Automated Grading of UML Class Diagrams

A Thesis Submitted to the Committee on Graduate Studies
in Partial Fulfillment of the Requirements for the Degree of Master of Science
in the Faculty of Arts and Science

TRENT UNIVERSITY

Peterborough, Ontario, Canada

Applied Modelling and Quantitative Methods

M.Sc. Graduate Program

September 2020

Abstract

Automated Grading of UML Class Diagrams

Weiyi Bian

Learning how to model the structural properties of a problem domain or an object-oriented design in form of a class diagram is an essential learning task in many software engineering courses. Since grading UML assignments is a cumbersome and time-consuming task, there is a need for an automated grading approach that can assist the instructors by speeding up the grading process, as well as ensuring consistency and fairness for large classrooms. This thesis presents an approach for automated grading of UML class diagrams. A metamodel is proposed to establish mappings between the instructor solution and all the solutions for a class, which allows the instructor to easily adjust the grading scheme. The approach uses a grading algorithm that uses syntactic, semantic and structural matching to match a student's solutions with the instructor's solution. The efficiency of this automated grading approach has been empirically evaluated when applied in two real world settings: a beginner undergraduate class of 103 students required to create a object-oriented design model, and an advanced undergraduate class of 89 students elaborating a domain model. The experiment result shows that the grading approach should be configurable so that the grading approach

can adapt the grading strategy and strictness to the level of the students and the grading styles of the different instructors. Also it is important to considering multiple solution variants in the grading process. The grading algorithm and tool are proposed and validated experimentally.

Keywords: automated grading, class diagrams, model comparison

Co-Authorship and Related Publication

This thesis is based on two conference papers co-authored with my supervisor Professor Omar Alam and Professor Jörg Kienzle. Here is the list of papers included in this thesis (in both two papers, I was the primary author):

- Automated Grading of Class Diagrams. **Bian, W.**, Alam, O., Kienzle, J. (2019, September). In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (pp. 700-709). IEEE. (Parts of Chapter 3, Chapter 4 and Chapter 5 are based on this paper). *In this paper, I proposed the approach for automated grading of UML class diagrams, including the algorithm and the metamodel. Prof. Alam and Prof. Kienzle participated in the discussions and manuscript preparations. I presented the paper in the conference.*

- Is Automated Grading of Models Effective? Assessing Automated Grading of Class Diagrams. **Bian, W.**, Alam, O., Kienzle, J. (Parts of Chapter 6 and Chapter 8 are based on this paper). *In this paper, I worked with Prof. Alam and Prof. Kienzle on empirically evaluating the efficacy of the automated grading approach. I worked on the*

case study and result analysis. The co-authors participated in the discussions and manuscript preparations. This paper has been submit to 2020 ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS) and is under review.

Acknowledgements

I wish to thank various people for their contributions to this thesis. Without you all, it would not be possible for me to finish this thesis.

First of all, my parents, who gave me a precious chance to study in Canada and have always respected my choice and supported me without hesitation. Thank you for everything.

The TouchRam group in McGill University - the best team I have been lucky enough to join. I could not conduct the research smoothly without your great effort on TouchCore tool. Meng and Daniel - working with you was fun and thanks for helping for my research and daily life.

I would like to thank my supervisor Dr.Omar Alam and Dr.Jörg Kienzle from McGill University. Thank you for providing me such valuable opportunity to work alongside both of you. Your suggestions and encouragement supported me throughout the thesis study. Without your assistance, I could not finish two conference papers in my master study. Also, thank you to my supervisor committee for your time and advice.

Finally, thank you to Trent University for financial support. The two years of studying at Trent University will be the most precious time in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software engineering education is high in demand driven by the fast-changing job market. This created a supply-demand imbalance between computing college graduates and the available technology jobs. Motivated by projections of US employment [1], computing schools experience an increase in enrolment as students rush into computer science programs in record numbers [2]. The increasing number of computing students increases the workload on instructors as they have to grade large number of assignments. Besides the increased workload, instructors struggle to grade assignments and exams fairly, which is not an easy task. It is difficult for human graders to precisely follow the grading formulae when grading each individual assignment, especially when grading subjective topics. As a result, automated grading is increasingly popular in Computer Science. Many approaches to automati-

cally assess programming assignments have been proposed and are currently being used in the classroom [3; 4]. As the class sizes also increase in advanced undergraduate software engineering classes, we now face the need to determine effective algorithms for automating the grading of models.

In addition, automated grading is crucial for e-learning and Massive Online Open Courses (MOOCs) [5; 6]. Large numbers of students, sometimes thousands, enrol in these online courses. Therefore, it becomes difficult to manually grade their assignments and exams. Furthermore, in MOOCs, students need tools to self-assess their knowledge to decide whether they want to move to the next topic in the course. Such self-assessment methods have been implemented in popular online learning platforms, e.g. Khan Academy [7], Udemy [8], and Coursera [9]. Automated assessment can also be used to calibrate a learner's prior knowledge [10], i.e., to assess their prior knowledge on the subject when starting a new course. In addition, in the context of online courses, automatic grading can be used to provide timely feedback to the learner [11]. Finally, online learning has become a subject of heightened focus during the COVID-19 crisis [12–18]. Schools and universities quickly moved their courses online across the world. Different strategies have been proposed and discussed to support this transition to online delivery, including automatic grading and personalized intelligent learning [19; 20].

Grading of models is difficult. This is partially due to the fact that mod-

elling problems are often *ill-defined* problems, where multiple solutions may exist for a particular problem [21; 22]. Unlike well-defined problems, where a solution can be either correct or incorrect, a diagram design problem involving class diagrams can have a large solution space. For example, solutions can vary based on the class names, i.e., a student's solution can use a synonym for a class name instead of the exact name used in the teacher's solution. Solutions also can vary based on the structure, e.g., adding attributes to the subclasses instead to the superclass. These variations create an additional overhead on the instructors when grading assignments, as they have to spend longer time to evaluate a student's answer. Furthermore, instructors often revise their marking scheme after grading several student papers. For example, an instructor may want redistribute the grades when the instructor discovers that students had trouble with a particular part of the model, which is an indication that the problem description was maybe not clear. In such cases, the instructor might want to adjust the grading weights for parts of the model to compensate. Unfortunately this means that the instructor has to manually update the grades for the students that already graded by revisiting the students solutions again using the new marking scheme. Finally, after receiving their grades, many students may request that their copies be reevaluated, often because the instructor may not have been consistent when grading, for example, a large class over a longer period of time. Nevertheless,

several approaches for automating the grading of models have been defined in recent years, most of them targeting and tailored to specific modelling notations, e.g. [23–25]. However, despite these efforts, little work has been done to automated grade the UML class diagram and evaluate the grading result.

1.1 Problem Summary

This thesis attempts to solve the problem raised in the introduction. The problem this thesis tries to solve can be summarized as follows: "Is automated grading effective in helping the instructors grade UML class diagrams?"

In order to solve this problem, this thesis proposes an automated grading approach for UML class diagrams. This thesis presents an algorithm that establish mappings between model elements in the instructor's solutions to elements in the student solutions, exploiting syntactic, semantic and structural matching strategies. The student gets full mark for each element that is perfectly matched. Partial marks are given to solutions that are partially correct, e.g. an attribute that is placed in a wrong class. Two metamodels have been proposed to support the automted grading approach in this thesis. One metamodel is used to store grades for each model element, e.g. classes, attributes, and associations. The mappings between the student's model

elements and the instructor's model elements are stored by using another metamodel, which makes it possible to update the grading scheme later on. The grading algorithm and metamodels are implemented in the TouchCORE tool [26], which visually shows the grades for the class diagram and prints out feedback to the student.

To evaluate the efficiency of the proposed automated grading approach, we conduct a case study of using automated grading for two assignments, one was given to 103 students of an introductory course in software modeling, and another was given to 89 students of an advanced course in software engineering.

1.2 Thesis Contributions

The contributions of thesis are organized into there parts:

Part I Algorithm (Chapter 3, Chapter 4):

- The thesis introduces and defines several matching strategies when grading the UML class diagram. The process is illustrated by presenting motivated model examples.

- The thesis demonstrates the algorithm for automated grading of UML class diagrams.

Part II Tool (Chapter 5):

The thesis introduces two metamodels to support the automated grading algorithm. Moreover, the thesis illustrates the grading tool that implements the automated grading algorithm and the metamodels.

Part III Validation (Chapter 6, chapter 7 and Chapter 8):

The thesis applies the grading approach to a case study of two modeling assignments. By comparing the results of automated grading with manual grading conducted by the instructors, the following research questions can be answered:

- **RQ_1 Are there different grading criteria for class diagrams?** It is true that while the instructors use a common base strategy for grading, they deal differently with missing model elements, unnecessary model elements and modelling alternatives.

- **RQ_2 Does the use of configuration settings improve the accuracy of automated grading?** A configurable grading algorithm can produce grades that are closer to the instructor's manual grading scores. Overall, in one case study, the difference is reduced from 6.53% to 4.8%, while in another case study, the difference is reduced from 24.6% to 13%.

- **RQ_3 Does the accuracy of automated grading improve when**

multiple solutions are matched against? A class diagram modeling problem can have more than one correct solution. It is possible to support multiple correct solutions in automated grading. In one case study, when this feature is incorporated, 40 out of 89 students that have used an alternative solution in their models received higher grades, reducing the gap between the automated and manually determined grades.

- **RQ_4 Does automated grading save time?** Automated grading has the potential to save a significant amount of time. While with manual grading it takes hours to grade a class of 80-100 students, a properly configured algorithm that is given all relevant modelling solutions can accomplish the grading in a matter of seconds.

- **RQ_5 Does automated grading help to ensure fairness?** Manual grading is prone to inconsistencies. Automated grading is able to point out the inconsistencies in manual grading.

1.3 Thesis Organization

This thesis is divided into nine chapters. We discuss the related work about automated grading in Chapter 2. Then, we provide examples that mo-

tivate our matching strategies in Chapter 3. Based on the matching strategies, Chapter 4 details the algorithms that compare the student's model with the instructor's model. Chapter 5 demonstrates the metamodels and grading tool that support the automated grading. Chapter 6 outlines the case study setup. We present some first results and discuss the limitations threats to validity. In Chapter 7, we elaborate the configuration options for automated grading and discuss our configuration panel. Chapter 8 discusses how we deal with the multiple correct solutions and analyses the effectiveness of automated grading. Chapter 9 concludes this thesis and discuss directions for future work.

Chapter 2

Background and Related Work

2.1 Automated Grading Tools

Automated grading has been researched for many years, a lot of automated grading tools have been developed, e.g. to grade mathematical questions [27; 28], automated short answer grading (ASAG) [29–31], and grading simple essays [32; 33]. Particularly in the field of computer science, a number of automated grading tools and systems have been designed to evaluate programming assignments, e.g. AutoGrader for Java [34], CourseMarker for Java and C++ [35], and Marmoset for Java, C, and Ruby assignments [36].

Empirical assessments have been conducted to understand the efficacy of automated grading programming assignments [35–37]. For example, Gao et. al. [38] created a framework for grading C++ programming assignment

by generating testing cases automatically. They used it on five different assignments for 173 students in an introductory level programming course. The Online Judge tool [39], which supports grading C/C++ and Java assignments, runs the student's program with strict memory and time limits and compares the output with predefined answers to grade the program. This tool was used to evaluate 2 first year programming assignments, on 712 and 795 students, and on an introductory high school course of 40 students. Spacco et. al surveyed 70 students on efficacy of their Marmoset tool [36]. CourseMarker [35] grades the students by checking typographical layout and running test cases. A survey found that by using the CourseMarker, 94% of students found that the tool helped improve their programming skills.

2.2 Automated Grading Tools for Models

Several automated grading approaches have been proposed for grading specific models, such as deterministic finite automatons (DFA) [23] and Entity-Relationship (ER) diagrams [24; 40; 41]. For UML models, such as Use Case Specifications, there are approaches that proposed automated assessment of UML models. Jayal and Shepperd [42] proposed a method for label matching for UML diagrams and using different levels of decomposition and syntactical matching. They evaluate their approach using a case

study on matching activity diagrams. Striewe et al. [43] present an approach to grade the activity diagram based on a reference about trace information. Tselonis et. al. [44] introduced a diagram marking technique based on graph matching. Thomas et. al. [45] introduced a framework that used synonyms and an edit distance algorithm to mark graph-based models. Vachharajani et al. [46] introduced a framework for automatic assessment of Use Case Diagrams using syntactic and semantic matching. Sousa and Paulo [47] introduced a structural approach for graphs that establishes mappings from a teacher's solution to elements in the student solution that maximizes a student's grade.

2.3 Automated Grading Tools for UML Class Diagram

A number of approaches have been proposed that compare UML class diagrams. Haggarth and Lockyer [25] proposed a framework that compares the symbols within a student's model with the instructor model and gives feedback. Ali et. al. [48] proposed an assessment tool to compare a class diagram drawn by a student with the instructor's diagram. Their approaches only can return feedback rather than numerical grade. Soler et. al. [49]

developed a web-based tool for correcting UML class diagrams. They conducted a comparative experiment on 24 students and found that students who used the tool did better in solving modeling problems. However, their approach relies only on string matching. Our matching algorithm uses syntax, semantic and structural matching to compare models. Hasker [50] applied the UMLGrader, which only provided a binary pass-fail grade, to a course which had 37 students with 38% sophomores, 38% juniors, and 24% seniors. Chaudron et al. [51] applied machine learning to automated grading the class diagrams. They extracted features from the UML models and implemented classification and regression experiments with machine learning algorithms, e.g., random forest algorithm. In a case study of 99 student pairs, three experts conducted manual grading on a 10-point scale. Then the grades were converted to a 5-point and 3-point scale. The results showed that the classification and regression models were not reliable to grade students class diagrams on a 10-point scale, since the highest accuracy was 42.76%. However, they received higher accuracy when classifying in a 3-points (fail, pass, good) or 2-points (fail, pass) scale, i.e., their approach can give a rough assessment of a student's performance. But our approach can return a numerical grade with detailed feedback as human instructor does in the grading process.

2.4 Conclusion

A number of researches have been conducted to automatically grading assignments in many fields, especially grading programming assignments. However, grading UML class diagram has received little attention. There are three main differences between our approach and the existing approaches discussed above: (1) our approach combines syntactic, semantic and structural matching for grading class diagrams. In addition to using Levenshtein distance for syntactical matching, our approach uses three algorithms for semantic matching and performs structural matching between two diagrams. Based on the matching results, the approach assigns marks to the model elements. Most of the above approaches are limited to syntactic matching of names. (2) Our approach proposes a non-invasive grading metamodel that stores the determined grades alongside the model as feedback to the students. As a result, our grading algorithm can return a determined numerical grade with feedback to student, while most of the above approaches can only return feedback or a rough grade (fail, pass). (3) Our approach proposes a new classroom metamodel that allows for saving and automatic updating the grades of a group of students in case the teacher changes the grading scheme.

Also the aforementioned work did not conduct a detailed assessment of automated grading of class diagrams. Motivated by the luck of such assess-

ment, we conduct a study on 192 students of two different undergraduate grades to study the efficacy of automated grading. In addition, we study whether automated grading can support different grading styles, and whether automated grading is more fair and consistent than manual grading. To conduct our experiment, we added a feature (called *configuration panel*) to the tool to allows the grading criteria be customized for a particular instructor style. This feature is discussed in more detail in RQ1. To the best of our knowledge, customizing the grading criteria for class diagrams has never been explored before.

Chapter 3

Motivating Examples

In this chapter, we motivate our approach using a simple class diagram modeling a university. The approach motivated here will be implemented in our automated grading algorithm, which will be detailed discussed in the chapter 4.

3.1 University Models

The model description as follows:

> *Each person in a university should have name and age. For each person, the system should be able to print the basic information and get his/her age. Teachers and students are persons. A teacher has an attribute for department and is assigned to courses. A student has year and can select*

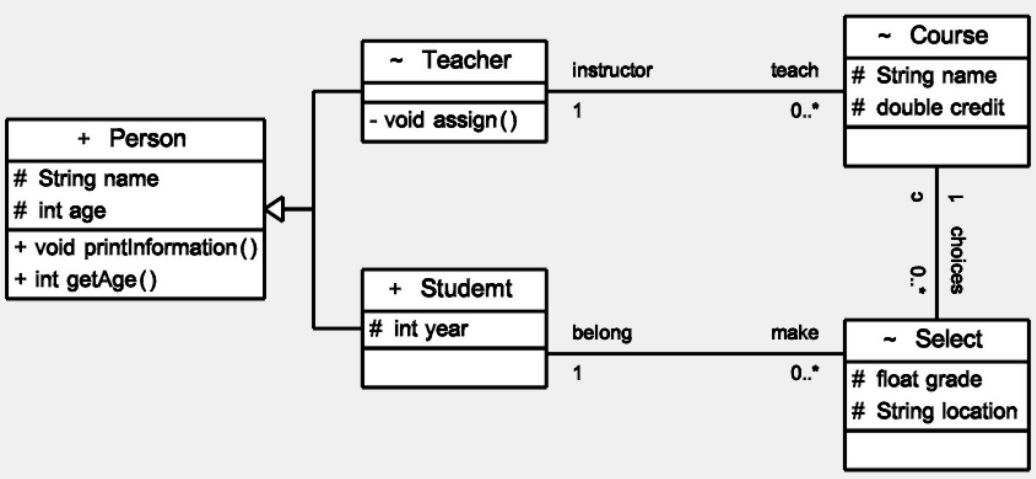

Figure 3.2: Student Solution Model 1

the course. Each course has a name, a credit and a class location. A teacher can teach multiple courses. Each course selection will contain the final grade for the course.

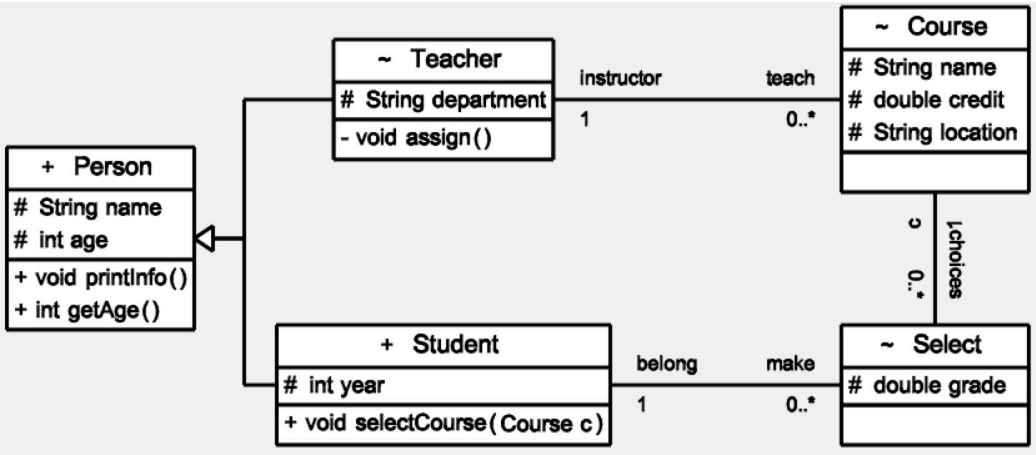

Figure 3.1: Instructor Solution for University Model

Fig. 3.1 shows the instructor solution. The first student solution, shown

in Fig. 3.2 uses as a name for the *Teacher* class the word *Instructor*, uses the wrong spelling form for *Studemt*, and uses *Select* instead of *Selection*. Although the student uses different names for classes, we want our matching algorithm to determine that *Instructor* is a synonym for *Teacher*, which we call a *semantic match*. The class *Student* should be matched with the class *Studemt* syntactically, even though there is a spelling difference. In a similar way, the class *Select* should be matched with *Selection* and the operation *printinfo* matched with *printinformation*. We also notice that the attribute *location* is misplaced, i.e., it is added to the class *Select*, which is wrong. Although *location* is misplaced, one can argue that the student should receive partial marks for including it to the model. Finally, two elements, the attribute *department* and the operation *selectCourse* are missing in the student solution, i.e. they can not be matched syntactically or semantically with any element.

Fig. 3.3 shows a solution by a different student. There are three important comparison checkpoints in this model: (1) Class *Subject* has the same attributes as the class *Course* in the instructor's solution in Fig. 3.1. It is reasonable to consider that these two classes should match due to their similar structure. (2) The class *Register* has associations with class *Student* and *Subject* (*Subject* is matched with *Course* using semantic match). Therefore, we can consider that class *Register* is matched with *Selection*, although their

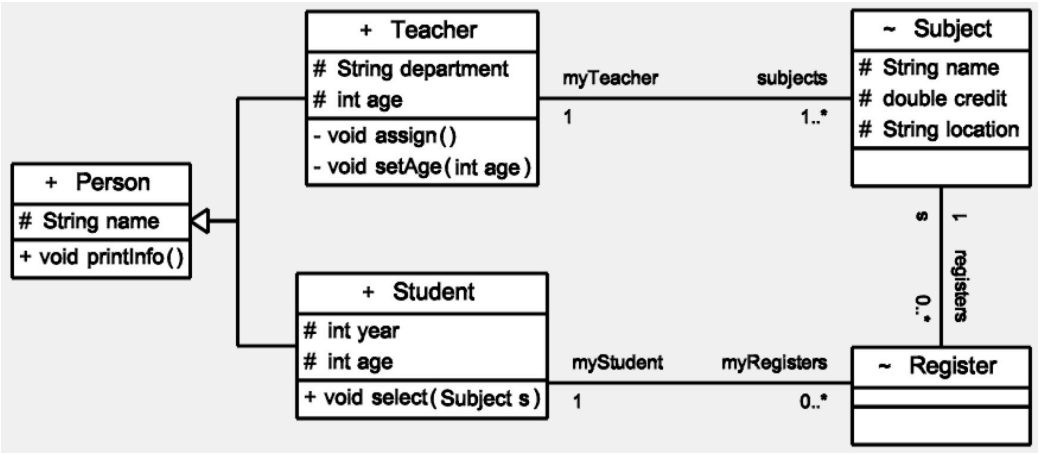

Figure 3.3: Student Solution Model 2

names do not match, neither syntactically nor semantically. Again, this is a *structural match* based on the similarity of the associations with other classes in the respective models. (3) In the instructor's model, the attribute *age* belongs to the superclass *Person*. In the solution model in Fig. 3.3, the student added two *age* attributes to the subclasses, *Teacher* and *Student*. We should give these two attributes partial marks.

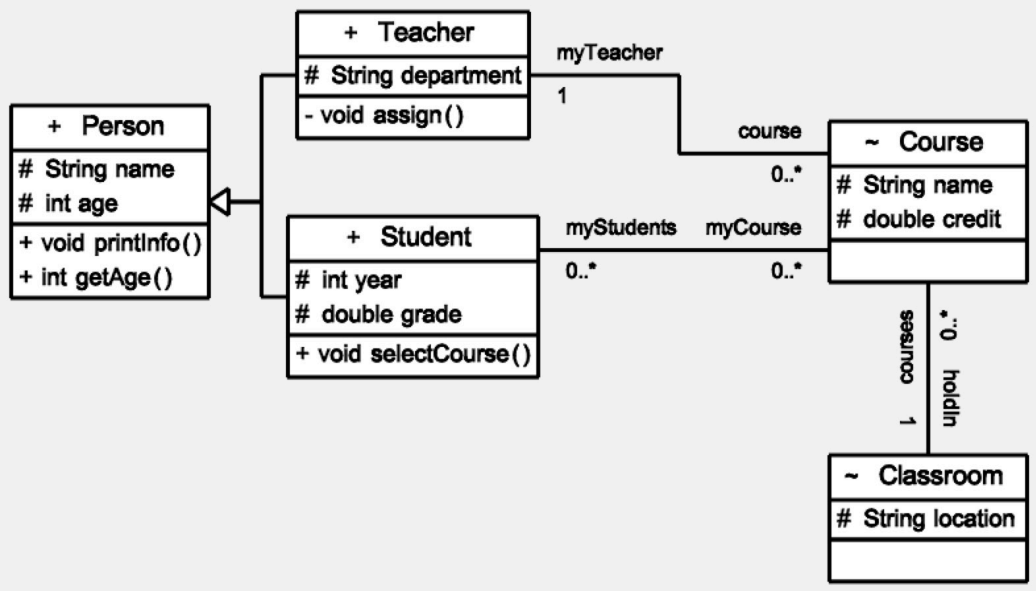

Figure 3.4: Student Solution Model 3

The third student solution shown in Fig. 3.4 illustrates two interesting cases, class splitting and class merging. (1) Class *Classroom* does not syntactically or semantically match any class. Furthermore, its content does not provide enough information to match with any class structurally. However, based on attribute matching, the attribute *location*, which belongs to the class *Course* in the instructor's model has been misplaced in the class *Classroom* by the student. Together, Class *Course* and class *Classroom* in the student's model have the same attributes of the class *Course* in the instructor's model. Also, there is a 1-to-multiple association between class *Classroom* and class *Course* in student's model, allowing a particular value

for *location* to be associated with multiple courses. We can therefore consider that the student has split the class *Course* into two classes, *Course* and *Classroom*. (2) Class *Selection* seems to be missing from the student's model because it fails to match with any element using the matching methods that we discussed before. Based on the attribute and operation matching results, we detect that all properties of class *Selection*, i.e. attribute *mark*, have been misplaced to class *Student* in the student's model. Also in the instructor's model, class *Selection* has an association with class *Student*. Therefore, we consider that the class *Student* in the student's model is a combination of class *Student* and class *Selection* in the instructor's model, and might want to give partial marks.

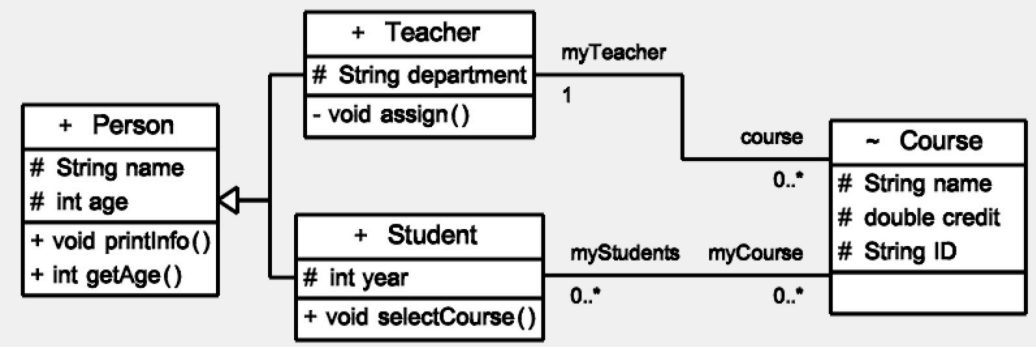

Figure 3.5: Student Solution Model 4

The fourth solution, shown in Fig. 3.5, illustrates how associations are matched. In this model, the association *Teacher-Course* in the student's model can not be matched with the association between the class *Teacher-*

Course in the instructor's model only by the name of the association ends, since the name of association end *teach* can not be matched with the name of association end *courses*, neither syntactically nor semantically. But we detect that these two associations connect two pairs of matched classes, i.e., the class *Course* in the student's model matches with the class *Course* in the instructor's model and the class *Teacher* in the student's model matches with the class *Teacher* in the instructor's model. As a result, we consider that when we match the associations, we should focus on the classes that an association connected with. Furthermore, in this model, the student forgot the class *Selection*. There is no association between the class *Student* and *Course* in the instructor's model, but the class *Selection* has two associations, with class *Student* and *Course* with multiplicities 1 on both ends. Therefore, the student's association between *Student* and *Course* can be considered a *derivative association* and should receive partial marks.

3.2 Conclusion

From examples above, we identified four matching strategies which should be taken into account by the algorithm. First, strict string matching is not sufficient for grading. It is essential to combine *syntactic matching* (eliminating spelling mistakes) and *semantic matching* (considering synonyms and

words with related meaning) for strings in the algorithm. Second, *structural matching* strategies should be incorporated, e.g. matching by comparing the contents of a class, similarity based on the associations with other classes, and considering classes that are split or merged. Third, the grading algorithm should handle class inheritance properly, i.e. handle the class elements that are misplaced within the inheritance hierarchy. Fourth, the algorithm should be able to match associations based on the connected classes, including finding potential *derivative associations*.

Chapter 4

Grading Algorithm

In this chapter, we discuss the algorithm for automated grading of class diagrams. The overall algorithm is divided into six parts: matching classes, match split classes, merged classes, attributes and operations, associations and enumerations. In the following, the six parts are explained in detail.

4.1 Matching Classes

Algorithm 1 illustrates this process in detail. The algorithm takes as input the instructor model, $InstructorModel$, and the student model, $StudentModel$.

Two different strategies are used to compare the names of the classes. To perform a *syntactic match* (line 5), the Levenshtein distance [52] is used to measure the similarity between the two names. The Levenshtein distance

Algorithm 1 Compare Classes

```
1:  procedure COMPARECLASS(InstructorModel, StudentModel)
2:      instList← InstructorModel.getClass()
3:      studList← StudentModel.getClass()
4:      for all Class C_i in instList, C_s in studList do
5:          if syntacticMatch(C_s.name, C_i.name) or
6:              semanticMatch(C_s.name, C_i.name) ) or
7:                  contentMatch(C_s.content, C_i.content) then
8:              storePossibleMatch(C_i, C_s)
9:      for all Class C_i in instList do
10:         if ∃ matched classes for C_i then
11:             find among the matches of C_i the C_best
12:                                 that obtains the highest mark among the matches
13:             classMatchMap.put(C_i, C_best)
14:     for all Class C_i in missClassList do
15:         for all Class C_s in studList do
16:             if no match exists for C_s then
17:                 ListI← C_i.getAssociationEnds()
18:                 ListS← C_s.getAssociationEnds()
19:                 if assocMatch(ListS,ListI) then
20:                     classMatchMap.put(C_i, C_s)
21:         if no match exists for C_i then
22:             missClassList.add(C_i)
        return classMatchMap, missClassList
```

calculates the minimum number of single-character edits required to change one word into another. Two classes are matched when their Levenshtein distance is smaller than 40 percentage of the longest name string length.

The second strategy involves a *semantic match* (line 6). We use a similarity metric based on SEMILAR [53], which all calculate a similarity metric between two words based on the WordNet database: Hirst and St-Onge Measure (HSO) [54], Wu and Palmer (WUP) [55] and LIN [56]. The combined use of three measures presents a better performance than using only a single

measure. If the determined score is satisfactory, then the match is stored.

- HSO: This measure compares two concepts based on the path distances between them in the WordNet database. It measures the similarity by the number of directions change which should be needed to connect one concepts to another.

- WUP: Given two concepts, WUP measures their similarity by the number of common concepts from the root concepts to these two concepts.

- LIN: Lin is an improvement of the Resnik measure [57] and uses Information Content (IC) of two concepts to calculate their semantic similarity. IC of a term is calculated by measuring its frequency in a collection of documents.

The class structural similarity match strategy (named contentMatch in line 7) includes property similarity match which compares the properties of two classes. Two properties, e.g., two attributes, with matched names would be regarded as similar. The number of properties that are needed to be edited to change one class to another is key to determine whether two classes could be regarded as structurally similar.

After we find all potential matched classes, we calculate the grades for each potential match (lines 9 - 13). If C_s (in the student solution) is a

potential match for class C_i (in the instructor's solution), we calculate the points that C_s would give the student based on the grades attributed to C_i and its content. The matched class that gets the highest grade in the student solution is then retained as the final match for C_i.

After finding possible matching classes based on their names and content, we additionally search for classes that could be matched based on their associations to other classes. Lines 14 - 20 illustrate this process. For each pair of classes that is not yet matched, we look at their association ends, and if two classes have similar association ends, we consider them matched.

4.2 Matching Class Contents

While Algorithm 1 matches the classes, there could be attributes or operations that are misplaced, i.e., are placed in the wrong class in the student model. Let A_i be one property (attribute or operation) in the instructor model and A_s one property in the student model. There are four scenarios: (1) the names of A_i and A_s match and their containing classes also match. (2) the names of A_i and A_s match while their containing classes do not match. In this case A_s is considered misplaced. Based on the grading policy, misplaced properties should score fewer points. For example, for the case study presented in the next section, we deducted 0.5 points for each

misplaced property. (3) the names of A_i and A_s match, however, A_i belongs to a super class and A_s belongs to one of the subclasses. If A_i is not private, then A_i and A_s are considered as matched. However, in this case, the student could also only get partial marks because the scope of the property is too limited. (4) Ai and As could not match with each other at all. Algorithm 2 finds the matched attributes and operations in two models. In addition to the instructor and student models, this algorithm takes as input the matched class map which was populated by Algorithm 1, $classMatchMap$. The algorithm starts by finding the matched attributes in the same classes, i.e., same matched classes (line 4-10), if it does not find a corresponding matched attribute in the same class, it will look for it in the super class (line 11-12). It is not shown in Algorithm 2, but we traverse the inheritance hierarchy all the way up. If it does not find the attribute in the super class, it will look for it in other classes in the model that are not matched with the class. If the attribute exists in an unmatched class, then it is considered to be misplaced and should be given a partial grade. Operations are matched in a similar way (line 19-32). After finding all the matches, the algorithm returns a map of matched attributes, $matchedAttrMap$, a map of misplaced attributes, $misplaceAttrMap$, a map of matched operations, $matchedOperMap$ and a map of misplaced operations, $misplaceOperMap$.

4.3 Matching Split Class

Algorithm 3 checks whether the student splits one class into two classes. Let be C_{s0} and C_{s1} be two classes in the student model. The algorithm first checks if there is a 1-to-multiple association between C_{s0} and C_{s1} (line 6). If an attribute is extracted from a class A and placed in a different class B, then, there should be 1-to-multiple association from the B to A. This allows a value for the attribute in B to be associated with multiple instances of A, as discussed previously in the example of Fig. 3.4. Then if there exists one class Ci in the instructor model that has the similar properties in both C_{s0} and C_{s1}, we consider that class Ci has been split into C_{s0} and C_{s1} by student. The algorithm returns a map of split classes, $splitClassMap$.

4.4 Matching Merged Class

Algorithm 4 checks whether two class in the instructor model could be matched with one class in the student model, which means the student merged the two classes into one class in the student's solution model. Let be C_{i1} and C_{i2} two classes in the instructor model, where all properties of C_{i1} have been misplaced into class C_s in the solution model. If C_s is already matched with C_{i2} based on the class matching algorithm and C_{i1} and C_{i2} have an association between them (line 3), we can consider that the student

used Cs to combine both C_{i1} and C_{i2}. We only give points to two classes that are merged into one class. We do not give points to more than two classes that are merged into one classes. In that case, the merged class will become quite complex and with less cohesive. After finding all the merged classes, the algorithm returns a map of merged classes, $mergeClassMap$.

4.5 Matching Association

Algorithm 5 matches the associations in two models. Contrary to other matching algorithms mentioned before, this algorithm does not focus on comparing associations based on their names, rather it compares them based on the classes that an association is connected with. Let C_0 and C_1 be two classes connected by association Ai in the instructor model, and C_2 and C_3 be two classes connected by association As in the student model. If C_0 and C_1 in the instructor model, and C_2 and C_3 in the student model could be matched as two pairs of classes, As and Ai should be also matched. Moreover, The association from the student model that contains association class preferentially matches with the association that also contains association class from the instructor model. Then, if some classes are missing in the student model, we try to find potential derived associations that could go through the missing class. For each missing class, we first find the classes that it

is connected with (line 8-9). We do this process recursively, although, it is not shown in the algorithm. This means that we also find classes that the missing class is connected with indirectly, i.e., through other classes. Then, we search if there is an association in the student model which has one end connected to the missing class (line 10-16). The algorithm returns a map of matched associations, $associationMatchMap$ and a list of derived association, $derivationList$. It is important to note that a grader may want to give grades for any derivative association, i.e., not necessarily derived from a missing class in the student solution. In that case, we have to relax the condition check for missing classes in this algorithm. In the case study discussed in the next section, the instructor opted to give grades for any derived association.

4.6 Matching Enumeration

An enumeration is a type that has a defined number of possible values. Algorithm 6 matches enumerations of two models. The straightforward way to match enumerations is to compare their names and their literal values. Let be Ei be an enumeration in instructor model and Es be an enumeration in the solution model. If the entries of Ei and Es could be matched by their names, Ei and Es would be considered as matched. It is possible that

student does not model the enumeration perfectly, in which case there will entries in Ei that are not matched with entries in Es. The algorithm returns a map of matched enumerations, $enumMatchMap$.

For the missing literals in the enumeration, it is possible that the student used other model elements, such as an attribute or a class, to represent a missing entry in the enumeration. If a literal e in Ei could not be matched with any entry in enumeration Es, we search whether there is a class or attribute in the student solution which name matches with e. Depending on the grading scheme, the instructor can opt for giving a full point or partial point when a student use an attribute or a class to represent an enum literal.

4.7 Conclusion

Comparing class diagram models should consider different kinds of variations. This chapter outlines the grading algorithm, which is divided into six comparison parts. For comparing classes, not only we compare the names of the classes, including syntactic match and semantic matches, but also structural match and match classes based on their associations with other classes. This strategy can match as many classes as possible. Next, matching attributes and operations are mainly based on the their names and containing classes. Misplaced attributes and operations should also be matched.

Moreover, based on the misplaced attributes and operations matching result, merged classes and split classes should be checked. This algorithm compares associations based on the classes that an association is connected with. Also the derived associations that are related to the missing classes should be found and given partial points. Finally, matching enumerations is based on their names and literal values. The other model elements can be used to represent a missing entry in the enumeration. In the next chapters, we will introduce the tool where we implement the grading algorithm and metamodel. Then we will apply the algorithm to two case studies to verify the effectiveness of the algorithm.

Algorithm 2 Compare attributes and operations in `InstructorModel` with `StudentModel`

1: **procedure** COMPARECONTENT(`InstructorModel`, `StudentModel`, `classMatchMap`)
2: `instList`← `InstructorModel`.getAttribute()
3: `studList`← `StudentModel`.getAttribute()
4: **for all** Attribute A_i **in** `instList`, A_s **in** `studList` **do**
5: $C_i \leftarrow A_i$.eContainer()
6: $C_s \leftarrow A_s$.eContainer()
7: **if** A_i is synatax or semtantic match for A_s **then**
8: **if** `classMatchMap`.get(C_s).equals(C_i) **then**
9: `matchedAttrMap`.put(A_s, A_i)
10: **else if** C_i is superClass of `classMatchMap`.get(C_s) **and** A_i is not private **then**
11: `matchedAttrMap`.put(A_s, A_i)
12: **for all** Attribute A_i **in** `instAttrList`, A_s **in** `studAttrList` **do**
13: **if** A_s not matched **And** A_i is synatax or semtantic match for A_s **then**
14: `misplaceAttrMap`.put(A_s, A_i)
15: `instList`← `InstructorModel`.getOperation()
16: `studList`← `StudentModel`.getOperation()
17: **for all** Operation O_i **in** `instList`, O_s **in** `studList` **do**
18: $C_i \leftarrow O_i$.eContainer()
19: $C_s \leftarrow O_s$.eContainer()
20: **if** O_i.synMatch(O_s) **or** O_i.semanticMatch(O_s) **then**
21: **if** `classMatchMap`.get(C_s) equals C_i **then**
22: `matchedOperMap`.put(O_s, O_i)
23: **else if** O_i is superClass of `classMatchMap`.get(C_s) **and** O_i is not private **then**
24: `matchedOperMap`.put(O_s, O_i)
25: **for all** Operation O_i **in** `instOperList`, O_s **in** `studOperList` **do**
26: **if** O_s is not matched **And** O_i.synlMatch(O_s) **or** O_i.semanticMatch(O_s) **then**
27: `misplaceOperMap`.put(O_s, O_i)
28: `instOperList`.put(O_i, true)
29: **return** `matchedAttrMap,misplaceAttrMap, matchedOperMap,misplaceOperMap`

Algorithm 3 Check whether a class is split into two classes

```
1: procedure CLASSSPLITMATCH(InstructorModel, StudentModel)
2:     instList← InstructorModel.getClass()
3:     studList← StudentModel.getClass()
4:     for all Class C_s0 in studList, zhuzhuC_s1 in studList do
5:         if C_s0 and C_s1 has 1-to-multiple association then
6:             for all Class C_i in instList do
7:                 if C_i has same properties with C_s0 and C_s1 then
8:                     splitClassMap.put(C_i, <C_s0,C_s1>)
9:                     break
10:    return splitClassMap
```

Algorithm 4 Check whether a class is merged into another class

```
1: procedure CLASSMERGEMATCH(InstructorModel, StudentModel)
2:     for all Class C_i1 in InstructorModel matched with C_s in StudentModel
   do
3:         for all Class C_i2 in InstructorModel which content is misplaced in Cs
   do
4:             if C_i1 has association with C_i2  then
5:                 mergeClassMap.put(C_s,<C_i1C_i2>)
6:                 break
7:             return mergeClassMap
```

Algorithm 5 Compare association in `InstructorModel` and `StudentModel`

```
1: procedure COMPAREASSOC(InstructorModel,StudentModel, missClassList)
2:     instAssocList ← InstructorModel.getAssociation()
3:     studAssocList ← StudentModel.getAssociation()
4:     for all Association A_i in instAssocList, A_s in studAssocList do
5:         if A_i and A_s connect two pairs of matched classes then
6:             associationMatchMap.put(A_s, A_i)
7:     for all class C in missClassList do
8:         for all Class C_i in InstructorModel is connected with C do
9:             possibleAssocMap.get(C).add(C_i)
10:    for all Association A_s in studAssocList do
11:        endClass1 ← A_s.getEnd1()
12:        endClass2 ← A_s.getEnd2()
13:        for all Key Class C in possibleAssocMap do
14:            possibleClassList←possibleAssocMap.get(C)
15:            if endClass1 in possibleClassList and endClass2 in possibleClassList then
16:                derivationList.add(A_s)
17:        return associationMatchMap, derivationList
```

Algorithm 6 Compare ENUM in `InstructorModel` and `StudentModel`

```
1: procedure COMPAREENUM(InstructorModel, StudentModel)
2:     instENUMList ← InstructorModel.getENUM()
3:     studENUMList ← StudentModel.getENUM()
4:     for all ENUM E_i in instENUMList, E_s in studENUMList do
5:         if syntacticMatch(E_s.name, E_i.name) or
6:             semanticMatch(E_s.name, E_i.name)) then
7:             enumMatchMap.put(E_s, E_i)
8:         else if E_s and E_i have similar literal values then
9:             enumMatchMap.put(E_s, E_i)
10:    studClassList ← StudentModel.getClass()
11:    studAttrList ← StudentModel.getAttribute()
12:    for all ENUM E_i in instENUMList do
13:        for all literal L in E_i.literal do
14:            for all Attribute A_s in studClassList do
15:                if A_s.Name.syntacticMatch(L.Name) or
   A_s.Name.semanticMatch(L.Name) then
16:                    consider A_s represent L
17:            for all class C_s in studClassList do
18:                if C_s.Name.syntacticMatch(L.Name) or
   C_s.Name.semanticMatch(L.Name) then
19:                    consider C_s represent L
20:    return enumMatchMap
```

Chapter 5

Grading Metamodels and Tool Support

This chapter discusses the metamodels and grading tool that supports the automated grading approach. We introduce the details of the metamodels that are defined to store the grade of a model and mappings between the student's model elements and the instructor's model elements. The grading approach and metamodels are implemented in the TouchCORE tool [26]. By presenting the graphical user interface(GUI) of the TouchCORE tool, we discuss the automated grading process in our tool and some key features in more details.

5.1 Metamodels

This thesis defines two metamodels to support the automated grading approach which is discussed in Chapter 4. Rather than augmenting the class diagram metamodel to support the definition of grades and matchings for model elements, this thesis defines separate metamodels. This is less invasive, as it leaves the class diagram metamodel unchanged, and hence all existing modelling tools can continue to work. Furthermore, we avoid referring to class diagram metaclasses directly, but instead use the generic `EClass`, `EAttribute` and `EReference` (as this thesis assumes metamodels expressed in the metametamodelling language *ECore* provided by the Eclipse Modelling Framework). As a result, the grading metamodels can be applied to any modelling language with a metamodel expressed in *ECore*.

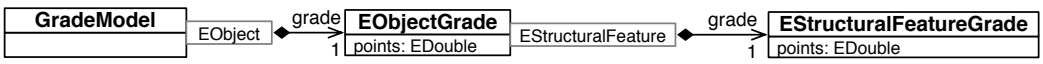

Figure 5.1: Grade Metamodel

Figure 5.1 shows the metamodel that augments any model expressed in ECore with grades. The *GradeModel* maps *EObject* to *EObjectGrade*, which contains a *points* attribute. That way any modelling element in a language that is modelled with a metaclass can be given points to. In order to give points for properties of modelling elements, *EObjectGrade* maps

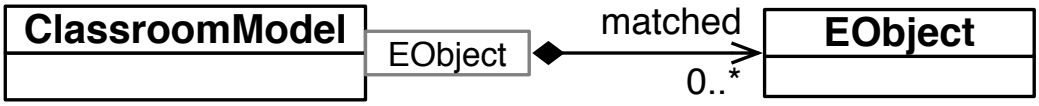

Figure 5.2: Classroom Metamodel

EStructuralFeature, the ECore superclass of *EAttribute* and *EReference*, to *EStructuralFeatureGrade*, which contains again a *points* attribute.

To illustrate the use of the grade metamodel, imagine a metamodel for class diagrams where attributes are modelled with a metaclass *CDAttribute* that has a *type EReference* that stores the type of the attribute. Now imagine the case where we want to give 2 points for the *age* attribute of the *Person* class in Figure 3.1, and an additional point if the type of the attribute is `int`. In this case one would create a *EObjectGrade* and insert it into the grade map using as a key *CDAttribute*, and assign the points value 2.0. Additionally, one would create a *EStructuralFeatureGrade*, insert it into the grade map using as a key the *EReference* `type` of *CDAttribute*.

Figure 5.2 depicts the *Classroom* metamodel which is used after the automated grading algorithm is run to store the mappings that were discovered. It simply associates with each model element in the instructor solution (EObject key) a list of *EObject*s in the student solutions that were matched by the algorithm. After the algorithm has been run, the matchings in this data structure can be updated by the grader if necessary. The information

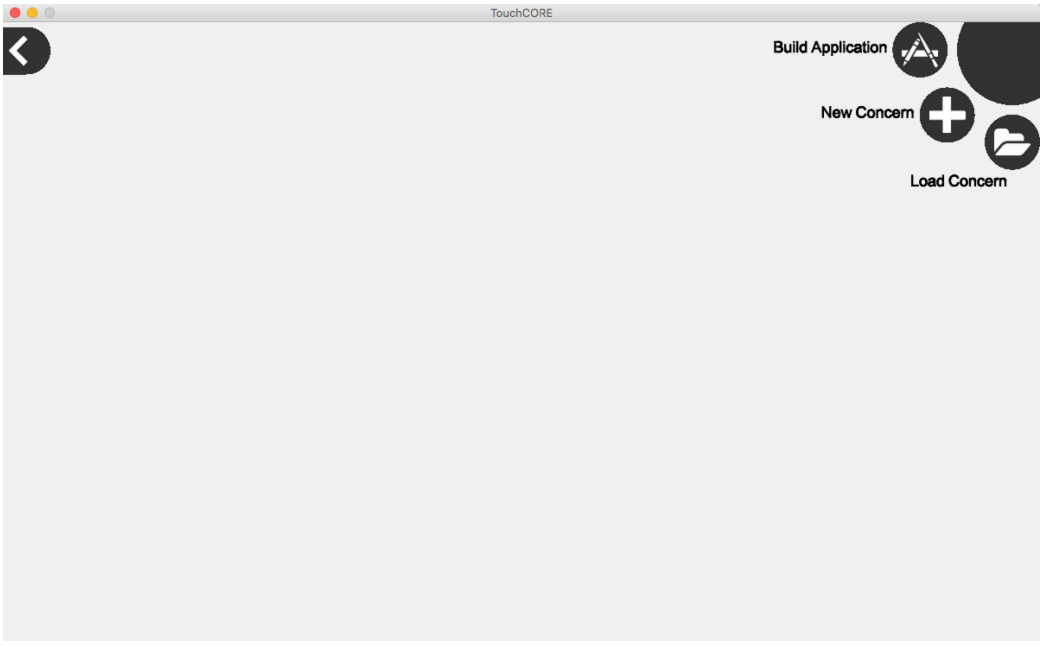

Figure 5.3: TouchCORE Main Graphical User Interface

can also be used to automatically update the grades of the students in case the instructor decides to change the point weights in the instructor solution.

5.2 Grading Tool Support

This section discusses the grading tool that we developed to support the automated grading approach. To use the grading metamodels described earlier and implement the matching algorithm, We extend a tool called TouchCORE [26].

TouchCORE is a multitouch-enabled tool for agile concern-oriented software design modeling aimed at developing scalable and reusable software

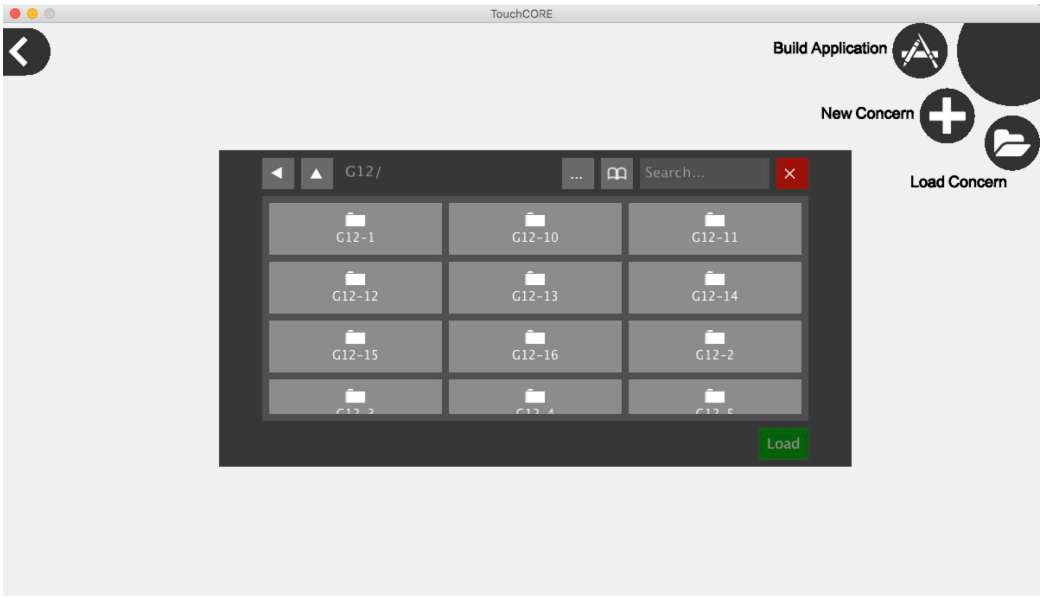

Figure 5.4: Student Models in TouchCORE File Browser

design models. TouchCORE is built on top of the Eclipse Modeling Framework (EMF)[58] and MT4J (multi-touch library for Java)[59]. Therefore, the TouchCORE tool supports different kinds of gestures, such as tap, drag and zooming. Fig. 5.3 shows the main graphical user interface (GUI) of the TouchCORE. A user can press the "New Concern" button on the main menu (the icon showing a "+" in Fig. 5.3) to select a folder to create models.

By pressing the "Load Concer" button, the user can also load the existing models from a file browser. The Fig. 5.4 shows the some student models in case study 2. Each student's model is placed in its own folder, which make the instructor easy to check and grade. Fig. 5.5 shows a student's domain model for hotel reservation system.

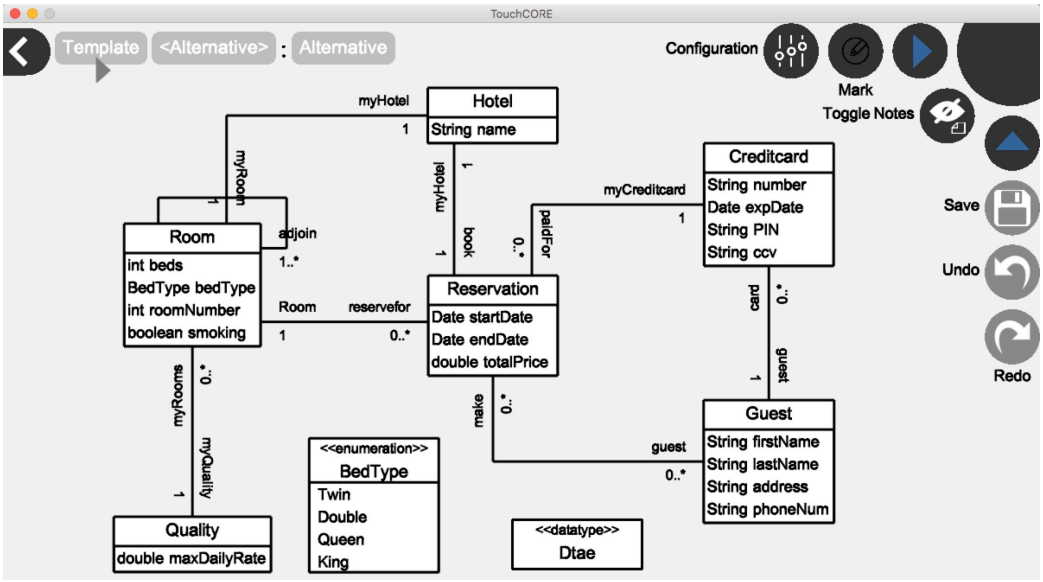

Figure 5.5: Student Models Display in TouchCORE

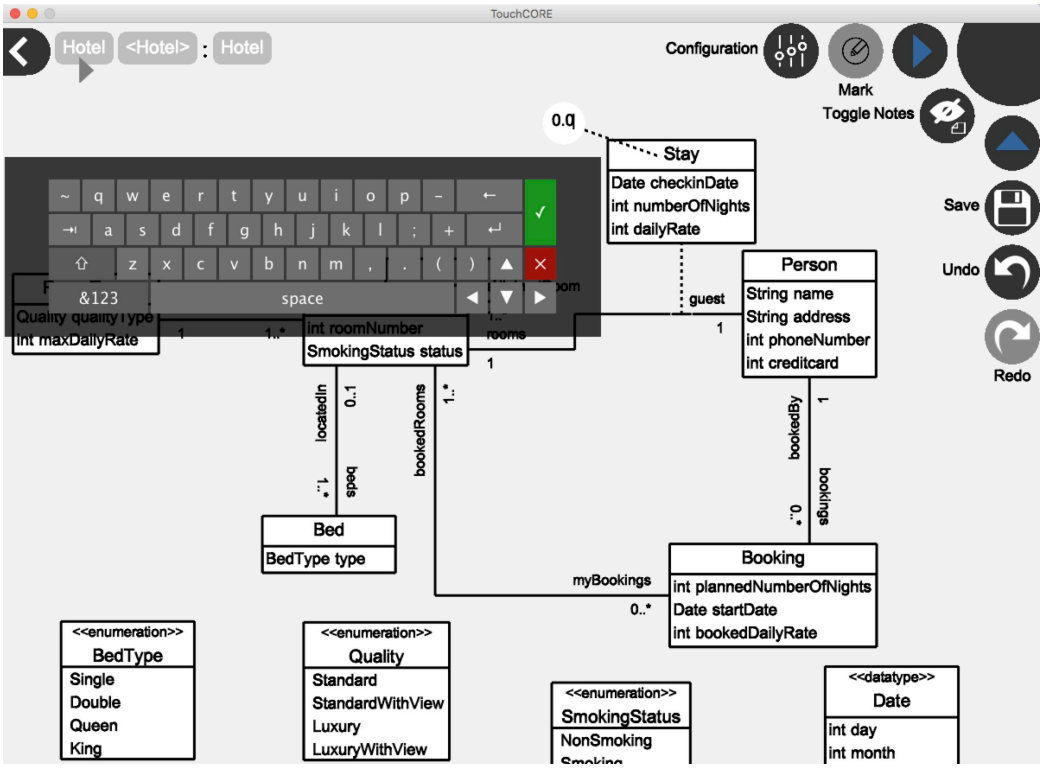

Figure 5.6: Give Mark for Model Element

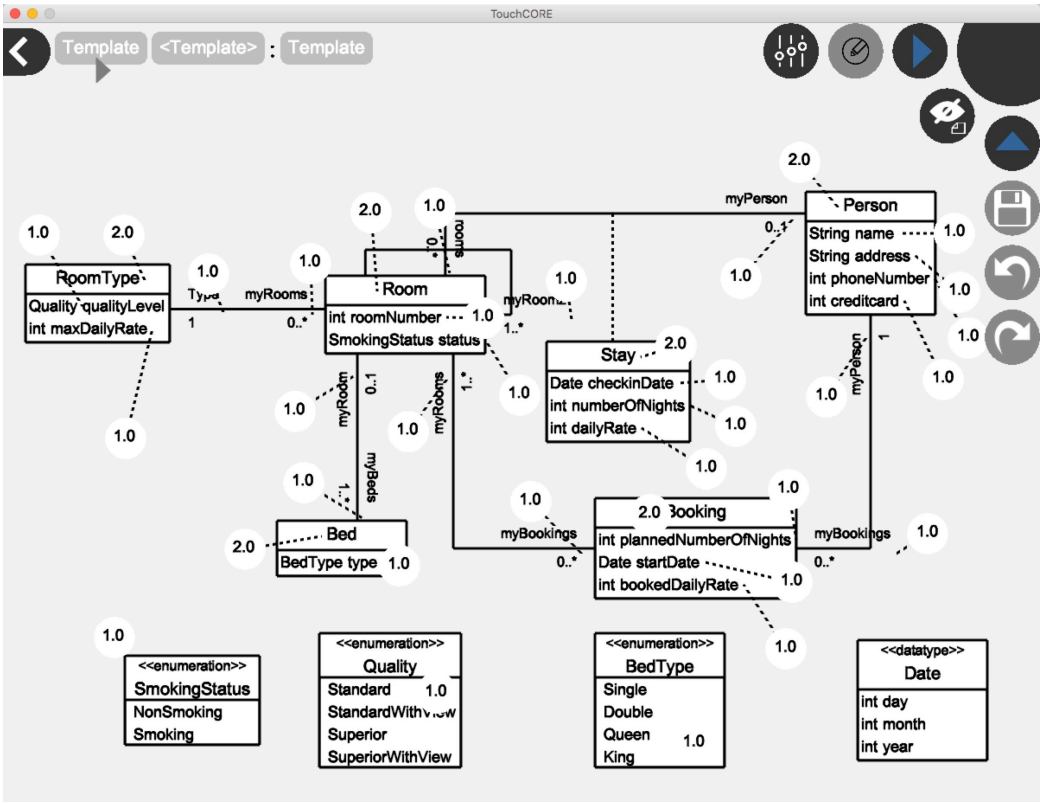

Figure 5.7: Instructor's Model with Grades in TouchCORE GUI

Fig. 5.6 shows the procedure for assigning grades to the model elements. By pressing on the "Mark" button on the top, the user can get into the marking mode. In the marking mode, the user can press on any model element, such as the class *Stay* in Fig. 5.6, and hold few seconds, then the grade will be successfully assigned to the model element. The grade is shown in a circle beside the model element and there is a dotted line which connects the model element with the grade. The default value of each grade is zero, but the teacher can double-click the grade to open the keyboard to change

the grade value. The user also can drag the grade in the tool to put it anywhere. Finally by tapping the "Mark" button again, the color of the button will become black, which means the user turns off the mark mode and can modify the model itself. Fig. 5.7 shows the instructor's model for hotel reservation system with grades.

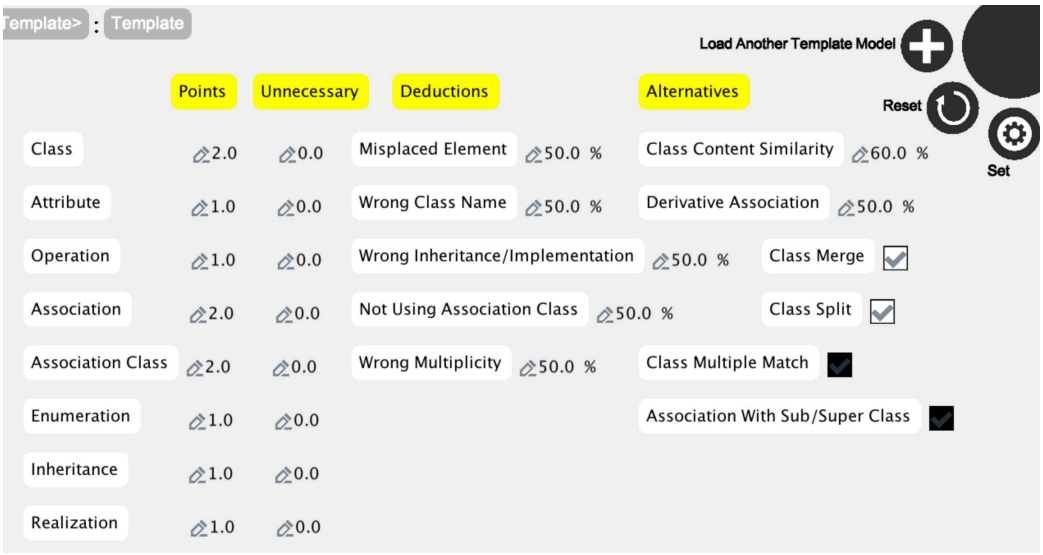

Figure 5.8: Grading Configuration Panel in TouchCORE GUI

After giving grades to all the model elements, the user can press on the "Configuration" button on the upper-right corner to get into the configuration panel, which is shown in Fig. 5.8. The details of the configuration panel will be discussed in the Chapter 7. After adjusting the configuration options, the user can press the "set" button to set this model to be the instructor model. If there are alternative solution models, the user can use the "Load Another Solution Model" to load alternative solution models. The details

about how to load alternative solution models will be discussed in Chapter 8.

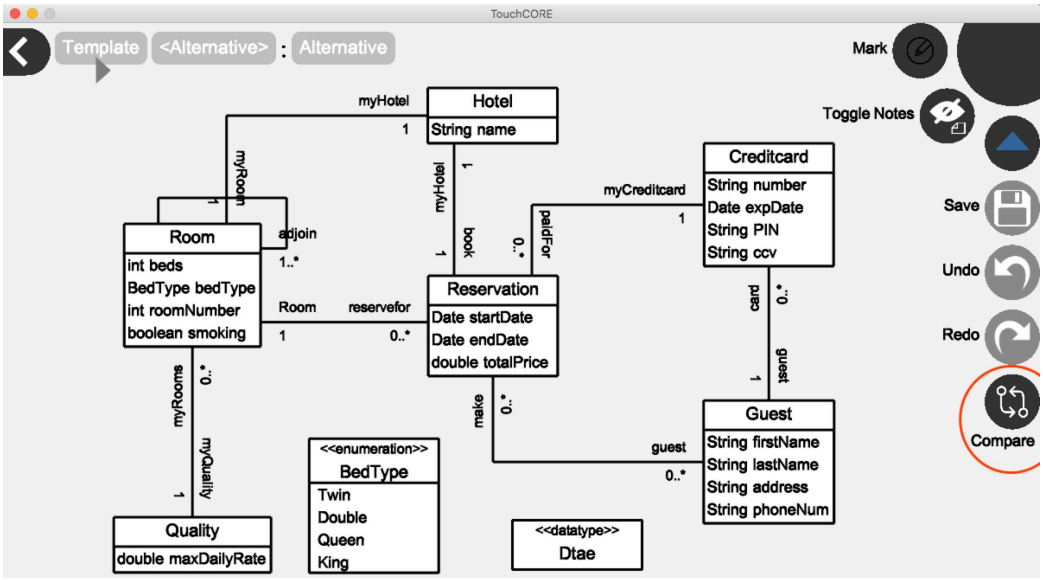

Figure 5.9: Compare Button in TouchCORE GUI

After setting the instructor's model and opening a student's model, there will be a new button named "Compare", which is highlighted by a red circle in Fig. 5.9. The user can press this button to compare the student model with the instructor's models by using the automated grading algorithms.

Fig. 5.10 displays the grading result for a student. The student's original model is shown in Fig. 5.5. When student makes any mistakes, the grade for the model element will be highlighted in yellow in the model. For example, in Fig. 5.10, the class *Quality* is match with class *RoomType* which has *maxDailyRate* as an attribute in the instructor model. However, the class name *Quality* can not be syntactically or semantically matched with the class name *RoomType*. So for the class *Quality* in the student model, half

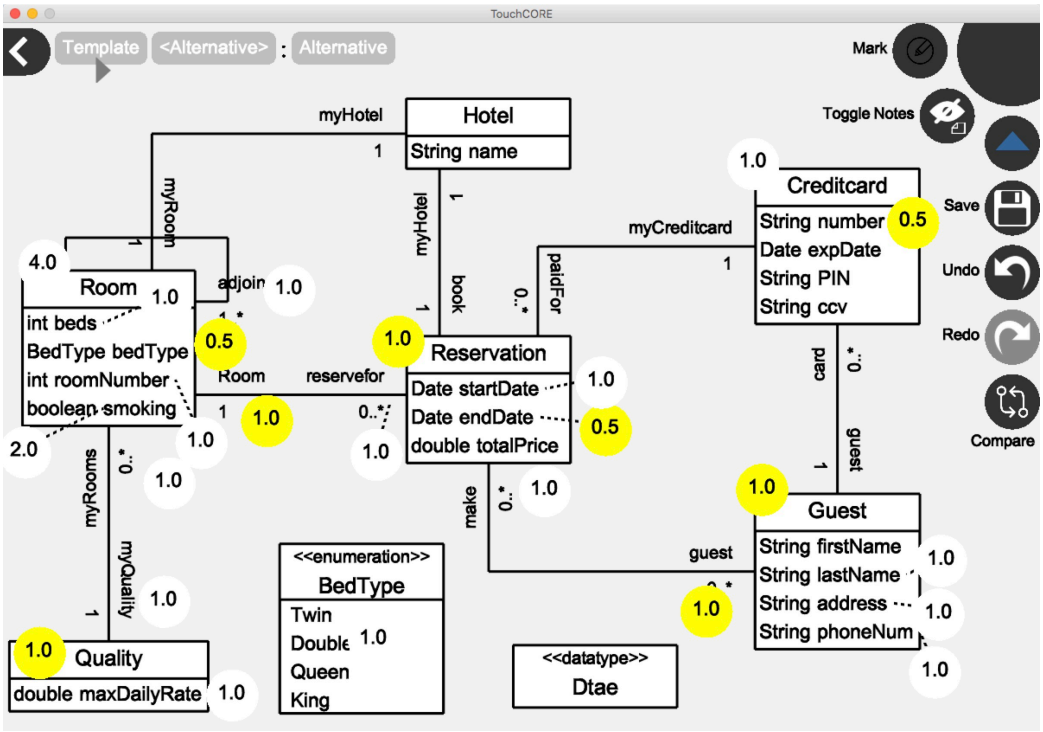

Figure 5.10: Student Model Grading Result in TouchCORE GUI

of the points are deducted. The tool also prints a feedback sheet for the student listing the points that she received and points that she missed. The student also receives a feedback sheet explaining where they received or lost marks. An excerpt of this sheet is shown in Table 5.1. Finally, our tool implementation allows the instructor to change the grading scheme in the instructor's solution model, such the model in Fig. 5.7. Since we keep a map to the students using the metamodels that were discussed earlier, it is easy to update the grades of the students based on the new grading scheme.

Table 5.1: Feedback for One Student's Model for Hotel Reservation System

Final Grade: 26.5/42
Classes: Guest: 1.0/2.0, matches with Class *Person*(Different Class Name) Room: 4.0, merge Class for Class *Bed* and *Room* Creditcard: 1.0, class represents attribute *creditcard* in Class *Person* ...
Attributes: roomNumber in Class *Room*: 1.0/1.0, matches with roomNumber in Class *Room* dailyRate in Class *Stay*: 0.0/1.0, missing attribute bedType in Class *Room*: 0.5/1.0, misplaced attribute ...
Associations: Room-Room: 2.0/2.0, matches with association between *Room* and *Room* Room-Quality: 2.0/2.0, match association between *Room* and *RoomType* Reservation-Guest: 2.0/2.0, match association between *booking* and *Person* ...
Enumeration: Quality: 0.0/1.0, missing enumeration

BedType: 1.0/1.0, match with enumeration *BedType*

...

5.3 Conclusion

In this chapter, we focus on the two metamodels and grading tool that support the automated grading. One metamodel stores grades for model elements, as well as the grades for the element's structure features. The other metamodel stores the mappings between student model elements with the instructor's model elements. By using these two metamodels, we can show the grades for model elements in our grading tool and allow the instructor easily changes the grading scheme since the tool can automatically update the grades for all students. Next, we present the TouchCORE tool where we implement our grading approach and metamodels. By showing the GUI of the TouchCORE tool, we demonstrate the automatic grading process in the TouchCORE tool and display the feedback information to the students.

Chapter 6

Case Study

To investigate the effectiveness of automated grading, this thesis analyzes models handed in by students as assignments of two undergraduate courses. Each course was offered at a different university, and each instructor has a PhD degree in Computer Science and several years of experience teaching university-level software engineering courses. The students and the instructors did not use the tool that is used in this case study. The instructors took note of the time they needed to grade each student model on paper. After they finished grading the assignments, the instructors gave us access to the paper copies of the student hand-ins, as well as the model solution that they used for grading, and their grading scheme that detailed the number of points they had assigned to each relevant model element. We then manually reproduced all models in the tool.

In this chapter, we start by discussing the set-up for the case studies. Then we show the first grading result and discusses the limitations threats to validity. The further assessment of automated grading will be discussed in the following chapters.

6.1 Case Study 1: Animal Design Model

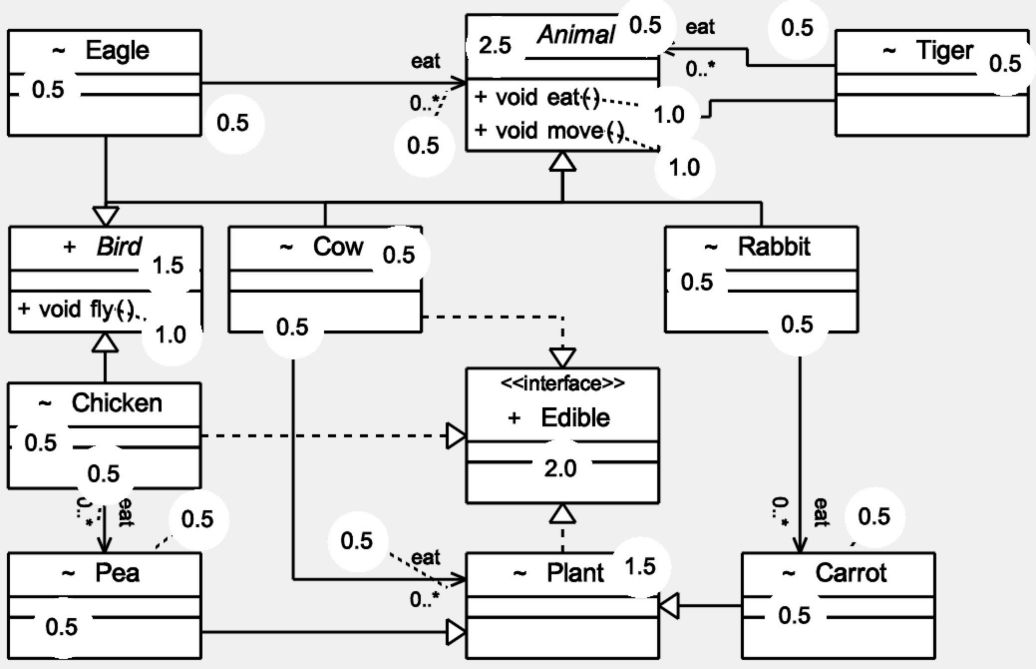

Figure 6.1: Animal Design Model of Instructor

The first assignment (case study 1) was performed in the context of a beginner software design and modeling course. This is the first course in software modeling in the curriculum of the students, and students took only one prior programming course before enrolling in this course. The problem

description was as follows:

> *An animal can eat and move. A bird is an animal that can fly, and chicken and eagle are birds. Cow, rabbit, and tiger are animals too. Tigers and eagles can eat any animal. A chicken can eat a pea, a cow can eat any plant, and a rabbit can eat a carrot. Cow, chicken and plant are edible.*

The students were asked to create a design class diagram that includes operations to model behaviour, and that uses association and inheritance to express structural dependencies and relationships. Fig. 6.1 depicts the instructor's solution model and grading scheme for the animal question. In this model, there are 5 kinds of animals (*Cow*, *Tiger*, *Rabbit*, *Eagle* and *Chicken*), 2 kinds of plants (*Pea* and *Carrot*), and multiple relationships between the classes. Subclassing and inheritance is used to express that, e.g, a *Chicken* can only eat *Pea*s, and a *Tiger* can eat any animal. The points for a model element are shown in small circles next to the model element, e.g. the class *Animal* yields 2.5 points. We collected 103 student assignments for this case study.

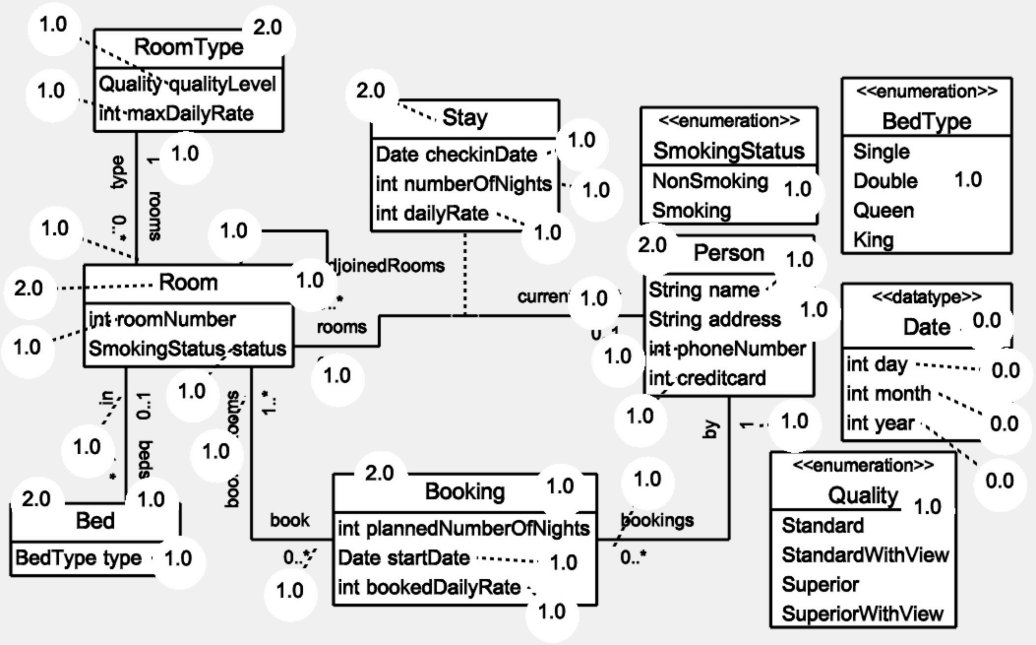

Figure 6.2: Hotel Domain Model of Instructor

6.2 Case Study 2: Hotel Domain Model

The second experiment (case study 2) was performed on advanced undergraduate students that are enrolled in a software engineering project course that runs over 2 semesters. Most students were in their final year of Computer Science or Software Engineering studies. As such, these students have advanced programming skills, considerable knowledge of data structures, and important software development and modeling experience. The students were tasked to create a domain model for the following problem statement:

You have been asked to create a system to manage the front-desk activities of a hotel. The system will be used to enter reservations as

well as to check people in and out of the hotel.

The hotel contains 30 rooms in which guests can stay. Some hotel rooms adjoin others, i.e., there are internal doors between them, so a guest can stay either in an individual room or a suite (multiple rooms). Each hotel room is assigned a quality level (e.g., a larger room or a room with a view would be better than a smaller room without a view). Each room also has a certain number and type of beds, a room number, and a smoking/non-smoking status. Each quality level has a maximum daily rate, although the rate that a guest pays may be less.

When a hotel guest wishes to make a reservation, the hotel clerk asks him or her what nights he or she wants to stay and the type of room he or she wants. The system must verify if room(s) are available on those nights before allowing a reservation to be made.

The hotel needs to record basic information about each guest, e.g., his/her name, address, telephone number, credit card, etc. A reservation can be cancelled at any time.

When a guest checks in, a room is allocated to him/her until s/he checks out. The system must keep track of the guest's account, and print his or her bill.

Fig. 6.2 shows the instructor's solution and grading scheme for the Hotel

domain model. The class *Person* is associated with *Room* via an association class called *Stay* to keep track of the guests staying at the hotel. The class *Booking* is related to both *Room* and *Person*. *Room* has a reflexive association to capture which rooms are adjoined. *Room*s also have associated *Bed*s and a *RoomType*. The model declares three enumeration types, *Quality*, *SmokingStatus* and *BedType*. We collected 89 student models for case study 2.

6.3 First Automated Grading Result

To measure the effectiveness of the automated grading algorithm, after the first round automated grading, we compared the total score of each student obtained by the automated grading algorithm with the score determined manually by the instructor. Here is a summary of the deduction policy that we adopted at first:

- Class use the different name: deduct half of point.
- Misplaced attribute/operation: deduct half of point.
- Derived association: give half of the point of the correct derived association.
- Missing element: deduct whole point.

- Wrong inheritance/realization relationship: deduct whole point.

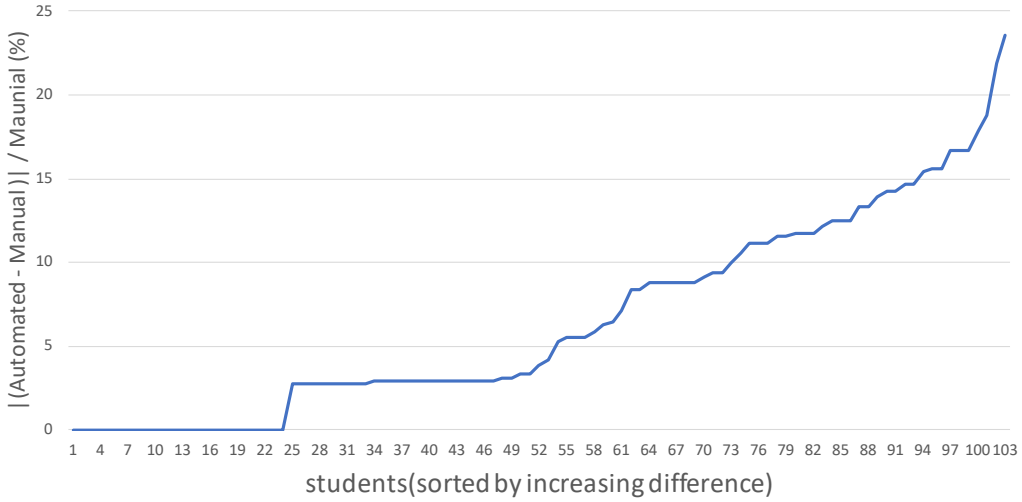

Figure 6.3: Auto vs. Manual for Animal Case Study

After the first round automated grading, the grading results have been collected and compared with the instructor's manual grading result. The blue line in Fig. 6.3 shows the difference between the instructor's manual grades and the automated grading for *case study 1*. We notice that for 24 students the score of the automated grading algorithm is identical to the instructor's score. For 73 students the difference is less than 10%, and only for 3 models the difference is above 20%. The average difference for the 103 models is 6.5%.

The blue line in Fig. 6.4 plots the difference in percentage between the instructor's manual grading and the automated grading for case study 2. We note that for no model the algorithm obtained the exact same score as the instructor. Out of 89 models, only 6 models have a score with less than 10%

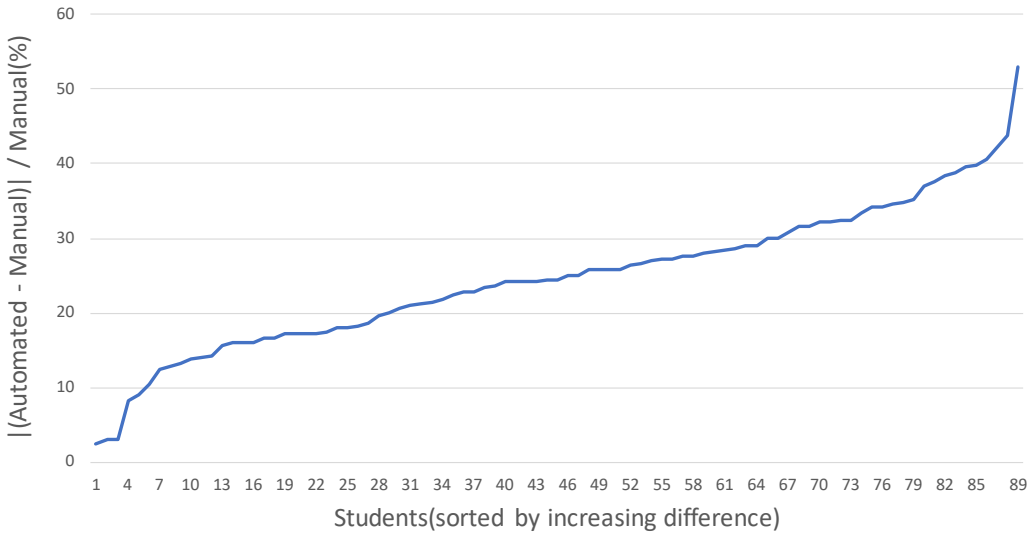

Figure 6.4: Auto vs. Manual for Animal Case Study

difference, and 30 models have less than 20% difference. In other words, for *more than half of the models* the score obtained by automated grading differs significantly from the one obtained by manual grading. The average difference for 89 models is 24.6%.

6.4 Limitations and Threats to Validity

Currently, the tool that is discussed in Chapter 5 in this thesis, does not support assigning points to visibility, aggregation and compositions relationships, and role names in associations. However, in the case studies, the instructors did not assign grades to those elements.

• **Internal Validity** threats related to biases when grading the assignments.

We mitigated this threat by automatically grading the assignments only after they have been manually graded by the two instructors. Instructors did not know about the automated grading results until after they have already graded the assignments manually.

• **External Validity** threats relate to the generalizability of our findings. We mitigated this concern by conducting two case studies of undergraduate assignments done by students of two different universities covering introductory and advanced courses in software engineering. Our experiments have been conducted on class diagrams, so it remains to be seen whether our conclusions can be generalized to other models, e.g., behavioural models.

• **Construct Validity** threats relate to the difficulty in finding alternative grading criteria. We allowed the hard-coded algorithm parameters and options in the tool to be configured. It is possible, tough, that there are common grading options that we did not consider. We mitigated this concern by consulting two instructors with several years of teaching experience, and as a result added two new configuration options to the original algorithm. Furthermore, we extended the tool to be able to consider any number of alternative solution models, which allows an instructor to cover any possible alternative grading situation.

6.5 Conclusion

This chapter outlines the case studies that we conducted to evaluate the effectiveness of the automated grading algorithm which is discussed in Chapter 4. We apply the grading algorithm to two case studies. There are 103 student models in case study 1 and 89 student models in case study 2. The details of two modeling questions are explained and we show the first result. Comparing the automated grading result of the grading algorithm with instructor's manual grading result, the average difference for case study 1 is 6.5%, while for case study 2, the average difference is 24.6%. The automated grading result, especially the result of case study 2, indicates that the current algorithm's grading strategy and the deduction policy are different with two instructors' strategies, which results in large differences. Therefore, the grading tool should be able to tailor different grading strategies to different instructors, which will be detailed discussed in the next chapter.

Chapter 7

Exploring Grading Strategies

This chapter discusses the configuration panel that can tailor different grading strategies to different instructors. At first we choose a sample of assignments from the two case studies and discuss with respective instructor about their grading strategies. Next, we make the automated grading tool configurable and answer the research question 1. Finally, in order to answer the research question 2, we use the configuration options to adjust the automated grading algorithm to match the grading strategies of the instructors and re-grade all student models. The results show that by using the configuration panel, the automated grading result becomes closer to instructor's manual grading.

7.1 Grading Strategies Assessment

Research in psychology of education suggest that instructors are not uniform in their grading styles [60; 61]. Personal and socio-cultural issues affect how instructors choose their grading criteria [62; 63], and they consider customized criteria that are suitable for different situations [64–67]. Even when encountering the same problem, different instructors may choose different criteria that result in different scores [67]. Some instructors can be lenient, while others can be strict. Also, instructors adjust their grading criteria based on the level of expertise of the students in the class.

This was the case in our experiments, as the students in case study 1 were mostly first year undergraduates, whereas the students in case study 2 were in their last year. Similarly, the problem description in case study 1 is clear and the resulting solution design model straight forward, whereas the description in case study 2 is more complex and the solution domain model non trivial.

The difference in efficiency of the automated grading algorithm in the two case studies reported in the previous chapter motivated us to investigate whether different grading styles and strategies are also employed when grading models. If yes, an automated grading algorithm would have to be extended with configuration options in order to be able to adapt the grading

style to different contexts.

7.2 RQ1: Are there different grading criteria for class diagrams?

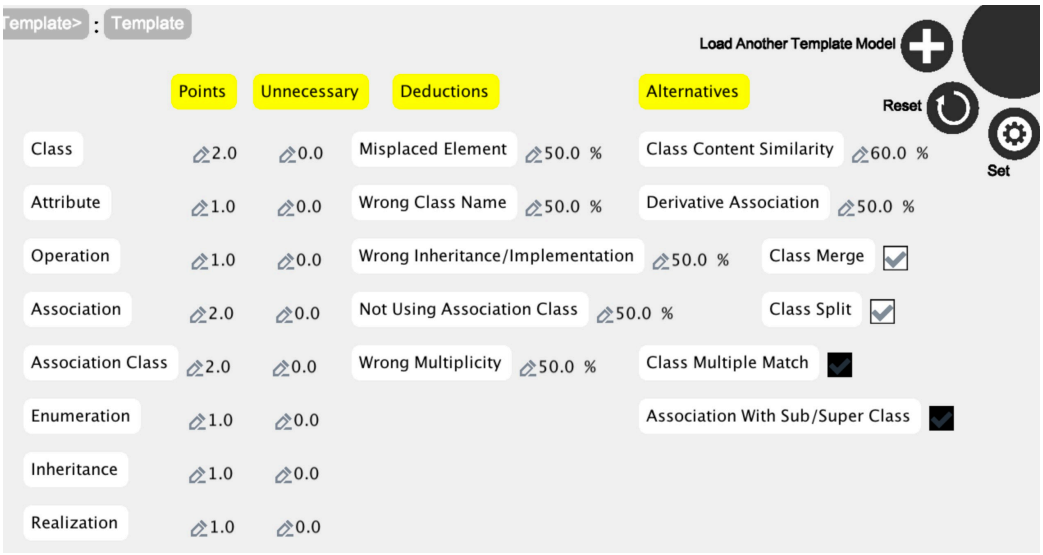

Figure 7.1: Grading Configuration Panel

To answer this question, we randomly chose about 20% of the assignments (i.e, 21 assignments for case study 1 and 16 assignments for case study 2) and showed the scores obtained with the original automated grading algorithm to the respective instructor. The original automated grading algorithm means the algorithm that presented in the chapter 4. They compared the automated grading scores with the manual grading scores for each student. Whenever there was a difference, we asked the instructor to explain

why they gave the student additional points.

7.3 Automated Grading Configuration Settings

We modified the grading tool to be configurable by means of a configuration panel shown in Fig. 7.1, which is the same figure with the Fig. 5.8 in chapter 5. For convenience, we show the figure again here. The Configuration panel allows the instructor to tailor the algorithm to his/her specific needs. The configuration panel has four parts: *Points*, *Unnecessary*, *Deductions* and *Options*.

7.3.1 Points

lists the default points assigned to model elements in a class diagram. The model elements with default point values are *classes*, *attributes*, *operations*, *associations*, *association classes*, *enumeration* types, *inheritance* and *realization* relationships. Setting *class* to 2.0 means that, by default, every class in the instructor's solution model contributes 2 points to the total score. Of course, the instructor can override this default value by assigning a different points value to a model element in the instructor's solution model.

7.3.2 Unnecessary

Students sometimes add irrelevant elements to their model. Although these elements many not make the model incorrect, they may add unnecessary details. For example, a domain model should not contain platform-specific structural details. Also, some students might try to optimize their grade by adding everything they can possibly think of to their model to increase the probability of not omitting what the instructor is looking for. To deal with this situation, the instructor might want to deduct points for having unnecessary elements in the model. This was, for example, the strategy of the instructor of case study 1.

The *Unnecessary* list in the configuration panel allows instructors to specify how much they want to deduct if students have unnecessary classes, attributes, operations, association ends, association classes, enumerations, inheritance or realization relationships in their model.

7.3.3 Deductions

The third group of configuration settings is *Deductions*, which lists strategies for making deductions. The default deduction strategy for all kinds of mistakes was initially hard-coded in the algorithms to 50%. Now we allow instructors to specify a deduction percentage for each kind of mistake.

The four kind of mistakes currently handled are:

(1) Misplaced Element. A student may put an attribute or an operation into the wrong class or association class.

(2) Wrong Class Name. A student might use a different class name than the one chosen by the instructor. In some cases, although a class may have the expected content or associations with other classes, the name chosen by the student might be so misleading that it deserves a deduction.

(3) Wrong Inheritance/Realization. A student may use a wrong inheritance or realization relationship between classes or interfaces.

(4) Not Using Association Classes. In cases where an association class is necessary, a student might forget to use one or use a class instead.

(5) Wrong Multiplicity. A student may use the wrong association end multiplicity value, i.e., a wrong lower bound or upper bound.

7.3.4 Options

The fourth part in the grade configuration panel relates to optional matching steps in the algorithm. In total, we identified 6 configuration options that can alter the grading process.

(1) Class Merge. In some cases it is reasonable that students may use one class to represent two classes of the instructor's solution model.

(2) Class Split. Students may choose to split the content of one class of the instructor's solution model into two separate classes.

(3) Class Content Similarity. The algorithm matches classes not only by their names but also by examining their contents. The original algorithm already performed structural matching, but the similarity threshold was hard-coded. Now instructors can set how similar classes need to be, content-wise, to decide whether they should form a match.

(4) Derived Association. A derived association is an association that can be derived from the existing associations in the instructor's solution model. The point for a derived association is hard-cored in the original algorithm. Now instructors can set how many points should be given to students for each derived associations.

Figure 7.2: Class Multiple Match Example Model

(5) Multiple Matching of Classes and Attributes. In the original algorithm, when a class C in the student's model has been matched with a class in the instructor's solution model, the same class could not be used again for another purpose, e.g., to represent an attribute of the instructor's solution model. Fig. 7.2 illustrates this case, which depicts a partial model for a

student. In this model, class *QualityLevel* in the student's model could match with class *RoomType* in the instructor's solution model shown in Fig. 6.2 because both classes have similar contents and both are associated with the class *Room*. However, the class name *QualityLevel* in Fig. 7.2 could also be considered to represent the attribute *qualityLevel* in the instructor's solution model. With the original algorithm, the student would loose the points for not having the attribute *qualityLevel*. Out of the 16 initial models from *case study 2*, we discovered that in 8 cases the instructor used this strategy to give the students additional points. After reviewing all 89 models for *case study 2*, we found that the instructor used this strategy in 23 models.

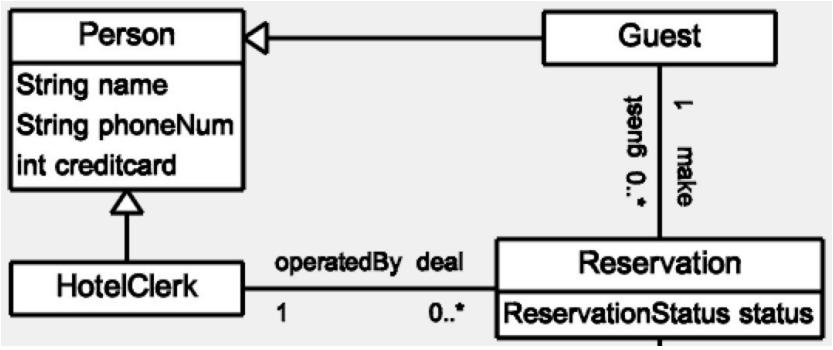

Figure 7.3: Association With Subclass Example Model

(6) Association With Sub/Super Class. When it comes to associations and inheritance, it is possible that students may associate with a subclass or a superclass rather than the original class, especially when students have added additional inheritance relationships to their model compared to the instruc-

tor's solution. For instance, Fig. 7.3 shows a student's partial model in *case study 2*. The grading algorithm matches the class *Person* and *Reservation* in the student model with the class *Person* and *Booking* in the instructor's solution model. However, the student adds two additional classes, *Guest* and *HotelClerk*, and adds an association between *Reservation* and *Guest* and, between *Reservation* and *HotelClerk*. We notice that the class *Guest* and *HotelClerk* are subclasses of the class *Person*. Therefore, it is correct to match the association between *Reservation* and *Guest* in the student's model with the association between *Booking* and *Person* in the instructor's solution model shown in Fig. 6.2. In *case study 2*, the models of 4 of the 16 students in our first random group and and 18 students out of all 89 students had similar inheritance and association relationships to superclasses in their model. This grading strategy was not supported in the original algorithm, so we added it.

7.3.5 Algorithm Configuration Assessment

Using the configuration options discussed above, the instructor can now configure the grading algorithm to adapt to the course level and to suit his/her style. By adjusting the deduction percentage, the instructor can determine how many points students loose when they make mistakes. Also,

the instructor can use the grading configuration options to adjust the sensitivity of the similarity checks, as well as enable or disable optional matching strategies.

7.4 RQ2: Does the use of configuration settings improve the accuracy of automated grading?

To answer this question, we used the discussions from the interviews with the instructors to configure the algorithm to match their grading strategies, and then applied the configured algorithm to re-grade all student models.

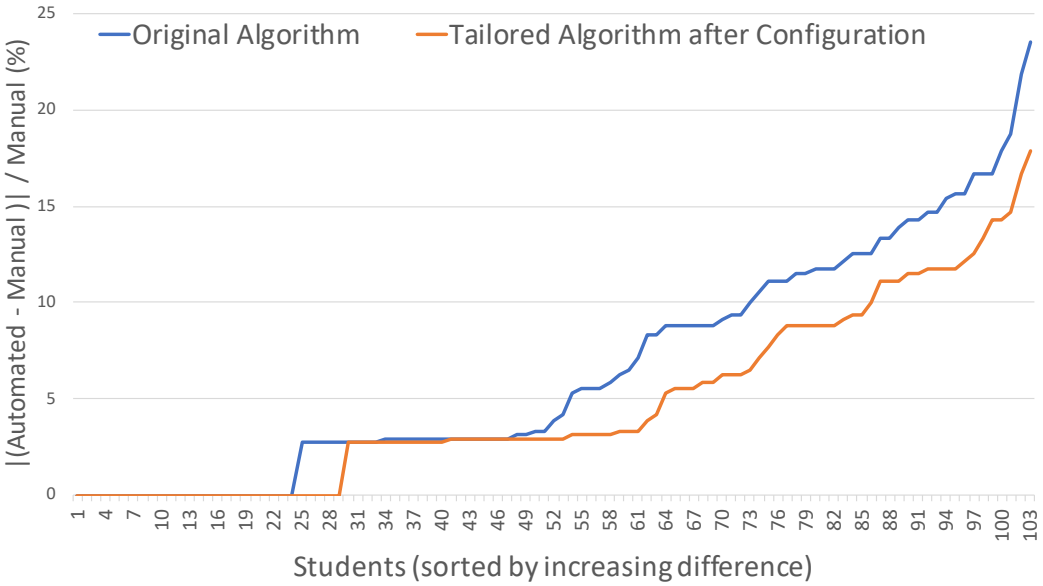

Figure 7.4: Original vs. Tailored for Animal Case Study

7.4.1 Case Study 1: Animal Design Model

For case study 1, the instructor suggested that deductions for *Wrong Inheritance/Realization* should be set to 100%, because this exercise was designed specifically to test their knowledge about inheritance relationships. For example, in this assignment, class *Cow*, *Rabbit* and *Tiger* should inherit from the class *Animal*. If a student fails to model these inheritance relationships or misplaces them, they will loose 100% of the points. The instructor also suggested to choose the option *Association with Sub/Super Class*, because for the association *Tiger-Animal* in the instructor's solution model, it is acceptable that a student replaces it with associations between the class *Tiger* and all the subclasses of the class *Animal*.

Fig. 7.4 shows the difference between the instructor's manual grades and the automated grading for *case study 1*. The blue line shows the difference for the original algorithm, whereas the orange line shows the difference for the tailored algorithm. We notice that the tailored algorithm slightly outperforms the original one. Thanks to the configuration settings, the number of perfectly graded models increased from 24 to 29. With configuration, 86 students received grades with less than a 10% difference compared to manual grading, and no model differed by more than %18. Out of a maximum of 19 points, the manual grade average was 16.36, whereas the average of the

grades of the tailored algorithm was 15.72, and the average of the original algorithm 15.34. Configuration therefore reduced the average difference in points from 1.02 to 0.75 (or from 6.52% to 4.80%).

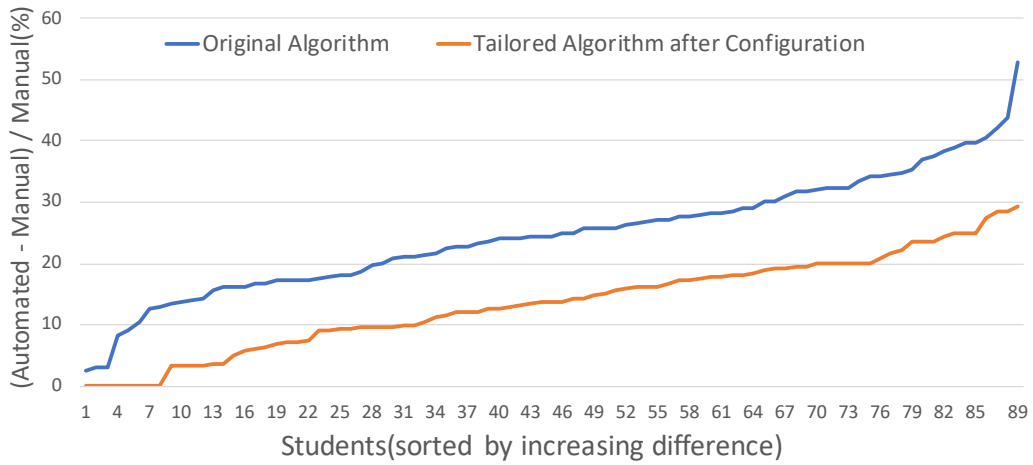

Figure 7.5: Original vs. Tailored for Animal Case Study

7.4.2 Case Study 2: Hotel Domain Model

For case study 2, the instructor declared that: (1) As long as a class in the student's model can be matched with a class in the instructor's solution model, the instructor did not deduct any points for making a mistake in the class name. (2) Putting an attribute into a different class was not considered wrong, as in most cases the student would already loose points for a missing class. (3) No points were deducted for unnecessary inheritance or realization relationships.

In addition, the instructor chose to (1) Match a class in the student

model multiple time with classes/attributes in the instructor's solution model. (2) Allowed the students to establish associations to superclasses/subclasses without loosing points.

Fig. 7.5 plots the difference between the scores of the instructor and the ones obtained with automated grading. The blue line represents the original algorithm without configuration, whereas the orange line shows the scores of the algorithm tailored according to the instructor's declarations. Whereas there were no models for which the original algorithm achieved a score identical with the one given by the instructor, the tailored algorithm scores perfectly for 8 of the 89 models. Furthermore, 32 models now have a grade that deviates by less than 10% compared to only 6 models for the original algorithm. With the tailored algorithm, 84% of the scores have a difference of 20% or less, and the maximum difference is under 30%. For the whole class of 89 students, the average difference between the manual grading the automated grading using the adjusted configuration was 13%, while with default configuration the difference was 24.6%.

7.5 Conclusion

Grading strategies can vary from instructor to instructor. Also, the level of expertise of the students in the class also effect the instructor's grading

criteria. Therefore, it is essential to make the automated grading tool configurable. In this chapter, we present a configuration panel that is able to tailor different grading criteria for instructors. In this configuration panel, we identified 27 configurable settings, including basic ones, such as default points for classes and attributes, and more complex options such as matching a class with multiple classes or attributes. When automated grading is configured to match the context and instructor's style, it produces scores that are close to instructor's manual grading. In case study 1, the average difference between manual and automated grading was less than 5%, while in the case study 2 the average difference was 13%. Compared with the first result, the configuration panel can reduce the difference between automated grading result and the instructor's manual grading result.

Chapter 8

Assessing Automated Grading

So far, this thesis has verified the automated grading tool through a comparison between the configured automated grading result with the original automated grading result. And the first two research questions has been answered. This chapter discusses how to validate the automated grading tool. The first section discusses the feature that the automated grading tool can deal with multiple correct solutions and answers the research question 3. The second part of this chapter is focused on the effectiveness of the automated grading. By answering the research question 4 and 5, we demonstrate that the automated grading is able to improve efficiency and ensure fairness in the grading process.

8.1 Dealing with Multiple Solutions

Modeling problems are considered to be ill-defined and open-ended [68; 69]. When it comes to modelling, more than one correct answer can exist for a particular problem [70]. When we discussed the grading of models of the Hotel case study, the instructor informed us about two cases where more than one correct solution is possible.

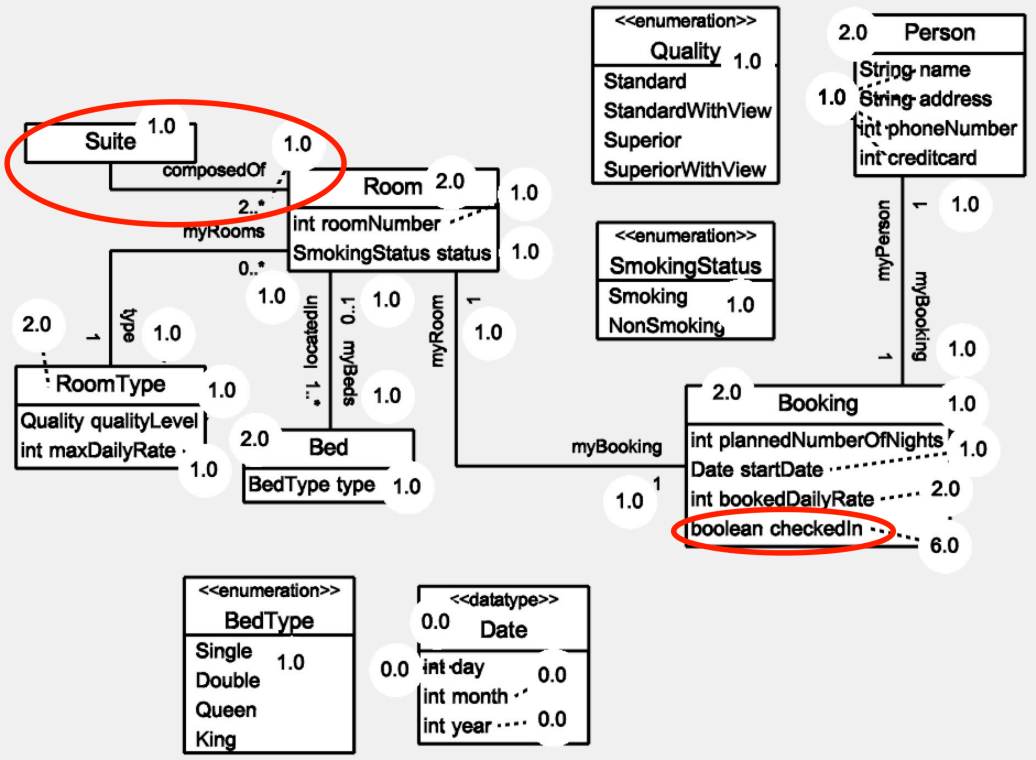

Figure 8.1: Alternative Solution for Hotel Domain Model

The first case is about the reflexive association *adjoinedRooms*, which instructor's solution uses to represent that more than one room can be grouped together as a suite as shown in Fig. 6.2. In that case study, the description

states that a guest can stay in an individual room or a suite, which is composed of several adjoined rooms. However, Fig. 8.1 shows an alternative way to model this situation. In Fig. 8.1, adjoined rooms are reified in a class named *Suite*. A *Suite* groups 2..* rooms together. The second case is related to *Stay* association class in Fig. 6.2, which is used to ensure that a Person that has a booking stays in the room. An alternative solution would not use this association class. Instead, it would use an attribute (*checkedIn*) in the *Booking* class as shown in Fig. 8.1. Therefore, we adjust our grading tool to allow instructors to upload multiple solution models. In the configuration panel, which is shown in Fig. 7.1, there is a "+" button on the upper-right corner with the name "Load Another Solution Model". After clicking that button, the tool will show a file browser, which is shown in Fig. 8.2. Instructors can use the file browser to load a different solution model.

We added each of these two cases as an alternative instructor solution model. As a result, for case study 2, we have 3 instructor solution models, the model that was shown in Fig. 6.2, the model that uses the class*Suite*, and the model that uses the attribute *checkedIn*.

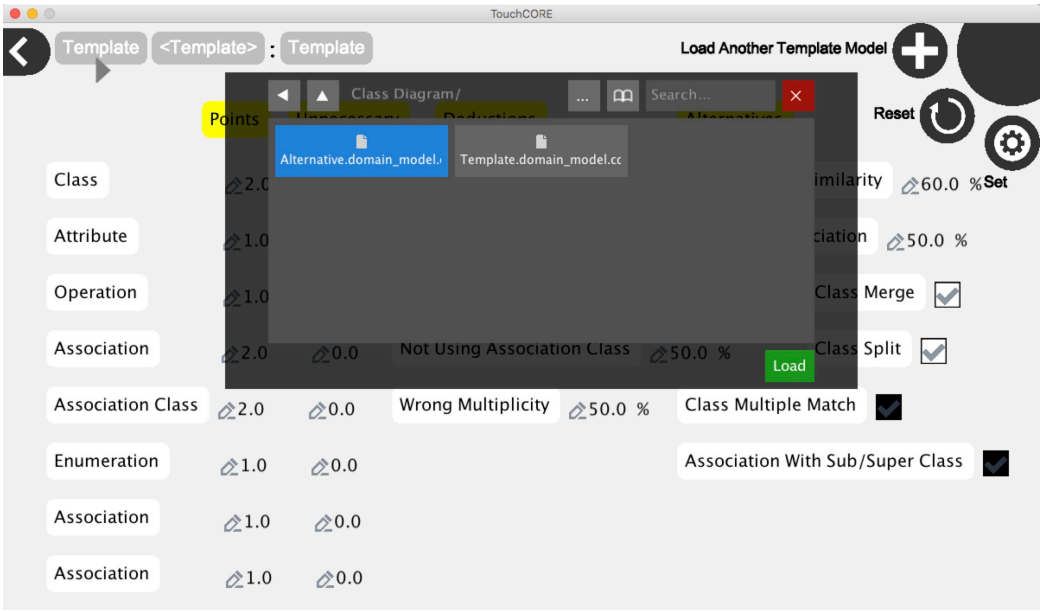

Figure 8.2: File Browser for Loading Alternative Model

8.1.1 RQ3: Does the accuracy of automated grading improve when multiple solutions are matched against?

In order to grade problems that have more than one correct solution, we added the capability to load more than one instructor model in the configuration panel. When the algorithm is running, the student's model is compared with each instructor model that is currently loaded. The highest mark is taken as the student's final grade.

In *case study 2*, where the average for manual grading is 31.19. When only one instructor model was used for grading, i.e. the solution shown previously in Fig. 6.2, the average grade for automated grading using the

configured algorithm was 26.97. When using an additional solution model (i.e. the one with class *Suite*), the average increased to 27.32. We identified 17 out of 89 students who obtained higher grades with the this additional case, 14 created the class *Suite* in their models and 3 students added an attribute to represent the class *Suite*. When we used all three solution models, the original solution shown in Fig. 6.2, the solution with the class *Suite* and the solution with the attribute *checkedIn*, the average became even closer to that of manual grading: 28.37. We identified 23 students who used the attribute *checkedIn*. As a result, the average difference between manual grading and automated grading for case study 2 improved to 9% when all 3 instructor solution models were used. The tool chooses the solution that produces the highest grade for the student.

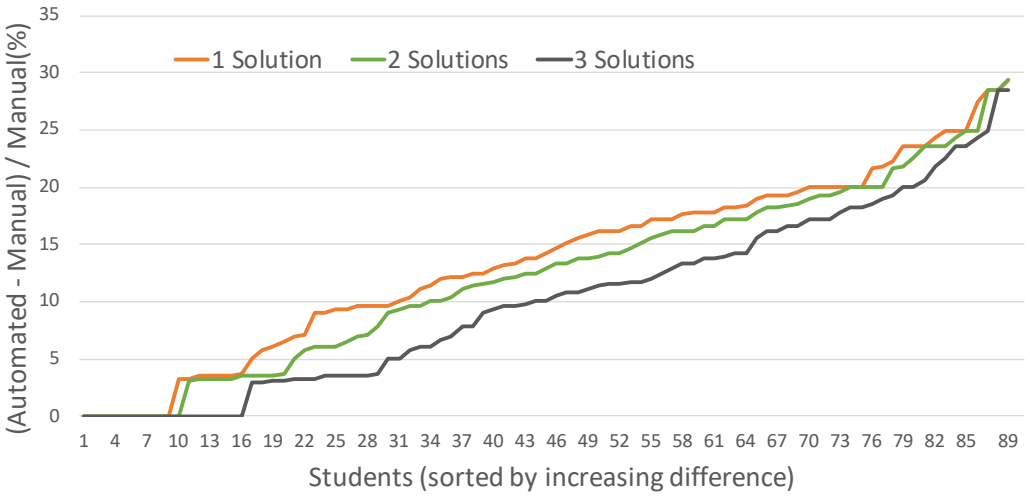

Figure 8.3: 1 vs 2 vs 3 Solutions for Hotel Case Study

Fig. 8.3 shows the percentage difference between automated grading and manual grading when using 1 solution (orange line), 2 solutions (green line), and 3 solutions (black line). The orange graph was discussed before in RQ2. We notice that the percentage difference decreases when using 2 solutions, and decreases further when using 3 solutions. When grading with 1 solution, 32 models had a grade difference that was less than 10%. With 2 solutions, 35 models had a grade difference less than 10%, and when considering 3 solutions, this number increased even to 46, i.e. more than half of the models. Furthermore, the automated grading algorithm determined the same score than the instructor for 16 students. In the end, with 3 solutions, the average grade difference dropped from 13% to 9%.

8.2 Assessing Automated Grading Effectiveness

This section sheds light on the potential of automated grading in the classroom by reporting on the time saved in our two case studies, as well as the fairness issues that we discovered in the manual grades thanks to automated grading.

8.2.1 RQ4: Does automated grading save time?

Grading is a tedious and time consuming process [39; 71–73]. To study whether automated grading saves time, we asked the instructors to record the time that they needed to manually grade each student model. We also measured the time that the tool needed to grade each model.

For case study 1, the instructor spent approximately 2 hours to grade the entire class. On average 1 minute and 7 seconds was spent on manually grading one student model. The tool spent 0.1 seconds to grade one model, i.e., about 11 seconds to grade the entire class automatically.

For *case study 2*, it took the instructor 4 hours and 11 minutes to grade the 89 students. On average the instructor spent 4 minutes and 10 seconds to manually grade one student model. The automated tool takes 0.16 seconds to finish grading one model, i.e., it took 14.24 seconds to grade the whole

class.

Of course this comparison is not fair. Both instructors explained that typically they prepare one initial model with the grading scheme before they start grading the first student. This step is also required for automated grading.

As the instructors are grading the first students, they might encounter a situation in which a student model contains an alternative solution that the instructor had not considered in his initial model. In that case, the instructors give the student full points. Ideally they would also write down this alternative solution, because from now on, whenever they encounter that alternative solution in another student's model they should also give them full points to be fair.

In our experiments we created all the alternative solutions in advance, but in a real-world setting they would also have to be discovered somehow. This *incremental discovery of alternative solutions* is not automated in our current implementation. Currently an instructor would create one solution, run the algorithm, and then look at the grades of the students that did not get high scores to check whether they might have used an alternative solution. If yes, the instructor would have to create another solution model and run the grading again. This can be repeated until the instructor is confident that all valid solutions have been considered.

To avoid this manual incremental discovery of alternative solutions, one could imagine a strategy where the tool applies standard refactorings and patterns to the initial solution to automatically create other solution variants, but this is left for future work.

8.2.2 RQ5: Does automated grading help to ensure fairness?

Fairness and consistency are in the mind of instructors when they grade students [74–76]. Student comments about grading are most often fairness-related [77; 78]. However, ensuring fairness is challenging for instructors when there are several correct solutions and/or when grading large number of students.

Table 8.1: Grades Difference for Animal Class Diagram

#	I	T	Reason for Difference
1	17	15.5	Missing Inheritance *Animal* (I: 1.5, T: 0.5), Missing Realization*Edible* (I: 1, T: 0.5)
2	17.5	17	Missing Realization *Edible* (I: 0.5 , T: 0)
3	17.5	18	Wrong Inheritance *Animal* (I: 1.5 , T: 2.0)
4	17	16.5	Wrong Inheritance *Bird* (I: 1 , T: 0.5)
5	14	12.5	Wrong Inheritance *Animal*: (I: 1.5, T: 0.5), *Bird*:(I: 1, T: 0.5)
6	17	16	Wrong Inheritance *Bird* (I: 1 , T: 0.5), *Edible* (I: 1 , T: 0.5)
7	17	16.5	Wrong Inheritance *Bird* (I: 1 , T: 0.5)
8	13	13.5	Wrong Realization *Edible* (I: 1 , T: 1.5)
9	17	16.5	Wrong Inheritance *Bird* (I: 1 , T: 0.5)
10	15	14.5	Wrong Inheritance *Bird* (I: 1 , T: 0.5)
11	17.5	17	Missing Realization *Edible* (I: 0.5 , T: 0)
12	17.5	17	Missing Realization *Edible* (I: 0.5, T: 0)
13	18	17	Wrong Inheritance *Bird* I: 1, T: 0.5, *Plant* (I: 1, T: 0.5)
14	18	17.5	Wrong Inheritance *Bird* (I: 1 , T: 0.5)

When we showed the automated grading scores (using adjusted configurations) to the instructor of *case study 1*, he re-examined some of the student models where the grade difference between the manual and automated grade

was high. He then discovered that he was not fair to 14 students.

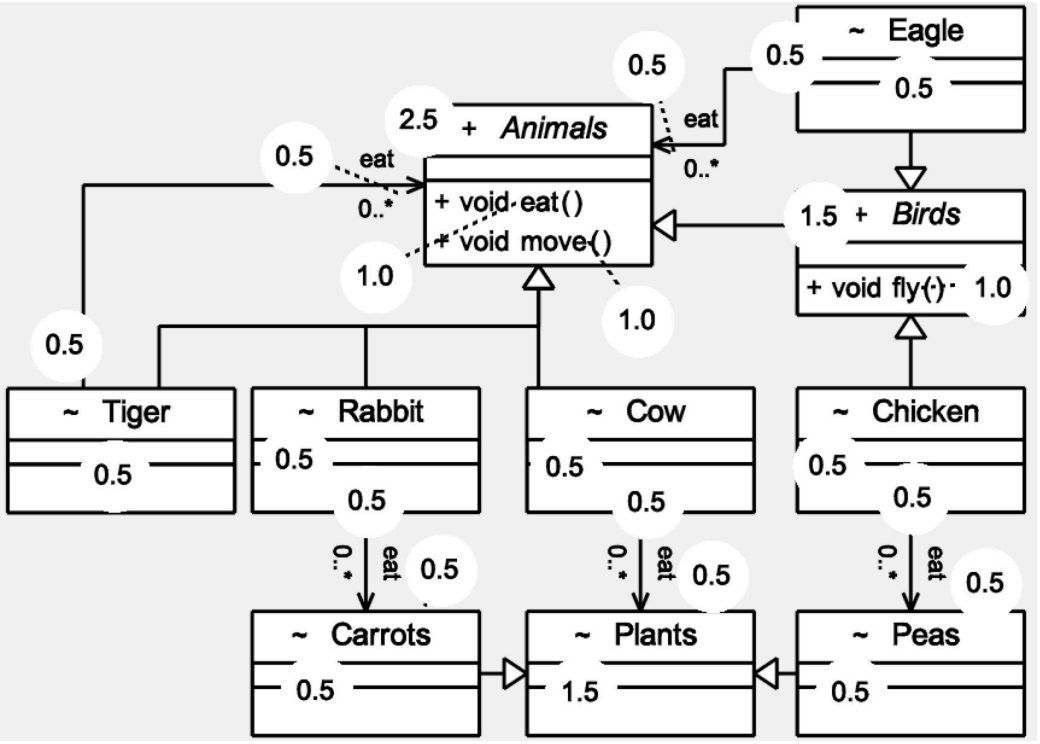

Figure 8.4: Student 12 Solution Model for Case Study 1

Table. 8.1 lists the details of 14 models on which the tool's grades were more reasonable than manual grading by instructor. For example, Student 11, shown in Fig. 8.4, failed to add the interface *Edible*. Therefore, class *Cow*, *Chicken* and *Plant* in the student's model are missing the realization from *Edible*. Based on the adjusted grading configuration for case study 1, the student should lose 0.5 mark for missing the interface *Edible* and lose 0.5 point for each missing class realization relationship (a total of 1.5 points). Therefore, the tool gave 0/2 for *Edible* and the realization relationships. However, the instructor only deducted 1 point for missing the

realization relationships. As a result, the instructor gave 0.5/2 points.

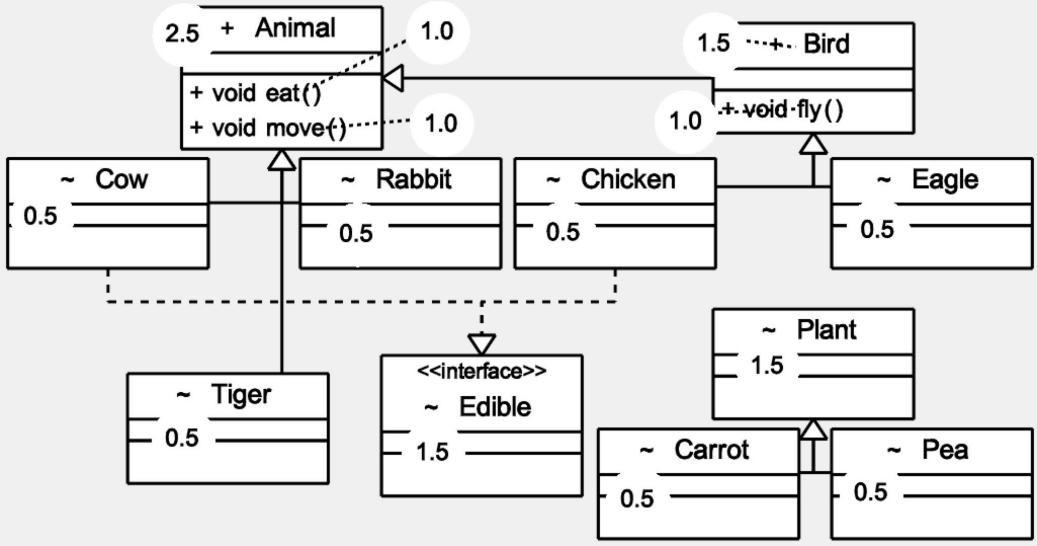

Figure 8.5: Student 8 Solution Model for Case Study 1

For student 8, shown in Fig. 8.5, again, the instructor deducted 1 point for the missing realization relationships, although the student is only missing 1 realization relationship, i.e. between *Plant* and *Edible*. The tool gave 1.5/2 and the instructor gave 1.0/2. For both examples, the instructor realized that the tool gave the correct points, and he revised his grades accordingly. After revising manual grading result, the average difference between manual and automated grading has decreased to 4.32%, while previous difference was 4.8%.

Table 8.2: Grades Differences for Hotel Domain Model

#	I	T	Reason for Difference
1	30	24	*Stay*: (I: 3, T: 0), *Reservation*: (I: 10, T: 6), *RoomType*: (I: 2, T: 3)
2	39	34	*RoomType*: (I: 3, T: 4), *Stay*: (I: 6, T: 2) *Duration*: (I: 1, T: 0)
3	31	31	*RoomType*: (I: 4, T: 5), *Reservation*: (I: 8, T: 7)
4	24	21	*Stay*: (I:2, T:0), *GuestInfo*: (I:1, T:0)
5	33	26	*Stay*: (I: 2, T: 0), *Reservation*: (I: 11, T: 8) *RoomType*: (I: 3, T: 1)
6	30	27	*Reservation*: (I: 8, T: 5)
7	40	31	*Stay*: (I: 6, T: 4), *Reservation*: (I: 14, T: 7)
8	41	37	*Reservation*: (I: 13, T: 9)
9	32	24	*RoomType*: (I: 3, T: 1), *Reservation*: (I: 10, T: 4)
10	32	28	*Stay*: (I: 0, T: 4), *Reservation*: (I: 13, T: 5)
11	29	25	*Reservation*: (I: 9, T: 5)
12	35	30	*Stay*: (I: 1, T: 0), *Reservation*: (I: 12, T: 8)
13	34	30	*Reservation*: (I: 10, T: 6)
14	31	28	*Reservation*: (I: 10, T: 7)
15	29	25	*Reservation*: (I: 10, T: 6)
16	35	25	*Reservation*: (I: 13, T: 5), *Stay*: (I: 2, T: 0)
17	37	30	*RoomType*: (I: 3, T: 1), *Stay*: (I: 3, T: 0) *FirstNight*: (I: 1, T: 0), *LastNight*: (I: 1, T: 0)
18	20	22	*Reservation*: (I: 3, T: 5)
19	28	23	*Reservation*: (I: 8, T: 3)
20	40	38	*Reservation*: (I: 14, T: 12)
21	37	32	*Reservation*: (I: 11, T: 6)
22	31	26	*Stay*: (I: 4, T: 0), *tele*: (I: 1, T: 0)
23	34	26	*Reservation*: (I: 11, T: 7), *Tel*: (I: 1, T: 0), *Stay*: (I: 5, T: 4)

When we showed the automated grading scores (using adjusted configurations with 3 solutions) to the instructor of *case study 2*, he discovered that he was not consistent in 23 student models, which are listed in Table. 8.2.

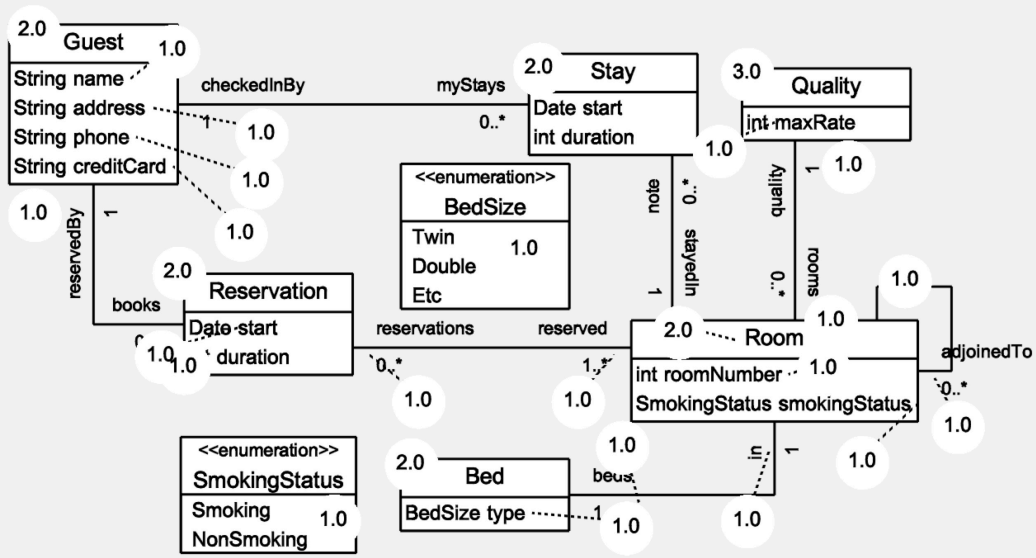

Figure 8.6: Student 2 Solution Model for Case Study 2

For students 1-3, the instructor gave less than what the student deserves for the class *RoomType*. For example, in the model of Student 2, shown in Fig. 8.6, the tool assigned 4 points for the class *Quality*, while the instructor assigned 3 points. The tool matched *Quality* with the class *RoomType* and the attribute *qualityLevel* in the instructor solution, shown in Fig. 8.1. This gave the student 3 points, 2 points for *RoomType* and 1 point for *qualityLevel*. The tool also matched the attribute *maxRate* in the student's model with the attribute *maxDailyRate* in the instructor model. Thus, the tool gave the student an extra 1 point. After discussing with the instructor, he admitted that he forgot to give one mark for this student.

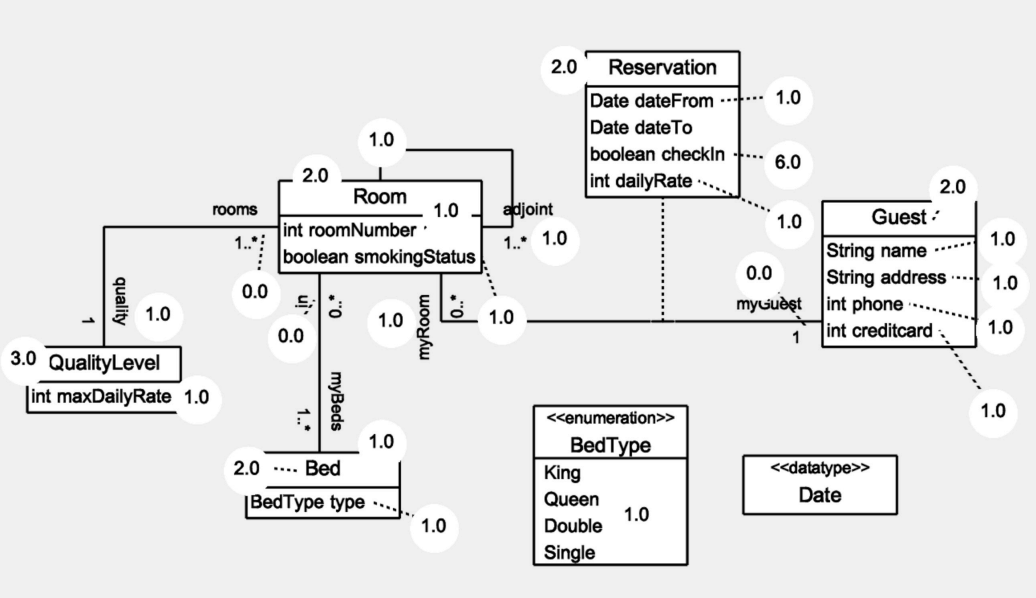

Figure 8.7: Student 21 Solution Model for Case Study 2

For students 4-23, the instructor gave too many points to them. For instance, Student 21, shown in Fig. 8.7, made *Reservation* to be an association class for the association *Room-Guest*. The instructor assigned 11 points for the class *Reservation*. He did not deduct points for making the class *Reservation* an association class rather than a regular class. When we discussed with the instructor, he admitted that making class *Reservation* an association class was incorrect and student 22 should not receive the 11 points, but rather should receive 7 points (the tool assigned 6 as shown in Table. 8.2). The 4 extra points that the instructor assigned were the following: 2 points for each of the associations between *Room-Reservation* and between *Reservation-Guest*. The tool did not give these extra points. Also, the tool

did not give points for the attribute *dateTo*, but the instructor decided that it matches with attribute *plannedNumberOfNights* and gave 1 points. Nevertheless, the instructor revised his grading after looking at the automated grading score. After revising manual grading results, the average difference between manual and automated grading was 8.27%, while previous difference was 9% (when using 3 solutions as discussed previously).

8.3 Conclusion

In this chapter, we discuss the assessment of the automated grading. First, we focus on the feature that our tool can deal with the questions which have multiple correct solutions. Our tool can load multiple correct solution models and compare the student model with all correct solution models to generate the final grading result. For situations in which alternative solutions are possible, running the automated grading algorithm several times and using the highest grade as the final grade brings the average automated grading scores closer to the manual grade. The average difference in case study 2 improved from 13% to 9%.

We also find that automated grading has the potential to save a significant amount of time over manual grading. It takes hours to manually grade the whole class of 80-100 students, while an automated tool once con-

figured with all acceptable solution variants grades the class in a few seconds. Also, automated grading can ensure fairness in the grading process. Manual grading is prone to unfairness. In our experiments, both instructors were not consistent in the grading process. Grades determined with a deterministic automated grading algorithm are more consistent. After the instructors adjusted their grades, the average difference between manual grading and automated grading improved from 4.8% to 4.32% for case study 1 and from 9% to 8.27% for case study 2.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

UML diagrams in general, and class diagrams in particular, are widely used in computer science and software engineering education. In many courses, computer science students are required to solve assignments or answer exam questions involving class diagrams. Instructors usually grade these diagrams manually by comparing each student solution with the template solution that they prepared for the assignment/exam. This could be a cumbersome task, especially when they have to grade large number of student papers. Furthermore, a particular problem could have different possible design solutions using class diagrams. Solutions could vary based on the names of the classes, their properties, or relationships between classes. Furthermore,

instructors are not uniform in their grading styles, and a modeling problem itself can have multiple correct solutions. It is therefore essential to devise effective automated grading approaches that can take this into account.

This thesis proposes an automated grading approach for class diagrams. In particular, two metamodels have been proposed in this thesis, one to establish mappings between an instructor's solution and student solutions, the other metamodel assigns grades to model elements and stores them. We also introduced a grading algorithm that matches model elements in the student model with elements in the instructor model. Moreover, we assess the efficacy of our automated approach for grading class diagrams in practice. In particular, we conduct two case studies in which we compare manual grading with automated grading. We develop a configuration panel which contains 27 configurable options, that allow the grading tool that we used to be tailored for a particular instructor's grading style. We find that configuring the algorithm to an instructor's style brings the automated grading scores closer to manual grading scores. In the case study on an introductory modeling assignment with 103 students, after adjusting the grading configuration to match with the instructor's style, the average difference between manual grading and our tool's grading is less than 5%. The second case study is with 89 students of an advanced modelling course on a domain model for a Hotel Reservation System. This study has more than one correct solution. Therefore, we add

a feature in the tool to allow instructors to upload multiple solution models. We found that 40/89 students used an alternative correct solution in their assignment. The average difference between manual grading and our tool's grading with multiple solutions is 9%.

We also find that automated grading has the potential to save time. While it takes hours to grade a class of about 100 students, automated grading accomplishes the task in a matter of seconds once the algorithm has been configured with the set of alternative solutions and to the desired grading style. Finally, compared with manual grading, automated grading has shown to be more consistent and able to ensure fairness in the grading process. In our case studies, we identify 37 cases in which the automated grading is more consistent than manual grading.

9.2 Future Work

Right now, when the instructor grades a modeling assignment with multiple correct solution models, we need the instructor to create the correct alternative models, then load them manually. As a result, it is necessary for the instructor to check the student models to find whether there are potential correct solution models, since instructor may not consider all the solution models at the first time. Therefore, we plan to allow the tool able to select

the alternative solution models automatically, which can improve the grading efficiency. Moreover, we plan to build a classroom platform, where the instructor can set the solution model and upload all the student models. After grading, it can show some statistics, such as the average for the whole class and the correctness rate of each element, e.g., for one class C, how many students in the whole classroom represent the class C in their models. Also, we plan to run more experiments with assignments obtained from different instructors. This can help us obtain more configuration options to make the automated grading tool suitable for a large number of instructors. Finally, we plan to extend our approach to grade other UML models, e.g., sequence diagrams and activity diagrams.

Bibliography

[1] J. Adams, *Computing Is The Safe STEM Career Choice Today*, November 3, 2014. [Online]. Available: https://cacm.acm.org/blogs/blog-cacm/180053-computing-is-the-safe-stem-career-choice-today/fulltext

[2] N. Singer, *The Hard Part of Computer Science? Getting Into Class*, January 24, 2019. [Online]. Available: https://www.nytimes.com/2019/01/24/technology/computer-science-courses-college.html

[3] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '10. New York, NY, USA: ACM, 2010, pp. 86–93. [Online]. Available: http://doi.acm.org/10.1145/1930464.1930480

[4] J. Caiza and J. Del Alamo, "Programming assignments automatic grad-

ing: Review of tools and implementations," in *INTED2013 Proceedings*, ser. 7th International Technology, Education and Development Conference. IATED, 4-5 March, 2013 2013, pp. 5691–5700.

[5] T. Daradoumis, R. Bassi, F. Xhafa, and S. Caballé, "A review on massive e-learning (mooc) design, delivery and assessment," in *2013 eighth international conference on P2P, parallel, grid, cloud and internet computing*. IEEE, 2013, pp. 208–213.

[6] H. Correia, J. P. Leal, and J. C. Paiva, "Enhancing feedback to students in automated diagram assessment," in *6th Symposium on Languages, Applications and Technologies*, 2017, p. 11.

[7] *khanacademy*, https://https://www.khanacademy.org/, 2020.

[8] *Udemy*, https://https://www.udemy.org/, 2020.

[9] *Coursera*, https://https://www.coursera.org/, 2020.

[10] R. F. Kizilcec, C. Piech, and E. Schneider, "Deconstructing disengagement: analyzing learner subpopulations in massive open online courses," in *Proceedings of the third international conference on learning analytics and knowledge*, 2013, pp. 170–179.

[11] M. Kassop, "Ten ways online education matches, or surpasses, face-to-face learning," *The Technology Source*, vol. 3, 2003.

[12] C. C. Leung, T. H. Lam, and K. K. Cheng, "Mass masking in the covid-19 epidemic: people need guidance," *Lancet*, vol. 395, no. 10228, p. 945, 2020.

[13] C. Wang, Z. Cheng, X.-G. Yue, and M. McAleer, "Risk management of covid-19 by universities in china," 2020.

[14] W. Van Lancker and Z. Parolin, "Covid-19, school closures, and child poverty: a social crisis in the making," *The Lancet Public Health*, vol. 5, no. 5, pp. e243–e244, 2020.

[15] *Nine Ways To Reimagine Higher Education*, https://www.forbes.com/sites/annkirschner/2020/05/14/nine-ways-to-reimagine-higher-education/#727f6c73767d, 2020.

[16] *Coronavirus: Alberta universities look to mostly online courses for fall semester*, https://globalnews.ca/news/6947098/coronavirus-alberta-universities-fall-semester/, 2020.

[17] *Fall classes will mostly be done online amid COVID-19, some Canadian universities say*, https://globalnews.ca/news/6935364/coronavirus-canadian-university-fall-classes/, 2020.

[18] *How Online Learning Kept Higher Ed Open During the Coronavirus Crisis*, https://spectrum.ieee.org/tech-talk/at-work/education/

how-online-learning-kept-higher-ed-open-during-the-coronavirus-crisis, 2020.

[19] R. Huang, D. Liu, A. Tlili, J. Yang, H. Wang *et al.*, "Handbook on facilitating flexible learning during educational disruption: The chinese experience in maintaining undisrupted learning in covid-19 outbreak," *Beijing: Smart Learning Institute of Beijing Normal University*, 2020.

[20] J. Sáenz, I. G. Gurtubay, Z. Izaola, and G. A. López, "pygiftgenerator: A python module designed to prepare moodle-based quizzes," *arXiv preprint arXiv:2005.00910*, 2020.

[21] N.-T. Le, F. Loll, and N. Pinkwart, "Operationalizing the continuum between well-defined and ill-defined problems for educational technology," *IEEE Trans. Learn. Technol.*, vol. 6, no. 3, pp. 258–270, Jul. 2013.

[22] P. Fournier-Viger, R. Nkambou, and E. M. Nguifo, *Building Intelligent Tutoring Systems for Ill-Defined Domains*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 81–101.

[23] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan, "Automated grading of dfa constructions," in *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[24] H. Simanjuntak, "Proposed framework for automatic grading system

of er diagram," in *2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE)*. IEEE, 2015, pp. 141–146.

[25] G. Hoggarth and M. Lockyer, "An automated student diagram assessment system," *SIGCSE Bull.*, vol. 30, no. 3, pp. 122–124, Aug. 1998.

[26] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher, "Feature modelling and traceability for concern-driven software development with touchcore," in *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, 2015, pp. 11–14.

[27] A. S. Lan, D. Vats, A. E. Waters, and R. G. Baraniuk, "Mathematical language processing: Automatic grading and feedback for open response mathematical questions," in *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, 2015, pp. 167–176.

[28] J. Kadupitiya, S. Ranathunga, and G. Dias, "Automated assessment of multi-step answers for mathematical word problems," in *2016 Sixteenth International Conference on Advances in ICT for Emerging Regions (ICTer)*. IEEE, 2016, pp. 66–71.

[29] M. Mohler and R. Mihalcea, "Text-to-text semantic similarity for auto-

matic short answer grading," in *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, 2009, pp. 567–575.

[30] S. Jing, O. Santos, J. Boticario, C. Romero, M. Pechenizkiy, and A. Merceron, "Automatic grading of short answers for mooc via semi-supervised document clustering." in *EDM*, 2015, pp. 554–555.

[31] S. Roy, S. Dandapat, A. Nagesh, and Y. Narahari, "Wisdom of students: A consistent automatic short answer grading technique," in *Proceedings of the 13th International Conference on Natural Language Processing*, 2016, pp. 178–187.

[32] S. P. Balfour, "Assessing writing in moocs: Automated essay scoring and calibrated peer review™." *Research & Practice in Assessment*, vol. 8, pp. 40–48, 2013.

[33] Y. Attali and J. Burstein, "Automated essay scoring with e-rater® v. 2," *The Journal of Technology, Learning and Assessment*, vol. 4, no. 3, 2006.

[34] M. T. Helmick, "Interface-based programming assignments and automatic grading of java programs," in *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, 2007, pp. 63–67.

[35] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, pp. 5–es, 2005.

[36] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, "Experiences with marmoset: designing and using an advanced submission and testing system for programming courses," *ACM Sigcse Bulletin*, vol. 38, no. 3, pp. 13–17, 2006.

[37] M. Amelung, P. Forbrig, and D. Rösner, "Towards generic and flexible web services for e-assessment," in *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, 2008, pp. 219–224.

[38] J. Gao, B. Pang, and S. S. Lumetta, "Automated feedback framework for introductory programming courses," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, 2016, pp. 53–58.

[39] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon, "On automated grading of programming assignments in an academic institution," *Computers & Education*, vol. 41, no. 2, pp. 121–131, 2003.

[40] P. Thomas, K. Waugh, and N. Smith, "Using patterns in the automatic

marking of er-diagrams," in *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, 2006, pp. 83–87.

[41] F. Batmaz and C. J. Hinde, "A diagram drawing tool for semi–automatic assessment of conceptual database diagrams," Jan 2006. [Online]. Available: https://hdl.handle.net/2134/4536

[42] A. Jayal and M. Shepperd, "The problem of labels in e-assessment of diagrams," *J. Educ. Resour. Comput.*, vol. 8, no. 4, pp. 12:1–12:13, Jan. 2009.

[43] M. Striewe and M. Goedicke, "Automated assessment of uml activity diagrams," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, 2014, pp. 336–336.

[44] S. TSELONIS and W. MCGEE, "Diagram matching for human-computer collaborative assessment," in *IN: Proceedings of the 9th CAA Conference, Loughborough: Loughborough University.* c Loughborough University Please cite the published version., 2005.

[45] P. Thomas, K. Waugh, and N. Smith, "Learning and automatically assessing graph-based diagrams," in *Beyond Control: learning technology for the social network generation. Research Proceedings of the 14th As-*

sociation for Learning Technology Conference (ALT-C, 4—6 September, Nottingham, UK, 2007), 2007, pp. 61–74.

[46] V. Vachharajani and J. Pareek, "Framework to approximate label matching for automatic assessment of use-case diagram," *International Journal of Distance Education Technologies (IJDET)*, vol. 17, no. 3, pp. 75–95, 2019.

[47] R. Sousa and J. P. Leal, "A structural approach to assess graph-based exercises," in *International Symposium on Languages, Applications and Technologies*. Springer, 2015, pp. 182–193.

[48] N. Haji Ali, Z. Shukur, and S. Idris, "Assessment system for uml class diagram using notations extraction," *International Journal of Computer Science and Network Security*, vol. 7, no. 8, pp. 181–187, 2007.

[49] J. Soler, I. Boada, F. Prados, J. Poch, and R. Fabregat, "A web-based e-learning tool for uml class diagrams," in *IEEE EDUCON 2010 Conference*, April 2010, pp. 973–979.

[50] R. W. Hasker, "Umlgrader: An automated class diagram grader," *J. Comput. Sci. Coll.*, vol. 27, no. 1, pp. 47–54, Oct. 2011.

[51] D. R. Stikkolorum, P. van der Putten, C. Sperandio, and M. Chaudron, "Towards automated grading of UML class diagrams with machine

learning," in *Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019*, ser. CEUR Workshop Proceedings, K. Beuls, B. Bogaerts, G. Bontempi, P. Geurts, N. Harley, B. Lebichot, T. Lenaerts, G. Louppe, and P. V. Eecke, Eds., vol. 2491. CEUR-WS.org, 2019. [Online]. Available: http://ceur-ws.org/Vol-2491/paper80.pdf

[52] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[53] V. Rus, M. Lintean, R. Banjade, N. B. Niraula, and D. Stefanescu, "Semilar: The semantic similarity toolkit," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2013, pp. 163–168.

[54] G. Hirst, D. St-Onge *et al.*, "Lexical chains as representations of context for the detection and correction of malapropisms," *WordNet: An electronic lexical database*, vol. 305, pp. 305–332, 1998.

[55] Z. Wu and M. Palmer, "Verbs semantics and lexical selection," in *Proceedings of the 32nd annual meeting on Association for Computational*

Linguistics. Association for Computational Linguistics, 1994, pp. 133–138.

[56] D. Lin, "An information-theoretic definition of similarity," in *Proceedings of the Fifteenth International Conference on Machine Learning*, ser. ICML '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 296–304. [Online]. Available: http://dl.acm.org/citation.cfm?id=645527.657297

[57] P. Resnik, "Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language," *J. Artif. Int. Res.*, vol. 11, no. 1, pp. 95–130, Jul. 1999.

[58] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[59] U. Laufs, C. Ruff, and J. Zibuschka, "Mt4j-a cross-platform multi-touch development framework," *arXiv preprint arXiv:1012.0467*, 2010.

[60] B. V. DeBoer, D. M. Anderson, and A. M. Elfessi, "Grading styles and instructor attitudes," *College Teaching*, vol. 55, no. 2, pp. 57–64, 2007.

[61] N. Resh, "Justice in grades allocation: Teachers' perspective," *Social Psychology of Education*, vol. 12, no. 3, pp. 315–325, 2009.

[62] R. Doran, F. Lawrenz, and S. Helgeson, "Research on assessment in science," *Handbook of research on science teaching and learning*, pp. 388–442, 1994.

[63] L. Biberman-Shalev, C. Sabbagh, N. Resh, and B. Kramarski, "Grading styles and disciplinary expertise: The mediating role of the teacher's perception of the subject matter," *Teaching and Teacher Education*, vol. 27, no. 5, pp. 831–840, 2011.

[64] W. Carbonaro, "Tracking, students' effort, and academic achievement," *Sociology of Education*, vol. 78, no. 1, pp. 27–49, 2005.

[65] G. S. Leventhal, *Fairness in social relationships*. General Learning Press Morristown, NJ, 1976.

[66] J. Randall and G. Engelhard, "Examining the grading practices of teachers," *Teaching and Teacher Education*, vol. 26, no. 7, pp. 1372–1380, 2010.

[67] S. Fitzgerald, B. Hanks, R. Lister, R. McCauley, and L. Murphy, "What are we thinking when we grade programs?" in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: Association

for Computing Machinery, 2013, p. 471–476. [Online]. Available: https://doi.org/10.1145/2445196.2445339

[68] M. De Marsico, F. Sciarrone, A. Sterbini, and M. Temperini, "Supporting mediated peer-evaluation to grade answers to open-ended questions," *EURASIA J. Math. Sci. Technol. Educ*, vol. 13, no. 4, pp. 1085–1106, 2017.

[69] J. Evermann and Y. Wand, "Ontological modeling rules for uml: An empirical assessment," *Journal of Computer Information Systems*, vol. 46, no. 5, pp. 14–29, 2006.

[70] V. L. Pavlov and A. Yatsenko, "Using pantomime in teaching ooa&ood with uml," in *18th Conference on Software Engineering Education & Training (CSEET'05)*. IEEE, 2005, pp. 77–84.

[71] D. Jackson and M. Usher, "Grading student programs using assyst," in *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, 1997, pp. 335–339.

[72] C. Wilcox, "The role of automation in undergraduate computer science education," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 2015, pp. 90–95.

[73] J. Carter, K. Ala-Mutka, U. Fuller, M. Dick, J. English, W. Fone, and

J. Sheard, "How shall we assess this?" in *Working group reports from ITiCSE on Innovation and technology in computer science education*, 2003, pp. 107–123.

[74] J. W. Howatt, "On criteria for grading student programs," *ACM SIGCSE Bulletin*, vol. 26, no. 3, pp. 3–7, 1994.

[75] R. D. Tierney, "Fairness in classroom assessment," *Sage*, 2012.

[76] R. D. Tierney, M. Simon, and J. Charland, "Being fair: Teachers' interpretations of principles for standards-based grading," in *The Educational Forum*, vol. 75, no. 3. Taylor & Francis, 2011, pp. 210–227.

[77] K. Sambell, S. Brown, and L. McDowell, "" but is it fair?": An exploratory study of student perceptions of the consequential validity of assessment." *Studies in educational evaluation*, vol. 23, no. 4, pp. 349–71, 1997.

[78] P. L. Nesbit and S. Burton, "Student justice perceptions following assignment feedback," *Assessment & Evaluation in Higher Education*, vol. 31, no. 6, pp. 655–670, 2006.