed using popular catalogs of scientific literature like Google Scholar and IEEE.

Before examining the listed approaches it should be defined what *quality* means in the context of this thesis. *Quality* implies a lot of things, but actually describing what it *is*, turns out to be rather difficult. Quality is a measure between perfect and really bad. What is either good or bad however, can depend on any number of arbitrary factors. It was found, that quality in this context can be generally equated with the confidence an instructor has in the tool he employs. The tool does a *good job* if it evaluates the student-created models *correctly*. If it awards points exactly where due, the instructor can be confident in the tools results. If, however, the tool he employs, does a *bad job* and, for example, does not recognize valid inputs as such or finds errors where there are none, the instructors confidence in the tool will decrease.

The listing in [14] differentiates between types of evaluation systems. There are two classes: *System* and *Method* - as well as two classes for the systems input: *Web* and *Tool*. This differentiation will not be made here. The undertaken research showed, that it does not play a major role, in terms of the applied strategy for quality validation, to which of the classes the described system belongs. Most of the listed approaches try to create and therefore result in an evaluation. The data available on the created evaluation does not differ much between classes. The data structure of the result, not how it was created, is critical to whether a validation strategy can be employed. The systems were examined regarding their quality assurance measures. Table 2.1 lists the analyzed systems and indicates the kind of quality validation they perform. In some cases, the undertaken quality assurance measures were described in a separate paper. In these cases, the system, along with the additional literature is consolidated in one table entry.

Table 2.1. Quality validation performed in systems for student model evaluation. The column *system* lists a name of the system/method/approach where one was specified. For systems where no calling name was found, the name of the main author is used instead.

System	*Source(s)*	Quality Validation Performed
INLOOM	[14]	Grade Quotient Strategy
Bian	[5, 7]	Grade Quotient Strategy
Schramm	[24]	Didactic Evaluation
UML GRADER	[15]	Grade Quotient Strategy
Striewe	[28]	Grade Quotient Strategy
Demuth	[11]	No Evaluation
Baghaei	[2]	Didactic Evaluation
Le	[21]	Didactic Evaluation
CourseMaster	[16, 17]	No Evaluation
Artemis	[4, 20]	Grade Quotient Strategy
Beck	[3]	No Evaluation
Sousa	[27]	No Evaluation
Smith & Thomas	[26, 30, 31]	Grade Quotient Strategy
Prados	[23]	No Evaluation
Tselonis	[32]	Grade Quotient Strategy

To avoid repeating the description of a frequently employed concept for quality validation, it is opportune to combine and define those. One of the most common strategies seems to be not to validate the quality of the produced evaluations at all. At least no publication of a description of a quality validation process was found for five of fifteen of the examined systems [3, 11, 17, 23, 27]. These are collected under the term *No Evaluation*. Seven of the publications describe a validation that is based on a comparison of the grade equivalent of the respective method [4–6, 26, 28, 32]. Such strategies will be discussed in their own subsection and are combined under the umbrella term *Grade Quotient Strategy*. This kind of quality validation is also described for INLOOM in [14]. For the rest of the systems, the authors detail strategies for performing an evaluation, regarding the didactic use of their respective tools [2, 21, 24] (*Didactic Evaluation*). Such an evaluation might be interesting in the future, but is not within the scope of any of the questions this thesis is intended to answer. These approaches were not examined any closer.

It must be mentioned that there do exist quite a few more systems for evaluating student created models [1, 18, 20]. These are not listed and were not examined closer, because they do not employ a constraint-based approach or at least result in a grade. The research was thus limited for the following reason: The data available for quality testing INLOOM is limited, as will be elaborated in a later section, to the results a constraint-based system *can* produce. A constraint-based approach obviously results in data that details, which constraints were met and which were not. A superficial analysis of systems, that approach the evaluation differently, was performed. It was found that evaluation strategies that are not constraint-based, or do at least result in a grade, will approach validating the quality of their results, with a completely different focus, than it is required in the context of INLOOM. Any strategy for evaluating the quality of a fundamentally different result will ultimately not be applicable, since it is not based on data anything alike the results INLOOM produces.

2.2.1. Grade Quotient Strategy

Grade Quotient Strategy describes a strategy that is based on comparing two evaluations based on the grades they awarded. The grade depends only on the total number of points awarded. It is commonly represented by the percentage of achievable points achieved. The literature [4, 5, 14, 15, 26, 28, 32] agrees, that manual evaluations of student models are - speaking in terms of quality - the *best* evaluations known. Therefore, they are the measure of quality applied to automatic evaluations. The usual approach to evaluate the results of a given assessment method using a *Grade Quotient Strategy*, was found to be to collect as many real live student solutions, as possible and evaluate them twice. Once manually and once using the assessment tool. Two *grades* are extracted from each student solution this way. A manual and automatic *grade*. By comparing these two, a rough impression of the similarity of the two ratings is gained.

All employed calculations are very simple. A *grade* can be calculated easily from the number of $points_{auto/man}$ achieved and the maximum number of points achievable $points_{max}$. After determining a grade for both the automatic and manual evaluations $grade_{auto}$ and $grade_{man}$, a grade quotient Q can be calculated. Another way to represent the difference between the two is the absolute point difference percentage (Δ). These values can then be averaged over a number of student solutions, to gain insight into the employed methods average result *quality*.

$$grade = \frac{points * 100}{points_{max}} \qquad (2.1) \qquad\qquad Q = \frac{grade_{auto}}{grade_{man}} \qquad (2.2)$$

$$\Delta = \frac{|points_{auto} - points_{man}| * 100}{points_{max}} \qquad (2.3)$$

An especially mentionable paper whose authors employed a *Grade Quotient Strategy*, was [8]. An extensive effort to compare different evaluation methodologies is described. However, the described validation process is not specialized in validating the results of a constraint-based approach. Thus the described process' is designed to be applicable to several different evaluation approaches, and thus must consider a mutable format for both the manual and the automatic evaluation. These restrictions do not exist for validating INLOOMs results.

2.2.2. Grace Points and Clean Scores

The literature [14, 26, 30, 31] points out that the ratings of different reviewers can differ significantly. [14] describes the concept of *grace points*. Points that should not have been awarded if the instructor had followed the evaluation scheme exactly during the correction. It is the instructors prerogative to turn a blind eye, if the students solution is *close enough*. If a constraint-based evaluation tool was to make such a judgement call however, it would indicate something is broken. Since every evaluator has their own style and preferences, this can lead to a measurable deviation between two manual evaluations of the same student solution that were created by different reviewers. *Clean scores* and other concepts like *moderated human marks*[30] are designed to mitigate the negative effects the described deviations have on the applicability of the manual evaluations as a quality standard. By averaging multiple manual evaluations of the same student solution or by evaluation (peer) review, an even *better* quality standard, the automatic evaluation must measure up against, is created.

To decrease the influence the differing preferences of evaluators can have on the value of their evaluation, as a benchmark, even further, the authors of [30] employed *Inter-Rater-Reliability* statistics.

2.2.3. Inter-Rater-Reliability

Statistics like Cohens Kappa[9], Scott's Pi[25], Fleiss Kappa[13] and Gwets AC1-Statistic [10] are measurements for the similarity of the work, of two or more evaluators. How many evaluators can be compared at once varies between the listed statistics. All of these values originate from a psychological or sociological background. They are designed to compare evaluators whose evaluation consists of categorizing a person in a number of categories, based on the answers to arbitrary questions and chance observations. In this kind of scenario, one has to factor in, or rather out, a chance agreement of the evaluators, who all have their individual practices, yet have to use the same form. Inter-Rater-Reliability statistics aim to remove the habitual or by chance agreement, which results, from the equation. The general mathematical approach is to assume for a number of evaluations in a number of categories by a number of evaluators , that they were created by chance. The number of times where two or more evaluators agreed is totaled. From the totals, the likely hood for the evaluators to *agree by chance* is calculated and once known can be factored out.

The authors of [30] employed both Fleiss Kappa and the AC1-Statistic, to compare the results of their evaluation tool with manual evaluations. Fleiss Kappa is the more general of the two approaches and most Inter-Rater-Reliability statistics are calculated in an at least a similar fashion. All Inter-Rater-Reliability statistics have in common that they are intended to compare evaluation processes that happen on a nominal scale.

2.2.4. Informedness, Markedness

A popular and very well known technique to validate the results of machine learning algorithms, is to calculate *Precision and Recall*. It is commonly used in scenarios where an algorithm has to make a binary decision. *Precision* specifies for how many of the items, the algorithm chose a certain option for, that decision was correct. *Recall* specifies how many of the items, for that a certain option would have been correct, were categorized correctly. The concept is extended by *Informedness and Markedness* [22]. Unlike *Precision and Recall* these consider the *true negatives* in the calculation of the benchmark and are thus able to factor out the general likelihood of an item, to belong to a certain category.

3. QIT: Quality Inspection Tool

This chapter aims to present the design of *QIT*, a software that allows the inspection and validation of the evaluations INLOOM [14] produces. For this purpose, the requirements that are placed on the intended application are first compiled. The data sets available for validation are analyzed and a data structure to be employed by QIT is derived. Continuing to follow this data-driven approach, a possible design for the intended application is presented.

3.1. Software Requirements

INLOOM is supposed to evaluate student submissions to modelling tasks. Since this evaluation is to be done without further human supervision, the system must be tested thoroughly to avoid grading students unfairly or in error. The maintainer of INLOOM must be enabled to get an insight into the current quality of INLOOMs results and to quickly react to newly encountered sources of error. This leads directly to two leading requirements (SR) for the software proposed here.

SR1 The software must be able to automatically test the *quality* of the results INLOOM generates.

SR2 It must be possible to present the results of the performed validation in an easily comprehensible way, to quickly gain insight about the current state of affairs.

3.2. Available Data

The biggest limitation for the test system is the availability of test data. The complexity of solutions to modelling tasks cannot be predicted. There can be multiple correct solutions to the same task, which makes evaluating the submitted solutions an ill-defined problem. The literature research [4, 5, 14, 15, 26, 28, 32] showed that the only feasible method of validating automatic evaluations is to compare them to manually created ones. Manual evaluations are the gold standard against which automatic evaluations are measured. The *quality* of an automatic evaluation is expressed by the amount of deviation from the manual evaluation. The greater the discrepancy between the two, the lower the quality of the automatic evaluation.

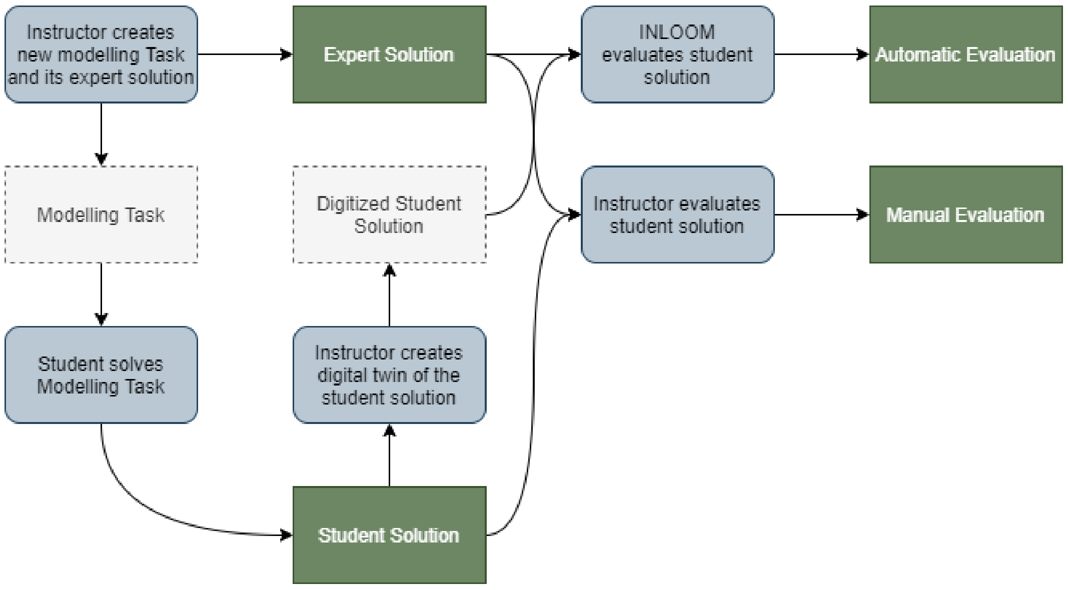
The usefulness of faking student solutions for quality testing is limited. Any testing done using faked up data would ultimately result in unit tests for the constraints. Thus, the only way to gain a reliable impression of the quality of INLOOMs results is to use it in a live scenario or to at least use real data for testing.

Every aspect of the design depends on the underlying data structure. The underlying data structure depends on the data available. As mentioned before, the data available consist of automatically generated evaluations for already manually graded student solutions, as well as the respective manual evaluations in analog form. This leads to an obvious third requirement or rather essential limitation, for the software proposed here.

SR3 All tests must be performed using the data available.

SR3.1 The software must employ a data structure that is able to hold information collected on automatic and manual evaluations and enables comparing the two.

Figure 3.1. Abstracted workflow of the creation of manual and automatic evaluations. Rectangles mark data elements, while ellipses represent process steps. The rectangles marked in green, represent the data available for testing purposes.


Instructor creates new modelling Task and its expert solution
Expert Solution
INLOOM evaluates student solution
Automatic Evaluation
Modelling Task
Digitized Student Solution
Instructor evaluates student solution
Manual Evaluation
Student solves Modelling Task
Instructor creates digital twin of the student solution
Student Solution

3.2.1. INLOOM Result Files

The first part of the available data are the results produced by INLOOM. INLOOM persists its results in form of XML files. Each of the XML files contains the results of the constraints applied to one student solution, as well as some meta-data. The format used in these files has changed since the publication of [14] and is now described by Document Type Definition (DTD) listed in Listing 3.1.

```
<!ELEMENT TestResult (TestData, Results, ResultPoints)>
<!ELEMENT TestData (ExpertModel, TestModel)>

<!ELEMENT ExpertModel EMPTY>
<!ATTLIST ExpertModel id CDATA #REQUIRED>

<!ELEMENT TestModel EMPTY>
<!ATTLIST TestModel id CDATA #REQUIRED>

<!ELEMENT MetaModel EMPTY>
<!ATTLIST MetaModel type CDATA #REQUIRED>

<!ELEMENT MCSIdentifier EMPTY>
<!ATTLIST MCSIdentifier id CDATA #REQUIRED>

<!ELEMENT MCSVersion EMPTY>
<!ATTLIST MCSVersion value CDATA #REQUIRED>

<!ELEMENT Results (CResult+)>

<!ELEMENT CResult (
    ExpertObject, ExpertType, TestObject, TestType
    Rule, Category, Points, Msg
)>
<!ELEMENT ExpertObject (#PCDATA)>
<!ELEMENT ExpertType (#PCDATA)>
<!ELEMENT TestObject (#PCDATA)>
<!ELEMENT TestType (#PCDATA)>
<!ELEMENT Rule (#PCDATA)>
<!ELEMENT Category (#PCDATA)>
<!ELEMENT Points (#PCDATA)>
<!ELEMENT Msg (#PCDATA)>

<!ELEMENT ResultPoints (MaxPoints, TestPoints)>
<!ELEMENT MaxPoints (#PCDATA)>
<!ELEMENT TestPoints (#PCDATA)>
```

Listing 3.1 XML format currently used to persist the results the INLOOM software generates.

The root of the XML files is the "TestResult". All meta-data is stored in "TestData", while the individual constraint results are persisted as a list of "CResult" under "Results". In the "TestData" branch, information about the evaluation is available. The *id* contained in "ExpertModel" references the expert solution the students work was compared to. "TestModel" identifies the evaluated student solution. "MetaModel", "MCSIdentifier" and "MCSVersion" contain versioning information about the rule set employed to generate the evaluation. All other data contained in the XML is constraint result specific. The results of the constraint are contained in the previously mentioned XML tag "Results" and are modelled using the XML element "CResult". Each such "CResult" identifies the element used during the constraint generation from the expert solution in "ExpertObject" and "ExpertType". The matching element of the student solution is stored in "TestObject" and "TestType". The "Object" Part holds the label or name of the used element. The "Type" Part stores the type of the element in the diagram. What types INLOOM is able to detect and grade, depends on the meta-model used for the evaluation [14]. The "Rule" tag of "CResult" holds the id of the rule employed to generate the "CResult". "Category" specifies the category assigned the match. Five categories can be assigned: Error, Warning, Correct, Missing, Info. The remaining tags "Points" and "Msg" contain the points awarded for the feature, the constraint tested, and an optional feedback message for the student.

The number of test data sets will most probably not increase dramatically in the near future, so there is no reason to reduce the result data in any way before using it for testing.

3.2.2. Manually Evaluated Student Solutions

The second part of the test data are manual evaluations of student solutions. Thirty already graded pen-and-paper student solutions to exam tasks were digitally reproduced by [14], to evaluate them using INLOOM. Of these thirty solutions, ten are solutions to the exam tasks of the summer term exams 2017, 2018, and 2019, respectively. Of the evaluations of this student solutions, only an analog version exists. It should be noted that only the evaluations of the summer term exam 2019 used the same uniform grading scheme for both the manual and automatic evaluation by default [14]. Every meaningful comparison of two evaluations requires the two to be based on the same grading scheme. This is the first formal limit (L) of the application.

L1 The system requires manual and automatic evaluations as inputs that were created using the same grading scheme.

The literature [4, 5, 14, 15, 26, 28, 32] agrees that the manual evaluations of these digitized student solutions are the best evaluations known for the specific solution and therefore, they are the only measure of quality one can apply to INLOOM. It can be assumed that the automatic evaluation's quality is sufficient, if it reaches the same result as the manual evaluation. To compare these manual evaluations with the ones automatically generated by INLOOM, however, they need to be digitized. Right now, the available manual evaluations consist of a number of handwritten annotations on the student solutions. The annotations are mostly check marks and points awarded for elements of the model. What feature the annotation references is indicated by its position in the student solution. Due to that format and the fact that the student solutions were stored as black and white scans, it is unlikely that the evaluation data can be automatically extracted. Therefore, it is necessary to provide an evaluation digitization facility.

SR4 The software must include a UI facility to digitize manually created evaluations of student solutions.

3.2.3. Data Driven Data Structure

There are some elements automatic and manual evaluations clearly have in common. Others are more oblique and some transformation is required before they can be assumed present in both. Each of the evaluations was made for exactly one *student solution* to solve exactly one *exercise*. Each evaluation can only ever be created by one *evaluator* and using one *expert solution* for reference. A comparison of an automatic and manual evaluation can thus be identified by a key that consists of the *student id*, the *exercise id* and the *expert solution id*. There is no point in comparing an automatic evaluation to a manual one, if they differ in one of these attributes. Such a comparison will later be called *TestDataSet*. Both kinds of evaluation need to contain these key attributes and rather obviously do.

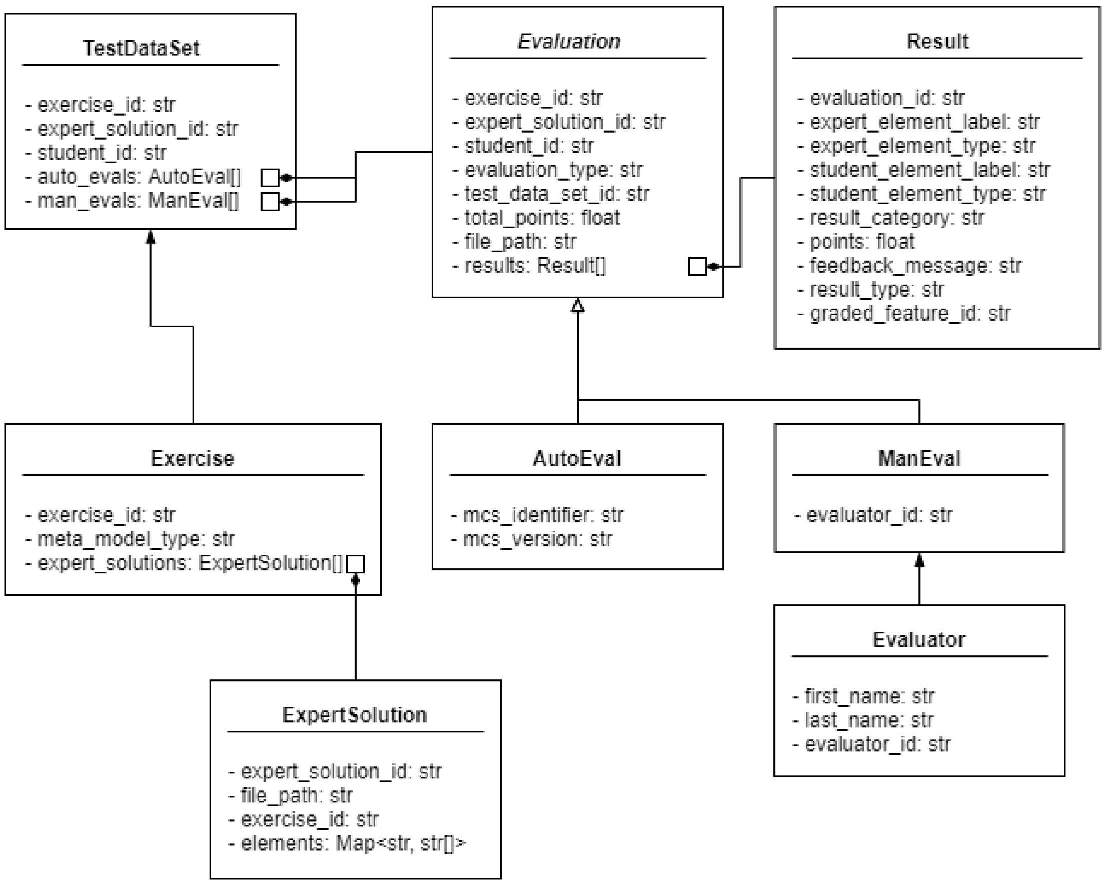
For one student solution, there can exist multiple automatic and manual evaluations. Multiple manual evaluators or different iterations of INLOOM, for example, using different rule set versions, can create evaluations for the same student solution. The literature [14, 26, 30, 31] points out that it can be interesting to inspect multiple manual evaluations of the same student solution, created by different evaluators. Each evaluator has his/her own style and preferences, which will be reflected in the evaluation. Enabling the application to store multiple manual corrections of the same student solution allows for easy calculation of previously described *clean scores* and any alike values.

SR3.2 The software must be able to persist multiple evaluations for the same student solution.

Every expert solution is a collection of elements and features the student solution needs to contain to be deemed correct. Both kinds of evaluation use an expert solution for reference. For each expected feature, the automatic evaluation adds a *Result* to its *Results*-list. The equivalent in the manual evaluation are the point annotations. Each of those awards points for a feature of the solution evaluated. A feature that has to be part of an expert solution in order for it to be correct. Each of the point annotations can thus be transformed into a "result" of the manual evaluation.

From the information collected about both the automatic and manual evaluations, using a data driven approach, a data structure can be inferred. Of the entities relevant to the system, only two can exist without any dependencies. The *Evaluator* and the *Exercise*. For each exercise, there can exist multiple *ExpertSolutions*. As previously described, a *TestDataSet* must reference an *Exercise*, an *ExpertSolution* and a student for it to be uniquely identifiable. For each *TestDataSet*, or rather each combination of keys that identify a *TestDataSet*, there can exist multiple automatic and manual evaluations: *AutoEvals* and *ManEvals*. Both of those are *Evaluations* and generally follow the format introduced by INLOOMs result XML files. The only difference between the two is that a manual evaluation must have been created by an *Evaluator*, while for the creation of an automatic evaluation the attributes *MCSIdentifier* and *MCSVersion* are required. The model presented in Figure 3.2 results from combining all of these required features.

Figure 3.2. Data model, that results from the structural analysis of the available data.



3.2.4. Digitizing Manual Evaluations

The need to digitize the manual evaluations before being able to compare them automatically to the evaluations INLOOM produces, means a substantial overhead for the testing process. Digitizing the manual evaluations is inevitable, however. This is reflected by SR4 (see 3.2.2). The effort required for digitizing manual evaluations must be reduced as much as possible. The process of extracting the data from the manual evaluations into a data structure like the one presented above, can roughly be separated into three steps and is the same for each manual evaluation.

1. Supplying identifying attributes (exercise, student, expert solution).
2. Supplying metadata (total points, evaluator).
3. Transforming point annotations into results and adding those results to the evaluation until the point total is accounted for.

SR4.1 The provided UI facility for digitizing manual evaluations must employ selections rather than inputs for as many of the values one has to enter as possible.

Since many of the attributes one needs to enter are repetitive and limited to a number of options, the UI facility must aim to always provide a selection rather than an input. This can clearly be done for both exercise and expert solution, since only a limited number of those are known to the application. The same holds true for the Evaluator.

More interesting, however, is easing the inputs required in the third step of the digitization. The third is the only step that will have to be repeated multiple times. In each repetition, six attributes need to be entered, in order to register a new *Result* of the manual evaluation one is digitizing.

1. Expert Element Label
2. Expert Element Type
3. Student Element Label
4. Student Element Type
5. Result Category
6. Points

Of these, only the student element label, student element type, and the awarded points are not generic. The rest of the attributes is limited to a number of options. The options, except the ones for the result category, are defined by the employed expert solution. The options valid for use as a result category are defined by INLOOM and are immutable. Adding a new *Result* can thus be reduced to entering three values and selecting the rest from predefined options.

3.3. KPI for Testing

The literature research showed quite clearly (see 2.2.1), that a *Grade Quotient Strategy* is the only real contender for evaluating the similarity of two evaluations. INLOOM is a constraint-based system and its results need not be compared on a nominal scale. It is thus not deemed necessary, to calculate any of the Inter-Rater-Reliability (IRR) statistics encountered in the literature. Assuming a nominal categorization would disregard a big amount of the data available. The authors of [30] assumed percentage difference intervals as categories artificially creating a nominal scale. Although this enables the employment of IRR, the values calculated based on such an assumption, do not add significant new information if no more than a single manual evaluation is available for comparison. Moreover, the results of IRR are often less intuitive than a simple quotient or difference percentage and are thus not deemed adequate to provide a first glance impression of the systems result quality.

IRR are designed to mitigate, the effects of habitual and random decision making, when validating evaluations that employ categories with ill-defined options. A big part of the randomness, in the typical use cases of IRR, results from these ill defined options. Since the evaluator is required to make a judgement call, it is possible, that not even the same evaluator will be able to repeat this decision, should they be confronted with an alike case. This makes for grade equivalents in these kind of evaluations, that can, by nature, not be vindicated after the fact. They are made up of partial decisions, that can not be repeated with any amount of certainty.

The use case of INLOOM is quite different. As previously described, the XML results of the system, list all the constraint results, the final grade is made up of. Thus, each step that leads to the final grade can be examined and it is most definitely possible to justify the final grade and to repeat the evaluation. For INLOOM one can conclude from its results, that the tool made an objectively wrong decision during its evaluation. This is not the case in IRRs typical use cases. In those one would usually not even be able to define *objectively wrong*.

Calculating IRR might still be interesting at a later point, since these statistics are employed in the literature and it might prove informative to be able to compare INLOOMs quality to the quality of other systems. IRR might also become interesting for INLOOMs use case, when more than one manual evaluation should be the norm rather than the exception, at some point in time. Under those circumstances, IRR will become useful to compare multiple manual evaluations against one another and create something alike a clean score automatically. Since the software proposed here is required by SR3.2, to be able to store multiple evaluations of the same student solution, it is opportune to add another requirement, that ensures the possibility to calculate additional KPI later on.

SR3.3 The data collected by the proposed application must be easily accessible to programmatic analysis, one might wish to perform on it in the future.

Since this work aims to validate the quality of a constraint-based system, with a uniform output format, no meta-analysis of the quality validation is required to extract additional information from a single value available. Instead of using meta-statistics to extract more information from the *Grade Quotient*, due to the uniform output format of INLOOMs result XML files, a more detailed analysis can be performed. As described earlier, data about each constraints result is available. These results can be numerically analyzed in a number of ways and on different levels of detail. A grade quotient equivalent can be calculated and compared in each category and for each level of detail. Such an examination will enable the user of the proposed application to not only gain a first impression on INLOOMs current performance, but also enable to quickly identify likely sources of error.

3.3.1. Comparison Detail Levels

A comparison of a manual and automatic evaluation, collected and persisted as described, can happen on a number of different levels of detail, or rather on a number of different scopes. On each of these detail levels, a number of categories has to be considered for comparison.

The finest detail level deemed interesting for meta-evaluation is an individual student solution. The application identifies each student solution using a key that consists of the student id, the exercise id, and the expert solution id. Each student solution translates to a *TestDataSet* in the application database. Each *TestDataSet* can be examined in a meta-evaluation and KPI can be calculated.

The lowest level of detail worth comparing is the average, across all meta-evaluations, of solutions compared to a given expert solution. A meta-evaluation on this level of detail provides a first glance impression on the performance of the assessment tool for a specific task and expert solution.

Less another level of detail but rather another scope, is meta-evaluating the evaluations created by a particular evaluator. Such a meta-evaluation could yield insights into the habits of different evaluators. By making data on the topic available it will be possible to better judge whether employing *Inter-Rater-Reliability* statistics is required.

3.3.2. Categories of Comparison

As was stated earlier, the main category of comparison of the two evaluations will be the grade. The final grade is a rating made up from a number of more detailed ratings and does by itself not allow for an inspection of its composition. After calculating such stats for each *TestDataSet* stored in the application database, they can be averaged over a group of these *TestDataSets*. QIT will most visibly present the comparison of the two grades, using a point difference percentage *Δ* (see 2.2.1).

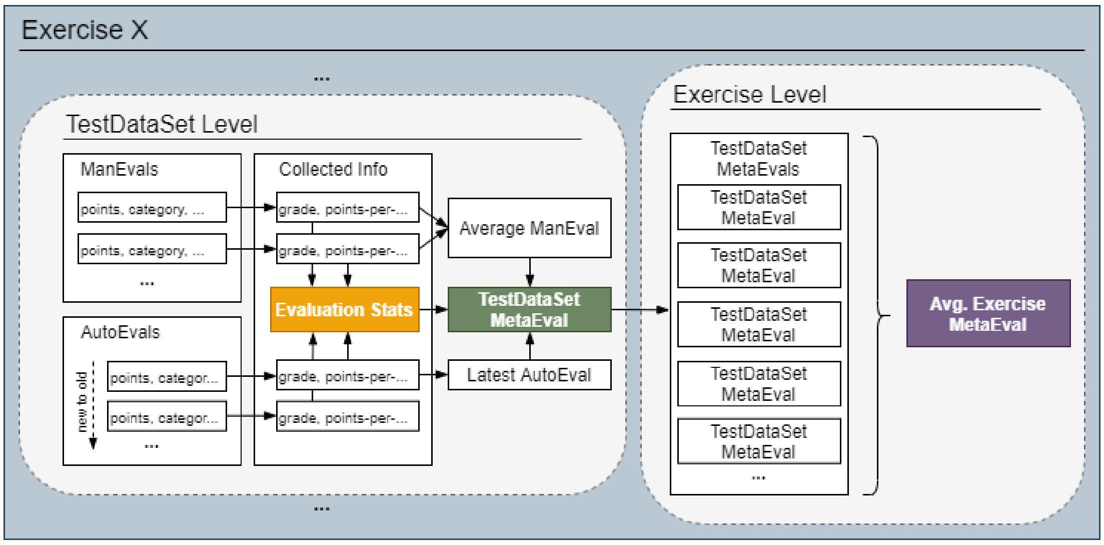


Figure 3.3. Abstract visualization of the undertaken comparisons on different levels of detail. The info-graphic is intended to clarify the composition of different levels of meta-evaluation result data. The elements marked in color are the elements data will be presented on.

In addition to the described difference percentage, a number of attributes of the evaluations meta-evaluated in the *TestDataSet* are worth presenting. QIT for both an individual *TestDataSet* and the average across groups of *TestDataSets* will display the number of points awarded by both the automatic and manual evaluation. Additionally, the grades, the evaluations awarded, and the grade Quotient *Q* that was described in Subsection 2.2.1, will be presented. Although more elaborate evaluations might prove interesting in the future, the application proposed here will only perform the most basic of evaluations on the *TestDataSet*-level. In compliance with SR3.3, it will later easily be possible to extend the application by any meta-evaluation feasible based on the available data. To enable the user to quickly identify sources of error, the application will group the found results by element type and sum up the points awarded to each group. The result of this evaluation will be tagged *points-per-expert-element-type*. The evaluation is intended to give the user an easy grasp of where eventual discrepancies originate. A second evaluation performed on the *TestDataSet*-level is tagged *points-per-result-category*. In it, the points awarded in combination with a certain result category are totaled. This meta-evaluation is intended to provide insights into the precision of the assessment algorithm. The last meta-evaluation performed is less an evaluation than rather a collection. To quickly identify differences in the categories, results were flagged with, in the compared evaluations, the categories assigned to each result are collected. Such a collection can be presented as a list view to ease a detailed error analysis of the assigned category flags.

As is expressed by the perceived need to employ IRR [30], evaluations by different reviewers can differ significantly. The application must take this into consideration and give the user the means to check for any negative effects using manual evaluations by multiple evaluators might have on the quality of the systems results. This is remedied by QITs ability to store more than one manual evaluation per student solution. Each described comparison on the *TestDataSet*-level is performed for an average manual evaluation. This *AvgManEval* results from calculating the mean of every meta-evaluation result over all *Evaluations* present in a *TestDataSet*. The average of its evaluations is the central component of every *TestDataSets* meta-evaluation.

TestDataSet-meta-evaluations will be grouped by evaluator, exercise or expert solution. By calculating an average value for the evaluations of a specific group, the user can examine INLOOMs results for error trends that arise from the structure of a specific expert solution, the style of a particular evaluator or in the context of a certain element type.

3.3.3. Visualizing Validation Results

Visualizations are unquestionably one of the most effective ways of presenting data. For visualizing the results of the validation, the proposed software is supposed to perform, several types of visualization are available for each presented level of detail.

The point difference percentage is the most comprehensive of the values calculated by the proposed software. Presenting it is one of the main tasks of the software's frontend. For that reason, it should be presented to the user as often as possible. However, visualizing a percentage difference is only interesting in contexts where there is more than one *TestDataSet*-meta-evaluation is available. Visualizing a single percentage would not gain anything. Individual values will therefore be presented to the user as they are.

Multiple values are available when grouping test data sets. This, as was described in 3.3.2, will be done using the evaluator id and the expert solution id as keys. In these cases, multiple point difference percentages have to be presented. The purpose of this kind of presentation is identifying unusual elements in the groups. Bar charts are suitable for this use case. Since difference percentages are being visualized, the range of possible values is limited and the individual values can easily be compared without any further need for preprocessing the data. A visualization of the values in form of a bar chart will enable easy comparison between items of the groups and thus help identify unusual items. Bar charts will also be employed to visualize the grade quotient and absolute point difference on the exercise level.

The results of the *TestDataSet*-level meta-evaluation tagged *points-per-expert-element-type* will also be displayed using a bar chart. Since the results of this validation are not very complex, but consist only of a number of points, awarded for a specific element type, by one of the compared evaluations, a simple bar chart is again suitable for their visualization. To enable easy comparison between the points awarded to a type by the automatic and manual evaluation, the points of both can be presented as adjacent bars. This will allow for the comparison of the two evaluations to one another while also enabling the assessment of the relative relevance of the inspected element type for the final score. The same chart implementation can be used for presenting the results of the meta-evaluation tagged points-per-category.

3.4. Summary

In this section the requirements collected are summarized. For each requirement, it is checked whether the proposed design is capable of satisfying it.

SR1 The software must be able to automatically test the *quality* of the results INLOOM generates.
Result: The software will be able to automatically validate the quality of INLOOMs result files, employing a strategy that is an extension of the *Grade Quotient Strategies* described in the literature.

SR2 It must be possible to present the results of the performed validation in an easily comprehensible way, to quickly gain insight about the current state of affairs.
Result: The software will visualize the results of the validations it performs using bar charts as well as common list and detail views of the stored data.

SR3 All tests must be performed using the data available.

- SR3.1 The software must employ a data structure that is able to hold information collected on automatic and manual evaluations and enables comparing the two.
  Result: Based on a data-driven approach, such a data structure was derived from the structures of the data available for quality testing.
- SR3.2 The software must be able to persist multiple evaluations for the same student solution.
  Result: The data structure described in 3.2.3 is up to this requirement.
- SR3.3 The data collected by the proposed application must be easily accessible to programmatic analysis, one might wish to perform on it in the future.
  Result: This requirement will be trivially tackled by storing all collected data in an independent database and providing all required interfacing functionalities.

SR4 The software must include a UI facility to digitize manually created evaluations of student solutions.
Result: The software will incorporate an adequate facility, as was described in Section 3.2.4.

- SR4.1 The provided UI facility for digitizing manual evaluations must employ selections rather than inputs for as many of the values one has to enter as possible.
  Result: The required values were analyzed and it was found that many of them can be entered using a selection rather than an input (see Section 3.2.4).

L1 The system requires manual and automatic evaluations as inputs that were created using the same grading scheme.

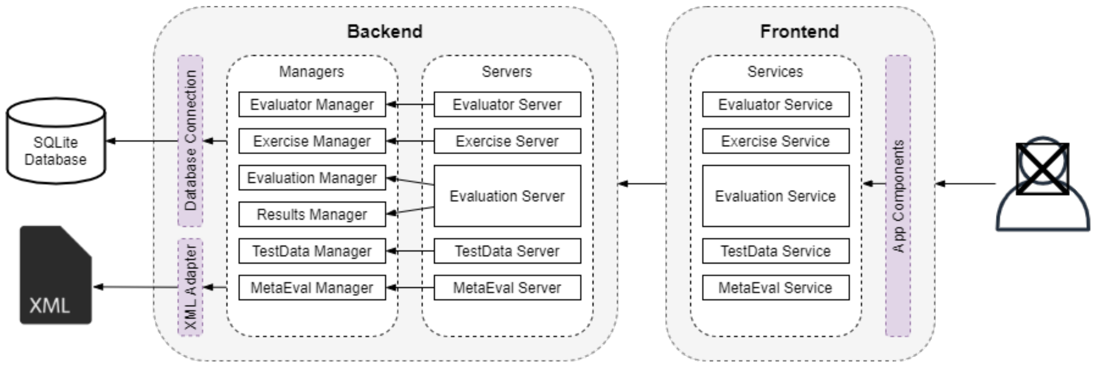
4. Implementation

An application that satisfies the listed requirements, has to be implemented. This section intends to present the implementation of QIT. An application for inspecting and validating the evaluation INLOOM generates.

4.1. Implementation as a Web-Application

The implementation proposed here was strongly influenced by the fact that INLOOM is currently a work-in-progress. It is thus inevitable that the circumstances of the implemented applications deployment will be subject to change in the future. This is reflected by the requirement SR3.3. It implies that it might become necessary to access the collected data for other purposes than the proposed applications. To enable easy modularizability, the application is implemented as a web application that will be locally deployed. The usual design of a web application allows for easy separation of concerns. All data the application uses is stored in a SQLite[1] file. A python Flask[2] app provides a simple URL API to access the stored and process input XML data. The front-end application, the user accesses, is implemented using Angular[3].

Figure 4.1. Abstract depiction of the proposed applications components.



Many modes of deployment are feasible for such an application and more than local deployment might become interesting in the future. For example, multiple reviewers could be adding evaluation data to the same database, while in the same local network.

[1]https://www.sqlite.org/
[2]https://flask.palletsprojects.com/
[3]https://angular.io/

Separating the app into the described parts makes it possible to reuse the collected data in multiple ways with ease. A new front-end application could be implemented to visualize other aspects of the collected data. In another scenario, a completely different application could access the database independently. Also feasible is adapting the back-end to access data from another kind of database. The previously described scenario, where multiple reviewers work on the same data would thus be possible without publicly hosting the application itself. This kind of flexibility in both implementation and deployment is unique to web apps and makes a web app an optimal architecture template for the proposed application.

4.2. Backend Implementation

The backend of the web app is implemented in Python using the Flask framework. Flask is one of several lightweight python frameworks for setting up web-applications and was chosen for its high flexibility. Only the most basic functionalities the framework supplies are employed. The backend is separated into two parts. The managers and the servers. All data processing is performed by the manager classes. The server classes supply the web API. The managers use a supplied database connection service to interact with SQLite. For reading the INLOOM result files, a XML adapter service is provided. The database is passive and *managed by* the QITs Backend. The frontend requires the backend to store and provide data and to perform the meta-evaluation of the stored data, whose results the frontend is intended to visualize. The data the backend is required to handle was described in 3.2.3. The presented model is extended by id and timestamp attributes required for processing. For the results of the meta-evaluation, there does not exist a specialized data element since it is performed each time requested and its results are not persisted. They consist of a collection of methods to read and write on the database and to transform the handled data to either store or serve it. It might be important to mention that the manager classes create the required database tables, should they not already exist. Thus the manager and the data element class are inextricably linked and must be understood as a unit. A manager is not meant to handle object classes *like* the one it was originally intended for, but only *that exact one*. If a data element class should be adapted in the future, the manager class will also need to be adapted.

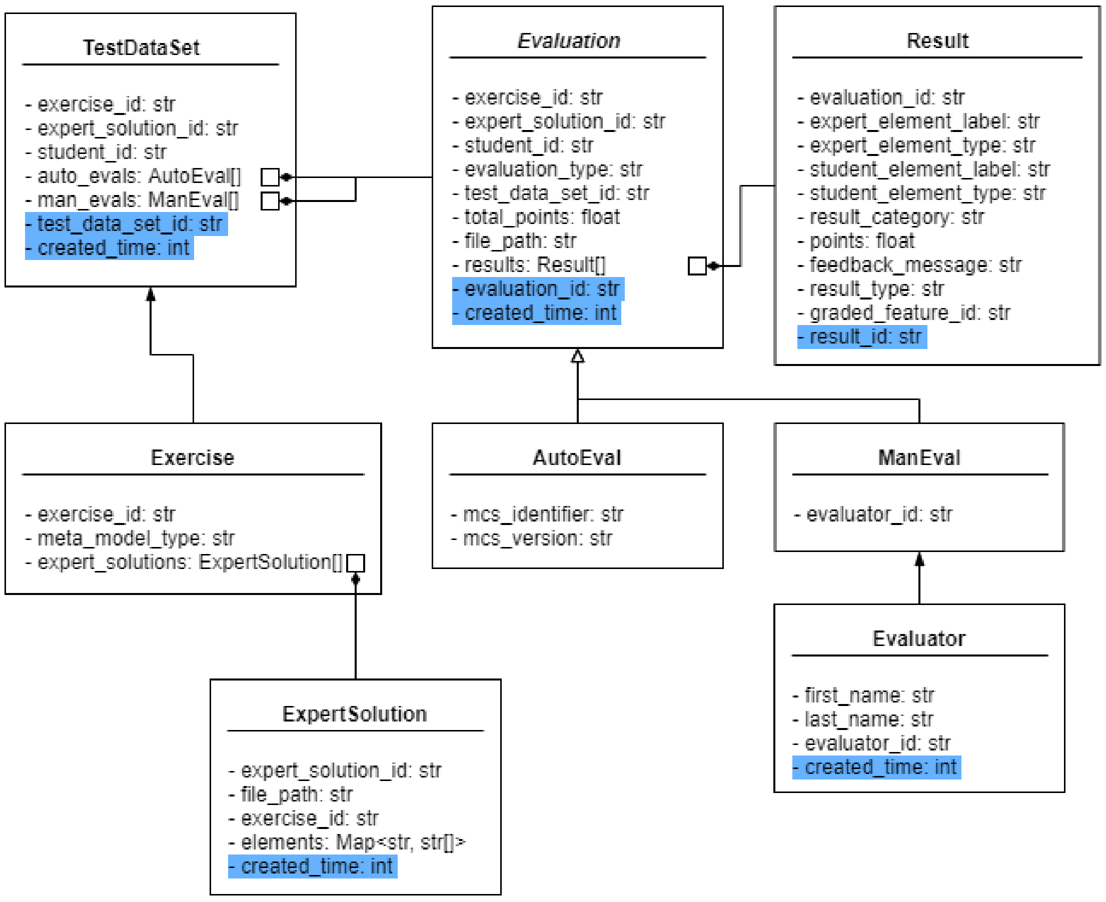


Figure 4.2. Extended diagram representation of the employed data elements. The newly added attributes are marked in blue.

4.2.1. Data Handling

As figure 4.1 is intended to illustrate: the question: "Who is responsible for what?" can be answered quite trivially.

Table 4.1. Data element classes, listed with the table their instances are stored in and operatives responsible for handling them.

Data Element Class	Database Tables	Manager Class	Server Class
TestDataSet	test_data_sets	TDManager	TDServer
Evaluation, AutoEval, ManEval	evaluations, man_evals, auto_evals	EvalManager	EvalServer
Result	results	ResultsManager	
Evaluator	evaluators	EvaluatorManager	EvaluatorServer
Exercise	exercises	ExerciseManager	ExerciseServer
ExpertSolution	expert_solutions		

Table 4.1 lists all data elements relevant to the application along with the database table the elements are stored in. The third and fourth columns of the table contain the manager and server responsible for handling them. For each of the data elements the software employs, there is a manager to manage it and a server to provide an API to access the functionalities the manager provides. There are two exceptions to this rule. First: For the results, there is no separate server. Result data is provided to the frontend by the EvalServer. This is because Results can only exist in the context of an Eval and there is no interest in inspecting them on an individual basis. The Results are an integral part of every Eval and are thus provided to the frontend as such. The second exception are ExpertSolutions. For those, there does not even exist a separate Manager. The ExpertSolution and the Exercise are in large parts equivalent items since almost all data about the Exercise is extracted from the actual expert solution, or rather from the result XML of an evaluation of an expert solution against itself. The exercise serves only as a key to collect multiple expert solutions on and to persist data common to all expert solutions it identifies in. The mentioned extraction process, required to collect data on the expert solutions, is further detailed in Section 4.2.2. The managers use a supplied database connection service to interact with the database and their implementations are for the most part quite generic.

4.2.2. XML Adapter

The data the proposed app is supposed to work on, can originate from one of two sources. Data on manual evaluations is entered by the user. Data on automatic evaluations is supplied in form of result XML files, INLOOM generates. The XML Adapter services task is to process input result XML files and extract all relevant information. It must be differentiated between extracting evaluation data and extracting expert solution data. Both extraction processes are performed by the XML Adapter but result in completely different data sets. Extracting the evaluation from the XML file means converting the data from XML to the representation employed by the proposed application. Extracting an expert solution from the evaluation XML file means surveying and collecting meta-data on it. Since the format of the XML file and all required XML tags are known (see Listing 3.1), the task of the XML Adapter is limited to reading the XML file and searching for all specified tags in it. All encountered data is collected into an Evaluation object. Each of the Result-tags listed in the XML file is transformed into a Result object. The Results are held by the evaluation.

The extraction of the expert solution is done in the same way. Instead of storing the collected data as an Evaluation, however, the data is processed further, to create an ExpertSolution. The ExpertSolution shares some of the values of the evaluation, but storing each result is not required. The solution is no evaluation and thus does not need to persist a mapping between two elements, but only a list of elements is contained. The values INLOOM stores in the ElementLabel tag are encoded for some of the available element types and do not always match the actual label of the element in the expert solution. This is, for example true for relationship elements of the expert solution. These are encoded as "rX" where "X" is the index of an enumeration of all relationship elements. To present the element in question to the user as a selection option, to comply with SR3.2.4, a more verbose label is required. To obtain such a label, the feedback messages, provided by every result are analyzed. Since these feedback messages are presented to the student and are therefore required to be rather verbose, most of them contain at least some additional information on the element graded by the analyzed result. The XML Adapter is primed to extract this information using element type specific regular expressions to search the feedback messages. The extraction process results in an ExpertSolution object, that holds a list of elements available in the expert solution. For each element, the content of the original ElementLabel XML tag (*name*), the element *type*, and a *label* is stored. The *name* is gathered from the content of the ElementLabel XML tag, while the *label* holds a more verbose label, enriched with the information extracted from the feedback messages. The intention is, to enable the user to identify the element in a student solution he is digitizing. Dealing with encoded element labels as options would hinder that.

The described procedures for reading data from the XML files require the files to have an immutable format. They must contain all tags the adapter tries to read from and use a format for the feedback messages the supplied regular expressions are compatible with. If the format of the XML files should be subject to change in the future, the XML Adapter will have to be adjusted as well. It was implemented concerning this fact. It allows setting the names of the tags, the reader tries to read a specific value from, as well as setting the employed regular expression on a per element-type basis.

4.2.3. Meta-Evaluation

The purpose of the application proposed here is to give the user an impression of the current state of the system. It aims to provide all facilities to evaluate INLOOMs evaluation. For that purpose it creates a *Meta Evaluation* or *MetaEval*. The MetaEval is created by the backend of the software. The collected evaluation data is evaluated and several KPI values are calculated. This task is performed by the *MetaEvalManager*. The class to provide meta evaluations is implemented as a manager since it operates on the same level as the other managers and is required to interact with the database. The meta-evaluation is performed on a per TestDataSet basis. Each TestDataSet can reference multiple manual and automatic evaluations. For each evaluation referenced, several data collection tasks are performed. The data that results from the meta-evaluation is collected in a Python dictionary and sent to the frontend encoded as JSON.

The process to create the meta-evaluations can be separated into three distinct parts. In the first step, all evaluations the *TestDataSet* references are collected. In a second step, a number of data collection tasks are performed for each found evaluation. In the third and final step, the data, collected on the evaluations, is processed further, to calculate averages over multiple evaluations. The *TestDataSet*-meta-evaluation is created using average manual evaluation values and the latest automatic evaluation. The collection tasks mentioned, result in separate evaluation specific sub mappings. Listing 4.2.3 shows the implementation of the first two steps of the process. First, all evaluations of a *TestDataSet* identified by id are queried from the EvalManager.

Listing 4.1 First part of the python implementation for collecting and evaluating Evaluations registered for a TestDataSet. The listed function *get_tds_meta_eval* is part of the MetaEvalManager.

```python
def get_tds_meta_eval(self, test_data_set_id: str) -> Mapping:
    """Get meta-eval for a test data set."""

    # Creating a dict to store the results in
    tds_meta_eval: Dict = {}

    # Get all evaluations for this tds from the database
    auto_evals: List[AutoEval] = EvalManager() \
        .get_all_of(test_data_set_id, AutoEval)
    man_evals: List[ManEval] = EvalManager() \
        .get_all_of(test_data_set_id, ManEval)

    ...

    # Get the ExpertSolution employed
    expert_solution: ExpertSolution = ExerciseManager() \
        .get_one_expert_solution(
        auto_evals[0].exercise_id,
        auto_evals[0].expert_solution_id)

    # Collect the stats for the Evaluations
    # {.., evalId -> {stat: {..}, ..}, ..}

    tds_meta_eval['eval-stats'] = {}
    auto_eval_stats: Mapping[str, MutableMapping] = {
        auto_eval.evaluation_id: self._collect_information(auto_eval, expert_solution)
        for auto_eval in auto_evals
    }
    tds_meta_eval['eval-stats'].update(auto_eval_stats)

    man_eval_stats: Mapping[str, MutableMapping] = {
        man_eval.evaluation_id: self._collect_information(man_eval, expert_solution)
        for man_eval in man_evals
    }
    tds_meta_eval['eval-stats'].update(man_eval_stats)
    ...
```

For each of the thus collected Man- and AutoEvals, a number of data collection tasks is performed. To enable the developer to add new data collection tasks with ease, the tasks are collected in a list of tasks to perform. By adding a new function to the list, one adds the collection it performs to the mapping that results from the meta-evaluation.

Listing 4.2 Python implementation to commence data collection. The listed function is part of the MetaEvalManager.

```python
def _collect_information(self, evaluation: Evaluation, expert_solution: ExpertSolution) -> Mapping:
    """Collect information about an evaluation."""

    stats: Dict[str, Any] = {}

    # Run all listed functions on the list of results

    for _func in [
        self._collect_points_per_element_type,
        self._collect_points_per_result_category,
        partial(self._calculate_grade, expert_solution)
    ]:
        key, value = _func(evaluation)
        stats[key] = value

    # Add meta-data for convenience
    stats['created'] = evaluation.created_time
    stats['type'] = evaluation.evaluation_type

    return stats
```

In addition to the function employed to gather information from the evaluations, the manager defines a function to calculate the average of every key of several evaluation information mappings, as they result from the function listed in 4.2.3. The average-function results in a mapping that contains an entry for each key found in the original mappings. The entries of the resulting mapping are composed of the key found in the original mapping and the average value of the values found for this key in the input mappings.

Listing 4.3 Python implementation for calculating the average of every key of several alike mappings. The listed function is part of the MetaEvalManager.

```python
@staticmethod
def _calculate_dict_average(stats: Mapping[str, Mapping[str, Mapping]]):
    """Calculate the average for every key in a dict and
    reconstruct a dictionary with the same structure."""

    eval_stat_keys: Set[str] = set()

    for stat in stats.values():
        eval_stat_keys.update(set(stat.keys()))

    # For each key in eval_stat_keys
    # calculate the average of the values of
    # that key in the mappings in stats

    collected_stats: Mapping = defaultdict(lambda: defaultdict(list))

    for stat in stats.values():
        for eval_key in eval_stat_keys:
            collection: Mapping[str, float] = stat \
                .get(eval_key, defaultdict(lambda: defaultdict(float)))

            for col_key, col_value in collection.items():
                collected_stats[eval_key][col_key].append(col_value)

    average_stats: Dict[str, Mapping] = {
        key: {k: mean(v) for k, v in value.items()}
        for key, value in collected_stats.items()
    }
    return average_stats
```

The function listed in 4.2.3 is employed to generate an *average manual evaluation* from all evaluations registered for the evaluated TestDataSet. The calculated average values are used to calculate TestDataSet level meta-evaluation stats.

Listing 4.4 Second part of the python implementation for collecting and evaluating Evaluations registered for a TestDataSet. The listed function *get_tds_meta_eval* is part of the MetaEvalManager.

```
def get_tds_meta_eval(self, test_data_set_id: str) -> Mapping:
    """Get meta-eval for a test data set."""

    ...

    # Get the latest auto eval from the available ones
    auto_evals: List[Evaluation] = sorted(
        auto_evals, key=lambda e: e.created_time)

    latest_auto_eval: Evaluation = auto_evals[-1]

    # Calculate TDS Level Statistics
    grade_quotients: Dict[str, float] = {
        m_eval.evaluation_id: (latest_auto_eval.total_points / m_eval.total_points)
        for m_eval in man_evals
    }

    # Calculate average points
    avg_man_total_points: float = mean([e.total_points for e in man_evals])
    latest_auto_total_points: float = latest_auto_eval.total_points

    # Calculate grade quotients
    grade_quotients['average-man-eval'] = \
        latest_auto_total_points / avg_man_total_points
    tds_meta_eval['grade-quotients'] = grade_quotients

    # Getting stats on the latest auto eval
    latest_auto_eval_stats: MutableMapping = auto_eval_stats.get(latest_auto_eval.evaluation_id)
    latest_auto_eval_stats['total-points'] = latest_auto_total_points
    latest_auto_eval_stats['grade'] = (latest_auto_total_points * 100) / expert_solution.maximum_points
    tds_meta_eval['eval-stats']['latest-auto-eval'] = latest_auto_eval_stats

    # Getting the average of all the
    # entries of the man eval stats
    avg_man_eval_stats: MutableMapping = self \
        ._calculate_dict_average(man_eval_stats)
    avg_man_eval_stats['total-points'] = avg_man_total_points
    avg_man_eval_stats['grade'] = (avg_man_total_points * 100) / expert_solution.maximum_points
    tds_meta_eval['eval-stats']['avg-man-eval'] = avg_man_eval_stats

    # Append id attributes
    tds_meta_eval['student-id'] = latest_auto_eval.student_id
    tds_meta_eval['expert-solution-id'] = latest_auto_eval.expert_solution_id
    tds_meta_eval['exercise-id'] = latest_auto_eval.exercise_id

    return tds_meta_eval
```

The meta-evaluation that results from the function listed in 4.2.3 and 4.2.3 is delivered to the frontend by the MetaEvalServer.

4.3. Frontend Implementation

Both SR2 and SR4 require the proposed application to have a frontend for interacting with the stored evaluation data and adding to it. This frontend is implemented using Angular. Angular is one of several modern java-/typescript frameworks. Examples of competitors that would have been equally up to the task are Vue and React. However, neither framework offers significant advantages over Angular for the use case at hand.

4.3.1. Services

For each server in the backend, there exists an Angular-Service in the frontend. Services in Angular are injectable singletons and can be employed by all components the application uses. Due to their singleton nature they can be used to cache the data they get from the server. This is done to reduce the amount of API queries to a necessary minimum. After an initialization, where the service queries data from the server once, a refresh of that data will only take place if a service is explicitly instructed to refresh it.

4.3.2. Page Tree

The frontend was implemented using a data driven approach. All views it presents exist in the context of one or more of the previously described data elements.

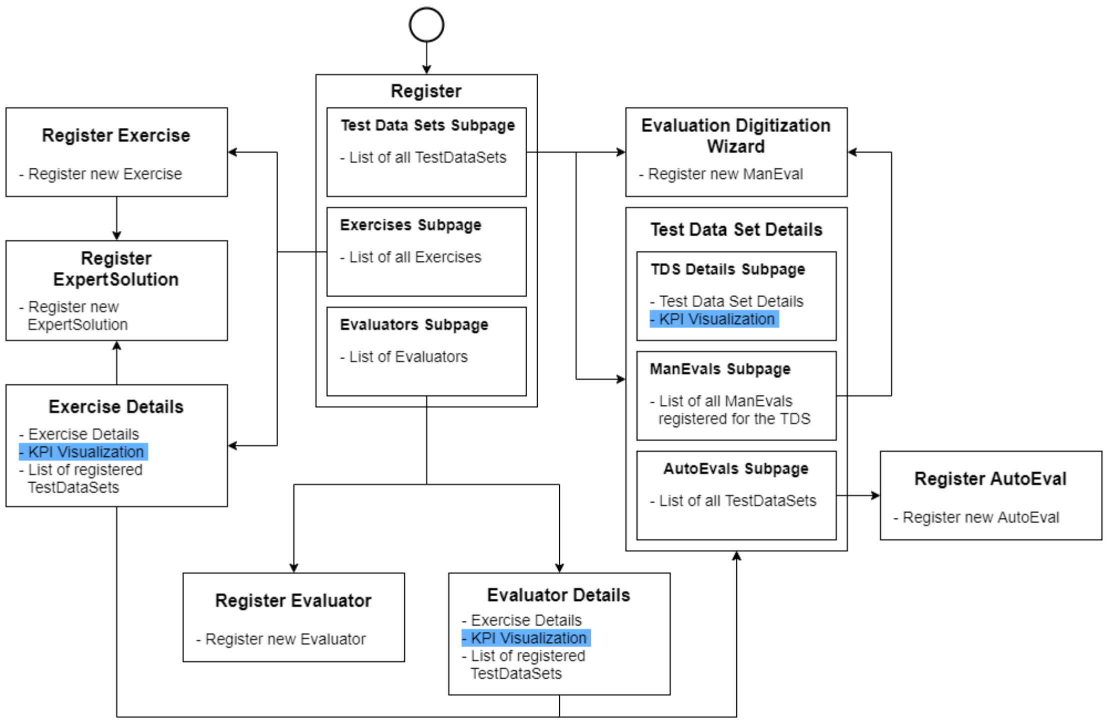
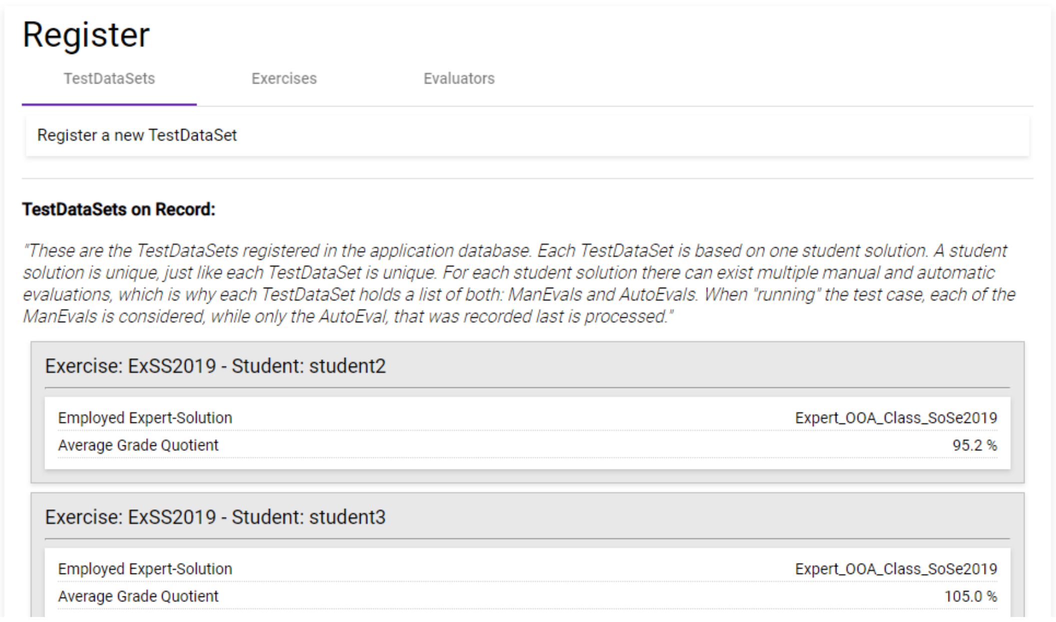


Figure 4.3. Abstract visualization of the navigation options the frontend supplies to the user. Visualizations are marked in blue for reference in ??.

This description of the frontend will detail the individual views in the order in which they are intended to be used. The use case is assumed that a new user wants to enter a completely new test data set. In order to do so, a new evaluator and a new task must first be defined. The central entry point of the frontend application is the *Register*. Starting from here, the user has the opportunity to inspect all data elements registered


Register

TestDataSets Exercises Evaluators

Register a new TestDataSet

TestDataSets on Record:

"These are the TestDataSets registered in the application database. Each TestDataSet is based on one student solution. A student solution is unique, just like each TestDataSet is unique. For each student solution there can exist multiple manual and automatic evaluations, which is why each TestDataSet holds a list of both: ManEvals and AutoEvals. When "running" the test case, each of the ManEvals is considered, while only the AutoEval, that was recorded last is processed."

Exercise: ExSS2019 - Student: student2

Employed Expert-Solution	Expert_OOA_Class_SoSe2019
Average Grade Quotient	95.2 %

Exercise: ExSS2019 - Student: student3

Employed Expert-Solution	Expert_OOA_Class_SoSe2019
Average Grade Quotient	105.0 %

Figure 4.4. Exemplary screenshot of the *Register* view where the TestDataSet sub-page is opened. The general structure of the illustrated view is repeated for both the Exercise and Evaluator sub-pages.

in the application database. The *Register* possesses three sub-pages, as is illustrated in Figure ??. The first - *Test Data Set Subpage* lists all known TestDataSets. Each element of the list can be clicked to navigate to a detail view of the TestDataSet. The Exercises sub-page lists all stored Exercises. Again each element of the list can be clicked to maneuver to a detail view of the Exercise. The last sub-page used to present Evaluator data elements works the same. To register a new evaluator with the application the user navigates to the *Register Evaluator* view. He is required to enter all relevant data. The next required step is adding a new Exercise. Again the user navigates to the appropriate view. He is forwarded to the *Register ExpertSolution* view, for reasons discussed in 4.2.2. To add a new ExpertSolution the user is required to upload an XML file that resulted from INLOOMs evaluation of the employed expert solution. Adding an *ExpertSolution* will automatically add an *Exercise*. Additional *ExpertSolutions* can be added to a task later. However, there can never be a case where there is an *Exercise* without at least one *ExpertSolution*. With both an *Evaluator* and an *Exercise* registered in the application database, the next step is adding the TestDataSet itself. For this purpose the user navigates to the *Evaluation Digitization Wizard*, which will be discussed in its own subsection (Section 4.3.3). After completing all steps of the wizard though, only half the data required to make up a *TestDataSet* was entered. The user still needs to add an *AutoEval* to compare the added *ManEval* to. He does this by navigating to the *TestDataSet Details* view of the *TestDataSet* he just created by adding a new *ManEval*. From here he can access the input mask that allows him to upload the result XML of an automatic evaluation. Most if not all of the views feature an explanatory text, that is intended to guide users through the process.

After adding at least one *TestDataSet* in its entirety, the user has access to the *Detail Views* regarding Evaluator, Exercise as well as TestDataSet, that will be discussed in their own subsections.

4.3.3. Evaluation-Digitization-Wizard

The Evaluation-Digitization-Wizard is to the manual evaluations, what the XML Adapter is to the automatic ones. It is the facility employed to extract data from the manual evaluations and ensures compliance with SR4. The user must enter all relevant data on a manual solution, as was described in 3.2.4. For each of the three described steps, the wizard has a page. Other than one page for each of the three steps, there is only one additional page to the wizard. On an initial page, the user has the opportunity to select the PDF file of a manual evaluation. The user is not required to perform this step. If he does, the PDF file is shown to the side and can be used for easy reference. The titles of the wizard pages are the following:

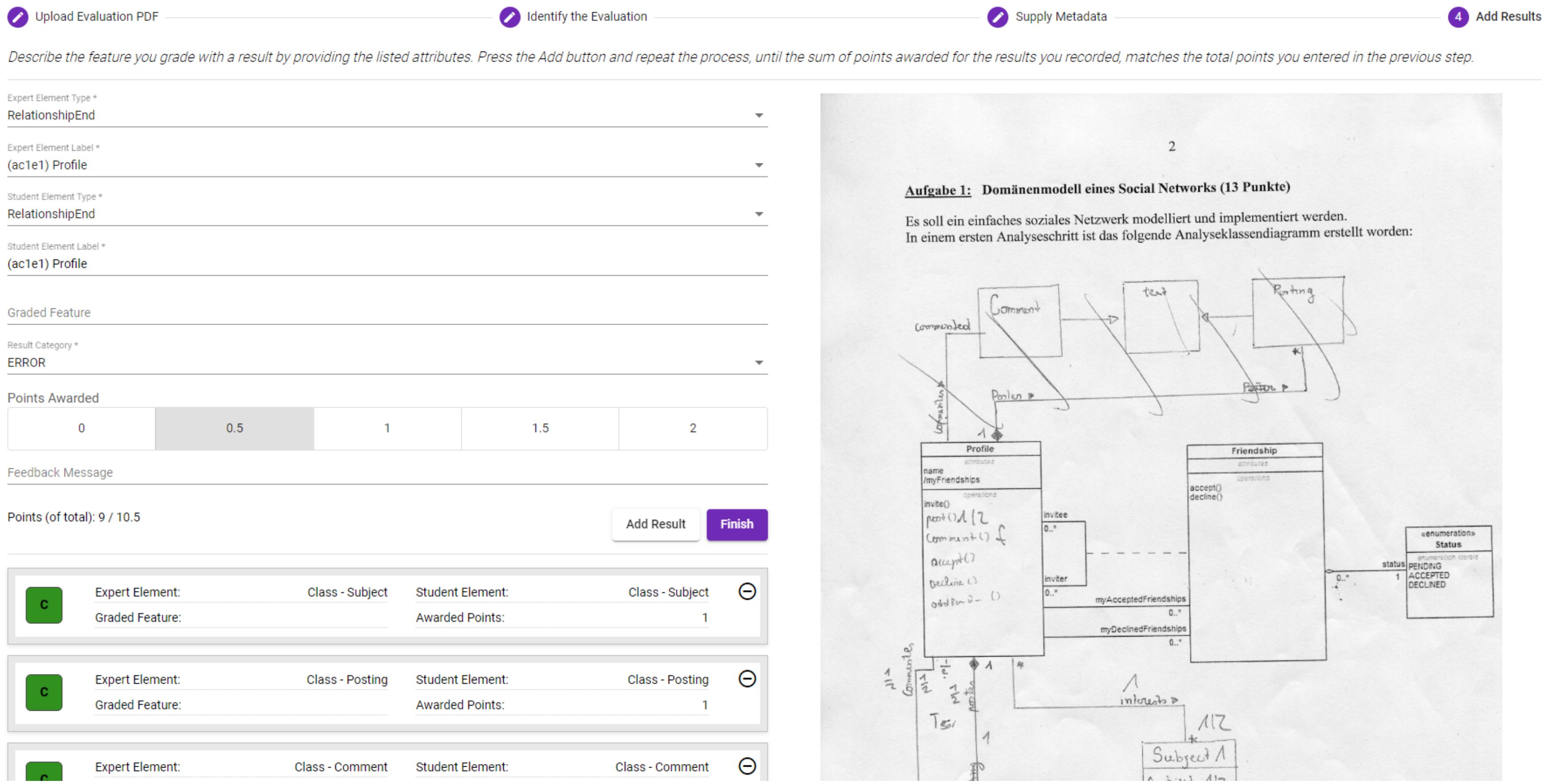
1. Upload Evaluation PDF
2. Identify the Evaluation
3. Supply Metadata
4. Add Results.

The first three wizard pages are rather generic and allow for entering the data required by the step they were implemented for. As was already described, the first wizard page allows selecting the PDF file of a manual evaluation. The second wizard page, requires the user to specify the solved exercise, the expert solution it was evaluated against and the id of the student who submitted the evaluated solution. On the third wizard page, the evaluator and total points awarded to the submitted solution need to be specified. The fourth page is the most complex.

The user has to create a Result for each graded element of the student solution. SR4.1 demands as many options rather than inputs as possible. For each Result the user wants to create, he has to select an *Expert Element Type*, an *Expert Element Label*, a *Student Element Type*, a *Result Category* and the number of *Points Awarded* for the graded feature. For each Result only one other attribute, the *Student Element Label* must be entered. Although, the user may choose to enter a *Graded Feature* and a *Feedback Message*. The first is intended to be used to further describe, what the points awarded were awarded for. The *Feedback Message* field can be used to persist any textual feedback annotation the manual evaluator may have left while reviewing the student solution. It may be interesting to remark, that this wizard is where the enriched labels created by the XML Adapter come into play. They are used as options for the Expert Element Label. Registering a Result for the evaluation is further eased by the fact, that setting an Expert Element Label, will by default set the Student Element Label to the same value.This is based on the expectation, that most times, the matched elements will employ the same label. The same holds true for the Expert Element Type Once a Result is added, it is shown in a list view of all currently added results. Figure 4.5 - a screenshot of the fourth wizard page is intended to convey the general idea of the wizard.

4.3.4. TestDataSet Details View

Each complete *TestDataSet* represents a comparison of at least one *ManEval* with at least one *AutoEval*. The *Detail View* of a TestDataSet represents the results of the performed comparison. This is done at different levels of detail. As mentioned before, for each TestDataSet at the highest level, a theoretical average manual correction is compared with the most recently added automatic correction. The results of this



Upload Evaluation PDF — Identify the Evaluation — Supply Metadata — 4 Add Results

Describe the feature you grade with a result by providing the listed attributes. Press the Add button and repeat the process, until the sum of points awarded for the results you recorded, matches the total points you entered in the previous step.

Expert Element Type *
RelationshipEnd

Expert Element Label *
(ac1e1) Profile

Student Element Type *
RelationshipEnd

Student Element Label *
(ac1e1) Profile

Graded Feature

Result Category *
ERROR

Points Awarded

0	0.5	1	1.5	2

Feedback Message

Points (of total): 9 / 10.5

Add Result | Finish

C	Expert Element:	Class - Subject	Student Element:	Class - Subject
	Graded Feature:		Awarded Points:	1
C	Expert Element:	Class - Posting	Student Element:	Class - Posting
	Graded Feature:		Awarded Points:	1
C	Expert Element:	Class - Comment	Student Element:	Class - Comment

2

Aufgabe 1: **Domänenmodell eines Social Networks (13 Punkte)**

Es soll ein einfaches soziales Netzwerk modelliert und implementiert werden.
In einem ersten Analyseschritt ist das folgende Analyseklassendiagramm erstellt worden:

Figure 4.5. Screenshot of the fourth wizard page described in 4.3.3 while creating results from a manual evaluation. The PDF file of the manual evaluation can be seen on the right, while on the left both the form for entering result values and the result list are located.

comparison are presented on the *TDS Details Subpage*. The *TDS Details Subpage* is the first sub-page displayed. The *Man- and AutoEval Subpages* list the *Evaluations* of the respective types.

4.3.5. Exercise Details View

4.3.6. Evaluator Details View

5. Conclusion and Future Work

5.1. Research Questions

5.2. Future Work

Many features of the implemented application offers are yet rather theoretical, as there does not exist any actual data.

Some designs are too generic still and should be adjusted to the individual use case. An example is the list of evaluations displayed in the EvaluatorDetails view. Right now a common implementation is used for the list here and the list in the TestDataSetDetails sub-pages. The list should not show the evaluators name in each item of the list on the EvaluatorDetails page, but is required to on the TestDataSetDetails sub-pages. Problems like this reveal the prototype nature of the current implementation.

5.3. Conclusion

List of Figures

3.1. Abstracted workflow of the creation of manual and automatic evaluations. Rectangles mark data elements, while ellipses represent process steps. The rectangles marked in green, represent the data available for testing purposes. 12
3.2. Data model, that results from the structural analysis of the available data. 16
3.3. Abstract visualization of the undertaken comparisons on different levels of detail. The info-graphic is intended to clarify the composition of different levels of meta-evaluation result data. The elements marked in color are the elements data will be presented on. 19

4.1. Abstract depiction of the proposed applications components. 22
4.2. Extended diagram representation of the employed data elements. The newly added attributes are marked in blue. 24
4.3. Abstract visualization of the navigation options the frontend supplies to the user. Visualizations are marked in blue for reference in **??**. 31
4.4. Exemplary screenshot of the *Register* view where the TestDataSet subpage is opened. The general structure of the illustrated view is repeated for both the Exercise and Evaluator sub-pages. 32
4.5. Screenshot of the fourth wizard page described in 4.3.3 while creating results from a manual evaluation. The PDF file of the manual evaluation can be seen on the right, while on the left both the form for entering result values and the result list are located. 34

List of Tables

2.1. Quality validation performed in systems for student model evaluation. The column *system* lists a name of the system/method/approach where one was specified. For systems where no calling name was found, the name of the main author is used instead. 8

4.1. Data element classes, listed with the table their instances are stored in and operatives responsible for handling them. 25

Bibliography

[1] N. H. Ali, Z. Shukur, and S. Idris. “A Design of an Assessment System for UML Class Diagram”. In: *2007 International Conference on Computational Science and its Applications (ICCSA 2007)*. 2007 International Conference on Computational Science and its Applications (ICCSA 2007). Aug. 2007, pp. 539–546.

[2] Nilufar Baghaei, Antonija Mitrovic, and Warwick Irwin. “Supporting collaborative learning and problem-solving in a constraint-based CSCL environment for UML class diagrams”. In: *International Journal of Computer-Supported Collaborative Learning* 2.2 (Sept. 1, 2007), pp. 159–190.

[3] Philip-Daniel Beck et al. “COCLAC - Feedback Generation for Combined UML Class and Activity Diagram Modeling Tasks”. In: (), p. 8.

[4] Jan Philip Bernius and Bernd Bruegge. “Toward the Automatic Assessment of Text Exercises”. In: (), p. 4.

[5] Weiyi Bian, Omar Alam, and Jörg Kienzle. *Automated Grading of Class Diagrams*. Sept. 11, 2019.

[6] Weiyi Bian, Omar Alam, and Jörg Kienzle. *Automated Grading of Class Diagrams*. Sept. 11, 2019.

[7] Weiyi Bian, Omar Alam, and Jörg Kienzle. “Is automated grading of models effective? assessing automated grading of class diagrams”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS ’20. New York, NY, USA: Association for Computing Machinery, Oct. 18, 2020, pp. 365–376.

[8] Younes Boubekeur, Gunter Mussbacher, and Shane McIntosh. “Automatic assessment of students’ software models using a simple heuristic and machine learning”. In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS ’20. New York, NY, USA: Association for Computing Machinery, Oct. 16, 2020, pp. 1–10.

[9] Jacob Cohen. “A Coefficient of Agreement for Nominal Scales”. In: *Educational and Psychological Measurement* 20.1 (Apr. 1, 1960), pp. 37–46.

[10] *Computing inter-rater reliability and its variance in the presence of high agreement - Gwet - 2008 - British Journal of Mathematical and Statistical Psychology - Wiley Online Library*. URL: `https://bpspsychub.onlinelibrary.wiley.com/doi/full/10.1348/000711006X126600?%20casa_token=5iF-i8Qxc_MAAAAA%3AyH8jCJPXFxwcVx1sMkF-Y--pUY_OV6JtT_mrpWzuBArpfyrWidFIVq0WoSIEEa%20Z1ftUiczdqLTMkAvR9` (visited on 12/29/2020).

[11] B. Demuth and D. Weigel. “Web Based Software Modeling Exercises in Large-Scale Software Engineering Courses”. In: *2009 22nd Conference on Software Engineering Education and Training* (2009).

[12] James DeVaney et al. “Higher Ed Needs a Long-Term Plan for Virtual Learning”. In: *Harvard Business Review* (May 5, 2020).

[13] Joseph L. Fleiss. “Measuring nominal scale agreement among many raters.” In: *Psychological Bulletin* 76.5 (1971), pp. 378–382.

[14] Markus Hamann. “Automatic Feedback for UML Modeling Exercises as an Extension of INLOOP”. TU Dresden, 2020.

[15] Robert W. Hasker. “UMLGrader: an automated class diagram grader”. In: *Journal of Computing Sciences in Colleges* 27.1 (Oct. 1, 2011), pp. 47–54.

[16] Colin Higgins and Brett Bligh. “Formative computer based assessment in diagram based domains”. In: ACM SIGCSE Bulletin. Vol. 38. June 26, 2006, pp. 98–102.

[17] Colin Higgins, Pavlos Symeonidis, and Athanasios Tsintsifas. “The marking system for CourseMaster”. In: ACM Sigcse Bulletin. Vol. 34. Sept. 1, 2002, pp. 46–50.

[18] Gil Hoggarth and Mike Lockyer. “An automated student diagram assessment system”. In: *ACM SIGCSE Bulletin* 30.3 (Aug. 1, 1998), pp. 122–124.

[19] *INLOOP: interactive learning center for object-oriented programming*. Github. URL: `https://github.com/st-tu-dresden/inloop` (visited on 12/20/2020).

[20] Stephan Krusche and Andreas Seitz. “ArTEMiS: An Automatic Assessment Management System for Interactive Learning”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE ’18: The 49th ACM Technical Symposium on Computer Science Education. Baltimore Maryland USA: ACM, Feb. 21, 2018, pp. 284–289.

[21] Nguyen-Thinh Le. “A Constraint-based Assessment Approach for Free Form Design of Class Diagrams using UML”. In: (Nov. 27, 2020).

[22] David Powers. “Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation”. In: *Mach. Learn. Technol.* 2 (Jan. 1, 2008).

[23] Ferran Prados et al. “An automatic correction tool that can learn”. In: *Proceedings - Frontiers in Education Conference* (Oct. 1, 2011).

[24] Joachim Schramm et al. “Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System”. In: May 25, 2012.

[25] William A. Scott. “Reliability of Content Analysis: The Case of Nominal Scale Coding”. In: *The Public Opinion Quarterly* 19.3 (1955), pp. 321–325.

[26] N. Smith, P. Thomas, and K. Waugh. “Automatic Grading of Free-Form Diagrams with Label Hypernymy”. In: *2013 Learning and Teaching in Computing and Engineering*. 2013 Learning and Teaching in Computing and Engineering. Mar. 2013, pp. 136–142.

[27] Rúben Sousa and José Leal. “A Structural Approach to Assess Graph-Based Exercises”. In: June 18, 2015, pp. 182–193.

[28] Michael Striewe and Michael Goedicke. "Automated checks on UML diagrams". In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ITiCSE '11. New York, NY, USA: Association for Computing Machinery, June 27, 2011, pp. 38–42.

[29] *The COVID-19 pandemic has changed education forever. This is how*. World Economic Forum. URL: `https://www.weforum.org/agenda/2020/04/coronavirus-education-global-covid19-online-digital-learning/` (visited on 12/03/2020).

[30] P. Thomas, N. Smith, and Kevin G. Waugh. *Automatic assessment of sequence diagrams*. undefined. 2008. URL: `/paper/Automatic-assessment-of-sequence-diagrams-Thomas-Smith/%20ef570840cbb182d6e8f861ced321992e20b94f93` (visited on 11/27/2020).

[31] Pete Thomas, Neil Smith, and Kevin Waugh. "Automatically assessing graph-based diagrams". In: *Learning Media and Technology* 33 (Sept. 1, 2008).

[32] Christos Tselonis, John Sargeant, and Mary McGee Wood. "Diagram matching for human-computer collaborative assessment". In: (Jan. 1, 2005).

[33] *Zahl der Studierenden erreicht im Wintersemester 2019/2020 neuen Höchststand*. Statistisches Bundesamt. URL: `https://www.destatis.de/DE/Presse/Pressemitteilungen/2019/11/PD19_453_213.html` (visited on 11/24/2020).

A. Weitere Latex-Dokumentation

Statement of authorship

I hereby certify that I have authored this Bachelor Thesis entitled *Concepts of Quality Assurance for the Constraint-based E-Assessment of Models* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 5th February 2021

Paul Erlenwein