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

Master's Thesis

AUTOMATIC FEEDBACK FOR UML MODELING EXERCISES AS AN EXTENSION OF INLOOP

submitted by

Markus Hamann

born 16.August.1987 in Löbau

Matriculation Number: 3490881

Supervisor:

Dr.-Ing. Birgit Demuth

Professor:

Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted 25.February.2020

Contents

1 Introduction

1.1 Motivation

In recent years, e-learning systems have become an important part of normal and university education. The *TU Dresden* has multiple projects that are using these types of systems. One of them is the *INLOOP* [20] project of the *Chair of Software Technology*. It is used in the software engineering beginner course to improve the student's object-oriented programming experience and significantly improved the quality of exam results. In the last years, there is a growing interest to extend this automatic feedback to the other part of this beginner course which is the field of object-oriented modeling, too. The hope is to gain the same improvements in the quality of the students modeling skills. The other goal of the system stems from the wish of many students to have a way to better assess their modeling skills and the quality of their models. One way to achieve this outcome, is to the extent the already existing e-learning system of INLOOP with model correcting and feedback possibilities.

This work tries to extends the existing INLOOP with means of automatic correcting and grading *UML* models of students on the background of the chairs beginner software engineering course. To reach this goal, the systems requirements must be established and existing systems need to be surveyed and categorized, to conclude a method that fits the requirements of this new system. Then, the method must be integrated into the existing architecture of INLOOP. In the second step, a proof-of-concept implementation for the automatic correction and grading of UML analysis class diagrams is given. This implementation is then evaluated against student's solutions and their scores from old exams.

1.2 Research Questions

This thesis establishes multiple research questions:

RQ1 Is there a method that can be used for the automatic correction and grading of student solutions in a beginner software engineering course?

RQ2 How could a design for the architecture and workflow of the automatic assessment system look like?

RQ3 How can the system be implemented in the existing INLOOP architecture, without changes to the architecture?

RQ4 Which data can be extracted from the existing data sets and workflows? How can this data be reused to help the design process?

RQ5 Is the presented realization suitable for the goals of the system?

The first research question (RQ1) is needed to find the best method for the needs of a beginner software engineering course. It is answered by giving a literature survey of the existing methods and systems for automatic model correction and rate them against the requirement of the course. Answering the second research question (RQ2) should describe the architecture, workflow, and parts of the system around the chosen correction method, while RQ3 shows the integration of the parts in the existing INLOOP architecture and possible conflicts. RQ4 evaluates which data sets are already existing and how they can be used to generate input and rules for the new system. At last, the usefulness of the example implementation needs to be evaluated to answer the last research question (RQ5).

2 Related Work

As a first step, the related work needs to be viewed and understood. Since this work builds onto the *INLOOP* [20] system, this system must be looked into first. Then a literature survey on model correcting and feedback systems and methods must be conducted to decide which way the extension needs to be designed.

2.1 INLOOP

The goal of this work is to extend the already existing INLOOP with automatic model correction and grading possibilities. For this, an overview of the systems architecture and workflow is essential to design the components and if necessary extensions.

INLOOP [20] stands for *INteractive Learning center for Object-Oriented Programming*. It is a web-based e-learning system that was created by the *Chair of Software Technologies* of the *Technische Universität Dresden* (TU Dresden) and is also maintained by this chair [21]. It is currently used to support the beginner software engineering course with additional practical programming tasks. The programming language currently used for the tasks is Java and the testing engine for the solutions is *JUnit*. Tasks solved by the students locally can be uploaded as solution source files and are automatically tested against pre-generated test cases. At the moment tasks can be categorized in beginner, exercise and exam categories. All tasks can be provided with a start and end date automatic publishing and closing them over the course of a term. To motivate students for exam categorized tasks bonus points for the upcoming exam are rewarded. To minimize fraud, at the end of the term an automatic plagiarism check is used on all exam solutions.

The architecture of INLOOP is depicted in Figure 1 and can roughly be separated into four individual components [20].

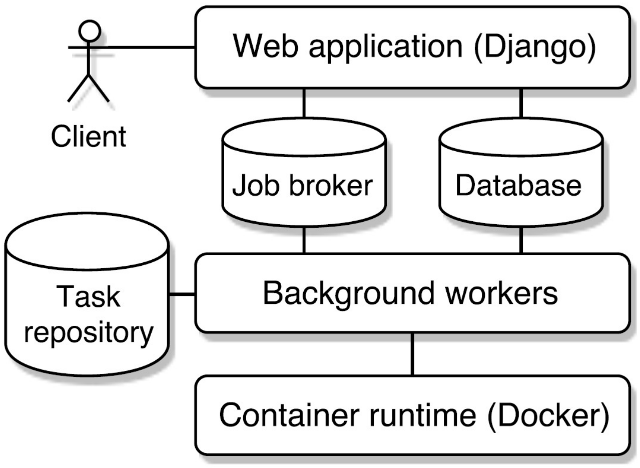


Figure 1: The architecture overview of the INLOOP system [20].

The first layer is the *Web Application* which is implemented in *Python.* It connects the web-based browser clients with the rest of the architecture which runs on a private network. It ships the tasks and associated information and artifacts from the task repository as generated HTML-websites to the student's browser. For the solution, file import functionality is provided and imported solutions can be viewed. For test results, the test engine outputs are transformed into HTML fragments and integrated into the user interface. As to the date of the work, an online editor is provided, but no complex integrated web-programming environment (IDE). Students often need to program the solution offline on their machine and IDE and later test their work online. So often no immediate feedback is possible. This is negated by allowing the students to download a test set for their task.

The second component is the job queue and the *Background Workers.* These workers test the student's solutions with a test engine against the test cases of the task defined in the task repository. There can be multiple workers that work in parallel. Each solution is tested through a worker in a *Docker* instance separated from anything else. The used test engine, Docker and ant run configurations, and other inputs are defined in the *Task Repository.*

The first and second components are connected by the *Job Broker* and a persistent *Database.* The task of this connecting layer is first to outsource the testing of the solutions to the Background Worker containers and second to provide data to the test workers and persist user, solutions, and test results.

The last internal component is the *Task Repository.* It persists the multiple data sets of the defined tasks. Globally, it holds configuration data for the workers as well as the used test engines. For every task itself, it contains the descriptions and attachments that are delivered as HTML to the students. Also, it contains test cases for every task. This repository is designed as a *Git* repository to take advantage of the features of version control systems, like peer-reviewed changes, branches, change history, roll-backs, and commit-triggered scrips. This type of repository is also the core of the INLOOP workflow that is described next.

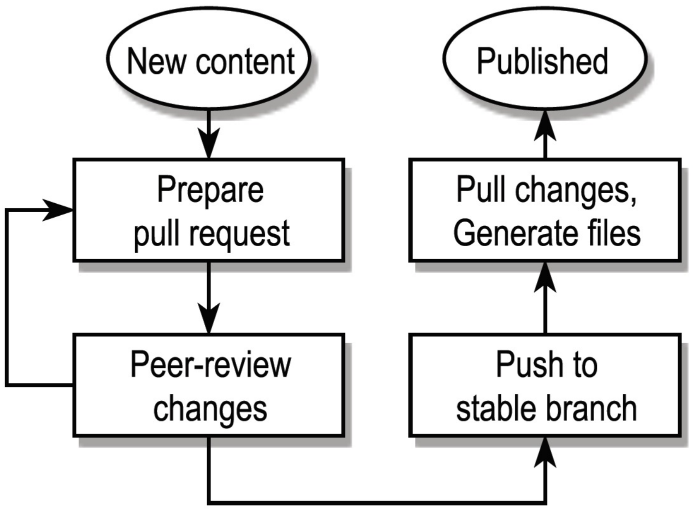


Figure 2: The Workflow of the publishing or updating of task files in INLOOP (*continuous publishing*) [20].

An important part of INLOOP is the workflow of task generation and updating [20]. This workflow is called *continuous publishing* (Figure 2) by its creators. It uses the version control of the Git task repository to streamline and reduce errors in the task generation and updating process. New content can be prepared on the instructor's machine and/or a development branch and can then be committed to the Git repository. In this process, through a pull request, a peer-review can be completed. After the push on the publishing branch, INLOOP is notified by Git. It then syncs with the repository and publishes the task. At the same time, the retrieved artifacts can be transformed or generated.

2.2 Systems for Automatic Modeling Feedback

In this subsection, several systems or methods that are dealing with automatic correction of modeling assignments are surveyed. The literature search was conducted by a keyword search in the most popular scientific catalogs like *Google Scholar*, *ACM* and *IEEE*. For Example, keywords combined to be used in the search were "automatic", "automated" "feedback", "grading","UML","class diagram", "model", "exercises", "assessment", "e-learning" and similar ones. One base of the found literature, a reference-based search was conducted on the newer works to find ground-laying older literature. As a last step, work that features the same systems was combined or partially put aside. To manage the scope of surveyed literature during the process a thematic limit was enforced. Only systems that used *UML*-based models were included. There were a few exceptions to that rule. The first was more general approaches that mention or used UML models as examples. The second was some systems with *Entity-Relationship* (ER) models because of the similarities to UML class diagrams. The second limit was that only systems or methods were allowed that correct a student model against an expert solution or task description. Meaning no systems that only check on correctness of the model against its metamodel are included. Also, no systems that only check against modeling standards or conventions like anti-patterns [18] are included, too.

In general, it can be mentioned that there is not that much unique literature on the topic of automatic feedback of model assessment. This is also often mentioned in the literature itself. But, in the last years the interest in the topic became noticeably larger. Two causes of the increased interests can be growing student numbers and the prominence of web technologies.

Table 1 gives a complete overview of all found systems or methods sorted decreasing with the publication date. This was done to show the more current trends. Not all literature was included in this table since often more than one work was published per system or method without adding crucial new information. In other cases, multiple pieces of literature, that supplement each other, were combined in one row of Table 1. In the remainder of this chapter, a row in Table 1 is called *Entry*. As a critic, it must be said that there is the possibility that are more systems or methods than what is sown there. This set of systems and methods should give a good overview of the field, nonetheless. That is because many projects that are listed here have a similar or nearly identical way of solving the problem of automatic correction of modeling assessments. In the next sections, all important criteria are explained and discussed.

Table 1: An overview of model assessment systems that can be found in the literature. The table is sorted from the newest entry to the oldest one.

Paper	Type		Diagramtype	Editor		Metamodel	Methodologies											Feedback	Grades
							Labels		Source			Matching					Learning		
									Expertmodel										
	System	Method		Tool	Web		Forced	Matched	Single	Multiple	Constraints	Element	Constraints	Graph	Similarity	Execute			
Bian 2019 [6]		x	CD	x		(EMF)UML, Grades	x	x	x			x	x					x	x
Vachharajani 2019, 2014 [33, 34]		x	UC	x		UML		x	x			x		x				x	
Bernius 2019, Krusche 2018 [5, 14]	x		general		x	UML				x							x	x	x
Beck 2015 [4]	x		CD/AD		x	UML	x				x					x		x	x
Sousa 2015 [26]		x	general/AD	x		Graph, UML			x					x	x				x
Striewe 2014 [28]		x	AD			UML					x					x			x
Smith 2013, Thomas 2007 [24, 30]	x		general/ER	x		Graph, ER		x	x						x				x
Schramm 2012 [23]	x		CD/AD		x	(Argo)UML	x		x			x	x					x	
Prados 2011 [22]	x		general		x	Graph				x				x			x	x	x
Hasker 2011 [8]	x		CD	x		(RR)UML	x		x			x	x					x	
Striewe 2011 [27]		x	general			UML	x				x		x	x				x	
Soler 2010 [25]	x		CD		x	(text)UML	x			x		x						x	
Demuth 2009 [7]		x	CD	x		(EMF)UML	x		x		x	x	x					x	
Jayal 2009 [12]		x	AD			UML		x	x			x							
Thomas 2008 [31]	x		SD	x		UML			x						x				x
Baghaei 2007 [3]	x		CD		x	UML	x		x		x	x	x					x	
Ali 2007 [1, 2]		x	CD	x		(RR)UML	x		x			x						x	
Le 2006 [16]		x	CD	x		(Argo)UML	x		x		x	x	x					x	
Higgins 2006, 2002 [9, 10]	x		general	x		ER/OO	x				x		x					x	x
Tselonis 2005 [32]	x		genaral	x		Graph			x					x	x			x	x
Hoggarth 1998 [11]	x		general	x		CASE	x		x			x						x	

2.2.1 Technical Criteria

The first criterion is the *Type* of work that is surveyed. First, there are *Methods*. These are mostly ideas or prototype implementations of automatic correction systems without a working outer framework to connect it to a larger student body [2, 6, 7, 12, 16, 26–28, 33]. But these can have multiple tools to support them. Sometimes, the input of both the expert and the student model is done by an instructor [6]. The second type of entries is *Systems*. These have a framework described that allows them to work automatically with many users and give immediate feedback [3, 4, 8, 10, 11, 14, 22–25, 31, 32]. This criterion is very loose and an entry can often sit between both types. This criterion should only be used to show the relative range of a system.

The next criterion is the *Editor* that an entry uses for model creation. The feedback generator is in almost all cases a different program and is not meant in this criteria. There a two options used in the literature: A *Local Tool* or a *Web-based Online Editor*. The local tool means that the student works on his solution with an existing modeling tool like *IBM Rational Rose*, *ArgoUML*[1], *Dia*[2] or an exclusive tool [7, 8, 12, 16, 27]. There are cases, where only the instructor has this tool to digitized student solutions [6, 12]. This is often the case in method typed entries. These local tool is often used for specific local course exercises. That type of editor often leads to the correction of handed in student assessments without immediate feedback. If online capabilities are needed this type of editor is paired with an online file upload system [27]. This can also be used to give immediate feedback to a student. If system typed entries use a local tool this file upload or a editor with online capabilities is used [9, 27, 32]. Web-based online editors are used by system entries to give immediate feedback to a large student body [4, 10, 15, 22, 23]. They became more important due to the prominence of online technologies in the last ten years.

2.2.2 Model Criteria

The third and fourth criteria are the *Models* and *Metamodels* the entries are designed for. The metamodel is almost always UML since this was one of the limits for the literature research. If special implementations of UML are mentioned they are also given in the table. For example, some entries are using IBM Rational Rose (RR) [2, 8] which saves the models in a textual form or using their own textual UML annotation (textual) [25]. Other special implementations are ArgoUML (Argo) [16, 23] and the *Eclipse Modeling Framework*[3] (EMF) [6, 7] which save the models in *XML Meta-data Interchange* (XMI) format. The EMF UML implementation has the advantage that many modeling tools and editors support its format. Some special entries have different metamodels. *Bian et al.* [6] has an second model for *Grades*. *Higgins et al.* [10] and *Smith et al.* [24] support *ER models* which are similar to UML class diagrams and some entries using general *Graph* [22, 32] structures with UML as examples. *Higgins et al.* [9] also using the more general term of *object-oriented* models (OO). *Hoggarth et al.* [11] is often used as a starting work and describes a general approach to compare *CASE* models for better feedback in courses with many students.

[1]http://argouml.tigris.org, 10.02.2020
[2]http://dia-installer.de, 10.02.2020
[3]https://www.eclipse.org/modeling/emf, 10.02.2020

The majority of entries only support only one or two UML model types. The most supported type is the *class diagram* (CD) with nearly half of the entries specifically mention it [2–4, 6–8, 16, 23, 25]. This can be due to the fact that class diagrams are often the first diagram type which is introduced in beginner modeling courses and one that can be the most complex. So there is a great need in tools to help the student to assesses their skill and for the instructor to save time correcting the models. After the class diagram, the *activity diagram* (AD) is the second most often mentioned UML model [4, 12, 23, 26, 28]. It represents the efforts to automatically correct behavior-based UML models. Other UML model types are scarcely found in the literature. Only one unique entry that mentions UML *sequence models* (SD) [31] and one for UML *use case models* (UC) [33] were found. But there are eight entries that describe methods to correct (UML) models in *general* [10, 11, 14, 22, 24, 26, 27, 32]. These often use transformation techniques to transform the UML models into graphs, or constraint sets or use similarity techniques [22, 24, 26, 27].

2.2.3 Methodologies

The *methodologies criteria* can be subdivided into three sub-criteria. The most important of these sub-criteria is the *matching technique*. Supportive of the matching are the *source* files, *learning* capabilities, and the handling of the model elements labels. Based on the literature, the used matching techniques could be summarized in five categories. These categories are not exclusive. An entry can use techniques of multiple categories. The five categories are:

Element Matching:
An algorithm tries to match one element of the student solution model directly to an element of the model of the expert solution. This can be one the base of labels or sub-elements and is the simplest form of matching technique [1, 2]. This technique is the simplest one to implement but can of its one only reach limited results and have many requirements on the used models like nearly identical labels and structures.

Constraint Matching:
Rules or constraints are defined and the student solution is matched against those. These constraints can be rules used by a verification engine or they can be of algorithmic nature. This technique is one of the broader approaches and can yield a good result with limited requirements but is very dependent on the quality of its constraints.

Graph Techniques:
Different Graph techniques are used on the student model and the expert solution. Examples for these techniques are the transformation in a general graph [27], graph matching [22], and graph rewriting [26]. It uses a very general approach and can work on most UML diagrams.

Similarity Techniques:

There, the goal is to obtain an abstract value of how similar the student and the expert model are. This happens by using a means to calculate how similar an element or an element group is to another element or group. This is often done by a heuristic approach like converting labels, relationships and attributes of an element in a string and calculate a number of how similar the strings are [32]. The matching of elements is done by maximizing the similarity. Based on the similarity algorithm it can be used on all UML diagrams. A disadvantage of this technique is its dependency on the similarity thresholds which can be hard to locate. *Thomas et al.*, for example, tries to generate them by using training sets and machine learning techniques [30, 31].

Execution:

For this technique, the student model is transformed into an executable state and is then tested. This technique is only used with behavior-based UML models like activity models [4, 28]. One example is the conversion into Java code and the usage of a testing environment [4]. Another example is a token-based execution through a reasoner [28].

All techniques found in literature could be categorized in one of those five technique categories. Also, all of these techniques have been proven to work in automatic model assessment systems through their literature. Which categories were used were often decided on the requirements of the entries. After looking through the literature a rough categorization of the categories can be made. Each category of techniques is categorized in an abstract *Generality* level. Higher Generality means that a class has lower requirements on the expert and student model and can even be used on multiple model types. It should be said that a low Generality level does not mean that a category is of lower quality. It only means that it has higher requirements to work and often lower Generality categories are easier to implement like Element Matching. Element Matching and Execution should have the lowest generality. Element Matching can only be used in nearly identical models. Execution, on the other hand, can only be used on behavior-based models like activity, state and sequence diagrams. Constraints Techniques can be of variable Generality based on the rules that are used. If very simple rules are used they are on the level of Element Matching. Many general rules can result in a high Generality. Summarized, constraint-based techniques should be categorized as middle Generality with high variance based on the implementation. Graph and Similarity techniques are categories with high Generality because they use very general concepts. Figure 3 was created to visualize the generality of the technique categories. Lastly, it should be mentioned that most entries using more than one category of techniques to achieve the best result. Common combinations are Element Matching and Constraint Matching, and Graph and Similarity Matching. In the first combination, the simple approach of Element Matching is supported by constraints that allow the matching of elements of the student solution that are different from the expert solution [3, 6–8, 16, 23]. The second combinations are graph and similarity techniques. There the model is transformed in a graph to generalize the elements before the similarity algorithm is activated [26, 32].

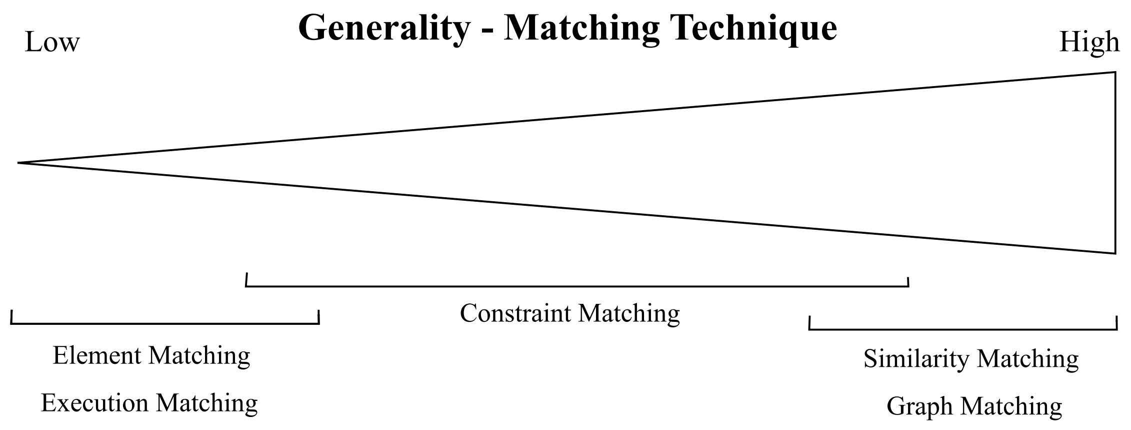


Figure 3: Overview of the *Generality* of the matching techniques. Generality describes the needed requirements of the techniques. High Generality means less requirements on the matched models or model types.

Besides the matching techniques, there are multiple other criteria the entries are using. These criteria are supporting ones that are used in combination with the matching techniques to improve the results of the systems.

The first supportive criterion is the evaluation of the model element *Labels*. The majority of entries choose to *force labels*. This means the labels the students can use are defined in the description of the task. This is often used by entries that are using Element Matching or Constraint Techniques to simplify the process. It should be noted that simple techniques on labels like lower-casing or trimming are also included in this category. This form of label evaluation has the advantages of reducing the solution space and is often used in exams of beginner software engineering course. This simplifies the correction, and the reduced complexity can help some students to solve the task. The disadvantage is reduced freedom in solving the task that can hinder some students in solving the task in their own way. The second label evaluation technique is a complex *matching of labels*. This means that there are algorithms that used language processing techniques to match the labels even if they are not identical. That can be a rather simple algorithm like the *Levenshtein distance* [17] that checks the number of changes to transform a label into the other label [6]. Or it can be a multi-layer pipeline that is using stemming and dictionary techniques to find the meaning behind the label to compare this to the other label [12, 33]. These algorithms are often necessary if an element matching or constraint-based system does not want to force labels. The advantages of label matching are that it gives the student more freedom in his modeling choices and can drastically increase the scope of possible solution matches of the whole system. The disadvantage is the increased complexity of the matching system.

The second supportive criterion is the *expert source data* that must be provided to the system or method to work. The first data that is often provided, is an *Expert Model* of the solution. It is used as the foundation for the direct matching of the student model. Almost all entries are using an expert model as source data for their systems or methods. Some entries are using *Multiple Expert Models*. This is done to broaden the space of possible student solutions without changing the complexity of the matching algorithm. For the feedback, the expert model with the highest similarity or score to the student solution is used [22, 25]. None of the entries use the multiple models simultaneously to assess the student solution. The last source of data that is provided, are *Constraint Definitions*. These hold the rules and constraints that are used by a verification engine. This input data is often used to complement an expert model in the matching process [3, 7, 16]. Some entries are using only global or task-specific constraint definitions for the matching process, too [4, 27]. Constraints that are only statically included in algorithms of the matching engine are not included in this subcategory [6].

The last supportive criterion is the notion that a system can *learn*. Learning means that a system uses previous results to find suitable feedback for the current student solution. There are only two entries that describe learning systems and both use different ways to implement this criterion. The first entry uses multiple expert solutions as sources and checks them with the student solutions with graph matching techniques [22]. If no suitable match is found an instructor manually assess the student solution and inserts that solution with feedback as a new expert solution. The second entry uses semi-automatic machine learning to assess the student solution [14, 15]. To best of our knowledge, there is no literature that specifies how it is done in reality. There is only literature on how a text assessment is done with the same system [5]. From that, it is possible that an instructor must do a manual assessment if the machine learning algorithm is not confident enough. Then the manual assessed student solution is used to train the machine learning algorithm. The advantages of learning systems are that even a simple matching algorithm can give very good results if it learned enough. The disadvantage is that these types of systems need a good learning data set. Generating these sets needs enough sample data and if the quality is not good enough the quality of the assessment can be lower than by other techniques.

2.2.4 Feedback Criteria

Without surprise, all surveyed automatic assessment system generates a form of feedback for the student or instructor. The first form of feedback is the *textual Feedback* on the student solution model. All forms of matching techniques are capable of giving this type of feedback. This form of feedback is mostly given on found elements to help the student finding errors and explaining already found solution parts. Examples of this type of feedback are:

- "City: 2.0/2.0, matches with Class City" [6]
- "Incorrect role name for role crossing from Street" [8]
- "Class Route should be associated with a different class than Street" [8]
- "Multiplicity of association named ... is not correct" [25]
- "There is/are <number of elements> <name of element>, this is too much." [23]

To avoid revealing the solution, feedback on missing parts is rarely given and if, then in a more general way.

- "Note! <number of missing element(s)> class(es) or attribute(s) missing." [23]

The second type of feedback is to calculate a *Grade* or point value (40/100 Points) for the student solution. This type of feedback is not so common like the textual feedback since a differentiation between correct, partially correct, and incorrect solutions must be possible to calculate a sound grade for a solution. Constraint-based techniques are capable of this if there are constraints for each solution type and each constraint is given a point value [6, 27]. The entries that can learn are using the assessment grade of an instructor on old solutions to calculate a grade for a new solution [22]. Similarity techniques are using the similarity value of the elements or element groups to convert this value to a point value [24, 26, 30, 31].

2.2.5 Summary

Through all entries that are surveyed have sightly different approaches and implementations, most of them can be abstracted into one of six types of *Assessment Systems* (AS), each using a different main approach. These main approaches are based on the already discussed matching techniques and supportive criteria and each entry was assessed of which criteria were their defining ones. After this assessment, only six main approaches remained, which can be declared as the abstract assessment system types that are currently used in model assessments. It should be noted that each entry can have other matching or support criteria outside their defining ones. The system types can be seen as a general overview of the topic and are later used to decide on a system type for the extension of INLOOP. These systems, ranked after the number of included entries, are shown in Table 2 and are also categorized in generality levels in Figure 4.

Table 2: Summary of the types of automatic model assessment systems that can be found in the literature. The table is sorted by the number of entries that belong to a type.

Assessment System	Feedback		Generality	Entry Count	Entries
	Textual	Grades			
Constraint-based Assessment Systems	x	x	Middle	8	Bian 2019 [6], Schramm 2012 [23], Hasker 2011 [8], Striewe 2011 [27], Demuth 2009 [7], Baghaei 2007 [3], Le 2006 [16], Higgins 2002 [9]
Similarity-based Assessment Systems	x	x	High	3	Sousa 2015 [26], Smith 2013 [24, 30], Thomas 2008 [31]
Element-based Assessment Systems	x		Low	3	Soler 2010 [25], Ali 2007 [1, 2], Hoggarth 1998 [11]
Label-based Assessment Systems	x		Middle	2	Vachharajani 2019 [33, 34], Jayal 2009 [12]
Learning-based Assessment Systems	x	x	High	2	Bernius/Krusche 2019/2018 [5, 14], Prados 2011 [22]
Execution-based Assessment Systems	x	x	Low	2	Beck 2015 [4], Striewe 2014 [28]

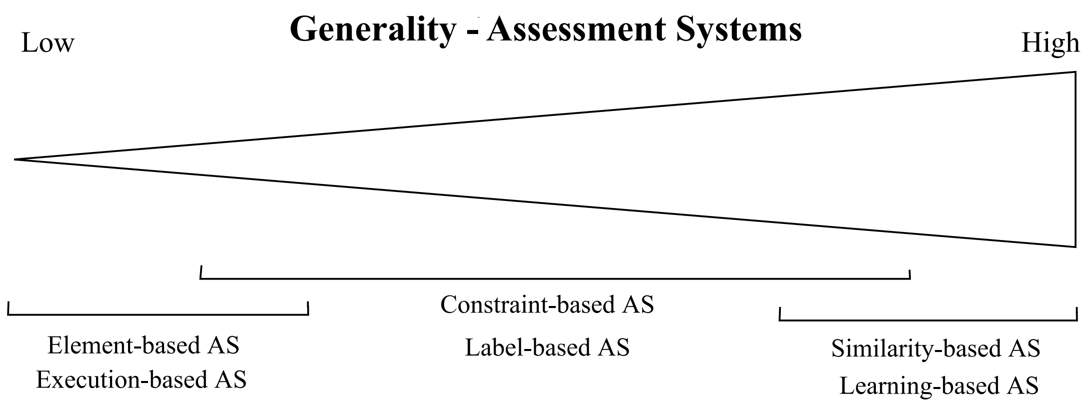


Figure 4: Overview of the *Generality* of the assessment system (AS) groups. Generality describes the needed requirements of the system groups. High Generality means less requirements on the matched models or model types.

3 INLOOM - A Constraint-based Modeling Assessment System

In this chapter, a possible design for the extension of *INLOOP* [20] with model assessment capabilities is discussed. Later in this work, this extension will be called *INLOOM* (*INteractive Learning center for Object-Oriented Modeling*). For this, at first, the general requirements of the future system will be specified. Then possible types of exercises are analyzed and a plan for the future implementation is developed. Last a possible implementation design is created and integrated into the INLOOP architecture. This design includes the user interface for the student and instructor, possible feedback and grading options, and the selection of the used matching technique.

3.1 General Requirements

The first task designing a model assessment extension for INLOOP is to specify the requirements (RQ) and the limitation of the future system.

The first requirement specified out of the main goal of the system. The goal is to prepare the student for the exam in the beginner software engineering course. That should be achieved by giving them modeling assessments that are based on the tasks of the exams. The system should allow to do this. There are multiple ways to achieve this. First, the types of exercises should be similar to the ones that are used in the exam as well as in the exercise lessons of the course. This means, that the task descriptions need to include all keywords for the model highlighted. The students are only to use these keywords when modeling the solution. The next limitation is the metamodel used in the course. In the analysis phase of the software development cycle, the course uses a revised metamodel of the *UML* metamodel called *Analysis UML* (aUML). This metamodel removes more technical elements from the UML metamodel and simplifying it in the process. The system must be able to use these simplified metamodels as well as the normal UML models for the design phase.

(RQ1)
The system should prepare the students for the exam of the beginner software engineering course.

(RQ1.1)
The system should use exercise types that are used in the exam and exercise lessons.

(RQ1.2)
The system should use forced keywords for the exercises.

(RQ1.3)
The system must use aUML and UML metamodels.

The second requirement requires the system to give feedback to the student without giving the solution away. For this, the system should give three types of feedback. One, a textual feedback message to the student that allows him to evaluate his solution. Second, a grade described by a maximal and a received score, or a similar measurement, should be calculated. And third, a complete report for the instructor to save for later examination or evaluation. This report could also be used partially for the feedback of the student like marking the right or wrong elements in his editor.

(RQ2)
The system should give appropriate feedback to the students and instructors.

- **(RQ2.1)**
  The system must give textual feedback to the student for his submitted solution.
- **(RQ2.2)**
  The system should give a grade to the student for his submitted solution.
- **(RQ2.3)**
  The system must give a complete matching report to the instructor for later evaluation.

The last requirement is that the system must be compatible with the architecture of INLOOP. This means that the system must be able to work with the *continuous publishing* paradigm. The student should be able to upload his or her solution over a web interface. The instructor must be able to prepare his task and solution easily on his one local machine and publish it over a git pull request without needing to re-implement any engine algorithm. All needed artifacts should be able to be generated by importing it into INLOOP.

(RQ3)
The system must be compatible with the architecture of INLOOP.

- **(RQ3.1)**
  The system should implement continuous publishing.
- **(RQ3.2)**
  The system must have a web interface for the students to receive their tasks and upload solutions.
- **(RQ3.3)**
  The system must give the instructor the ability to upload his task descriptions and solutions with a Git commit.

3.2 Types of Modeling Tasks

To decide the solution for the requirement RQ1.1 a survey on the exam tasks of the last 11 years was conducted. The time span of 11 years was decided one the fact that in this span the exam was roughly the same style as today. In the end, 21 exams of the summer and winter terms were surveyed after their modeling task. In all 21 exams, a number of 81 modeling tasks can be found. This data could be interesting for other works and the planning of future exams. In this work findings important to the topic of assessment systems are discussed. The whole data set can be found in the digital attachments this work. Table 3 gives an overview of the data this section is based of.

Table 3: Overview of the modeling tasks of the last 21 exams. Overall 81 task were surveyed. Each task was categorized by its *Phase* in the software development cycle, their diagram *Type*, and their *Difficulty*. The difficulty[4] increases from *Diff 1* to *Diff 3*. *Count* summarize all task of the same type and "%" shows their percentage value to the complete number of modeling tasks.

Phase	Type	Diagram	Difficulty			Count	%
			Diff 1	Diff 2	Diff 3		
OOA	Structure	Class	9	6	16	31	38,3
		Object	1		2	3	3,7
	Behavior	Sequence				0	0,0
		Activity		1		1	1,2
		State (B)	1	2	3	6	7,4
		State (P)		1	4	5	6,2
	System	Context			1	1	1,2
		TLA			1	1	1,2
		Use Case		1	3	4	4,9
OOD	Structure	Class	7	3	4	14	17,3
		Object			4	4	4,9
	Behavior	Sequence		3	5	8	9,9
		Activity					
		State (B)	1		1	2	2,5
		State (P)			1	1	1,2

3.2.1 Model Generation and Transformation

The first finding of the survey was that there are two types of modeling tasks. One of them is the *Model Generation* task type. There only a textual description of the task is given and the student must model the desired diagram type. There are 56 tasks of this type which is 69% of all modeling tasks. The other task type is the *Model Transformation* task type. This type gives the student a source model of one type and a description. The student must create a solution diagram of another type. 25 Tasks (31%) of all tasks are of this type. For the future assessment system, both types of tasks should be implemented but the focus should be lying on the model generation task type.

[4] See Section 3.2.3 on Page 19

3.2.2 Object-oriented Analysis and Design

The next finding is the phase of the software development cycle. Without surprise, all tasks are in the *Object-oriented Analysis* (OOA) or in the *Object-oriented Design* (OOD) phase. 52 tasks lie in the object-oriented analysis phase (64%) and 29 in the object-oriented design phase (36%).

The object-oriented analysis phase includes nine diagrams in three diagram types. All but one (Sequence diagram) are used in exam tasks.

Structure Diagrams:

- Class Diagram
- Object Diagram

Behavior Diagrams:

- Sequence Diagram
- Activity Diagram
- Behavior State Diagram (State(B))
- Protocol State Diagram (State(P))

System Diagrams:

- Context Diagram
- Top Level Architecture Diagram (TLA)
- Use Case Diagram

The object-oriented design phase includes six diagrams in two diagram types. From these six diagrams, five are used in exam tasks with the exception of the activity diagram.

Structure Diagrams:

- Class Diagram
- Object Diagram

Behavior Diagrams:

- Sequence Diagram
- Activity Diagram
- Behavior State Diagram (State(B))
- Protocol State Diagram (State(P))

From this, it can be concluded that both phases should be included in the assessment system but the focus should lie on the object-oriented analyses tasks since both the count of tasks and the number of diagram types are higher. Another fact that supports this decision is that the exercise lessons of the course also favor the object-oriented analysis phase.

3.2.3 Difficulty Levels

In the process of surveying the exam tasks, a pattern in the *Difficulty* of exam modeling tasks was found. It can be seen that the task followed one of three types of difficulty levels. These three levels can also be seen in the exercise lessons of the course. In the first level of difficulty (*Diff 1*), most of the model is already prepared for the student. The student is then tasked to add some additions to the existing model. In this type, the student's choices are usually very limited and the existing model can help to find the solution, often making this type of task the easiest ones. The second difficulty level (*Diff 2*) also gives the student a pre-existing model to work with. In contrast to the first difficulty this pre-existing model only holds a few anchor points for the student. Most of the model is to be designed by the student. The difference between the first and second difficulty level can be in times relative unclear. To decide which difficulty a task is from, the scope of the student work and the pre-existing elements can be used. For difficulty one roughly more than three-quarters of the work on the model should be pre-existing. The Distinction is important nonetheless since the two difficulties represent two different concepts. At the first difficulty, the student only *adds* to an existing model. At the second difficulty, the student *creates* the model with some hints or anchor-elements as help. For the highest difficulty three (*Diff 3*) the student must model the complete model him- or herself. This type of task giving the student the highest degree of freedom and the least help. Because of this, it should be the highest difficulty for tasks.

In the future system, tasks for all three difficulty levels should be provided. Level one difficulty should be for beginner categories. Normal or exercise categories could be providing all difficulty levels with a focus on the first two levels. Exam categories should be focusing one the last difficulty.

3.2.4 Implications for Realization

After discussing the findings on the surveyed old exams, new requirements for the system can be concluded. These requirements can be placed below RQ1.1 and must be met to resolve this requirement.

(RQ1.1)
The system should use exercise types that are used in the exam and exercise lessons.

(RQ1.1.1)
The system must be able to exclude pre-existing elements from feedback generation.

(RQ1.1.2)
The system should be able to be used on many different model types without changing the core process.

The differentiation between generation and transformation task does not add any new requirements to the system. Both types of tasks have the same process to resolve them in an assessment. The only point that can be made is that the transformative tasks need a figure of the source model in the task description. This feature is already included in the HTML artifact generation of INLOOP. Differently, the difficulty levels generate a new requirement. To implement these the instructor must be able to mark elements as pre-existing and exclude them from all feedback generation. Lastly, the system should be able to deal with many different diagram types possible without needing new code implementations for all.

Through the great number of model types, there is a high possibility that each model type must be included after each other into INLOOM over a period of time. In this case, it is important to classify an order of importance for the inclusion of the modeling types. Looking at the surveyed data it is clear that the most important model type is the analysis class diagram since nearly 40% of all exam modeling tasks are of this type. It is also the most prominent model type in the exercise lessons of the course. Both points suggest that INLOOM should include this type first. After that other OOA models should be included. Though the design class diagram is the second most occurring model type in exams the object-oriented analysis itself is much more prominent in the exam and the exercise lessons. The most important model types after the analysis class diagram should be the analysis state models followed by the use case diagram. After that, the rest of the analysis diagrams should be included due to the importance of the phase in the course. Then the design diagram should be added beginning with the design class diagram and continuing with the sequence diagram.

That order of implementation has another origin. The aUML models are normally more simple than the *Design UML* (dUML) ones. That should make it easier to automatically assess them and reduces the resources needed to implement an assessment system for them. Because of this, it would be a good choice to include them in INLOOM first.

Design UML models are often more complex then aUML models. Once, there are more model element types that can be used in dUML models. Second, there are often multiple solutions for a task, due to the inclusion of technical design decisions. These can be, for example, used design patterns, or programming languages. Both problems should make it harder to automatically assess them. Because of this, they should be included in INLOOM after the aUML models.

3.3 Model Representation

3.3.1 Selection of the Model Representation

After discussing which modeling exercises the INLOOM system should implement a usable model representation for students solution and possible expert models must be found. A fitting model representation can help to easily integrate the future system in existing systems like databases and user interfaces and can drastically reduce development time for the needed model-driven matching tools.

In the literature of this topic, multiple representations were mentioned though the majority did not discuss the underlying models. Some Systems are using completely original model representations. These have the advantages that they can be implemented to perfectly fit the requirements of the future system. The price for this is that everything from matching systems, user interface tooling to the representation itself must be implemented anew in the process. Through highly customizable the workload included in this approach is very high. In conclusion, this approach should be only used as a last resort since there are multiple alternatives. The first existing representation was *IBM Rational Rose* (RR) models which have an original more textual description of the model. IBM Rational Rose is a populate modeling tool with multiple tools working with it. This can have advantages like support and popularity. Negative points seem to be the closed commercial nature of the products. There also seems to be no tooling to easily generate new model-driven tools for this representation. In conclusion, new tools must be implemented. These disadvantages discourage using this model representation especially since IBM Rational Rose is not used prominently by our course. The second used model representation is the *Eclipse Modeling Framework* (EMF) model. An EMF model is based on an *Ecore* Metamodel and is saved in *XMI* format. EMF has multiple advantages and is used with the Eclipse IDE. First, it is free to use and open source. Second, it has a large pool of tooling choices that can be used to process models or generate new tooling for the models with relatively less expenditure. Third, it can be used outside of the *Eclipse*[5] IDE. The last point is that the XMI format is very structured and widespread, making usage outside of the EMF ecosystem possible. Naturally, the EMF model representation has a few downsides as well. It is highly interlinked with the Eclipse IDE and working outside of this IDE can be more problematic but is very possible nonetheless. Also, not all tools and tool generation tooling are available as a standalone outside of the Eclipse IDE. Before using it a review for each tool is needed. And lastly, there are problems with the EMF model implementation when working with modern databases or web technologies like *JSON*. This can restrict the usage of web services.

As a model representation for the INLOOM system, EMF is chosen. The decision was made mostly because of the model-driven tool ecosystem and the model format and automatic model implementation in *Java*. There is also a *UML2*[6] implementation in Java that can be used with many different external UML modeling programs like *MagicDraw*[7]. This could help for a possible future swap between original EMF models and UML in the tooling if that is needed. The problems with the standalone tools not available are solved in a later section of this chapter[8]. Last, the problems with modern web technologies are already addressed by a couple of helper

[5]https://www.eclipse.org, 10.02.2020
[6]https://wiki.eclipse.org/MDT/UML2, 10.02.2020
[7]https://www.nomagic.com/products/magicdraw, 10.02.2020
[8] See Section 3.4.3 on Page 29

tools like the *emfjson-json* project[9] for web systems or Eclipse *CDO* Model Repository[10] for databases. For databases, it is also possible to store model files as a whole in XMI format. This is enough if no search on the model elements in the database is necessary. In this use case, this should be the case.

3.3.2 Model Definition

After choosing EMF as our model representation the used models themselves must be defined. The first question is the scope of the models. There it is to decide if the more general UML2 metamodel implementation of EMF is used or if a new metamodel is created to fit the requirements more. The already existing UML2 implementation has the advantages that it is already existing and it is used by multiple UML model environments that could be used for the student solution model. The disadvantages are that in the object-oriented analysis of the course only simplified aUML models (see Figure 5) are used. That means that each received UML2 model must first be checked against these simplified rules. These checkers must be created for all nine analysis model types which consume time and resources and can introduce new error sources. The alternative is to create an original metamodel for each of the simplified aUML model types. This is a higher initial expenditure but removes the need for checking for structural unnecessary elements or structural errors. Which our focus of the analysis phase of the software development cycle the second option seems to be the better one. Later for the design phase, the UML2 implementation can be used.

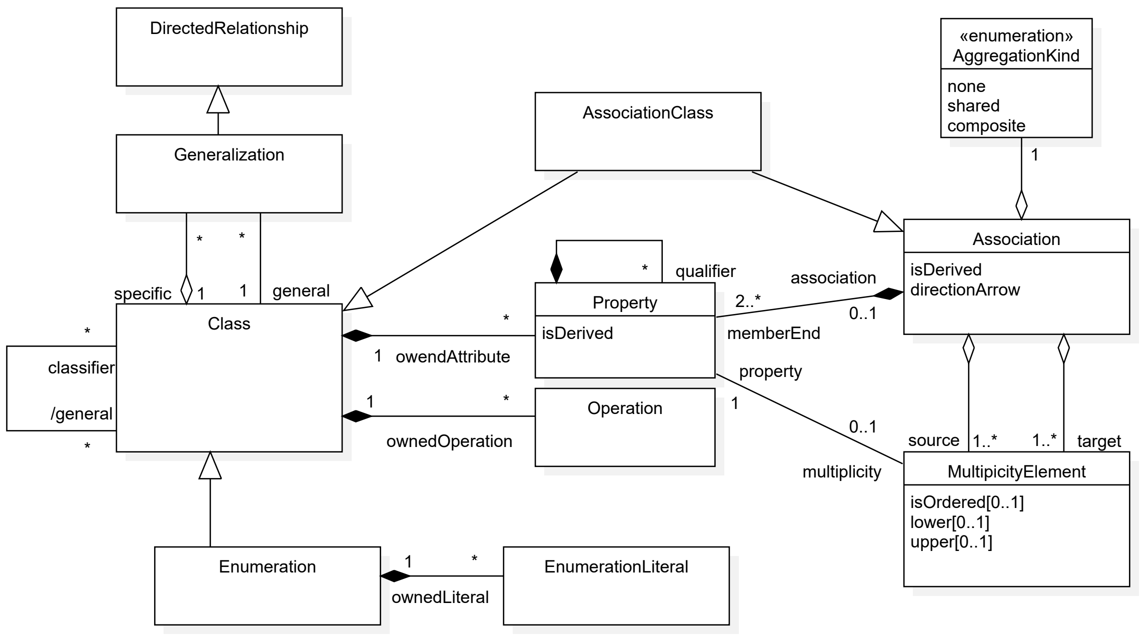


Figure 5: Simplified metamodel of the aUML class diagram used in the beginner software engineering course of the *Chair of Software Technologies* of the *TU Dresden.*

[9]https://emfjson.github.io, 10.02.2020
[10]https://www.eclipse.org/cdo, 10.02.2020

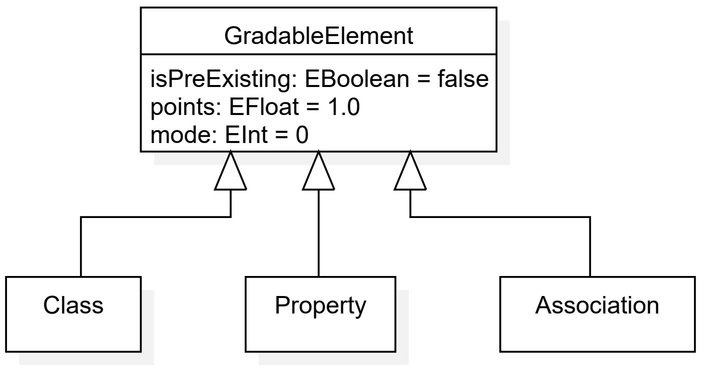


Figure 6: Implementation of the feedback and grade information in the aUML metamodels for INLOOM. The *GradableElement* metaelement hold all these information and other metaelements inherit from it to be included in the feedback generation. *Class*, *Property* and *Association* are examples that stand for these elements.

The next decision is on how to design the information necessary for grades. In the literature one time, the information was saved in its one model, in the expert solution itself or in an external data sheet. All three alternatives are possible for INLOOP. Since it is already decided that we use original metamodels for the analysis phase models, including the grading information into these metamodels can be done without problems and further resources. This reduces inter-model complexity and allows every expert solution to have their own grading scheme. On student solutions, this information can be ignored by the matching engine. Figure 6 shows the implementation of these grading informations in the metamodels. Every element that is included for feedback and grade generation inherit from the *GradableElement* metaclass. With this inheritance, multiple information for feedback and grades can be set. First is the *isPreExisting* option that excludes an element from feedback and grade generation if it is set to true. This allows the system to meet with the requirement RQ1.1.1 and allow tasks with preexisting elements of the difficulty one and two (Diff 1 and Diff 2). Next, a *point* value for the element can be given. This value represents the maximal points that are given to the student if the matching algorithm can match the element. The last attribute is a *mode* value. In the exams, there are often different grading schemes for different elements. The mode attribute is included to allow the matching algorithm to react to different schemes if it is wished for. For better student feedback, a standardized grading scheme should be prioritized.

There were two attributes that were considered but in the end not included in the GradableElement. For completion, these are also mentioned in case of future extensions of the system. First is the *pointDeduct* value. It can specific how many points of the maximal point value will be deducted per error in the student solution element. This was dismissed by the fact that mostly 1 or 0.5 points with 0.5 point reduction were given to an element in the exams so a simple half point scheme was always sufficient. The last not included attribute is a specific *feedback message* for the element. This is not necessary since these messages would not vary enough between elements of the same type. All assessment systems in literature also only give specific feedback based on the element type.

3.4 INLOOM Matching System

The next step to the INLOOM System is the design of the matching system. For this, one of the assessment systems types must be chosen and the chosen system must be designed to work with continuous publishing.

3.4.1 Assessment System Type of INLOOM

Since INLOOM should use tasks similar to the ones used in the exams there are some already mentioned restrictions. First, the labels of the used elements are forced to specific keywords and second the task description is specifically worded to restrict the solution space as much as possible. With this, assessment systems with high generality like *Learning-based AS* and *Similarity-based AS* are not needed to reach the requirements. Also, there is not too much already digitized data on the exam task and solutions. So training the learning systems and calculate similarity thresholds would take many resources and long low-value starting time. Similarity-based AS can also have difficulties with concrete textual feedback for the student and their grades can be hard to retrace which can lead to doubt from the students. These costs will properly not be worth the possible additional gains. An *Execution-based AS* is also not chosen. These work best with behavior-based model types and so do not fulfill requirement RQ1.1.2. *Label-based AS* are not needed as well since the labels are forced so only small label correction techniques like lower-casing or trimming are needed. Later on, better label matching techniques like *Levenshtein* [17] could be added if needed. E*lement-based* and *Constraint-based AS* could both be used under the limitation of the system. A survey of older exam solution shows that even with the limited solution space there are some differences in expert and student solution models so pure element matching is not enough. Also, pure element matching can only give general feedback and rough grades without nuances like half points and error-aware feedback.

The choice at the end of these advantages and disadvantages is a Constraint-based Assessment System. Those systems do not need many resources to start running with sufficient results. They also have a wide range of generality and have been proven to work on many different model types. They can also be calibrated by their constraints on the limitation of the solution space. The last point is the ability to give detailed textual feedback and grades through multiple constraints per element. All these points make them an ideal model assessment system for the existing limitations and requirements.

3.4.2 Matching Algorithm

After deciding on a *Constraint-based Assessment System* (CAS), the algorithm must be designed. In the literature, there are multiple ways to generate and use the constraints.

Sometimes the constraints are implemented as part of an algorithm, or they are designed in files of their own and then only used by the matching algorithm. Since the INLOOM CAS should work with many model types and should not change their workflow or implementation for different model types, the second option seems to be the better one. The second option makes later changes on the constraints easier since the matching algorithm can stay the same, too. This would also be a better fit for the continuous publishing paradigm of INLOOP since the instructor only needs to publish the constraints and do not need to care about the algorithm behind INLOOM.

Next must be discussed how the constraints are created. In literature, one way to do that is that the constraints are designed by the instructor for the task. With this, the instructor can perfectly customize the constraints to the solution domain that he envisions. The disadvantage is that the instructor must have high domain knowledge and must be able to design the constraints in the constraints language. Another way to do it is that the instructor designed an expert solution model and there are global constraints that are pre-designed for all tasks of a model type. In this case, the requirements for the instructor are less than in the first option but the global constraints maybe can not be perfectly matched to the expert solution. Also, both the expert solution and the global constraints must be interwoven at the time of the matching process which generates another layer of complexity.

For the INLOOM CAS a *two-phase process* is proposed that hopefully contains the advantages of both constraint creation options. It also uses the constraints as individual files and keeps the underlying algorithm as clean as possible. The abstract process is described ones based on the activities by Figure 7 and second based on the object or file flow by Figure 8. First, there is the *Preparation Phase.* This phase started only when the instructor publishes a new solution for a specific task. The instructor commits a new model file as an *expert solution.* This file can be created by a provided modeling tool and the instructor must not define any constraints, keeping the requirements for the instructor low. After the upload, the model is automatically given to a *Constraint-based Test Generator.* This generator uses the model to automatically generate a *Constraint-based Test-Set* out of the model and write the constraints into files for later use. Each of these files should contain the constraints for one element of the expert solution. The generator also checks if an element is a gradable element and if the element is pre-existing. For the later, no constraint file is created and for the former, the grading and feedback information is included in the constraints. With this, element-based constraints are generated which should better describe the domain as global constraints but do not need specialized knowledge from the instructor. This process also generates a text-based abstraction layer between the expert solution model and the matching to the student solution model. Such an abstraction layer should make it easier to use the complete process with multiple model types. The generator generates the element-based constraints on the base of a *Master Constraint-Set* for the model type of the expert solution. This master constraint-set is created one time for one model type and can be used on all expert models of the same type. This set can be created by an expert in the constraint domain without the instructor's involvement. In the best case, the master constraint-set includes for each element type of the model type one file that describes a constraint for that element type and when they are included to the element-based constraint-set. For the INLOOM CAS these master constraint-sets are the main quality variable that can be used to customize the quality of the matching and the scope of covert solution space. A disadvantage of working with these master constraint-sets is that these sets' complexity can rise drastically with the count of element types and the possible solution space. This means that tasks on the base of aUML should be inside the parameter of the system since aUML models are often simplified and can have, with a good task description, a small solution space. Normal design UML models could be problematic to describe in the master constraint-sets since they have a much higher complexity as aUML models. They often have more model element types and cover a greater solution space. The greater solution space means that there are multiple possible design solutions for one abstract one. One cause for this are technical design decisions. First, different design patterns could be used. Second, the design solution could be for different programming languages. These languages could have different syntax and limitations. All that is especially the case for UML design class models and can hinder the usage of INLOOM for dUML models.

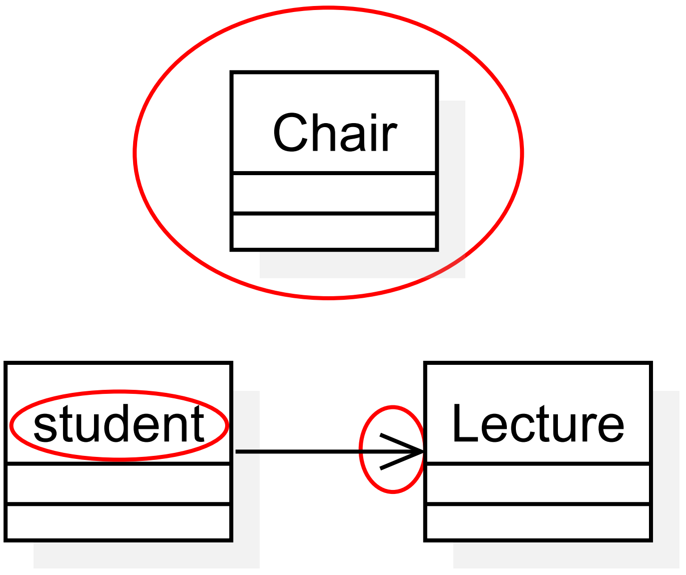


Figure 9: Examples for error found by global constraints (red circles). These include isolated classes, lower case class names, and directed associations in aUML class diagrams.

After the constraint-based test-sets for a task are generated the second phase of the process can be started. Each time a student solution is submitted a *Constraint-based Test Engine* will start matching that solution against the task-specific test set. This phase can be called the *Testing or Matching Phase*. Since the constraint files of this test-set are all equal there is the possibility of using additional *Global Constraint-Sets* (see Figure 9) in the same process. These global constraints can be used for example to check on model conventions like upper-case class names. Another usage of these can be the grading or feedback to model wide issues like isolated classes. After the test, an *Output File* is generated that holds the test results, textual feedback, and grades for the student. There are multiple advantages of using this isolated second phase. First, since only textual constraint files are used global constraints can be easily added to improve the quality of the matching and feedback. Second, the constraint test-sets hold only the important constraints for only the important elements. All filtering is done in the preparation phase, so the second phase should be needing fewer resources. This is important since the matching phase will be triggered by every submitted student solution. The last advantage is that the expert solution itself is not used in the step. This create an abstract layer between the expert and student model. This can lead to more security since the solution is not needed to be in the same space as the tested models. Another point is the flexibility gained by the abstraction. Dependent on the implementation of the Constraint-based Test Engine, theoretical no model type information should be needed. But, this could be hard to achieve using the EMF framework. One point of this flexibility was already mentioned with the additional global constraints but other changes are also possible. For example, it could be possible to change or add element-based constraints to generate multiple expert solutions around a core structure of core element-based constraint test files.

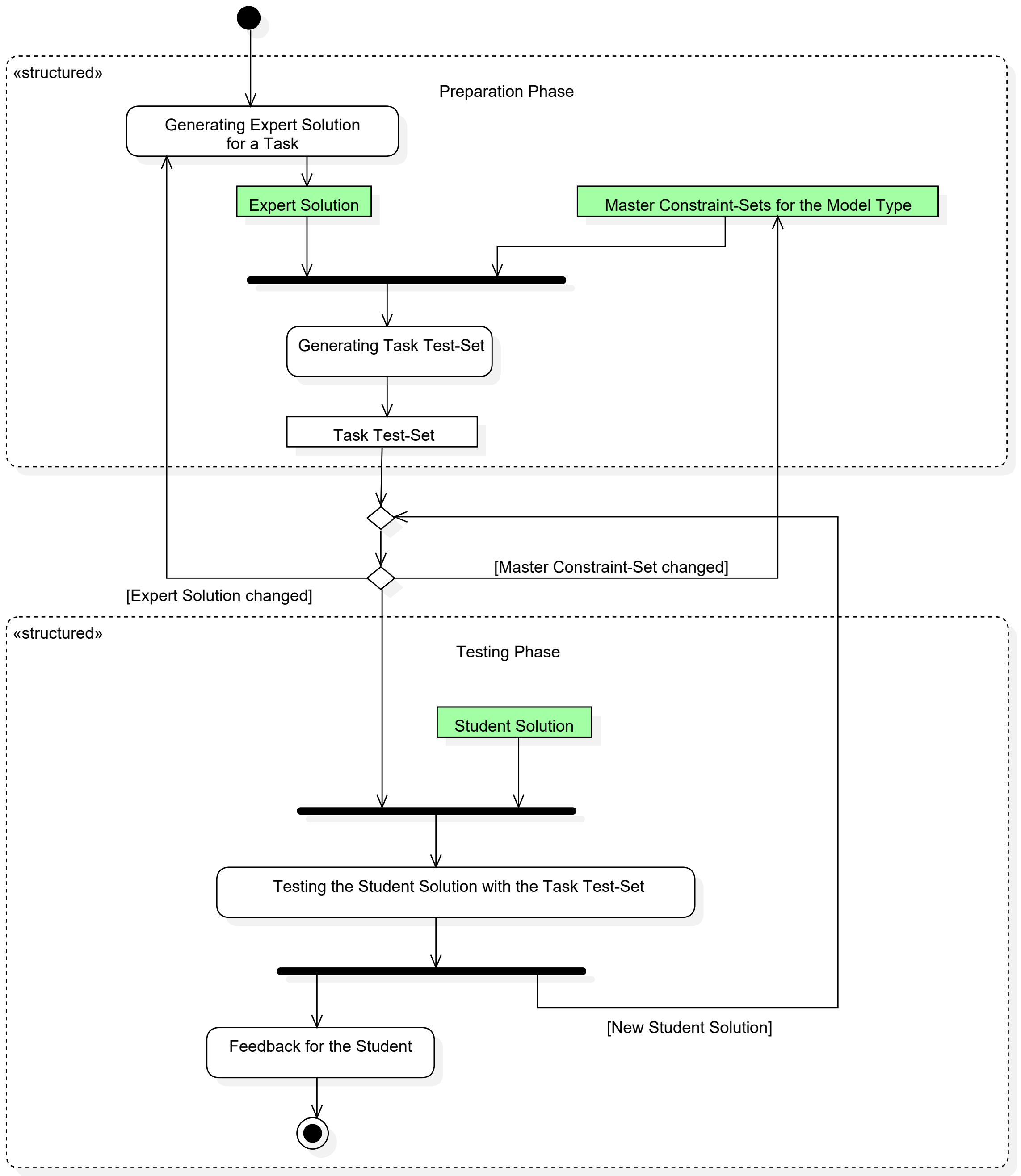

«structured»
Preparation Phase
Generating Expert Solution for a Task
Expert Solution
Master Constraint-Sets for the Model Type
Generating Task Test-Set
Task Test-Set
[Master Constraint-Set changed]
[Expert Solution changed]
«structured»
Testing Phase
Student Solution
Testing the Student Solution with the Task Test-Set
[New Student Solution]
Feedback for the Student

Figure 7: Workflow of the INLOOM matching system. There are two phases. In the *Preparation Phase* an *Constraint-based Test-Set* is generated out of an *Expert Solution Model* for a task and a model type specific *Master Constraint-Set*. In the second *Testing Phase* the Test-Set is used to match a *Student Solution Model*, generating feedback and grades for the student.

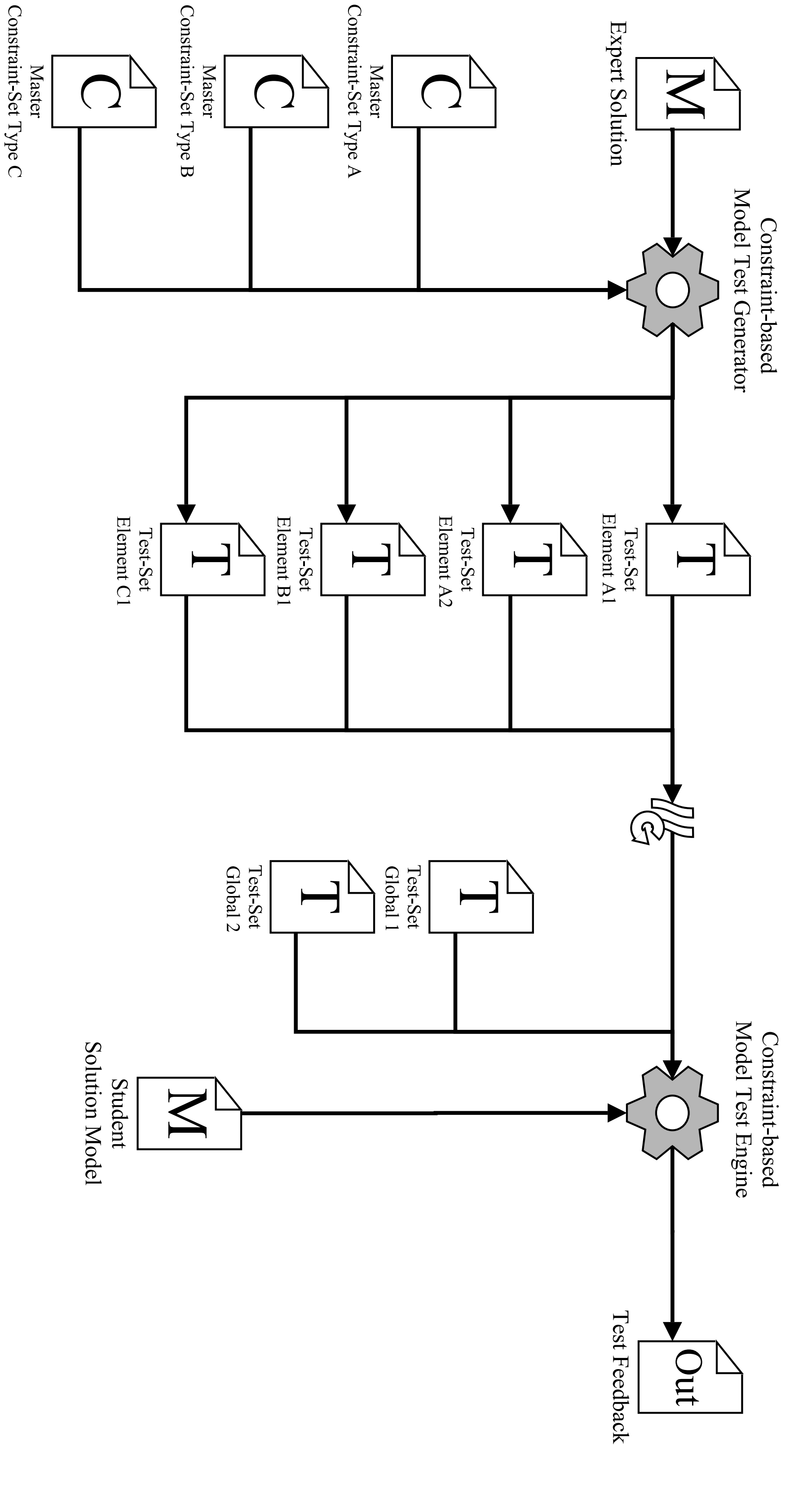


Figure 8: Object- or Fileflow of the INLOOM matching system. There are two phases. In the *Preparation Phase* an *Constraint-based Test-Set* is generated out of an *Expert Solution Model* for a task and a model type specific *Master Constraint-Set*. In the second *Testing Phase* the Test-Set and additional *Global Constraint Test-Sets* are used to match a *Student Solution Model*, generating feedback and grades for the student.

3.4.3 Possible Realization

After describing the abstract concept behind the INLOOM CAS, this section presents a possible realization for such a matching system. This realization was then used in the next chapter to implement a proof-of-concept aUML class diagram matching program.

Since we are using the EMF framework as our model representation it would be optimal to use one of the tools that are already integrated into the framework. This would lessen the time and resources compared to creating an original tool. One of the tools that can do that is the *Epsilon*[11] [13] tool collection. Epsilon comes with multiple tools for model transformation and processing and is running with EMF models like the original aUML models or the EMF UML2 implementation. It also solves the problem of using the EMF models outside of the Eclipse IDE since it provides the tool as a Java library for standalone applications. Epsilon has multiple tools and is based on the *Epsilon Object Language* (EOL) [13, p.25-61] with similar dialects for all tools. It provides Interfaces to Java libraries, too. Two tools of the Epsilon collection are interesting for the realization of INLOOM.

First is the *Epsilon Generation Language* (EGL) [13, p.105-121] engine. This Engine takes a model file, the metamodel Ecore file, and multiple templates to generate at run-time multiple textual output files. The templates contain both text and programmable slots like for- and if-Loops, variable programming, or interface control to Java, the file system and the console. It can be used to generate the constraint-based test files out of master constraint-sets based on these template files and the expert model based on EMF.

The second interesting tool is the *Epsilon Verification Language* (EVL) [13, p.63-82] engine. This tool is a rule- or constraint-based verification engine. It takes multiple rule files, a model file, and the metamodel Ecore file to verify the model at run-time based on the provided rules in the rule text files. It also allows everything the EGL engine allows. If the text file output of the EGL engine is in the EVL rule format the output files can be used in the EVL engine. In the context of INLOOM, it means that the constraint-based test files, that were generated from the EGL engine, can be inserted into the EVL verification engine. Together with a student solution, an output feedback file in XML format can be generated.

Summarized, the are multiple advantages of Epsilon for INLOOM. First, it can implement the INLOOM matching process completely inside its tool collection and language space. Second, it can be used in the form of two small Jar programs outside of eclipse. Naturally, this means that INLOOM must use the Java architecture. And secondly, the engine implementation itself must not be changed. All algorithms come from textual input files that can be changed at run-time. All of these points make Epsilon to a top contender for the implementation of the INLOOM matching engine.

[11] https://www.eclipse.org/epsilon, 10.02.2020

3.5 Integration into the INLOOP Architecture

After designing the INLOOM matching engine, the next step is to integrate it into the INLOOP architecture shown in Figure 1. Another point is the until now overlooked user interfaces for the students and instructors.

3.5.1 Student User Interface

Two user interface types must be looked after. The first is the web interface for students that must be integrated into the *Web Application* component of INLOOP. For the task description part, no changes must be made. The HTML fragments INLOOP created can be used without problems for the model domain description and also for the existing model figures of model transformation tasks. For the solution submission, a few choices must be made.

If the model solution must be created on the student's local machine the file upload of the current INLOOP can be used with some adjustments. In this case, a model tool must be provided for the student to design the EMF-based aUML models. This would be possible through EMF-based editor which could be provided as a downloadable Eclipse plugin. A possible tool to create such an editor plugin could be *Sirius*[12]. A second option would be a local or web-based external modeling tool that could export its models to the EMF/Ecore UML2 implementation. An example of such a tool would be MagicDraw. In this case, a catalog of tools must be maintained and possible licenses must be provided. Additionally, a checker and transformer must be implemented to transform UML2 to aUML models and check for excluded element types. The advantage of this submission option would be that the student can work on the task offline and can with the second option use his favorite modeling tool. Also, the check and transformation into aUML can help to learn the differences between aUML and dUML. There are disadvantages as well. First, with this option, immediate feedback to the student is not possible and there is no way to relieve this by giving out test cases like it is done by INLOOP. Second, the many additional steps to the submission and the result can deter students from using the tool due to their limited time. And the last disadvantage is the easier way to plagiarise compared to the next option since the files could be easily shared between students.

The second submission option is to implement its own web-based modeling editor for the aUML models. Then students could design and upload their solutions directly on the INLOOM website. With this approach, students would get immediate feedback in the editor and no additional transformation or model-checking tools would be needed. Also since the student models directly in INLOOM copy-and-paste plagiarism would be harder to achieve. Another advantage is the possibility to include the feedback generated from the system directly into the editor for better visualization. The greatest disadvantage is that there are no really customizable UML web editors. The used editors must be completely self-implemented. This could be countered by the notion that aUML models are often simpler as normal UML and that there are good graph editor tool sets like *mxGraph*[13] that can be used for the implementation.

[12]https://www.eclipse.org/sirius, 10.02.2020
[13]https://github.com/jgraph/mxgraph, 10.02.2020

For a summary, both submissions options have advantages and disadvantages but with the increasing importance in web technologies today, and the importance of immediate student feedback, the second option of a web-based editor should be prioritized. It should be also mentioned that most of the newer assessment systems for bigger student bodies that are mentioned in the literature are using basic web-based editors for their systems.

The creation and design of this web-based editor would be too much for this thesis and could be considered a perfect future work.

3.5.2 Instructor User Interface

The second user interface is the interface the instructor uses for publishing new tasks. One part of this is the committing process to the task repository. Since the task repository is based on *Git/GitHub* this is already provided. The instructor can commit drafts on a development branch, ask for peer reviews for commits and later publish onto the master branch. The only thing that should be observed strictly is the separation of task commits and master constraint-set commits. Ideally, the task can be uploaded by any instructors with the right rights, while commits for the master constraint-set should only be made by specialized instructors. A strict separation of specific development branches should be enforced.

The second part of the process is the preparation of the published artifacts. Since EMF is used, a prepared Eclipse IDE could be used for this. Eclipse already has Git and *Markdown* support plugins for the task descriptions artifacts. For the design of the expert solution, an EMF modeling tool plugin could be provided. Sirius is for this purpose a lightweight and easy to use graphical editor toolkit for EMF metamodels. An editor plugin made with Sirius could be provided for the instructor's Eclipse instance. Generally, such a prepacked Eclipse IDE with Git, Markdown, and the editors could be provided as ready-to-start packages by download to all instructors.

The same could be done for the special instructors that design the master constraint-set. Their packages could hold the Epsilon plugin and include test data to confirm their constraints. For example, a test package could include digitized expert solutions and student solutions for old exams. Together with a data sheet of the original grades, the new master constraint-set could be tested against these original grades. The new master constraint-set should only be published if its results are within a certain margin to the old grades. This would increase the quality of the master constraint-sets. The packaged Eclipse IDE for the constraint-responsible instructors could include the Epsilon plugin and the model type-based test databases.

3.5.3 Task Repository and Background Workers

This part of the INLOOP architecture is important for the *continuous publishing* paradigm (see Figure 2). The workflow of INLOOM that is integrated into the INLOOP *continuous publishing* workflow is shown in Figure 10. Like already said, the normal instructor only needs to prepare the task description artifacts and an expert model solution for the task. This part fulfills the *New content* step of the continuous publishing paradigm. After committing these artifacts to the *Task Repository*, the *Peer-review changes* and *Push to the stable branch* are the same as in the normal INLOOP. They are normal Git/GitHub provided steps. It should be noted that on the Git repository, only the expert model needs to be present and not the generated test sets. This should make the repository cleaner. Also, the correctness of the generated test sets could be already tested in the *Peer-review changes* step or before in the development branch. In the *Pull changes, Generate files* step INLOOM syncs with the repository. There the constraint-based test sets are generated out of the master constraint-sets and the Constraint-based Test Generator, deployed as a Java jar file. They are then saved in the INLOOM intern space for each task. This step can be accomplished by using the *Makefile* INLOOP provided. The task description is also transformed into HTML fragments for the *Web Application* component. This process is the same as in the original INLOOP. With the HTML artifacts and the test-ready constraint-based test sets, the task can be considered *Published*. The continuous publishing cycle is now finished. The master constraint-sets are at this step no longer needed and could be deleted from the INLOOP intern space. This could be done when resources are limited. But it would mean that they must be imported again at the next publishing cycle.

The master constraint-sets can be published with the same process. They are then used to create the constraint-based tests-sets in the *Pull changes, Generate files* step. After that, they are no longer used in the *Published* phase. The only difference is that they are not task-dependent and can be saved in their own static file structure.

For the *Background Worker* to work, theoretically only the constraint-based test-set, the student solution, and the Constraint-based Test Engine is needed. If Epsilon is used as the test engine, an Ecore metamodel of the model type is needed, too. If a student solution is submitted, a new Background worker in a *Docker* instance is created. This can be done by the same configuration file type INLOOP uses today. The only difference is that not *JUnit* is called but the Constraint-based Test Engine jar file. Then the test-set and the student solution are put in the input folder. At the end of the process, the Constraint-based Test Engine should give feedback over the standard output. That output is used by INLOOP for feedback. Also, a XML file output for archival purposes is possible.

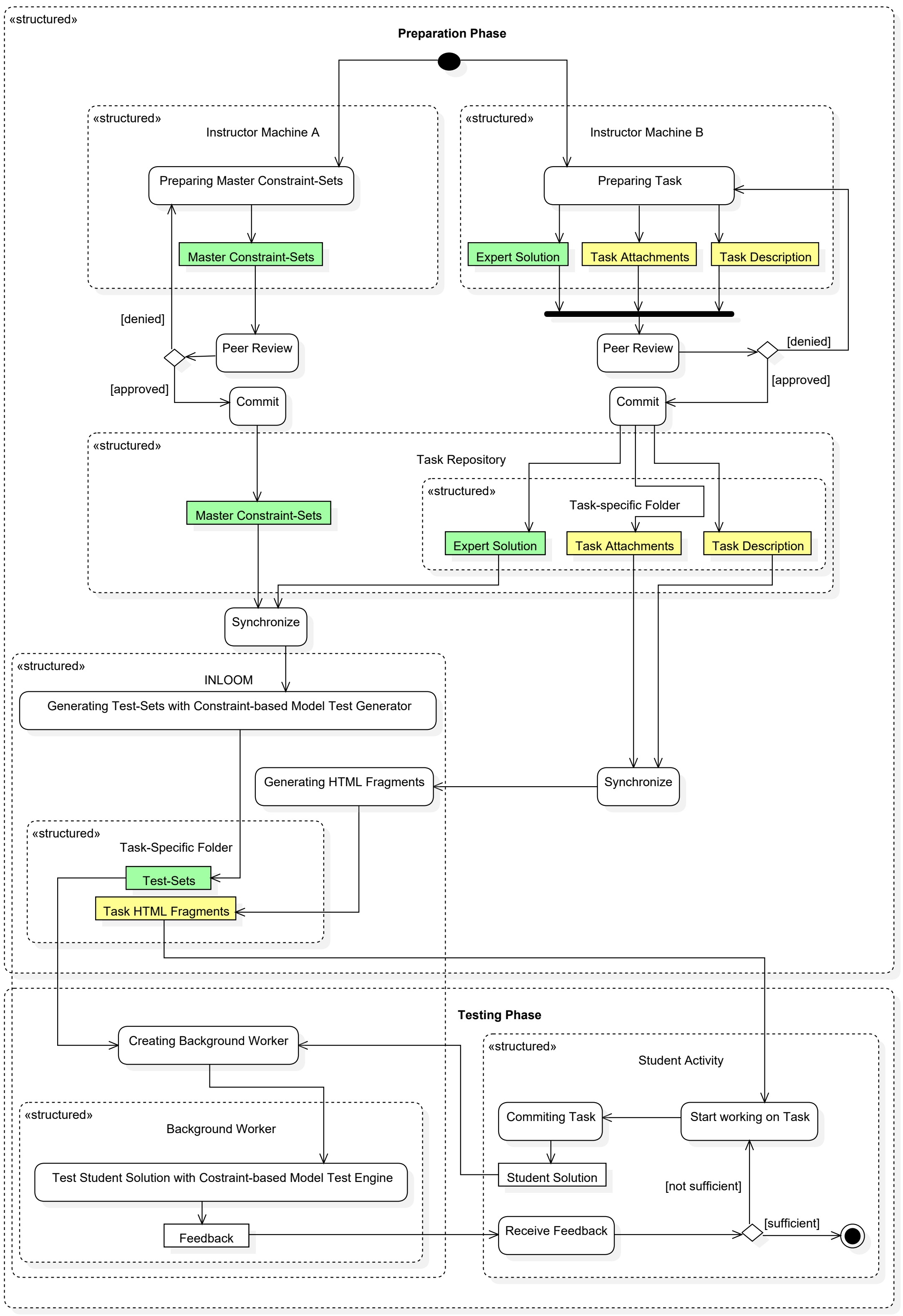


Figure 10: Workflow of the INLOOM system integrated into the INLOOP *continuous publishing* workflow. The green objects are part of the generation of the task-specific test-sets. Yellow objects are part of the generation of the task-specific Website fragments.

3.6 Grading and Feedback Options

This section discusses the problems and solutions of the topic of feedback and grades for model assessment systems. There are usually two types of feedback given by a model assessment system. First, there is the textual feedback that should give hints for the solution, and second the grades that are a qualified value of the solution overall.

3.6.1 Feedback Classes

Since INLOOM uses a constraint-based matching algorithm, the constraints must have different feedback categories. These are needed to differentiate between correct, partially correct, and false solutions. For this reason, a classification is needed that assesses all possible states the matching of a student solution model element can reach. To find these categories, the results in exam and exercise correction, as well as old solution models, were surveyed. The following results for the matching of an element were found. Additionally, for every category, the desirable textual feedback and grading information is mentioned.

1. **Correct:**
   The element in the student solution matches with the expert solution or differs only a bit without missing any information. This result would provide the full amount of points attached to the element. The textual feedback should state that all is correct.

2. **Warning:**
   The element in the student solution does not match with the expert solution, but it is valid. It is not the optimal solution to the problem described in the task. The student must know that his solution is correct, but not optimal. That is important in case he wants to find the optimal one. This result would provide the full amount of points attached to the element. The textual feedback should state that the solution is correct but not optimal.

3. **Error:**
   The element in the student solution could be matched but has missing information. The student must know that his solution could be found, but contains errors or is missing something. This result would provide a reduced amount of points attached to the element. The textual feedback should state that the solution partially incorrect or is missing Information.

4. **Missing/Wrong:**
   The element in the student solution could not be matched to the expert solution. This result would provide no points to the solution. The textual feedback should not be shown to the student. That is to avoid giving away parts of the solution.

5. **Info:**
   The element in the student solution breaks style or modeling convention. This result should not provide or reduce points. The textual feedback should help the student to improve their models.

In INLOOM every constraint should be of one of these categories. The first, second, and third categories should be covered by the normal element-based constraints of the test-sets. Since most of the literature does not use feedback on missing elements or elements that only exist in the student model, no fourth category constraint should be existing. The fived category can be perfectly covert by global constraint test-sets because constrains of this category must apply to all elements.

Since every element should only trigger one constraint, these categories force a certain processing order to the system. First, an element of the student model must be checked against all first category constraints. If none of the first category constraints are triggered, it is checked against all second category constraints. After this, all third category constraints should be checked. If none of these layers work, the fourth level constraint can trigger, or the element is discarded. Info constraints are excluded from this process order and can be called for every element, regardless of other triggered constraints.

3.6.2 Textual Feedback

Textual feedback should be a short sentence that helps the student to understand which parts of his submitted model are correct or incorrect. It should also give hints on how to improve his model. That should happen without giving away part of the solution. For this, the text should be short and as abstract as possible.

The following enumeration shows template sentences that can be used for this purpose. Since the fourth class should not be used, it was excluded. These templates are based on the surveyed literature of Section 2.2.

1. **Correct:**
   <Element Type> <Element Name> was found.
   The <Relationship Type <between < Source Name> and <Target name> was found.

2. **Warning:**
   <Element Type> <Element Name> was found but could be modeled in a better way.

3. **Error:**
   <Element Type> <Element Name> was found but was not completely correct.

5. **Info:**
   <Element Type> <Element Name> does not comply to the used modeling conventions.

Naturally, it is possible to add additional information for the student as well. This can be especially helpful for beginner tasks. If that is the case, this can increase the complexity of the constraints. That is because one of multiple messages must be chosen, or the number of constraints must increase proportionally to the messages.

2. **Warning:**
   <Element Type> <Element Name> was found but could be modeled in a better way. Why must <Element Name> be a <Element Type>?

3. **Error:**
   <Element Type> <Element Name> was found but was not completely correct. Check the <Source of Error >.

3.6.3 Grades

A grade should be a qualified value that shows the student how much of the defined problem his solution solves.

Grades could be shown next to the textual feedback, only as a value to maximal point value (40/100), or as the value of a percentage. From these options, the first option is often the preferred one. It gives the student the best information and leads to a transparent grading scheme for the student.

In INLOOM, all points are based on the GradableElements of the expert solution, which are defined by the instructor. These point attributes are then included in the constraints, with the first and second category constraints giving the full amount of points. Third category constraints give a reduced amount. The maximum amount of points for a task can be calculated from all not pre-existing GradableElements. It can be then passed to a global main constraint that formats the output.

Though not necessary, a uniform grading scheme per model type is advisable. This helps the student to better assess his models and his modeling skills before and after submitting them. It also helps to later estimate the quality of his exam solution, preventing surprises and frustration. Unfortunately, the grading schemes for old exams are not uniform. This makes it necessary to generate an average grading scheme per model type.

3.6.4 Feedback Format

To gather all feedback data for the use in the web-based modeling editor, and for later evaluation, a uniform feedback format is necessary. This format must be generated by the Constraint-bast Testing Engine. This can be done in a dynamically and easily changeable way by a specific global constraint file generated for each task. This constraint file saves all the results of the other constraints and sends them to the standard output or file.

For INLOOM, a simple XML format is proposed. The design is shown in the Listing 1. For every student solution, there is one XML file with one *TestResult*. This TestResult holds the general *TestData*, a list of *Result*s, and the *ResultPoints*. The TestData has two data sets. The first is *RuleModel* which holds the name or id of the expert model, the constraints are generated from. Later the name or id of the task could be added there as well. *TestModel* hold the id of the student model that is tested. This id should be combined from a unique model id and the counter of tries. That helps to better visualize the history of the solution-finding. The list of Results holds the matching results of all found matches between constraints and student model elements. The results of global constraint matches are included as well. Each Result holds multiple data sets. *TestObject* is the element name in the student solution that got matched by a constraint. *RuleObject* is the element in the expert solution that generated the constraint. *RuleSet* is the id of the set of constraints. An example is that all class-related constraints could belong to the RuleSet R01. *Rule* is the id of the constraint inside a RuleSet. 0001 is an example of this. Those two data points show the constraint that delivers the Result. *Category* means the feedback category, the constraint belonged to, meaning Correct, Warning, Error, or Info. *Points* are the rewarded points for the student through the constraint. *Msg* means Messages and defines the textual feedback for the student that is generated by the constraint. The last data inside the TestResult is the ResultPoints. These represent the grade rewarded to the submitted solution. This includes the maximal possible points (*ModelPoints*) and the granted points (*TestPoints*).

```
<?xml version="1.0" encoding="UTF-8"?>
<TestResult>
 <TestData>
  <RuleModel name="ExSS2015_expert"/>
  <TestModel name="ExSS2015_student_1234"/>
 </TestData>
 <Results>
  <Result>
   <TestObject>stellung</TestObject>
   <RuleObject>Stellung</RuleObject>
   <RuleSet>R01</RuleSet>
   <Rule>0010</Rule>
   <Category>Correct</Category>
   <Points>1.0</Points>
   <Msg>Class Stellung was found.</Msg>
  </Result>
   ...
  <Result>
    ...
  </Result>
 </Results>
 <ResultPoints>
  <ModelPoints>27.5</ModelPoints>
  <TestPoints>27.0</TestPoints>
 </ResultPoints>
</TestResult>
```

Listing 1: A proposed standard XML format for the generated feedback. It includes data for the instructor as well as the textual feedback messages (*Msg*) and scores (*Points*) for the student.

All of this information should give a complete overview of the matching results that can be archived for later evaluation. The Points, Messages(Msg), and ResultPoints can be given to the student as their feedback. A possible format that includes all important feedback information for this can be:

- <Category1>: (<Points1> Points) <Msg1>
- <Category2>: (<Points2> Points) <Msg2>
- .....
- Grad: <Testpoints> /<ModelPoints>

3.6.5 Critic and Problems

There can be multiple problems with feedback given to students, especially if its immediate feedback. First, if not given in the right quantity, it can give away parts of the solution of a task. An example of this is the number of missing elements. If this feedback is given, it would tell the student the number of elements of all the different types. That is especially the case if given for an empty model. Another example of this problem is the feedback that a specific element is not found. That would reveal the element and its type. The last example is the feedback of an element that should not be included in the solution. Because of this, it should be avoided to give feedback on wrong or missing elements.

But even corrective feedback can have these effects. For example, someone can try to submit a model with all nouns in the task description placed as classes. In this case, the feedback on the correct classes gives away which nouns are classes and which are, for example, attributes. This problem can not be solved easily since some feedback must be given to the student.

To reduce these problems, in the literature other restrictions were placed on feedback generation. An often used restriction is the number of tries a student can use to submit a model. With this restriction, trial and error strategies can not be used by the student. With this, the problems above will probably be alleviated. The problem with this approach is that more information must be saved per student. Also, the limitation on tries can be a deterrent for many students. Another version of this solution is to limit the tries but only block the student for a specific time frame. This way, the student can submit unlimited solutions in the end. With this, a trial and error strategy will consume a great amount of time.

The second counter-strategy for the problems above is to limit the feedback and grading the student sees. That limitation is maintained until he completes a percentage of the model. This way, trial and error strategies, especially at the beginning of the model process, can be blocked.

Last, there are also multiple options in the literature that say that such trial and error strategies are a normal form of learning. In that light, they do not need to be countered. This can also be a valid option.

3.7 Summary

After designing the INLOOM CAS, it must be confirmed that the designed system complied with all the requirements stated previously.

(RQ1)

The system should prepare the students for the exam of the beginner software engineering course.

(RQ1.1)

The system should use exercise types that are used in the exam and exercise lessons.
Result: The system can use the same task types as in the exam and exercise. A complete list can be found in Section 3.2.

(RQ1.1.1)

The system must be able to exclude pre-existing elements from feedback generation.
Result: The System can exclude elements from the matching using the isPerExisting variable of the metamodel element GradableElement. See Section 3.3.2.

(RQ1.1.2)

The system should be able to be used on many different model types without changing the core process.
Result: The original expert model is transformed to a constraint-based test-set. This test-set is of textual nature and can be generated from any model type using a master constraint-set. The Test Engine only uses this test-set which makes the process itself unconstrained by model type. Some possible implementations, like Epsilon, need a metamodel file, but the model type is not included in the algorithm. It must be said that there can be problems with the complexity of the master constraint-sets, especially with dUML models. See Section 3.4.2.

(RQ1.2)

The system should use forced keywords for the exercises.
Result: The system can use forced keyword matching in its constraints. It is also possible to use simple label matching algorithms, like Levenshtein, if needed. That is dependent on the implementation of the master constraint-sets. See Section 3.4.2.

(RQ1.3)

The system must use aUML and UML metamodels.
Result: The system uses EMF/Ecore as the model representation. Because of this, it is possible to create different aUML metamodel types to use as well as use the already existing Ecore/EMF UML2 Implementation. As a disadvantage, that also means that the system is strongly interwoven with Java and the EMF modeling ecosystem. This can hinder future extensions or improvements in performance. See Section 3.3.2.

(RQ2)
The system should give appropriate feedback to the students and instructors.

(RQ2.1)
The system must give textual feedback to the student for his submitted solution.
Result: The system can have a textual feedback message for every constraint. It only gives feedback on found matches and not on missing ones. This message is part of the output format of the system. See Sections 3.6.2 and 3.6.4.

(RQ2.2)
The system should give a grade to the student for his submitted solution.
Result: The system can have a point value for every constraint. These can be used to calculate a grade at the end of the matching. This grade, as well as the point values, are part of the output format of the system. See Sections 3.6.3 and 3.6.4.

(RQ2.3)
The system must give a complete matching report to the instructor for later evaluation.
Result: There is a proposed XML report format that holds all relevant information about the matching. See Section 3.6.4.

(RQ3)
The system must be compatible with the architecture of INLOOP.

(RQ3.1)
The system should implement continuous publishing.
Result: The workflow of the system is compatible with the continuous publishing paradigm and the overall architecture of INLOOP. See Section 3.5.3.

(RQ3.2)
The system must have a web interface for the students to receive their tasks and upload solutions.
Result: The system can use the already existing INLOOP web application together with a local tool. It is possible to implement a web-based online editor in the web application to allow immediate feedback. See Section 3.5.1.

(RQ3.3)
The system must give the instructor the ability to upload his task descriptions and solutions with a Git commit.
Result: Instructors can use a pre-packed Eclipse IDE to prepare their tasks and design expert model solutions. Specialized instructors can use their own pre-packed Eclipse IDE to design and test the master constraint-sets. Both can then upload their work with Git to a task repository that process and publish it to INLOOM. See Section 3.5.2.

4 Realization of a Analysis Class Diagram Matching Engine

In this chapter a proof-of-concept implementation of the *INLOOM CAS* matching engine is presented. Since the most important model type of the exams is the analysis class diagram this model type will be implemented. Since it is not feasible to implement the whole *INLOOP* [20] system with the web application and background workers in the time frame of this thesis only the two stages matching framework will be implemented. The goal of this implementation is that it is possible to model a expert model and student solution model and match these two against each other. Last, the feedback in XML format should be generated. The basic architecture of the implementation will be the architecture mentioned in the last chapter[14].

The implementation was realized as follows. First, a metamodel of the analysis class diagram was generated using the *EMF* framework. Then a grading scheme was summarized out of the grading schemes of former exam task for this model type. Third, the master constraint-set was designed and the model test generator was integrated. Lastly the test engine and the feedback generation were implemented. The next sections will discuss each of these steps.

4.1 Analysis Class Diagram Metamodel

The first step of the implementation of the INLOOM CAS matching engine was the implementation of the analysis class diagram metamodel. This was done through the EMF framework. First, an *Ecore* model, based on the simplified metamodel shown in Figure 5, was created. In the process, some changes were made to the simplified metamodel, which makes it easier to use by the INLOOM engines. This metamodel was then extended by the *GradableElement* to include the grading information. The final metamodel is presented in Figure 11.

The metamodel was designed in Ecore and EMF. It was then used to automatically generate a simple EMF editor. That editor can be used to model the expert solution and digitize student solution models. The complete description of how to generate the editor and to model *aUML* class diagram can be found in the attachments at the of this thesis. The digital version holds all needed Eclipse projects and includes multiple expert solutions for old exam tasks.

[14] See Section 3 on Page 15

The following paragraphs describe the metamodel of the aUML class diagram. The whole model is represented by the *OOAClassModel* element, which holds the *id* of the model. The model holds the *AbstractClass* element that can be an *Enumeration* or a *Class*. Enumerations have *LiteralGroups* that hold multiple *EnumLiterals*. Classes have *Properties* and *Operations*, which are defined by their *name*. The former can also have a *lower* and *upper* multiplicity value. The default value is one. Next to these Elements the model also holds relationships. One of those is the *Generalization*, which is defined by *id*. It shows the inheritance between a *general* and a *specific* class. The other type is the *Relationship* between AbstractClasses. These Relationships are also defined by their *id*. They have two *RelationshipEnds* to two AbstractClasses, a *name*, and a *readingDirection* to one of the ends. Last, they can have an *AssociationClassEnd* to symbolize a class as an association class. The RelationshipEnds hold a defining *id*, multiplicity values, and a *type*. The *RelationshipType* shows if the Relationship is an Association (none), an Aggregation (shared), or a Composition (composite). It also tells from which end the aggregation or composition is starting. Last, a RelationshipEnd can have a *Role*, defined by a name. The last element of the metamodel is the *GradableElement* holding the feedback information. AbstractClass, Property, Operation, EnumLiteral, Generalization, Role, Relationship, and both RelationshipEnd and AssociationClassEnd inherit from this Element. With this, they are included in the feedback generation. These inheritance candidates were chosen through old exam grading schemes. Element types that are included in these old grading schemes were set to inherit from GradableElement. It must be mentioned that the mode attribute of the GradableElement was not used in this implementation. That is due to the usage of a standardized grading scheme, leaving only the points changeable. In a later version, if more variability is needed, support of that could be added.

It should be noted that this metamodel is strictly following the original metamodel, used in the beginner software engineering course. This means typical model errors that are based on a misunderstanding of the limitation of aUML can not be produced with this metamodel. That includes direction arrows in relationships, types of properties, or enumerations as association classes. That has advantages and disadvantages. First, as an advantage, the student can only generate valid models and has a better start in learning to model. On the other hand, this can lead to a dependency on pre-selected options. That can cause problems for some students if they use more general modeling editors later. This disadvantage could be negated by showing the student all UML model elements in the later web-based editor but block all not aUML elements. An advantage of this solution would be that only one editor must be implemented for aUML and *dUML*.

After designing this metamodel and generating the editor based on it, new models can be created. For this, objects of each meta element can be created. Objects of gradable meta elements can be given individual point numbers. For example, the property *age* could be worth *0.5* points or the optional property *siblings(0...1)* could be worth *1.0* points. The metamodel is the same for the student solution. There is no dedicated metamodel for student solutions. Instead, feedback information is simply ignored by the test engine.

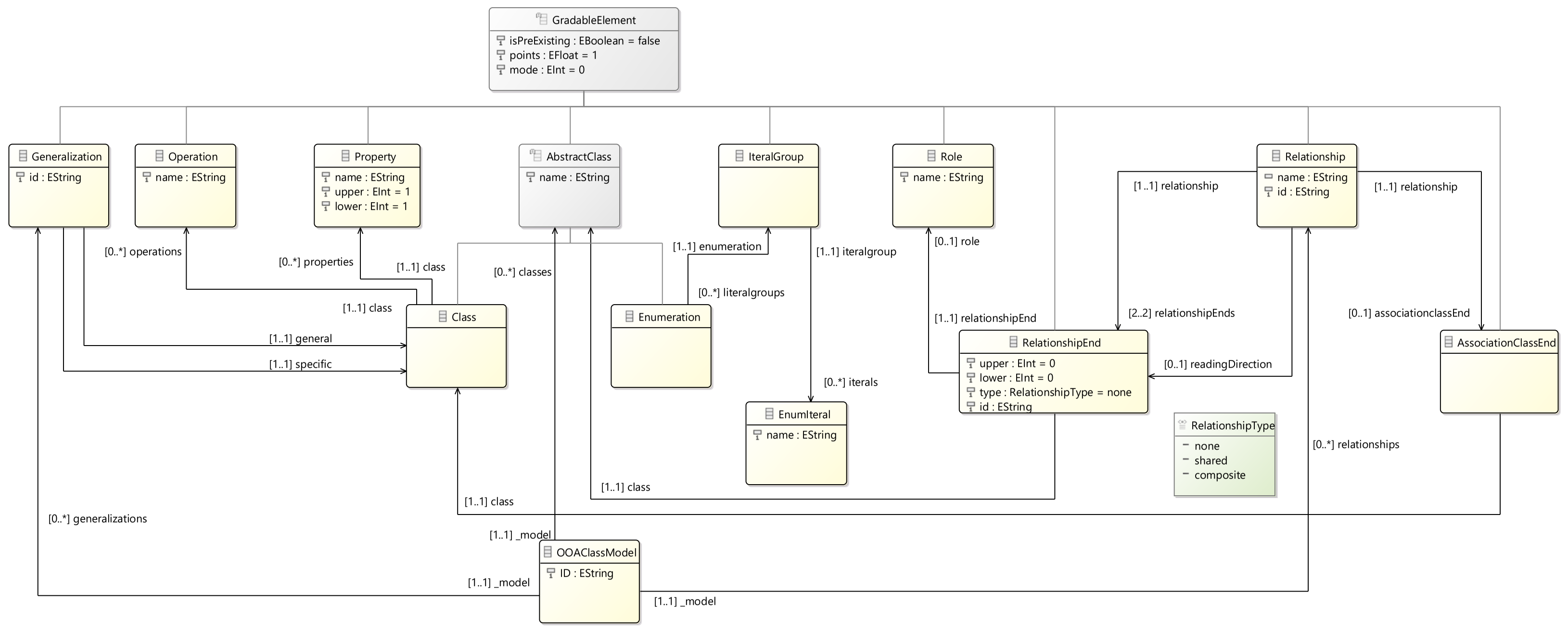


Figure 11: Ecore metamodel of the analysis class diagram used in the proof-of-concept implementation of the INLOOP matching engine for aUML class diagrams. The model is taken from the Ecore editor and describes the data structure of the aUML class models.

4.2 Grading Scheme

The second step for the implementation was to find a viable grading scheme for expert models. In old exams, the grading scheme was not uniform since the number of points was also dependent on the other tasks. For an assessment system like INLOOP, a *Uniform Grading Scheme* would be better. This would help the student to better assess their models and skills since the grades over all models would be comparable. Another advantage would be the better comparability of the matching results from an evaluation respective. That is because fewer variables would be introduced. Last, a uniform grading scheme would reduce the complexity of the constraints. Because of this, a uniform grading scheme is preferable.

After surveying the grading schemes of old exams, an average uniform grading scheme for aUML class diagrams was conducted. The scheme includes the following grades for possible elements.

Class:
: 1.0 Points for each Class

Property:
: 0.5 Points for each normal Property
: 1.0 Points for each special Property with multiplicity or marked as deductible (/age)
: 0.5 Points for each special Property (multiplicity or deductible) that contains errors
: 0.25 Points for each normal Property that contains errors

Operation:
: 0.5 Points for each Operation

Enumeration:
: 1.0 Points for each Enumeration

LiteralGroup:
: 1.0 Points for each Literalgroup if all EnumLiterals are correct
: 0.5 Points for each Literalgroup if EnumLiterals are missing or incorrect

Generalization:
: 0.5 Points for each Generalization that is correct

Relationship:
: 1.0 Points for each Relationship that is correct and has no association class (Roles are not included)
: 0.5 Points for each Relationship that is matched but has errors and has no association class (Roles are not included)

Role:
: 0.5 Points for each Role that is correct
: 0.0 Points for each Relationship that is matched but is incorrect

Association Class:

0.0 Points for each Relationship that has a association class
1.0 Points for each AssociationClassEnd that is correct
0.5 Points for each AssociationClassEnd that is matched but incorrect
1.0 Points for each RelationshipEnd that is correct (maximal 2.0 Points per association class)
0.5 Points for each RelationshipEnd that is matched but incorrect

This grading scheme is advisable for the future tasks in INLOOM that have the modeling type of an aUML class diagram. In general full points means a first- or second-level constraint (Correct or Warning) covers this item. Every partial point is covert by third-level constraints (Error).

The implementation is not designed for this grading scheme alone. If other point values are used, the general approach is that the constraints of the first and second categories give full points. The third category constraints are rewarding half the points. The only exception from this rule is the Role element, which gives zero points on third category constraints. This is done to the fact that the role was separated from the Relationship element. This was done to reduce the variation in the complexity of relationships to get full points. Giving half points to roles would create too great of a difference to the usual exam grading scheme. Because of this, the exception was put in this place.

The overall differentiation and element combination is implemented over the master constraint-sets. If another grading scheme that uses different element combinations is needed, the used master constraint-sets must be modified. That can be done in combination with the mode attribute, provided by the GradableElement metamodel element. One number of the mode variable is equivalent to one unique grading scheme. This allows multiple grading schemes to exist next to each other and be even mixed if necessary. In theory, each of the grading schemes used in the old exams can be recreated by this. Since this would go over the scope of this thesis and is only of limited use, it is not realized. Only the INLOOM grading scheme described in this section will be implemented in the proof-of-context implementation.

4.3 Constraint-based Model Test Generator

To implement the *Constraint-based Test Generator* of the INLOOM CAS, several steps must be taken. First, master constraint-sets must be designed. Then an implementation method for the test and the generator must be found and implemented.

4.3.1 Definition of the Master Constraint-Sets

At first, the number and scope of each *master constraint-set* must be defined. Since the meta-model holds several element types that inherit from GradableElement, having one set for each of these types is preferable. This helps with the separation of the constraints and so with the future extensions of the constraint-sets. Also, each of these meta elements has one subsection in the uniform grading scheme. A separation of the master constraint-sets along the grading scheme would help with future extensions of these schemes. There is one exception to this. The relationship encompasses associations, aggregations, and compositions which each have different constraints. Because of this, each type of relationship generates its own master constraint-set. Non-gradable meta element types do not need own constraint-sets since they have no influence on the feedback and grades. Information from them can be used nonetheless in the constraints of the gradable elements. These meta element types can also be included later by global constraint test-sets. In the end, since all master constraint-sets are converted into test-sets, the composition of the master constraint-sets does not greatly affect the test engine. This means the composition can be decided based on the facts mentioned above.

The following Tables 4 till 13 give an overview of the master constraint-sets, with each table holding on set. In the end, 54 element-based master constraints were created in 10 master constraint-sets. This was done by using the experience of assessing the aUML class diagrams of exams for multiple years (3 terms), as well as holding model exercises of the beginner software engineering course for multiple years (5 terms), too. From these experiences, common misconceptions and errors of beginner level students were collected and used to generate the master constraints. These constraints are by no means complete but should guarantee sufficient matching quality. In the future, by adding more constraints this quality can be further increased. For comparison, in the literature, some systems for UML class diagrams are using ca. 130 constraints to match student solutions [3]. The proof-of-context is using simplified aUML, and the number of constraints could easily be increased. So it can be said that at the moment, the number of constraints is not too excessive in comparison to other similar systems.

Table 4: Master constraints included in the master constraint-set of the *Class* element. The points rewarded by the constraint are the points of the expert solution element multiplied by the corresponding number of the *Points* column (x<number>).

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Class	R01	0010	Correct	Matched against class name	x1	Class <name>was found.

Table 5: Master constraints included in the master constraint-set of the *Property* element.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Property	R01	1010	Correct	Matched against name, class and multiplicity	x1	Property <name>in Class <class name>was found.
		1011	Error	Matched against name without "/", and class	x0.5	Property <name>in Class <class name>was found, but not completely correct.
		1020	Warning	Matched against a property with the same name in another class. The multiplicity is 1. Aggregation to this class with the correct name as role and multiplicity exists.	x1	Property <name>in Class <class name>was found , but can be modeled in a more optimal way.
		1021	Warning	Matched against a property with the same name in another class. The multiplicity is correct. Aggregation to this class with the name as role and multiplicity of 1 exists.	x1	Property <name>in Class <class name>was found , but can be modeled in a more optimal way.
		1022	Error	Matched against a property with the same name in another class. Aggregation to this class with the name as role exists.	x0.5	Property <name>in Class <class name>was found , but can be modeled in a more optimal way. It contains errors/missing things.
		1030	Warning	Matched against a property with the same name in another class. The multiplicity is 1. "has"-Association to this class with the correct name as role, reading direction and multiplicity exists.	x1	Property <name>in Class <class name>was found , but can be modeled in a more optimal way.
		1031	Warning	Matched against a property with the same name in another class. The multiplicity is correct. "has"-Association to this class with the name as role, reading direction and multiplicity of 1 exists.	x1	Property <name>in Class <class name>was found , but can be modeled in a more optimal way.
		1032	Error	Matched against a property with the same name in another class. Association to this class with the name as role exists.	x0.5	Property <name>in Class <class name>was found , but can be modeled in a more optimal way. It contains errors/missing things.
		1040	Warning	Matched against a class with the same name. Aggregation to this class with the correct name as role and multiplicity exists.	x1	<name>was found , but can be modeled in a more optimal way.
		1041	Error	Matched against a class with the same name. Aggregation to this class exists.	x0.5	<name>was found , but can be modeled in a more optimal way. It contains errors/missing things.
		1050	Warning	Matched against a class with the same name. "has"-Association to this class with the correct name as role, reading direction and multiplicity exists.	x1	<name>was found , but can be modeled in a more optimal way.
		1051	Error	Matched against a class with the same name. Association to this class exists.	x0.5	<name>was found , but can be modeled in a more optimal way. It contains errors/missing things.
		1060	Warning	Matched against a property with the same name in another class. The multiplicity is correct. A generalization from this class exists.	x1	Property <name>was found , but can be modeled in a more optimal way.
		1061	Error	Matched against a property with the same name in another class. A generalization from this class exists.	x0.5	Property <name>was found , but can be modeled in a more optimal way. It contains errors/missing things.

Table 6: Master constraints included in the master constraint-set of the *Operation* element.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Operation	R01	2010	Correct	Matched against name and class	x1	Operation <name>in Class <class name>was found.
		2020	Warning	Association exists to a class. The class name is contained in the operation. Association name is matched against operation name. Reading direction is to the class.	x1	<name>was found, but can be modeled in a more optimal way.
		2030	Warning	Matched against a operation with the same name in another class. A generalization from this class exists.	x1	Opertation <name>was found , but can be modeled in a more optimal way.

Table 7: Master constraints included in the master constraint-set of the *Enumeration* and *Literalgroup* elements.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Enumeration	R02	0010	Correct	Matched against name.	x1	Enumeration <name>was found.
LiteralGroup		0020	Correct	Matched against all Literals in the group	x1	Enumerationliterals <names>were found in <Enumeration name>.
		0021	Error	Matched against some literals in the group	x0.5	Enumerationliterals <names>were found in <Enumeration name>, but were not correct.
Enumeration		0030	Error	Matched against a class with the same name.	x0.5	<name>was found, but not correct.
LiteralGroup		0040	Correct	Matched against all literals in the group but as properties in a class.	x1	<names>were found in <class name>.
		0041	Error	Matched against some literals in the group but as properties in a class.	x0.5	<names>were found in <class name>, but were not correct.

Table 8: Master constraints included in the master constraint-set of the *Generalization* element.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Generalization	R03	0010	Correct	Matched against specific and general classes.	x1	Generalization between <specific class> and <general class>was found.
		0020	Error	Matched against a "is_a"-association.	x0.5	Relationship between <specific class> and <general class>was found, but not correct.
		0030	Error	Matched against a aggregation.	x0.5	Relationship between <specific class> and <general class>was found, but not correct.
		0040	Error	Matched against a composition.	x0.5	Relationship between <specific class> and <general class>was found, but not correct.

Table 9: Master constraints included in the master constraint-set of the *Association* element. There are two additional general checks. First, on every constraint there is a check if a role is modeled as an object, with no point penalty. Second, in all error type constraints it is checked if the association is modeled in the opposite way.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Association	R04	0010	Correct	Matched against all parts of a association. (type, name, end classes, multiplicities, and reading directions)	x1	Relationship between <class 1> and <class 2>was found.
		0011	Warning	Matched against all parts of a association. (type, end classes, multiplicities) The name was matched against the role in the right direction.	x1	Relationship between <class 1> and <class 2>was found, but can be modeled better
		0012	Error	Matched against some parts of a association. (type, end classes, and name/role or multiplicities)	x0.5	Relationship between <class 1> and <class 2>was found, but not correct.
		0013	Error	Matched against some parts of a association but not the type. (end classes, and role(name) or multiplicities)	x0.5	Relationship between <class 1> and <class 2>was found, but not correct.

Table 10: Master constraints included in the master constraint-set of the *Aggregation element*. There are two additional general checks. First, on every constraint there is a check if a role is modeled as an object, with no point penalty. Second, in all error type constraints it is checked if the association is modeled in the opposite way.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Aggregation	R05	0010	Correct	Matched against all parts of a aggregation. (type, end classes, multiplicities)	x1	Relationship between <class 1> and <class 2>was found.
		0011	Warning	Matched against all parts of a ”has”-association. (type, end classes, multiplicities, name, and reading direction)	x1	Relationship between <class 1> and <class 2>was found, but it could be more specific.
		0012	Error	Matched against all parts of a composition. (type, end classes, multiplicities)	x0.5	Relationship between <class 1> and <class 2>was found, but is not correct.
		0013	Error	Matched against some parts of a association. (end classes, and type or name or multiplicities)	x0.5	Relationship between <class 1> and <class 2>was found, but is not correct.

Table 11: Master constraints included in the master constraint-set of the *Composition* element. There are two additional general checks. First, on every constraint there is a check if a role is modeled as an object, with no point penalty. Second, in all error type constraints it is checked if the association is modeled in the opposite way.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Composition	R06	0010	Correct	Matched against all parts of a composition. (type, end classes, multiplicities)	x1	Relationship between <class 1> and <class 2>was found.
		0011	Warning	Matched against all parts of a aggregation. (type, end classes, multiplicities)	x1	Relationship between <class 1> and <class 2>was found, but it could be more specific.
		0012	Warning	Matched against all parts of a "has"-association. (type, end classes, multiplicities, name, and reading direction)	x0.5	Relationship between <class 1> and <class 2>was found, but it could be more specific.
		0013	Error	Matched against some parts of a association. (end classes, and type or name or multiplicities)	x0.5	Relationship between <class 1> and <class 2>was found, but is not correct.

Table 12: Master constraints included in the master constraint-set of the *Association Class* element.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Association Class	R07	0010	Correct	Matched against the AssociationclassEnd of the right association	x1	Association class <class name>was found.
		0011	Error	Matched against a class that is connected by associations to both classes of the original RelationshipEnds of the AC relationship.	x0.5	Class <class name>was found, but its Relationship was not correctly modeled
		0110 / 0120	Correct	Matched against a RelationshipEnd (type, class, multiplicity, and opposite End)	x1	Relationship between <class 1> and <class 2>was found,
		0111 / 0121	Error	Matched partially against a RelationshipEnd. (class and opposite End)	x0.5	Relationship between <class 1> and <class 2>was found, but is not correct.
		0112 / 0122	Warning	Matched against a RelationshipEnd (type, class, multiplicity) with the Association class as opposite end.	x1	Relationship between <class 1> and <ass. class>was found.
		0113 / 0123	Error	Matched partially against a RelationshipEnd (class) with the Association class as opposite end.	x0.5	Relationship between <class 1> and <ass. class>was found, but is not correct.

Table 13: Master constraints included in the master constraint-set of the *Role* element. Error type constraints of this constraint-set do not give points since the role originally should be worth zero points but give away a error message as a hint.

Element	RuleSet	Rule	Category	Description	Points	Textual Feedback
Role	R08	0010	Correct	Matched against the role name and the including association	x1	Role <name>was found.
		0011	Warning	Matched against an association name, reading direction, and the association itself.	x1	<name>was found, but can be modeled in a more optimal way.
		0012	Error	Matched against the role name and the including association but is on the wrong end.	x0	Role <name>was found, but is not correct.
		0013	Error	Matched against an association name, wrong reading direction, and the association itself.	x0	<name>was found, but is not correct.
		0014	Error	Matched against the role name and the including association but missing a derived marking	x0	Role <name>was found, but is not correct.
		0020	Error	Matched against a class with the same name and two associations to the original end classes.	x0	<name>was found, but is not correct.
		0030	Error	Matched against a class with the same name and one association to original opposite end class and one generalization to the original role end class.	x0	<name>was found, but is not correct.
		0040	Error	Matched against a association class with the same name and two associations to the original end classes.	x0	<name>was found, but is not correct.

Each master constraint-set has a RuleSet, and each constraint a Rule id which together forms the complete id of the constraint.

The RuleSet id is given to each set by its defining element type. It has the format *RXX*. *R01*, for example, stands for class constraints and *R02* for enumeration constraints and so on. In general, each set has its one RuleSet with properties and operation as exceptions. These are defined by their including class and have the R01 class RuleSet.

The Rule id has 4 digits and can be described by the *ABBC* format. The *A* digit stands for a general group of constraints. 0-5 are for normal constraint definitions with the implementation using mostly 0 since there was often no need to differentiate further. 6-9 are defined for helper constraints not listed in the tables. Examples for these are global constraints for model conventions that have the number 9, or relationship end matching constraints that are included in the number 6. The *B* digit can hold a number of constraints per A group. The *C* digit can describe constraints that fall under a B group constraint. For example, can the 0*01(B)0(C)* constraint check, if a property is completely correct as a Correct category constraint. The *001(B)1(C)* constraint can check roughly the same but with partial correctness as an Error category constraint.

The absolute id of a master constraint is defined as <RuleSet><Rule >. For example, a normal class master-constraint has the id *R010010*, and a global class constraint has the id *R019010*.

4.3.2 Implementation of the Master Constraint-Sets

After defining the master constraint-set, the implementation of those is the next step. First, an implementation framework must be chosen. A possible candidate is the Ecore/EMF-based *Epsilon* tool kit, already descried in Section 3.4.3. For the realization, the *Epsilon Eclipse IDE plugin* was used. Later, this can be changed to a standalone *Java* application. This application can use the same files as the plugin.

Like already mentioned, the best option to implement the *master constraint-sets* and the *Constraint-based Model Test Generator* is the *EGL* tool or engine. There each master constraint-set can be implemented as an *EGL template file*. The general format of an EGL template file is formatted text with slots at variable places. *Slots* are programmable symbols the EGL engine later fills with model data. Listing 2 shows examples of these slots.

```
1 [%=obj.name%]
2 [% var expertModelPoints = 0.0;%]
3 [% for (obj in col) {%] ... [%}%]
4 [% if (exp) {%] ... [%}%]
```

Listing 2: Example slot types of the EGL template format. From top: include value insertion, variable definition, for-loop, if-loop

These slots can be used to fill the template with data out of a model (line 1). The slots can also be used to program with the data of the model before using it in the template. Through using a Java like dialect called *EOL* (Epsilon Object Language), all normal programming structures like variable definition (line 2), for-loops (line 3), and if-loops (line 4) are possible. Especially the for and if-loops are interesting since they can multiply or hide formatted text inside their scopes. That can be used to create variable constraints based on their model element's data.

Next, the formatted text inside the EGL file must be designed to be later used by the Constraint-based Model Test Engine. This engine will be implemented using the *EVL* tool of Epsilon, and so the generated test-sets must be in the EVL file format. Listing 3 shows the simplified general format of this file type. A *pre* block (lines 1-3) can be used to define variables or call functions. It is processed before all rules. In the listing, a *block variable* is defined that will be explained later. The *context* block (lines 5-26) defines which element type the inside rules are called for. It will be processed once for every element of this type. The *context guard* expression (line 7) will only allow the context to be processed when true. A context can have multiple *rules* (lines 9-20, 24). A rule will be called once per element the context is executed for. The type can be *constraint* or *critique* to show the severity of the rule. This is only shown in the intern output, which is not actively used in the implementation. Each rule must also have a unique name. Inside a rule, the *guard* expression (line 11) will stop the engine from entering a rule if false. If the engine enters a rule the *check* expression (line 13) is called, and if false the *message* block (lines 15-19) is called. In this implementation, the verification engine should send the message when a constraint is met. Such, the matching expression will always be negated. While sending the message, the student feedback is generated. The block variable is also swapped, and at last, an intern message is sent to the engine. This intern message will not be used actively since its information value is not high enough. It can be used for a quick overview in the Eclipse IDE.

```
 1  pre <block name> {
 2          var <Block Variable Name> = true;
 3  }
 4
 5  context <model name>!<Model Element> {
 6
 7    guard : <guard expression of the context>
 8
 9    <rule type> <rule name> {
10
11      guard : <guard expression of the rule>
12
13      check : not(<matching constraint>)
14
15      message     {
16                        setOutput(...);
17                        <Block Variable Name> = false;
18                        return "<Intern message>";
19                  }
20     }
21
22     ...
23
24     <rule type> <rule name> { ...}
25
26  }
```

Listing 3: General format of the EVL files used in the proof-of-concept implementation

After designing the formatted text, for each constraint of a master constraint-set, a rule was designed. Then slots were inserted to later fill the template constraint-set with the information from the expert solution. This was done for one EGL file for each master constraint-set. The following Listing 4 shows this, for the example, for the master constraint-set of the class element type. The slots will later, through the EGL engine, be filled with the data of one class of the expert solution at the time. After the generation of the *test-set*, the id of the element-based constraint becomes of the format <master constraint id><element name>. In the example Listing 5, the master constraint *R010010* becomes the element-based constrain *R010010Stellung*.

```

pre satisfiabilityClass[%=_class.name%] {
        var satisfiabilityClass[%=_class.name%] = true;
}

context student!Class {


  guard : satisfiabilityClass[%=_class.name%]


  critique R01_0010_[%=_class.name%] {

    guard : satisfiabilityClass[%=_class.name%]

    check : not(self.name.toLowerCase()
                == "[%=_class.name.toLowerCase()%]")

    message     {
                setOutput("R01" , "0010",
                       "Correct",
                       self.name,
                       "[%=_class.name%]",
                       [%=_class.points%],
                       "Class [%=_class.name%] was found" );

                satisfiabilityClass[%=_class.name%] = false;

                return "CORRECT: R01_0010[[%=_class.name%]] was
                                  met for Class "+ self.name;
               }
   }

}
```

Listing 4: EGL file of the *Class master constraint-set*

First, a variable is defined in the *pre* block (line 3). This variable saves, whether one constraint for this element was already met. It prevents that one element can produce multiple feedback messages and points. This is important since third category constraints often have fewer requirements than first category ones. Since the EVL engine calls the constraints from the top of the file, the order of first, then second, and then third category constraints guarantees that always the best constraint is met first. It also secures that generated test-set rewards only once feedback and points. That's true even in the case of multiple elements with nearly the same data. To make this variable unique, it ends with the id or name of the expert solution element the test-set is for. This variable is used in the *guards* of the constraint rules (line 14). That is to prevent the rule to be executed if another constraint was already met. To reduce the resources needed in the verifying process the same is done for the *context guard* (line 9). This prevents other student solution elements to be later tested against the test-set.

The *context* of each constraint-set is set so that only the intended element type of the student solution are tested against the generated test-set (line 6).

For each constraint defined in the master constraint-set the EGL file belongs to, a rule is implemented. Next, each rule will be made unique by adding the id or name of the element that the generated test-set is for (line 12).

The *check* expression of each rule is now implemented to fulfill the description in its table row. For that, data from the expert solution element is filled in the implementation with slots. In the example of Listing 4, the equality equation is filled with the lowercase name of the class the test-set is for (lines 16-17). The check will later test if the tested element of the student solution (self) has the same lowercase name as the class in the expert solution.

Each rule has a *message* block that is executed if the negated expression in check is true. In this block, first, the feedback is saved by the *setOutput* function. The function parameters are pre-filled with the constraint id, the feedback information, and the points of the expert solution element (lines 19-25). Then the *block variable* is set false to stop every rule or context execution from now on for this test-set (line 27). Last, the required *intern message* for the engine is sent (lines 29-30).

This template will now be used to generate a test-set as EVL files for every class object that is defined in the expert model. An example output test-set of the template is shown in Listing 5 for a Class *Stellung* from an expert model. This test-set file can now be used to verify student models in the Constraint-based Model Test Engine.

```

pre satisfiabilityPropertyStellung {
        var satisfiabilityClassStellung = true;
}

context student!Class {

  guard : satisfiabilityClassStellung

  critique R01_0010_Stellung {

    guard : satisfiabilityClassStellung

    check : not(self.name.toLowerCase() == "stellung")

    message      {
                 setOutput("R01", "0010",
                        "Correct",
                        self.name ,
                        "Stellung",
                        1.0,
                        "Class Stellung was found" );

                 satisfiabilityClassStellung = false;

                 return "CORRECT: R01_0010[Stellung] was
                                   met for Class "+ self.name;
                }
   }

}
```

Listing 5: EVL file (*test-set*) generated from the class *Stellung* of a expert solution and the master constraint-set for classes

4.3.3 Implementation of the Test Generator

The last step to finish the Constraint-based Model Test Generator is to implement the logic that combined the expert model with the EGL master constraint-sets. This is done by creating a main *EGX*[15] file. EGX is a coordination language for EGL to coordinate the generation of files out of EGL templates. This file includes which model element is used to generate a test-set and the corresponding EGL master constraint-set. In this file, the elimination of preexisting model elements from the test-set generation is carried out, too. Last, the file contains the format of naming the generated test files.

Listing 6 shows the general syntax of a EGX file. Like in most of the Epsilon languages, there is a *pre* block (lines 2-5) that can be used to initialize variables. In the implementation that block is used to define the *output directory* (line 3) and the *test-set file extension* (line 4). Then for each element type in the imported model that should generate an output file, a *rule* must be implemented (lines 7-11). These rules must have *Rule Names*, a *Reference* for the matched element, and an *Element Type* (line 7). The rule will then be called once for each element of that element type in the imported model. Inside of the rule and inside the used template, the element is referenced by the Element Reference. Next, the rule must have the Path to a *template* (line 9) the rule is using to generate the output file. This generation can be controlled by a *guard* expression (line 8) that forbids the creation if the expression is false. The last information a rule needs is the path and name of the output file (*target*) (line 10).

```
 1
 2 pre {
 3   var outDirLib : String = ”../evl/”;
 4   var extension : String = ”.evl”;
 5 }
 6
 7 rule <rule name> transform  <Element Reference> : <Element Type>
      {
 8   guard:        <expression>
 9   template:     ”<Template Path>”
10   target:       ”<Path for Output File>”
11 }
```

Listing 6: General syntax of an EGX file that is used in the implementation of the proof-of-concept matching engine.

In the implementation of the Constraint-based Model Test Generator, these rules are used to create one EVL test-set for each expert model element. Some examples are shown in Listing 7. That means, there is one rule for every master constraint-set. The EGL files for these sets are used as templates (lines 9, 15) in these rules. The guard expression is normally used to check if an element is marked as pre-existing (line 8). In the case of relationships, this is extended to a check if association classes existing, and the type of relationship (lines 14-16). The name and path of the output test-set are generated from the names or ids of the elements (lines 10, 19). There is also the option to generate static files like global constraint test-set out of slot-less templates (21-24). This only happens once for the expert model. If not needed, such a global constraint-set could also be created directly as an EVL file.

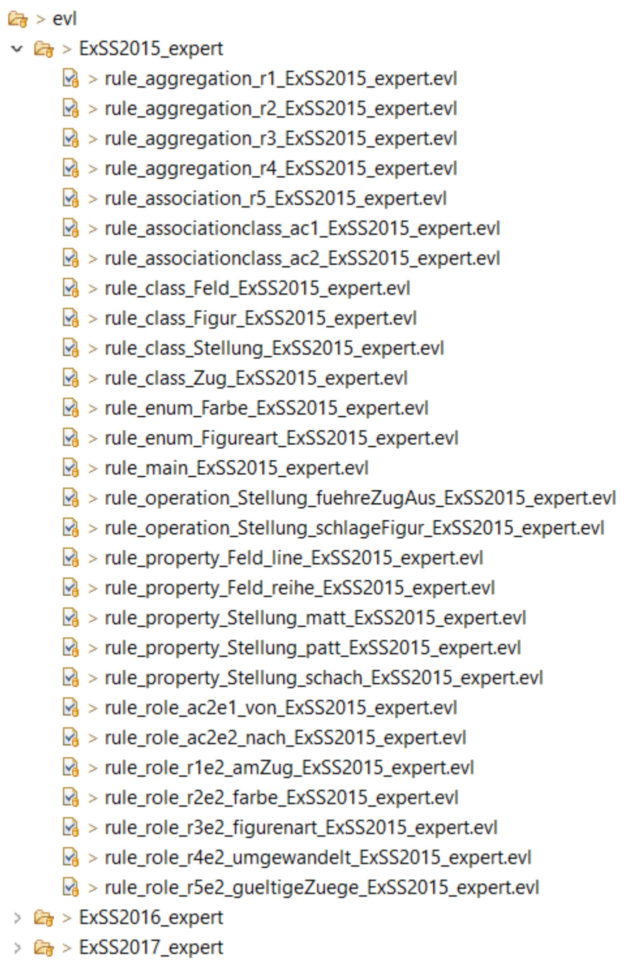
[15] https://www.eclipse.org/epsilon/doc/egx, 10.02.2020

```

pre {
  var outDirLib : String = "../evl/";
  var extension : String = ".evl";
}

rule class_rule transform _class : Class {
  guard: not _class.isPreExisting
  template: "rule_class.egl"
  target: outDirLib + _class._model.ID + "/" + "rule_class_" +
      _class.name + "_" + _class._model.ID + extension
}

rule assoziation_rule transform _ass : Relationship {
  guard: (not _ass.isPreExisting)
         and _ass.associationclassEnd == null
         and not _ass.relationshipEnds
             .exists(re:RelationshipEnd| re.type.value <> 0)
  template: "rule_association.egl"
  target: outDirLib + _ass._model.ID + "/" + "rule_association_"
      + _ass.id + "_" + _ass._model.ID + extension
}

rule global_rule transform _model : OOAClassModel {
  template: "rule_global.egl"
  target: outDirLib + _model.ID + "/" + "rule_global" +
      extension
}
```

Listing 7: Examples of EGX rules that are used in the implementation of the proof-of-concept matching engine.

To start the test generation the *Epsilon EGL Engine* must be called. This tool takes the place of the *Constraint-based Model Test Generator*, mentioned in the former chapter[16]. To work, it now needs the main EGX file, the expert model, and the Ecore metamodel file for the aUML class diagram as parameters. The EGL template files only need to be found under the path that is mentioned in the EGX file. Now, an EVL file will be generated for every GradableElement of the expert model, using the corresponding EGL master constraint-set. A complete user guide for the test-set generation in an Eclipse IDE can be found in the attachments of this work. Figure 12 shows, as an example, the output test-sets for an old aUML class diagram exam task (Summer term 2015). These files can now be used to test student models in the Constraint-based Test Engine. The *rule_ main_ ExSS2015.evl* file will be discussed in the following section[17].

[16] See Section 3.4 on Page 24
[17] See Section 4.4 on Page 60



```
> evl
v > ExSS2015_expert
    > rule_aggregation_r1_ExSS2015_expert.evl
    > rule_aggregation_r2_ExSS2015_expert.evl
    > rule_aggregation_r3_ExSS2015_expert.evl
    > rule_aggregation_r4_ExSS2015_expert.evl
    > rule_association_r5_ExSS2015_expert.evl
    > rule_associationclass_ac1_ExSS2015_expert.evl
    > rule_associationclass_ac2_ExSS2015_expert.evl
    > rule_class_Feld_ExSS2015_expert.evl
    > rule_class_Figur_ExSS2015_expert.evl
    > rule_class_Stellung_ExSS2015_expert.evl
    > rule_class_Zug_ExSS2015_expert.evl
    > rule_enum_Farbe_ExSS2015_expert.evl
    > rule_enum_Figureart_ExSS2015_expert.evl
    > rule_main_ExSS2015_expert.evl
    > rule_operation_Stellung_fuehreZugAus_ExSS2015_expert.evl
    > rule_operation_Stellung_schlageFigur_ExSS2015_expert.evl
    > rule_property_Feld_line_ExSS2015_expert.evl
    > rule_property_Feld_reihe_ExSS2015_expert.evl
    > rule_property_Stellung_matt_ExSS2015_expert.evl
    > rule_property_Stellung_patt_ExSS2015_expert.evl
    > rule_property_Stellung_schach_ExSS2015_expert.evl
    > rule_role_ac2e1_von_ExSS2015_expert.evl
    > rule_role_ac2e2_nach_ExSS2015_expert.evl
    > rule_role_r1e2_amZug_ExSS2015_expert.evl
    > rule_role_r2e2_farbe_ExSS2015_expert.evl
    > rule_role_r3e2_figurenart_ExSS2015_expert.evl
    > rule_role_r4e2_umgewandelt_ExSS2015_expert.evl
    > rule_role_r5e2_gueltigeZuege_ExSS2015_expert.evl
> > ExSS2016_expert
> > ExSS2017_expert
```

Figure 12: Generated *test-set* files of the class diagram task of the summer term 2015 exam. Taken from the Eclipse IDE project explorer.

4.4 Constraint-based Model Test Engine

The *Constraint-based Model Test Engine* uses the *Epsilon EVL engine*. This tool is equivalent to the Constraint-based Model Test Engine mentioned in the former chapter[18]. It generates feedback out of the test-sets that were generated in the previous section[19], the Ecore metamodel file, and the student models.

For that, the main EVL file must be created that controlled the testing process. It can be also called the *main script* program that the test engine executes. That includes the output generation and the order of test-set execution. Listing 8 shows the simplified syntax of such a main EVL file.

```
1
2  import "<Imported Test-Set 1>";
3  ...
4  import "<Imported Test-Set N>";
5
6  pre {
7          // Pre matching work like variable initalization
8  }
9
10 post {
11         // Post matching work like output generation
12 }
```

Listing 8: Simplified syntax of the EVL main file for the test-sets of a expert model.

First, the EVL test-sets are imported (lines 2-4). The order of *import* decided the order of execution of the test-sets. After that, a *pre* block (lines 6-8) can be defined that will launch before the test-set execution. In it, variables can be defined. These variables can be used in any imported EVL file. Last, a *post* block (lines 10-12) can be defined. This block will launch after the last test-set is executed. In there, cleaning work can be done or output can be generated.

The main EVL file is also generated out of an EGL template. It is created at the same time as the other EVL test-set through the EGX file. Listing 9 shows a shortened version of the generator EGL file. First, the *imports* are created by iterating through the model elements. For every not pre-existing element, an import is created using the same naming scheme as in the EGX file (lines 2-11). After creating the imports for the test-sets, the *pre* and *post* blocks must be defined. These contain mostly static formatted text that will be explained in the next section[20]. Only the rule model name (line 20) and the maximal point amount (lines 33-34, 38) must be pre-filled through slots. As an example, the Listing 10 shows the generated main EVL file of the class diagram task of the summer term 2015. That file was generated with the EVL files shown in Figure 12

[18] See Section 3.4 on Page 24
[19] See Section 4.3 on Page 46
[20] See Section 4.5 on Page 64

```

[% for (_class in _model.classes) {%]
 [% if (not _class.isPreExisting) {%]
  [% if (_class.isTypeOf(Class) ) {%]
import "rule_class_[%=_class.name%]_[%=_class._model.ID%].evl";
  [%}%]
  [% if (_class.isTypeOf(Enumeration) ) {%]
import "rule_enum_[%=_class.name%]_[%=_class._model.ID%].evl";
  [%}%]
 [%}%]
[%}%]

...

pre {

        var output = '<?xml version="1.0" encoding="UTF-8"?>\n';
        output += "<TestResult>\n";
        output += ' <TestData>\n';
        output += '  <RuleModel name="[%=_model.ID%]"/>\n';
        output += '  <TestModel name="'+student!OOAClassModel.allInstances.first().ID+'"/>\n';
        output += ' </TestData>\n';
        output += " <Results>\n";

        var pointSum = 0.0;

        var usedRelationships = new Sequence();

}

post {

        [% var expertModelPoints = 0.0;%]
        [% for (e in expert!GradableElement.allInstances){ if( not
             e.isPreExisting){expertModelPoints += e.points;}} %]

        output += " </Results>\n";
        output += " <ResultPoints>\n";
        output += "  <ModelPoints>" + [%=expertModelPoints%] + "</ModelPoints>\n";
        output += "  <TestPoints>" + pointSum + "</TestPoints>\n";
        output += " </ResultPoints>\n";
        output += "</TestResult>";

        if(fileOutput){
                var file =new Native("java.io.File")(outputDir + "OUTPUT_" +
                     student!OOAClassModel.allInstances.first().ID + ".xml");
                var writer =new Native("java.io.FileWriter")(file);
                writer.write(output);
                writer.close();
        }

        output.println();

}

...
```

Listing 9: Shortened EGL file for the generation of the EVL main file for the test-sets of a expert model.

```

import "rule_class_Stellung_ExSS2015_expert.evl";
import "rule_class_Feld_ExSS2015_expert.evl";
import "rule_class_Zug_ExSS2015_expert.evl";
import "rule_class_Figur_ExSS2015_expert.evl";
import "rule_enum_Farbe_ExSS2015_expert.evl";
import "rule_enum_Figureart_ExSS2015_expert.evl";

import "rule_property_Stellung_schach_ExSS2015_expert.evl";
import "rule_property_Stellung_matt_ExSS2015_expert.evl";
import "rule_property_Stellung_patt_ExSS2015_expert.evl";
import "rule_operation_Stellung_fuehreZugAus_ExSS2015_expert.evl";
import "rule_operation_Stellung_schlageFigur_ExSS2015_expert.evl";
import "rule_property_Feld_line_ExSS2015_expert.evl";
import "rule_property_Feld_reihe_ExSS2015_expert.evl";

import "rule_aggregation_r1_ExSS2015_expert.evl";
import "rule_aggregation_r2_ExSS2015_expert.evl";
import "rule_aggregation_r3_ExSS2015_expert.evl";
import "rule_aggregation_r4_ExSS2015_expert.evl";
import "rule_association_r5_ExSS2015_expert.evl";
import "rule_associationclass_ac1_ExSS2015_expert.evl";
import "rule_associationclass_ac2_ExSS2015_expert.evl";

import "rule_role_r1e2_amZug_ExSS2015_expert.evl";
import "rule_role_r2e2_farbe_ExSS2015_expert.evl";
import "rule_role_r3e2_figurenart_ExSS2015_expert.evl";
import "rule_role_r4e2_umgewandelt_ExSS2015_expert.evl";
import "rule_role_r5e2_gueltigeZuege_ExSS2015_expert.evl";
import "rule_role_ac2e1_von_ExSS2015_expert.evl";
import "rule_role_ac2e2_nach_ExSS2015_expert.evl";

pre {

        var output = '<?xml version="1.0" encoding="UTF-8"?>\n';
        output += "<TestResult>\n";
        output += ' <TestData>\n';
        output += '  <RuleModel name="ExSS2015_expert"/>\n';
        output += '  <TestModel name="'+student!OOAClassModel.allInstances.first().ID+'"/>\n';
        output += ' </TestData>\n';
        output += " <Results>\n";

        var pointSum = 0.0;

        var usedRelationships = new Sequence();

}

post {

        output += " </Results>\n";
        output += " <ResultPoints>\n";
        output += "  <ModelPoints>" + 27.5 + "</ModelPoints>\n";
        output += "  <TestPoints>" + pointSum + "</TestPoints>\n";
        output += " </ResultPoints>\n";
        output += "</TestResult>";

        if(fileOutput){
                var file =new Native("java.io.File")(outputDir + "OUTPUT_" +
                        student!OOAClassModel.allInstances.first().ID + ".xml");
                var writer =new Native("java.io.FileWriter")(file);
                writer.write(output);
                writer.close();
        }

        output.println();

}

operation setOutput(rs:String, r:String, c:String, to:String, ro:String, p:Real, msg:String){

                                        output += "  <Result>\n";
                                output += "    <TestObject>" + to + "</TestObject>\n";
                                output += "    <RuleObject>" + ro + "</RuleObject>\n";
                                output += "    <RuleSet>" + rs + "</RuleSet>\n";
                                output += "    <Rule>" + r + "</Rule>\n";
                                output += "    <Category>" + c + "</Category>\n";
                                output += "    <Points>" + p + "</Points>\n";
                                output += "    <Msg>" + msg + "</Msg>\n";
                                output += "  </Result>\n";

                                pointSum += p;

  }
```

Listing 10: EVL main file for the *test-sets* of the expert model. Generated for the aUML class diagram task of the summer term 2015 exam.

If this main file is existing, the EVL engine can be used to generate feedback from student models. The engine needs the main EVL file, the Ecore aUML class metamodel, and the student solution as parameters. Normally, the feedback for the student is sent to the standard console output. By setting two additional parameters (*fileOutput:Boolean*, *outputDir:String*), an additional XML file can be generated. A complete user guide for the testing in an Eclipse IDE can be found in the attachments of this work.

4.5 Feedback Generation

At last, the generation of feedback for the student must be discussed. In the proof-of-concept implementation, the output generation is implemented as simple as possible. The feedback generation is mostly implemented in the main EVL file. It generates the same XML structure that was discussed in the previous chapter[21]. There are three components.

The first component is the *pre* block shown in Listing 10 line 33 till 47. There, an *output* variable is defined before the execution of the test-set (line 35). This variable is then filled with the first part of the output XML file structure (line 36-41). This includes the expert models and the tested model's ids. Also, a variable for the granted points (line 43) is initialized.

The second component is the provided *setOutput* operation (lines 69-83). This operation takes in feedback data from a constraint and generates a corresponding XML *Result* structure. It also added the points, granted from the constraint, to the point amount of the solution. This operation is available for all imported test-sets and is called in the message block of their constraints.

Last, in the *post* block (lines 49-67), the XML structure is closed with the addition of the maximal and granted points (lines 51-56). Then the output is printed to the standard console output (line 65). If desired, a file output is generated using standard Java libraries (lines 58-63).

There is an advantage of using the EVL main file of a test-set collection, to generate the feedback. The generation algorithm is so separated from the constraints, making it easy to change the feedback generation method later. It would also be possible to outsource the feedback generation to its one EVL file if even more flexibility is needed. It must also be said that the current way the feedback generation is not optimal. Especially in terms of performances, all three components could be streamlined further. In the proof-of-concept implementation, this was purposefully not done. An easy to understand algorithm was prioritized.

[21] See Section 3.6.4 on Page 36

4.6 Critic and Discussion

The proof-of-concept implementation described in this chapter successfully implemented the INLOOM CAS matching engine for aUML class models.

It provided a possibility to generate expert models and digitize student solutions through the EMF framework. Through, at the moment only the rudimentary EMF standard editor is available. This editor lacks the comfort of a real graphical or textual editor and is only a temporary solution.

The use of the Epsilon tool set works fine for the definition of the master constraint-sets. The generation of the test-sets with EGL works dynamically. This means that all input data is not compiled and can be changed independently. Only the main EGX file is interwoven to the master constraint-sets but is in itself also easily changeable. So, the Constraint-based Model Test Generator has no static constraints defined. The same can be said for the EVL Test Engine which uses the textual test-sets. These are changeable and, as long the main EVL file is updated, easily extendable. It also has no compiled static constraints. It must be said that the implementation of the constraints themselves could be streamlined much further. That is due to the introduction of a new framework and mindset. Nonetheless, the generation of test-sets out of old exam tasks solution is done instantly, even on less powerful machines. The same is true for the testing itself, meaning the performance, for a prototype system is sufficient enough. The performance in grading will be evaluated in the next chapter.

Sadly, the Constraint-based Model Test Engine is not completely independent of anything but the test-sets and the student model. Through the limitation of Epsilon, the Ecore metamodel file is still needed. This limits the separation layer between both steps.

The system can create all necessary feedback for the student and the instructor. It can collect textual feedback and points from each met constraint. These points will generate a grade together with the maximal points of the expert model. All this is used to generate the complete feedback in XML format. The feedback creation is also mostly separated from the constraints. This means the feedback generation algorithm can be easily changed.

Last, the system uses, at the moment, forced labels with only lowercasing. That is enough to reach the defined requirements. In the future, complex label matching can be introduced easily. Normal algorithms like *Levenshtein* [17] can be implemented like the *setOutput* operation[22]. That can be happening in the main EVL file or in an EVL file of its own. Such, the label matching could be very variable. Since Epsilon allows access to Java libraries, in the future, dedicated label matching frameworks could also be used to some extent. The limitation for that is only that it would be hard to combine structure-sensitive label matching with the current system.

In conclusion, the proof-of-concept realization of the INLOOP CAS matching engine can realize many of the concepts of the previously discussed design in a sufficient way. Sadly, it can not completely separate the model type-dependent test generation from the later testing step.

[22] See Section 4.5 on Page 64

5 Evaluation

After the realization of the *INLOOM CAS aUML class model* test engine, the next step is to evaluate the decisions made in the last chapter. Two points can be evaluated. The first is the quality of the unified grading scheme for aUML class diagrams. The second is the quality of the assessment results of the class model testing engines.

5.1 Evaluation of the Uniform Grading Scheme

The quality of the unified grading scheme is important for the acceptance of the system by the students. The *unified grading scheme*'s scores could be too far from the scores of exam grading schemes. In that case, the students would feel that INLOOM would not help them to assess their modeling skills for the exam. That could be fatal since this is one of the main goals of INLOOM. But the alternative, changing grading schemes for every task, could confuse the students, too. This could also lead to the inability to asses their skills. So, a uniform grading scheme in necessary and the difference range to the original grading schemes must be explored.

To evaluate if the uniform grading scheme is close enough to the exam grading schemes, both schemes are compared. For that, the last 9 exams expert solutions for class model tasks were digitized and the maximal score was generated through the implemented test generator engine. These exams were the most recent ones with most students using them to prepare for the exam. The older the exams are, the more they lose prominence to the students. So there is a high probability that these nine exams would be the main comparison target for INLOOM tasks. The exam of the winter term 2017/2018 (WS 17) was dismissed in the choosing process since it was using a completely different grading scheme. This would not yield any comparable results to the rest of the exams. After this, the Difference between the original maximal score of the exam and the maximal score of the uniform grading scheme was calculated. The results can be found in Table 14. The digitized exams expert solutions could later be used for testing purposes or INLOOM tasks. They are included in the digital data set of this thesis.

It can be seen that the uniform grading schemes maximal scores are always higher than the original exam scores. That is expected since normally roles are not gradable elements themselves but parts of points for relationships. So, the more roles are in the exam task the more points the uniform grading scheme differs from the original score. Other factors are the attributes and operations. These elements often only collectively reward points. Those collective points for multiple elements of an element type are often used to reduce the maximal score to a necessary level. That is because exam task scores must fit into a specific score range. The more an exam uses them in its original grading scheme, the more the uniform grading scheme score differs. This difference is always positive. Only the summer term 2019 exam has an equal score to the uniform grading scheme. That is the case because both use the same scheme.

Table 14: Evaluation of the uniform grading scheme of aUML class models against old exam grading schemes. *Task* gives the number of the aUML class model task in the *Exam*. *Exam Points* are the maximal scores of the exam task using the original grading scheme. *Uniform Points* is the maximal score using the uniform grading scheme. *Difference* is the percentage of the uniform points to the exam points. Last, the *Average Difference* is the arithmetic average over all exams.

Exam	Task	Exam Points	Uniform Points	Difference
SS 15	2	22	27.5	125%
WS 15	1	38	47	124%
SS 16	1	20	23,5	118%
WS 16	1	26	32,5	125%
SS 17	1	25	29	116%
WS 17	1	32	-	-
SS 18	1	18	27	150%
WS 18	1	24	27	113%
SS 19	1	13	13	100%
			Average Difference:	121%

Over all eight exams, the uniform score is roughly 20% above the original grading schemes scores. The range goes normally between 0% and 25% difference. There is only one exception. The exam of the summer term 2018 (SS 18) has a 50% difference between the original maximal score and the uniform maximal score. After checking the exam, it became clear why there is though a great difference. The exam *"SS 18"* is probably a worst-case scenario for the uniform grading scheme. It gives points to only relationships and these relationships have mostly roles attached to them. It also gives no points to other elements to equalize the final maximal score. With the detachment of roles from relationships in the uniform scheme, this generates a large number of points. This should also mean that the maximal difference should not be much higher than the 50% of this exam.

In conclusion, the difference between the grading schemes is normally a 0% to 25% difference in maximal scores. The best-case scenario is that both, the original and the uniform scheme, using the same grading scheme. Then the difference is 0%. In extreme cases, the difference can go up to roughly 50%. That should be only the case for worst-case scenarios like the "SS 18" exam. But it only happens once in the exams of the last four years.

In literature, nothing can be found on this topic, so there is no comparison to other cases. In any case, compared to the advantages a uniform grading scheme gives, an average difference of 20% to old exam schemes should be sufficient enough. So, the uniform grading scheme for aUML class models can be used by the INLOOM Model Test Engine. To reduce possible future problems, a transparent approach could be used by telling the students that a uniform grading scheme is used.

5.2 Evaluation of the Quality of the Constraint-Sets

One of the most important evaluations is the quality of the *mater constraint-sets* used in the implemented INLOOM aUML class model test engine. To assess the quality, the results of the automatic assessment are compared to the results of exam task solutions. These exam task solutions were manually assessed by an instructor.

5.2.1 Preparation of the Exam Solution Data

For this evaluation, 30 student solutions for old exam tasks were chosen. These 30 solutions were separated into 3 solution sets. Each set holds 10 solutions for one task of one old exam. The tasks are naturally all model task for aUML class models. The chosen exams were the summer term exams of 2017 (SS 17), 2018 (SS 18), and 2019 (SS 19). There are three reasons for using these exams. First, they are the most recent exams. Second, they are all summer term exams which are attended by the highest number of students. Last, they represent a worst-case (SS 18), a best-case (SS 19), and a normal case (SS 17) in regards to the uniform grading scheme. For each set, 10 student solutions were randomly chosen out of all available solutions. The only real limitation on the randomness was that in the end three of the solutions must be solutions with high scores, three must be of low scores, and four must be in between. That was done to reach a broad representation of possible scores. Also, 0 points and full point solutions were avoided since matching them would be trivial. These student solutions were numbered 1 to 10 for each exam year and ordered after their score. Obviously, the students numbered the same (for example Student 1) in different years are not the same person.

After that, the student solutions were digitized using the generated *EMF* editor. Since the solution was originally written on paper, the students did not necessarily strictly follow the aUML metamodel. In the case of a human instructor, this often resulted in a one-time point reduction for the whole solution. The not metamodel conform elements would then be assessed normally. In the process of digitizing, this was simulated as much as possible. All non-usable elements were transformed into metamodel conform elements, giving the student the benefit of the doubt.

For each solution, the score given by the original instructor was noted. To simulate the none metamodel elements, the end score was raised by the deducted points for none metamodel conform elements. Also in the exam correction, there are "*grace points*". These are used to mark points that are given for borderline wrong elements, out of goodwill from the instructor. Mostly, a given grace point means 0.5 points more than in a case without grace points. These points are used to simulate a range of the score given by a human instructor to a solution. The score with grace points is called the *Exam Score* and represents the highest score the human instructor would give for a solution. The score without grace points is called the *Clean Exam Score* and is the lowest possible score the human instructor would give for a solution. The Clean Exam Score is calculated by reducing the Exam Score by 0.5 points for every grace point given.

After this, the already digitized exam expert solution models are now used in the implemented *Epsilon Constraint-based Model Test Generator* together with the implemented *master constraint-sets*. The generated *Test-Sets* were then used together with the digitized student solution in the *Epsilon Constraint-based Model Test Engine*. This generated the *Test Score* for the student solution as well as the *feedback XML files*. The feedback XML files, as well as the digitized student solutions, can be found in the digital attachments of this thesis.

It must be said that this process was done again to fine-tune the strictness of the constraint test sets. But no constraints were added or removed. An example of that is that, in the first run, a relationship could be found in an Error constraint if the multiplicity of both ends were right. That was done to avoid to be too lenient. In the end, that was to strict and the instructors often gave the point if one multiplicity was right. It resulted in, on average, over 30% fewer relationships were rewarded with partial points then by the instructor. So in the second run that was changed to one of the multiplicities must be right, to give better feedback.

All data was transformed to percent values to their maximal score. This was done to make the manual human given scores and the automatic test scores comparable to each other. The scores given by the human instructor are based on the original grading scheme of the exam, while the automatic assessment engine uses the uniform grading scheme. By calculating their percent-based scores both scores become roughly comparable. But since their base is different, conclusions must be done carefully. Another used value is the *Difference* between the Exam Score and the Test Score. It shows how many percentage point the Test Score and the Exam Score are away from each other. A positive Difference means that the automatic assessment has given the student a higher score (in percentage points) than the human instructor. This means the human instructor was more strict as the test engine. The opposite is that the test engine is more strict than the instructor, resulting in a negative Difference. A Difference close to null is preferable. This means that the human instructors and the automatic test engine's assessment is nearly the same. The whole data set can be found in the digital version and the attachments of this thesis.

Following the results based of the last run of the test engine are discussed. The feedback of the assessment system was additionally manually confirmed for roughly half the student solutions. That was done to find possible errors in the assessment system.

5.2.2 Comparison between Automatic and Human Assessments

Figures 13, 14, and 15 show an overview of the comparison between the scores of a human instructor and the automatic assessment engine. The blue bar shows the Exam Score of on Student and the red bar the corresponding Clean Exam Score. The green bar gives away the Test Score of the automatic assessment engine. All bars using the left axis and show percentage values to their corresponding maximal scores. The orange line shows the Difference between the Exam Score and is using the right axis. The scale of the right axis is the same over all three diagrams. This was done to make comparing this value easier. Even if shown as a line in the diagrams, the Difference values are independent of each other. The line between the Difference values per student was included to make it easier to find patterns between the values along the decreasing scores.

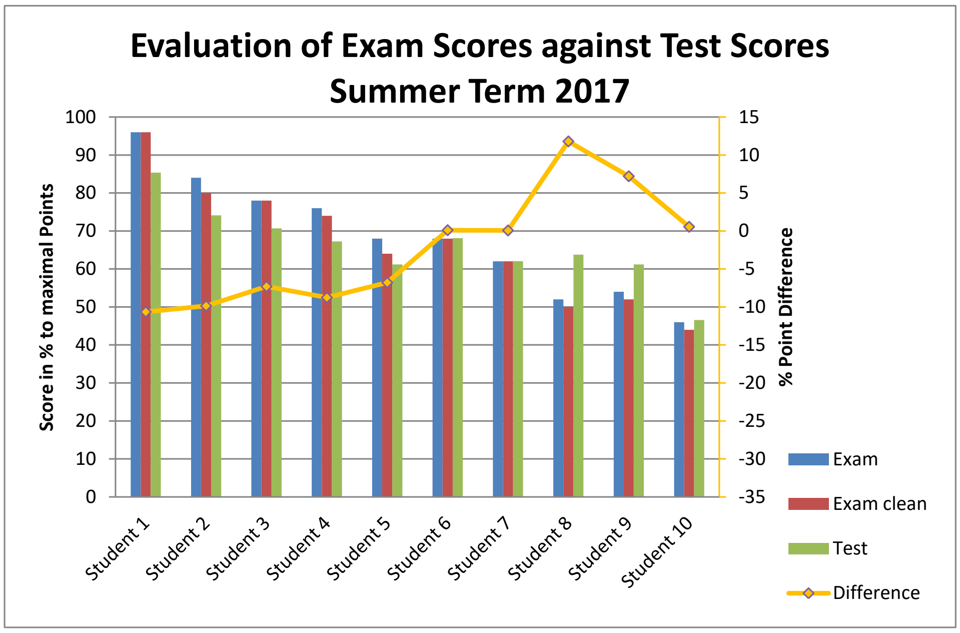


Figure 13: Evaluation of the Automatic Assessment System against the manual assessment of an instructor. The base are ten student solutions of the summer term 2017 exam. The bars show the *Score* as percentage of the maximal score. There are the exam score (blue bar), cleaned exam score(red bar), and the score of the automatic assessment (green bar). Clean exam scores are scores without "*grace points*". The orange line shows the *Difference* between exam and automatic assessment in percent points.

The first thing to mention is the varying range of the maximum exam score and the minimum exam score between the diagrams. The maximum score of the student solutions of an exam is always above 80% and below 100%. The minimum score of the student solutions of an exam lies around 40% in 2017 and 2019 but is roughly 15% in 2018. That difference can have a reason other than random selection. In 2017 and 2019 class and attribute elements were asked in the solution for around half the points. These elements are normally easier to find. In 2018 only relationships gave points which are harder to model right. This could result in a smaller score range in the 2017 and 2019 exams. This means that the score range of below roughly 30% is underrepresented in the evaluation.

The next focus point is the difference between the Exam Score and the Clean Exam Score. This difference is mostly only around or below 5%. That means, the human instructors were mostly certain of their grading and only sometimes found borderline elements. The only exception is the solution for Student 4 in the year 2019. There, the instructor was extremely uncertain about his grading and the difference between Exam and Clean Exam Score was roughly 10%. The student modeled around an abstract element that, in this context, was not meant to exist in the task description.

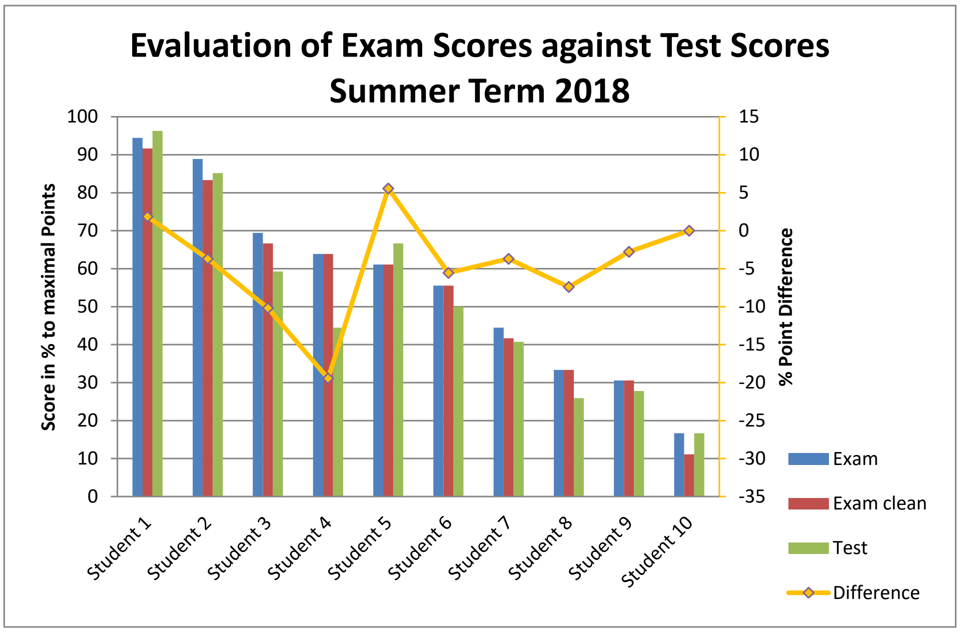


Figure 14: Evaluation of the Automatic Assessment System against the manual assessment of an instructor. The base are ten student solutions of the summer term 2018 exam. The bars show the *Score* as percentage of the maximal score. There are the exam score (blue bar), cleaned exam score(red bar), and the score of the automatic assessment (green bar). Clean exam scores are scores without "*grace points*". The orange line shows the *Difference* between exam and automatic assessment in percent points.

The most interesting evaluation is the Difference between Exam and Test Score. This value shows the similarity of the test engine to a human instructor. Over all three data sets, the difference of the test engine seems to be mostly between 10 percentage points from a human instructor. This means that the assessment system grading is near the human instructors grading. The arithmetic average in the Difference of all solutions is -4 percent points. It seems that the assessment system is slightly stricter than the human instructor. That should be normal since the human is able to find more semantic in the models as the constraint-based system. But it also seems that the system overall is not configured to lenient.

Looking at the development of the Difference with decreasing scores, there seems to be no causation. In the 2017 data, it seems that the assessment system is more strict in the high score solution and more lenient in the lower score section compared to a human instructor. But in other year's data, the overall Difference is more average over all scores. It also often switches between positive and negative differences from one score to another. If looking into the data in detail, all three data sets have another curve. In 2017, the differences become clearly more positive(more lenient compared to human) the lower the score is. In 2018 the differences are on average lower (stricter compared to human) in the high score and low score sections. Last, in 2019, the differences become on average lower (stricter compared to human) the lower the score section is.

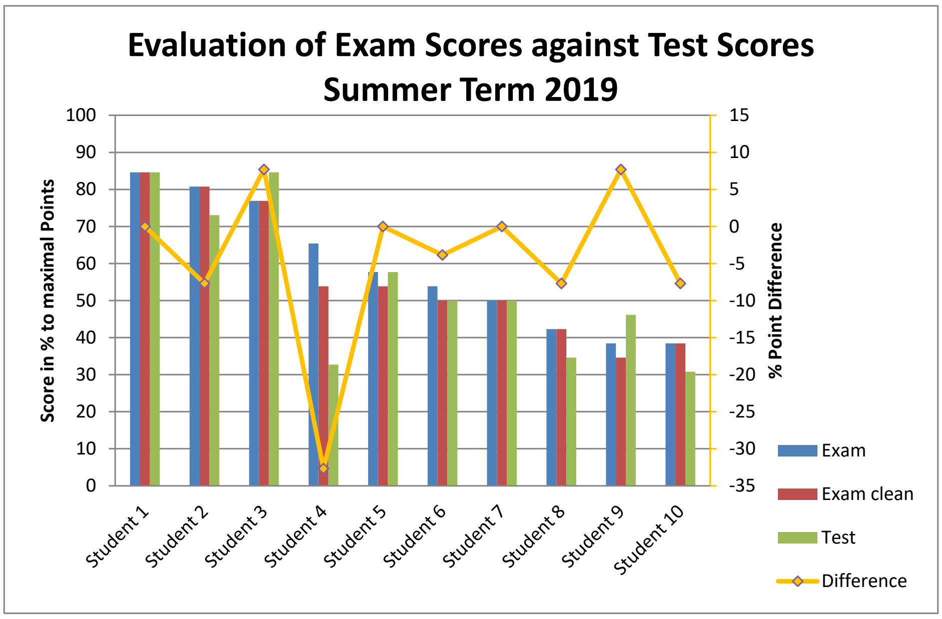


Figure 15: Evaluation of the Automatic Assessment System against the manual assessment of an instructor. The base are ten student solutions of the summer term 2018 exam. The bars show the *Score* as percentage of the maximal score. There are the exam score (blue bar), cleaned exam score(red bar), and the score of the automatic assessment (green bar). Clean exam scores are scores without "*grace points*". The orange line shows the *Difference* between exam and automatic assessment in percent points.

All that seems to indicate that the strictness of the assessment system is not dependent on the score value. That means the master constraint-sets are sufficiently configured in the strictness of their trigger. With the data like that, in 2017, it seems more likely that the human instructor was more in favor of good solutions and became stricter when grading poor solutions. Other possibilities could be a second instructor or a change in personal grading practices. But it must be said that the difference is not that large.

There are two student solutions that have a larger distinction in Difference than the rest. First is the solution Student 4 of the data set of 2018. It has a Difference of negative 19 percent point to a human instructor. That is nearly double the Difference to each of the other student solutions. Because of this exception from the norm, the feedback was looked into in detail. It seems that the assessment system and the human instructor agreed almost always in the assessment of the solution. So, the difference between humans and the system was not that great. The greatest portion of the later Difference comes from different grading schemes. It seems the student solution lacks most of the roles. The original grading scheme would deduct points from their relationships. The system also deducted points of the relationships because of other errors that also existed in the solution. Because of the missing roles, more points were lost. The huge Difference comes from this worst-case scenario for the student, caused by the change in the grading scheme. That concludes that the Difference of the solution of Student 4 is not the automatic assessment systems fault.

The second exception is the solution of Student 4 of the data-set of 2019. For this data point, the Difference is -33 percent points. This data point was already mentioned above for its great difference in Exam and Clean Exam Score. This was caused by the complicated modeling of the student around a not intended abstract element. The system could not resolve this part of the model, only rewarding minimal points. That leads to a -33 percent point Difference to the Exam Score or a -21 percent point Difference to the Clean Exam Score. This shows that the system, at the moment, can only deal with small structural or semantic changes from the expert model. But, it should be mentioned that the human instructor was also not clear on how to grad it or was very generous with its grading. That is shown by the great difference between the Exam Score and the Clean Exam Score.

In conclusion, there was only one solution of the 30 student solutions that the implemented automatic assessment system could not assess in a sufficient range to the human instructor. For a proof-of-concept implementation, that should be a sufficient ratio.

Since the different grading schemes introduce a level of uncertainty to this evaluation, no further statistical methods were used. That was to prevent conclusions based on small findings due to an uncertain database.

5.2.3 Conclusion

In the literature, there are only a few systems that published concrete data on the difference between the system and human instructors.

There is only one other constraint-based system that shows the data of their evaluation [6]. It is also used on class diagrams but there are no given restrictions on the used UML metamodel, like aUML. *Bian et al.* [6] mention that their algorithm-based constraint-based system can asses student solution with an, on average, less than 14 percentage points difference to a human instructor. This result should be comparable to the tested implementation of INLOOM, through the UML specification of *Bian et al.* is broader.

There are other systems from the literature that provide data as well [24, 29–32]. The problem is that these systems have other matching techniques like similarity matching. These types of systems have high accuracy if their thresholds are trained enough. They also use only examples with small scores like 6 to 8 points for their evaluation. That decreases the comparability with the evaluated implementation. But, a rough outline can be collected.

In these pieces of literature, every difference under 5 percent points seems to be a great result, and roughly 10 percentage points seem to be considered good, too. That was concluded from the point that only results of resulting in roughly these numbers are mentioned in the literature. From *Bian et al.* [6] as well, it can be considered that less than 15 percent point is also sufficient enough.

From this consideration and the evaluation of the last section[23], Table 15 was created.

[23] See Section 5.2.2 on Page 70

Table 15: Distribution of the difference between manual instructor assessment and automatic assessment for the 30 student solutions. The *Difference* is the difference in percentages points between the manual and automatic assessment. The count row shows the number of solutions in this Difference range. *Count %* shows the percentage of the number to the maximal number of solutions. *Cum. %* calculate the number of solutions that have equal or less Difference as a percentages value. A value of below 10% Difference is considered a good value and below 15% a sufficient value.

Difference	0%	0%<x<5%	5%<x<10%	10%<x<15%	>15%
Count	6	6	14	2	2
Count %	20%	20%	47%	7%	7%
Cum. %	20%	40%	87%	93%	100%

It can be seen that, with under 5 percentage points Difference, 40% of the assessment results of the implementation of INLOOM can be considered great assessments. Another 47% are good matches that reach under 10 percentage points. In the direct comparison with *Bian et al.* [6], 93% of the resulted feedback is within their mentioned threshold of 14%. This number goes up to 96,5% if the result of Student 4 in the year 2018 is considered inside this range. This should be the case since the Difference of 19 percentage points was due to the different grading schemes.

As a conclusion, the proof-of-concept implementation of an aUML class diagram assessment system for INLOOM can favorably compare to other assessment systems. It reaches the number of roughly 95% of assessments with a less than 15 percentage points Difference to a human instructor. This can be considered a good result.

But, it must be also mentioned that the majority of results are in the higher and only considered "good" range between 5 percent to 10 percentage points Difference. There is also one result that can be considered a failure that has over 30 percent points difference. So it can be said that the quality and number of constraints should be improved further to increase the accuracy of the system. This could increase the quality of the assessments even further.

6 Conclusion and Future Work

After designing the *INLOOM CAS* and implementing the *aUML class model* assessment engine, a conclusion must be drawn. For that, the Research Questions that were defined at the start of this thesis, are answered in this chapter. Also, possible future work based on this work will be discussed.

6.1 Research Questions

To assess if the thesis has reached its goals, the findings of this work are used to answer the defined Research Questions.

RQ1 Is there a method that can be used for the automatic correction and grading of student solutions in the beginner software engineering course?

This Research Question can be answered with yes.
There are multiple types of assessment systems that can be used for grading and feedback generation on student solutions. These types are *Element-* and *Execution-based Matching* if there are high limitations on the student's modeling. Systems for free modeling of UML diagrams are *Learn-* and *Similarity-based Systems*. For systems that are between these extremes, there are *Constraint-* and *Label Matching-based Assessment Systems*. The systems are described in detail in Section 2.2
Beginner software courses often use limitations on the student's modeling but need some configurable freedom since beginners often make mistakes. That leads to Constraint-based Assessment Systems as the best choice. These systems are also capable of configurable grade and feedback generation.

RQ2 How could a design for the architecture and workflow of the automatic assessment system look like?

In Chapter 3, a design for such a system is presented.
It uses an expert model and *master constraint-sets* in a *Constraint-based Model Test Generator* to generate textual constraint-based *test-set*. These test-sets are each based on one element of the expert solution and can be easily interchanged or expanded with other test-sets. The collection of test-sets is then used with a *Constraint-based Model Test Engine* to assess a student model solution. At the end, feedback and a score are generated as an XML file.
This system fulfills all requirements that were found for the automatic assessment system.

RQ3 How can the system be implemented in the existing INLOOP architecture, without changes to the architecture?

Section 3.5 gives a overview how the system defined in Chapter 3 can be integrated in *INLOOP* [20].

The design of the expert models and master constraint-sets could be done on the machine of the specific instructors. The instructors can then commit their work on a *Git Task Repository*. INLOOP/INLOOM will then sync with the repository and publish the task description to its *Web Application*. At the same time, the Model Test Generator will be called and the test-sets will be generated. When a student uploads its solution, a *Background Worker* will receive the test-sets, student solution, and the Model Test Engine and generate an output. This output can be now archived and given to the Web Application to extract score and feedback messages.

This workflow coincides with the existing INLOOP architecture and the *continuous publishing* workflow.

RQ4 Which data can be extracted from the existing data sets and workflows? How can this data be reused to help the design process?

The existing data that could be reused in the process of this thesis are already existing aUML metamodels and grading schemes. But, the most important data to reuse are the data set around the old exams. The exams can be reused as tasks in INLOOM and their solutions as expert models. The archived student solutions can be investigated to find common mistakes and misconceptions of beginner students. These can be then used to create and refine the master constraint-set. Digitized, the student solutions can be testing data for these constraint-sets, to ensure a certain quality of the assessment.

RQ5 Is the presented realization suitable for the goals of the system?

The realization of the aUML class diagram assessment system seems suitable for the goals of the system. It completely realized the design that was found to fulfill the requirements of the desired system. Through an evaluation in Chapter 5 with student solutions of old exams, it was found that it has sufficient accordance with human instructors. The difference to human instructors was below 15% with roughly 90% of the results below 10%. A comparison with the information found in the literature shows that these results are considered a successful automatic assessment.

6.2 Future Work

There are a few questions and topics that could not be included in this thesis but are connected to INLOOM. These topics could be great starting points for research based on this work in the future.

Web-based UML Modeling Editor

The first topic is a web-based UML modeling editor that could be integrated into the INLOOM web application. This editor must be configurable to include aUML and normal UML. It should also easily expendable with new (a)UML modeling types. There is also the possibility to extend the INLOOM CAS to non-UML model types. A search for existing options would be needed and, if no suitable product is found, a new one must be designed and implemented.

Test Suite for Master Constraint-Set

The second topic is to secure the quality of the master constraint-set while they are extended and refined. This must be done to increase the covered solution space of the assessment system, as well as decrease the range to human assessments. It is also vital when designing master constraint-sets for new model types. To do this a testing suite is needed. This suite must be designed to be easily manageable and must be populated with test data. This data can be collected out of the solutions of old exams. It should be part of the development environment of the instructor, which develops the constraints.

Assessment of Textual Task Descriptions

One of the biggest error sources of the new system are the textual task descriptions. In the assessment of exam solutions, it often happens that these descriptions are not clear enough or contain errors. This results in an increase of the possible solution space and increases the difficulty of automatic assessments. Since the expert models in INLOOM are based on metamodels, this could be resolved by comparing the models and the description. One way, a modeled expert solution could be compared to the description text and possible problems could be marked in both. Another way could be the generation of one out of the other. For example, an expert model could exist first and out of this model, an example description text could be generated. The instructor could now refine the text and compare it to the model to find problems arose from the refinement. The opposite would also be possible. An instructor writes the description text and then generate a starting model. This model is then refined and compared again to the description.

This system would be a great expansion of the task instructor's development environment, greatly increasing the assessment quality.

Motivation and Plagiarism

With the start of INLOOM, the point of motivating the students to use the system will become a problem. INLOOP solves this problem with bonus points for the next exam that are granted for solving exam tasks. This could also be done by INLOOM. The problem of this is how to find fraudulent submissions. INLOOP uses a textual plagiarism checker to find those submissions. In INLOOM, the submissions are in the form of an Ecore/EMF file which is an XMI format. These files are also generated and not directly written by the student. This makes them less individual as source code. There are attempts to find plagiarism by using similarity matching on all submitted solutions [19]. That could work, but aUML models have relatively few features and the task description and labels limit the students modeling. So there are concerns that these approaches could not be working correctly.

Another option would be to limit the viability of plagiarism. That could be done through the editor by disallowing copy and paste operations or marking solution elements. Another way would be to use the flexibility of the INLOOM test-sets to generate not one task but a group of similar tasks around one main task expert model. These sub-tasks differ only by one or two elements. The students then each get one of these sub-tasks. The testing would be performed by swapping one or two test-sets before the test, with most of the test-sets staying the same. That would increase the difficulty of the simplest form of plagiarism. To look into all these options would necessary to include bonus points to INLOOM.

6.3 Conclusion

University education has become more and more popular with student numbers rising year after year. To cope with this increase, e-learning has become an important part of new educational practices. INLOOP is an e-learning system that helps participants of beginner software engineering courses to train their object-oriented programming skills. This helps them to prepare for the exams of these courses, too. Since INLOOP helped to increase the quality of the exam results, the same concept is hoped to be achieved for object-oriented modeling. This thesis gives an overview of possible automatic assessment systems types that could provide this. It also introduced the concept of INLOOM. INLOOM is a variable constraint-based model assessment system build onto the INLOOP architecture. It has a two-stage system to separate the publishing and preparing of tasks from the testing of student models. It let course instructors define tasks, expert solution models, and model-type-dependent, highly configurable master constraint-sets. These are used to generate a textual test-set for each task. These test-sets are later used to, separated from the first step, assess committed student solutions. It can use the constraints to reward student solution elements with partial or full points and is able to search for alternative solution patters. For the students, it generates a grade and textual feedback as help. As a proof-of-concept, an analysis UML class diagram assessment system, using EMF and Epsilon was implemented. This realization was compared with the grading of human instructors using 30 student exam solutions from three separate exams. It was shown that the proof-of-concept realization for class diagrams could compare favorably with other systems described in the literature. The system was able to assess student solutions within 15% range to the score given by a human instructor. This range could also decrease in the future by refining and extending the used constraints. This shows that INLOOM can support beginner software engineering courses with e-learning capabilities for object-oriented modeling.

Table 16 shows an overview of INLOOM's information in contrast to other model assessment systems found in the literature. It can be seen that it uses the same concepts and the same concept combination as other systems. Its advantage over other systems is the two-stage workflow. Through this workflow, it can remove many problems of other constraint-based systems. First, often constraints are more general global constraints or the task itself must be defined in task-specific constraints. In INLOOM, the more complex and specialized definition of the constraints can be done by a separate specialized instructor for each model type. The solution of the task can be done by a more simple modeling of an expert solution model. This is more accessible for the majority of instructors. Due to combining these two in the Constraint-based Test Generator, both the global and the task-specific constraint test-sets can be generated. This makes it easier for the instructors to define tasks and makes the constraint definition more variable. The second advantage is that, due to the second stage, the Constraint-based Model Test Engine is not model type-specific, and the test-sets can be easily replaced. In conclusion, INLOOM is a very flexible assessment system that can be easily changed and extended. Third, to our knowledge, INLOOM is the only system that can easily be integrated into the continuous publishing paradigm of INLOOP. These three advantages help INLOOM to differentiate itself from other existing model assessment system.

Table 16: An overview of model assessment systems, including the new INLOOM system. The table is sorted from the newest entry to the oldest one.

Paper	Type		Diagramtype	Editor		Metamodel	Methodologies											Feedback	Grades
							Labels		Source			Matching					Learning		
									Expertmodel										
	System	Method		Tool	Web		Forced	Matched	Single	Multiple	Constraints	Element	Constraints	Graph	Similarity	Execute			
INLOOM 2020	**x**		**CD/(general)**		**x**	**(EMF)aUML/ (EMF)UML**	**x**		**x**		**x**		**x**					**x**	**x**
Bian 2019 [6]		x	CD	x		(EMF)UML, Grades	x	x	x			x	x					x	x
Vachharajani 2019, 2014 [33, 34]		x	UC	x		UML		x	x			x		x				x	
Bernius 2019, Krusche 2018 [5, 14]	x		general		x	UML				x							x	x	x
Beck 2015 [4]	x		CD/AD		x	UML	x				x					x		x	x
Sousa 2015 [26]		x	general/AD	x		Graph, UML			x					x	x				x
Striewe 2014 [28]		x	AD			UML					x					x			x
Smith 2013, Thomas 2007 [24, 30]	x		general/ER	x		Graph, ER		x	x						x				x
Schramm 2012 [23]	x		CD/AD		x	(Argo)UML	x		x			x	x					x	
Prados 2011 [22]	x		general		x	Graph				x				x			x	x	x
Hasker 2011 [8]	x		CD	x		(RR)UML	x		x			x	x					x	
Striewe 2011 [27]		x	general			UML	x				x		x	x				x	
Soler 2010 [25]	x		CD		x	(text)UML	x			x		x						x	
Demuth 2009 [7]		x	CD	x		(EMF)UML	x		x		x	x	x					x	
Jayal 2009 [12]		x	AD			UML		x	x			x							
Thomas 2008 [31]	x		SD	x		UML			x						x				x
Baghaei 2007 [3]	x		CD		x	UML	x		x		x	x	x					x	
Ali 2007 [1, 2]		x	CD	x		(RR)UML	x		x			x						x	
Le 2006 [16]		x	CD	x		(Argo)UML	x		x		x	x	x					x	
Higgins 2006, 2002 [9, 10]	x		general	x		ER/OO	x				x		x					x	x
Tselonis 2005 [32]	x		genaral	x		Graph			x					x	x			x	x
Hoggarth 1998 [11]	x		general	x		CASE	x		x			x						x	

A Attachments

A.1 User Guides for the Proof-of-Concept Implementation

A.1.1 Requirements

The proof-of-concept implementation has the following requirements and is only tested for the given operating system and tool versions. The implementation can work on other versions, too.

Operating System
Windows 10

Eclipse Modeling Tools
Version: 2019-09 R (4.13.0)
Download: https://www.eclipse.org/downloads/packages/release/2019-09/r/eclipse-modeling-tools

Epsilon
Version: 1.5.1
Download: Eclipse Marketplace

All following user guides requiring basic understanding in working with the Eclipse IDE.

A.1.2 Creating aUML Class Models

The following user guide describes how to use the proof-of-concept implementation to generate a basic EMF editor for aUML class models and use it to create aUML class models.

1. Generating the aUML Class Model Editor

1. Download the Eclipse Modeling Tool Packages and unpack it.
2. Start the Eclipse IDE (eclipse.exe) and create a new Workspace.
3. *File–>Import...–>General–>Projects from Folder or Archive.*
4. Under *"Import source"* click on *Directory* and choose the *inloom.emf.ooa.classdiagram folder* (it can be found in the digital attachments under *Implementation/emf/*)
5. Click on *Finish*
6. The imported EMF project can be found in the *Model Explorer*
7. In the project the metamodel data can be found under *model–>ooa_ classdiagram.ecore.*
8. Open *ooa_classdiagram.genmodel* and right-click on *Ooa_classdiagram.*
9. Generate the EMF Implementation of the metamodel and the editor with *Generate All.*
10. Two new projects that implement the editor should be generated: *inloom.emf.ooa.classdiagram.edit* and *inloom.emf.ooa.classdiagram.editor*
11. Right-click on the *inloom.emf.ooa.classdiagram* project and then on *Run As–>Eclipse Application.*
12. Click on *Continue* and a new Eclipse instance should open. This instance includes the working EMF editor for aUML class models.

2. Creating a new aUML Class Model

1. Open the second instance of Eclipse.
2. Create a new project with *File–>New–>Project–>General–>Project–>Next.*
3. Under *"Project name"* insert an name for the new project. Also a *Location* can be chosen.
4. Click on *Finish.*
5. Right-click on the new project and choose *New–>Folder.*
6. Under *"Folder name"* insert an name for the new folder.
7. Click on *Finish.*
8. Right-click on the new folder and choose *New–>Other.*
9. To create a new aUML class model: *Example EMF Model Creation Wizards–>Ooa_classdiagram Model–>Next.*
10. Under *"File name"* insert a name for the new model. Then click on *Next.*
11. Under *"Model Object"* choose *"OOA Class Model".*
12. Click on *Finish* to create the model.

13. Existing models can be dragged into the folder by drag-and-drop out of the file system or over *Import*.

3. Editing a aUML Class Model

1. Open the model in the second instance of Eclipse with a double-click or *right-click–>Open*.
2. Expand the opened tree-editor till the *OOA Class Model* node.
3. With *Right-click–>New Child–>...* on a node, new model elements can be added to the model. Only metamodel-conform options out of the elements are displayed.
4. To edit a element click on the node. At the bottom of the IDE, the *Properties* tab should open. There all attributes of an element can be edited.
5. It is important that at least the attribute *ID* of the *OOA Class Model* is set.
6. With a *right-click–>Validate* on the tree, the model can be validated against the metamodel.
7. Save the model click on *Save* in the top of the IDE.
8. To export models, simply drag-and-drop them into the file system or other Eclipse instances.

A.1.3 Generating Test-Sets out of an aUML Class Model

The following user guide describes how to use the existing master constraint-sets for aUML class models to generate test-sets from an (expert) aUML class model.

1. Import the Master Constraint-Sets for aUML Class Models

1. Start the Eclipse IDE (eclipse.exe) and open the Workspace with the EMF projects from the last user guide.
2. Install Epsilon from the Eclipse Marketplace: *Help–>Eclipse Marketplace* and Search for "Epsilon" in the search bar. Click on *Install* and continue through the dialog.
3. Open the inloom.emf.ooa.classdiagram project and click on model–>right-click on ooa_classdiagram.ecore–>Register EPackages.
4. Create a new project with *File–>New–>Project–>General–>Project–>Next*.
5. Under *"Project name"* insert an name for the new project. Also a *Location* can be chosen.
6. Click on *Finish*.
7. Right-click on the new project and choose *New–>Folder*.
8. Under *"Folder name"* insert an name for the new folder.
9. Click on *Finish*.
10. Too import the Master constraint-sets right-click on the new folder.
11. *Import...–>General–>File System–>Next*.
12. Under *"From directory"* click on *Browse* and choose the *egl folder* (it can be found in the digital attachments under *Implementation/epsilon/inloom.epsilon.rules/*)
13. Select egl in the left field. All files on the right side should now be selected.
14. Click on *Finish*
15. The imported files can be found in the *Model Explorer* under the new folder.

2. Importing the aUML Class Model

1. Right-click on the project created in the last step and choose *New–>Folder*.
2. Under *"Folder name"* insert an name for the new folder.
3. Click on *Finish*.
4. Import a (expert) aUML class model by dragging it into the folder by drag-and-drop out of the file system or the other Eclipse instance.

3. Generating the Test-Sets

1. To generate the test-sets from a aUML class model, the Epsilon EGL Engine must be run.
2. Right-click on the project in the *Model Explorer* an choose *Run As–>Run Configurations*.
3. Double-click on *EGL Generator*. A new sub-node should be created. Open it by clicking on it.
4. Under the tab *"Template"*: Click on *Browse Workspace* under *"Source"* and search for *"main.egx"*. Select the *main.egx* file and click on *OK*.
5. Under the tab *"Models"*: Click on *Add–>EMF Model*.
6. Insert *"expert"* in the field *Name*.
7. Under *"Model file:"* click on *Browse Workspace*.
8. Search for the model file name of your expert model. Click on the found file and on *OK*.
9. Under *"Metamodels:"* a refernce to the *ooa_classdiagram* should appear. If not, click on *Add file* and add the *ooa_classdiagram.ecore* file.
10. Click on *OK*.
11. Click on *Run*.
12. In the project next to the folder with the egl files, a *"evl" folder* should be created. It include the Test-Sets created from the chosen model.
13. After one successful run, the run can be started again by simply selecting the sub-node and click on *Run*.

A.1.4 Testing an aUML Class Model against the Test-Sets

The following user guide describes how to test a (student) aUML class model against existing test-sets of an expert solution.

1. Import the (Student) aUML Class Model

1. Start the Eclipse IDE (eclipse.exe) and open the Workspace from the last two user guides.
2. Create a new folder in the same project in which the evl and egl files can be found. Another option is the use of the folder of the expert model file.
3. Import a (student) aUML class model by dragging it into the folder by Drag-and-drop out of the file system or the other Eclipse instance.

2. Testing the (Student) aUML Class Model

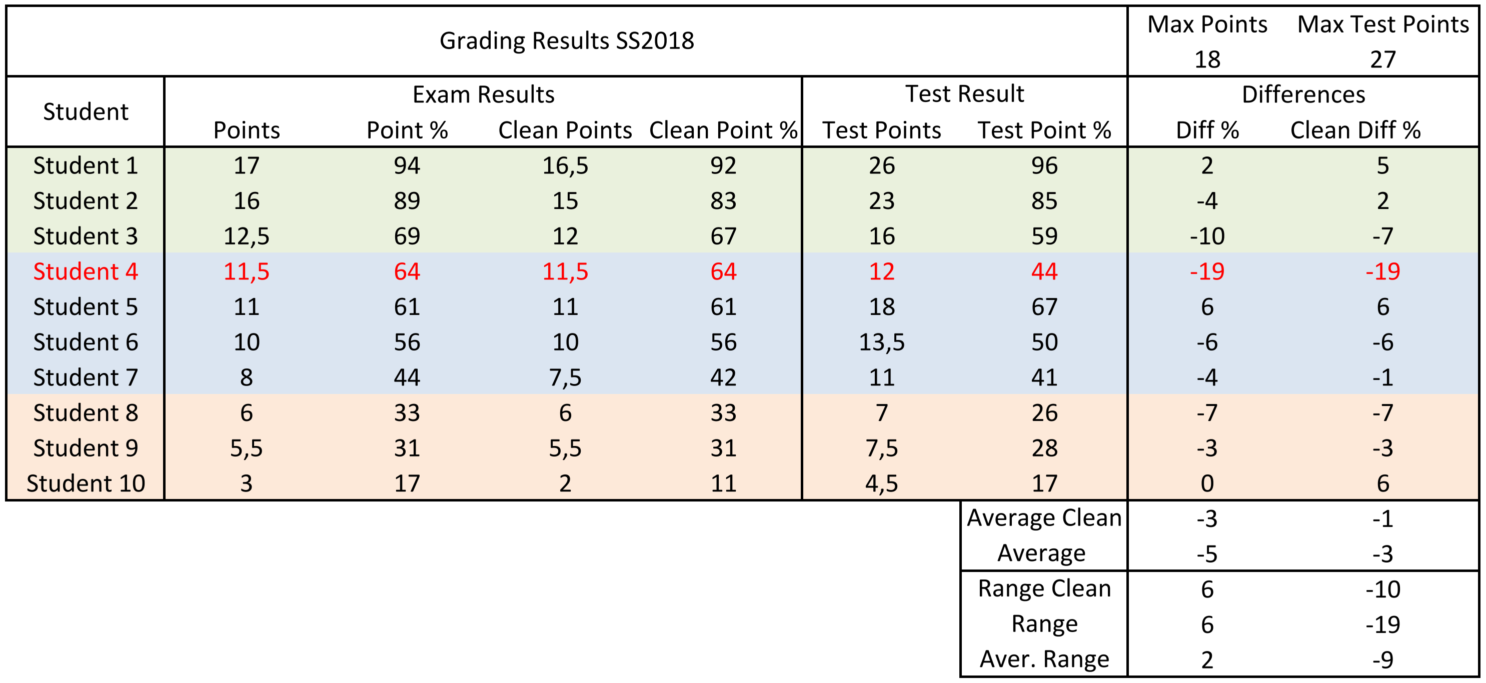
1. To test the (student) aUML class model, the Epsilon EVL Engine must be run.
2. Right-click on the project in the *Model Explorer* an choose *Run As–>Run Configurations.*
3. Double-click on *EVL Validation*. A new sub-node should be created. Open it by clicking on it.
4. Under the tab *"Source"*: Click on *Browse Workspace* under *"Source"* and search for *"rule_main*.evl"*. Select the *rule_main_** EVL file of the test-set you want to use and click on *OK*.
5. Under the tab *"Models"*: Click on *Add–>EMF Model*.
6. Insert *"student"* in the field *Name*.
7. Under *"Model file:"* click on *Browse Workspace*.
8. Search for the model file name of your student model. Click on the found file and on *OK*.
9. Under *"Metamodels:"* a refernce to the *ooa_classdiagram* should appear. If not, click on *Add file* and add the *ooa_classdiagram.ecore* file.
10. Click on *OK*.
11. Under the tab *"Parameter"* click on *Add* and insert the following values into the created parameter.
12. Name: *"fileOutput"*
13. Type: *Boolean*
14. Value: *true*

15. Under the tab *"Parameter"* click on *Add* a second time and insert the following values into the created parameter.
16. Name: *"outputDir"*
17. Type: *String*
18. Value: *Absolute path* of the created output XML file without the filename and with "\" at the end. All folders must be existing (Example: *D:\data\model\*). This parameter is optional if the parameter *"fileOutput"* is *false*.
19. Click on *Run*.
20. The output XML file should be created at the registered path. In the bottom of the IDE the *"Console"* tab holds a copy of the output in XML format. Under the *"Validation"* tab the intern messages are displayed.
21. After one successful run, the run can be started again by simply selecting the sub-node and click on *Run*.

A.2 Evaluation Results on the Comparison between Human and Assessment System

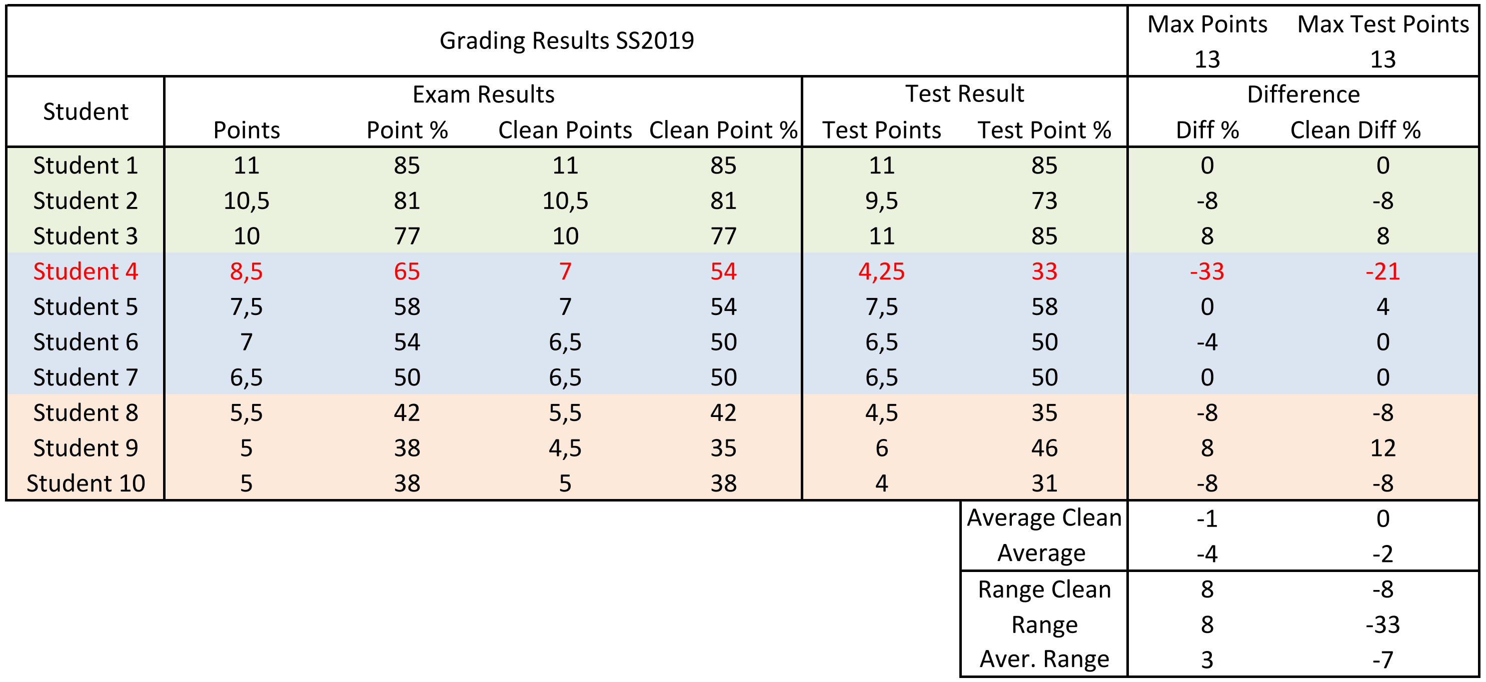
Grading Results SS2017							Max Points 25	Max Test Points 29
Student	Exam Results				Test Result		Differences	
	Points	Point %	Clean Points	Clean Point %	Test Points	Test Point %	Diff %	Clean Diff %
Student 1	24	96	24	96	24,75	85	-11	-11
Student 2	21	84	20	80	21,5	74	-10	-6
Student 3	19,5	78	19,5	78	20,5	71	-7	-7
Student 4	19	76	18,5	74	19,5	67	-9	-7
Student 5	17	68	16	64	17,75	61	-7	-3
Student 6	17	68	17	68	19,75	68	0	0
Student 7	15,5	62	15,5	62	18	62	0	0
Student 8	13	52	12,5	50	18,5	64	12	14
Student 9	13,5	54	13	52	17,75	61	7	9
Student 10	11,5	46	11	44	13,5	47	1	3
						Average Clean	-2	-1
						Average	-2	-1
						Range Clean	12	-11
						Range	12	-11
						Aver. Range	4	-9

Figure 16: Evaluation of the comparison between the scores of the automatic assessment system (*Test Result*) and a human instructor (*Exam Result*). The compared models are student solutions for the summer term 2017 exam. *Differences* shows the difference between the human instructor and the assessment system in percentage points. *Clean* values are scores without *"grace points"*. *Average* shows the arithmetic average of all entries (normal and clean are separated). *Range* shows the maximal and minimal normal Difference values. *Average Clean* and *Range Clean* are the Average and the Range without extreme values (red entries).



Grading Results SS2018							Max Points 18	Max Test Points 27
Student	Exam Results				Test Result		Differences	
	Points	Point %	Clean Points	Clean Point %	Test Points	Test Point %	Diff %	Clean Diff %
Student 1	17	94	16,5	92	26	96	2	5
Student 2	16	89	15	83	23	85	-4	2
Student 3	12,5	69	12	67	16	59	-10	-7
Student 4	11,5	64	11,5	64	12	44	-19	-19
Student 5	11	61	11	61	18	67	6	6
Student 6	10	56	10	56	13,5	50	-6	-6
Student 7	8	44	7,5	42	11	41	-4	-1
Student 8	6	33	6	33	7	26	-7	-7
Student 9	5,5	31	5,5	31	7,5	28	-3	-3
Student 10	3	17	2	11	4,5	17	0	6
						Average Clean	-3	-1
						Average	-5	-3
						Range Clean	6	-10
						Range	6	-19
						Aver. Range	2	-9

Figure 17: Evaluation of the comparison between the scores of the automatic assessment system (*Test Result*) and a human instructor (*Exam Result*). The compared models are student solutions for the summer term 2018 exam. *Differences* shows the difference between the human instructor and the assessment system in percentage points. *Clean* values are scores without *"grace points"*. *Average* shows the arithmetic average of all entries (normal and clean are separated). *Range* shows the maximal and minimal normal Difference values. *Average Clean* and *Range Clean* are the Average and the Range without extreme values (red entries).



Grading Results SS2019							Max Points 13	Max Test Points 13
Student	Exam Results				Test Result		Difference	
	Points	Point %	Clean Points	Clean Point %	Test Points	Test Point %	Diff %	Clean Diff %
Student 1	11	85	11	85	11	85	0	0
Student 2	10,5	81	10,5	81	9,5	73	-8	-8
Student 3	10	77	10	77	11	85	8	8
Student 4	8,5	65	7	54	4,25	33	-33	-21
Student 5	7,5	58	7	54	7,5	58	0	4
Student 6	7	54	6,5	50	6,5	50	-4	0
Student 7	6,5	50	6,5	50	6,5	50	0	0
Student 8	5,5	42	5,5	42	4,5	35	-8	-8
Student 9	5	38	4,5	35	6	46	8	12
Student 10	5	38	5	38	4	31	-8	-8
						Average Clean	-1	0
						Average	-4	-2
						Range Clean	8	-8
						Range	8	-33
						Aver. Range	3	-7

Figure 18: Evaluation of the comparison between the scores of the automatic assessment system (*Test Result*) and a human instructor (*Exam Result*). The compared models are student solutions for the summer term 2019 exam. *Differences* shows the difference between the human instructor and the assessment system in percentage points. *Clean* values are scores without *"grace points"*. *Average* shows the arithmetic average of all entries (normal and clean are separated). *Range* shows the maximal and minimal normal Difference values. *Average Clean* and *Range Clean* are the Average and the Range without extreme values (red entries).

Bibliography

[1] N. H. Ali, Z. Shukur, and S. Idris. Assessment System For UML Class Diagram Using Notations Extraction. International Journal of Computer Science and Network Security, 7 (8):181–187, 2007.

[2] N. H. Ali, Z. Shukur, and K. Terengganu. A Design of an Assessment System for UML Class Diagram. In 2007 International Conference on Computational Science and its Applications (ICCSA 2007), pages 539–546, Kuala Lampur, Malaysia, 2007. IEEE.

[3] N. Baghaei, A. Mitrovic, and W. Irwin. Supporting collaborative learning and problem-solving in a constraint-based CSCL environment for UML class diagrams. International Journal of Computer-Supported Collaborative Learning, 2(2):159–190, 2007.

[4] P.-D. Beck, T. Mahlmeister, M. Ifland, and F. Puppe. COCLAC - Feedback Generation for Combined UML Class and Activity Diagram Modeling Tasks. In 2. Workshop "Automatische Bewertung von Programmieraufgaben" (ABP'2015),. CEUR-WS, 2015.

[5] J. P. Bernius and B. Bruegge. Toward the Automatic Assessment of Text Exercises. In ISEE 2019: 2nd Workshop on Innovative Software Engineering Education @, pages 19–22, Suttgart, Germany, 2019.

[6] 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), pages 700–709, Munich, Germany, 2019. IEEE.

[7] 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, pages 138–141, Hyderabad, Andhra Pradesh, India, 2009. IEEE.

[8] R. W. Hasker. UMLGRADER : An Automated Class Diagram Grader. J. Comput. Sci. Coll., 27(1):47–54, 2011.

[9] C. Higgins, P. Symeonidis, and A. Tsintsifas. The Marking System for CourseMaster. In Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '02, pages 46–50, New York, NY, USA, 2002. Association for Computing Machinery.

[10] C. A. Higgins and B. Bligh. Formative Computer Based Assessment in Diagram Based Domains. SIGCSE Bull., 38(3):98–102, 2006.

[11] G. Hoggarth and M. Lockyer. An Automated Student Diagram Assessment System. SIGCSE Bull., 30(3):122–124, 1998.

[12] A. Jayal and M. Shepperd. The Problem of Labels in E-Assessment of Diagrams. J. Educ. Resour. Comput., 8(4):Article 12, 2009.

[13] D. Kolovos, L. Rose, R. Paige, and A. Garcia-Dominguez. The Epsilon Book. Eclipse, 07.2018 edition, 2010.

[14] S. Krusche and A. Seitz. ArTEMiS - An Automatic Assessment Management System for Interactive Learning. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education, pages 284–289, Baltimore, Maryland, USA, 2018. Association for Computing Machinery.

[15] S. Krusche et al. Artemis: Interactive Learning with Individual Feedback, 2016. URL `https://github.com/ls1intum/Artemis`. Last visited 2020-01-20.

[16] N.-T. Le. A Constraint-based Assessment Approach for Free-Form Design of Class Diagrams using UML. In Proceedings of Workshop on Intelligent Tutoring Systems for Ill-Defined Domains, 8th International Conference on ITS, pages 11–19, Jhongli, Taiwan, 2006.

[17] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, volume 10, pages 707–710, 1966.

[18] M. T. Llano and R. Pooley. UML Specification and Correction of Object-Oriented Anti-patterns. In 2009 Fourth International Conference on Software Engineering Advances, pages 39–44, Porto, Portugal, 2009. IEEE.

[19] S. Martínez, M. Wimmer, and J. Cabot. Efficient plagiarism detection for software modeling assignments. Computer Science Education, pages 1–29, 2020.

[20] M. Morgenstern and B. Demuth. Continuous Publishing of Online Programming Assignments with INLOOP. In ISEE 2018: 1st Workshop on Innovative Software Engineering Education, pages 32–33, Ulm, Germeny, 2018. CEUR-WS.

[21] M. Morgenstern, D. Muhs, and P. Matthes. INLOOP: Interactive learning center for object-oriented programming., 2015. URL `https://github.com/st-tu-dresden/inloop`. Last visited 2020-01-31.

[22] F. Prados, J. Soler, I. Boada, and J. Poch. An Automatic Correction Tool That Can Learn. In 2011 Frontiers in Education Conference (FIE), pages F1D–1–F1D–5, Rapid City, SD, USA, 2011. IEEE.

[23] J. Schramm, S. Strickroth, N.-t. Le, and N. Pinkwart. Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System. In Twenty-Fifth International Florida Artificial Intelligence Research Society Conference, pages 472–477. AAAI Publications, 2012.

[24] 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, pages 136–142, Macau, China, 2013. IEEE.

[25] J. Soler, I. Boada, F. Prados, and J. Poch. A web-based e-learning tool for UML class diagrams. In IEEE EDUCON 2010 Conference, pages 973–979, Madrid, Spain, 2010. IEEE.

[26] R. Sousa and P. Leal. A Structural Approach to Assess Graph-Based Exercises. In Languages, Applications and Technologies, pages 182–193. Springer International Publishing, 2015.

[27] M. Striewe and M. Goedicke. Automated Checks on UML Diagrams. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ITiCSE '11, pages 38–42, Darmstadt, Germany, 2011. Association for Computing Machinery.

[28] M. Striewe and M. Goedicke. Automated Assessment of UML Activity Diagrams. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, pages 336–336, Uppsala, Sweden, 2014. Association for Computing Machinery.

[29] P. Thomas, W. Hall, and P. Thomas. Marking Diagrams Automatically. Technical Report January, Department of Computing Faculty of Mathematics and Computing The Open University, Milton Keynes, United Kingdom, 2004.

[30] P. Thomas, S. Neil, and K. Waugh. Learning and automatically assessing graph-based diagrams. In Beyond control: learning technology for the social network generation. Research Proceedings of the 14th Association for Learning Technology Conference (ALT-C 2007), number September, pages 61–74, Nottingham University, England, UK, 2007.

[31] P. Thomas, N. Smith, and K. Waugh. Automatic Assessment of Sequence Diagrams. In 12thInternational CAA Conference: Research into e-Assessmen, Loughborough University, UK., 2008.

[32] C. Tselonis, J. Sargeant, and M. M. Wood. Diagram matching for human-computer collaborative assessment. In Proceedings of the 9th CAA Conference, Loughboroug, 2005.

[33] V. Vachharajani and J. Pareek. Framework To Approximate Label Matching For Automatic Assessment Of Use-Case Diagram. International Journal of Distance Education Technologies (IJDET), 17(3):75–95, 2019.

[34] V. Vachharajani, J. Pareek, and D. Ph. A Proposed Architecture for Automated Assessment of Use Case Diagrams. International Journal of Computer Applications, 108(4):35–40, 2014.

List of Figures

List of Tables

Listings

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Markus Hamann
Dresden, 25.February.2020