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: 2018

## 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

## 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

# Contents

# 1. Introduction

## 1.1. Motivation

Modelling is an essential part of software engineering and therefore an essential part of any degree programme that qualifies students to work in this field. At TU Dresden, the basics of software modelling are taught as a compulsory module in all computer science bachelor's and diploma degree studies, as it is a skill that no computer scientist can afford to dispense with.

Modelling poses some specific challenges for teaching staff, which are inherent to the nature of classical model representations such as UML, which is the modelling language taught by the aforementioned course. A modelling task, set by the teaching staff for its students can, depending on the exact wording of the task, be solved optimally in several ways. The optimal solution for any modelling task may be ill-defined. This signifies that for most models, once an optimal solution has been found, it cannot be conclusively proven that no other equally optimal solution exists. As is to be expected, this and a number of other factors makes reviewing student solutions a rather complex and time consuming task. *Time consuming* is the keyword to focus on.

The number of students is rising almost constantly [8]. Still: There are not enough IT-specialists in Germany [11]. From these two pieces of information, one obvious conclusion emerges: more students per semester will have to be taught at universities in the future. Whether this will be possible will depend to a large extent on the ability of universities to cope with an increased number of students while maintaining the same standards of breadth and quality of the education they offer. It is therefore necessary to reduce time-consuming tasks to enable teaching personnel to manage the increased workload. Time-consuming tasks such as reviewing student solutions to modelling tasks.

To address this and a host of other issues, an automatic assessment system for student models is being developed at TU Dresden. The project is an extension to the already existing INLOOP software [18]. INLOOP (*INteractive Learning center for Object-Oriented Programming* is employed to evaluate student solutions to programming tasks students have to solve for the same compulsory course. The tool, which is currently being developed to extend the capabilities of the platform by evaluating solutions students submit for modelling exercises, is called INLOOM [13]. INLOOM is an abbreviation for *INteractive Learning center for Object-Oriented Modelling*. INLOOM evaluates and grades student solutions to modelling exercises using a constraint-based approach.

Being assisted by such a tool will reduce the workload of the teaching personnel and provide students with an opportunity to gain additional experience. However, for any tool that is intended to be employed in a teaching environment it is of utmost importance to be tested thoroughly, since all students must be treated equally. Cases were students suffer unfair treatment due to errors in the software must be avoided. Only if the employed software meets the highest standards it will be possible to fully *lean on it* and include it in the common teaching workflow. To guarantee the quality of INLOOMs performance, it is necessary not only to validate its results, to make sure it provides a fair environment for everyone involved, but also to maintain this state of the system over an extended period of time. This thesis will investigate how such continued testing of INLOOMs quality can be facilitated.

## 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 [13]. 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 this thesis will aim to answer the following research questions:

**RQ1** Which values can be extracted from the manual and automatic evaluations?
**RQ2** How can the *quality* of INLOOMs results be quantified?
**RQ3** What methodologies are employed by existing automatic model assessment systems to validate their results?
**RQ4** How can the developer/tutor/lecturer best be supported in collecting and pre-processing data on the evaluation whose quality they intend to assess?
**RQ5** How can test results be presented to greatest effect?

The answer to research question 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 RQ2 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 quantified on a conclusive scale. Research question RQ2 aims to identify such a scale. Answering research question RQ3 will be approached by performing a literature survey concerning the quality validation methodologies of existing software solutions and automatic grading methods. The goal is to analyze existing methods to be able to build on them.

To evaluate evaluations, data on them will have to be collected. Any kind of data collection entails effort that has to be reduced as much as possible to add to the usefulness of the tool that this work intends to result in. Research question RQ4 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 the intended *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 answers to research questions RQ2 and RQ3.

# 2. Literature Research

## 2.1. INLOOM

This work aims to validate the quality of the results the INLOOM software [13] 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[18] 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 to, very hard, if not impossible depending on the exact wording of the exercise. 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

Before going any further into how quality testing can be performed, 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. The DIN ISO standard 9000:2015 [24] defines quality as follows: "[...] The quality of an organization's products and services is determined by the ability to satisfy customers and the intended and unintended impact on relevant interested parties. The quality of products and services includes not only their intended function and performance, but also their perceived value and benefit to the customer."

Quality is a measure between perfect and really bad and can, by definition, not be quantified objectively. What is either good or bad, can depend on any number of arbitrary factors and is ultimately defined by the person performing the quality assessment. This thesis will in accordance with the cited standard, equate quality with the satisfaction of the customer. The customer in this case is the person who employs INLOOM in a teaching environment. The students taught are an additional group of stakeholders. This thesis is based on the assumption, that both teaching personnel and students require fairness and correctness above all else. INLOOMs *quality is high* or *it does a good job*, if it evaluates the student-created models *fairly* and *correctly*. If it awards points *exactly where due*.

INLOOMs *quality is low* or *it does a good job* if, for example, it does not recognize valid inputs as such, finds errors where there are none or awards points were none are due. This definition of quality has the inherent problem that it depends on the question whether or not points *should be awarded* for a feature of the student solution. Since no objectively correct evaluation of a student solution exists, this question cannot be answered conclusively. To still be able to ascertain the quality of an evaluation, it is thus necessary to establish a reference evaluation that is assumed to be optimal.

## 2.3. Quality-Testing of existing Model Evaluation Systems

In this section, the results of a literature research, into concepts for testing the quality of evaluations, automatic grading systems produce are presented. Starting point for the research, was a collection of such systems, referenced in [13]. 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 [6], that was not listed. The literature research was performed using popular catalogs of scientific literature like Google Scholar and IEEE.

The listing in [13] 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. A brief inspection of the listed systems brought found 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 result in some kind of evaluation, expressed by a number of output values. 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 (see Section 2.2) validation performed in systems for student model evaluation. For systems where no calling name was found, the name of the main author is used in the system column instead.

| *System* | *Source(s)* | Quality Validation Performed |
|---|---|---|
| INLOOM | [13] | Grade Quotient Strategy |
| Bian | [5, 6] | Grade Quotient Strategy |
| Schramm | [25] | Didactic Evaluation |
| UML GRADER | [14] | Grade Quotient Strategy |
| Striewe | [29] | Grade Quotient Strategy |
| Demuth | [10] | No Publication |
| Collect-UML | [2] | Didactic Evaluation |
| Le | [21] | Didactic Evaluation |
| CourseMaster | [15, 16] | No Publication |
| Artemis | [4, 20] | Grade Quotient Strategy |
| COCLAC | [3] | No Publication |
| Sousa | [28] | No Publication |
| Smith & Thomas | [27, 30, 31] | Grade Quotient Strategy |
| Prados | [23] | No Publication |
| 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, 10, 16, 23, 28]. These are collected under the term *No Publication*. Seven of the publications describe a validation that is based

on a comparison of the grade equivalent of the respective method [4, 5, 13, 14, 27, 29, 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 [13]. For the rest of the systems, the authors detail strategies for performing an evaluation, regarding the didactic use of their respective tools [2, 21, 25] (*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, 17, 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.

### 2.3.1. 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, 13, 14, 27, 29, 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* (2.1) 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 (2.2). Another way to represent the difference between the two is the absolute point difference percentage $\Delta$ (2.3). 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 [7]. 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.

### 2.3.2. Grace Points and Clean Scores

The literature [13, 27, 30, 31] points out that the ratings of different reviewers can differ significantly. [13] 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.

### 2.3.3. Inter-Rater-Reliability

Statistics like Cohens Kappa[9], Scott's Pi[26], Fleiss Kappa[12] and Gwets AC1-Statistic [19] 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.

### 2.3.4. 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 *Quality Inspection Tool* for INLOOM. A software that enables inspecting and validating the evaluations INLOOM [13] produces. Creating an appropriate design will be approached by the following means. First, the requirements to be met by the designed application are compiled. In a second step, the data available as a basis for the evaluation to be carried out, will be examined and a possible data model for the QIT will be derived. In the last step, based on the first two, the design of a meta-evaluation, will be presented.

## 3.1. Software Requirements Collection

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. To achieve this, the intended software will have to meet a number of requirements. This sections subsections will list the identified high-level software requirements and limitations (*SR*) and explain the rationale behind each of them.

### 3.1.1. General Requirements

There are two requirements placed on the proposed application that are self-evident at this point. First: QIT must be able to perform an evaluation of INLOOMs output. Second: It must be able to present the results of this evaluation to the user.

SR1 The software must be able to automatically perform a comprehensive evaluation of 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.

INLOOM currently is a work in progress. This implies, that is very well possible that at least some of the currently employed standards and formats will be subject to change in the future. This makes designing the intended application to be as flexible as possible an important requirement.

SR3 The design of the software must be as flexible as possible to allow adapting it to extended or reduced use cases in the future.

### 3.1.2. Data-Driven Requirements

A very significant criterion for the design presented here is the availability of test data. The literature research [4, 5, 13, 14, 27, 29, 32] showed that the most commonly employed method of validating automatically generated evaluations is to compare them with manually created ones. Manual evaluations are treated as 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. To employ a comparison-based strategy, however, there needs to be a sufficient amount of student solutions that were evaluated by both a manual evaluator and INLOOM.

Faking up student solutions to address this issue was considered. Forging student solutions for test purposes could be done in several ways. For example, by randomly generating UML models that use the same terminology as the exercise at hand. It is also conceivable to assemble student solutions from predefined blocks or, in the simplest case, to create them manually. All of these approaches come with some downsides, however, because of which they were finally dismissed. The first two strategies have in common that they would require an implementation effort that could very well exceed the scope of this thesis. The remaining viable approach is to manually forge student solutions for testing purposes, which would be significantly less economical than using student solutions created by real students, given the amount of work involved. Especially when considering that randomly generated solutions would have to be evaluated manually to use the evaluation data in the intended comparison, forging student solutions for testing purposes becomes unthinkable. Reviewing additional randomly generated student solutions seems wasteful, when actual submissions have to be graded anyways.

Using solutions created by real students is not only the most realistic test case but also reduces the required effort to transforming the manual evaluations into a format that allows comparing them to the automatically created ones. The intended software will have to include a facility appropriate for quickly collecting data on manual evaluations and another for extracting all relevant data from the XML files INLOOM stores its results in. QIT must be able to store and provide all evaluation data once collected.

**SR4** QIT must include a UI facility that allows the user to enter data on manual evaluations, with as much ease as possible.

**SR5** QIT must provide the means to extract data on the evaluation INLOOM performs from the output XML files it generates.

**SR6** QIT must employ a data structure that is able to hold information collected on automatic and manual evaluations and enables comparing the two.

For one student solution, there can exist multiple automatic and manual evaluations. Multiple manual evaluators or different iterations of INLOOM, for example, using different constraint sets, can create evaluations for the same student solution. The literature [13, 27, 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 evaluations of the same student solution allows for easy calculation of meta-statistics like *clean scores* described in Section 2.3.2.

**SR7** The software must be able to persist multiple manual and automatic evaluation for each student solution.

### 3.1.3. Imposed Limits

All tests the application will be able to perform are limited to the data extracted from INLOOMs evaluations and collected on the manually created ones. To increase the level of detail at which the evaluated evaluations can be analysed, a larger amount of test data is desirable. The extent to which data on the automatic evaluation is available is solely dependant on the output INLOOM generates. Although it would probably be feasible to adjust the structure and content of the output generated by INLOOM, this thesis is based on the working hypothesis that the outputs are immutable.

L1 All tests must be performed using the data available, rather than data one *could* collect during INLOOMs evaluation process.

Another limitation found necessary to limit the scope of the implementation, is allowing only manual and automatic evaluations that have been evaluated using the same grading scheme, to be used in the performed evaluation. For two evaluations that used differing grading schemes to still be compared in a meaningful way, one of the evaluations would have to be *translated* to another grading scheme. This would introduce another most probably workload intensive step into the process. A step that is deemed unnecessary, since manual evaluations using the grading scheme also employed by INLOOM are available for testing.

L2 The application requires manual and automatic evaluations as inputs that were created using the same grading scheme.

### 3.1.4. Summary

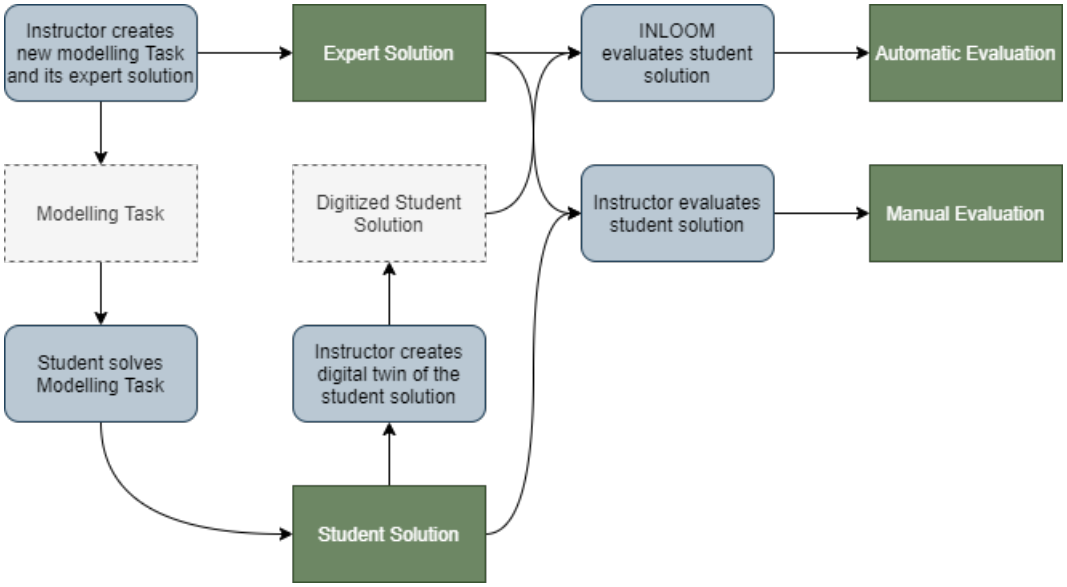
The requirements and limits placed on the proposed application can thus be summarized:

SR1 The software must be able to automatically perform a comprehensive evaluation of 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.
SR3 The design of the software must be as flexible as possible to allow adapting it to extended or reduced use cases in the future.
SR4 QIT must include a UI facility that allows the user to enter data on manual evaluations, with as much ease as possible.
SR5 QIT must provide the means to extract data on the evaluation INLOOM performs from the output XML files it generates.
SR6 QIT must employ a data structure that is able to hold information collected on automatic and manual evaluations and enables comparing the two.
SR7 The software must be able to persist multiple manual and automatic evaluation for each student solution.
L1 All tests must be performed using the data available, rather than data one *could* collect during INLOOMs evaluation process.
L2 The application requires manual and automatic evaluations as inputs that were created using the same grading scheme.

## 3.2. Available Data

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. Figure 3.1 shows the abstracted workflow of the current process employed to validate INLOOMs evaluations. The green rectangles represent accessible files that result from or are created during this process. The rectangles colored light grey and drawn with a dotted line also represent file like entities. These are created during the collection process, but are either not persisted or not available for testing purposes. Lastly, the rectangles with a blue background and using rounded corners, represent steps of the process.

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.




As was previously stated the two sources of data QIT will employ to perform an evaluation of INLOOMs result quality on, are pairs of manual and automatic evaluations of solutions students submitted for a given modelling exercise. What data can be extracted from both of them will be discussed in the following subsections.

### 3.2.1. INLOOM Result Files

The result INLOOM generates is one of the most central item of the entire design. To use the evaluation data INLOOM stores in XML files, these XML files have to be read and the contained data transformed into the format required by SR6. Each of INLOOMs result 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 [13] and is now described by the 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 with. "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 [13]. 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 and 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. As required by SR5, QIT must provide the means to adapt the data found in the XML files of the described form, into a format that allows easy comparison of evaluations, as required by SR6.

### 3.2.2. Manually Evaluated Student Solutions

The second part of the test data are manual evaluations of student solutions. The literature [4, 5, 13, 14, 27, 29, 32] agrees that the manual evaluations of student submissions are the best evaluations known for the specific solution. Therefore, they are the most probable measure of quality one can apply to INLOOM. Such a comparison based evaluation strategy assumes that the automatic evaluation's quality can be assessed based on the amount of deviation between the compared evaluations. The more *like* the manual evaluation the automatic one is, the better its quality. To map a given amount of deviation into terms like *good* or *bad*, confidence intervals will have to be defined. However, to define one on the basis of what little data is available seems premature. The design presented here is limited to the calculation and presentation of the deviation on a normalised scale. The definition of confidence intervals on the scale is left to the user of the designed application.

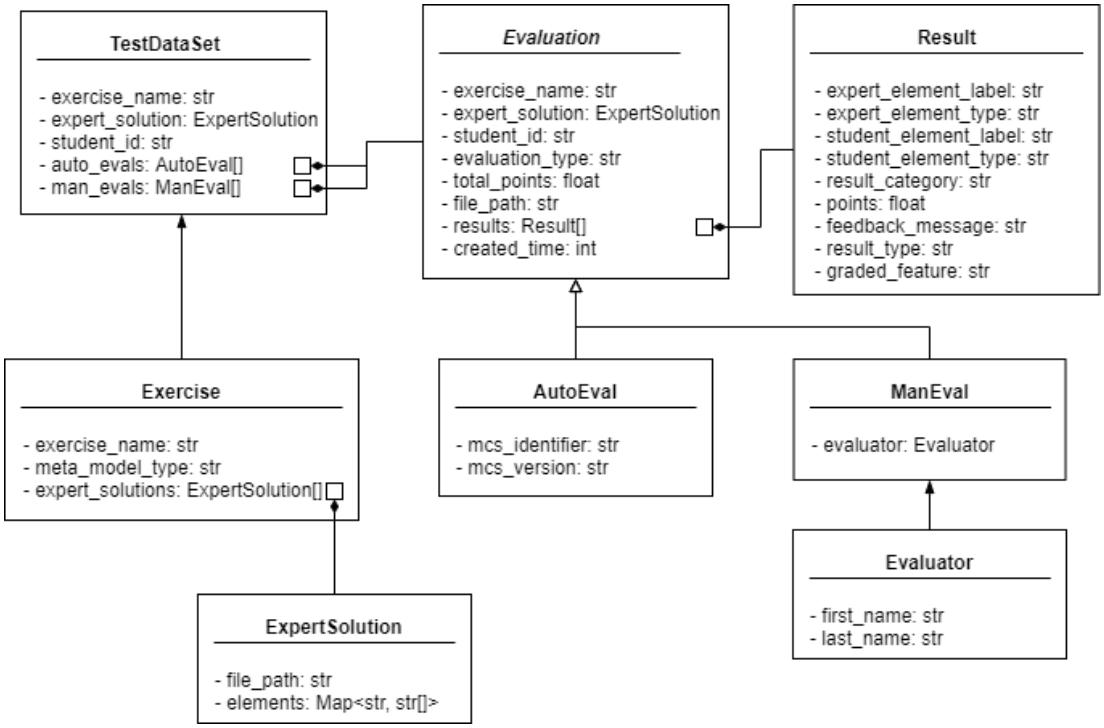
Thirty already graded pen-and-paper student solutions to exam tasks were digitally reproduced by [13], to be evaluated again using INLOOM. Of these thirty solutions, ten are solutions to the exam tasks of the summer term exams 2017, 2018, and 2019, respectively. The manual evaluations included in this test data set consist of handwritten annotations captured in scans of pen and paper student solutions. Only these *analog* versions of the manual evaluations exist. Of the available evaluations, only those created to grade submissions for the summer term exam 2019 used a uniform grading scheme in both the manual and automatic evaluation. In accordance with L2, these will provide test cases for QIT. It will be assumed that the type and form of data, collected on the evaluations of student solutions submitted for the summer term exam 2019, is representative for the usual data basis for the quality analysis to be carried out.

To compare these manual evaluations with the ones automatically generated by INLOOM, they need to be digitized. The annotations the manual evaluations consist of are mostly check marks and points awarded for elements of the model. Which feature an annotation references is indicated by its position in the student solution. Due to the format and the fact that the student solutions were stored as black and white scans, it is unlikely that the evaluation data can be extracted automatically. In any case, extracting the required information directly from the PDFs would most definitely exceed the scope of this thesis. The design will be based on the working hypothesis that data on manual evaluations has to be entered manually.

### 3.2.3. Data Driven Data Structure

There are some attributes automatic and manual evaluations clearly have in common. Others they share more obliquely and some transformation is required before they can be assumed present in both. This section will analyse the test data available on both sides of the comparison and derive a data structure in order to meet the posed requirement SR6. Figure 3.2 shows the analysis model that resulted from the efforts described in this section.

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




The central entity QIT will have to handle is the *Evaluation*. Each *Evaluation* is created for exactly one student solution. A student solution can be identified based on the student who created it and the exercise it is intended to solve. Student solutions will not be modelled by the application, since they are solely interesting in the context of an *Evaluations*. Each *Evaluation* can only ever be created by one *Evaluator* and using one *ExpertSolution* for reference.

An *ExpertSolution* models an optimal solution created by an expert, that is employed to compare student models against. The essential part of an *ExpertSolution* is a collection of elements and features the student solution needs to include to be deemed correct. What elements a solution needs to include depends on the actual optimal solution. An *ExpertSolution* can only exist in the context of an *Exercise*. An *Exercise* as the name suggests models an exercise students are required to solve. Each *Exercise* needs to reference at least one *ExpertSolution*.

The attributes required to identify an evaluation can easily be extracted from both the manual and automatic evaluation data sources. An evaluation can be uniquely identified based on the student who created the evaluated solution, the *Exercise* the submitted solution tries to solve, the *ExpertSolution* it was compared against and the type of the *Evaluation*, which is either automatic or manual. Additionally, to distinguish manual *Evaluations*, the *Evaluator* who created the evaluation and the time it was created can be used. To do the same for automatic evaluations one can utilize the *MCSVersion* and *MCSIdentifier*. However, since these attributes do not provide any information on how INLOOM was configured when the evaluation was created. For this reason, why time the evaluation was registered in QITs database is additionally used as an identifying attribute. The practice is based on the assumption that the frequency with which new evaluations for the same student solutions are registered in the system will not be high enough to make easily distinguishing evaluations based on their creation time impossible. Manual and automatic evaluations are modelled using the classes *ManEval* and *AutoEval* using the common abstract parent class *Evaluation*. *Evaluations* are collected in *TestDataSets*. Each *TestDataSet* persists the data available in the context of one student solution. It holds both a list of *Auto-* and *ManEvals*. A *TestDataSet* is identified by the student solution it persists combined with the *ExpertSolution* that was employed for the evaluation of the student solution. As was described in Section 3.2.2, for each element the *ExpertSolution* requires, INLOOM adds a "CResult" to the list of "Results", it persists in its output. 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 the used 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. Both the individual "CResult" elements of INLOOMs output files and the annotations found in manual evaluations are modelled as *Results* by QIT. The *Result* class is closely related to the "CResult" XML elements.

### 3.2.4. Digitizing Manual Evaluations

The need to digitize the manual evaluations using the facility required by SR4, before being able to compare them automatically to the evaluations INLOOM produces, means a substantial overhead for the testing process. This effort should be reduced as much as possible to make entering manual evaluations easier and more economical. 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.

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.

Most interesting, however, is easing the inputs required in the third step of the process. The third is the only step that will have to be repeated multiple times. In each repetition, six attributes need to be entered, 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 and the student element type are not generic. The rest of the attributes are limited to a number of options. The options for the result category as well as the points that can be awarded to a feature are defined by the employed expert-solution. Both are limited to a number of options by the INLOOMs implementation. Adding a new *Result* can thus be reduced to entering two values and selecting the rest from predefined options. Unless it should become possible to automatically transform manual evaluations from an analog format into one fit for automatic comparison, entering the manual evaluation data manually will remain necessary. The described input process will have to be modelled by the implementation of the application.

## 3.3. Comparison-based Meta-Evaluation

The literature research showed quite clearly (see 2.3.1), that a *Grade Quotient Strategy* is the most suitable approach for evaluating the similarity of two evaluations. INLOOM is a constraint-based system and its results should 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, because assuming a nominal categorization would disregard a big amount of the available data. 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 the evaluated student solution.

As was described in Section 2.3.3, 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 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 kinds 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 assigned by INLOOM and to repeat the evaluation. 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.

"Informedness and Markedness", which were briefly described in Section 2.3.4, were disregarded for similar reasons as IRR. The only categorization INLOOM performs that the concept could be used to evaluate, is assigning the result category flags described in Section 3.2.2. It seems unlikely, however, that it will be possible to extract information on the manual evaluations on a level of detail, sufficient to examine the assignment of individual category flags.

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 constraint 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. Meta-Evaluation Detail Levels

A comparison-based meta-evaluation of a manual and automatic evaluation, collected and persisted as described, can happen on a number of different levels of detail, or rather, on number of different scopes. On each of these detail levels, a number of categories have to be considered for comparison. Figure 3.3 is intended to convey the scope of these levels of detail. The elements marked in color represent data collections deemed worth presenting to the user.

The highest level of detail a meta-evaluation can take place on, is the comparison of a manual and automatic Evaluation. By combining the evaluations contained in a *TestDataSet* the next lower level of detail is gained. By calculating the mean of every KPI over all *ManEvals* stored in a *TestDataSet* data on an artificial meta-evaluation of a *average manual evaluation* is created. This *AvgManEvalStats* can be used to compare INLOOMs performance between different *TestDataSets*.

The lowest level of detail deemed 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.

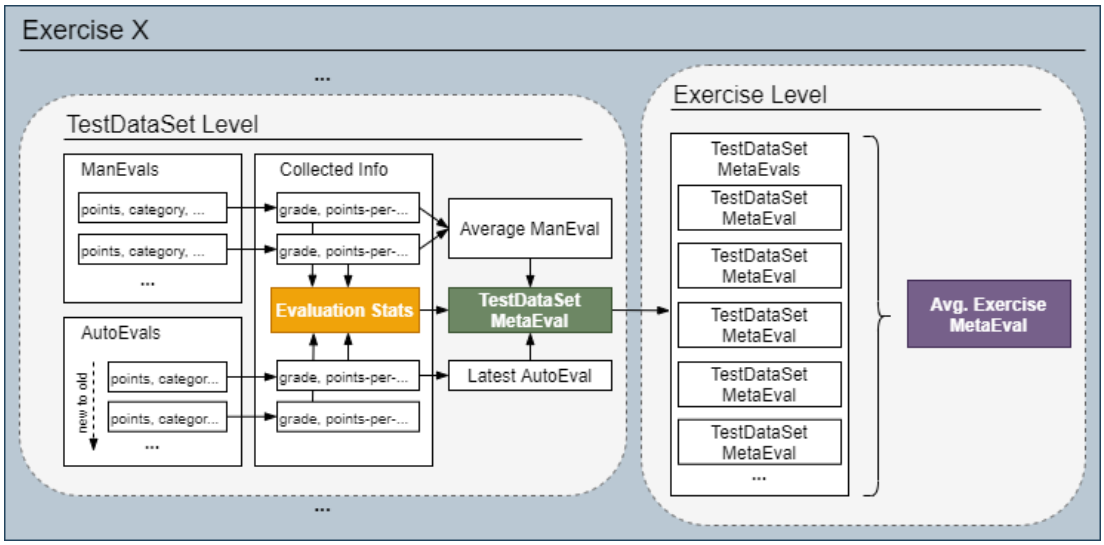



Figure 3.3. Abstract visualization of the levels of detail QIT will perform meta-evaluations on.

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.

### 3.3.2. Meta-Evaluation Categories

As was stated earlier, the main feature that will be compared between evaluations is the final 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. QIT will most visibly present the comparison of the two grades, using a point difference percentage $\Delta$ (see 2.3.1). Additionally, a number of KPIs of the meta-evaluated comparison of two *Evaluations* are considered worthy of being calculated and presented for assessment purposes. The simplest among those are the absolute point difference and a quotient of the grades of the compared evaluations, as it was described in Section 2.3.1.

The rest of the intended meta-evaluations on the *Evaluation* level are more complex. There are three such. Although more elaborate evaluations might prove interesting in the future, the application proposed here will only perform rather basic evaluations because they were found to be best suited to gain and convey insight on INLOOMs current performance. As was initially stated, a major problem with a meta-evaluation based purely on the final grade awarded, is that the grade contains no information about how it was determined. A scenario in which the grades of a compared *Man-* and *AutoEval* are equal whilst the points the grades resulted from were awarded for completely different elements of the student solution, is easily imaginable. To remedy this and to enable the user to quickly identify not only the existence of discrepancies between the compared evaluations but also their origin, 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*.

A second evaluation, performed for each comparison of *Evaluations* 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. It is similar to another meta-evaluation called *category-count*, which will, as the name suggests count the number of times a category flag was assigned by an evaluation. The last intended meta-evaluation, called *category-combinations* 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. This meta-evaluation is somewhat special, since it is the only one of the meta-evaluations performed on an individual comparison that can not be averaged over all *ManEvals* of a given *TestDataSet*.

The evaluations of different manual evaluators may differ significantly (see Section 2.3.2). This complicates the evaluation of INLOOMs results as all manual evaluations are considered equally optimal. The Design of QIT addresses the issue by calculating the characteristics of a theoretical average *ManEval*. Such a construct incorporates the findings of all manual evaluations equally. It represents the mathematical compromise between several evaluations and is thus a fair combination of the assessments of all evaluators who have performed a manual evaluation of the student's solution. As was described in Section 3.3.1 these *AvgManEvalStats* are the primarily inspected element on the next lower level of detail - the scope of an entire *TestDataSet*. Through creating the mentioned *AvgManEvalStats*, by averaging the results of each meta-evaluation over all *ManEvals* contained in a *TestDataSet*, meta-evaluation values that are representative for the entire *TestDataSet* are gained. The *AvgManEvalStats* will contain a value for each: The point difference percentage, the absolute point difference and the grade quotient, as well as result values for the meta-evaluations called *points-per-expert-element-type* and *points-per-result-category*.

For the lowest level of detail, *TestDataSet* meta-evaluation will be *grouped* by *Evaluator*, *Exercise*, *Expert Solution*, or a combination of these keys. By calculating average values for from the meta-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. As was stated in Section 3.3.1 QIT will group *TestDataSet* meta-evaluations by their *Exercise* and, in a separate collection operation, by their *Evaluator*. Since comparing the more complex meta-evaluations will not work on an exercise level, the meta-evaluations on the scope of an *Exercise* or an *Evaluator* will only show comparisons of the meta-evaluations that resulted in a single characteristic value for each *TestDataSet*.

### 3.3.3. Visualizing Meta-Evaluation Results

Visualizations are an effective way of presenting data. For visualizing the results of the meta-evaluations, the proposed software is intended to perform, appropriate visualizations have to be found. Those will be employed to display the data on each of the described levels of detail for each of the meta-evaluation categories listed.

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 available. Individual values will therefore be presented to the user as they are.

Multiple values are available when grouping test data sets. Grouping as was described in 3.3.2, happens using the *Evaluator* or the *Exercise* as keys to collect *TestDataSets*, on. 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*.

### 3.3.4. Meta-Evaluation Summary

This subsection is intended to summarize the design decisions made in Section 3.3. Table 3.1 lists the intended meta-evaluations in combination with the level of detail they are intended to be performed on and the visualization deemed adequate to present its results. In compliance with SR3, it will later be possible to extend QIT by any meta-evaluation feasible based on the available data.

| Detail-Level | Category | Visualization |
|---|---|---|
| Evaluation | Point Difference Percentage | Individual Value |
| Evaluation | Absolute Point Difference | Individual Value |
| Evaluation | Grade Quotient | Individual Value |
| Evaluation | points-per-expert-element-type | Comparative Bar Chart |
| Evaluation | points-per-result-category | Comparative Bar Chart |
| Evaluation | category-count | Comparative Bar Chart |
| Evaluation | category-combinations | List View of Expert-Elements |
| TestDataSet | Point Difference Percentage | Individual Value |
| TestDataSet | Absolute Point Difference | Individual Value |
| TestDataSet | Grade Quotient | Individual Value |
| TestDataSet | points-per-expert-element-type | Comparative Bar Chart |
| TestDataSet | points-per-result-category | Comparative Bar Chart |
| TestDataSet | category-count | Comparative Bar Chart |
| Exercise | Grade Comparison | Comparative Bar Chart |
| Exercise | Grade Quotients | Common Bar Chart |
| Exercise | Total Point Differences | Common Bar Chart |
| Exercise | Point Difference Percentages | Common Bar Chart |
| Evaluator | Grade Comparison | Comparative Bar Chart |

**Table 3.1.** Table of intended Meta-Evaluations.

## 3.4. Summary

This section summarizes whether the design presented in Sections 3.2 and 3.3, meets the software requirements identified and listed in Section 3.1.

SR1 The software must be able to automatically perform a comprehensive evaluation of 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 and introduced in Section 2.3.1.

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. A collection of the intended meta-evaluations and visualizations can be found in Table 3.1.

SR3 The design of the software must be as flexible as possible to allow adapting it to extended or reduced use cases 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. The need for data collected by QIT to be accessible to other software in the future was taken into account at every step of the design.

SR4 QIT must include a UI facility that allows the user to enter data on manual evaluations, with as much ease as possible.
**Result:** The software will model the process required to digitize manual evaluations, as was described in Section 3.2.4.

SR5 QIT must provide the means to extract data on the evaluation INLOOM performs from the output XML files it generates.
**Result:** The software will model the process required o digitize manual evaluations, as was described in Section 3.2.4.

SR6 QIT 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. The data structure that resulted from the performed analysis was described in Section 3.2.3.

SR7 The software must be able to persist multiple manual and automatic evaluation for each student solution.
**Result:** The same data structure used to satisfy SR6 is employed for this purpose.

L1 All tests must be performed using the data available, rather than data one *could* collect during INLOOMs evaluation process.
**Result:** The analysis of the available data was thus limited. Keeping within the boundaries of this limitation did not result in any difficulties.

L2 The application requires manual and automatic evaluations as inputs that were created using the same grading scheme.
**Result:** Data for testing purposes that complies with this limitation is available. As described in Section 3.2.1 and Section 3.2.2 it will be possible to extract the required values from the files they are stored in.

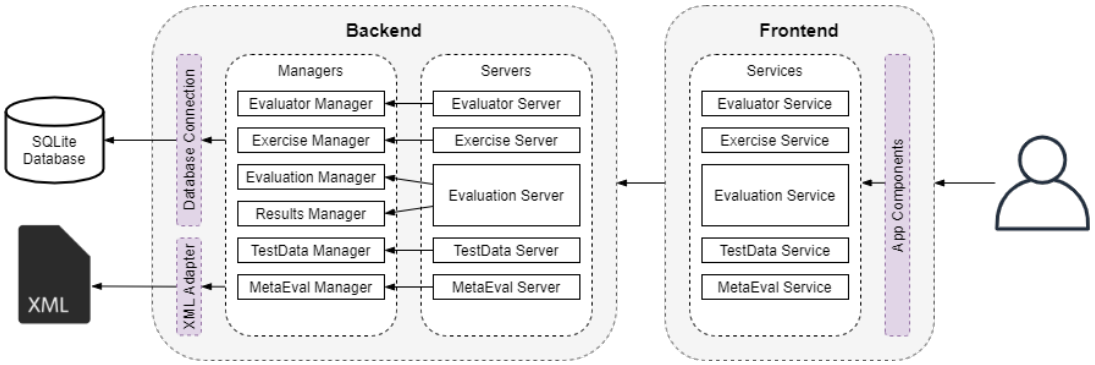
# 4. Implementation

This section intends to present the implementation of QIT. An application for inspecting, evaluating and validating the evaluations 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. 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 QIT uses is stored in a SQLite[1] file. A python Flask[2] app provides a simple URL API to enable access to the stored data. The front-end application, the user accesses, is implemented using Angular[3]. For creating visualization the popular JavaScript framework d3.js[4] will be employed. Figure 4.1 shows the intended parts of the application to be implemented. The individual parts will be discussed in their own subsections.

Figure 4.1. Abstract depiction of the proposed applications parts.




[1] https://www.sqlite.org/ - visited on 27. Dec. 2020
[2] https://flask.palletsprojects.com/ - visited on 27. Dec. 2020
[3] https://angular.io/ - visited on 27. Dec. 2020
[4] https://d3js.org/ - visited on 27. Dec. 2020

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. 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

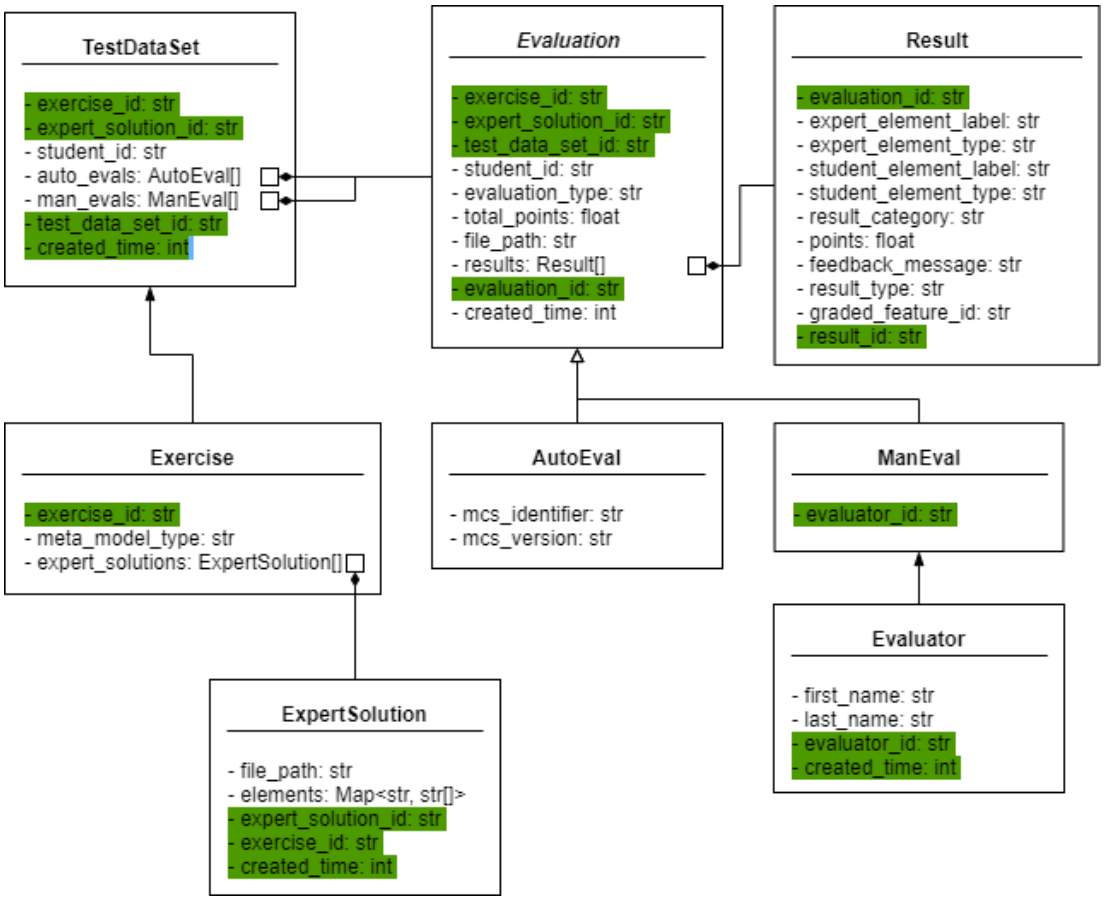



Figure 4.2. Diagram representation of the employed data structure.

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 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.

The managers consist of a collection of methods to access 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. The manager and the data element class are thus 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.

For reading the INLOOM result files, a XML adapter service is provided. The database is passive and *managed by* 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 is described in Section 3.2.3. The model presented there is extended by id and timestamp attributes. Figure 4.2 shows a diagrammatic representation of the resulting model. The newly added attributes are marked in green. A timestamp in most cases is only added for convenience. The only class that strictly requires it is the *Evaluation*. The id attributes were added to ease managing data in the SQLite database file QIT connects to. The problem which is addressed by adding these ids, is that most of the identifiers supplied by INLOOMs result files allow entering an arbitrary string. The use of such identifiers would introduce a large number of edge cases and complications that would have to be considered in the implementation, which unnecessarily inflates the required effort. Instead, unique ids are generated and used as primary keys for the database. The only noteworthy exception to this course of action is the *Exercise*. The name of an *Exericse* is used to uniquely identify it. 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.

### 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. However, 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 all data common to all expert solutions for that exercise. 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 QIT. 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 based on a very similar approach. Instead of storing the collected data as an *Evaluation*, however, the data is processed further, to create an *ExpertSolution*. The *ExpertSolution* shares many of the values of the *Evaluation*, storing each result, however, 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 anticipated elements. 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, rather then requiring them to enter the element label manually for each considered element, a more verbose label is required. To obtain such a label, the feedback messages provided by INLOOMs results 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 quality of INLOOMs results. 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. The specific values QIT calculates, were described in Section 3.3. 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 results of the data collection tasks are processed further to perform the meta-evaluaions described in Section 3.3. The final results are collected in a Python dictionary and sent to the frontend encoded as JSON. The process to create the meta-evaluations can be separated into four distinct parts. In the first step, all evaluations the *TestDataSet* references are collected. Listing 4.2.3 shows an excerpt from the *get_tds_meta_eval* function. This function is used to create a meta-evaluation of a *TestDataSet* which is specified by its id.

Listing 4.1 Excerpt from *get_tds_meta_eval*, the main meta-evaluation function. It 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."""

    # 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'] = {}

    # Collect stats for AutoEvals
    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)

    # Collect stats for ManEvals
    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)

    ...
```

In a second step, for each of the thus collected Man- and AutoEvals, a number of data collection tasks are 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. The collection tasks mentioned, result in separate evaluation specific sub-mappings.

Listing 4.2 Python implementation to commence a number of data collection tasks.

```
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,
        self._count_category_flags,
        partial(self._calculate_grade, expert_solution)
    ]:
        key, value = _func(evaluation)
        stats[key] = value

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

    return 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 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 mapping. The calculated averages are used as *TestDataSet* level meta-evaluation stats as was described in Section 3.3.1.

Listing 4.3 Implementation employed to create *AvgManEvalStats*.

```
@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()))

    # Create a mapping to hold the functions results in
    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
```

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 AutoEval from the available ones
    # Sort the Evals by time inc. and get the last item
    auto_evals: List[AutoEval] = sorted(
        auto_evals, key=lambda e: e.created_time)
    latest_auto_eval: Evaluation = auto_evals[-1]

    # 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

    # Adding 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

    # Adding stats on an average man eval
    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'][self.AVG_MAN_EVAL_KEY] = avg_man_eval_stats

    # Calculate Comparison Stats
    tds_meta_eval['comparison-stats'] = self.calculate_comparison_stats(
        auto_evals, man_evals, avg_man_total_points, latest_auto_total_points, expert_solution)

    # 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
```

In a final step, for each combination of a *ManEval* and a *AutoEval* that was found in the *TestDataSet* comparison specific values are calculated. Those, namely, are, the point difference percentage, the total point difference and the grade quotient. 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.

It should be mentioned, that any colorizations the application uses in as an indicator is based on confidence intervals supplied by the maintainer of INLOOM. The specific colors used did play a very minor role during the implementation. It will be easily possible to change the colors used, by adapting a color mapping in the service that supplies them. As the exact color values were the result of a very late decision, this description of QITs frontend will not address them.

### 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 app-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 number 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 occur 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.

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. 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 in the application database.

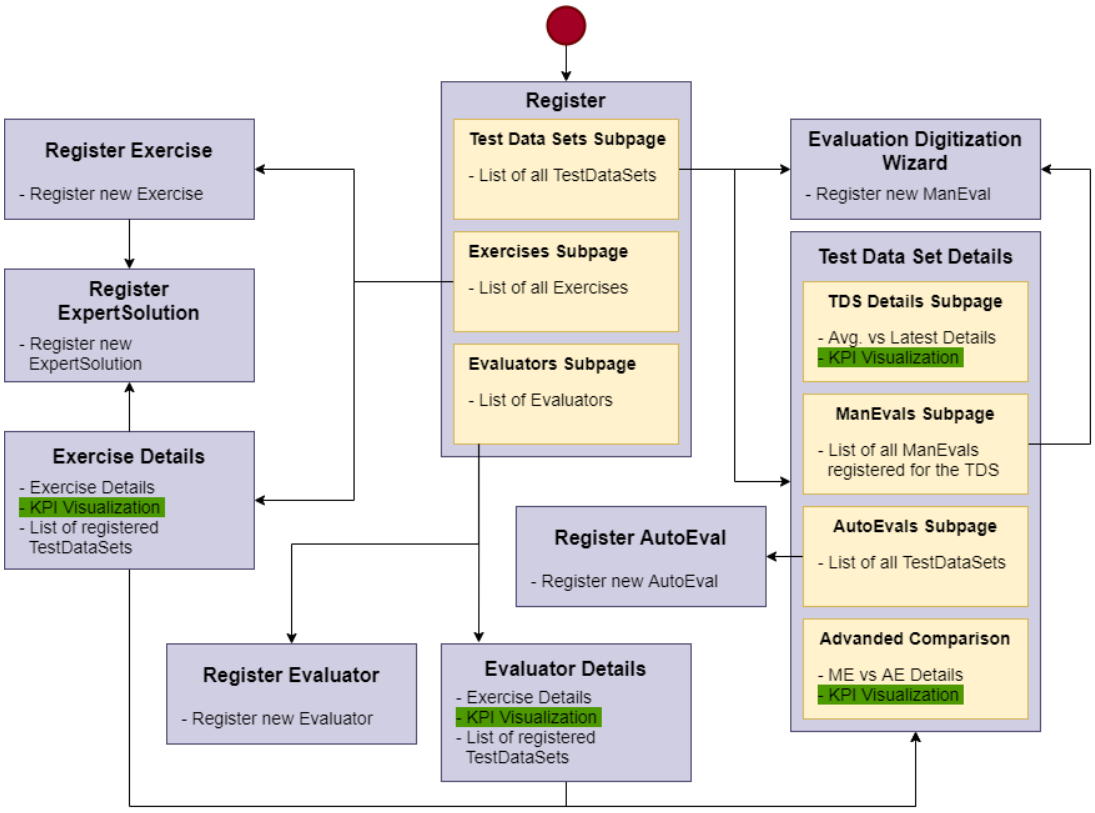



Figure 4.3. Abstract visualization of the navigation options the frontend supplies to the user. Top Level Views are marked in purple while sub-views, usually implemented using view tabs, are colored orange. The depicted arrows represent navigation options the user has in the particular view.

The *Register* possesses three subpages, as is illustrated in Figure 4.4. 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 subpage lists all stored Exercises. Again, each element of the list can be clicked to maneuver to a detail view of the Exercise. The last subpage, used to display the evaluators registered in the system behaves in the same way. To register a new evaluator in the application database, the user navigates to the *Register Evaluator* view. He is required to enter all relevant data. What data is required to create a new instance of one of the employed data elements, was discussed in Section 3.2.3 and will not be repeated for instance creation here. 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 Section 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.

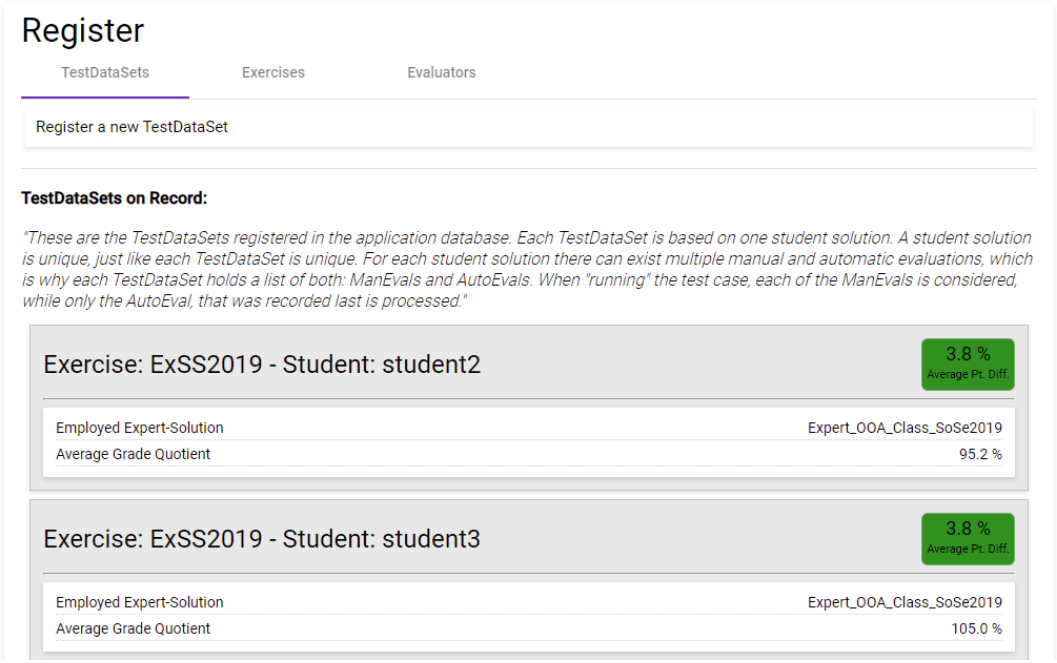



Figure 4.4. Exemplary screenshot of the *Register* view where the TestDataSet sub-page is opened. The general structure of the illustrated view is repeated for both the Exercise and Evaluator sub-pages.

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 (see 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 that contains the automatic evaluation. Most, if not all, 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 *Detail Views* regarding Evaluator, Exercise and TestDataSet. These 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 required values of 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.

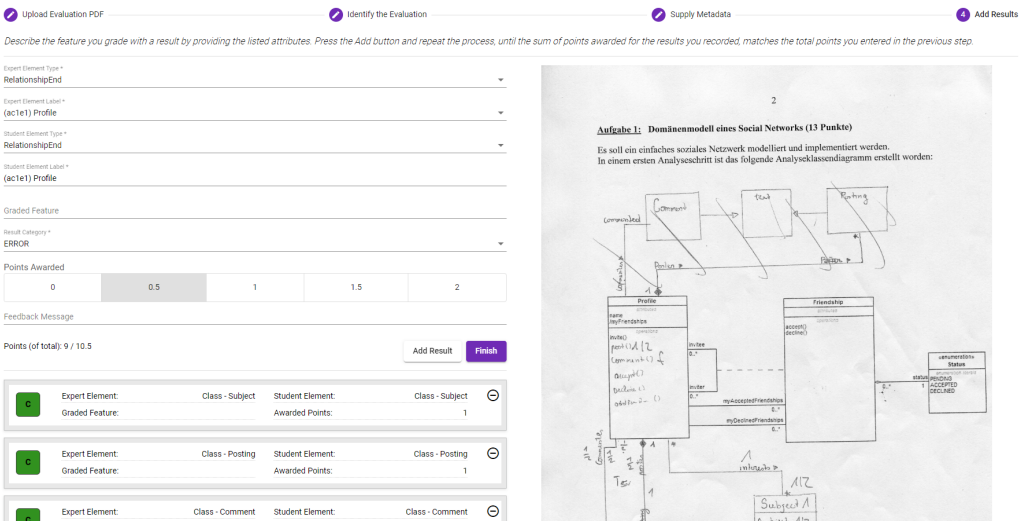



Figure 4.5. Screenshot of the fourth wizard page described in 4.3.3 while creating results from a manual evaluation. The PDF file of the manual evaluation can be seen on the right, while on the left both the form for entering result values and the result list are located.

The user has to create a *Result* for each graded element of the student solution. 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 two more optional attributes: 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 the comparison of at least one *ManEval* with at least one *AutoEval*. The *Detail View* of a TestDataSet represents the results of this comparison. *TestDataSets* can hold several evaluations and not all are presented with equal prominence. The meta-evaluation displayed in the initially shown *TestDataSet Details*-subpage results from the comparison of the *AvgManEval* with the latest *AutoEval*. The page shows the results of the *TestDataSet* meta-evaluation described in Section 3.3.1. To inspect the same information for a different comparison, the user has to navigate to the *Advanced Comparison*-subpage. The views for both the *TestDataSet Details*-subpage and the *Advanced Comparison*-subpage present almost the same content, however.

Both views show the results of the comparison of two evaluations. The total points awarded by both evaluations are displayed, as well as the grades, grade quotients, point, percentage differences and category matches. The *Advanced Comparison* subpage, in a tab for each, shows the results of the *points-per-expert-element-type*, *category-count* and *points-per-result-category* meta-evaluations, in a visualization. Additionally, a list view of the found result-categories per element can be displayed, as was described in Section 3.3.2.

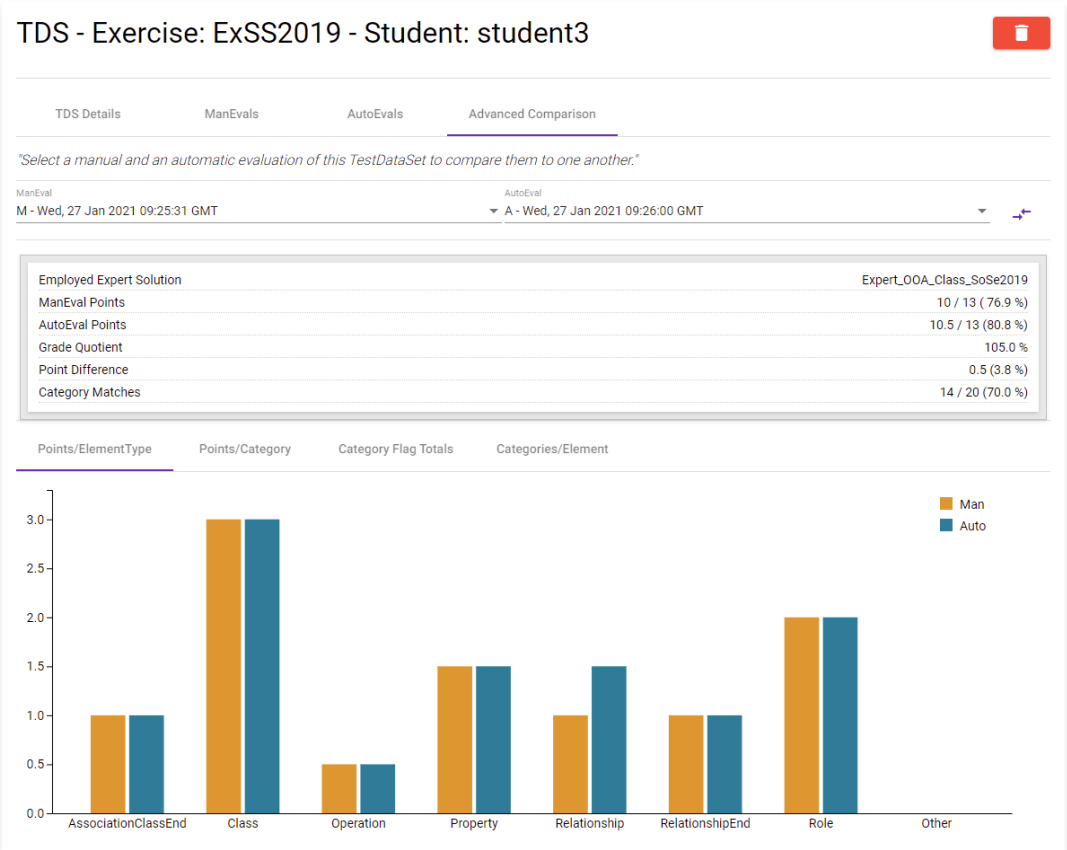



Figure 4.6. Screenshot of the *TestDataSet Details* view with the *Advanced Comparison*-subpage opened. The displayed chart shows the results of the meta-evaluation tagged *points-per-expert-element-type*.

### 4.3.5. Exercise Details View

The *Exercise Details*-view displays the results of the *Exercise*-level meta-evaluation. As was described in Section 3.3.1 the *Exercise*-level meta-evaluation is based on the *AvgManEvals* calculated for the *TestDataSets* that reference the examined *Exercise*. It follows that all values shown in the Exercise Details view are such average values, or values calculated from such. The main indicator for the quality of the evaluations INLOOM created in the context of an examined *Exercise* is the *Average Point Difference Percentage*, displayed in the top right corner of the *Exercise Details*-view. As was described in Section 3.3.2 it is calculated as the average of average *TestDataSet* point difference percentages. The displayed data sheet additionally shows the average of average grade quotients and the average total point difference. To enable the user to get a quick overview of the quality of the evaluations created by INLOOM, the previously listed values are additionally presented in the form of a comparative bar chart. These diagrams are intended to allow the user to identify *TestDataSets* among the examined ones, where INLOOM has achieved only an evaluation of particularly poor quality. As can be seen in Figure 4.7 the user has the opportunity to navigate to the detail views of the individual *TestDataSets* by clicking them in the list view displayed below the chart area of the view.

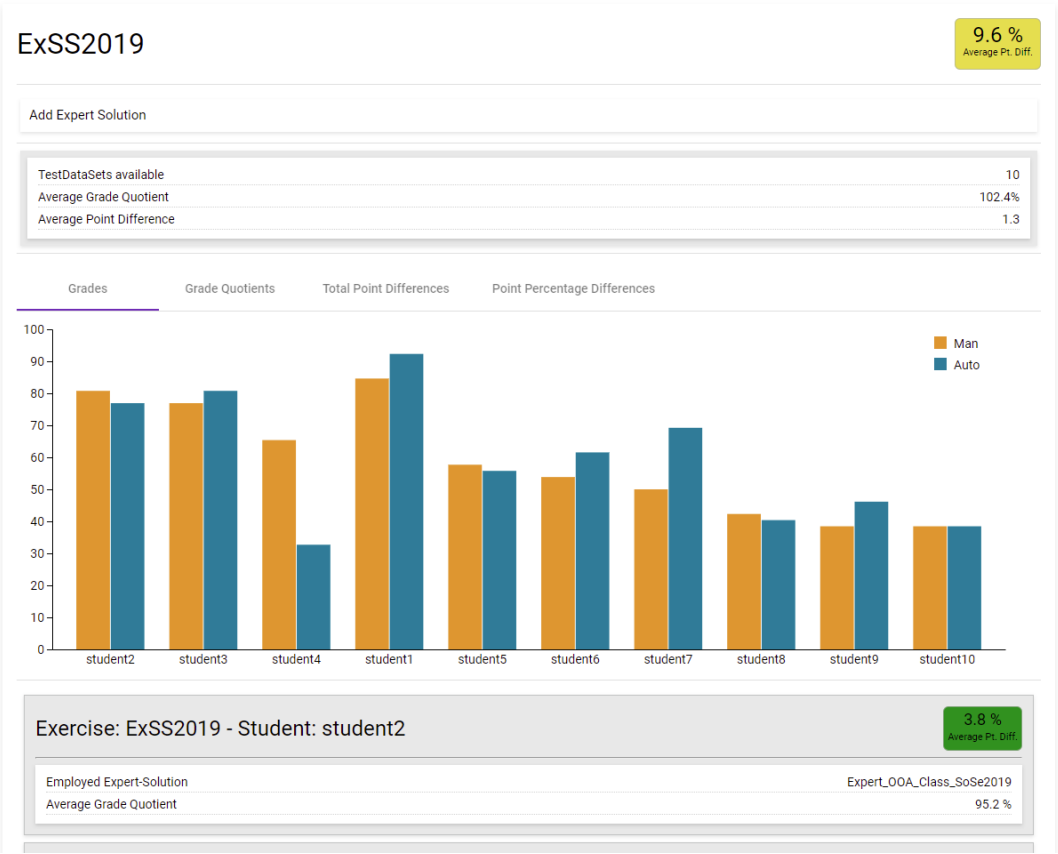



Figure 4.7. Screenshot of the *Exercise Details* view. The displayed chart shows a comparison of the grades awarded by the *AvgManEval* and latest *AutoEval* of each *TestDataSet* recorded for the inspected *Exercise*.

### 4.3.6. Evaluator Details View

The Evaluator Details view displays data on the evaluations created by a specific *Evaluator*. As was described in Section 3.3.1, an inspection of this level of detail is intended to detect patterns in the evaluation behaviour of the inspected *Evaluator*. For this purpose the *Evaluator Details* view contains a chart representation of the grades of all *ManEvals* created by the *Evaluator*, compared to the grade of the respective latest *AutoEval*. A graph of this form should help identify recurring patterns, such as typically lower scores, in the evaluators scoring methodology.

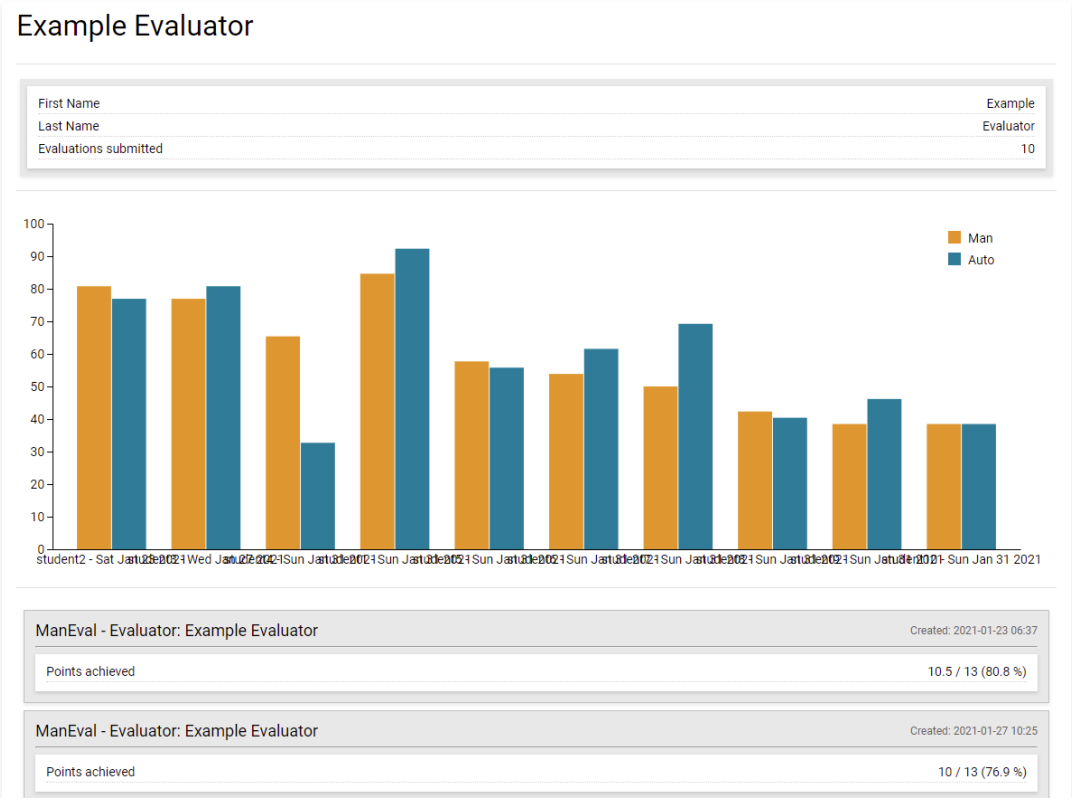



Figure 4.8. Screenshot of the *Evaluator Details* view. The displayed chart shows a comparison of the grades awarded by the *ManEvals* of the examined *Evaluator* with the ones awarded by the latest *AutoEval* of each *TestDataSet* recorded for the inspected *Exercise*.

# 5. Evaluation

The goal of this chapter is evaluating the evaluation of an evaluation. It turns out: evaluating the evaluation of an evaluation is quite a daunting task. It is approached by assessing whether the implementation of QIT presented in Chapter 4 meets the software requirements listed in Section 3.1. Some of the listed software requirements can be assessed objectively. QIT *does* employ a data structure that meets the specifications listed in SR6 and allows persisting multiple *Man-* and *AutoEvals* as is required by SR7. In addition, reasonable steps have been taken to meet the requirement set by SR3 insofar as possible. Data storage, management and presentation are supplied in separate processes as was described in Section 4.1. To satisfy SR5 QIT supplies the *XMLAdapter* (see Section 4.2.2) and allows the user to access it using the facilities described in Section 4.3.2. For the other software requirements, however, any assessment performed by the author will probably be highly subjective. Formally measuring, to what extent they were met will not be possible within the time frame of this thesis.

A number of features were implemented to meet these requirements. QIT allows the user to enter data on manual evaluations by means of the *Evaluation-Digitization-Wizard* described in Section 4.3.3. to satisfy SR4. However, it is difficult to assess how easy it is for the user to operate the wizard. SR1 requires the implemented application to perform a "comprehensive" evaluation of the *quality* of the system. As was discussed in Section 2.2 defining quality in the context of automatic assessments is no trivial task, nor is objectively judging whether the performed evaluation is "comprehensive". The same problem occurs when trying to assess to which extent the presented implementation meets the requirement set in SR2. Both SR1 and SR2 in the last effect require a usability feature. A properly conducted usability study of the presented software would involve several test users whose experiences with the software would have to be collected and evaluated. However, such an approach would go beyond the time frame of this thesis and would only indirectly contribute to answering the research questions formulated in Section 1.2. To still give an impression of the extent to which the implemented features fulfil SR1 and SR2, the use of the software in a typical use case is described. Experiences and encountered problems are detailed. In addition, the resulting meta-evaluation is compared with one based solely on the resulting final score, as it was frequently described in the literature. This intends to demonstrate what distinguishes the more extensive evaluation to be undertaken by QIT from such an evaluation. Two use cases are examined. The first one is adding a new *TestDataSet* to the application database. The second inspecting INLOOMs performance using QITs results. Put together, these two use cases involve all features implemented to meet requirements SR1, SR2 and SR4.

## 5.1. Test Case Evaluation: Adding a TestDataSet

To add a new *TestDataSet* to QITs database, a user has to add a *ManEval* and an *AutoEval* for the same student solution. This use case was described superficially in Section 3.2.4. This test case evaluation focuses on requirement SR4. Adding a new *AutoEval* will not be evaluated since it was established in Section 5, that requirement SR5 which requires QIT to allow adding *AutoEvals*, can be assumed met.

Adding manual evaluations using the *Evaluation-Digitization-Wizard* is intended to be a straight forward process. Design and implementation focused on the wizards *ease of use*, which is why this test case evaluation will do likewise. To test the *Evaluation-Digitization-Wizard*, the available manual evaluations of the student solutions for the exam of the summer term 2019 were digitized with it. There were ten such evaluations available. These were evaluated based on the final grade and the point difference percentage, in an initial evaluation of INLOOMs results performed by the authors of [13]. This subsection will recount the experiences made while digitizing these ten manual corrections. The structure of the report will follow the structure of the subsection. Any kind of assessment regarding the usability of the features employed during the digitization is based on the assumption that a user has at least basic knowledge of the problem domain. After all required data has been entered, the user registers the new manual evaluation in the application database by finishing the last wizard page.

### 5.1.1. Wizard Page 1 - Upload Evaluation PDF

Throughout the performed digitizations, no problem was encountered in the context of the first page of the wizard. The page is used to upload a manual evaluation to be presented in the side-by-side view the wizard intends. Although, the wizard allows for it, the option to not upload a *ManEval* PDF file was not employed.

### 5.1.2. Wizard Page 2 - Identify the Evaluation

There were more problems when using Wizard page two. The intended use of the page is - selecting the *Exercise* the student solution was submitted for from the exercises registered in QITs database - selecting the *ExpertSolution* the submission was compared against and - entering the id of the student who submitted it. The first two proved to be trouble-free. However, entering the student id did not. The student id is important since it is one of the attributes that is used to match manual evaluations to automatic ones. It has to be entered *correctly*. The only clue the user gets about the correct id, however, is the filename of the PDF file they select. This was the reason for some frustration and should be addressed to make QIT more *useable*.

### 5.1.3. Wizard Page 3 - Supply Meta-Data

Selecting an evaluator and entering the number of total points awarded to the submitted solution did never prove to be a problem. However, it must be acknowledged that it might become one in the future, since the current PDF files do not contain any information on the manual evaluator.

### 5.1.4. Wizard Page 4 - Add Results

As was to be expected, since it is the most complex of the four wizard pages, the fourth was the wizard page where the greatest number of difficulties was encountered. These, for the most part, resulted from the users inexperience with grading student solutions, however, since digitizing manual evaluations became *easier* over time. Since the test user had a deep understanding of the tool he was using, it can be assumed, with a decent amount of certainty, that knowing about the evaluated exercise was the pivotal issue.

In many cases identifying the correct expert element type and label involved a lot of puzzling. As was mentioned in Section 3.2.2 the manual evaluations available for digitization consist of annotations in black and white scans of student solutions that are stored as PDF files. This format often makes it very difficult to decide what certain annotations are referring to. However, there was only one case in which a point annotation could not be matched at all. This was the case for the student solution submitted by the student tagged "student4". The submitted solution was very crowded and used a rather novel approach to solve the task at hand.

Another problem presented itself in the form of the category flag. Deciding after the fact, if the manual evaluator accepted the supplied solution as *CORRECT* or as *WARNING* is not possible. Additional information on the amount of reluctance with which the evaluator awarded the annotated points was seldom available. Also without intimate knowledge of the grading scheme, it often was no simple task to identify an *ERROR* as such based just on the points awarded for the feature. Especially when switching rapidly between expert element types, it was proved hard not to get confused about what feature of what type awarded how many points. As with identifying what an annotation relates to, correctly identifying an *ERROR* as such became easier over time. The number of points was the only required value whose specification was not a problem in any of the exercised test cases.

## 5.2. Test Case Evaluation: Quality Inspection

After adding data for multiple manual and automatic evaluations, an inspection of INLOOMs current result quality can be performed. The evaluation QIT performs on the exercise level is very similar to the one performed in the original publication of INLOOM [13]. Other than the described one, QITs evaluation is not solely based on the final grade. QIT is intended to additionally allow an examination of the composition of that grade. By presenting the results of such a more detailed analysis, QIT aims to convey information not only about if INLOOMs evaluation resulted in the correct final grade, but also if that final grade was awarded for the correct *reasons*.

The workflow for an inspection of the quality of INLOOMs results in the context of a specific exercise begins with examining the *Exercise* level *MetaEval*. The displayed data should allow gaining a first impression and help the user identify the *TestDataSets*, available in the context of the *Exercise*, where there are the biggest discrepancies between evaluations. The next step of the inspection process is examining the *MetaEvals* of these *TestDataSets*. In this step, the goal of the user is to identify the *origin* of the difference between the grades INLOOM and a manual evaluation award to the student solution of a particular *TestDataSet*. This step is what distinguishes the evaluation QIT performs from the evaluation performed in the initial publication.

### 5.2.1. Exercise Level Meta-Evaluation

To inspect the *Exercise* level *MetaEval* the user navigates to the *Exercise Details View*, which was described in Section 4.3.5. Using the various meta-evaluations visualized in the view (see Figure 5.1) the user has access to about the same amount of data presented for the meta-evaluation performed in the initial evaluation of INLOOM [13].

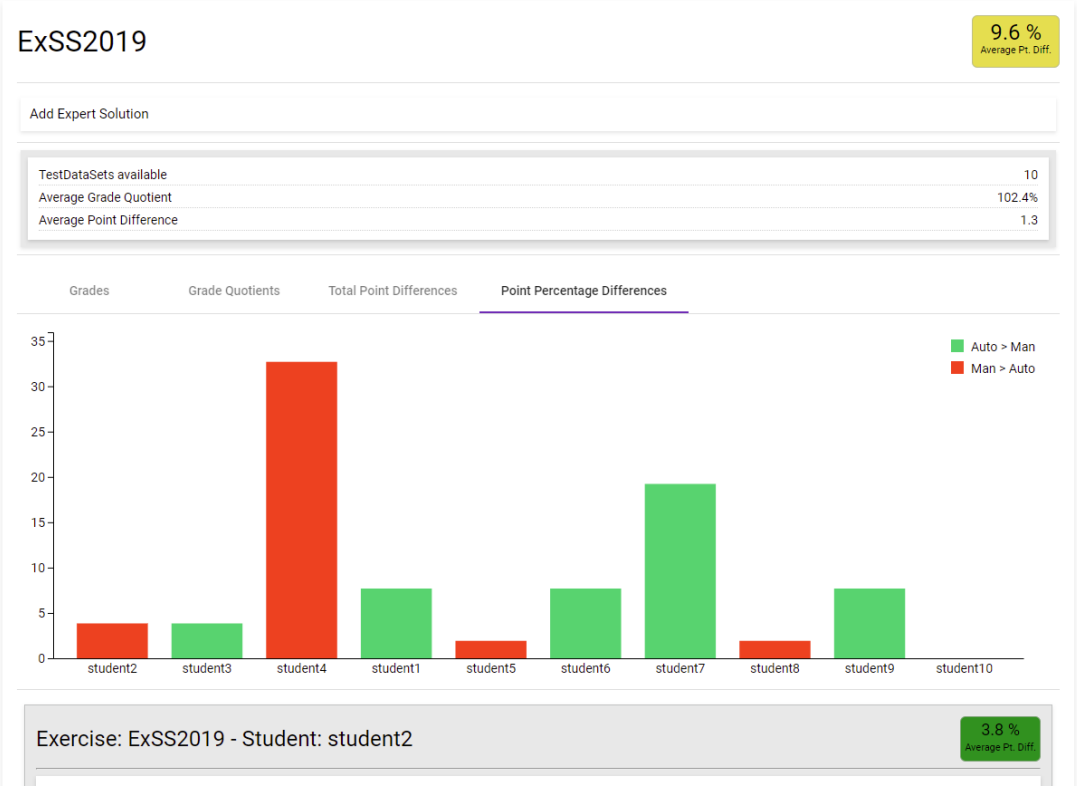



Figure 5.1. Screen-capture of the *Exercise Details View* with the tab visualizing the point percentage differences of all *TestDataSets* available for the *Exercise*.

The meta-evaluation of the evaluations created for student submissions for the exam of the summer term 2019 showed that there is on average a 9.6% deviation between the total number of points awarded by the *Man-* and *AutoEval*. This information is readily available to the user, as it is displayed in a colored button in the top right corner of the view. From the indicator he gets a very general first impression of the quality of INLOOMs results for the *Exercise*.

The visualizations in the *Exercise Details View* let the user inspect the KPIs calculated for each *TestDataSet* of the *Exercise* at once. Using the visualization displayed in the *Point Percentage Differences* tab of the view, the user can identify the *TestDataSets* of the student solutions "student4" and "student7" as the ones with the greatest difference between the grades awarded by the compared evaluations. The user is supplied the opportunity to investigate these *TestDataSets* further by navigating to their respective detail views. They can achieve this by clicking the entry of the *TestDataSet* they are interested in, in the list displayed at the bottom of the *Exercise* details view.

### 5.2.2. TestDataSet Level Meta-Evaluation

After navigating to the *TestDataSet Details View* of the *TestDataSet* they are interested in, the user intends to use the presented information to identify existing sources of error. They can perform an in-depth examination of the meta-evaluation they are interested in, by selecting a comparison of evaluations in the *Advanced Comparison* tab of the detail view.

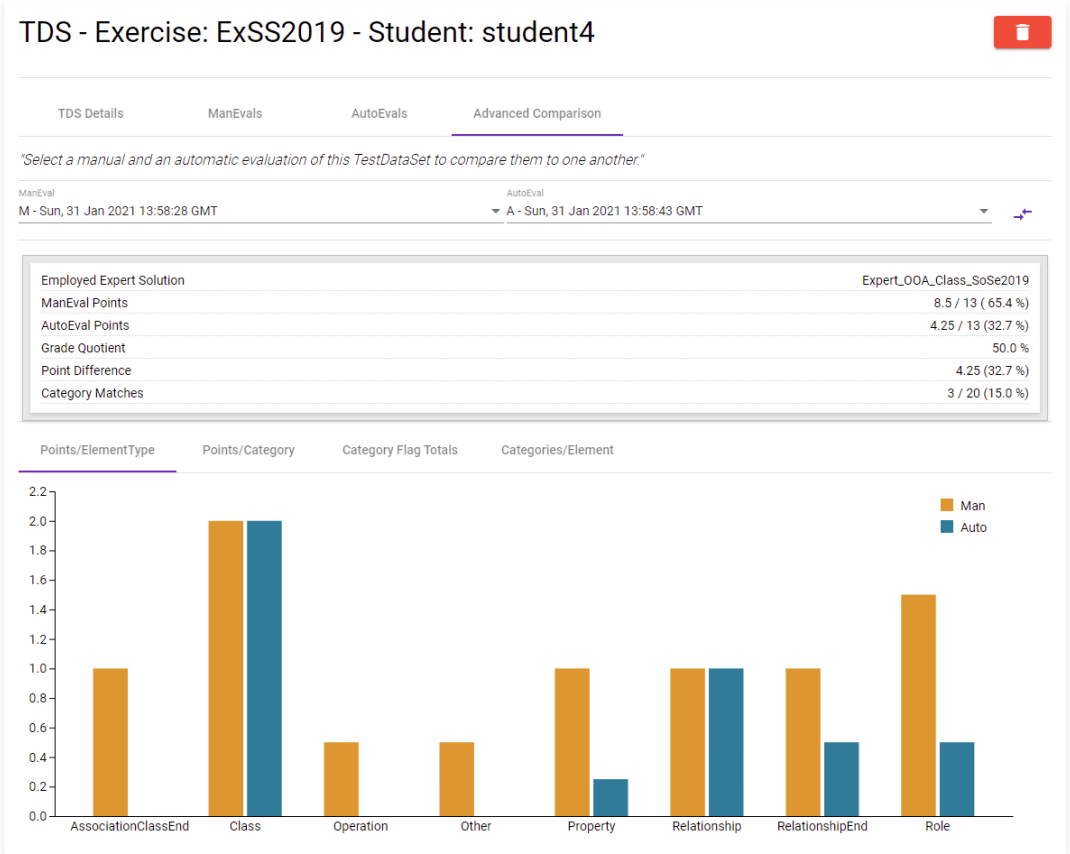



Figure 5.2. Advanced Comparison tab of the "student4" *TestDataSet*.

The *Points/ElementType* tab shown in Figure 5.2 lets the user identify the types of element where the evaluations differ. In the case of "student4", the user can assume from the displayed visualization that INLOOMs evaluation, other that the manual evaluator has not been able to identify many of the elements the *ExpertSolution* demands. It awards no points at all *for* many of the listed element types. If the user is very familiar with the expert solution, he can also conclude that the difference probably results from difficulties while assessing the association class required in the expert solution.

By checking the *Category Flag Totals* tab, the user can test the validity of their earlier assumption. The visualization depicted in Figure 5.3 proves them right.

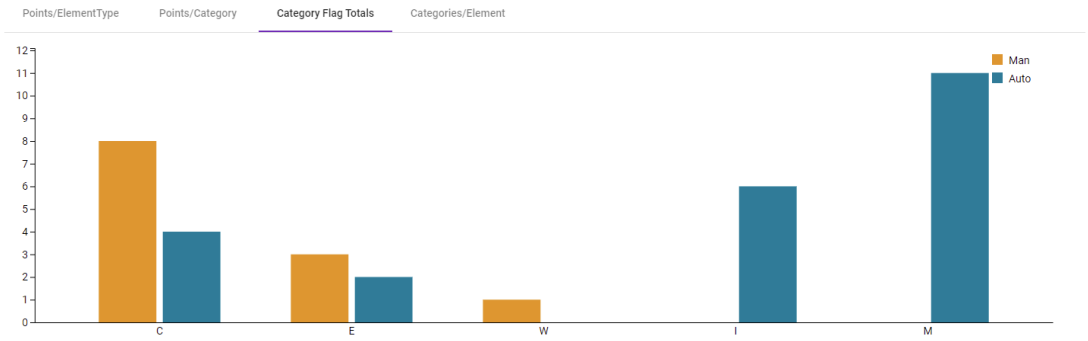



**Figure 5.3.** Screen-capture of the *Category Flag Totals* tab.

This first use case of the *TestDataSet Details View* enables the user to identify major sources of error. It is assumed that an experienced user, from the data displayed, will be able to make an educated guess on where problems with certain characteristics might originate. In the case of "student4" a probable assumption could be, that there was a problem in INLOOMs element matching, or that the student employed novel labels for the classes he drew. A manual evaluator might judge those as sufficient while INLOOM is not able to make sense of them at all.

A more advanced use case of QIT is inspecting *TestDataSets* for their precision. In Section 3.3.2 the possibility is discussed that the grades two evaluations result in are almost equivalent, yet are result of points awarded to different features. This is, for example, the case for "student10" as is shown in Figure 5.4.

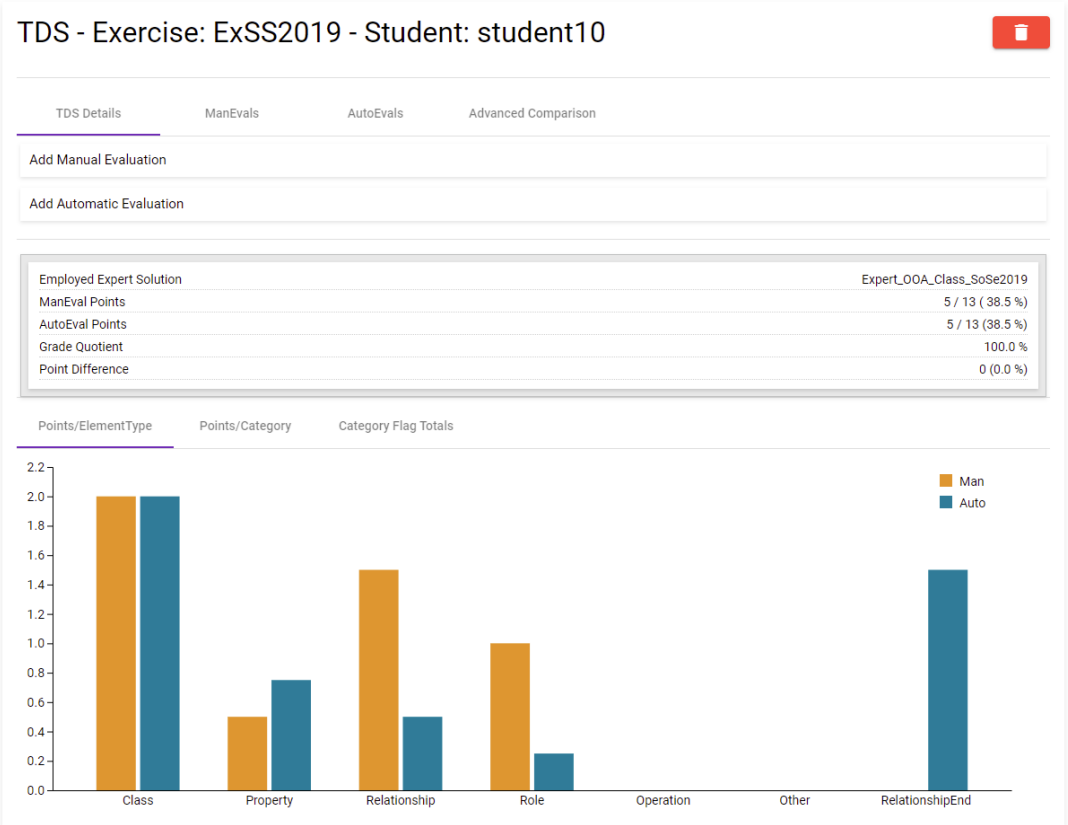



**Figure 5.4.** Screen-capture of the *Points/ElementType* of *TestDataSet* "student10".

Both evaluations awarded the same amount of points. However, the automatic did so for *RelatitonshipEnd*-type elements, while the manual evaluator awarded points to elements of the *Relationship* type. The user will probably ignore a difference on this scale. It seems reasonable that the evaluations might differ in this way. Performing a meta-evaluation on the *Exercise* level only however, could not have enabled the user to choose to ignore the issue but would have left him ignorant about it.

The *TestDataSet* for the context of the student solution submitted by "student3" averages at 3.8% point difference percentage. This is a greater deviation than the 0% "student5" showed, yet the user, from the data displayed in the *TestDataSet Details View* of "student3", will conclude a way higher quality than he does from the what "student3" shows. The *TestDataSet Details View* of "student3" is depicted in Figure 5.5.

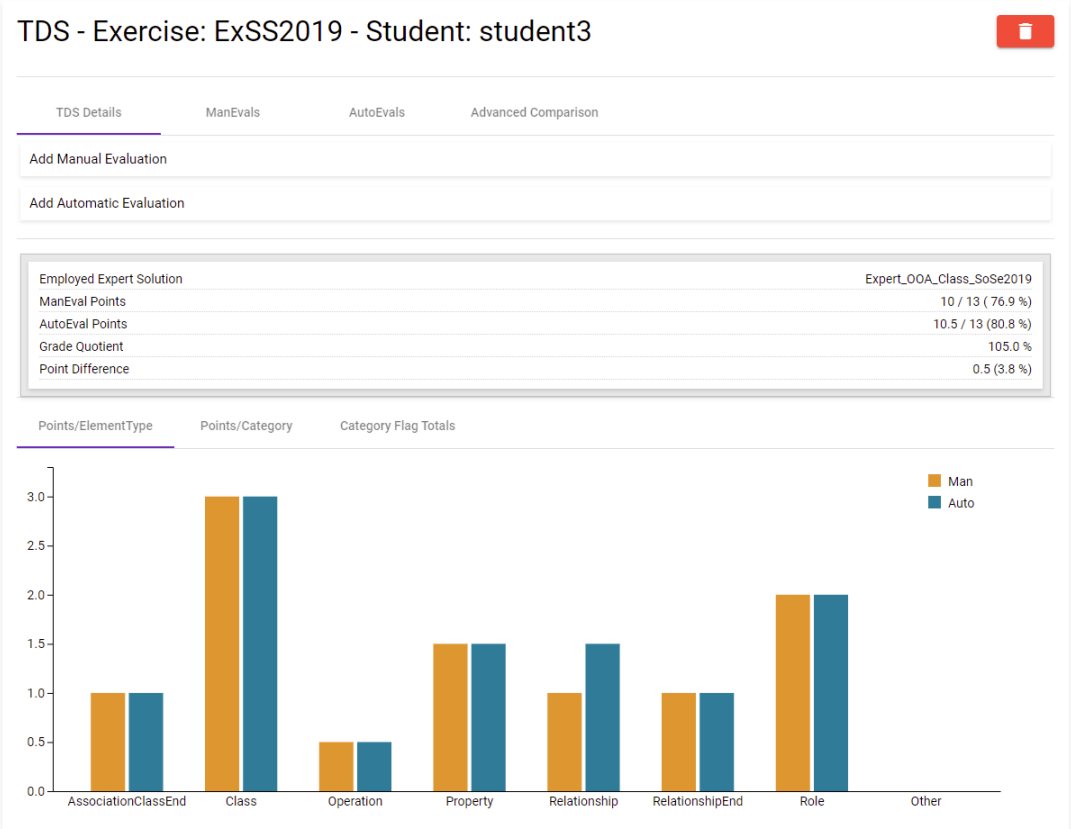



Figure 5.5. Screen-capture of the *Points/ElementType* tab of the "student3" *TestDataSet*.

## 5.3. Summary

In summary, it can be stated that the evaluation, carried out in form of the two presented test cases, has shown that all requirements have been fulfilled at least to a certain extent. Whether the meta-evaluation carried out is comprehensive or not is left to the user's assessment for the time being. The presented *Evaluation-Digitization-Wizard* allows the digitization of manual evaluations from PDF files and original paper versions of student solutions. How well the user is able to operate it, however, depends on his knowledge of both the tool and the *ExpertSolution* the evaluation used for reference. The meta-evaluation performed by QIT is more detailed than a simple comparison based solely on the final grade. It allows inspection individual comparisons of automatic and manual evaluations and thus the inspection of the composition of the final grade. Yet, once again, how useful the data QIT provides is in the particular use case of a specific user can not be estimated. The implementation allows for extensions to be added in the future and thus at least enables users who's needs, the data QIT displays does not satisfy, access to pre-processed data to use in the evaluation they intend. The presented prototype implementation matches the expectations the author nurtured during the design phase of the project.

# 6. Conclusion and Future Work

## 6.1. Research Questions

The problem this thesis addresses was approached by formulating five research questions. To check to what extent these research questions have been answered, this section summarizes what steps have been taken to that purpose.

RQ1 Which values can be extracted from the manual and automatic evaluations?

It was found that although the availability of data presents a considerable limitation, all data required to completely reproduce an evaluation digitally, to the extent required to perform a in-depth meta-evaluation, can be extracted from both manual and automatic evaluation input files. What data is available for extraction is discussed in the Section 3.2. The implementation of an adapter class, which is employed to read all required values from the automatic evaluation XML files, is described in Section 4.2.2. For extracting all required data from the manual evaluations provided as black and white PDF files, the implementation of a UI facility was described in Section 4.3.3. The evaluation of this implementation (see Section 4.3.3) confirmed that it is sufficiently operational.

RQ2 How can the quality of INLOOMs results be quantified?

To answer this research question, a definition of quality in the context of this thesis was presented in Section 2.2. It was established that for an assessment of the quality of an subjective evaluation, the evaluation must be measured against a reference evaluation that is assumed optimal. A literature research into the quality validation of existing systems for automated assessment of student solutions, described in Section 2.3, showed that a manual evaluation, created by a human reviewer is commonly employed as such an *optimal reference*. INLOOM's quality, in the sense of the definition given in Section 2.2, can thus be quantified by the degree of deviation between the evaluation produced by INLOOM and one produced by a human reviewer.

RQ3 What methodologies are employed by existing ITS to validate their results?

This research question was approached with a literature research into the quality validation strategies of existing model assessment systems (see Section 2.3). It was found that strategies which are based on numerically comparing the final grades awarded by the compared evaluations are the most commonly employed.

Many of the publications examined during the research did not perform a quality validation regarding the correctness of their presented method's results. Some, however, did perform an examination of the didactic use of their tools. Such evaluations were disregarded as they examine a quality criterion that is not part of the definition of quality adopted here (see Section 2.2).

RQ4 How can the developer/tutor/instructor best be assisted in collecting and preprocessing the evaluation data required for quality validation?

Section 3.2.4 discusses how the user can be assisted in collecting the required data. Based on this discussion, in Section 4.3.3 the implementation of a facility to aid in digitizing manual evaluations was described. Section 4.2.2 presents the implementation of a facility to automatically read all required data from INLOOMs output XML files. However, this leaves the research question unanswered. Conclusively, determining how *best* to support the teaching staff, will probably remain an impossible task in the future.

The evaluation of the Digitization Wizard carried out in Section 4.3.3 could at least confirm that the input of all values necessary for the meta-evaluation proposed here is possible and can be managed without any problems, assuming expertise.

RQ5 How can test results be presented to greatest effect?

In Section 3.3.2 various meta-evaluations of the evaluation data were proposed. For each of the performed meta-evaluations, an appropriate visualization was found and described in the context of its *location* in the frontend of the software. The frontend facilities employed were described in the Sections 4.3.4, 4.3.5 and 4.3.6.

Once again, this leaves the actual research question unanswered. It will not be possible to determine conclusively, how the data can be presented to the *greatest* effect. However, the test case evaluation performed in Section 5.2 showed that, using QIT, it is possible to perform a detailed evaluation of INLOOMs current results. The research question remains unanswered, yet a workable solution to the problem it addresses was found.

## 6.2. Future Work

The one presented is a prototype implementation. As with any prototype implementation, there are still many unresolved issues and places where an extension of the scope of functionality would be worthwhile. The essential work, however, will be to integrate QIT into the INLOOM *workflow*. Many features QIT offers are yet rather theoretical, as there does not exist any actual data to use them on. For any meaningful evaluation of INLOOMs result quality, it will be necessary to collect way bigger numbers of manually evaluated student solutions to test INLOOM against. To this end it could be advisable to rethink the way student solutions to modelling tasks are evaluated. In the test case evaluation performed in Section 5.1, it was found that if the manually evaluated student solutions were to contain more meta-data, this would simplify digitizing them. In a first step, these values could be added. However, the issue of missing information on the student who submitted the solution, is just a symptom of a bigger problem. As was stated in Section 1.1 one of the main intentions behind employing tools for automatic assessment is *reducing* the amount of effort for the involved teaching staff.

If to make sure the employed automatic assessment tool works at an adequate quality level, it is necessary to, not only evaluate some of the student submissions manually still, but also to digitize these evaluations afterwards, there remains room for improvement. A reduction in the amount of work is still evident. Yet, it is only gradual because the effort to evaluate INLOOMs results for a particular task only has to be made once initially. It can be assumed that a regular check of the current quality will only require a fraction of the effort that was necessary to build up an initial basic data set. However, adding a new task to INLOOMs repertoire retains a high cost. To mitigate such issues, it might be opportune to standardize the manual evaluation process further, to make digitizing the resulting evaluations easier.

There also remains a lot to be done about QIT itself. As previously stated, the current implementation is a prototype and still very rough around the edges. Important features to interact with the recorded data, such as searching, editing and deleting entities, are missing from the application entirely. The focus during implementation was to facilitate the meta-evaluation. All these features remain to be implemented at a later time, once a successful deployment of QIT has vindicated such effort. Possibilities to increase the users commodity aside, there still are many ways to improve QIT. The application is explicitly designed to be extended in the future. A host of additional meta-evaluations of the collected data are feasible, as well as different ways to present the results of the meta-evaluations which are already being performed.

For many such features, however, the question, whether it is worth adding them depends on how good the quality of the digitized manual evaluations is. For example, an evaluation of the assigned category flags is only worthwhile if these flags are correctly recorded in the digitization of the manual evaluations. Since currently there is no way to extract the correct category flags from the manual evaluations with any amount of certainty, comparing them to the flags assigned by INLOOM has little point. If this problem was to be solved an even more detailed level of meta-evaluation could be performed. A meta-evaluation based on the accuracy with which INLOOM assesses or rather categorises a single expert element. Such an evaluation would provide even deeper insights than the meta-evaluation currently conducted, which provides results where one expert element type is the maximum resolution.

A possible solution for the aforementioned problems could be to unify the evaluation and digitization processes by evaluating the student solutions digitally. A facility like QITs *Evaluation-Digitization-Wizard* could be employed to give the evaluators a framework in which to create their evaluation. Such a framework would ease standardizing the process and thus increase the fairness of the grades the students receive. By digitizing the process workload could be distributed between evaluators easily and evaluation data would be available for instant analysis. Performing the evaluation in a digital environment would also allow collecting data on the evaluation process itself, which could help streamline the process further or be the base for adaptions in the wording of exercises or the configuration of INLOOM.

## 6.3. Conclusion

The title of this thesis is "Concepts of Quality Assurance for the Constraint-based E-Assessment of Models". The stated goal of the thesis was to identify a method to facilitate testing of INLOOMs quality. To this purpose, five research questions have been posed (see Section 1.2) and answered (see Section 6.1). A literature research into the quality validation strategies of existing automatic assessment systems was performed (see Chapter 2). The research showed that the most commonly employed strategy for validating the results of automatic evaluations is to perform a comparison between the evaluation created by the inspected tool and a manually created one. Such a comparison is usually conducted based on the total points, the compared evaluations awarded to the evaluated student solution. The design of QIT (*Quality Inspection Tool*) the prototype of a tool for meta-evaluating evaluations, which based on the same principle, was presented in Chapter 3. After a prototype implementation of QIT was presented in Chapter 4. QIT is implemented as a web-application that is deployed locally. This architecture allows for a high amount of flexibility and for the application to be extended or adapted to different use cases in the future. The implementation was evaluated based on test cases that reflect the two use cases that are anticipated to be required most frequently, in Chapter 5.

The performed Evaluation was able to show, that using QIT it is possible to perform a detailed analysis of INLOOMs current quality and to detect errors in the automatic evaluation, which it is not possible to detect when comparing evaluations based solely on the total number of points awarded or the grade.

# A. Attachments

## A.1. Deployment Guide

This guide is intended to explain how to deploy the implemented application.

### A.1.1. Requirements

The application was only tested on Microsoft Windows 10. However there should be no reason for QIT not to work on any other operating system that meets the rest of the requirements. Since QIT is implemented as a web application a browser is required to access it. Google Chrome was used during testing.

#### Python

- Version: 3.9.1
- Download: https://www.python.org/downloads/

#### Python Packages

- Flask:
  - Version: 1.1.2
  - Download: https://flask.palletsprojects.com/en/1.1.x/installation/

### A.1.2. Deployment

The user can run QIT by navigating to the "src" directory of the implementation and executing the command "python main.py". Alternatively they can use the file "start_qit.bat" which executes the command for them. This will make Flask serve the frontend application on localhost port 4200 and QITs backend on localhost port 3001. The user can access the app by navigating to to "localhost:4200" in their browser.

## A.2. Meta-Evaluation JSON

Listing A.1 Example for the json the *MetaEvalServer* serves to QITs frontend.

```
{
    "3461d41e-8838-44a1-9406-60ad5f781994": {
    "comparison-stats": {
      "category-combinations": {
        "9ca79272-16fe-4fef-abad-ad5d128fb75747f44aa8-faa3-4b36-b3b3-8ac93c786639": {
          "AssociationClassEnd - (ac1_ac) Comment": {
            "man": "C"
          },
          "Class - Comment": {
            "auto": "C",
            "man": "C"
          },
          "Class - Posting": {
            "auto": "M"
          },
          "Class - Subject": {
            "auto": "C",
            "man": "C"
          },
          "Operation - Posting_comment": {
            "auto": "M",
            "man": "E"
          },
          "Operation - Profile_post": {
            "auto": "M"
          },
          "Other - Other": {
            "auto": "I",
            "man": "W"
          },
          "Property - Comment_text": {
            "man": "E"
          },
          "Property - Posting_text": {
            "auto": "M"
          },
          "Property - Subject_subject": {
            "auto": "M",
            "man": "E"
          },
          "Property - text": {
            "auto": "E"
          },
          "Relationship - (r1) Profile-Subject-Shared": {
            "auto": "C",
            "man": "C"
          },
          "Relationship - r2": {
            "auto": "M"
          },
          "RelationshipEnd - (ac1e1) Profile": {
            "auto": "E",
            "man": "C"
          },
          "RelationshipEnd - ac1e2": {
            "auto": "M"
          },
          "Role - (ac1e1_commenter) commenter": {
            "auto": "M",
```

```
            "man": "C"
          },
          "Role - (ac1e2_commented) commented": {
            "auto": "M"
          },
          "Role - (r1e2_interest) interest": {
            "auto": "C"
          },
          "Role - (r2e1_poster) poster": {
            "auto": "M",
            "man": "C"
          },
          "Role - (r2e2_posting) posting": {
            "auto": "M",
            "man": "C"
          }
        }
      },
      "category-matches": {
        "9ca79272-16fe-4fef-abad-ad5d128fb75747f44aa8-faa3-4b36-b3b3-8ac93c786639": {
          "match": 3,
          "mismatch": 17,
          "pct-matched": 15.0
        }
      },
      "grade-quotients": {
        "9ca79272-16fe-4fef-abad-ad5d128fb75747f44aa8-faa3-4b36-b3b3-8ac93c786639": 0.5,
        "latest-auto-evalavg-man-eval": 0.5
      },
      "grade-quotients-pct": {
        "9ca79272-16fe-4fef-abad-ad5d128fb75747f44aa8-faa3-4b36-b3b3-8ac93c786639": 50.0,
        "latest-auto-evalavg-man-eval": 50.0
      },
      "point-diffs": {
        "9ca79272-16fe-4fef-abad-ad5d128fb75747f44aa8-faa3-4b36-b3b3-8ac93c786639": 4.25,
        "latest-auto-evalavg-man-eval": 4.25
      },
      "point-pct-diffs": {
        "9ca79272-16fe-4fef-abad-ad5d128fb75747f44aa8-faa3-4b36-b3b3-8ac93c786639": 32
                  .69230769230769,
        "latest-auto-evalavg-man-eval": 32.69230769230769
      }
    },
    "eval-stats": {
      "47f44aa8-faa3-4b36-b3b3-8ac93c786639": {
        "category-count": {
          "C": 8,
          "E": 3,
          "W": 1
        },
        "created": 1612101508,
        "grade": 65.38461538461539,
        "points-per-expert-element-type": {
          "AssociationClassEnd": 1.0,
          "Class": 2.0,
          "Operation": 0.5,
          "Other": 0.5,
          "Property": 1.0,
          "Relationship": 1.0,
          "RelationshipEnd": 1.0,
          "Role": 1.5
        },
        "points-per-result-category": {
```

```
          "C": 6.5,
          "E": 1.5,
          "W": 0.5
        },
        "total-points": 8.5,
        "type": "M"
      },
      "9ca79272-16fe-4fef-abad-ad5d128fb757": {
        "category-count": {
          "C": 4,
          "E": 2,
          "I": 6,
          "M": 11
        },
        "created": 1612101523,
        "grade": 32.69230769230769,
        "points-per-expert-element-type": {
          "Class": 2.0,
          "Operation": 0.0,
          "Other": 0.0,
          "Property": 0.25,
          "Relationship": 1.0,
          "RelationshipEnd": 0.5,
          "Role": 0.5
        },
        "points-per-result-category": {
          "C": 3.5,
          "E": 0.75,
          "I": 0.0,
          "M": 0.0
        },
        "total-points": 4.25,
        "type": "A"
      },
      "avg-man-eval": {
        "category-count": {
          "C": 8,
          "E": 3,
          "W": 1
        },
        "grade": 65.38461538461539,
        "points-per-expert-element-type": {
          "AssociationClassEnd": 1.0,
          "Class": 2.0,
          "Operation": 0.5,
          "Other": 0.5,
          "Property": 1.0,
          "Relationship": 1.0,
          "RelationshipEnd": 1.0,
          "Role": 1.5
        },
        "points-per-result-category": {
          "C": 6.5,
          "E": 1.5,
          "W": 0.5
        },
        "total-points": 8.5
      },
      "latest-auto-eval": {
        "category-count": {
          "C": 4,
          "E": 2,
          "I": 6,
```

```
          "M": 11
        },
        "created": 1612101523,
        "grade": 32.69230769230769,
        "points-per-expert-element-type": {
          "Class": 2.0,
          "Operation": 0.0,
          "Other": 0.0,
          "Property": 0.25,
          "Relationship": 1.0,
          "RelationshipEnd": 0.5,
          "Role": 0.5
        },
        "points-per-result-category": {
          "C": 3.5,
          "E": 0.75,
          "I": 0.0,
          "M": 0.0
        },
        "total-points": 4.25,
        "type": "A"
      }
    },
    "exercise-id": "ExSS2019",
    "expert-solution-id": "Expert_OOA_Class_SoSe2019",
    "student-id": "student4"
  }, ...
}
```

# List of Figures

# List of Tables

# 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] P. Beck et al. "COCLAC - Feedback Generation for Combined UML Class and Activity Diagram Modeling Tasks". In: *Revista Brasileira de Psiquiatria* (2015).

[4] Jan Philip Bernius et al. "Towards the Automation of Grading Textual Student Submissions to Open-Ended Questions". In: *Proceedings of the 4th European Conference on Software Engineering Education*. ECSEE '20. Seeon/Bavaria, Germany: Association for Computing Machinery, 2020, pp. 61–70.

[5] W. Bian, O. Alam, and J. Kienzle. "Automated Grading of Class Diagrams". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 700–709.

[6] 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.

[7] 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.

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

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

[10] 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, pp. 138–141.

[11] Bitcom e.V. *Erstmals mehr als 100.000 unbesetzte Stellen für IT-Experten*. URL: `https://www.bitkom.org/Presse/Presseinformation/Erstmals-mehr-als-100000-unbesetzte-Stellen-fuer-IT-Experten` (visited on 02/03/2021).

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

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

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

[15] 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.

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

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

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

[19] Gwet KL. “Computing inter-rater reliability and its variance in the presence of high agreement”. In: *Br J Math Stat Psychol.* (May 2008), pp. 29–48.

[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: *Proceedings of the Workshop on Intelligent Tutoring Systems for Ill-Defined Domains, the 8th Conference on ITS*. Nov. 27, 2020, pp. 11–19.

[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] *Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2015); Deutsche und Englische Fassung EN ISO 9000:2015*. Deutsches Institut für Normung e.V.

[25] Joachim Schramm et al. “Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System”. In: *International Conference of the Florida Artificial Intelligence Research Society*. Florida, USA, May 2012.

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

[27] 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.

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

[29] 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.

[30] P. Thomas, N. Smith, and Kevin G. Waugh. “Automatic assessment of sequence diagrams”. In: *12th International CAA Conference: Research into e-Assessment*. Loughborough University, UK, 2008.

[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: Loughborough University, Jan. 2005.